
# Multi-Turn Code Generation Through Single-Step Rewards

Arnav Kumar Jain , Gonzalo Gonzalez-Pumariega , Wayne Chen , Alexander M Rush ,Wenting Zhao , Sanjiban Choudhury

Mila – Quebec AI Institute , Université de Montréal , Cornell University

*Presented by Muhan Yuan*

ICML '25

# 作者团队

**研究方向**

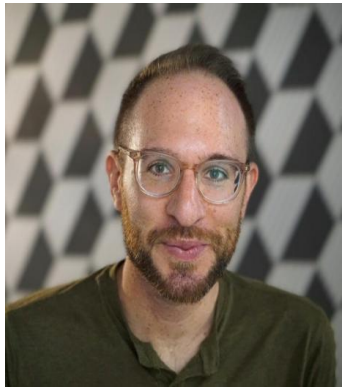
- 强化学习
- 多轮交互、验证器
- 学习搜索
- 代码生成智能体
- 规划与决策中的大模型

**Arnav Kumar Jain**

蒙特利尔大学博士生

**Gonzalez-Pumariega**

康奈尔大学博士生



**Alexander Rush**

康奈尔大学教授


1. [ICLR’25] Arnav Kumar Jain et al. “Non-Adversarial Inverse Reinforcement Learning via Successor Feature Matching”
2. [NeurIPS’25] Arnav Kumar Jain et al. “A Smooth Sea Never Made a Skilled SAILOR: Robust Imitation via Learning to Search”
3. [ICML’24] Arnav Kumar Jain et al. “Revisiting Successor Features for Inverse Reinforcement Learning”

# Outline

1 Background

2 Related work

3 Design

4 Experiments

5 Conclusion

# LLM Code Generation

**Code LLMs for Programming**

- 近几年Code LLM迅速发展：

  Copilot、Claude code 等已经能直接生成可运行代码

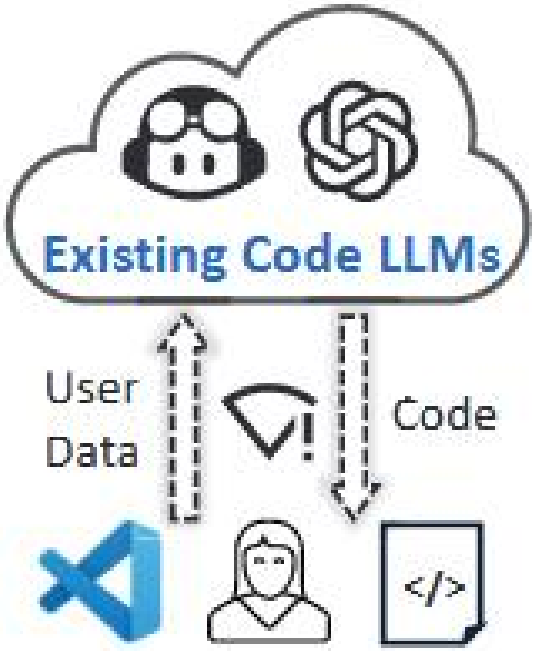
- 典型 task：

  自然语言题目 → 一段完整代码

- 常见评测方式：

  用一组隐藏测试用例（如 HumanEval、MBPP、CodeContests）

  指标 pass@k（至少有 k 次采样通过所有测试）

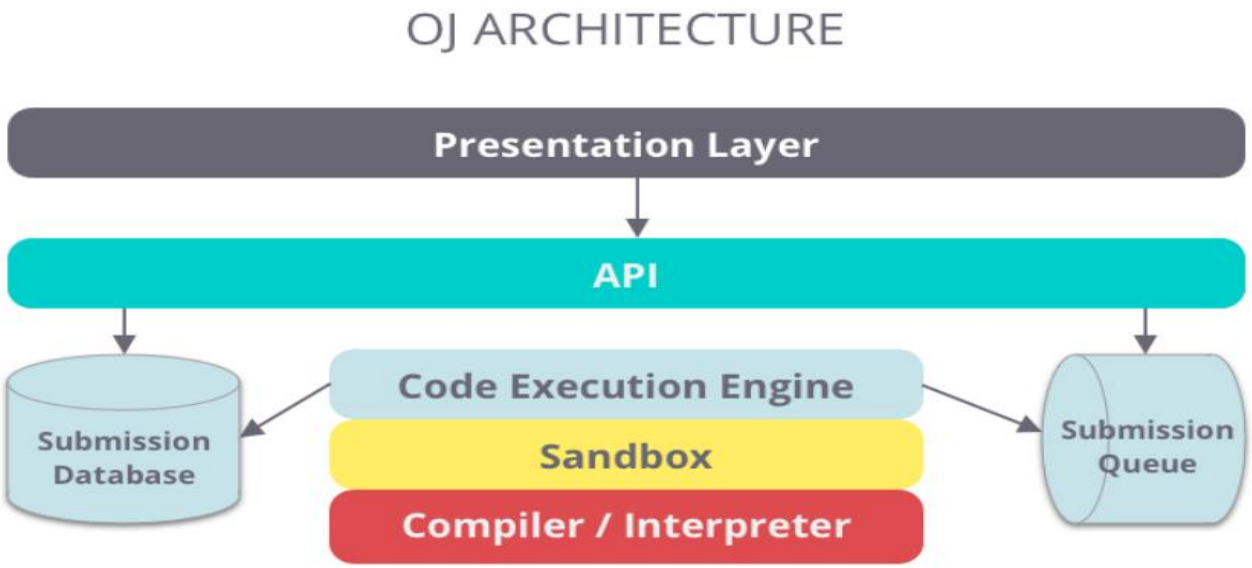



# Execution Environment

- 每个题目带有一组测试用例：
  - Public tests：开发者可见，用来本地调试
  - Private tests：不可见，只用来打最终分数
- 候选代码都能在 public tests 上执行得到：
  - 通过 / 失败情况
  - 报错类型、失败输入等信息

**执行反馈 = 很有价值的信号**

- 人类写代码时一定会用（写完就点 Run）
- 但很多 Code LLM 只在最后评测时用一次，不参与生成过程中的推理



# Motivation & Goal

**Question?**

- 在可以多次运行代码、获得执行反馈的环境中，**如何比一次性生成做得更好**?

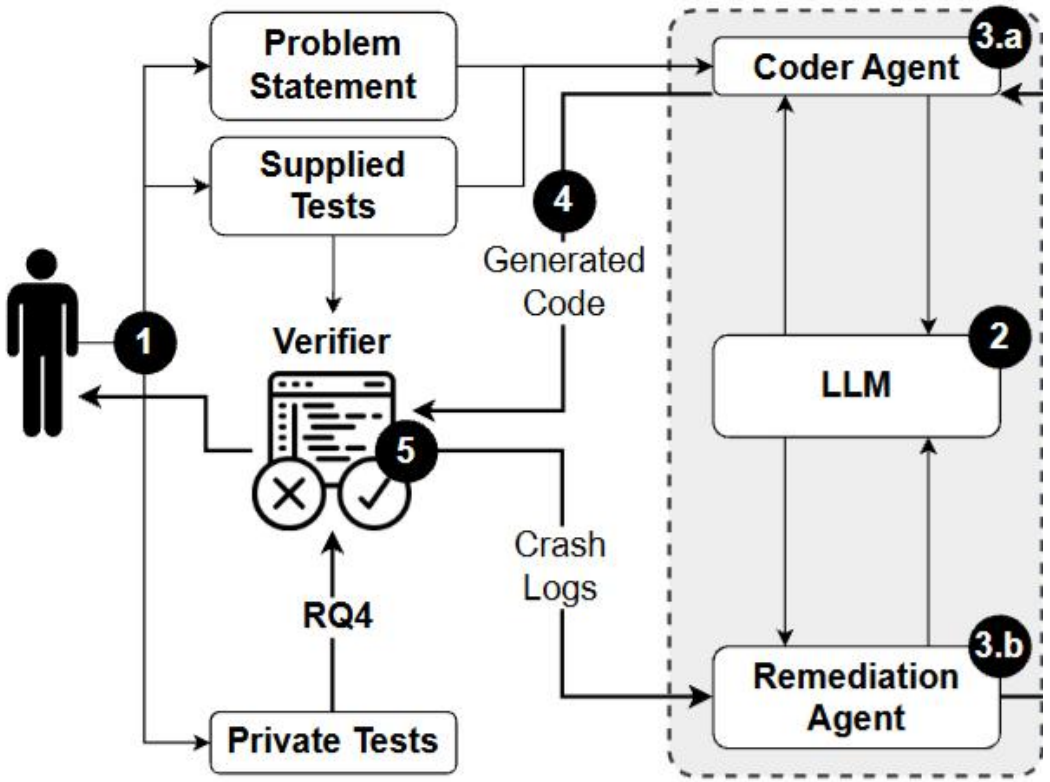
**Why?**

- 很多现有方法要么基本不利用多轮执行反馈，要么依赖复杂、多步 RL
- 奖励稀疏 → 训练不稳定、成本高，不利于在大模型上大规模部署

**Our goals:**

✓ 利用简单的单步奖励（是否通过测试），提升多轮代码生成效果

✓ 设计一个稳定、易训练、又能充分用好执行反馈的框架（μCODE）



# Outline

1 Background

2 Related work

3 Design

4 Experiments

5 Conclusion

# Single-Turn Code Generation

**单轮代码生成**

- 输入：函数签名 + 题目描述
  输出：模型一次性生成完整函数体，不根据结果修改

- 典型框架：HumanEval / MBPP 等基准
  – 每题自带单元测试，用 pass@k 评价：
    生成 k 个候选，只要有 1 个通过全部测试就算成功

- 代表模型：Codex、Code Llama、DeepSeek-Coder 等
  – 重点在预训练数据、模型规模、SFT/RLFT；
    评测依旧是单轮生成 + offline 测试

局限：测试环境只做离线打分，不参与生成，没有交互式反馈

```python
def incr_list(l: list):
    """Return list with elements incremented by 1.
    >>> incr_list([1, 2, 3])
    [2, 3, 4]
    >>> incr_list([5, 3, 5, 2, 3, 3, 9, 0, 123])
    [6, 4, 6, 3, 4, 4, 10, 1, 124]
    """
    return [i + 1 for i in l]
```

```python
def solution(lst):
    """Given a non-empty list of integers, return the sum of all of the odd elements
    that are in even positions.

    Examples
    solution([5, 8, 7, 1]) ==>12
    solution([3, 3, 3, 3, 3]) ==>9
    solution([30, 13, 24, 321]) ==>0
    """
    return sum(lst[i] for i in range(0,len(lst)) if i % 2 == 0 and lst[i] % 2 == 1)
```

```python
def encode_cyclic(s: str):
    """
    returns encoded string by cycling groups of three characters.
    """
    # split string to groups. Each of length 3.
    groups = [s[(3 * i):min((3 * i + 3), len(s))] for i in range((len(s) + 2) // 3)]
    # cycle elements in each group. Unless group has fewer elements than 3.
    groups = [(group[1:] + group[0]) if len(group) == 3 else group for group in groups]
    return "".join(groups)

def decode_cyclic(s: str):
    """
    takes as input string encoded with encode_cyclic function. Returns decoded string.
    """
    # split string to groups. Each of length 3.
    groups = [s[(3 * i):min((3 * i + 3), len(s))] for i in range((len(s) + 2) // 3)]
    # cycle elements in each group.
    groups = [(group[-1] + group[:-1]) if len(group) == 3 else group for group in groups]
    return "".join(groups)
```

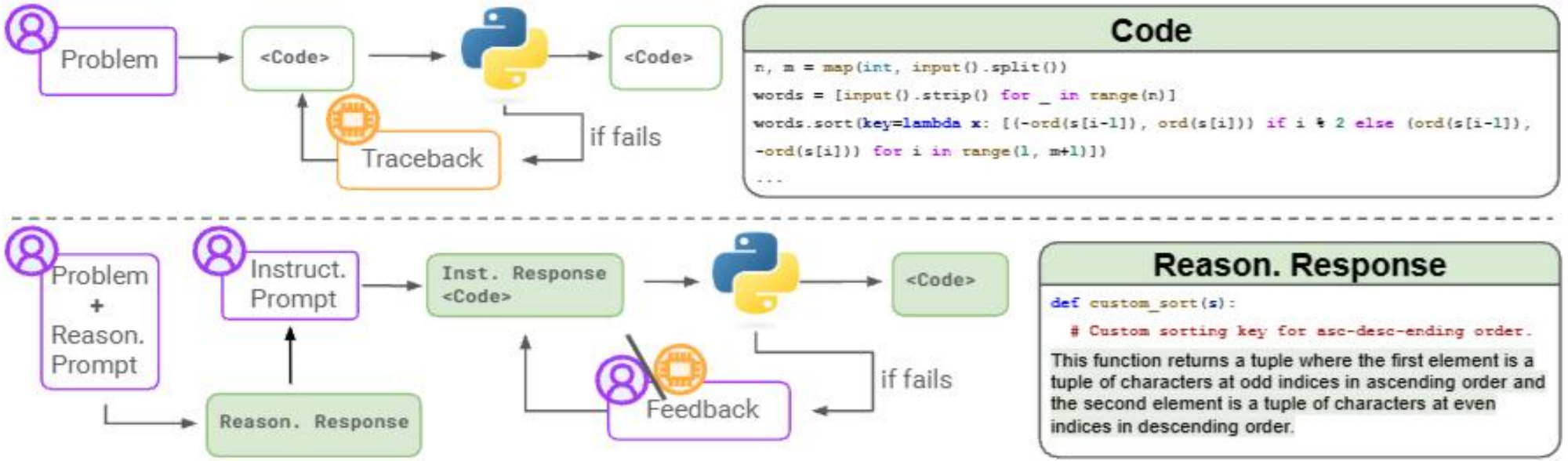
# Multi-Turn Code Generation

- **环境：题目 + 测试 / 执行器**
  模型可以多次生成代码，每一轮根据执行反馈（通过情况 / Traceback）继续尝试

- **典型流程：**
  Problem → 生成 Code → 运行 public tests → 得到失败用例 / Traceback → 生成下一版 Code

- **代表方法：**
  – 交互式 RL 环境：InterCode 等
  – 自修复 agent：ReAct、Self-Refine、CodeChain 等

- 局限：
  – RL 路线：多步奖励、critic / PRM，成本高且不稳定
  – Agent 路线：多靠启发式，缺乏简单统一的建模方式



# Where Does µCODE Fit?

| 方法类别 | 是否多轮 | 如何利用执行反馈 | 训练复杂度/成本 |
| --- | --- | --- | --- |
| 单轮代码生成（Single-turn LLM） | ✗ 一次生成 | 仅用于离线评测（pass@k），不参与生成过程 | 低：SFT / RLFT 为主 |
| 多轮 RL 方法（Multi-turn RL） | ✓ | 把执行结果作为多步奖励，优化整条轨迹（MDP / RL 环境） | 高：需要 critic / PRM，大量 rollouts，训练不稳定 |
| 多轮 Agent / 自修复方法 | ✓ | 根据报错 / Traceback 设计提示词和规则，自我反思、打补丁 | 中：主要是 prompt & 规则工程，缺乏统一建模 |
| **µCODE（本文）** | ✓ | **使用 public tests 的单步奖励 + 学习式验证器，指导每一轮生成** | **中：基于模仿学习 + expert iteration，避免复杂多步 RL** |

# Outline

1 Background

2 Related work

3 Design

4 Experiments

5 Conclusion

# Design Overview

**设计目标：**
• 在多轮代码生成 + 可执行测试的环境下，
充分利用执行反馈，但避免复杂多步 RL

**核心组件：**
• Code Generator πθ：根据当前状态 s 生成下一轮代码 y
• Verifier Rφ：根据题目、代码和执行结果给出单步评分

**训练流程**

1. Rollout generator πθ，在环境中多轮交互，收集数据 D

⇓

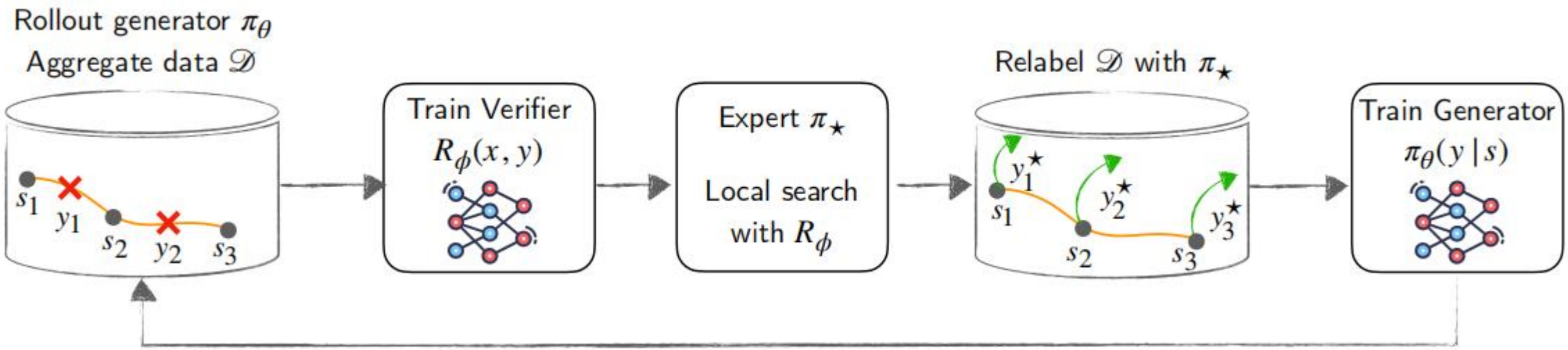
2. 用 D 训练 Verifier Rφ(x, y)

⇓

3. 用 Rφ 做 local search，得到 expert 策略 π⋆，重新标注 D

⇓

4. 用标注后的 D 做模仿学习，更新 generator πθ(y | s)



# Problem Setup

**任务与数据：**

- 每个题目 x 包含：
  – 问题描述（prompt）
  – 一组 public tests（可见，用于交互）
  – 一组 private tests（不可见，只用于最终打分）

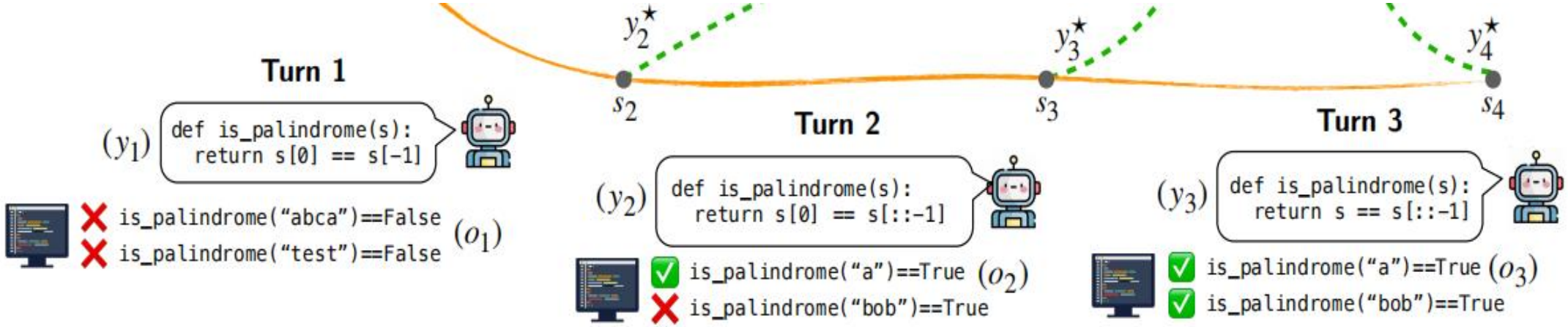
**优化目标：**

- 策略 π 的目标：

在最多 T 轮内，使某一轮代码通过全部 private tests 的概率最大，即P(∃ t ≤ T, $y_t$ 通过全部 private tests | x, π)

**多轮交互流程：**

- 在第 t 轮，模型看到：
  – 题目 x
  – 历史代码与执行结果 $h_t = \{(y_1, o_1), \ldots, (y_{t-1}, o_{t-1})\}$
- Code Generator 生成当前轮代码 $y_t$
- 环境在 public tests 上执行 $y_t$，得到反馈 $o_t$
- 将 $(y_t, o_t)$ 加入历史，进入下一轮（最多 T 轮）



# One-step recoverable MDP

**One-step recoverable：**

- 一步可恢复：对于任意中间状态 $s_t$，总存在某个动作 $a_\star$，可以让下一轮直接解出题目
- 在代码生成里，就是不管前面写了多少错误代码，只要下一轮输出正确解，环境立刻结束并判定成功

**图中示例：**

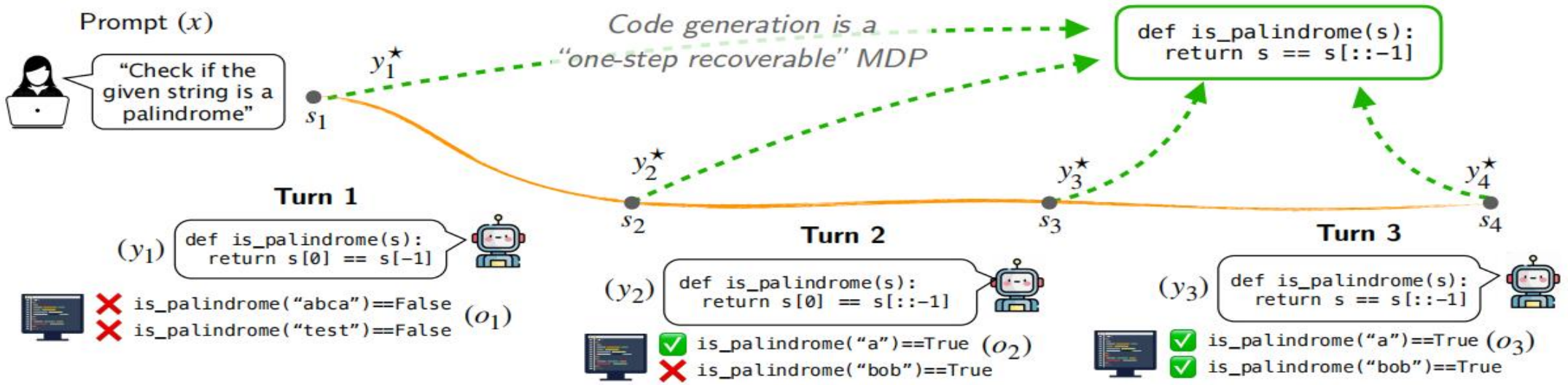
- 橙色轨迹：模型实际走的 3 轮代码 $y_1, y_2, y_3$
- 绿色虚线：从任意状态 $s_1, s_2, s_3, s_4$ 都有一条一步跳到正确代码 $y_\star$ = return s == s[::-1] 的路径

结论：

- 在这种一步可恢复的 MDP 中，最大化多轮内最终成功的目标，可以简化为在每个状态 s 选择单轮成功概率最大的动作：

  $\pi^*(s) = \text{argmax}_a P(\text{下一轮成功} \mid s, a)$

- 可以用 Verifier 来近似这个单步成功概率，而不是做复杂的多步 RL



# Single-Step Rewards from Execution Feedback

**训练阶段可见的信号：**

- 每个 (题目 x, 代码 y) 都可以在本地执行：
  public tests + private tests → 得到真·0/1 标签 r*(x, y)

**直接用 0/1 奖励的问题：**

- 奖励极度稀疏，大部分候选代码 r* = 0，学习信号很弱
- 推理阶段不访问 private tests，r* 在 test-time 用不上

**我们想要的单步评分：**

- 只依赖：题目 + 当前代码 + public tests 结果
- 输出：一个“这一步有多可能成功”的平滑分数
  (而不是硬性的 0/1)

→ 这就是接下来要介绍的 Verifier。

结论：

- 训练时虽有真·0/1 标签 r*(x, y)，
  但多数候选都是 0 → 奖励极度稀疏
- 推理阶段不能访问 private tests，
  r* 在 test-time 完全用不上

| 代码 | public tests | private tests | r*(x,y) |
|---|---|---|---|
| $y_1$ | 通过部分用例 | 隐藏用例失败 | 0 |
| $y_2$ | 大部分用例失败 | 隐藏用例失败 | 0 |
| $y_3$ | 通过全部用例 | 隐藏用例通过 | 1 |

# Verifier: Learning Single-Step Success Probability

**Verifier 做什么?**

• 输入：题目 x、当前代码 y、public tests 的执行结果
• 输出：分数 Rφ(x, y) ∈ [0, 1]，表示“这一版代码有多可能最终成功”

Key points：

• 用 r*(x, y) 监督学一个平滑单步分数

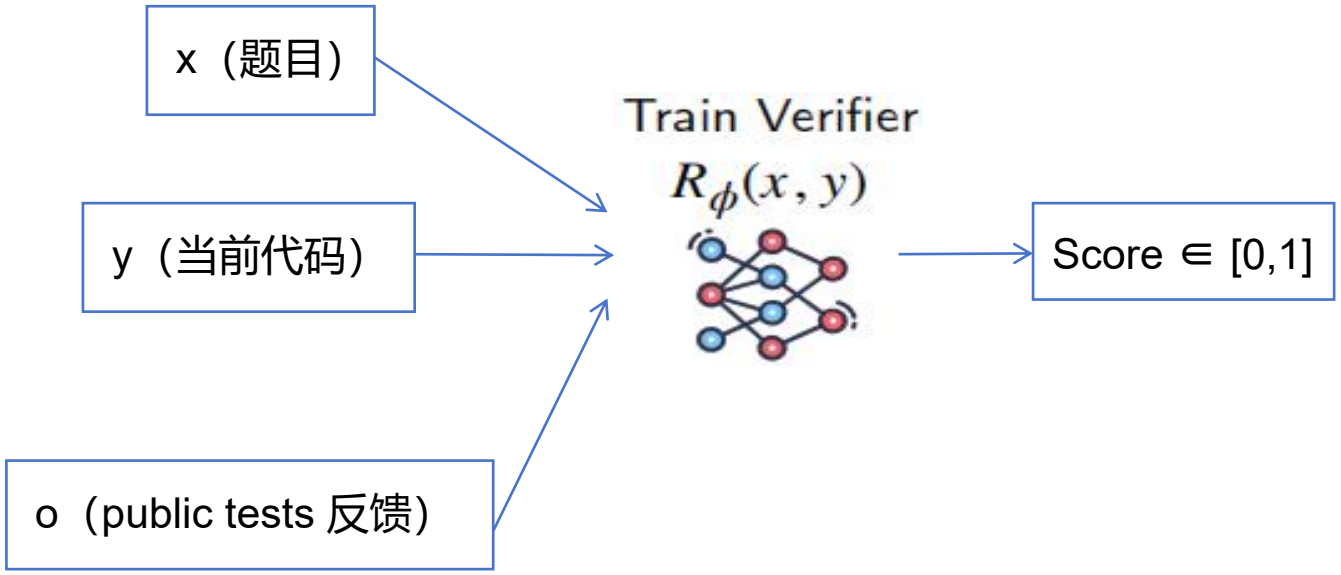
• 推理时只需 public tests 就能给代码打分

**如何训练?**

• 训练阶段有真·0/1 标签 r*(x, y)
用监督学习让 Rφ(x, y) ≈ r*(x, y)

**推理 / 使用方式?**

• 推理时只能看到 public tests，
先执行 y，再用 Verifier 给候选代码打分
→ 后面用来训练/重排序多轮代码



# Expert Policy & Relabeling D

**Expert 策略 $\pi_\star$ 做什么?**

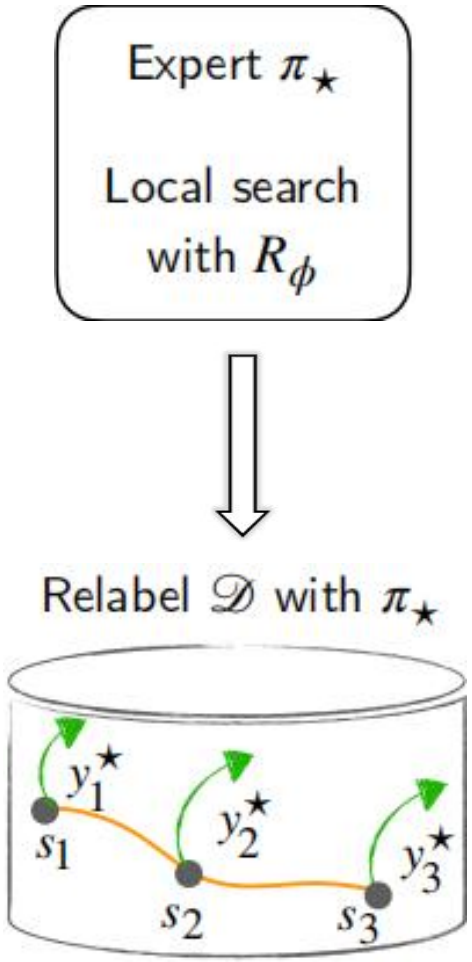
- 对于 rollouts 中的每个状态 s（题目 + 历史）：
  - 以当前 generator $\pi_\theta$ 为起点，采样/搜索一批候选代码
  - 用 Verifier Rφ 给候选打分，做局部搜索，选出得分最高的 $y_\star$

- 得到的 $\pi_\star(\cdot \mid s)$ 可以看作“局部最优”的专家策略

**如何重标数据集 D?**

- 把原来的 (s, y) 用专家动作 $y_\star$ 替换：
  - 得到一批 $(s, y_\star)$ 对，表示在这个状态下“更好的动作”

- 得到带专家标注的 D′：
  - 下一步用来模仿学习，训练新的 generator $\pi_\theta$



# Training the Generator on Expert Data

**训练数据：**

- 经过重标得到的专家数据集 $D' = \{(s, y_\star)\}$

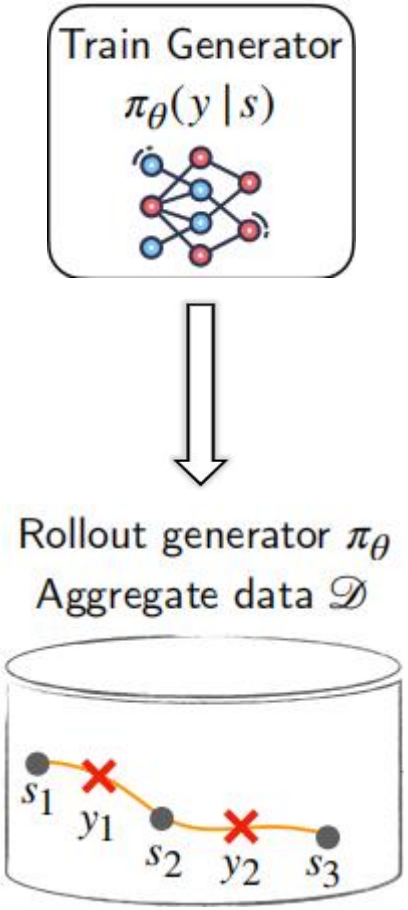
**训练目标（模仿学习 / 最大似然）：**

- 在 $D'$ 上最小化交叉熵：
  $L(\theta) = - E_{(s, y_\star)\in D'} [ \log \pi_\theta(y_\star \mid s) ]$

- 相当于让 generator $\pi_\theta$ 尽量给专家动作 $y_\star$ 高概率

**直观理解：**

- Verifier + $\pi_\star$ 提供高质量多轮轨迹
- generator $\pi_\theta$ 在这些轨迹上做 SFT / 行为克隆
  → 得到一个更擅长利用执行反馈的多轮代码生成器



# Inference: Multi-Turn Best-of-N Search

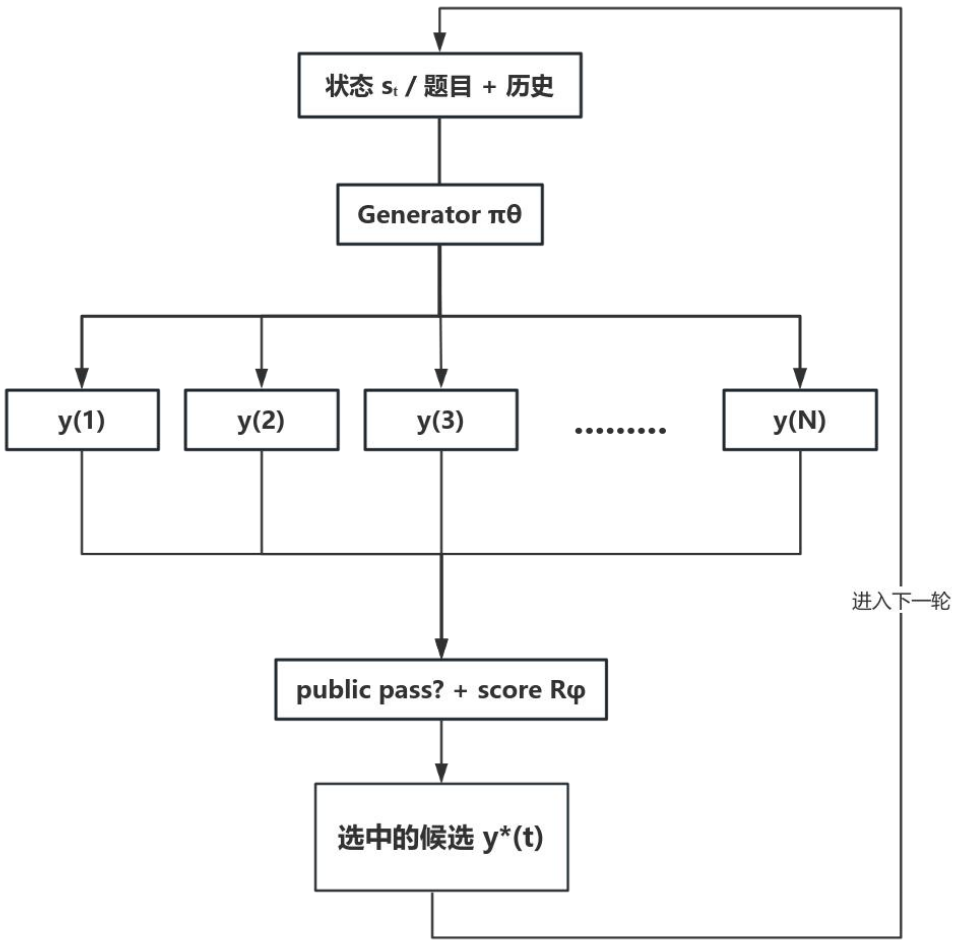
- **设置**
  - 已训练好的 Generator $\pi_\theta$、Verifier $R\varphi$
  - 每个题目最多 T 轮交互，每轮采样 N 个候选（例如 T=3, N=8）

- **第 t 轮流程**
  - 在当前状态 $s_t$ 上，$\pi_\theta$ 生成 N 段候选代码 $y_t^{(1\ldots N)}$
  - 对每个候选在 public tests 上执行，得到反馈 $o_t^{(1\ldots N)}$
  - Verifier 计算分数 $R\varphi(x, y_t^{(i)}, o_t^{(i)})$
  - 按选择规则挑出一个 $y^*_t$ 作为本轮提交，
    用它的反馈更新到下一状态 $s_{\{t+1\}}$

- **选择规则**
  - 若存在“通过全部 public tests”的候选，在其中选 $R\varphi$ 最高的
  - 否则在所有候选里选 $R\varphi$ 最高的一个

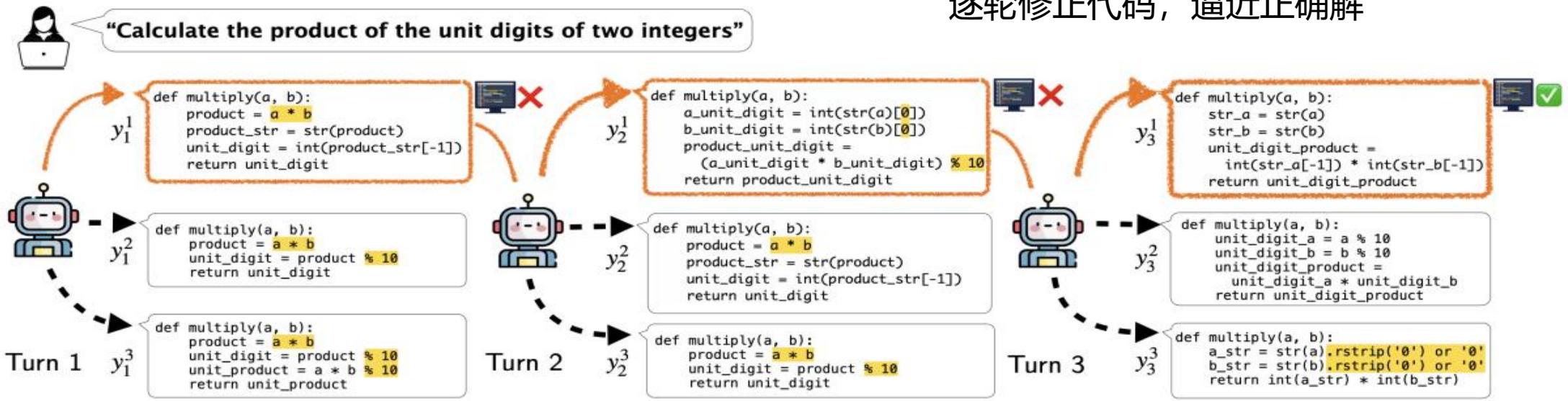
- **停止条件**
  - 某一轮有候选通过 private tests → 提前结束，记为成功
  - 否则最多运行到 t = T，用最后一轮的 best 解来评估

每轮：Best-of-N → 选 1 条路径继续



# Inference: Qualitative Example of Multi-Turn BoN

- **场景：**
  - Calculate the product of the unit digits of two integers
  - 每一列是一轮交互（Turn 1 / Turn 2 / Turn 3）
  - 每一列里有多段候选代码 $y_t^1$, $y_t^2$, $y_t^3$ …
- **Verifier 的打分：**
  - 黄色高亮：当前版本代码中的错误位置
  - 每个候选对应一个分数，分数越高这一步更可能最终成功
- **多轮 Best-of-N 过程：**
  - Turn 1：选中的候选仍然有多处错误
  - Turn 2：在 Verifier 引导下，错误减少，只剩下最后一步没改对
  - Turn 3：最终路径收敛到完全正确的实现
- **Takeaway：**
  - Verifier 提供的是“错误多少”的细粒度信息，而不仅仅是 pass / fail
  - 多轮 Best-of-N 可以在 dense 分数的帮助下，逐轮修正代码，逼近正确解



# Design Summary

- **Key ideas**

✓ 把多轮代码生成看作 one-step recoverable MDP，
在每个状态里选“单轮成功概率”最大的代码

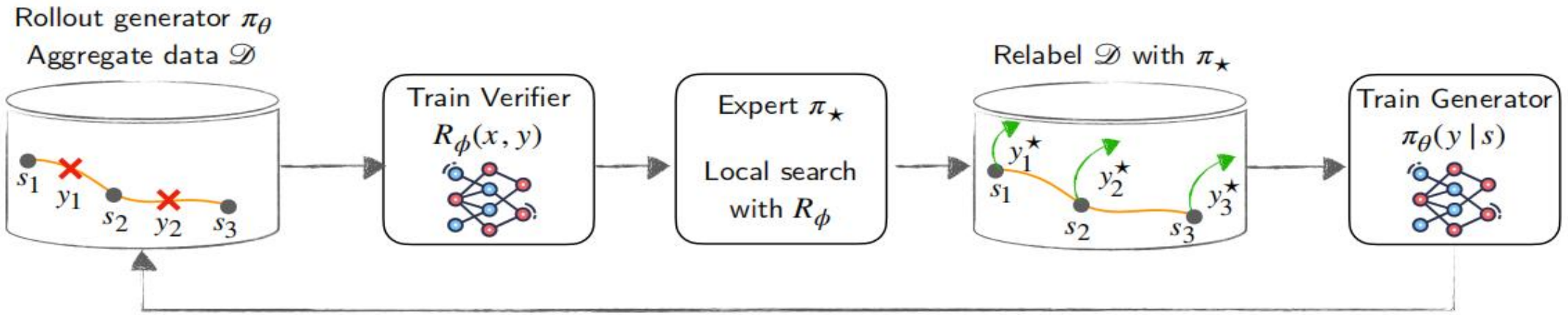
✓ 引入 Generator + Learned Verifier，
用 Verifier 把执行反馈变成 dense reward

✓ 用 expert iteration + multi-turn Best-of-N，
避免复杂多步 RL，又能充分利用执行反馈

- **Training loop**

1. Rollout generator $\pi_\theta$，收集多轮轨迹数据 D
2. 用 private tests 的 0/1 标签训练 Verifier Rφ
3. 用 Rφ 做 local search，得到 expert $\pi_\star$，重标注 D → D′
4. 在 D′ 上用监督学习更新 generator πθ

→ 新的 $\pi_\theta$ 再回到第 1 步，循环迭代



# Outline

1 Background

2 Related work

3 Design

4 Experiments

5 Conclusion

# Experimental Setup

**硬件:**

- 训练：8× NVIDIA A100 (80 GB)
- 推理：单卡 RTX 4090 (24 GB)

**软件:**

- 基础模型：Llama-3.2-1B / Llama-3.1-8B-Instruct
- 框架：PyTorch + Transformers
- 优化器：AdamW，学习率 2e-5，批次 32
- 训练轮数：3 epoch（多轮交互样本 ≈ 1.5 M 对）

**评测基准:**

**MBPP**
- 374 题训练，500 题测试
- 题目类型：函数级别基础编程

**HumanEval**
- 164 题，仅测试阶段使用，用于验证泛化

**CodeContests**
- 1000 题训练，165 题测试
- 难度更高，贴近在线评测任务

指标
- pass@k：代码通过全部 private tests 的比例
- Turns to Solve：平均成功轮数（越小越好）

# Main Results

| Method | N | Llama-3.2-1B | | Llama-3.1-8B | | |
|---|---|---|---|---|---|---|
| | | *MBPP* | *HE* | *MBPP* | *HE* | CC |
| *Single-Turn* | | | | | | |
| Instruct | 1 | 35.1 | 25.6 | 52.1 | 59.8 | 3.6 |
| RFT | 1 | 35.7 | 34.1 | 53.7 | 54.9 | – |
| *Multi-Turn* | | | | | | |
| Instruct | 1 | 35.1 | 31.1 | 60.3 | 59.7 | 4.8 |
| +*BoN* | 5 | 47.3 | 35.7 | 69.7 | 62.9 | 13.8 |
| RFT | 1 | 31.1 | 31.7 | 58.9 | 61.2 | 7.2 |
| +*BoN* | 5 | 46.7 | 34.1 | 68.4 | 62.8 | 14.9 |
| μCODE | 1 | 37.9 | 35.4 | 62.1 | 60.9 | 7.9 |
| +*BoN* | 5 | **51.1** | **41.5** | **70.6** | **63.8** | **16.3** |

**Multi-Turn 大幅提升：**
所有方法从单轮 greedy 到多轮 BoN，pass@k 都有明显增长

**1B 小模型收益更明显：**
在 MBPP / HumanEval 上，μCODE(1B, BoN@5) 比最佳 baseline +4.4 / +5.8 个百分点

**8B 大模型全面领先：**
Llama-3.1-8B 上，μCODE 在 MBPP / HE / CodeContests 上 BoN@5 均为最优或并列最优

**高难 CodeContests 也有优势：**
在 CodeContests 上，μCODE 的 BoN@5 比 RFT 再高约 1.4%

# Train and Verifier

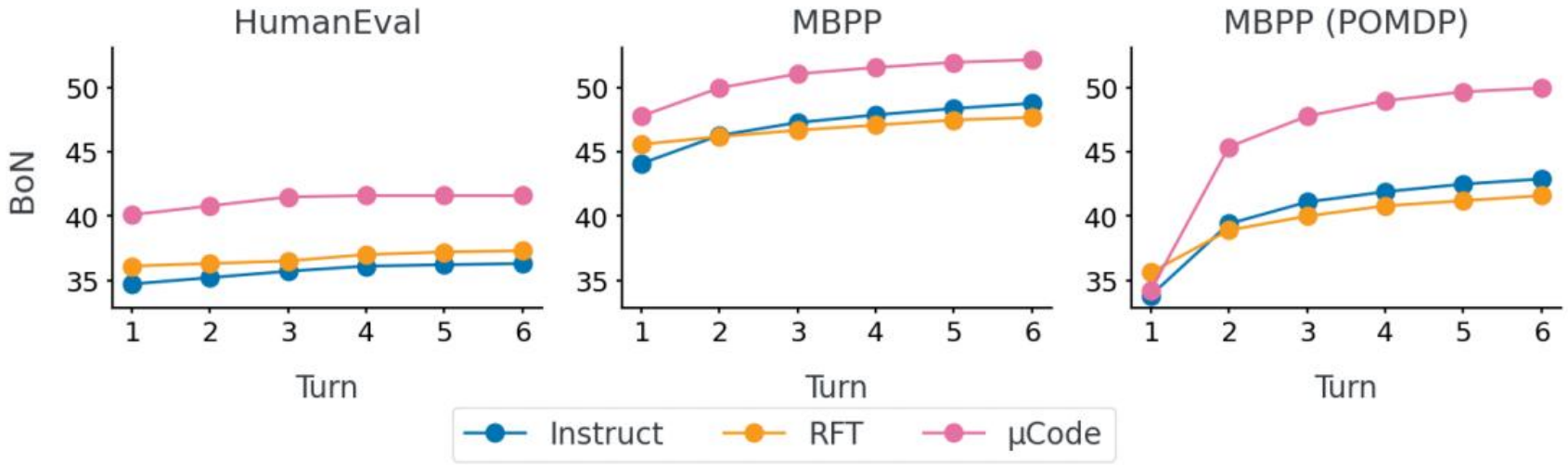
| Method | One-step | Verifier | *MBPP* | *HE* |
|---|---|---|---|---|
| RFT | ✗ | Oracle | 46.7 | 36.5 |
| $\text{RFT}_{\text{LV}}$ | ✗ | Learned | 49.0 | 38.9 |
| $\mu\text{CODE}_{\text{OV}}$ | ✓ | Oracle | 48.2 | 38.0 |
| $\mu\text{CODE}_{\text{LV}}$ | ✓ | Learned | 48.5 | 39.1 |
| $\mu$CODE | ✓ | Both | **51.1** | **41.5** |

- Learned Verifier 更好用
- One-step Relabeling 带来额外收益
- 双重 Verifier 最佳

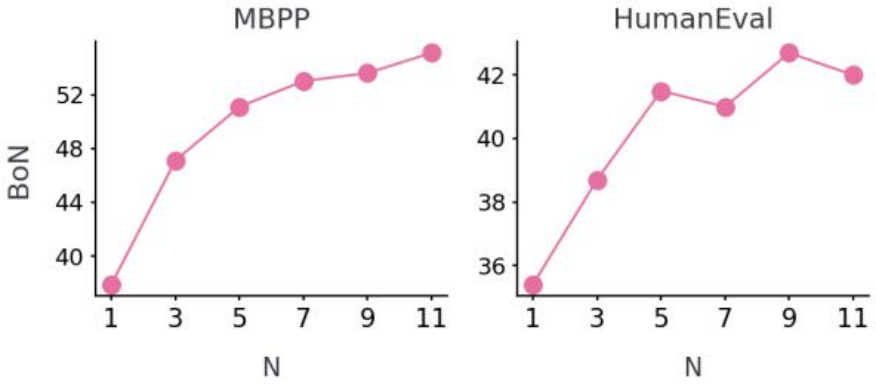
| Approach | Llama-3.2-1B | | Llama-3.1-8B | |
|---|---|---|---|---|
| | *MBPP* | *HE* | *MBPP* | *HE* |
| **Base** | | | | |
| Random | 31.7 | 24.6 | 58.0 | 57.9 |
| LV | 30.3 | 29.4 | 62.5 | 61.0 |
| PT | 46.4 | 33.4 | 68.4 | 60.7 |
| PT+LV | 47.3 | 35.7 | 69.7 | 62.9 |
| **RFT** | | | | |
| Random | 31.1 | 27.4 | 58.9 | 57.7 |
| LV | 33.2 | 29.6 | 62.2 | 61.3 |
| PT | 46.8 | 36.8 | 67.4 | 61.4 |
| PT+LV | 46.7 | 36.5 | 68.4 | 62.8 |
| **$\mu$CODE** | | | | |
| Random | 37.5 | 31.5 | 61.5 | 58.4 |
| LV | 43.3 | 36.5 | 64.8 | 60.6 |
| PT | 49.4 | 39.7 | 69.4 | 61.4 |
| PT+LV | **51.1** | **41.5** | **70.6** | **63.8** |

- Random < LV < PT
- PT+LV 互补
- 生成器 & μCODE 都受益

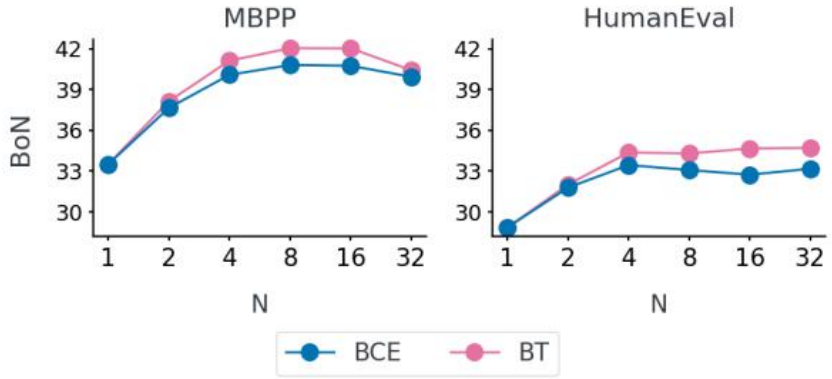
# Execution Feedback




多轮执行反馈




候选数 N 的搜索预算




Verifier Loss 设计

# Outline

1 Background

2 Related work

3 Design

4 Experiments

5 Conclusion

# Conclusion

- **Conclusion**
  - 提出了 μCODE：把多轮代码生成建模为 一步可恢复 的 MDP，用单步奖励 + learned verifier 来训练模型
  - 在 MBPP / HumanEval / CodeContests 上，多轮 BoN 设置下，相比 Instruct / RFT 都有明显提升，1B 小模型也能学会更好利用执行反馈
- **Inspiration**

➢ 能不能进一步提高?
  - 把 execution feedback 扩展成更丰富的信号，不仅看功能是否通过测试，还加入延迟、资源占用、部署是否成功等指标
  - 设计更强的 Verifier，结合静态检查、日志分析，而不仅仅是测一测对不对能不能跑
  - 从一步可恢复继续探索到更复杂的多步反馈、长期奖励（例如整个应用生命周期表现）

➢ 能不能用到我们的场景?
  - 把生成一套云边端应用 + 部署脚本,看成一次 multi-turn code generation，每轮根据仿真 / 容器运行结果给出执行反馈
  - 用户和Agent 多轮对话，后台自动跑 demo，运行日志 / 连通性 / 指标作为 reward 和训练信号，类似 μCODE 的框架迁移到 IoT 应用生成

➢ 泛化性?

# Q&A

2025.12.01