MULTI-TURN CODE GENERATION THROUGH SINGLE-STEP REWARDS

Arnav Kumar Jain[*1 2], **Gonzalo Gonzalez-Pumariega**[*3], **Wayne Chen**[3], **Alexander M Rush**[3], **Wenting Zhao**[† 3], **Sanjiban Choudhury**[† 3]
[1]Mila- Quebec AI Institute, [2]Université de Montréal, [3]Cornell University

ABSTRACT

We address the problem of code generation from multi-turn execution feedback. Existing methods either generate code without feedback or use complex, hierarchical reinforcement learning to optimize multi-turn rewards. We propose a simple yet scalable approach, μCODE, that solves multi-turn code generation using only single-step rewards. Our key insight is that code generation is a one-step recoverable MDP, where the correct code can be recovered from any intermediate code state in a single turn. μCODE iteratively trains both a generator to provide code solutions conditioned on multi-turn execution feedback and a verifier to score the newly generated code. Experimental evaluations show that our approach achieves significant improvements over the state-of-the-art baselines. We provide analysis of the design choices of the reward models and policy, and show the efficacy of μCODE at utilizing the execution feedback. Our code is available here.

1 INTRODUCTION

Software engineers often iteratively refine their code based on execution errors. A common strategy for machine code generation is thus to repair code using execution feedback at test time (Chen et al., 2024; Wang et al., 2024b; Zhao et al., 2024). However, prompting alone is insufficient as it cannot teach how to recover from all possible errors within a limited context.

We need to train models that can learn from execution feedback during training. Existing approaches fall into either single-turn or multi-turn settings. In the single-turn setting, methods either train without execution feedback (Zelikman et al., 2022) or perform one-step corrections (Welleck et al., 2023; Ni et al., 2024). However, these struggle to iteratively correct errors over multiple turns. Multi-turn approaches, on the other hand, rely on complex reinforcement learning (RL) (Gehring et al., 2024a; Kumar et al., 2024b; Zhou et al., 2024) to optimize long-term rewards. While effective in principle, these methods suffer from sparse learning signals which makes learning inefficient.

Our key insight is that code generation is *a one-step recoverable Markov Decision Process (MDP), implying that the correct code can be recovered from any intermediate state in a single step*. This allows us to greedily maximize a one-step reward instead of relying on complex multi-step reward optimization. As a result, this reduces the problem from reinforcement learning, which requires exploration and credit assignment, to imitation learning, where the model simply learns to mimic correct code, leading to a more stable and efficient training process.

We propose μCODE, a simple and scalable approach for multi-turn code generation from execution feedback. During training, μCODE follows an *expert iteration* (Anthony et al., 2017) framework with a *local search expert*, enabling iterative improvement of both the generator and the expert. The process begins by rolling out the current code generator to collect interaction data with execution feedback. A single-step verifier is then trained on this data and utilized to guide a local search expert in refining the code and generating training labels. Finally, the generator is fine-tuned using these labels. Given recent trends of test-time scaling in generating high quality solutions (Brown et al., 2024; Snell et al., 2024; Wu et al., 2024), μCODE also uses the learned verifier for inference-time

[*]Equal contribution. Correspondence to: Arnav (arnav-kumar.jain@mila.quebec) and Gonzalo (gg387@cornell.edu)
[†]Equal advising

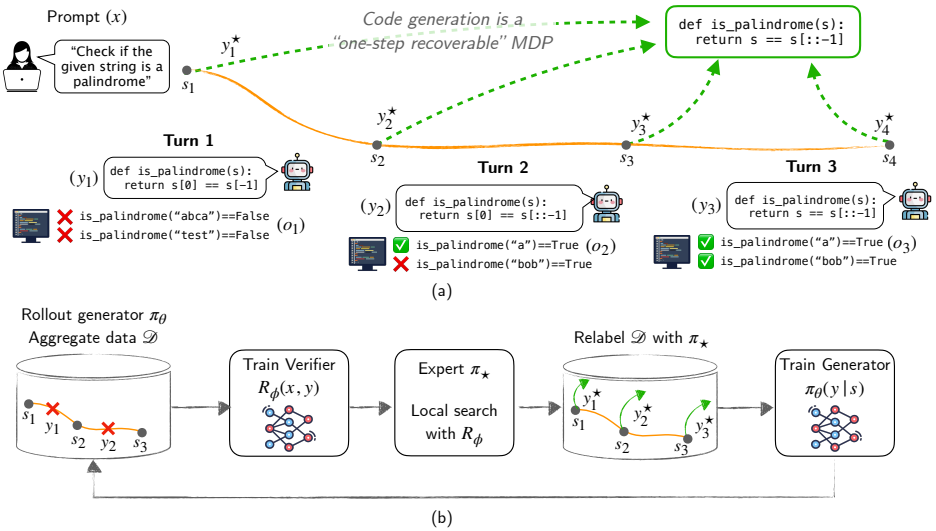

Figure 1: (a) We define the task of multi-turn code generation where for an initial problem x, the generator π_θ provides a solution y_1. This solution is evaluated with the public test to get execution feedback o_1. At a turn t, the generator is conditioned on the history to generate solution $y_t \sim \pi_\theta(.|x, y_{<t}, o_{<t})$. The rollout ends when the turn limit is reached or the public tests pass upon which the solution is executed on private tests. Since, the agents can generate the optimal solution at any turn, this is a 1-step recoverable process. (b) Training loop of our method μCODE – which comprises of a generator and a learned verifier. During each iteration, rollouts are collected using π_θ and we train a verifier R_ϕ to rank candidate solutions for a prompt. The verifier R_ϕ is then used to construct a local expert and relabel the collected rollouts. Lastly, the generator is fine-tuned with this expert dataset.

scaling. Here, μCODE samples N trajectories; at each step, μCODE picks the best code solution ranked by the learned verifier.

The key contributions of this work are as follows:

1. A novel framework, μCODE, for training code generators and verifiers through multi-turn execution feedback. We add theoretical analysis of performance bounds using the property of one-step recoverability for this task.
2. We propose a *multi-turn Best-of-N (BoN) approach* for inference-time scaling and present benefits of a learned verifier to select the code solution at each turn.
3. Our approach μCODE outperforms leading multi-turn approaches on MBPP (Austin et al., 2021) and HumanEval (Chen et al., 2021) benchmarks. Our ablations demonstrate that learned verifiers aid in learning better generators and show promising scaling law trends with higher inference budgets.

2 BACKGROUND

Multi-turn Code Generation as a MDP. In multi-turn code generation, an agent iteratively refines a program to maximize its correctness on private test cases. Given an initial problem prompt x, at each turn t, the agent generates a complete code snippet y_t and executes it on a set of public tests. The outcomes o_t from these tests serve as observations that guide subsequent refinements. This process continues until the agent generates a code snippet y_t that passes all public tests, at which point the episode terminates, or until the maximum number of turns T is reached without success. The first successful code, y_t, is then evaluated on private tests to compute the correctness score $C(x, y_t) \in \{0, 1\}$.

We model this as a Markov Decision Process (MDP), where the state is the interaction history $s_t = \{x, y_1, o_1, \ldots, y_{t-1}, o_{t-1}\}$ where $s_1 = \{x\}$, and the action is the code $a_t = y_t$. The oracle

Algorithm 1 μCODE: Training

input Initial generator π_0, multi-turn code environment $\mathcal{E}$, and max iterations M
1: **for** iteration i = 1 ... M **do**
2: Rollout generator π_θ in multi-turn environment $\mathcal{E}$ to collect datapoints $\mathcal{D}_i \leftarrow \{(x, s_t, y_t, o_t))\}$
3: Aggregate data $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}_i$
4: Train a verifier $R^i_\phi(x, y)$ on $\mathcal{D}$
5: Construct a local search expert using verifier $\pi^i_\star(x) = \arg\max_{y \in \mathcal{D}(x)} R^i_\phi(x, y)$
6: Relabel data $\mathcal{D}$ with $\pi^i_\star(x)$ to get $\mathcal{D}^i_\star$
7: Train π^i_θ with fine-tuning (FT) on $\mathcal{D}^i_\star$
8: **end for**
output Best generator π_θ and verifier R_ϕ

reward is defined as $R(s_t, a_t) = R(x, a_t) = C(x, y_t)$ if y_t passes all public and private tests (terminating the episode), or 0 otherwise.

During training, given a dataset of problem prompts $\mathcal{D}$, the goal is to find a generator $\pi_\theta(y_t|x, y_1, o_1, \ldots, y_{t-1}, o_{t-1})$, that maximizes the cumulative discounted reward $R(x, y_t)$:

$$\max_{\pi_\theta} \mathbb{E}_{x \sim \mathcal{D}, y_t \sim \pi_\theta(\cdot|s_t)} \left[\sum_{t=1}^{T} \gamma^t R(x, y_t) \right]. \tag{1}$$

3 μCODE: MULTI-TURN CODE GENERATION

We propose μCODE, a simple and scalable algorithm for multi-turn code generation using execution feedback. μCODE follows an *expert iteration* (Anthony et al., 2017) framework with a *local search expert*. μCODE iteratively trains two components – a *learned verifier* R_ϕ to score code snippets (Section 3.2), and a *generator* π_θ to imitate local search with the verifier (Section 3.3). This iterative process allows the generator and expert to bootstrap off each other, leading to continuous improvement. At inference time, both the generator and verifier are used as BoN search to select and execute code (Section 3.4). Finally, we analyze the performance of μCODE in Section 3.5.

3.1 THE μCODE ALGORITHM

Algorithm 1 presents the iterative training procedure. At an iteration i, the current generator π_θ is rolled out in the multi-turn code environment $\mathcal{E}$ to generate interaction data $\mathcal{D}_i \leftarrow \{(x, s_t, y_t, r_t)\}$. Every turn t in $\mathcal{D}_i$ includes the prompt x, interaction history s_t, code generated y_t and the correctness score from the oracle verifier $r_t = R(x, y_t)$. This data is then aggregated $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}_i$. The learned verifier R^i_ϕ is trained on the aggregated data $\mathcal{D}$. An expert is created using R^i_ϕ to perform local search to find the optimal action $\pi^i_\star(x) = \arg\max_{y \in \mathcal{D}(x)} R^i_\phi(x, y)$, where $\mathcal{D}(x)$ are all the code completions for a given prompt x. The expert $\pi^i_\star(x)$ relabels the data $\mathcal{D}$ with the optimal action. The generator π^i_θ is then trained via fine-tuning (FT) on $\mathcal{D}$. This process iterates M times, and the best generator and verifier pair on the validation dataset are returned.

3.2 TRAINING VERIFIER

The learned verifier provides dense scores to code solutions for a given problem. At train time, this is used by the expert to perform local search to obtain optimal code. At inference time, the verifier is used for multi-turn BoN (3.4) for efficient search. The learned verifier has two distinct advantages over process reward functions typically used in multi-turn RL: (1) It is conditioned only on the initial prompt and the current solution, and is not dependent on previous states (2) It is trained via supervised learning on oracle reward labels. We explore two different losses:

Algorithm 2 μCODE: Inference loop

input Generator π_θ, learned verifier R_ϕ, turn limit T, number of rollouts N, public tests, and private tests

Set $s_1 = \{x\}$, $t = 1$
while true **do**
  Generate N rollouts $\{y_t^n\}_{n=1}^N \sim \pi_\theta(.|s_t)$
  Choose best solution $y_t^* = \arg\max_n R_\phi(x, y_t^n)$
  Execute y_t^* to get execution feedback o_t
  if y_t^* passes public tests or $t = T$ **then**
    break;
  end if
  Update state $s_{t+1} = \{s_t, y_t^*, o_t\}$ and increment t
end while

output Return y^* to execute on public and private tests

Binary Cross-Entropy loss (BCE): The nominal way to train the verifier is to directly predict the oracle reward (Cobbe et al., 2021):

$$\begin{aligned}\mathcal{L}_{\text{BCE}}(\phi) = -\mathbb{E}_{(x,y,r)\sim\mathcal{D}}[&r \log R_\phi(x, y) \\ &-(1-r)\log R_\phi(x, y)]\end{aligned} \tag{2}$$

Bradley Terry Model (BT): Since the goal of the verifier is to relatively rank code solutions rather than predict absolute reward, we create a preference dataset and then train with a Bradley Terry loss (Ouyang et al., 2022). For every prompt x, we create pairs of correct y^+ (where $r = 1$) and incorrect y^- (where $r = 0$) code and define the following loss:

$$\mathcal{L}_{BT}(\phi) = -\mathbb{E}_{(x,y^+,y^-)\sim\mathcal{D}}[\log \sigma(R_\phi(x, y^+) - R_\phi(x, y^-))]. \tag{3}$$

where $\sigma(.)$ is the sigmoid function. We hypothesize that BT is strictly easier to optimize as the verifier has to only focus on relative performance. This is also consistent with observations made for training process reward models, where the advantage function is easier to optimize than the absolute Q function (Setlur et al., 2024).

3.3 Training Generator

μCODE comprises a generator π_θ trained to produce code solutions conditioned on the initial problem and execution observations from previous turns. Given a dataset $\mathcal{D}$, μCODE iteratively trains the generator to find the optimal code solution labeled using the local expert over the learned verifier. For this step, μCODE extracts all code solutions from $\mathcal{D}$ for every problem x. An expert is then created by picking the best solution, $y^\star$, which achieves the highest score using with the learned verifier $R_\phi(x, y)$ and is given by

$$y^\star = \pi_\star(x) = \arg\max_{y\in\mathcal{D}(x)} R_\phi(x, y). \tag{4}$$

Using this expert dataset, we relabel the dataset $\mathcal{D}$ with the optimal solutions for each prompt:

$$\mathcal{D}_\star = \{(x, s_t, y^\star) \mid (x, s_t) \sim \mathcal{D}\}, \tag{5}$$

where $\mathcal{D}_\star$ represents the expert dataset. The generator π_θ is then trained via fine-tuning (FT) on this expert dataset $\mathcal{D}_\star$.

3.4 Inference: Multi-turn Best-of-N

At inference time, the goal is to generate a code solution with a fixed inference budget – denoting the number of times generators can provide one complete solution. In this work, we propose to leverage the learned verifier to improve search and code generations over successive turns with *multi-turn Best-of-N* (BoN). To achieve this, μCODE uses a natural extension of BoN to the multi-turn setting. At each turn, the generator produces N one-step rollouts $\{y_t^n\}_{n=1}^N \sim \pi_\theta(.|s_t)$ and the learned verifier picks the most promising code solution among these candidates using

$$y_t^* = \arg\max_n R_\phi(x, y_t^n). \tag{6}$$

The selected code y_t^* is executed in the environment over public tests to obtain the execution feedback o_t. This solution and the feedback is provided as context to the generator at the next turn to repeat this procedure. The search ends once y_t^* passes all public tests or when the turn limit is reached. Consequently, even if $R_\phi(\cdot)$ grants a high score to a code solution, inference continues until the solution has successfully cleared all public tests, thus mitigating potential errors by $R_\phi(\cdot)$. The final response y_t^* is then passed through the oracle verifier to check its correctness. Algorithm 2 describes a description of multi-turn BoN. We found it beneficial to use the reward model trained with samples of the latest generator π_θ (see Table 1).

3.5 Analysis

μCODE effectively treats multi-turn code generation as an interactive imitation learning problem by collecting rollouts from a learned policy and re-labeling them with an expert. It circumvents the exploration burden of generic reinforcement learning which has exponentially higher sample complexity (Sun et al., 2017). We briefly analyze why this problem is amenable to imitation learning and prove performance bounds for μCODE.

Definition 3.1 (One-Step Recoverable MDP)**.** A MDP $\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R, \gamma)$ with horizon T is *one-step recoverable* if the advantage function of the optimal policy π^*, defined as $A^*(s,a) = Q^*(s,a) - V^*(s)$, is uniformly bounded for all (s,a), i.e. $A^*(s,a) \leq 1$.

Code generation is one-step recoverable MDP. Multi-turn code generation satisfies one-step recoverability because the optimal policy $\pi^*(y_t|s_t)$ depends only on the problem prompt x and not the interaction history $s_t = (x, y_1, o_1, \ldots, y_{t-1}, o_{t-1})$. Since the correctness of a code snippet y_t is fully determined by x, the optimal Q-function satisfies $Q^*(s_t, y_t) = R(x, y_t)$, where $R(x, y_t) \in \{0, 1\}$. The optimal value function is $V^*(s_t) = \max_{y_t} R(x, y_t)$, so the advantage function simplifies to $A^*(s_t, y_t) = R(x, y_t) - \max_{y_t'} R(x, y_t') \leq 1$.

Code generation enables efficient imitation learning. There are two challenges to applying interactive imitation learning (Ross et al., 2011; Ross & Bagnell, 2014) – (1) Existence of expert policies or value functions, and (2) Recoverability of expert from arbitrary states. First, for code generation, the expert is simply the one-step reward maximizer $\arg\max_y R(x, y)$. We can efficiently estimate $R_\phi(x, y)$ to compute the expert, without needing to compute value function backups. Second, even if the learner fails to imitate the expert at any given state, the expert can perfectly recover from the next state. This results in the best possible performance bounds for imitation learning, which we formalize below.

Theorem 3.2 (Performance bound for μCODE)**.** *For a one-step recoverable MDP $\mathcal{M}$ with horizon T, running N iterations of μCODE yields at least one policy π such that*

$$J(\pi^*) - J(\pi) \leq O(T(\epsilon + \gamma(N))). \tag{7}$$

where π^* is the expert policy, ϵ is the realizability error, and $\gamma(N)$ is the average regret.

Proof is in Appendix A.4. The bound $O(\epsilon T)$ is much better than the worst-case scenario of $O(\epsilon T^2)$ for unrecoverable MDPs (Swamy et al., 2021). Thus, μCODE exploits the structure of multi-turn code generation to enable imitation learning, bypassing the need for hierarchical credit assignment. More generally, this analysis suggests that for any task where the optimal action is history-independent and recoverable in one step, reinforcement learning can be reduced to efficient imitation learning without loss of performance.

4 Experiments

Through our experiments, we aim to analyze (1) How does μCODE compare to other state-of-the-art methods? (2) Does the learned verifier help during training and inference-time? (3) Which loss function works better for learning a verifier?

4.1 SETUP

Models. The generator model in μCODE is initialized with Llama-3.2-1B-Instruct or Llama-3.1-8B-Instruct (Dubey et al., 2024). The learned verifiers are initialized with the same models as generators and have a randomly initialized linear layer to predict a scalar score (Stiennon et al., 2020).

Datasets. We conduct experiments on MBPP (Austin et al., 2021) and HumanEval (Chen et al., 2021) where the agent needs to generate code solutions in Python given natural language descriptions. We train the methods on the MBPP training set which comprises 374 problems and evaluate on the MBPP test set and HumanEval (HE) dataset which have 500 and 164 problems. We further describe the prompts and the split of public and private tests in Appendix A.6 and A.7.

Baselines. We compare μCODE with single and multi-turn baselines. For the single and multi-turn settings, we report metrics by generating solutions from Llama models which we denote as Instruct. We also compare with STaR (Zelikman et al., 2022) where the correct solutions of the Instruct model are used for fine-tuning (FT). We also compare to a multi-turn version of STaR, called Multi-STaR. Here, we collect multi-turn rollouts using the Instruct model and use trajectories terminating in a correct code solution for FT. For multi-turn BoN search, we collect the solutions that pass public tests, and then we select the best one judged by a learned verifier. Note that this verifier is specifically trained for each generator.

Metrics. We measure the performance with the *BoN* accuracy, which quantifies the accuracy of the solution chosen by a verifier from N candidate solutions. The generator is allowed $T = 3$ turns and the final turn is used for evaluation over private tests. At each turn, the verifier ranks $N = 5$ solutions (unless stated otherwise) provided by the generator. For the BoN performance, we sample with a temperature of 0.7. We also report the accuracy of generating correct solutions via greedy decoding.

Approach		**Llama-3.2-1B**		**Llama-3.1-8B**	
	N	*MBPP*	*HE*	*MBPP*	*HE*
			Single-Turn		
Instruct	1	36.5	28.0	52.1	59.8
STaR	1	35.7	34.1	53.7	54.9
			Multi-Turn		
Instruct	1	38.9	29.3	58.9	60.4
+*BoN*	5	48.5	34.3	<u>68.1</u>	61.2
Multi-STaR	1	36.7	33.5	57.7	59.8
+*BoN*	5	47.9	39.6	<u>68.6</u>	**63.2**
μCODE	1	37.9	35.4	62.3	57.9
+*BoN*	5	**50.7**	**41.7**	**68.8**	<u>62.2</u>

Table 1: Comparison of our method μCODE with baselines across MBPP and HumanEval datasets. $N = 1$ denotes generating solutions with 0 temperature. The Best-of-N (BoN) accuracy is computed with $N = 5$ candidate solutions at each where the public tests and learned verifier is used for selection. We observe that μCODE outperforms competing methods based on Llama-3.2-1B-Instruct and Llama-3.1-8B-Instruct models. The best performance for each dataset and model-sized is highlighted in **bold** and similar performances (within 1%) are <u>underlined</u>.

4.2 RESULTS

In Table 1, we compare the proposed algorithm μCODE with the baselines. Here, we first evaluate the generators using code generated via greedy sampling for each problem ($N = 1$). This measures the accuracy of generating a correct solution with a turn limit of $T = 3$. We observe that multi-turn methods (both Instruct and Multi-STaR) perform better than their single-turn variants showing the importance of incorporating execution feedback. Our approach μCODE outperforms Multi-STaR across both benchmarks with 1B-sized model demonstrating the efficacy of training generators with data obtained with a learned verifier. To highlight, our method μCODE with a 1B parameter model achieves **1.9%** performance gains compared to Multi-STaR on the HumanEval dataset. With an 8B-sized model, μCODE outperforms baselines on MBPP whereas there is a drop when compared on HumanEval.

We further evaluate the effect of having a verifier for BoN search during inference, where a learned verifier selects the most promising candidate solution at each turn. In Table 1, we observe that all algorithms can benefit with BoN search. Remarkably, μCODE attains a performance gain of up

to **12.8**% with the multi-turn BoN approach compared to greedy. Moreover, μCODE outperforms leading baselines with BoN search on MBPP and HumanEval datasets by **2.8**% and **2.1**% with 1B sized-model and performs comparably on 8B-sized model.

4.3 ANALYSIS

To understand the improvements, we conduct a component-wise ablation study where we 1) check the effect of oracle and learned verifiers for relabeling to train the generator (4.3.1), 2) evaluate different generators trained with and without learned verifiers (4.3.2), 3) check which verifier performs better multi-turn BoN search at test-time (4.3.3). Additionally, we assess scaling behaviors at inference time with number of candidate generations (N) at each turn in Appendix A.2 and study the benefits of learned verifiers during evaluation in Appendix A.3.

4.3.1 VERIFIER FOR RELABELING

We compare different verifiers for relabeling data for training the generator. In contrast to μCODE where the learned verifier is used to relabel (LV), we compare with relabeling using a correct solution (attains an oracle reward $R = 1$) for the corresponding prompt (OV). We also compare with a variant where the generator is fine-tuned over combinations of data relabeled with both the oracle verifier and the learned verifier (OV+LV). Here, we concatenate the FT dataset obtained using both LV and OV. In Figure 2, we present results with the 1B-sized models across benchmarks and observe that having corrections with the oracle verifier outcome does not perform well. However, relabeling with both verifiers OV+LV outperforms the OV variant, further demonstrating that gains in the generator learned by μCODE are coming from relabeling with a learned verifier. Lastly, the LV variant performed best on MBPP and comparably on HumanEval dataset when compared with LV+OV.

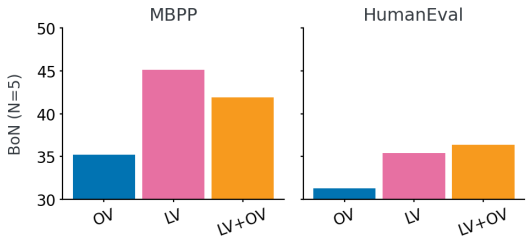

Figure 2: Comparison of relabeling with learned verifier (LV) and oracle verifier (OV) with the 1B model. The variant OV+LV aggregates a dataset from both verifiers for fine-tuning the generator. Note that OV+LV performs better than OV. However, relabeling with LV outperforms on MBPP and performs comparably on HumanEval, thereby demonstrating the benefits of leveraging the learned verifier for training the generator.

4.3.2 VARYING THE GENERATOR

In this section, we compare the multi-turn agents where the generator is trained with an oracle verifier (Multi-STaR) or a learned verifier (μCODE). We evaluate the ability of the trained generator to utilize execution feedback and improve the code response across turns. We report the BoN accuracy till a turn t, which denotes the BoN accuracy of obtaining a correct solution till turn t. In Figure 3, we present the results with 1B-sized models. We observe that BoN accuracy improves with turns for μCODE whereas the baseline agents show marginal improvements with successive turns. We believe that using a learned verifier for relabeling improves the generator's ability to generate solutions with high reward values, and indeed recover better at every turn by utilizing the execution feedback.

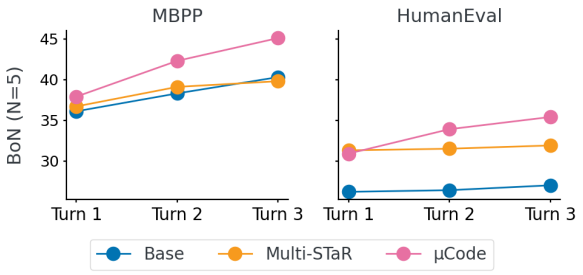

Figure 3: Comparison of μCODE and baselines with 1B models on the ability of the learned generator to incorporate execution feedback at each turn. We observe that μCODE consistently improves the BoN accuracy across turns on both datasets, whereas the baselines show marginal improvements with turns.

4.3.3 VERIFIER AT TEST-TIME

In our experiments with multi-turn BoN (Table 2), we pick the best solution based on the outcome of public tests and the scores of the learned verifier. In this experiment, we study different verifiers for ranking the candidate solutions at each turn. We test with *Random* strategy where the policy randomly picks from the N solutions at each step. We compare to the public tests (PT) outcome that picks any solution that passes the public test. Note that this involves evaluating all generated solutions at every turn with public tests. We also compare to selecting a solution based on scores obtained via the learned verifier only (LV). This is crucial as in certain applications such privileged information like public tests are not available and the agents can benefit from learned verifiers during inference. Lastly, we compare with the combination of public tests and leveraging the scores of the learned verifier to break ties at each turn (PT+LV).

Approach	**Llama-3.2-1B**		**Llama-3.1-8B**	
	MBPP	*HE*	*MBPP*	*HE*
Base				
Random	34.4	23.0	59.3	57.9
LV	40.3	27.0	61.1	61.2
PT	48.6	31.9	67.2	60.4
PT+LV	48.5	34.3	<u>68.1</u>	61.2
Multi-STaR				
Random	35.6	30.5	59.2	57.7
LV	39.8	31.9	61.2	62.8
PT	46.7	37.6	67.6	60.0
PT+LV	47.9	39.6	<u>68.6</u>	**63.2**
μCODE				
Random	37.9	31.5	60.5	59.1
LV	45.1	35.4	64.3	60.4
PT	<u>49.8</u>	39.0	<u>68.7</u>	59.1
PT+LV	**50.7**	**41.7**	**68.8**	<u>62.2</u>

Table 2: Comparing BoN with different ways of picking solutions at each turn for multi-turn BoN search using the 1B sized model. The hierarchical approach of using public test and learned verifier (PT+LV) outperforms only using either public tests (PT) or the learned verifier (LV). The best performance for each dataset and model-size is highlighted in **bold** and similar performances (within 1%) are <u>underlined</u>.

In Table 2, we compare Base, Multi-STaR and μCODE with different verifiers at test-time. We observe that LV outperforms Random strategy which shows that a learned verifier indeed aids in selecting better solutions among generations. Given the benefits of learned verifiers for multi-turn BoN search, they can be a good alternative when public tests are not available. Furthermore, using the outcome of public tests (PT) performed better than using learned verifiers (LV) except on the HumanEval datset for 8B-sized model. We believe that this gap can be further reduced by learning more powerful verifiers and leave it for future research. Interestingly, the hierarchical approach (PT+LV) that uses the learned verifier to break ties on the outcomes of public tests performed best across methods and datasets. We hypothesize that using learned verifiers is beneficial in two scenarios. Firstly, if multiple solutions pass the public tests, then the learned verifier can filter out incorrect solutions which may not pass private tests. Secondly, if all candidate solutions are incorrect, then the learned verifier should choose the most promising solution at each turn. This is crucial as picking a better solution with the learned verifier can lead to more relevant feedback for recovering the true solution.

5 CONCLUSION

We present μCODE, a simple and scalable method for multi-turn code generation through single-step rewards. μCODE models code generation as a one-step recoverable MDP and learns to iteratively improve code with a learned verifier to guide the search. Experimental results demonstrate that μCODE outperforms methods using oracle verifiers by a large margin. We acknowledge the following limitations of this paper. Due to a limited budget, we were only able to train models with up to eight-billion parameters. It is possible that the conclusions made in this paper do not generalize to models of larger scales. Additionally, we train models on MBPP, whose training set has only 374 examples. However, we hypothesize that more training examples will lead to better performance. Finally, our datasets are only in Python, and our findings might not generalize to other programming languages.

ACKNOWLEDGEMENTS

AJ is supported by Fonds de Recherche du Québec (FRQ), Calcul Québec, Canada CIFAR AI Chair program, and Canada Excellence Research Chairs (CERC) program. The authors are also grateful to Mila (mila.quebec) IDT and Digital Research Alliance of Canada for computing resources. AMR is supported in part by NSF CAREER #2037519 and NSF #2242302. SC is supported in part by Google Faculty Research Award, OpenAI SuperAlignment Grant, ONR Young Investigator Award, NSF RI #2312956, and NSF FRR#2327973.

REFERENCES

Thomas Anthony, Zheng Tian, and David Barber. Thinking fast and slow with deep learning and tree search, 2017. URL https://arxiv.org/abs/1705.08439.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*, 2024.

Thomas Carta, Clément Romac, Thomas Wolf, Sylvain Lamprier, Olivier Sigaud, and Pierre-Yves Oudeyer. Grounding large language models in interactive environments with online reinforcement learning, 2024. URL https://arxiv.org/abs/2302.02662.

Baian Chen, Chang Shu, Ehsan Shareghi, Nigel Collier, Karthik Narasimhan, and Shunyu Yao. Fireact: Toward language agent fine-tuning, 2023a. URL https://arxiv.org/abs/2310.05915.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests, 2022. URL https://arxiv.org/abs/2207.10397.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug, 2023b. URL https://arxiv.org/abs/2304.05128.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=KuPixIqPiq.

Sanjiban Choudhury and Paloma Sodhi. Better than your teacher: Llm agents that learn from privileged ai feedback, 2024. URL https://arxiv.org/abs/2410.05434.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

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

Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Taco Cohen, and Gabriel Synnaeve. Rlef: Grounding code llms in execution feedback with reinforcement learning. *arXiv preprint arXiv:2410.02089*, 2024a.

Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Taco Cohen, and Gabriel Synnaeve. Rlef: Grounding code llms in execution feedback with reinforcement learning, 2024b. URL https://arxiv.org/abs/2410.02089.

Xinyu Guan, Li Lyna Zhang, Yifei Liu, Ning Shang, Youran Sun, Yi Zhu, Fan Yang, and Mao Yang. rstar-math: Small llms can master math reasoning with self-evolved deep thinking, 2025. URL https://arxiv.org/abs/2501.04519.

Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. Mapcoder: Multi-agent code generation for competitive problem solving, 2024. URL https://arxiv.org/abs/2405.11403.

Sham Kakade and John Langford. Approximately optimal approximate reinforcement learning. In *Proceedings of the Nineteenth International Conference on Machine Learning*, pp. 267–274, 2002.

Aviral Kumar, Vincent Zhuang, Rishabh Agarwal, Yi Su, John D Co-Reyes, Avi Singh, Kate Baumli, Shariq Iqbal, Colton Bishop, Rebecca Roelofs, Lei M Zhang, Kay McKinney, Disha Shrivastava, Cosmin Paduraru, George Tucker, Doina Precup, Feryal Behbahani, and Aleksandra Faust. Training language models to self-correct via reinforcement learning, 2024a. URL https://arxiv.org/abs/2409.12917.

Aviral Kumar, Vincent Zhuang, Rishabh Agarwal, Yi Su, John D Co-Reyes, Avi Singh, Kate Baumli, Shariq Iqbal, Colton Bishop, Rebecca Roelofs, et al. Training language models to self-correct via reinforcement learning. *arXiv preprint arXiv:2409.12917*, 2024b.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning, 2022. URL https://arxiv.org/abs/2207.01780.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, December 2022. ISSN 1095-9203. doi: 10.1126/science.abq1158. URL http://dx.doi.org/10.1126/science.abq1158.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step, 2023. URL https://arxiv.org/abs/2305.20050.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. Octopack: Instruction tuning code large language models. *arXiv preprint arXiv:2308.07124*, 2023.

Ansong Ni, Miltiadis Allamanis, Arman Cohan, Yinlin Deng, Kensen Shi, Charles Sutton, and Pengcheng Yin. NExt: Teaching large language models to reason about code execution. In *Forty-first International Conference on Machine Learning*, 2024. URL https://openreview.net/forum?id=B1W712hMBi.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback, 2022. URL https://arxiv.org/abs/2203.02155.

Yuxiao Qu, Tianjun Zhang, Naman Garg, and Aviral Kumar. Recursive introspection: Teaching language model agents how to self-improve, 2024. URL https://arxiv.org/abs/2407.18219.

Tal Ridnik, Dedy Kredo, and Itamar Friedman. Code generation with alphacodium: From prompt engineering to flow engineering, 2024. URL https://arxiv.org/abs/2401.08500.

Stephane Ross and J Andrew Bagnell. Reinforcement and imitation learning via interactive no-regret learning. *arXiv preprint arXiv:1406.5979*, 2014.

Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pp. 627–635. JMLR Workshop and Conference Proceedings, 2011.

Amrith Setlur, Chirag Nagpal, Adam Fisch, Xinyang Geng, Jacob Eisenstein, Rishabh Agarwal, Alekh Agarwal, Jonathan Berant, and Aviral Kumar. Rewarding progress: Scaling automated process verifiers for llm reasoning. *arXiv preprint arXiv:2410.08146*, 2024.

Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K. Reddy. Execution-based code generation using deep reinforcement learning, 2023. URL https://arxiv.org/abs/2301.13816.

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.

Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F Christiano. Learning to summarize with human feedback. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 3008–3021. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/1f89885d556929e98d3ef9b86448f951-Paper.pdf.

Wen Sun, Arun Venkatraman, Geoffrey J Gordon, Byron Boots, and J Andrew Bagnell. Deeply aggrevated: Differentiable imitation learning for sequential prediction. In *International conference on machine learning*, pp. 3309–3318. PMLR, 2017.

Gokul Swamy, Sanjiban Choudhury, J Andrew Bagnell, and Steven Wu. Of moments and matching: A game-theoretic framework for closing the imitation gap. In *International Conference on Machine Learning*, pp. 10022–10032. PMLR, 2021.

Peiyi Wang, Lei Li, Zhihong Shao, R. X. Xu, Damai Dai, Yifei Li, Deli Chen, Y. Wu, and Zhifang Sui. Math-shepherd: Verify and reinforce llms step-by-step without human annotations, 2024a. URL https://arxiv.org/abs/2312.08935.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. In *Forty-first International Conference on Machine Learning*, 2024b. URL https://openreview.net/forum?id=jJ9BoXAfFa.

Sean Welleck, Ximing Lu, Peter West, Faeze Brahman, Tianxiao Shen, Daniel Khashabi, and Yejin Choi. Generating sequences by learning to self-correct. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=hH36JeQZDaO.

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.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. Star: Bootstrapping reasoning with reasoning. *Advances in Neural Information Processing Systems*, 35:15476–15488, 2022.

Yuexiang Zhai, Hao Bai, Zipeng Lin, Jiayi Pan, Shengbang Tong, Yifei Zhou, Alane Suhr, Saining Xie, Yann LeCun, Yi Ma, and Sergey Levine. Fine-tuning large vision-language models as decision-making agents via reinforcement learning, 2024. URL https://arxiv.org/abs/2405.10292.

Dan Zhang, Sining Zhoubian, Ziniu Hu, Yisong Yue, Yuxiao Dong, and Jie Tang. Rest-mcts*: Llm self-training via process reward guided tree search, 2024. URL https://arxiv.org/abs/2406.03816.

Wenting Zhao, Nan Jiang, Celine Lee, Justin T Chiu, Claire Cardie, Matthias Gallé, and Alexander M Rush. Commit0: Library generation from scratch. *arXiv preprint arXiv:2412.01769*, 2024.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Sglang: Efficient execution of structured language model programs, 2024. URL https://arxiv.org/abs/2312.07104.

Yifei Zhou, Andrea Zanette, Jiayi Pan, Sergey Levine, and Aviral Kumar. Archer: Training language model agents via hierarchical multi-turn rl. *arXiv preprint arXiv:2402.19446*, 2024.

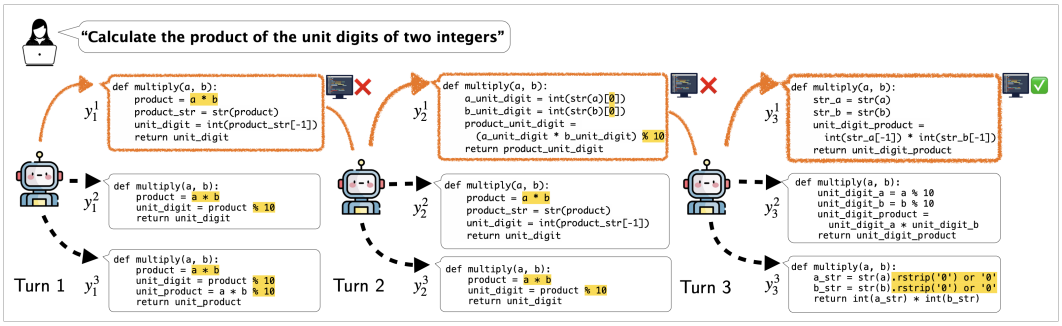

Figure 4: A qualitative example of multi-turn BoN search using dense rewards obtained via the learned verifier in μCODE. Here, we show the top 3 ranked solutions at each turn t where $R_\phi(x, y_t^i) \geq R_\phi(x, y_t^j)$ for $i < j$. We observe that the learned verifier selects the better solution (in orange) at each turn. The selected solution is passed to public tests to retrieve execution feedback for the generator to improve the next code solution. The selected solution at each turn is better than the last (less errors highlighted in yellow), with the final solution passing all tests. Note that there are 2 correct solutions at the final turn.

A APPENDIX

IMPACT STATEMENT

The proposed method for training code agents has the potential to streamline software development processes by automating routine coding tasks, thereby reducing human labor and accelerating production timelines. However, these advances will also introduce bugs, which can propagate at scale if no proper quality control is in place.

A.1 RELATED WORK

Prompting To Solve Multi Step Tasks A common framework for tackling multi-step tasks with LLMs is prompting-based agentic systems. Self-Debugging (Chen et al., 2023b) asks the LLM to iteratively improve code by providing execution feedback while CodeT (Chen et al., 2022) asks the LLM to generate test cases. AlphaCodium (Ridnik et al., 2024) first reflects on input instructions, generates and filters from multiple code generations, and finally iterates on public and self-generated test cases. MapCoder (Islam et al., 2024) incorporates four agents to generate example problems, plans and code, and then perform debugging. However, prompting-based agents yield limited improvements.

Training LLMs for Multi Step Tasks Some work has explored explicitly training critics or reward models for multi-step reasoning tasks. In the coding domain, CodeRL (Le et al., 2022) trains a token-level critic to aid in code generation and to perform inference-time search. CodeRL's mechanics are similar to our method, but their generator is not trained for multi-step: CodeRL trains a "code repairer" which conditions on one erroneous code completion while our generator incorporates multiple. ARCHER (Zhou et al., 2024), which frames multi-step tasks via a two-level hierarchical MDP, where the higher level MDP considers completions as actions and the lower level MDP considers tokens as actions. Another line of work utilizes Monte Carlo Tree Search (MCTS) methods for training: rStar-Math (Guan et al., 2025) trains a policy preference model to boost small LMs' math abilities to match or exceed large reasoning-based LMs and ReST-MCTS (Zhang et al., 2024) trains a process reward model (PRM) similarly to Math-Shepherd (Wang et al., 2024a). Although μCODE's BoN search resembles a tree search, our key insight that multi-step code generation resembles a one-step recoverable MDP allows us to collect training trajectories much more efficiently. Finally, some work has explored using verifiers only during inference time. In "Let's Verify Step by Step" (Lightman et al., 2023), the authors demonstrate that PRMs trained on erroneous math solutions annotated by humans outperform outcome reward models for filtering

multiple inference time generations. Meanwhile, AlphaCode (Li et al., 2022) trains a test generator to evaluate multiple code solutions.

Other works omit learning a critic or reward model altogether. In the coding domain, RLEF (Gehring et al., 2024b) derives rewards only on the executor's result on test cases and syntax checkers, and PPOCoder (Shojaee et al., 2023) additionally considers semantic and syntactic alignment, generated via data flow graphs and abstract syntax trees respectively, with a reference solution. The "oracle" rewards in these methods may not be informative for training, and in the case of PPOCoder, require complex constructs. We empirically show that having a reward model is beneficial by comparing μCODE against the Multi-STaR baseline. Meanwhile, SCoRe (Kumar et al., 2024a) splits training into a "generator" and "correction" phase, thus restricting the total number of turns to 2. RISE (Qu et al., 2024) generates recovery steps via a more powerful LLM or by selecting a sampled completion via the oracle rewards. Both methods are less efficient than μCODE, which doesn't require generating corrections beyond generating training trajectories. Finally, FireAct (Chen et al., 2023a) and LEAP (Choudhury & Sodhi, 2024) FT ReAct style agents while RL4VLM (Zhai et al., 2024) and GLAM (Carta et al., 2024) studies training LLMs with interactive environment feedback.

A.2 TEST-TIME SCALING

In the multi-turn setting, the number of candidate solutions can rise exponentially with the number of turns. To avoid this, μCODE uses the learned verifier during inference to select the most promising candidate among N candidates at each turn, leading to a linearly increasing number of calls to the generator. We study the inference-time scaling behaviors of μCODE where we scale the number of candidate generations N at each turn. Figure 5 plots the BoN with different values of N ($1 \leq N \leq 11$). The performance gains diminishes for larger N on both datasets. On the MBPP dataset, the performance gains diminish with $N \geq 5$, whereas on HumanEval dataset a dip is observed for $N > 7$.

In this section, we further study the importance of training the verifier with on-policy rollouts from the generator. We present a comparison of a verifier trained with samples from the Llama-3.2-1B-Instruct model (Base Verifier) and a verifier learned with samples from the generator of μCODE. Note that we use the generator of μCODE to obtain candidate solutions at each turn during evaluation. In Figure 5, we also present the scaling behaviors of different learned verifiers. We observe that using a verifier trained with on-policy samples obtained via the generator of μCODE performs better and showed significant improvements of up to **4.3**% for different values of candidate solutions N.

Figure 4 presents a qualitative example of multi-turn Best-of-N search with μCODE. Through this example, we demonstrate the advantages of dense scores from the learned verifier at facilitating efficient search across turns. We generate $N = 5$ code solutions at each turn and show the top 3 ranked solutions using the dense scores. At the first turn, we observe that the last solution y_1^3 is less accurate than the other 2 solutions y_1^1 and y_1^2. The top ranked solution is used to collect the environment feedback, upon which the generator comes up with N new candidate solutions. Upon the top 3 solutions, the last two snippets are similar to the candidates from the previous turn. However, the top ranked solution is a novel solution and is more accurate as the generated code learns to extract a single digit and multiply it. With the execution feedback, μCODE generates 2 correct responses– y_3^1 and y_3^2 and learned verifier chooses one of them compared to the incorrect response y_3^3.

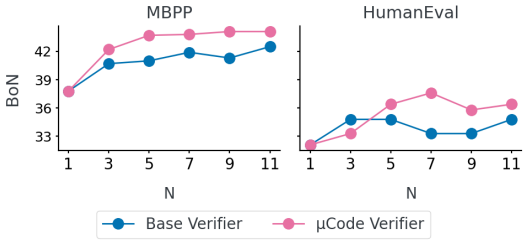

Figure 5: Test-time scaling with different values of candidate solutions N at each turn and different ways of learning verifiers. We compare with verifiers learned on samples from μCODE and base policy. The candidate solutions are obtained from the 1B generator of μCODE at each turn. We observe that the BoN performance improves with larger values of N on both datasets. The verifier learned with on-policy samples perform better.

A.3 LOSS FUNCTION FOR VERIFIER

As described in 3.2, we compare against different loss functions for training the verifier. For this experiment, we first generate multiple single step rollouts and label them via oracle verifier. Given oracle labels, we train verifiers with two loss functions – BCE and BT. During inference, the learned verifier picks the best ranked solution among the N solutions provided by the generator. Similar to (Cobbe et al., 2021), we report the BoN plot with different values of N obtained by first sampling N candidate solutions, choosing the top-ranked solution using the learned verifier, and then evaluating the solution against public and private tests. We calculate this metric over multiple samples for each value of N. In Figure 6, we observe that the verifier trained with BT loss consistently outperforms the verifier trained on BCE loss on both MBPP and HumanEval.

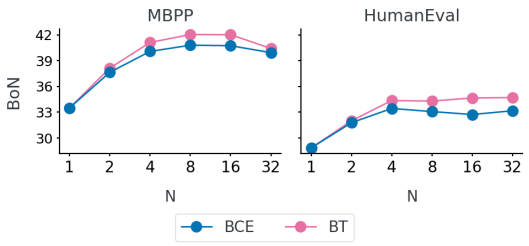

Figure 6: Comparison between BCE and BT loss function for training the verifier. We train the verifiers on samples generated by the base model (Llama-3.2-1B-Instruct). The learned verifier then ranks the candidate solutions from base model and the BoN performance of selected solution is reported. The verifier trained with BT loss performs better increasing value of N.

A.4 Proof of Theorem 3.2

The proof relies on two important results.

The first is the Performance Difference Lemma (PDL) (Kakade & Langford, 2002) which states that the performance difference between any two policies can be expressed as the sum of advantages.

$$J(\pi) - J(\pi') = \sum_{t=1}^{T} \mathbb{E}_{s_t \sim d_t^{\pi}} \left[\sum_{a_t} A^{\pi'}(s_t, a_t) \pi(a_t|s_t) \right] \tag{8}$$

where $s_t \sim d_t^{\pi}$ is the induced state distribution by π, and $A^{\pi'}(s_t, a_t) = Q^{\pi'}(s_t, a_t) - V^{\pi'}(s_t)$ is the advantage w.r.t. π'.

We apply the PDL between the expert π^* and the learner π

$$J(\pi^\star) - J(\pi) = \sum_{t=1}^{T} \mathbb{E}_{s_t \sim d_t^{\pi}} \left[\sum_{a_t} A^{\star}(s_t, a_t) \left(\pi^\star(a_t|s_t) - \pi(a_t|s_t) \right) \right] \tag{9}$$

where the result follows from $\left(\sum_{a_t} A^\star(s_t, a_t) \pi^\star(a_t|s_t) = 0 \right)$

According to the one-step recoverable MDP definition, $A^\star(s, a) \leq 1$ for all (s, a). Hence we can bound the performance difference as

$$\begin{aligned}
J(\pi^\star) - J(\pi) &= \sum_{t=1}^{T} \mathbb{E}_{s_t \sim d_t^{\pi}} \left[\sum_{a_t} A^{\star}(s_t, a_t) \left(\pi^\star(a|s_t) - \pi(a|s_t) \right) \right] \\
&\leq ||A^\star(.,.)||_\infty \sum_{t=1}^{T} \mathbb{E}_{s_t \sim d_t^{\pi}} ||\pi(.|h_t) - \pi^\star(.|s_t)||_1 && \text{(Holder's Inequality)} \\
&\leq \sum_{t=1}^{T} \mathbb{E}_{s_t \sim d_t^{\pi}} ||\pi(.|s_t) - \pi^\star(.|s_t)||_1 && \text{(One step recoverability)}
\end{aligned}$$

The second result we use us from interactive imitation learning DAGGER (Ross et al., 2011) that reduces imitation learning to no-regret online learning. DAGGER shows that with $\pi^\star$ as the expert teacher guarantees that after N iterations, it will find at least one policy

$$\mathbb{E}_{s \sim d^{\pi}} ||\pi(.|s) - \pi^\star(.|s)||_1 \leq \mathbb{E}_{s \sim d^{\pi}} ||\pi_{\text{class}}(.|s) - \pi^\star(.|s)||_1 + \gamma(N) \tag{10}$$

where $\gamma(N)$ is the average regret, and d^π is the time average distribution of states induced by policy π, π_{class} is the best policy in policy class.

Plugging this in we have

$$\begin{aligned} J(\pi^\star) - J(\pi) &\leq \sum_{t=1}^{T} \mathbb{E}_{s_t \sim d_t^\pi} ||\pi(.|s_t) - \pi^\star(.|s_t)||_1 \\ &\leq \sum_{t=1}^{T} \mathbb{E}_{s_t \sim d_t^\pi} ||\pi_{\text{class}}(.|s_t) - \pi^\star(.|s_t)||_1 + \gamma(N) \quad \text{From (10)} \\ &\leq T(\epsilon + \gamma(N)) \end{aligned}$$

A.5 Hyperparameters

Model	Generator	Verifier
Training Epochs	2	2
Learning Rate	5×10^{-7}	1×10^{-6}
Batch Size	32	64
Max seq length	8192	2048

Table 3: Hyperparameters for SFT and RM training.

A.5.1 Training Parameters

Table 3 contains hyperparameters for training the generator and reward model on both models (Llama-3.1-8B-Instruct and Llama-3.2-1B-Instruct) and datasets (MBPP and HumanEval). We perform 2 iterations of training with μCODE, starting from the base model each iteration. All training runs were on machines with either 4 RTX 6000 Ada Generation GPUs for 1B models with 48 GB of memory per GPU or 4 H100 GPUs for 8B models with 80 GB of memory per GPU.

A.5.2 Inference Parameters

We use SGLang (Zheng et al., 2024) to serve our models for inference. Greedy experiments use temperature 0 with flags *–disable-radix-cache –max-running-request 1* to ensure deterministic results while BoN search experiments use a temperature of 0.7. All experiments are capped to 1000 tokens per completion per turn.

A.6 Prompts

A.6.1 Single Step Prompt

Immediately below is the prompt template to generate 1 code completion in a single-step method or to generate the 1st step in a multi-step method. Below the prompt templates are examples of the code prompt and public tests for HumanEval and MBPP.

Single Step Prompt

```
Write a Python function implementation for the following prompt:

\{prompt\}

Your code should satisfy these tests:

\{test\}

Return only the implementation code, no tests or explanations. Be
sure to include the relevant import statements:
```python
code
```
```

HumanEval Prompt

```
from typing import List


def has_close_elements(numbers: List[float], threshold: float) ->
bool:
    """ Check if in given list of numbers, are any two numbers
closer to each other than
    given threshold.
    >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
    True
    """
```

HumanEval Test

```
def check(has_close_elements):
    assert has_close_elements([1.0, 2.0, 3.0], 0.5) == False
    assert has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
== True
check(has_close_elements)
```

MBPP Prompt

```
Write a function to find the minimum cost path to reach (m, n) from
 (0, 0) for the given cost matrix cost[][] and a position (m, n) in
 cost[][].
```

MBPP Test

```
assert min_cost([[1, 2, 3], [4, 8, 2], [1, 5, 3]], 2, 2) == 8
assert min_cost([[2, 3, 4], [5, 9, 3], [2, 6, 4]], 2, 2) == 12
assert min_cost([[3, 4, 5], [6, 10, 4], [3, 7, 5]], 2, 2) == 16
```

A.6.2 Feedback Prompt

Immediately below is the prompt template for how we provide feedback in multi-step methods. The feedback only consists of executor error traces, and we provide an example from HumanEval.

Multi-Step Feedback Prompt

```
Feedback:

\{feedback\}
```

HumanEval Multi-Step Feedback Prompt

```
Traceback (most recent call last):
  File "test.py", line 18, in <module>
    assert has_close_elements([1.0, 2.0, 3.0], 0.5) == False
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError
```

A.7 Public Private Tests

We choose a public-private test split for HumanEval and MBPP to ensure that naively passing the public tests does not guarantee private test success. For HumanEval, we use a single test from the code prompt's docstring as the public test and the remaining tests along with the official test suite as private tests. For ease of parsing, we utilize a processed version of HumanEval, HumanEvalPack (Muennighoff et al., 2023). For MBPP, we use a single test from the official test suite as the public test, and the remaining tests and any "challenge test list" tests as private tests.