
Reinforcement Learning for Machine Learning Engineering Agents

Sherry Yang*
Stanford University

Joy He-Yueya
Stanford University

Percy Liang
Stanford University

Abstract

Existing agents for solving tasks such as ML engineering rely on prompting powerful language models. As a result, these agents do not improve with more experience. In this paper, we show that agents backed by weaker models that improve via reinforcement learning (RL) can outperform agents backed by much larger, but static models. We identify two major challenges with RL in this setting. First, actions can take a variable amount of time (e.g., executing code for different solutions), which leads to asynchronous policy gradient updates that favor faster but suboptimal solutions. To tackle variable-duration actions, we propose *duration-aware gradient updates* in a distributed asynchronous RL framework to amplify high-cost but high-reward actions. Second, using only test split performance as a reward provides limited feedback. A program that’s nearly correct is treated the same as one that fails entirely. To address this, we propose *environment instrumentation* to offer *partial credit*, distinguishing almost-correct programs from those that fail early (e.g., during data loading). Environment instrumentation uses a separate static language model to insert print statement to an existing program to log the agent’s experimental progress, from which partial credit can be extracted as reward signals for learning. Our experimental results on MLEBench suggest that performing gradient updates on a much smaller model (Qwen2.5-3B) trained with RL outperforms prompting a much larger model (Claude-3.5-Sonnet) with agent scaffolds, by an average of 22% across 12 Kaggle tasks.

1 Introduction

Language model (LM) agents using external tools can perform complex tasks from writing software programs [1, 2] to conducting scientific research [3]. Perhaps eventually, LMs can make better versions of themselves through agents performing machine learning engineering (MLE) [4, 5]. Existing agents for MLE rely on simply prompting powerful LMs. While scaling up test-time compute [6, 7] with prompting alone can allow an agent to find better solutions, the agent’s behavior does not change drastically without gradient updates despite much more experiences being collected. As shown in Figure 1, running the best MLE prompting framework according to MLEBench [5] for days leads to only slightly better best solutions.

A natural approach to improving performance given past experience is to perform gradient updates using reinforcement learning (RL) [8]. However, agentic settings pose additional challenges to RL. First, action execution of an MLE agent can take a variable amount of time (e.g., training different ML models), which leads to asynchronous policy gradient updates that favor faster but suboptimal solutions. To overcome this challenge, we propose *duration-aware gradient updates* during distributed asynchronous RL training to balance gradient updates to faster and slower actions. With duration-aware updates, we observe that an agent stops favoring faster solutions, achieving better performance in the long run.

*Correspond to <sherry@stanford.edu>.

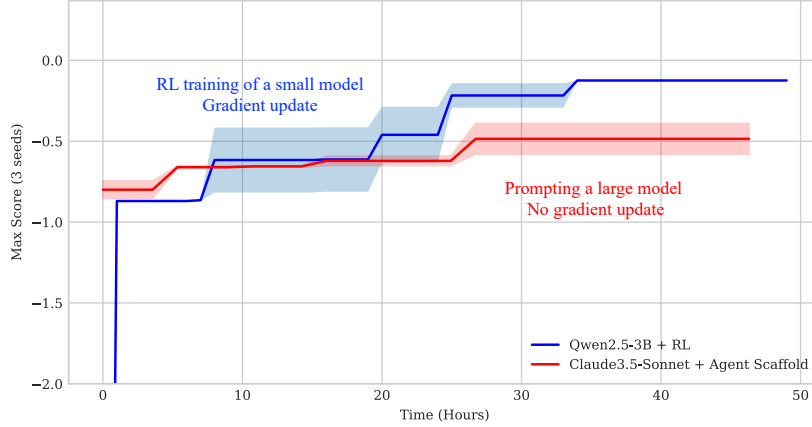


Figure 1: Performing gradient update with RL on Qwen2.5-3B (blue) is more effective in improving the best task performance than prompting Claude3.5-Sonnet with the best agent scaffold (red) on the “leaf-classification” task from MLEBench [5].

Second, while performance on the test split serves as a natural choice of rewards, they offer limited feedback, treating a nearly correct program (e.g., which failed during the last step of writing the solution to the correct location) the same as one that fails entirely (e.g., during data loading). Such limited feedback may lead to the agent stuck in suboptimal solutions (e.g., using the easiest way to produce test labels).

To address limited feedback, we propose *environment instrumentation* to offer *partial credit* to intermediate steps of completing an ML task (e.g., loading the data, building and training a model). We implement environment instrumentation by using a static copy of the original LM to insert print statements in the code generated by the agent, the execution of which provides partial credit. We observe that partial credit can gradually guide the agent away from making trivial mistakes (e.g., import errors, failures to load data) and towards improving ML techniques (e.g., feature engineering and hyperparameter choices).

Across a set of 12 Kaggle tasks from MLEBench [5], we show that RL training of a much smaller open-weight model (Qwen2.5-3B [9]) can eventually outperform prompting much larger close-weight models (e.g., Claude-3.5-Sonnet and GPT-4o) by an average of 22% and 24%, respectively. Our results suggest that future LM agent should learn to balance the compute resources spent across inference, interactions (action execution), and performing gradient updates, for tasks whose action execution incur non-trivial overhead, as in ML engineering.

2 Background

In this section, we provide relevant notations and define key learning objectives. We further discuss a few challenges of running standard RL algorithms in agentic settings.

MLE Agent in a Markov Decision Process (MDP). We consider a Markov Decision Process (MDP) [10] represented by a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, \mu \rangle$, consisting of a state space \mathcal{S} , an action space \mathcal{A} , a reward function $\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$, a state transition probability function $\mathcal{P} : \mathcal{S} \times \mathcal{A} \rightarrow \Delta(\mathcal{S})$, and an initial state distribution $\mu \in \Delta(\mathcal{S})$. A policy $\pi : \mathcal{S} \rightarrow \Delta(\mathcal{A})$ interacts with the environment, starting from an initial state $s_0 \sim \mu$. At each interactive step $k \geq 0$, an action $a_k \sim \pi(s_k)$ is sampled from the policy and applied to the environment. The environment then transitions into the next state $s_{k+1} \sim \mathcal{P}(\cdot | s_k, a_k)$ while a scalar reward $\mathcal{R}(s_k, a_k)$ is produced. Reinforcement Learning (RL) aims to find a policy π that maximizes the expected future rewards:

$$J(\pi) = E_{\pi, \mu, \mathcal{P}} \left[\sum_{k=0}^K \mathcal{R}(s_k, a_k) \right], \quad (1)$$

where K is the total number of steps. Standard RL algorithms such as policy gradient [11, 12] can be applied to learn the policy update rule by estimating

$$\nabla J(\pi_\theta) = E_{\pi, \mu, \mathcal{P}} \left[\sum_{k=0}^K \nabla_\theta \log \pi_\theta(a_k | s_k) \hat{A}(s_k, a_k) \right], \quad (2)$$

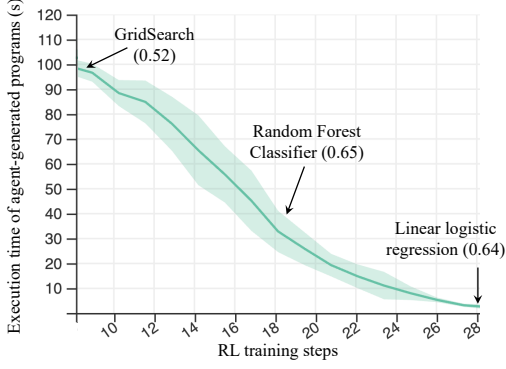


Figure 2: **Execution time of agent-generated programs** (averaged across 128 samples) decreases drastically as RL training progresses. Plot produced by running distributed RL [17] on the *random-acts-of-pizza* (text classification) Kaggle task from MLEBench [5]. The final solution converges to a fast but suboptimal solution (achieving score 0.64) of using linear logistic regression that takes less than 1 second to execute, as opposed to other solutions with better performance but takes longer to run.

where $\hat{A}(s_k, a_k)$ is some advantage function which can be separately estimated (e.g., by Monte-Carlo returns from π [11]).

In the MLE agent setting, \mathcal{S} captures input to the agent, including problem description, datasets, and any experiment history. \mathcal{A} captures solutions generated by an agent, including high-level plans (e.g., which family of models to use) and low-level code. \mathcal{P} captures any potential output from the environment during action execution (e.g., error messages). We focus on the $K = 1$ setting where the agent tries to solve the problem in one generation. \mathcal{R} captures rewards from the environment, such as performance on the test split or any additional scalar-value feedback the environment is able to offer to the agent.

Challenges of RL for MLE Agents. For sample efficiency [13, 14], many RL training frameworks implement an asynchronous distributed setup where multiple “actors” can interact with their own instances of the environment simultaneously, gathering experiences which are then sent to a “learner” for policy gradient updates [15, 16]. In agentic settings such as ML engineering, each action may take a variable amount of time to execute. As a result, running distributed RL training favors faster actions (slower actions might often time out). Moreover, time-consuming actions are sampled less frequently in a distributed training framework, leading to an uneven number of gradient updates for faster and slower actions. As shown in Figure 2, naively running a distributed RL framework [17] on Kaggle challenges from MLEBench [5] leads to the agent only generating quick solutions that barely take any time to execute.

Another challenge of RL for MLE agents is the limited feedback for intermediate progress. While performance on the test split is a natural reward, it does not distinguish between a solution failing to load data and one that is nearly correct. As correctly loading training data can be difficult for MLE agents, relying solely on test performance as a reward may lead to RL converging on a suboptimal solution: not using the training data at all. For instance, in the *tweet-sentiment-extraction* task where the agent needs to extract sentiment-supporting phrases from tweets, the agent converged to a suboptimal approach of directly code the Jaccard similarity evaluation function and search the test input for the best phrase (as shown in Figure 3), bypassing ML completely.

3 RL for MLE Agent

In this section, we propose duration-aware gradient updates (Section 3.1) and environment instrumentation (Section 3.2) to overcome the aforementioned challenges of applying RL to MLE agents in Section 2. The agent can further improve a previously generated solution, which can be further enforced using RL (Section 3.3). See Algorithm 1 in Appendix A.1 for the training loop of the RL agent.

```
1 import pandas as pd
2
3 test_df = pd.read_csv(
4     "/input/test.csv")
5
6 def jaccard(str1, str2):
7     a = set(str1.lower().split())
8     b = set(str2.lower().split())
9     c = a.intersection(b)
10    return float(len(c)) / (len(a) + len(b)
11                           - len(c))
12
13 for i, row in test_df.iterrows():
14     phrase = max(row['text'].split(), key=
15                 lambda x: jaccard(x, row['text']))
```

Figure 3: **Suboptimal convergence** due to limited feedback. In a task of extracting sentiment-relevant phrases from tweets (*tweet-sentiment-extraction*), the agent converged to a suboptimal solution of directly coding the Jaccard similarity and search for the best phrase in the test input, bypassing ML completely. This demonstrates how sparse rewards can lead to an agent exploiting evaluation metrics rather than learning desired behaviors.

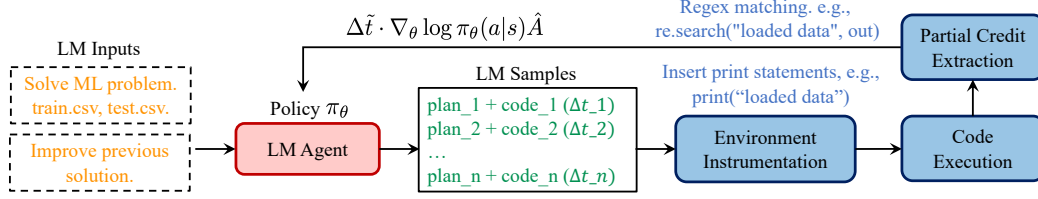


Figure 4: **Proposed framework overview.** Duration-aware gradient updates re-weights the policy gradient updates according to the execution duration of an action. Environment instrumentation inserts print statements using a static LM, the execution output can then be extracted for partial credit. The agent can be further asked to improve a previous solution, the response can further be enforced with RL.

3.1 Duration-Aware Gradient Updates for Variable-Time Action Execution

As discussed in Section 2, existing distributed RL frameworks [16, 17] often do not take into account the duration of action execution. In agentic settings such as ML engineering, each action involves running code that trains a model. As a result, different action samples have drastically different execution time (e.g., training a model for 1 versus 10 epochs). Applying existing distributed RL training results in the policy quickly converging to faster but suboptimal actions, as shown in Figure 2. A naïve solution to this problem is to wait for all actions to finish executing before performing any policy gradient updates, but this does not utilize resources well and is not scalable as training a model for a hard ML problem can take days.

The Issue with Variable-Duration Action Execution. We first provide a precise illustration of the issue with variable-time action execution in distributed RL training. Define n_x and n_y as the number of samples of actions x and y collected in time T . Denote \hat{A}_x and \hat{A}_y as the average advantage estimates for actions x and y . In a fixed training period of length T , we would collect approximately:

$$n_x \approx \frac{\pi(x|s) \cdot T}{\Delta t_x} \text{ samples of action } x, \quad n_y \approx \frac{\pi(y|s) \cdot T}{\Delta t_y} \text{ samples of action } y$$

Where $\pi(x|s)$ and $\pi(y|s)$ are the probabilities of selecting actions x and y under the current policy. The total gradient contribution for each action would be:

$$G_x = \frac{\pi(x|s) \cdot T}{\Delta t_x} \cdot \nabla_{\theta} \log \pi_{\theta}(x|s) \cdot \hat{A}_x, \quad G_y = \frac{\pi(y|s) \cdot T}{\Delta t_y} \cdot \nabla_{\theta} \log \pi_{\theta}(y|s) \cdot \hat{A}_y$$

Note that G_x and G_y are divided by Δt_x and Δt_y , meaning faster actions (smaller Δt) contribute proportionally more to the gradient.

Duration-Aware Gradient Updates. To counter the frequency bias above, we propose to weight each gradient update by the action duration, which gives

$$G'_x = \pi(x|s) \cdot T \cdot \nabla_{\theta} \log \pi_{\theta}(x|s) \cdot \hat{A}_x, \quad G'_y = \pi(y|s) \cdot T \cdot \nabla_{\theta} \log \pi_{\theta}(y|s) \cdot \hat{A}_y$$

With duration weighting, the Δt terms cancel out, leaving each action’s contribution to the gradient proportional only to its policy probability and advantage, not to its execution frequency. This ensures that actions with longer durations receive fair consideration in policy updates despite generating fewer samples in the same time period. Generalizing from this toy example to the continuous case with arbitrary action durations, we arrive at our duration-aware policy gradient update rule:

$$\nabla_{\theta} J(\pi_{\theta}) = E_{\pi, \mu, \mathcal{P}} \left[\sum_{k=0}^K \Delta t_k \cdot \nabla_{\theta} \log \pi_{\theta}(a_k|s_k) \cdot \hat{A}(s_k, a_k) \right] \quad (3)$$

Where Δt_k is the execution duration of action a_k taken at state s_k . This formulation ensures that in expectation, the contribution of each action to the policy gradient is proportional to its true value, regardless of how frequently it is sampled due to varying execution times. In practice, we rescale Δt_k by the average execution time in the batch to avoid overly large gradient updates.

3.2 Environment Instrumentation for Partial Credit

Another challenge of RL for MLE agents lies in sparse reward. The natural choice of reward for MLE tasks is the performance of the developed model on the test split, which would only be non-zero if an agent generates code that successfully completes every step from data loading to model training and inference. However, for tasks that involve images such as plant-pathology-2020-fgvc7, even loading the data correctly is non-trivial for the MLE agent backed by a small model. As a result, generated programs that fail to load data should be rewarded differently from programs that are almost correct.

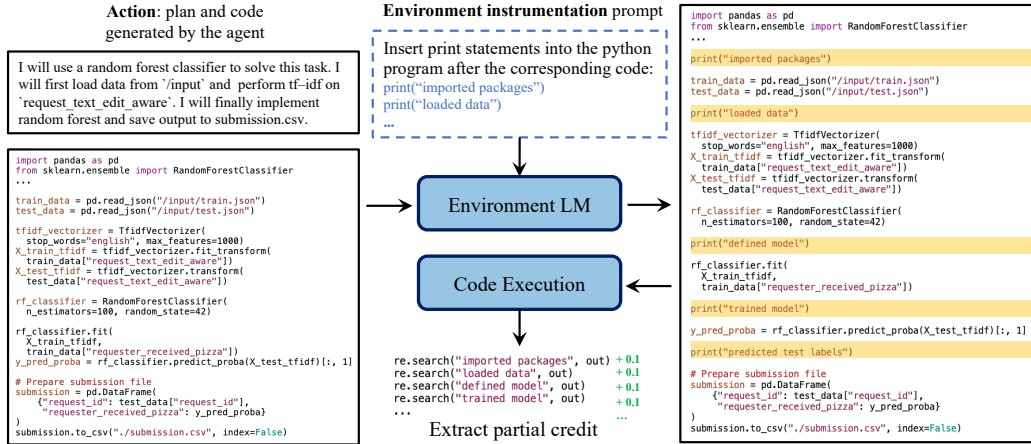


Figure 5: **Environment instrumentation overview.** Another copy of the small LM (Qwen2.5-3B) is prompted to insert print statement into the code generated by the agent. After code execution, output from the terminal is then parsed to assign partial credit by regex matching.

Environment instrumentation. We propose to introduce *partial credit* so that generated programs that fail in the beginning (e.g., during data loading) will receive less partial credit than programs that is almost correct (e.g., failed at saving output to the correct location). To avoid making too much assumption about how the agent should solve a problem, we assign partial credit only based on whether a solution completed high-level procedures including importing libraries, loading data, building ML model, training the model, and running the model on the test split. We propose to use another static copy of the original LM to instrument the code generated by the agent by inserting print statements into the program generated by the agent to track execution progress. The terminal output will be parsed through regex matching to provide partial credit based on whether expected print statements (e.g., “print(loaded data)”) are executed. To assign partial credit, we use -10 to denote programs that fails completely, and add 0.1 to each print statement found through regex, as shown in Figure 5. If a generated programs runs without error, the submission (label for the test split) is graded by the grader of the environment, and true task performance is used as reward (generally between -1 and 1). It is important to ensure that such reward instrumentation is performed by a separate copy of the LM as opposed to the agent being optimized with RL, as otherwise the agent is incentivized to generate such print statements to gather partial credits without actually performing the steps to achieve high reward.

3.3 Multi-Step RL with Self-Improvement Prompt

So far, we have discussed the setting where an agent is directly asked to generate plans and code solutions for solving MLE tasks. Next, we further explore whether we can directly instruct the agent to improve a previously generated solution. Specifically, we sample from two sets of prompts (with equal probability) to solve the problem from scratch and improve a solution generated by the previous step. We illustrate the two types of prompts in Figure 4. In the case of improving a previous solution, output of the terminal is given to the agent which includes information such as training and test accuracy (from environment instrumentation introduced in Section 3.2). We have also experimented with giving failed executions to the agent to self debug, but have noticed limited self-debugging abilities in small models. At test time, we both generate solutions from scratch and run the agent again to improve the generated solutions, and take the maximum between the two solutions (with and without explicit improvement).

4 Experiments

In this section, we evaluate our proposed improvements to the RL training of the MLE agent. We first discuss the evaluation setup and implementation details in Section 4.1. We then present the main evaluation results in Section 4.2, followed by ablation studies in Section 4.3.

4.1 Evaluation Setup and Implementation Details

Evaluation Setup. We follow the setup of MLEBench [5], which consists of 75 tasks of Kaggle challenges that range from various types such as classification and regression on image, text, and

Table 1: **Comparing RL of a small model to prompting large models** across 12 tasks from MLEBench. RL results are best scores among 128 samples after RL training has converged. Baseline results are from runs in [5], produced by prompting frontier models using AIDE agent scaffolds and continuing running for 24 or 100 hours. Numbers shown are mean and standard error across 3 runs. All except for the last column use the AIDE agent scaffold. \uparrow denotes the higher the score the better. N/A denotes no valid submissions were available. RL of a small model achieves the best final performance on 8 out of 12 tasks.

Tasks	Qwen2.5-3B	Llama3.1-405B	Claude3.5-Sonn	GPT-4o-100hrs	Qwen2.5-3B RL
detecting-insults-in-social-commentary (\uparrow)	0.870 +/- 0.009	N/A	N/A	N/A	0.895 +/- 0.001
learning-agency-lab-automated-essay-scoring-2 (\uparrow)	0.331 +/- 0.018	0.777 +/- 0.002	0.794 +/- 0.008	0.759 +/- 0.002	0.746 +/- 0.002
random-acts-of-pizza (\uparrow)	0.589 +/- 0.004	0.619 +/- 0.007	0.627 +/- 0.004	0.638 +/- 0.005	0.663 +/- 0.011
tweet-sentiment-extraction (\uparrow)	0.027 +/- 0.018	N/A	0.448 +/- 0.251	0.283 +/- 0.005	0.596 +/- 0.002
tabular-playground-series-may-2022 (\uparrow)	0.787 +/- 0.020	0.939 +/- 0.002	0.743 +/- 0.126	0.883 +/- 0.002	0.913 +/- 0.000
tabular-playground-series-dec-2021 (\uparrow)	0.827 +/- 0.044	0.771 +/- 0.188	0.645 +/- 0.315	0.957 +/- 0.000	0.951 +/- 0.000
us-patent-phrase-to-phrase-matching (\uparrow)	0.065 +/- 0.000	N/A	0.805 +/- 0.006	0.588 +/- 0.015	0.527 +/- 0.003
plant-pathology-2020-fgvc7 (\uparrow)	0.628 +/- 0.058	0.968 +/- 0.005	0.990 +/- 0.002	0.970 +/- 0.001	0.970 +/- 0.004
leaf-classification (\downarrow)	0.884 +/- 0.016	6.747 +/- 5.398	0.436 +/- 0.102	0.846 +/- 0.029	0.124 +/- 0.000
nomad2018-predict-transparent-conductors (\downarrow)	0.178 +/- 0.045	0.166 +/- 0.103	0.083 +/- 0.020	0.072 +/- 0.003	0.059 +/- 0.000
spooky-author-identification (\downarrow)	0.596 +/- 0.053	0.487 +/- 0.020	0.701 +/- 0.186	0.546 +/- 0.004	0.404 +/- 0.011
lmsys-chatbot-arena (\downarrow)	11.48 +/- 0.002	1.269 +/- 0.051	2.211 +/- 0.959	1.451 +/- 0.035	1.081 +/- 0.002

tabular data. We select a subset of 12 tasks for evaluation where the original Qwen2.5-3B model [9] could generate a valid initial solution in a batch of 128 samples with temperature 0.7 (for RL to make any progress). We use the grader from [5] to grade the final 128 samples after convergence of RL training and measure both the mean and the maximum performance for each run. We use the scores achieved by different frontier LMs and different agent scaffolds from the original runs of MLEBench as baselines. In evaluating against different frontier models, we use the AIDE scaffold, which organizes experience in a tree structure and saves the best solution seen so far for evaluation. In evaluating against different agent scaffolds, we use the results from the GPT-4o based agent running 24 hours using two additional agent scaffolds, OpenHands [18] and MLAgentBench (MLAB) [4] (which has outperformed LangChain [19] and AutoGPT [20]). To further understand the improvement progress of RL and prompting, we re-run the set of MLEBench experiments using Claude-3.5-Sonnet and AIDE agent scaffolding, while grading the intermediate best saved solutions, and compare that to intermediate solutions during RL training, both across three runs.

Implementation Details. To implement RL training of the Qwen model, we build on top of the distributed RL training framework in [17]. We implement a set of distributed sandboxed code execution environments similar to [5], where code execution takes place inside of the RL training loop as a part of the reward function implementation. To implement environment instrumentation, we load a separate copy of the original Qwen2.5-3B model (without performing any gradient updates on it) and ask the model to insert print statements before executing the code. The prompt for environment instrumentation can be found in Appendix C.3. To assign partial credit, we use reward -10 to denote solutions that fail completely (e.g., no plans or code, fail to import packages), and add 0.1 per regex match in the terminal output. If the solution is valid (according to the grader), we use the actual score from the grader as reward. We further experimented with normalizing the reward to a particular range (0 to 1) but did not observe significant difference. For tasks where the lower scores are better, we flip the signs of the scores to use as rewards. We use the Proximal Policy Gradient (PPO) [12] algorithm (with modification of duration-aware gradient) to train the Qwen2.5-3B model for each task until reward convergence, which generally took 1-3 days depending on the task using 8 A100-40GiB GPUs. We limit the input and output length to 1024 tokens. The model is trained using a batch size of 128 and learning rate of $1e-5$ (See complete set of hyperparameters in Appendix B.1).

4.2 Evaluation Results on MLEBench

Comparing against Different Frontier Models. We report the mean and standard error across three runs of RL training or prompting a frontier model using the AIDE scaffold in Table 1. Since the AIDE scaffold saves the best solution found through prompting, we also report the maximum among the 128 samples (See the mean scores across the 128 samples during RL training in Figure 10 in Appendix D.1). Qwen2.5-3B with RL outperforms prompting a frontier LM on 8 out of the 12 tasks, and achieves an average of 22% improvement (measured by improvements or degradation over the baselines) over prompting Claude3.5-Sonnet. For the tasks where Qwen could not outperform the frontier models, we still observe significant improvement running RL over prompting the Qwen model with the AIDE agent scaffolding (As shown in Column 2 of Table 1). We note that running AIDE for longer hours (e.g., GPT-4o 100 hours as opposed to the default 24 hours for other columns)

Generated plan	Generated code
To improve the previous solution and achieve a lower log loss on the test set, we can improve Feature Engineering : We can create additional features from the prompt and responses, such as the length of the texts, the presence of certain keywords, or the number of words.	<pre># Feature engineering train_df['response_a_length'] = train_df['response_a'].apply(len) train_df['response_b_length'] = train_df['response_b'].apply(len) train_df['response_diff'] = (train_df['response_a_length'] - train_df['response_b_length'])</pre>
Additional Feature Engineering: We can use additional features such as the word count difference, and average word length difference between the two responses are created to capture more information about the responses.	<pre># Additional features train_df['response_word_count_diff'] = train_df['response_a'].apply(lambda x: len(x.split())) - train_df['response_b'].apply(lambda x: len(x.split())) train_df['response_avg_word_length_diff'] = train_df['response_a'].apply(lambda x: np.mean([len(word) for word in x.split()])) - train_df['response_b'].apply(lambda x: np.mean([len(word) for word in x.split()]))</pre>

Figure 6: **Qualitative examples** of improvements proposed by the agent during RL training. [Top] earlier improvement proposed by the agent using difference between response length as features for preference prediction. [Bottom] later improvements proposed by the agent using additional features such as word count and average word length difference as features.

Table 2: **Comparing RL to different agent scaffolds.** RL training of a small model outperforms prompting GPT-4o with different agent scaffolds on 9 out of the 12 tasks. Results for agent scaffolds are taken from [5]. Numbers show mean and standard error of the final performance according to the grader in [5].

Tasks	GPT-4o AIDE	GPT-4o OpenHands	GPT-4o MLAB	Qwen2.5-3B RL
detecting-insults-in-social-commentary (↑)	NaN	0.867 +/- 0.017	0.749 +/- 0.039	0.895 +/- 0.001
learning-agency-lab-automated-essay-scoring-2 (↑)	0.720 +/- 0.031	0.681 +/- 0.010	0.533 +/- 0.080	0.746 +/- 0.002
random-acts-of-pizza (↑)	0.645 +/- 0.009	0.591 +/- 0.048	0.520 +/- 0.013	0.663 +/- 0.011
tweet-sentiment-extraction(↑)	0.294 +/- 0.032	0.415 +/- 0.008	0.158 +/- 0.057	0.596 +/- 0.002
tabular-playground-series-may-2022 (↑)	0.884 +/- 0.012	0.882 +/- 0.030	0.711 +/- 0.050	0.913 +/- 0.000
tabular-playground-series-dec-2021 (↑)	0.957 +/- 0.002	0.957 +/- 0.000	0.828 +/- 0.118	0.951 +/- 0.000
us-patent-phrase-to-phrase-matching (↑)	0.756 +/- 0.019	0.366 +/- 0.039	NaN	0.527 +/- 0.003
plant-pathology-2020-fgvc7 (↑)	0.980 +/- 0.002	0.680 +/- 0.113	0.735 +/- 0.052	0.970 +/- 0.004
leaf-classification (↓)	0.656 +/- 0.070	0.902 +/- 0.018	4.383 +/- 2.270	0.124 +/- 0.000
nomad2018-predict-transparent-conductors (↓)	0.144 +/- 0.031	0.183 +/- 0.120	0.294 +/- 0.126	0.059 +/- 0.000
spooky-author-identification (↓)	0.576 +/- 0.071	0.582 +/- 0.020	0.992 +/- 0.463	0.404 +/- 0.011
lmsys-chatbot-arena (↓)	1.323 +/- 0.147	1.131 +/- 0.019	10.324 +/- 4.509	1.081 +/- 0.002

did not lead to significantly better performance, indicating that solutions other than prompting a large frontier LM is required to effectively achieve self-improvement.

In Figure 6, we provide example solutions that the Qwen agent came up with during the RL training process in solving the lmsys-chatbot-arena task. This task requires the agent to come up with ML code and train a model to predict which responses generated by LMs a user will prefer. The Qwen agent is able to come up with various different feature engineering choices such as using the difference in response length, word count, and average word length as additional features.

Comparing against Different Agent Scaffolds. We now compare running RL on Qwen2.5-3B against running different agent scaffolds on GPT-4o. Table 2 shows that Qwen2.5-3B with RL outperforms prompting LMs with various agent scaffolds on 9 out of the 12 tasks, achieving an average improvement of 17.7% over the best scaffold for each task. We notice that the performance of prompting a frontier model does vary for each task across different agent scaffolds. For instance, AIDE achieved no valid submissions across 3 runs (score NaN) for Llama3.1, Claude3.5-Sonnet, and GPT-4o on detecting-insults-in-social-commentary, while other scaffolds can achieve valid solutions on the same task. Nevertheless, AIDE generally works better in achieving higher task performance compared to other agent scaffolds, suggesting that RL is a reliable way to improve performance that is agnostic to different choices of scaffolds.

Performance Improvement over Time. In Figure 7, we compare the max performance (across 128 samples) aggregated over time of training Qwen2.5-3B with RL against running AIDE agent scaffold using Claude-3.5-Sonnet. We observe that for many tasks such as learning-agency-lab-automated-essay-scoring-2, tweet-sentiment-extraction and random-acts-of-pizza, prompting the large model initially achieves much better performance than the small model. However, as RL training goes on, performance of the smaller model improves more with gradient updates, eventually exceeding prompting a large model.

4.3 Ablation Studies

We now present ablation studies on duration-aware gradient updates, environment instrumentation, and explicit self-improvement prompt.

Effect of Duration-Aware Gradient. We plot the average execution time (across 128 samples) during RL training with and without duration-aware gradient in Figure 8. With duration-aware gradient, the agent is able to find better solutions that take longer to execute (e.g., gradient boosting),

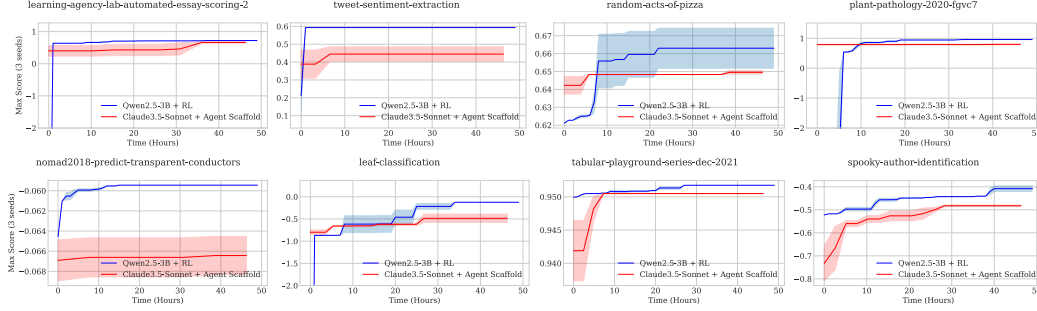


Figure 7: **The best scores achieved by the agent across time** comparing prompting a large model to RL training of a small model. A small model running RL starts off with low scores for many tasks, but eventually outperforms prompting a large model.

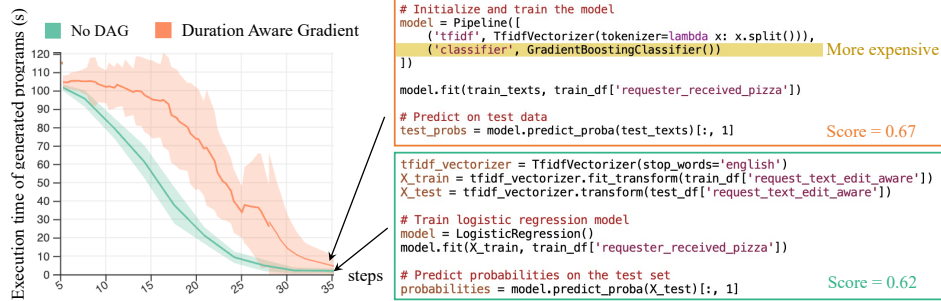


Figure 8: **Duration-aware gradient** enables the agent to explore more expensive but high-return actions by coming up with more expensive solutions such as gradient boosting, which achieves higher score than linear logistic regression. The RL agent still tends to find faster executing solutions over time.

whereas without duration-aware gradient, the agent quickly converges to fast but suboptimal solution (e.g., linear logistic regression). Nevertheless, we found that the RL agent still tends to find faster executing solutions, as the average execution time still tend to decrease over time.

Effect of Environment Instrumentation In Figure 9, we show the average task scores (across 128 samples) during RL training with and without environment instrumentation. The task scores are -10 if the solutions are invalid and the actual scores otherwise (partial credits from environment instrumentation are omitted from the plot but is included in the actual reward RL optimizes). We observe that environment instrumentation leads to faster growing and faster converging average scores. The high-variance in plant-pathology-2020-fgvc7 (right most subplot) was due to one RL training run not being able to produce any valid solution due to sparse reward, which we observe more frequently when environment instrumentation is absent.

Effect of Explicit Self-Improvement Prompt. Next, we compare explicitly asking the agent to improve a previous solution (50% of the time during RL training) to only having the agent solve the task from scratch. Asking the model to improve a previous solution leads to better final performance

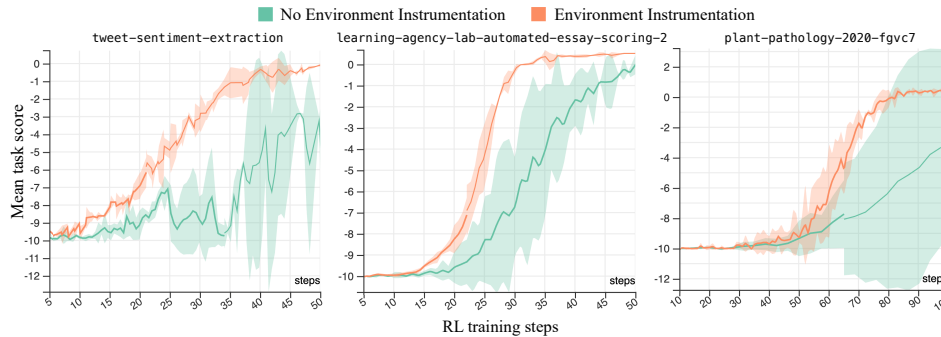


Figure 9: **Environment instrumentation ablation.** Plots show the mean task scores (excluding the partial credit from environment instrumentation) across 128 samples for 3 example tasks across RL training steps. Environment instrumentation improves RL training and enables faster convergence.

for 10 out of 12 tasks and achieves an average improvement of 8% over coming up with solution from scratch, suggesting that RL can simultaneously improve both initial solution generation and improve the ability to improve a previously solution. See the complete performance with and without self-improvement prompt in Table 4 of Appendix D.2.

5 Related Work

ML engineering agents. Many recent work has emerged for building LM agents that can solve machine learning benchmarks [4, 21, 5, 22, 23], data science tasks [24, 25, 26], or help with other aspects of ML such as data preprocessing and hyperparameter optimization [27, 28, 29]. Most existing work in this space has focused on prompting large frontier LMs as opposed to performing gradient updates. Existing work has used various agent scaffolds such as LangChain [19], AutoGPT [20], OpenHands [18], and AIDE [30] for in-context learning, and further try to improve agent performance by heuristic-based search during inference time [31]. While LM agents perform better with these scaffolds, they still face the challenge of achieving improvement reliably from prompting [32, 33]. We focus on RL training of smaller models instead of prompting large models.

RL for LMs. Since the development of both policy and value based RL algorithms [11, 34, 35, 36, 12, 37] extensively took places in simulated environment such as MuJoCo [38] and Atari [39], many RL optimization frameworks [16, 40] make the implicit assumption is that environment interactions take up a constant amount of time. More recently, RL has been used extensively in aligning LMs to human preferences [41, 42, 43, 44, 44], reasoning [45], and solving math [46] and coding [47] problems. However, this assumption persists, as rewards are often produced by a reward model [41] or verifiable answers to math or coding problems [48, 47]. As a result, the problem of variable-time action execution has not been extensively studied. However, this problem is highly relevant in practical agentic systems such as ML engineering. As RL being extended to a broader array of agentic applications, deriving optimization frameworks that take into account the time an action takes is essential. Meanwhile, directly applying existing RL training frameworks developed for simulation settings, math, and reasoning, such as [16, 17], results in poor agent performance.

RL for agentic systems and interactive tasks. Existing work has studied RL for agentic settings solving multi-step interactive tasks such as operating a unix terminal [49], booking flights [50], controlling devices [51], negotiating price [52], navigating through the web [53], and playing language-based games [54, 55]. However, most of these settings still neglect the time it takes to execute actions, and mostly leverage gamma discounting [8] to balance the influence of future rewards and immediate rewards. In these settings, the horizon is mostly determined by the number of interactive turns, as opposed to each turn taking a different amount of time, which is the focus of this work. Additionally, sparse reward has been challenging in many agentic settings. Existing work has leveraged LMs/VLMs as process reward [56, 57, 58] to provide dense reward signals for policy evaluation and RL training [59, 60, 51]. However, directly using LM as reward functions can be unreliable [61, 62]. We tackle sparse reward by having LM insert verifiable print statements as a form of reliable execution feedback to improve the agent through RL.

6 Conclusion

We have shown that performing gradient updates on a small language model through RL can be more effective than prompting a large frontier LM to solve ML engineering tasks with an LM agent. We have also shown that reweighting policy gradient updates based on action duration can overcome variable-duration interactions, while using an LM to perform reward instrumentation through code can mitigate sparse rewards. We discuss the limitation and impact of this work below.

Limitations and future work. Despite initial signs that RL can allow a small model to outperform prompted by a large model, scaling up RL on large models to solve a large set of problems is an interesting future area of research, which may require collaboration with industrial labs. Training a single agent to solve multiple tasks at once and investigating generalization to new tasks are another future research direction. While this work considers improving a previously generated solution as a multi-step interaction, alternatively one can formulate breaking down a complex ML problem and solving one component at a time as a multi-step process. Investigating RL for MLE agent in this multi-step setup is an interesting direction.

Social and broader impact. LM agent performing ML engineering might affect the job opportunities for actual software and ML engineers, which calls for additional research in policy. MLE agent having full access to the internet and running code might incur security risks, which calls for more sandboxing and security research.

References

- [1] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.
- [2] John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.
- [3] Jinheon Baek, Sujay Kumar Jauhar, Silviu Cucerzan, and Sung Ju Hwang. Researchagent: Iterative research idea generation over scientific literature with large language models. *arXiv preprint arXiv:2404.07738*, 2024.
- [4] Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. Mlagentbench: Evaluating language agents on machine learning experimentation. *arXiv preprint arXiv:2310.03302*, 2023.
- [5] 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. *arXiv preprint arXiv:2410.07095*, 2024.
- [6] Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.
- [7] Yangzhen Wu, Zhiqing Sun, Shanda Li, Sean Welleck, and Yiming Yang. Inference scaling laws: An empirical analysis of compute-optimal inference for problem-solving with language models. *arXiv preprint arXiv:2408.00724*, 2024.
- [8] Richard S Sutton, Andrew G Barto, et al. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.
- [9] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.
- [10] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.
- [11] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8:229–256, 1992.
- [12] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.
- [13] Sebastian B Thrun. *Efficient exploration in reinforcement learning*. Carnegie Mellon University, 1992.
- [14] Sham Machandranath Kakade. *On the sample complexity of reinforcement learning*. University of London, University College London (United Kingdom), 2003.
- [15] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. Rllib: Abstractions for distributed reinforcement learning. In *International conference on machine learning*, pages 3053–3062. PMLR, 2018.
- [16] Matthew W Hoffman, Bobak Shahriari, John Aslanides, Gabriel Barth-Maron, Nikola Momchev, Danila Sinopalnikov, Piotr Stańczyk, Sabela Ramos, Anton Raichuk, Damien Vincent, et al. Acme: A research framework for distributed reinforcement learning. *arXiv preprint arXiv:2006.00979*, 2020.
- [17] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint arXiv: 2409.19256*, 2024.

- [18] Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*, 2024.
- [19] Harrison Chase. LangChain, October 2022.
- [20] Significant Gravitas. AutoGPT.
- [21] 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.
- [22] 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.
- [23] Yunxiang Zhang, Muhammad Khalifa, Shitanshu Bhushan, Grant D Murphy, Lajanugen Logeswaran, Jaekyeom Kim, Moontae Lee, Honglak Lee, and Lu Wang. Mlrc-bench: Can language agents solve machine learning research challenges? *arXiv preprint arXiv:2504.09702*, 2025.
- [24] 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*, 2024.
- [25] Tommaso Bendinelli, Artur Dox, and Christian Holz. Exploring llm agents for cleaning tabular machine learning datasets. *arXiv preprint arXiv:2503.06664*, 2025.
- [26] Tidor-Vlad Pricope. Hardml: A benchmark for evaluating data science and machine learning knowledge and reasoning in ai. *arXiv preprint arXiv:2501.15627*, 2025.
- [27] Shujian Zhang, Chengyue Gong, Lemeng Wu, Xingchao Liu, and Mingyuan Zhou. Automl-gpt: automatic machine learning with gpt (2023). URL <https://arxiv.org/abs/2305.02499>.
- [28] Siyi Liu, Chen Gao, and Yong Li. Large language model agent for hyper-parameter optimization. *arXiv preprint arXiv:2402.01881*, 2024.
- [29] Yang Gu, Hengyu You, Jian Cao, Muran Yu, Haoran Fan, and Shiyong Qian. Large language models for constructing and optimizing machine learning workflows: A survey. *arXiv preprint arXiv:2411.10478*, 2024.
- [30] Zhengyao Jiang, Dominik Schmidt, Dhruv Srikanth, Dixing Xu, Ian Kaplan, Deniss Jacenko, and Yuxiang Wu. Aide: Ai-driven exploration in the space of code. *arXiv preprint arXiv:2502.13138*, 2025.
- [31] Zujie Liang, Feng Wei, Wujiang Xu, Lin Chen, Yuxi Qian, and Xinhui Wu. I-mcts: Enhancing agentic automl via introspective monte carlo tree search. *arXiv preprint arXiv:2502.14693*, 2025.
- [32] Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. *arXiv preprint arXiv:2310.01798*, 2023.
- [33] Federico Errica, Giuseppe Siracusano, Davide Sanvito, and Roberto Bifulco. What did i do wrong? quantifying llms’ sensitivity and consistency to prompt engineering. *arXiv preprint arXiv:2406.12334*, 2024.
- [34] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*, 12, 1999.

- [35] Sham M Kakade. A natural policy gradient. *Advances in neural information processing systems*, 14, 2001.
- [36] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.
- [37] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3):279–292, 1992.
- [38] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ international conference on intelligent robots and systems*, pages 5026–5033. IEEE, 2012.
- [39] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of artificial intelligence research*, 47:253–279, 2013.
- [40] Danijar Hafner, James Davidson, and Vincent Vanhoucke. Tensorflow agents: Efficient batched reinforcement learning in tensorflow. *arXiv preprint arXiv:1709.02878*, 2017.
- [41] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.
- [42] Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36:53728–53741, 2023.
- [43] Paul Christiano, Jan Leike, Tom B. Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences, 2023.
- [44] Daniel M. Ziegler, Nisan Stiennon, Jeffrey Wu, Tom B. Brown, Alec Radford, Dario Amodei, Paul Christiano, and Geoffrey Irving. Fine-tuning language models from human preferences, 2020.
- [45] Harrison Lee, Samrat Phatale, Hassan Mansoor, Kellie Ren Lu, Thomas Mesnard, Johan Ferret, Colton Bishop, Ethan Hall, Victor Carbune, and Abhinav Rastogi. Rlaif: Scaling reinforcement learning from human feedback with ai feedback. 2023.
- [46] Quy-Anh Dang and Chris Ngo. Reinforcement learning for reasoning in small llms: What works and what doesn’t. *arXiv preprint arXiv:2503.16219*, 2025.
- [47] Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I Wang. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution. *arXiv preprint arXiv:2502.18449*, 2025.
- [48] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.
- [49] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang and Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. Agentbench: Evaluating llms as agents, 2023.
- [50] Charlie Snell, Mengjiao Yang, Justin Fu, Yi Su, and Sergey Levine. Context-aware language modeling for goal-oriented dialogue systems. *arXiv preprint arXiv:2204.10198*, 2022.
- [51] Hao Bai, Yifei Zhou, Jiayi Pan, Mert Cemri, Alane Suhr, Sergey Levine, and Aviral Kumar. Digirl: Training in-the-wild device-control agents with autonomous reinforcement learning. *Advances in Neural Information Processing Systems*, 37:12461–12495, 2024.

- [52] Siddharth Verma, Justin Fu, Mengjiao Yang, and Sergey Levine. Chai: A chatbot ai for task-oriented dialogue with offline reinforcement learning. *arXiv preprint arXiv:2204.08426*, 2022.
- [53] Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried and Uri Alon, and Graham Neubig. Webarena: A realistic web environment for building autonomous agents, 2024.
- [54] Karthik Narasimhan, Tejas Kulkarni, and Regina Barzilay. Language understanding for text-based games using deep reinforcement learning. *arXiv preprint arXiv:1506.08941*, 2015.
- [55] Charlie Snell, Ilya Kostrikov, Yi Su, Mengjiao Yang, and Sergey Levine. Offline rl for natural language generation with implicit language q learning. *arXiv preprint arXiv:2206.11871*, 2022.
- [56] Sanjiban Choudhury. Process reward models for llm agents: Practical framework and directions. *arXiv preprint arXiv:2502.10325*, 2025.
- [57] Lunjun Zhang, Arian Hosseini, Hritik Bansal, Mehran Kazemi, Aviral Kumar, and Rishabh Agarwal. Generative verifiers: Reward modeling as next-token prediction. *arXiv preprint arXiv:2408.15240*, 2024.
- [58] Dakota Mahan, Duy Van Phung, Rafael Rafailov, Chase Blagden, Nathan Lile, Louis Castricato, Jan-Philipp Fränken, Chelsea Finn, and Alon Albalak. Generative reward models. *arXiv preprint arXiv:2410.12832*, 2024.
- [59] Jiayi Pan, Yichi Zhang, Nicholas Tomlin, Yifei Zhou, Sergey Levine, and Alane Suhr. Autonomous evaluation and refinement of digital agents. *arXiv preprint arXiv:2404.06474*, 2024.
- [60] David Venuto, Sami Nur Islam, Martin Klissarov, Doina Precup, Sherry Yang, and Ankit Anand. Code as reward: Empowering reinforcement learning with vlms. *arXiv preprint arXiv:2402.04764*, 2024.
- [61] Guijin Son, Hyunwoo Ko, Hoyoung Lee, Yewon Kim, and Seunghyeok Hong. Llm-as-a-judge & reward model: What they can and cannot do. *arXiv preprint arXiv:2409.11239*, 2024.
- [62] Shivam Singhal, Cassidy Laidlaw, and Anca Dragan. Reliability-aware preference learning for llm reward models.

Appendix

In this appendix, we provide additional details on the method (Appendix A), additional details on the experimental setups (Appendix B), prompts to LLMs (Appendix C), and additional experimental results (Appendix D).

A Additional Method Details

A.1 Algorithm for Duration-Aware Gradient Updates and Environment Instrumentation

Algorithm 1: Policy Gradient with Duration-Aware Gradient Updates and Environment Instrumentation

Input: An LM agent π_θ with policy parameters θ , learning rate γ , batch size B , sampling multiplier m , total training iterations N , a code execution environment \mathcal{P} , another copy of the LM `env-inst`, a dataset containing task descriptions \mathcal{D} , an empty buffer for previous solutions $\mathcal{D}_{\text{prev}}$.

for $iteration = 1$ **to** N **do**

 Sample $m \cdot B$ prompts $s \in \mathcal{S} \sim \mathcal{D} \cup \mathcal{D}_{\text{prev}}$

 Sample solution for each prompt from the policy $a \sim \pi_\theta(\cdot|s)$

 Perform environment instrumentation and execute solution $s' \sim \mathcal{P}(s, \text{env-inst}(a))$

 Wait until B executions complete, each of which takes Δt and emits a reward

$\mathcal{R}(s, \text{env-inst}(a))$

 Compute duration weighted policy gradient: $\nabla_\theta J(\pi_\theta) = E \left[\Delta t \cdot \nabla_\theta \log \pi_\theta(a|s) \hat{A}(s, a) \right]$

 Update policy parameters: $\theta \leftarrow \theta - \gamma \nabla_\theta J(\pi_\theta)$

 Update previous solutions: $\mathcal{D}_{\text{prev}} \leftarrow \text{self-improve prompt given } a \text{ and } s'$

B Additional Experimental Details

B.1 Hyperparameters

Table 3: Hyperparameters for RL training of MLE agent.

Hyperparameter	Value
max_prompt_length	1024
max_response_length	1024
train_batch_size	128
total_epochs	100
nnodes	1
n_gpus_per_node	8
actor_model_type	Qwen2.5-3B-Instruct
actor_enable_gradient_checkpointing	True
actor_ppo_mini_batch_size	128
actor_ppo_micro_batch_size	8
actor_grad_clip	1.0
actor_clip_ratio	0.2
actor_entropy_coeff	0.001
actor_ppo_epochs	100
actor_learning_rate	1e-5
reference_log_prob_micro_batch_size	8
rollout_name	vllm
temperature	0.7
top_k	-1
top_p	1
prompt_length	1024
response_length	1024
dtype	bfloat16
gpu_memory_utilization	0.2
ignore_eos	False
enforce_eager	True
tensor_model_parallel_size	8
log_prob_micro_batch_size	8
critic_optimizer	
critic_learning_rate	1e-5
critic_model_type	Qwen2.5-3B-Instruct
critic_enable_gradient_checkpointing	True
critic_ppo_mini_batch_size	128
critic_ppo_micro_batch_size	8
critic_forward_micro_batch_size	8
critic_ppo_epochs	100
critic_grad_clip	1.0
critic_cliprange_value	0.5
ppo_gamma	1.0
ppo_lam	1.0
adv_estimator	gae
ppo_kl_coef	0.001

B.2 Computational Resources

We train the Qwen2.5-3B-Instruct model until convergence. We train on each task using 8 A100-40GiB NVIDIA GPUs using the hyperparameters above. The amount of time until convergence depends on the task, which varies from 1 to 3 days.

C Example Prompts

C.1 Prompts for Solving Kaggle Challenge

Example Prompt for random-acts-of-pizza

<lim_start>system 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.<lim_end>
<lim_start>user First outline your proposed solution in natural language (3-5 sentences), followed by a single markdown code block (wrapped in “”). Note:

- Datasets train.json and test.json are available in ‘/workdir/random-acts-of-pizza/prepared/public/’.
- You MUST produce a submission file at ‘./submission.csv’ by running your model on the test split.
- Your task is to predict the probability that a request resulted in a free pizza.
- Submissions are evaluated on the ROC curve between the predicted probability and the observed outcomes.
- You can use pre-installed packages such as: ‘statsmodels’, ‘pandas’, ‘timm’, ‘bayesian-optimization’, ‘scikit-learn’, ‘xgboost’, ‘numpy’, ‘torch’, ‘torchvision’, ‘lightGBM’, ‘torch-geometric’.
- You can’t access the internet so don’t use any pre-trained models need downloading.
- ‘./submission.csv’ should have the following format: “ request_id,requester_received_pizza t3_i8iy4,0 t3_1mfqi0,0 etc “

• Data snippet: -> /workdir/random-acts-of-pizza/prepared/public/test.json:

```
[ "giver_username_if_known": "N/A",  
  "request_id": "t3_1aw5zf",  
  "request_text_edit_aware": "Basically I had unexpected expenses this month out of  
money and desperate for a pizza. I Have a Tera account with level 48 Beserker and the  
account has founder status.Its not much but only thing i have right now that i can live  
without. Eating is much higher on my priority list right now than playing Tera. If you  
don't want the account I will be happy to pay it forward to someone this friday when I  
get my paycheck.",  
  "request_title": "[Request] Don't have much but willing to trade.",  
  "requester_account_age_in_days_at_request": 165.9420949074074,  
  "requester_days_since_first_post_on_raop_at_request": 0.0,  
  "requester_number_of_comments_at_request": 13,  
  "requester_number_of_comments_in_raop_at_request": 0,  
  "requester_number_of_posts_at_request": 1,  
  "requester_number_of_posts_on_raop_at_request": 0,  
  "requester_number_of_subreddits_at_request": 6,  
  "requester_subreddits_at_request": [  
    "TeraOnline",  
    "Torchlight",  
    "funny",  
    "pics",  
    "todayilearned",  
    "windowsphone" ],  
  "requester_upvotes_minus_downvotes_at_request": 168,  
  "requester_upvotes_plus_downvotes_at_request": 240,  
  "requester_username": "VirginityCollector",  
  "unix_timestamp_of_request": 1364094882.0,  
  "unix_timestamp_of_request_utc": 1364091282.0 , ...  
<lim_end><lim_start>assistant
```

C.2 Prompts for Self-Improvement

Example Prompt for Self-Improvement

<lim_start>system

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.

<lim_end>

<lim_start>user

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. Previous solution:

“{previous_plan_code}”

<lim_end>

<lim_start>assistant:

C.3 Prompt for Environment Instrumentation

Environment Instrumentation

Please insert print statements in the given python script. The print statements are supposed to reflect the progress of executing a script that solves a Kaggle challenge machine learning benchmark. These print statements will be used to debug the python script so it needs to capture the progress of execution.

Print Statements:

- print("imported packages")
- print("loaded data")
- print("defined model")
- print("training loss:")
- print("trained model")
- print("testing loss:")
- print("predicted test labels")

Requirements:

- Only insert print statement AFTER an operation is actually performed (e.g., data have actually been loaded).
- Insert print statements for "training loss:" and "testing loss:" if applicable (i.e., the code actually computes training or testing losses).
- Output the entire python script after inserting print statements in a single markdown code block (wrapped in “”).
- Do not modify the original python code, other than inserting print statements.

Now please insert print statements for this python script: {code}

D Additional Results

D.1 Average Scores Achieved during RL Training

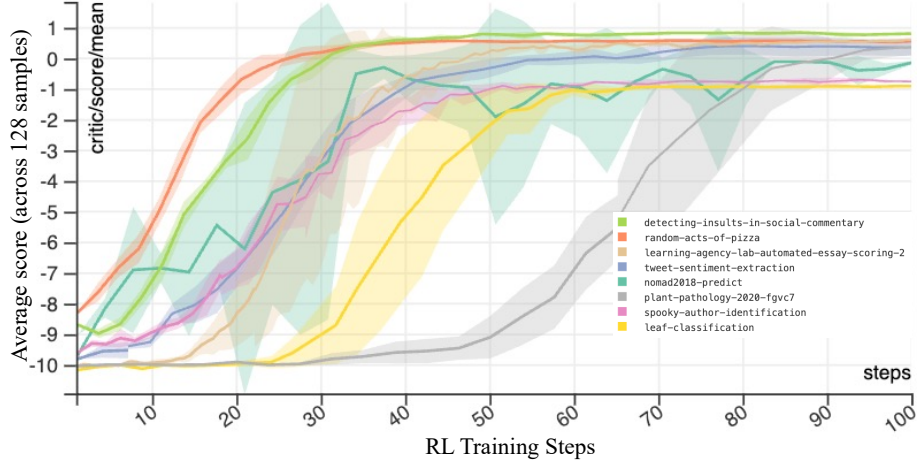


Figure 10: **Average scores** achieved during RL training for different tasks. Scores are -10 for invalid solutions and the actual score from the grader in MLEBench [5] if scores are valid. RL consistently improves average scores across tasks and across 5 seeds per task.

D.2 Full Results for Ablating Self-Improvement Prompt

Table 4: **Improve a Previous Solution** leads to further improvement than only solving the task from scratch on 10 out of the 12 tasks.

Tasks	From Scratch	Improve Previous
detecting-insults-in-social-commentary (↑)	0.898 +/- 0.003	0.895 +/- 0.001
learning-agency-lab-automated-essay-scoring-2 (↑)	0.729 +/- 0.004	0.746 +/- 0.002
random-acts-of-pizza (↑)	0.643 +/- 0.004	0.663 +/- 0.011
tweet-sentiment-extraction (↑)	0.593 +/- 0.000	0.596 +/- 0.002
tabular-playground-series-may-2022 (↑)	0.902 +/- 0.000	0.913 +/- 0.000
tabular-playground-series-dec-2021 (↑)	0.950 +/- 0.001	0.951 +/- 0.000
us-patent-pharse-to-pharse-matching (↑)	0.517 +/- 0.002	0.527 +/- 0.003
plant-pathology-2020-fgvc7 (↑)	0.949 +/- 0.017	0.970 +/- 0.004
leaf-classification (↓)	0.469 +/- 0.244	0.124 +/- 0.000
nomad2018-predict-transparent-conductors (↓)	0.060 +/- 0.000	0.059 +/- 0.000
spooky-author-identification (↓)	0.448 +/- 0.000	0.404 +/- 0.011
lmsys-chatbot-arena (↓)	1.098 +/- 0.003	1.081 +/- 0.002

D.3 Additional Qualitative Examples

D.3.1 Code Example before and after Environment Instrumentation

```
1 import pandas as pd
2 from sklearn.feature_extraction.text import TfidfVectorizer
3 from sklearn.model_selection import train_test_split
4 from sklearn.ensemble import RandomForestClassifier
5 from sklearn.metrics import roc_auc_score
6 import numpy as np
7
8 # Load and preprocess data
9 train_data = pd.read_json("/workdir/random-acts-of-pizza/prepared/public/train.json")
10 test_data = pd.read_json("/workdir/random-acts-of-pizza/prepared/public/test.json")
11
12 # Preprocess text data
13 tfidf_vectorizer = TfidfVectorizer(stop_words="english", max_features=1000)
14 X_train_tfidf = tfidf_vectorizer.fit_transform(train_data["request_text_edit_aware"])
15 X_test_tfidf = tfidf_vectorizer.transform(test_data["request_text_edit_aware"])
16
17 # Train a RandomForestClassifier
18 rf_classifier = RandomForestClassifier(n_estimators=100, random_state=42)
19
20 rf_classifier.fit(X_train_tfidf, train_data["requester_received_pizza"])
21
22 # Predict probabilities for the test set
23 y_pred_proba = rf_classifier.predict_proba(X_test_tfidf)[: , 1]
24
25 # Prepare submission file
26 submission = pd.DataFrame(
27     {"request_id": test_data["request_id"], "requester_received_pizza": y_pred_proba}
28 )
29 submission.to_csv("./submission.csv", index=False)
```

Figure 11: Example code before environment instrumentation for solving the random-acts-of-pizza Kaggle task.

```

1 import pandas as pd
2 from sklearn.feature_extraction.text import TfidfVectorizer
3 from sklearn.model_selection import train_test_split
4 from sklearn.ensemble import RandomForestClassifier
5 from sklearn.metrics import roc_auc_score
6 import numpy as np
7
8 print("imported packages")
9
10 # Load and preprocess data
11 train_data = pd.read_json("/workdir/random-acts-of-pizza/prepared/public/train.json")
12 test_data = pd.read_json("/workdir/random-acts-of-pizza/prepared/public/test.json")
13
14 print("loaded data")
15
16 # Preprocess text data
17 tfidf_vectorizer = TfidfVectorizer(stop_words="english", max_features=1000)
18 X_train_tfidf = tfidf_vectorizer.fit_transform(train_data["request_text_edit_aware"])
19 X_test_tfidf = tfidf_vectorizer.transform(test_data["request_text_edit_aware"])
20
21 # Train a RandomForestClassifier
22 rf_classifier = RandomForestClassifier(n_estimators=100, random_state=42)
23
24 print("defined model")
25
26 rf_classifier.fit(X_train_tfidf, train_data["requester_received_pizza"])
27
28 print("trained model")
29
30 # Predict probabilities for the test set
31 y_pred_proba = rf_classifier.predict_proba(X_test_tfidf)[: , 1]
32
33 print("predicted test labels")
34
35 # Prepare submission file
36 submission = pd.DataFrame(
37     {"request_id": test_data["request_id"], "requester_received_pizza": y_pred_proba}
38 )
39 submission.to_csv("./submission.csv", index=False)

```

Figure 12: Example code after environment instrumentation for solving the random-acts-of-pizza Kaggle task. All print statements are generated by environment instrumentation.

D.3.2 Example High-Performing Solutions Found by the Agent after RL Training

```
1 """Plan: My proposed solution involves training a machine learning model using a
    combination of text data from request titles and descriptions, along with user-
    specific features such as account age, number of posts, etc. I will preprocess the
    text data using TF-IDF vectorization and concatenate it with the numerical features.
    The model will be a random forest classifier to handle the binary classification
    task of predicting whether a request will result in a free pizza. After training the
    model on the training set, I will fine-tune its hyperparameters using Bayesian
    optimization, and then use it to predict the probability of a free pizza for each
    request in the test set."""
2
3 # Import necessary libraries
4 import pandas as pd
5 from sklearn.ensemble import RandomForestClassifier
6 from sklearn.feature_extraction.text import TfidfVectorizer
7 from sklearn.model_selection import GridSearchCV
8 import numpy as np
9
10 # Load and preprocess data
11 train_df = pd.read_json('/workdir/random-acts-of-pizza/prepared/public/train.json')
12 test_df = pd.read_json('/workdir/random-acts-of-pizza/prepared/public/test.json')
13
14 # Feature engineering
15 tfidf = TfidfVectorizer(stop_words='english', max_features=1000)
16 tfidf.fit(train_df['request_text_edit_aware'])
17 train_tfidf = tfidf.transform(train_df['request_text_edit_aware'])
18 test_tfidf = tfidf.transform(test_df['request_text_edit_aware'])
19
20 # Combine features with user-specific features
21 train_features = np.hstack([train_df[['requester_account_age_in_days_at_request',
22                                     'requester_days_since_first_post_on_raop_at_request',
23                                     'requester_number_of_comments_at_request',
24                                     'requester_number_of_comments_in_raop_at_request',
25                                     'requester_number_of_posts_at_request',
26                                     'requester_number_of_posts_on_raop_at_request',
27                                     'requester_number_of_subreddits_at_request',
28                                     'requester_upvotes_minus_downvotes_at_request',
29                                     'requester_upvotes_plus_downvotes_at_request']],
30                             train_tfidf.toarray()])
31 test_features = np.hstack([test_df[['requester_account_age_in_days_at_request',
32                                     'requester_days_since_first_post_on_raop_at_request',
33                                     'requester_number_of_comments_at_request',
34                                     'requester_number_of_comments_in_raop_at_request',
35                                     'requester_number_of_posts_at_request',
36                                     'requester_number_of_posts_on_raop_at_request',
37                                     'requester_number_of_subreddits_at_request',
38                                     'requester_upvotes_minus_downvotes_at_request',
39                                     'requester_upvotes_plus_downvotes_at_request']],
40                             test_tfidf.toarray()])
41
42 # Train model
43 rfc = RandomForestClassifier(n_estimators=100, random_state=42)
44 param_grid = {'max_depth': np.arange(3, 15), 'min_samples_leaf': np.arange(2, 10)}
45 grid_search = GridSearchCV(rfc, param_grid, cv=3, scoring='roc_auc')
46 grid_search.fit(train_features, train_df['requester_received_pizza'])
47
48 # Predict probabilities on test set
49 test_probs = grid_search.predict_proba(test_features)[: , 1]
50
51 # Prepare submission file
52 submission_df = pd.DataFrame({'request_id': test_df['request_id'],
53                               'requester_received_pizza': test_probs})
54 submission_df.to_csv('./submission.csv', index=False)
```

Figure 13: An example of high-performance (score 0.66) high cost (115 seconds) solution for random-acts-of-pizza achieved after RL training.

```

1 """Plan: The proposed solution involves preprocessing the train and test datasets to
   extract relevant features, training a model on the extracted features, and then
   predicting scores for the test set. For this task, I will use a transformer-based
   model from the 'timm' library to process the text data, and a lightGBM classifier
   for the prediction task. The model will be trained on the text features extracted
   from the train data and then evaluated on the test set to generate the submission
   file."""
2
3 import pandas as pd
4 import numpy as np
5 from sklearn.model_selection import train_test_split
6 from lightgbm import LGBMClassifier
7 from xgboost import XGBClassifier
8 from sklearn.feature_extraction.text import TfidfVectorizer
9 from tqdm.notebook import tqdm
10
11 # Load data
12 train_df = pd.read_csv("/workdir/learning-agency-lab-automated-essay-scoring-2/prepared/
   public/train.csv")
13 test_df = pd.read_csv("/workdir/learning-agency-lab-automated-essay-scoring-2/prepared/
   public/test.csv")
14
15 # Extract Text Features
16 tfidf_vectorizer = TfidfVectorizer()
17 train_tfidf = tfidf_vectorizer.fit_transform(train_df['full_text'])
18 test_tfidf = tfidf_vectorizer.transform(test_df['full_text'])
19
20 # Train Model
21 model = LGBMClassifier(n_estimators=1000, max_depth=10)
22 model.fit(train_tfidf, train_df['score'])
23
24 # Predict on Test Set
25 test_preds = model.predict(test_tfidf)
26
27 # Save Submission
28 submission = pd.DataFrame({
29     "essay_id": test_df.essay_id,
30     "score": test_preds
31 })
32 submission.to_csv("submission.csv", index=False)

```

Figure 14: An example of high-performance (score 0.73) high cost (281 seconds) solution using gradient boosting for learning-agency-lab-automated-essay-scoring-2 achieved after RL training.