Revisiting Chain-of-Thought in Code Generation: Do Language Models Need to Learn Reasoning before Coding?

Liu Enbiao , Li Anqi, Yang Chaoding, Sun Hui, Li Ming

♠ NanJing University

ICML '25

Presented by Muhan Yuan

作者团队

Lamda实验室研究方向

- NLP
- Data Mining
- Pattern Recognition
- Code Intelligence
- Computer Vision

刘仁彪

南京大学在读博士

黎铭

南京大学教授

OpenCodeEval：

易用性: 将复杂的代码生成评测流程封装起来，让研究者可以更方便地评估自己的模型。

可扩展性: 支持多种主流的代码生成基准测试（如 HumanEval, MBPP 等），并且容易扩展以支持新的基准。

Outline

1 Background

2 Related work

3 Design

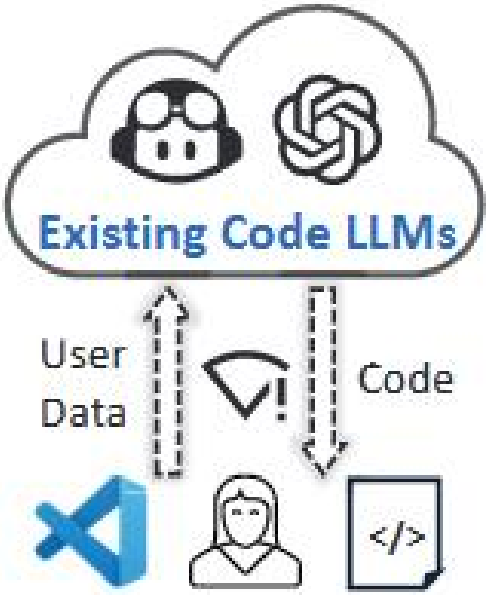
4 Experiments

5 Conclusion

LLMs驱动的软件工程革命

Large Language Models (LLMs)

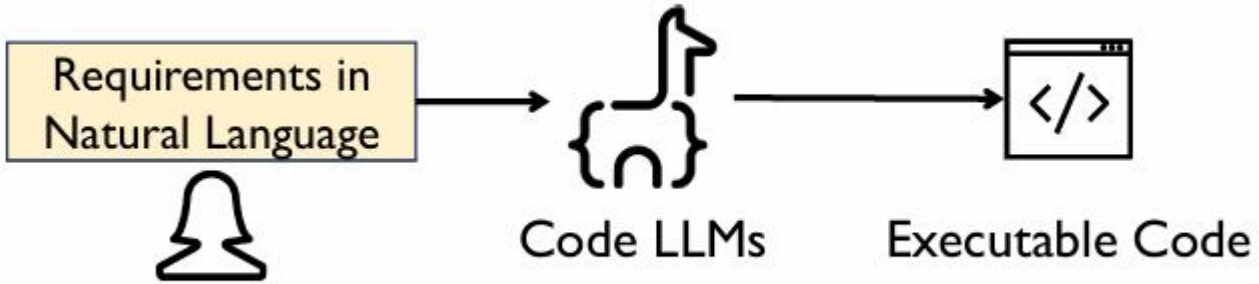
- LLMs 改变了我们与 AI 的互动方式
- 卓越的自然语言理解能力
- 在代码生成有广泛的应用前景



- 代码补全
- 代码生成
- 缺陷检测
- 文档编写

ChatGPT (OpenAI)

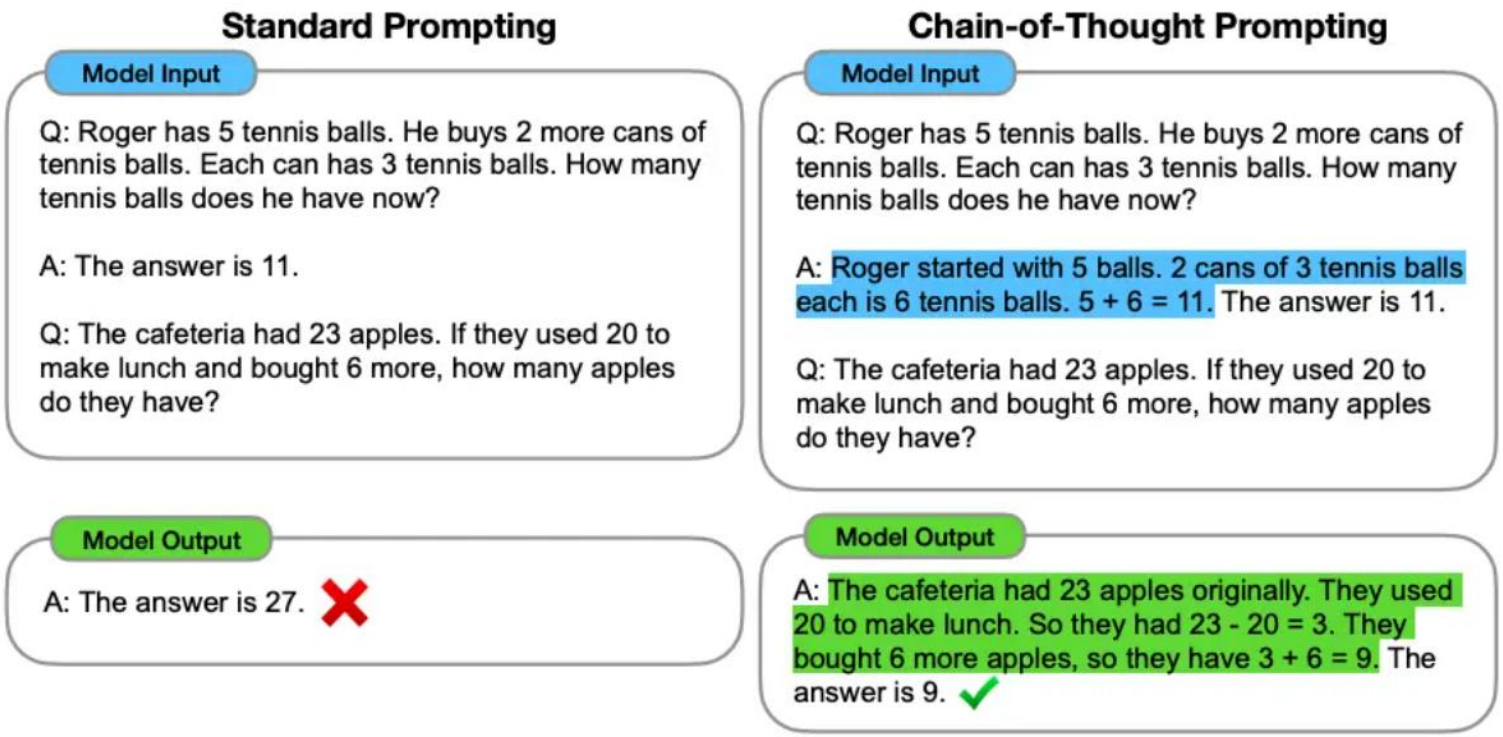
DeepSeek-Coder



CoT——提升复杂推理能力

什么是思维链 (Chain-of-Thought, CoT)?

- 2022年由Google Research提出
- 核心思想: 模仿人类“分步思考”的过程
- 实现方式: 在提示中加入详细的推理步骤范例
- 最终目的: 引导模型解决复杂问题，而非直接猜测答案



代码生成中的“常识”与“疑问”

“常识”：一个看似完美的模式?

- 代码生成任务复杂，需要严谨的逻辑推理
- 因此，研究者们很自然地将思维链（CoT）引入，形成了CoT -> Code的经典范-式

“疑问”：这是代码生成的唯一答案吗?

- 然而，这个看似完美的模式真的是最优解吗?
- 高质量的代码本身就是一种结构化、逻辑化的推理过程。
- **我们真的需要让模型先用自然语言“思考”一遍，再用代码“实现”吗?**

Outline

1 Background

2 Related work

3 Design

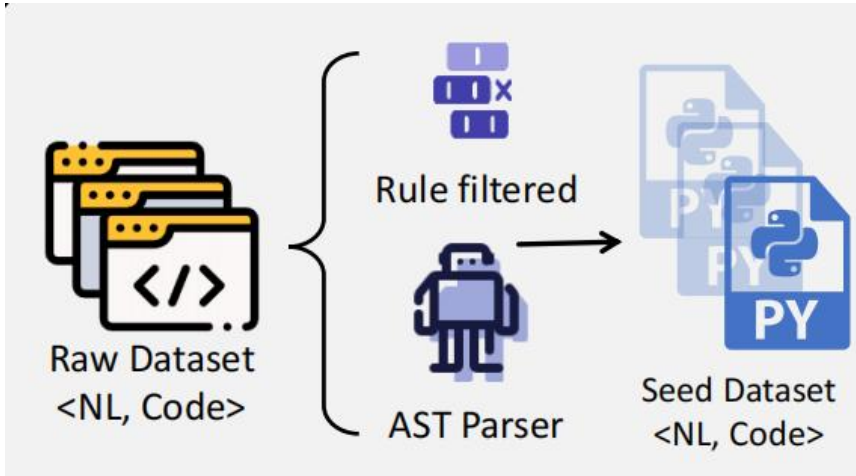
4 Experiments

5 Conclusion

Related work

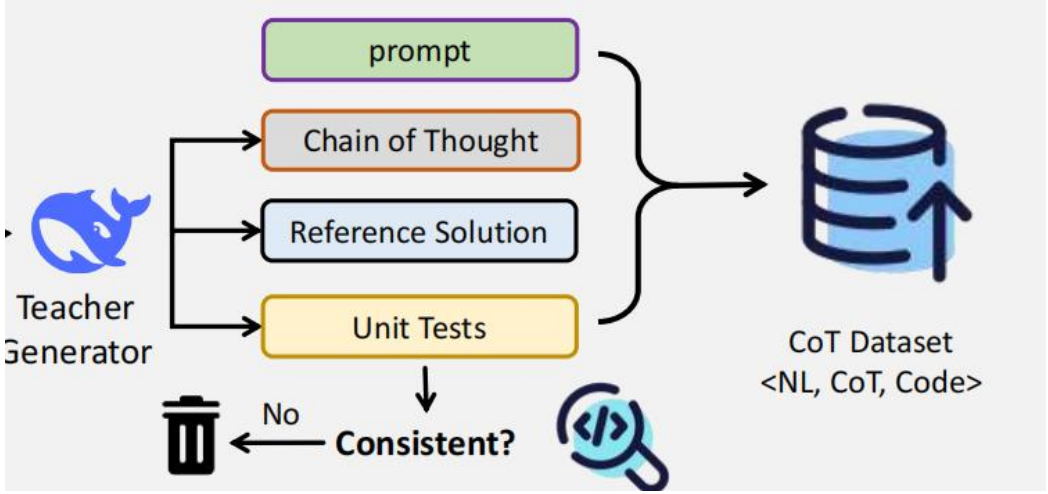
基于指令微调的代码生成

- **SFT (监督式微调):** 为提升Code LLM遵循指令能力的核心技术范式
- **Self-Instruct & Evol-Instruct:** 业界主流的自动化指令数据构建技术。通过模型自生成或演化，来创造大规模的SFT数据集（例如Code Alpaca，WizardLM）



基于提示工程的推理增强

- **提示工程 (Prompting):** 设计不同的CoT提示策略来引导模型思考，例如零样本CoT、结构化CoT等
- **知识蒸馏 (Distillation):** 将大模型生成CoT的“思考能力”迁移到小模型上



Related work

现有研究的局限性

L1: 数据集质量与格式不统一

- 现有用于代码生成的开源SFT数据集中，CoT和Code的格式五花八门，质量也参差不齐
- 这使得我们无法进行严格的受控实验，去精确地分离和评估CoT或Code在模型训练中的独立贡献

L2: 训练范式未经审视

- 几乎所有工作都无条件接受了CoT -> Code顺序，并将其作为训练模型时的铁律
- 这种“隐含的假设”从未被系统性地挑战或验证过。研究者们都在研究如何优化CoT的内容，却忽略了CoT与Code的相对顺序这个更基本的问题，可能导致整个社区在一个次优的范式上不断“内卷”

L3: 潜在的性能瓶颈与“过度思考”

- 强迫模型在生成结构化代码前，先生成一遍自由度更高的自然语言推理，可能并不符合代码生成的内在规律
- 这种固定的顺序可能成为模型性能的瓶颈。对于模型而言，这可能是一种低效的“过度思考”，不仅无法带来收益，甚至可能因为引入噪声而损害性能

Outline

1 Background

2 Related work

3 Design

4 Experiments

5 Conclusion

Design

解码器架构 (Decoder-only LLM)

- 将模型内部的**逻辑得分**，转化为每个词的**出现概率**（最小化模型预测与真实代码的差异）

$$\Longrightarrow \quad p = \mathrm{Softmax}(z) \quad \text{and} \quad p_j = \frac{\exp(z_j)}{\sum_{k=1}^{|\mathcal{V}|} \exp(z_k)}$$

监督式微调 (SFT) + **损失函数公式**

- 找到一组最优的**模型参数θ**，使得模型预测下一个词的概率，与“标准答案”无限接近

$$\Longrightarrow \quad \theta^* = \arg\min_{\theta} \sum_{i=1}^{|\mathcal{D}|} \sum_{t=1}^{|l_i|} \mathcal{L}\left(y_{t+1}^{(i)}, \mathrm{Softmax}(\mathrm{Linear}(h_t^{(i)}; \theta))\right)$$

评测指标 (Pass@k)

- 评估生成k次代码中，至少有1次正确的概率

$$\Longrightarrow \quad \mathrm{Pass@k} = 1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}$$

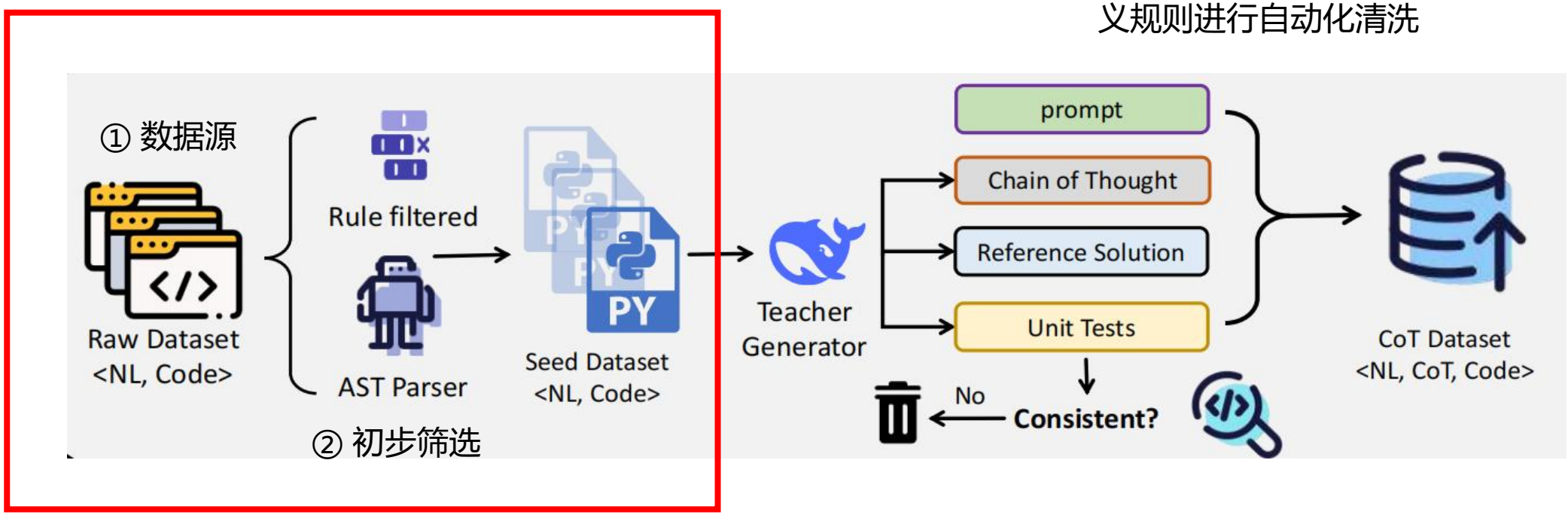
Design

C1: 现有数据集格式混乱、质量参差

- 现有开源数据集中，CoT与Code的格式不统一，难以进行受控实验
- 代码质量参差不齐，存在大量无法通过测试的样本，会引入噪声，干扰模型学习

M1: 设计高质量CoT数据生成流水线

- 流水线核心包含“初步筛选”和“教师模型精加工与质检”两大阶段
- 数据源广泛采集自Magicoder, Evol-Instruct-Code等6个主流开源数据集
- 初步筛选采用AST (抽象语法树)解析器和自定义规则进行自动化清洗



Design

教师模型

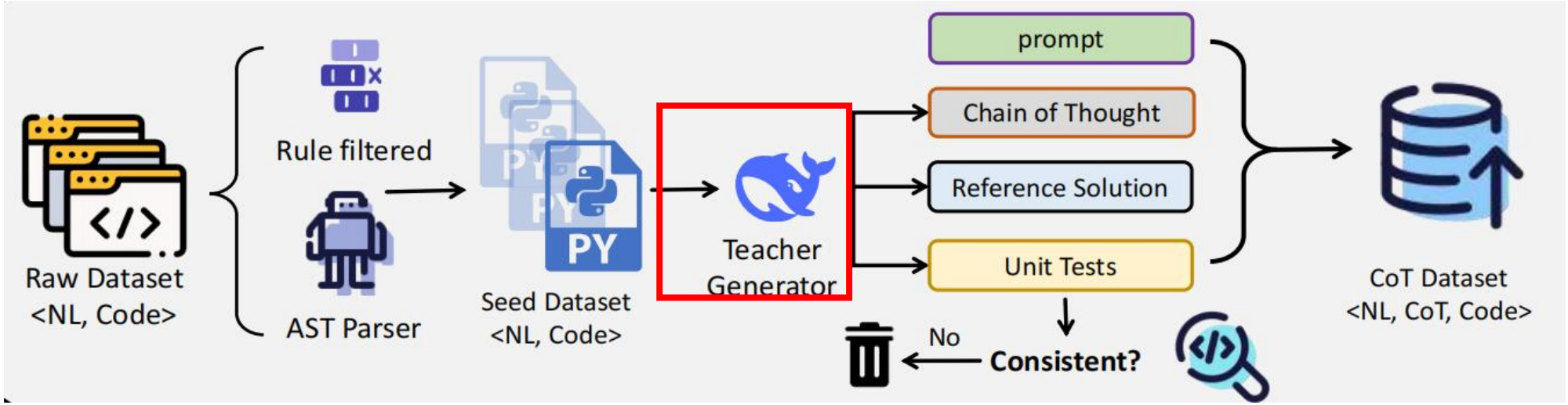
- 采用“上下文蒸馏 (Context Distillation)”方法，引入一个强大的预训练模型作为“教师”
- 本文选用 DeepSeek-V2.5 (当时最强的开源模型之一) 作为教师模型，保证了生成数据的质量上限

自动化数据标注

- 利用教师模型生成能力，代替人工标注，实现高质量数据的自动化生产

统一数据标准

- 通过教师模型和统一的Prompt，确保最终五万多条数据在风格、格式、质量上的高度一致，为后续进行公平的对比实验奠定了基础



Design

提示词工程

为了让教师模型能稳定、标准化地输出我们需要的内容，设计了一个精巧的Prompt，明确要求模型必须输出三个部分：

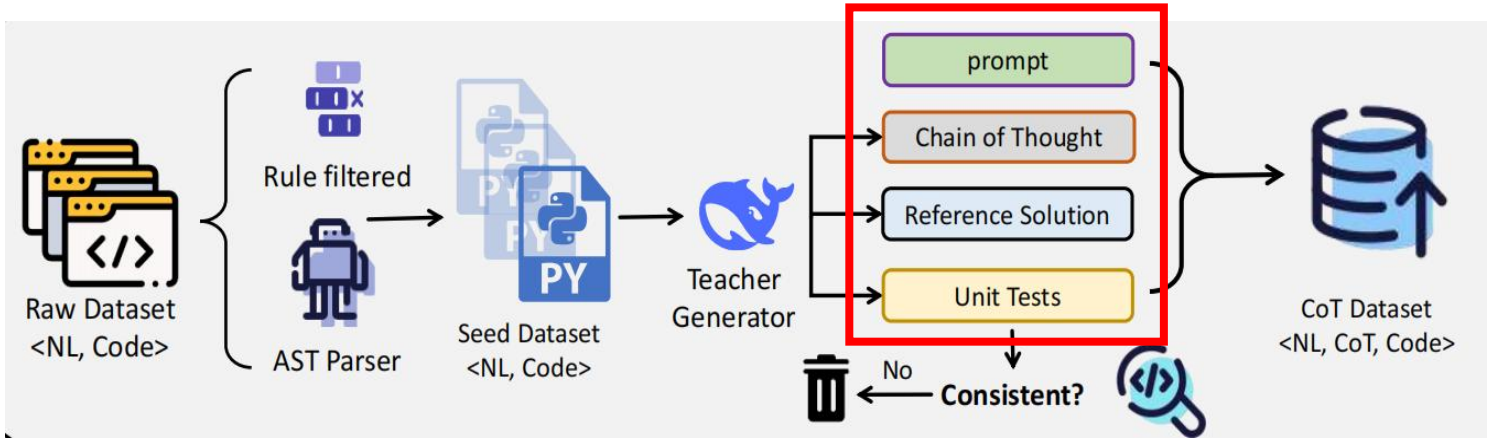
- [Analysis] (分析): 即我们需要的思维链 (CoT)
- [Solution] (方案): 即我们需要的代码 (Code)
- [Test] (测试): 即用于后续自动化质检的单元测试 (Unit Tests)

Prompt for Instruction Synthesis

You are a teaching assistant helping to create a Python programming task from a given code snippet. You will respond best to the Python programming task, including the reasoning process, reference solutions, and test code.

[Code Snippet]
{Code}

The response must have these parts:
[Analysis]
{Analyze the task and reason about the given task step by step}
[Solution]
{Write a high-quality reference solution in a self-contained script that solves the task}
[Test]
{Provide ten assert statements to check the correctness of your solution}



Design

自动化检测

- 采用自洽性过滤 (Self-Consistency Filter) 作为“自动化质检模块”
- 不仅仅检查代码的语法正确性，更要验证其功能正确性，即代码的运行结果是否符合预期

自动化工作流程 (Workflow)

1.输入: 模块接收由教师模型生成的 **[Solution]** 和 **[Test]**

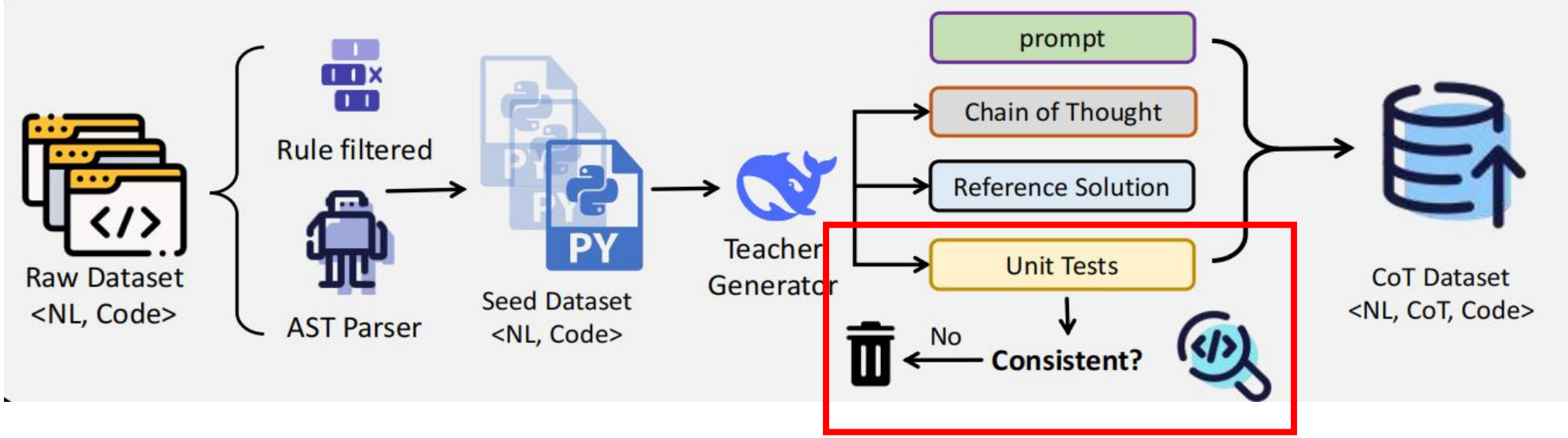
2.执行：在一个安全的沙箱环境 中，自动编译并执行[Solution]中的代码

3.验证：将代码的实际运行输出，与[Test]中预设的断言进行逐一比对

4.决策 (Decision):

IF 所有assert全部通过 **THEN** 数据合格 -> **保留**

ELSE 任何一个assert失败或程序报错 **THEN** 数据不合格 -> **丢弃**



Design

C2: 现有工作默认遵循CoT -> Code顺序

- 这个“隐含的假设”从未被系统性地挑战或验证过，可能导致研究在一个次优的范式上进行

M2: 验证传统范式的有效性

- 通过搭建四条并行的训练流水线，其唯一变量为输入数据的组织格式，以正面验证传统范式的有效性，并探索新范式的可能性

四类实验对象	用途	数据格式	意义
Seed Dataset	基线	Input: (问题) Output: (原始回答)	作为原始基线，衡量后续处理带来的提升
Code without CoT	消融实验	Input: (问题) Output: (代码)	移除CoT，用于验证CoT本身的必要性
Code follow CoT	传统范式	Input: (问题) Output: (CoT + 代码)	模拟传统范式，作为主要被挑战者
Code precede CoT	**本文新范式**	Input: (问题) Output: (代码 + CoT)	探索本文新范式，作为主要挑战者

Design

C3: 潜在的性能瓶颈与“过度思考”

M3: 多维度可量化的模型工具集

如何根据Experiments部分科学地解释**不同范式的性能差异**?
如何设计一套**可量化的指标**来客观地衡量是否过度思考?

诊断工具集详解:

1.模型学习效率分析 (Learning Efficiency Analysis):
- 采用条件困惑度 (Conditional Perplexity)分析
- 通过量化模型在预测CoT和Code时的“困惑程度”，来判断哪种顺序对模型来说学习难度更低、效率更高

2.泛化能力分析 (Generalization Analysis):
- 采用KL散度和验证集损失 (Validation Loss)分析
- 区分模型的表现是来源于“死记硬背” (Memorization)还是真正的“举一反三” (Generalization)

3.模型注意力机制分析 (Attention Mechanism Analysis):
- 注意力权重可视化
- 通过观察模型在生成内容时，将“注意力”更多地放在了哪些输入部分（是CoT还是Code），来推断其内部的决策逻辑

4.关键代码元素分析 (Key Element Analysis):
- 消融实验 (Ablation Study)
- 通过移除代码中的特定部分（如函数签名、注释），来测试哪些元素是模型理解和推理的关键“锚点”

Outline

1 Background

2 Related work

3 Design

4 Experiments

5 Conclusion

Experiments

硬件 (Hardware):
- 微调 (Fine-tuning): 8 x NVIDIA A100 GPU (80 GB)

软件 (Software):
- 基础模型 (Base Model): DeepSeek-Coder-Base-6.7B
- 训练框架 (Training Framework): DeepSpeed (ZeRO-3)
- 评测框架 (Evaluation Framework): OpenCodeEval (由作者本人开发)

训练数据集 (Training Dataset)
- 训练集 (Training Set): 51,200 条
- 验证集 (Validation Set): 1,093 条

总计: 52,293 条 (问题, CoT, 代码) 三元组

评测基准 (Evaluation Benchmarks)
- HumanEval & MBPP (+): 用于评测模型基础的代码生成和问题解决能力。
- LiveCodeBench: 用于评测模型的跨时间域外(OOD)泛化能力，有效避免数据污染。
- BigCodeBench: 用于评测模型遵循复杂指令和调用外部API的能力。
- MultiPL-E & EvalPerf: 分别用于评测跨编程语言的迁移能力和生成代码的运行效率。

Experiments

核心发现 (Core Finding):

- 新范式precede(先编码, 后解释) 的平均性能达到了 71.88
- 传统范式follow}(先思考, 后编码) 的平均性能仅为 65.43

惊人结论 (Key Conclusion):

- 仅仅改变**CoT和Code的顺序**，就带来了高达 **9.86%** 的相对性能提升

意外发现 (Unexpected Finding):

- 传统范式follow的表现甚至不如不使用CoT的w/o(65.43 vs 70.88)，这说明不恰当的训练范式反而有害。

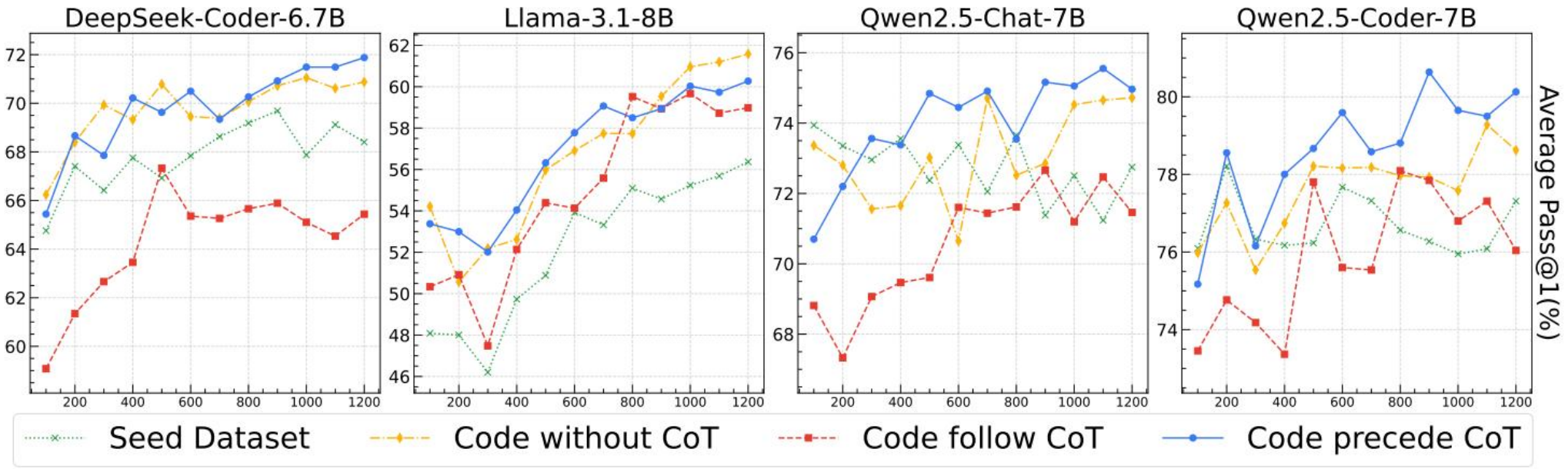
Method	HumanEval(+)	MBPP(+)	Average
Seed	68.29(62.20)	77.51(65.61)	68.40
$C_{\text{w/o}}$	70.73(64.63)	80.42(67.72)	70.88
C_{follow}	67.07(59.75)	74.33(60.58)	65.43
C_{precede}	71.95(67.68)	80.69(67.20)	71.88

Experiments

跨基础模型验证：

- 无论基础模型是什么、架构有何不同，precede(蓝色线) 的性能曲线，几乎在所有模型的训练全程中都稳定地、显著地优于传统范式follow (红色线)

结论：性能差异与模型架构无关



Experiments

跨任务难度验证：

- 结果显示，新范式precede不仅在简单问题上表现优异，在需要更强逻辑推理能力的Medium和Hard级别的难题上，其性能优势更加明显

结论：新范式对复杂问题同样有效

Method	Easy(45)	Medium(91)	Hard(44)	Overall(180)
Seed	42.2%(19)	15.4%(14)	9.1%(4)	20.6%(37)
$C_{\text{w/o}}$	42.2%(19)	16.5%(15)	2.3%(1)	19.4%(35)
C_{follow}	31.1%(14)	14.3%(13)	4.5%(2)	16.1%(29)
C_{precede}	44.4%(20)	24.2%(22)	6.8%(3)	25.0%(45)

Experiments

跨外部环境验证：

- 更换教师模型: 即便将数据生成的教师模型从DeepSeek-V2.5更换为GPT-4o，precede更优的结论依然成立
- 更换数据来源: 即便从零开始，使用The Stack这一大规模原始代码库来合成数据，结论也同样稳固

结论：差距不受外部因素影响

Method	HumanEval(+)	MBPP(+)	Average
$C_{\text{w/o}}$	71.95(65.85)	77.77(67.46)	70.76
C_{follow}	65.24(59.14)	76.45(62.96)	65.95
C_{precede}	72.56(66.46)	78.57(67.72)	71.33

更换教师模型为GPT-4

Method	HumanEval(+)	MBPP(+)	Average
$C_{\text{w/o}}$	68.29(61.58)	76.98(62.43)	67.32
C_{follow}	61.58(55.48)	75.13(60.31)	63.13
C_{precede}	69.51(64.63)	77.24(63.22)	68.65

更换数据来源为The Stack

Experiments

深度原因探究

回顾结论:前面的实验结果清晰地证明了Code -> CoT新范式的优越性

提出问题:那么，为什么仅仅改变顺序，就会带来如此大的性能差异？其背后的深层原因是什么？

为了回答这个问题，我们启动在Design部分M3模块中设计的**“多维度诊断工具集”**，从模型行为的蛛丝马迹中寻找答案

1. 模型学习效率分析
2. 泛化能力分析
3. 模型注意力机制分析
4. 关键代码元素分析

Experiments

模型学习效率分析

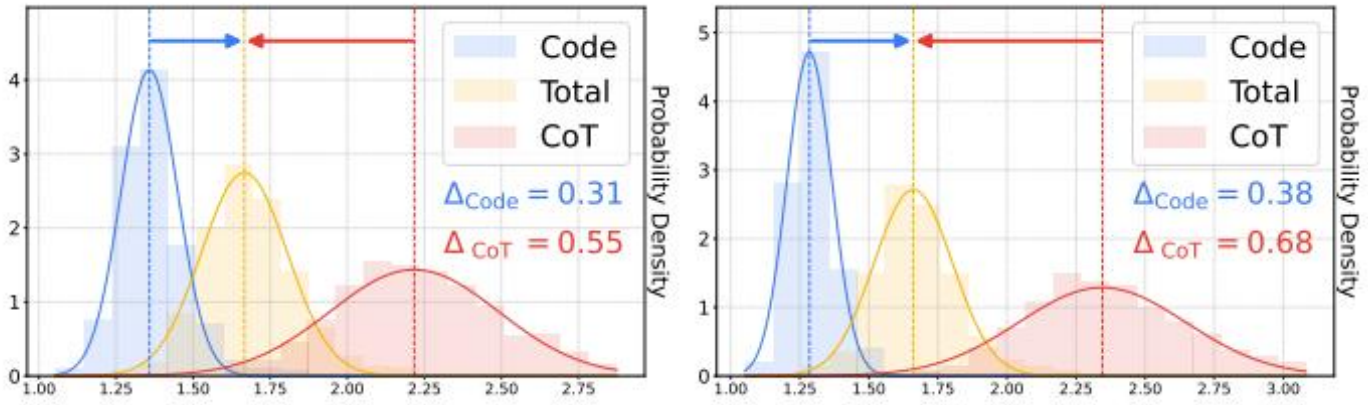
诊断工具: 条件困惑度 (Conditional Perplexity)

技术解读:

- 困惑度(PPL)是衡量模型预测下一个词时“有多不确定”的指标，PPL越低，代表模型学得越轻松、越确定。
- 我们分别计算了CoT和Code两部分的PPL，两者PPL的差距 (Δ) 越小，说明模型在学习这两部分时难度越均衡。

结论：
Code -> CoT新范式对模型来说学习难度更低、更均衡

原因猜想:
代码拥有严格的语法结构，可以为后续生成自由度更高的CoT提供一个强大的“上下文锚点”，从而降低了整体的学习难度



Experiments

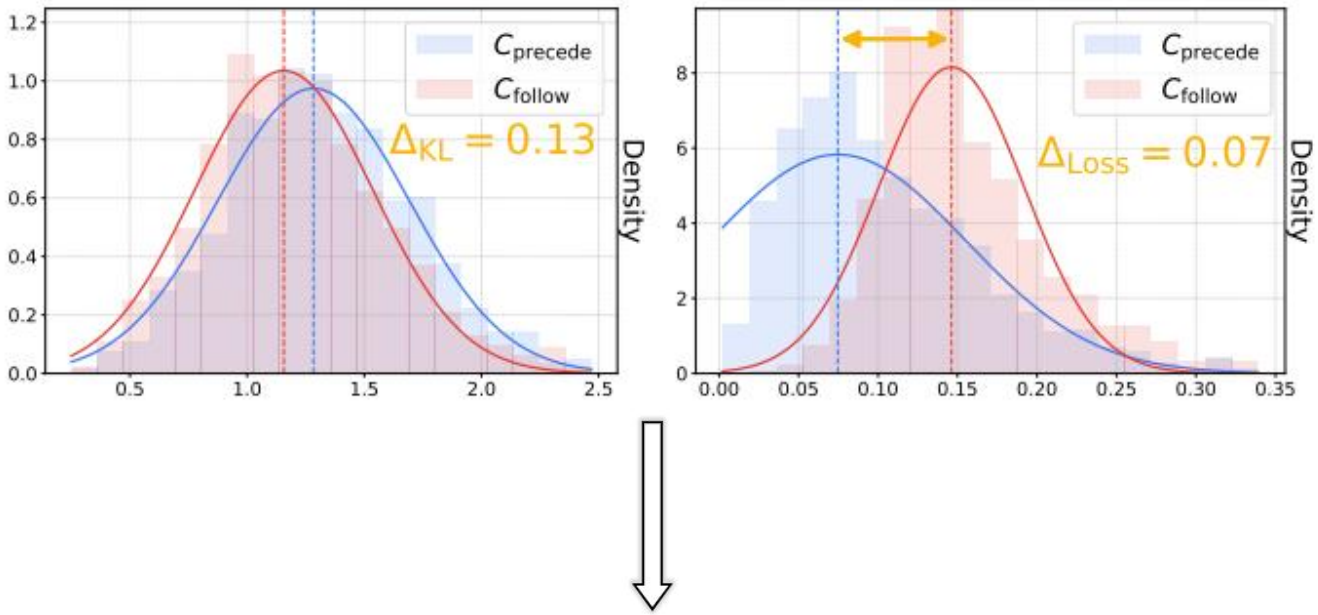
泛化能力分析

诊断工具: KL散度 (KL Divergence) & 验证集损失 (Validation Loss)

技术解读:

- **KL散度:** 衡量训练后的模型与原始基础模型之间的“差异程度”。可以理解为，SFT过程对模型进行了多大程度的“改造”

- **验证集损失:** 衡量模型在从未见过的验证数据上的表现。损失越低，说明模型的泛化能力越强，即“举一反三”的能力越强



左图 (KL散度):两种范式（蓝色线和红色线）的KL散度分布差异很小

结论: 这说明两种训练方式对模型的“改造”程度是相似的

右图 (验证集损失):新范式precede(蓝色线) 的验证集损失，显著低于传统范式follow(红色线)

结论 :在改造程度相近情况下，新范式学到的知识泛化能力更强，证明了新范式并非让模型死记硬背，而是教会了它一种更通用、更有效的解决问题的能力

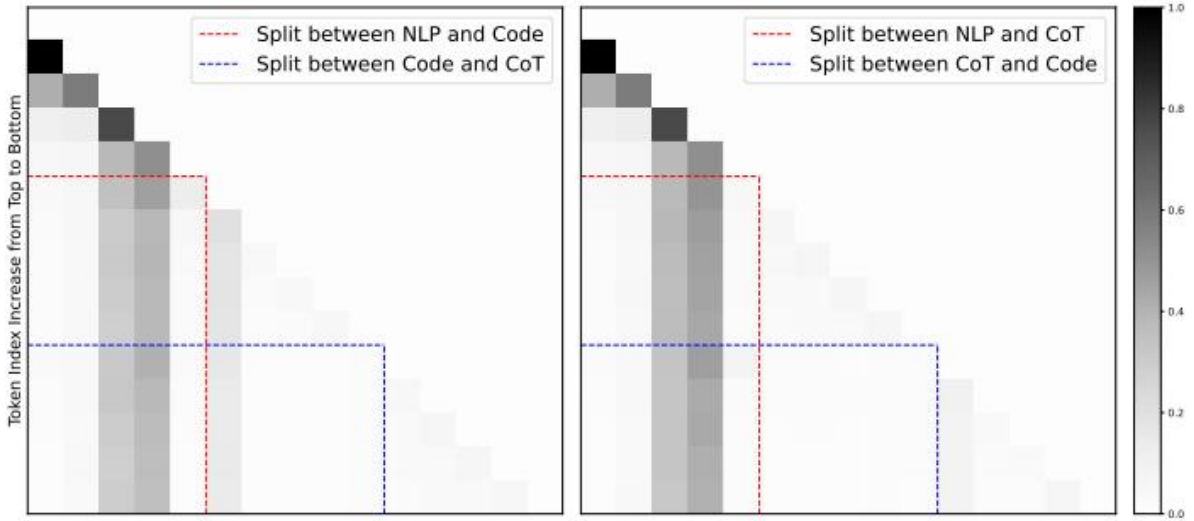
Experiments

模型注意力机制分析

诊断工具: 注意力权重可视化 (Attention Weight Visualization)

技术解读:

- 注意力权重可以理解为模型在处理信息时的专注度。权重越高，代表模型认为这部分输入信息对生成当前内容越重要。
- 通过观察注意力的分布，我们可以推断模型在“思考”时，更依赖哪些信息



右图分析 (传统范式):
注意力分布比较均匀，Code部分对CoT部分没有表现出特别的关注。

左图分析 (新范式):
模型在生成后续CoT（解释）的过程中，对前面的Code部分表现出了显著增强的注意力（热力图中对应区域颜色更深）

结论:
在新范式下，模型会回头审视已经生成的代码，并试图去深刻理解代码与后续解释之间的关系。这表明模型不仅仅是在生成，更是在理解与解释，而传统范式更像是在按步骤执行

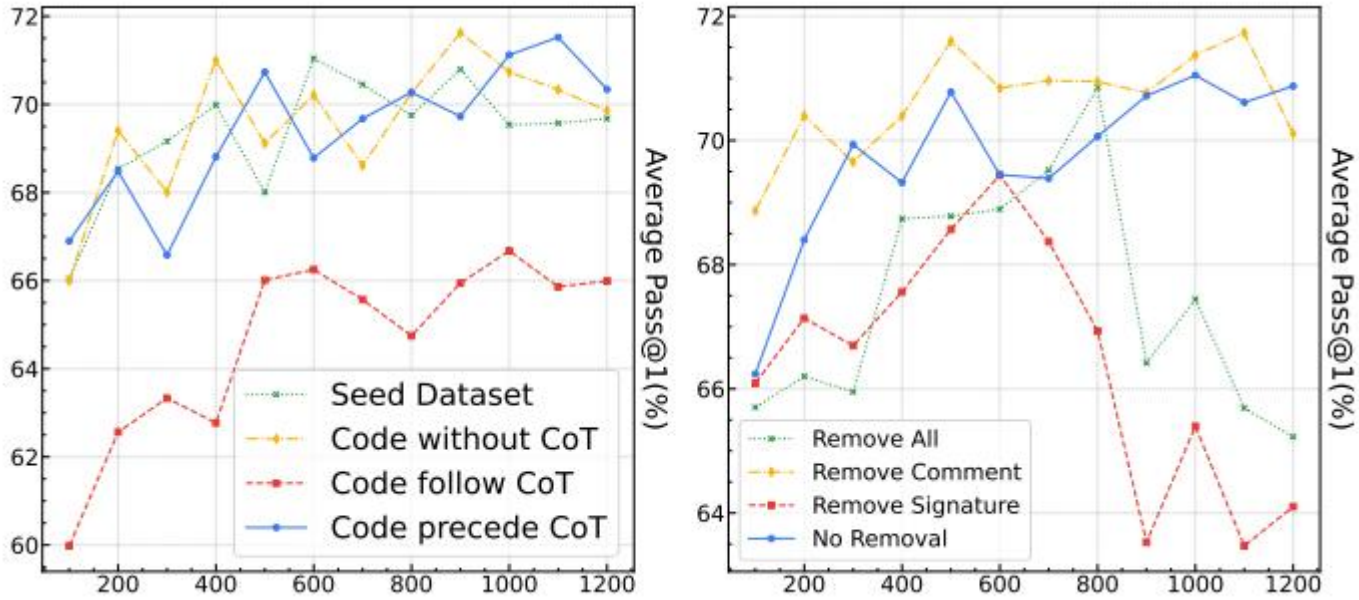
Experiments

关键代码元素分析

诊断工具: 消融实验 (Ablation Study)

技术解读:

- 一种通过做减法来研究系统组成部分的方法
- 系统性移除代码中的某部分（如注释或函数签名），重新训练模型，观察性能下降程度，判断移除部分重要性



移除注释 (黄色线): 模型的性能曲线与未作移除的基线（蓝色线）几乎重合

移除函数签名 (红色线): 模型的性能断崖式下跌，甚至比传统范式follow还要差得多

结论：
函数/类签名 (Signature) 是模型理解、推理代码的关键锚点。就像是连接“自然语言需求”和“编程语言实现”的核心桥梁。模型高度依赖签名来理解任务的目标、输入和输出。相比之下，自然语言的注释对模型的学习过程影响甚微

Outline

1 Background

2 Related work

3 Design

4 Experiments

5 Conclusion

Conclusion

Conclusion

- 重新审视并挑战传统范式：本文通过严谨的实验，首次系统性地挑战了代码生成中CoT -> Code的传统训练范式
- 验证新范式的优越性：实验证明，Code -> CoT（先编码，后解释）是一种更优的训练策略，仅改变顺序即可带来高达9.86%的显著性能提升
- 揭示深层原因：通过多维度诊断，证明新范式在学习效率、泛化能力、注意力机制上均表现更优，并揭示了函数签名在模型理解代码中的关键作用

高质量、结构化的代码本身就是一种**优质的思维链 (Code is a reasoning process)**，传统的自然语言CoT，更适合作为对代码的“解释 (Explanation)”，**而非前置的“推理 (Reasoning)”**

Conclusion

Inspiration

➢ **能不能进一步提高?**

- 探索更细粒度的代码推理: 本文证明了函数签名是关键，未来能否自动化地识别代码中其他关键的推理结构（如复杂的控制流、核心算法逻辑），并加以强化学习?
- 跨领域迁移: Code -> CoT的思想，能否推广到其他高度结构化的生成任务，例如公式证明、方程式生成、技术文档撰写等?

➢ **能不能用到我们的场景?**

- 数据集构建: 未来在为代码模型构建SFT数据集时，应优先考虑Code-first的数据格式
- 提示工程: 设计复杂代码生成提示时，引导模型先生成函数签名和代码骨架，再逐步完善细节和解释

➢ **泛化性?**

- 本文验证到了33B模型，那么对于更大规模的模型（如>100B），这个结论是否依然成立?
- Code -> CoT范式如何与其他技术（如RAG）结合？混合使用时是否存在冲突?

Q&A

2025.09.04