考核报告

利用提示词工程优化基座模型代码能力对齐教师模型探索报告

摘要

研究目标： 本项目的核心目标不是“单次刷分”，而是通过**可复用的提示词工程方法**，将基座模型（qwen3-8b）在代码生成任务上的行为逐步对齐到高能力教师模型（优先参考 qwen3_coder_plus_t2）的解题风格与稳定性。

研究方法： 先做多模型能力盘点与题型分布分析，再以教师模型在高差距题目的通过行为为参照，反向提炼“可执行约束”，逐版写入 `solution_t1.py` 到 `solution_t5.py` 的 system/user/retry 提示词中，形成“生成-评测-诊断-修复”的闭环。

方法创新：

1. 以“教师模型行为差距”驱动提示词改造，而非只靠经验调参。
2. 将自然语言建议压缩为结构化约束（`[PLAN]` / `[CHECK]`）并进行版本化对照实验。
3. 引入错误类型分流（WA/TLE/RE/FORMAT）与失败护栏（mapping/process/query/ties）并观察增益迁移。

主要结果： 基座模型从 `baseline=0.418750` 提升到 `baseline_t5=0.453125`。
核心提升：+0.034375（+11/320 通过样本）。

结论： 提示词工程可以在不改模型参数的前提下，稳定推动基座模型向教师模型行为对齐；下一步高收益方向集中在 qid=3、11 等高差距题专项约束。

关键词： 提示词工程；教师模型对齐；代码生成；错误分流；结构化约束；pass@1

目录

- 一、数据分析
  - 1.1 多模型能力对比
  - 1.2 题型分布与题型正确率
  - 1.3 教师模型选取
- 二、提示词优化
  - 2.1 总体优化逻辑（对齐教师模型）
  - 2.2 t1→t5 版本演化（代码级说明）

- 2.3 错误判断与修复分支
- 2.4 教师模型参考如何落到提示词
- 三、总结
  - 3.1 baseline 链路总结
  - 3.2 教师模型链路总结
  - 3.3 结论与下一步
- 参考文献
- 附录 A：全量分题矩阵（baseline 与 plus）
- 附录 B：加速方法

一、数据分析

1.1 多模型能力对比

1.1.1 多模型总体对比

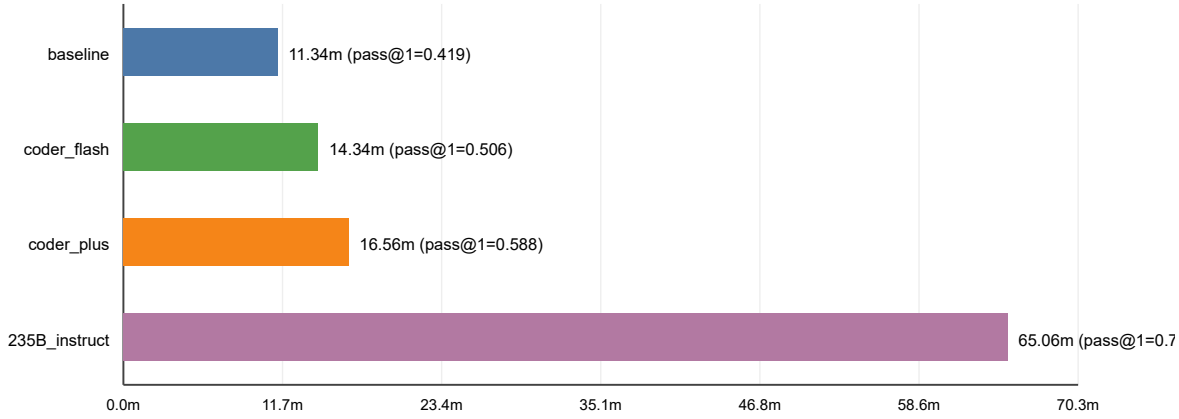
为了给基座模型选择合适的教师模型，我们先做了“同数据、同采样配置、同评测脚本”的横向评估。这里不仅看准确率（pass@1），也看推理耗时，因为本项目是提示词工程迭代任务，需要多轮实验反复验证，模型成本会直接影响可迭代速度。`pass@1` 作为代码生成评测核心指标，已被 HumanEval、MBPP、APPS、LBPP 等基准广泛采用[7-10]。综合“效果上限 + 调参可迭代性”两个维度，多模型总体结果如下：

表 1 多模型总体指标对比（准确率-耗时）

Run	avg pass@1	passed/320	elapsed(min)
baseline（qwen3-8b)	0.418750	134	11.34
qwen3_coder_flash(30b)	0.506250	162	14.34
qwen3_coder_plus(480b)	0.587500	188	16.56
qwen3_235B_instruct(235b)	0.734375	235	65.06

图 1 主要模型总耗时对比（分钟）

Figure 2. Total Elapsed Time by Model



图注：图 1 主要用于评估不同模型的实验成本。该图与表 1 共同构成教师模型选型依据。

结合表 1 与图 1 可以得到三点关键判断。第一，235B 的准确率上限最高，但耗时约为 coder_plus 的 3.9 倍（65.06 vs 16.56 分钟），在“需要频繁跑实验”的提示词优化阶段，时间成本过高。第二，coder_plus 在准确率上明显高于 baseline 与 coder_flash，同时耗时仍处于可接受范围，具备“高质量示范 + 可反复对比”的双重优势。第三，本项目目标是“让基座模型对齐教师模型的代码行为”，教师并不一定要绝对最强，而应优先选择“稳定、可复现、可高频使用”的模型。**因此，最终选取 qwen3_coder_plus（重点参考 t2）作为教师模型。**

1.1.2 多模型分题对比

仅看总体 pass@1 容易掩盖真实差异：有些题在所有模型都能做对，对优化没有指导意义；真正关键的是“哪些题教师能做对而基座做不对”。因此必须做分题对比，定位可迁移的高收益差距题，再把这些题上的教师行为提炼成提示词约束，避免在低价值题上反复迭代。

表 2 多模型分题通过率对比

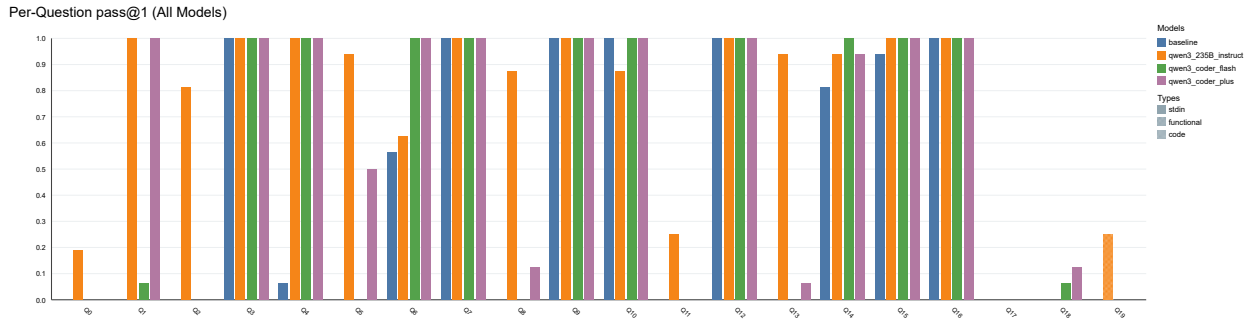
Q	qid	baseline	coder_flash	coder_plus	235B_instruct
0	0	0.0000	0.0000	0.0000	0.1875
1	3	0.0000	0.0625	1.0000	1.0000
2	4	0.0000	0.0000	0.0000	0.8125
3	6	1.0000	1.0000	1.0000	1.0000
4	11	0.0625	1.0000	1.0000	1.0000
5	13	0.0000	0.0000	0.5000	0.9375
6	17	0.5625	1.0000	1.0000	0.6250
7	20	1.0000	1.0000	1.0000	1.0000

Q	qid	baseline	coder_flash	coder_plus	235B_instruct
8	25	0.0000	0.0000	0.1250	0.8750
9	27	1.0000	1.0000	1.0000	1.0000
10	28	1.0000	1.0000	1.0000	0.8750
11	29	0.0000	0.0000	0.0000	0.2500
12	35	1.0000	1.0000	1.0000	1.0000
13	43	0.0000	0.0000	0.0625	0.9375
14	53	0.8125	1.0000	0.9375	0.9375
15	54	0.9375	1.0000	1.0000	1.0000
16	57	1.0000	1.0000	1.0000	1.0000
17	62	0.0000	0.0000	0.0000	0.0000
18	64	0.0000	0.0625	0.1250	0.0000
19	130	0.0000	0.0000	0.0000	0.2500

图 2 多模型分题通过率矩阵（模型 × 题目）



图 3 多模型分题通过率曲线对比



图注：图 2 用于观察不同模型在单题维度的高低分布，图 3 用于观察各模型在题目序列上的波动趋势。两图结合可以定位“教师可解、基座未解”的关键差距题。

数据结论：

1. `qid=3`、`qid=11` 是基座与教师之间的核心差距点（0 vs 1）。
2. `qid=64` 存在“模型非单调”，说明不是参数量越大越稳，提示词和行为约束同样重要。
3. 高分题（Q3/Q7/Q9/Q12/Q16）在多模型间趋同，增益应聚焦在低分难题。

重点：分题差距是提示词优化的直接靶点，尤其是 qid=3 与 qid=11。

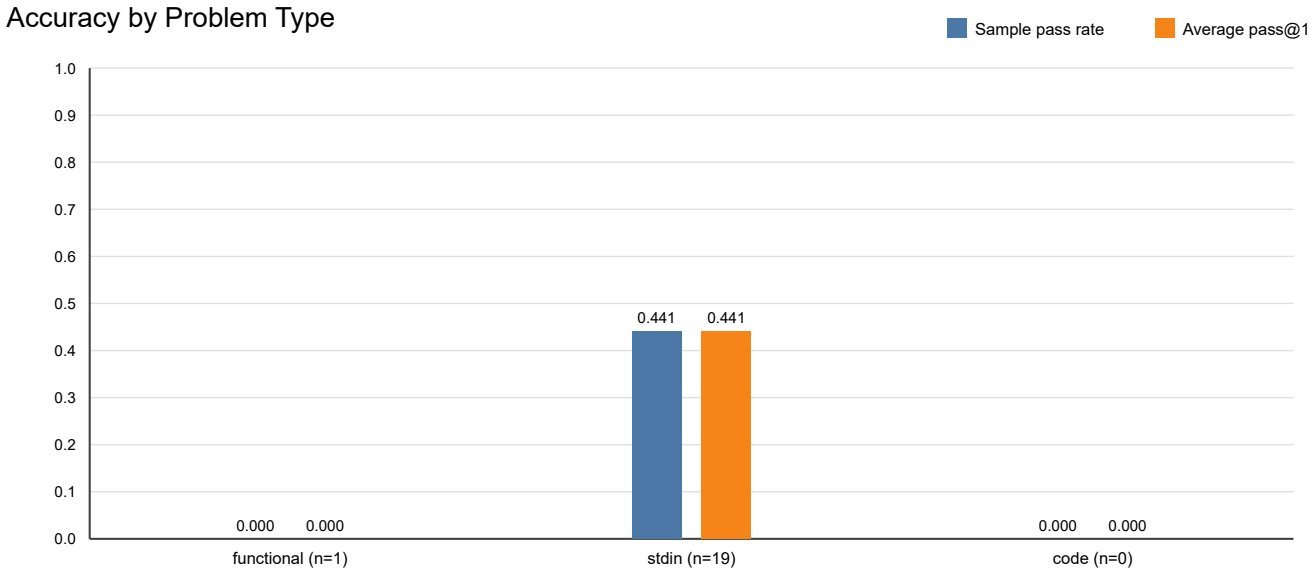
1.2 题型分布与题型正确率

1.2.1 题型分布

表 3 题型分布统计

题型	数量
stdin	19
functional	1
code	0

图 4 基座模型在题型维度的准确率表现



图注：图 4 展示 baseline 在不同题型上的准确率分布，用于判断优化主战场。

如图 4 所示，基座模型的主要可提升空间集中在 `stdin` 题型，functional 分支在当前配置下几乎没有有效通过样本。

1.2.2 题型正确率

表 4 不同模型在各题型上的正确率

Run	stdin	functional
baseline	0.440789	0.000000
qwen3_coder_flash	0.532895	0.000000
qwen3_coder_plus	0.587500	0.000000
qwen3_235B_instruct	0.759868	0.250000

结论： 本次对齐任务本质上是 `stdin` 稳定性对齐。
题型占比上，stdin=95%（19/20），决定了优化主战场必须是 stdin。
进一步分析上面的两张表，可以更明确地解释“为什么先做题型分析”。首先，数据集里 `stdin` 占 95%（19/20），这意味着任何泛化优化如果不能改善 `stdin`，整体分数几乎不会动。其次，functional 只有 1 题（Q19），但它在 baseline、coder_flash、coder_plus 上均为 0，仅 235B 达到 0.25，说明该题型对模型契约遵守（函数签名、返回路径、禁 stdin/print）要求更苛刻，后续应作为“分支模板专项问题”处理，而不是用统一 stdin 提示词强行覆盖。最后，题型分析还帮助我们避免误判：如果只看总体分数，可能会误以为“算法能力不足”是主因；而从分布看，真正主导

项是 stdin 的 I/O、语义执行一致性与复杂度稳健性。因此后续 t1~t5 的改动重点放在错误分流、结构化检查和 failure guards，是与数据分布一致的设计。

1.3 教师模型选取

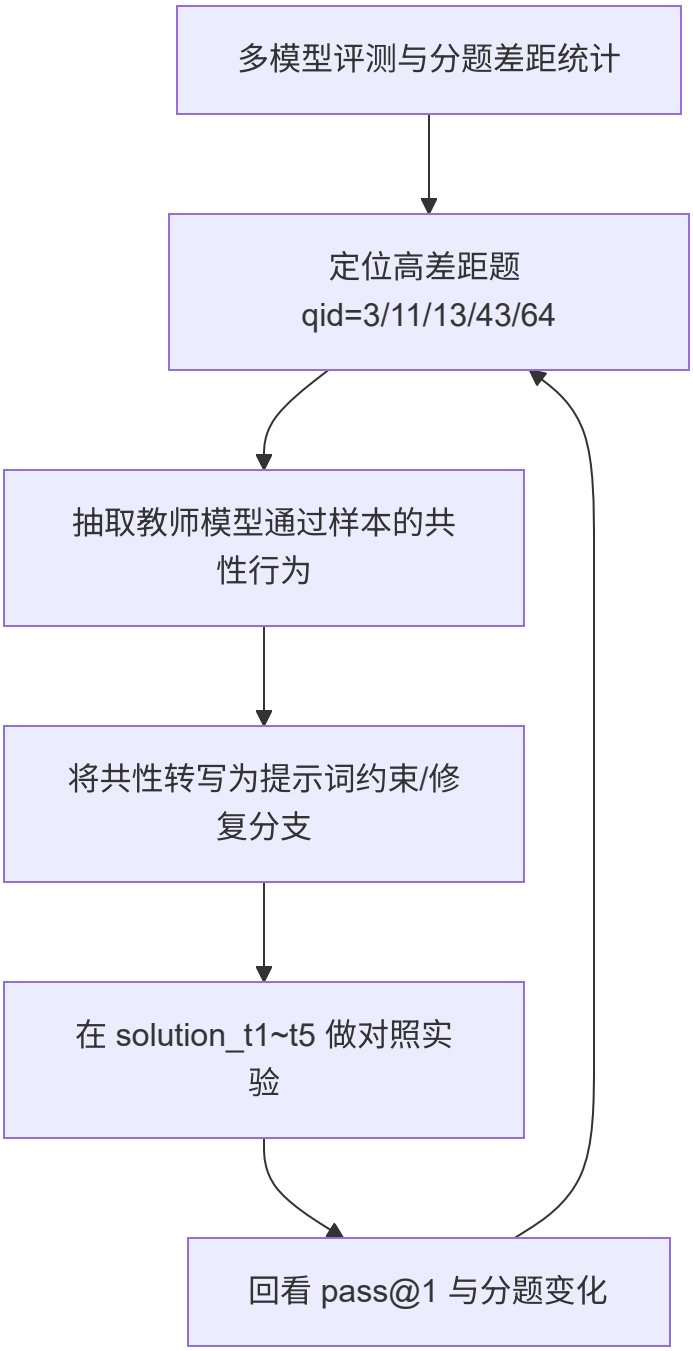
基于前两节分析，我们最终选择 `qwen3_coder_plus(480b)`（重点参考其 `t2` 结果）作为教师模型。原因是它在准确率上显著优于 baseline 与 coder_flash，同时计算成本远低于 235B，能够支撑“高质量示范 + 高频迭代验证”的项目需求。更关键的是，分题层面它在多个关键差距题（如 qid=3、11、13）上表现稳定，便于我们把教师模型的成功行为提炼成可执行提示词约束，持续迁移到基座模型。

教师模型选型结论：优先使用 qwen3_coder_plus_t2 作为对齐参照。

二、提示词优化

2.1 总体优化逻辑（对齐教师模型）

本项目采取“教师行为蒸馏到提示词约束”的路线，而非改模型参数：



关键思想：把“教师模型做对时隐含的工程行为”显式化，变成基座模型必须遵守的输出契约。

2.2 t1→t5 版本演化（代码级说明）

2.2.1 总表

表 5 baseline_t1 至 baseline_t5 的提示词策略演化

版本	关键代码改动	逻辑变化	结果
t1	`solution_t1.py` 增加 `_detect_task_profile`、强化 `_build_system_prompt`	从“统一提示”改为“任务画像 + 复杂度提醒 + 错误分流修复”	0.400000

版本	关键代码改动	逻辑变化	结果
t2	`solution_t2.py` 简化主提示，保留 `_classify_feedback` 与 repair 分流	从“重约束”回退到“轻约束 + 明确修复目标”	0.437500
t3	`solution_t3.py` 引入 `_build_plan_template` 强制 8 行 `[PLAN]`	将思考结构显式化，但计划-实现绑定过强	0.421875
t4	`solution_t4.py` 引入 `_build_check_template`（`[CHECK]`）	以布尔式 checklist 替代长 PLAN，降低冗长自然语言	0.443750
t5	`solution_t5.py` 增加 `_build_failure_guards` + retry `# [FIX]` 要求	在 CHECK 基础上加入反模式护栏，强化修复轮“针对性改错”	0.453125

2.2.2 关键代码片段（示意）

1. t1：引入任务画像与强约束（`solution_t1.py`）

t1 的改动动机是：在教师模型通过样本中，最稳定的共性不是“花哨技巧”，而是“先判断任务契约，再决定实现路径”。具体表现为：stdin 题优先保证输入输出严格匹配，functional 题优先保证签名与 return 契约，且在大约束场景会主动避免高复杂度写法。基于这一观察，t1 增加了 `_detect_task_profile`，希望把“先判题型再写代码”的行为前置到 system/user 提示里，减少基座模型在起手阶段走偏。虽然 t1 总分回退，但这一步建立了后续版本的基础骨架：任务感知 + 错误分流。

```
def _detect_task_profile(question_prompt: str) -> dict:
    ...
    return {
        "likely_task": "functional" if likely_functional else "stdin",
        "has_large_constraints": has_large_constraints,
    }
```

2. t3：强制 `[PLAN]` 头（`solution_t3.py`）

```
def _build_plan_template(task_type: str) -> str:
    lines = [
        "# [PLAN] TYPE=stdin",
        "# [PLAN] INPUT=...",
        ...
    ]
```

3. t4：改为 `[CHECK]` 布尔检查（`solution_t4.py`）

```
def _build_check_template(task_type: str, fix_target: str,
repeated_same_error: bool) -> str:
    lines = [
        "# [CHECK] TYPE=stdin",
        f"# [CHECK] FIX_TARGET={fix_target}",
        "# [CHECK] IO_EXACT=True",
        ...
    ]
```

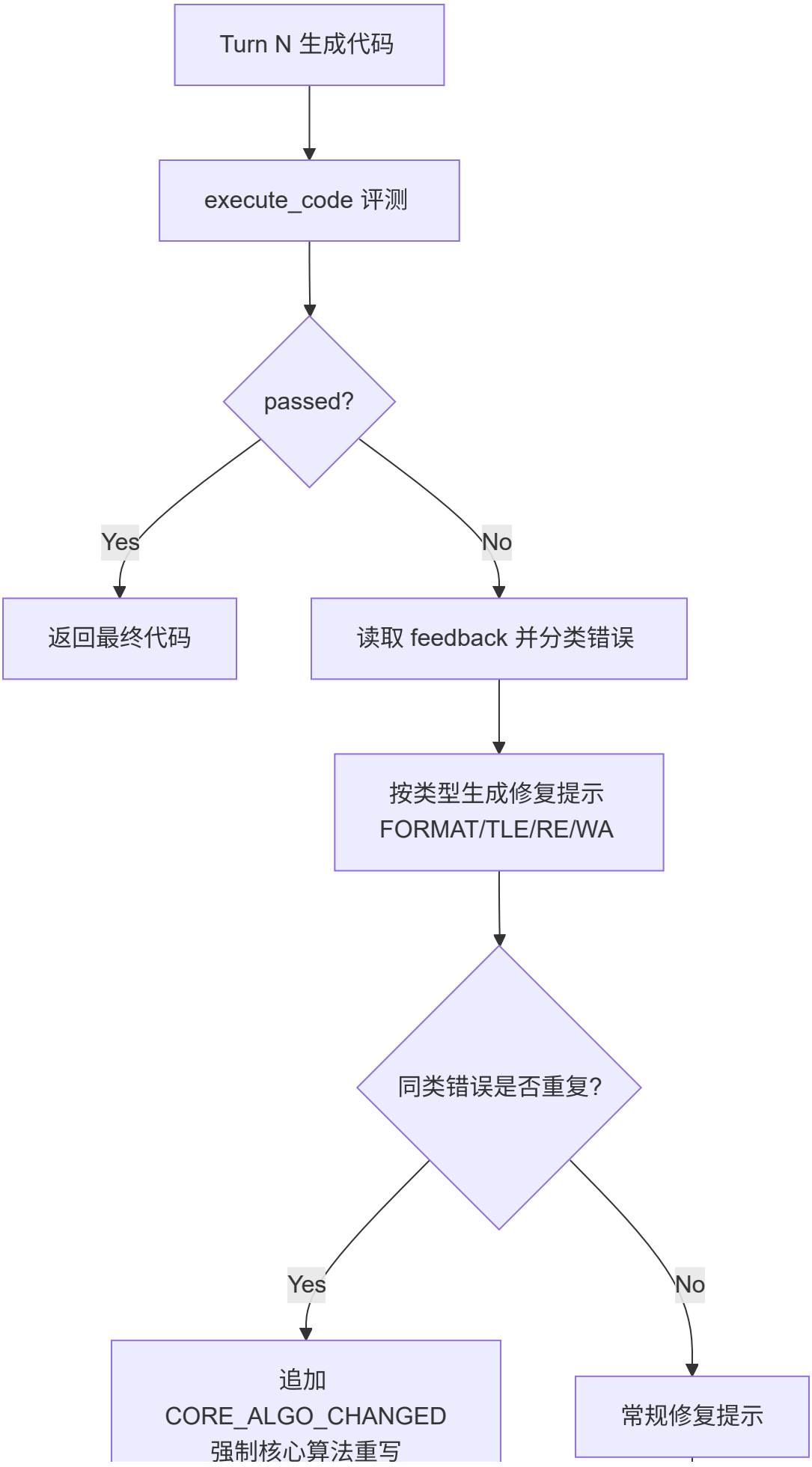
4. t5：加入 failure guards 与修复说明（`solution_t5.py`）

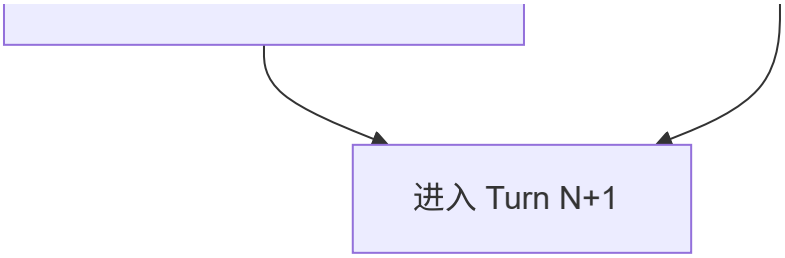
```
def _build_failure_guards(task_type: str) -> str:
    return (
        "1. Mapping/index tasks ...\n"
        "2. Process-simulation tasks ...\n"
        "3. Group/tie tasks ...\n"
        "4. Large N,Q queries ..."
    )
```

```
"Immediately after [CHECK] header, add one short line: # [FIX] <previous bug>
-> <current fix>"
```

2.3 错误判断与修复分支

这一模块的设计目标是把“失败反馈”转成“可执行的下一步动作”，避免重试轮变成重复采样。具体流程是：模型先生成候选代码，评测器返回 `passed/feedback`，系统根据反馈文本做错误类型识别（FORMAT/TLE/RE/WA），然后为不同错误类型注入不同的修复焦点；如果同一种错误重复出现，则升级为“核心算法重写”而非局部修补。该机制与现有关于 LLM 代码错误模式、系统性失效和修复策略的研究结论一致[1-6]：错误往往不是随机噪声，而是可归类、可诊断、可针对性修复的结构化失效。


Turn N 生成代码
execute_code 评测
passed?
Yes
No
返回最终代码
读取 feedback 并分类错误
按类型生成修复提示
FORMAT/TLE/RE/WA
同类错误是否重复?
Yes
No
追加
CORE_ALGO_CHANGED
强制核心算法重写
常规修复提示



2.3.1 错误判断（t1 起引入并持续保留）

表 6 错误类型判定与修复焦点

错误类型	判定依据（反馈关键词）	修复重点
FORMAT	`incorrect format`	强制单代码块输出、格式对齐
TLE	`time limit exceeded` / `time out`	更换低复杂度算法，不做常数级修补
RE	`runtime error` / `error:`	修复解析、边界、索引、状态
WA	其他未通过场景	回到题意语义重建算法

2.3.2 修复轮逻辑（所有版本共性）

修复轮的本质不是“再问一次”，而是“带着诊断结论再问一次”。实现上，每次失败都会把上一轮代码作为 assistant 历史写回对话，并在 user 侧附上压缩后的反馈与针对该错误类型的修复约束。这样下一轮不是从零开始，而是基于“失败证据”做定向修复。当系统检测到同类错误再次出现时，会额外施加核心算法改写信号（如 `CORE_ALGO_CHANGED`），强制模型停止局部打补丁。若到最后一轮仍未完全通过，系统返回最优候选 `best_response`，保证输出始终是可执行代码，便于后续离线分析与下一版提示词迭代。

2.4 教师模型参考如何落到提示词

这里是本项目最关键的“对齐链路”：不是直接抄答案，而是提炼教师模型的成功模式并转译为约束。

2.4.1 差距驱动（baseline_t5 vs plus_t2）

表 7 高差距题驱动的提示词改造方向

高差距题	baseline_t5	plus_t2	触发的提示词改造方向
qid=3	0.0000	1.0000	强化输入建模与语义映射一致性
qid=11	0.0000	1.0000	强化“输出域正确性 + 映射方向”
qid=13	0.1250	0.4375	强化分组/并列处理与复杂度护栏
qid=43	0.0000	0.1875	强化大输入场景的稳健解析

高差距题	baseline_t5	plus_t2	触发的提示词改造方向
qid=64	0.0000	0.1875	强化 query 场景避免 per-query 重建

2.4.2 对齐落地方式

1. 从教师优势题中提炼“共性风格”：I/O 严谨、语义不偷换、复杂度先行。
2. 在提示词中转写为可执行条目（`Task constraints`、`Failure guards`、`[CHECK]`）。
3. 在 retry 轮加入错误类型 focus，使修复方向和教师优势行为一致。

2.4.3 与“空泛优化”的区别

- 不是“多写提示词”，而是“把教师优势行为编码成检查项”。
- 不是“一次性大改”，而是“版本化 + 指标回看 + 反向修正”。

三、总结

3.1 baseline 链路总结

表 8 baseline 链路总体结果

版本	pass@1	passed/320	相对上一版
baseline	0.418750	134	-
baseline_t1	0.400000	128	-0.018750
baseline_t2	0.437500	140	+0.037500
baseline_t3	0.421875	135	-0.015625
baseline_t4	0.443750	142	+0.021875
baseline_t5	0.453125	145	+0.009375

图 5 baseline→t5 的平均 pass@1 趋势图

Figure 3. Baseline to T5 Stage Line Chart

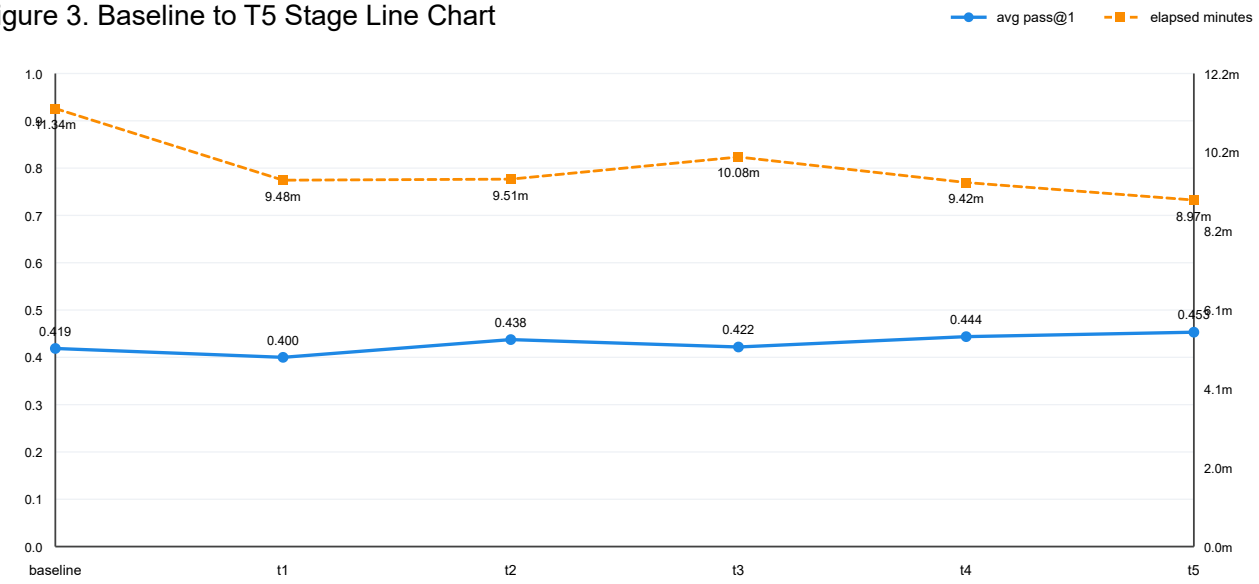
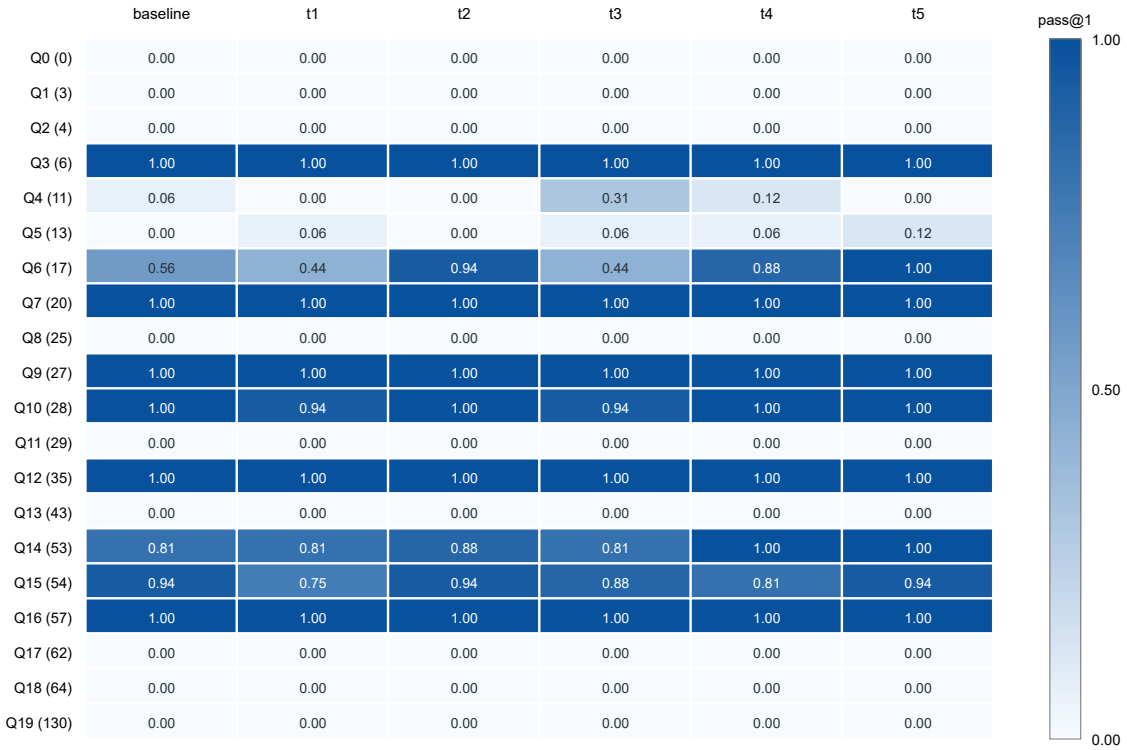


图 6 baseline 链路分题通过率热力图

Figure 4. Baseline Chain Heatmap (Q x Version)



图注：图 5 展示平均指标在版本迭代中的总体趋势，图 6 展示分题层面的结构性变化。两图共同用于识别"总体提升是否来自关键题修复"。

由图 5 与图 6 可见，baseline 链路的收益并非均匀扩散，而是集中在少数高收益题上，说明“基于差距题定向约束”的策略是有效的。

结论：

1. 最优版本为 t5，相对 baseline 提升 `+0.034375`。
2. 提升并非线性，t1 与 t3 回退说明“过重结构化约束”会损害 8B 执行质量。
3. 真实有效路径是：轻量结构化 + 错误分流 + 失败护栏。

baseline 最终结果：0.418750 → 0.453125（+8.21% 相对提升）

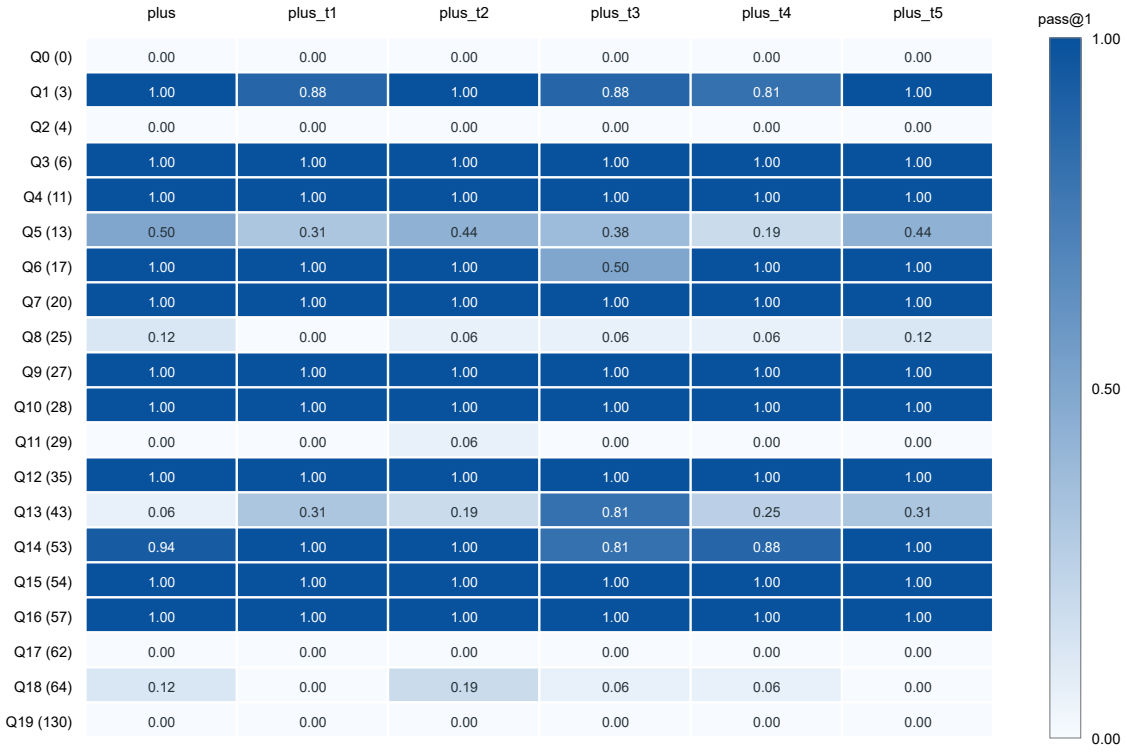
3.2 教师模型链路总结

表 9 教师模型（coder_plus）链路结果

版本	pass@1	passed/320	相对上一版
qwen3_coder_plus	0.587500	188	-
qwen3_coder_plus_t1	0.575000	184	-0.012500
qwen3_coder_plus_t2	0.596875	191	+0.021875
qwen3_coder_plus_t3	0.575000	184	-0.021875
qwen3_coder_plus_t4	0.562500	180	-0.012500

图 7 qwen3_coder_plus 链路分题通过率热力图

Figure 5. Plus Chain Heatmap (Q x Version)



图注：图 7 展示教师模型在 t1~t4 中的分题稳定性变化，用于验证提示词策略是否具有跨模型一致性。

图 7 的价值在于说明：即便是强模型，也会因提示词结构变化出现局部回退，因此基座侧优化必须坚持“短约束、强执行、可验证”的设计原则。

结论：

1. 教师侧最优同样出现在 t2，和基座侧趋势一致。
2. t3 的强 `[PLAN]` 在教师模型上也出现明显副作用，进一步证明“结构化不等于有效”。

3.3 结论与下一步

最终结论：

1. 本项目验证了“基于教师行为的提示词对齐”是有效路线，且具备可复用性。
2. 当前收益主要来自 `stdin`，功能题（Q19）仍是短板。
3. 下一阶段应做“高 gap 题专项模板化”而非继续加长通用提示词。

研究主结论：提示词工程能够在不修改模型参数的条件下，实现基座模型向教师模型的有效能力对齐。

建议下一步（按收益优先）：

1. 以 `qid=3`、`qid=11` 为核心做专题规则（输入建模 + 映射方向 + 输出域）。
2. 对 `functional` 分支单独强化签名/返回契约模板。
3. 把教师模型通过样本中高频代码习惯继续沉淀为短 checklist（避免长自然语言）。

参考文献

[1] Tambon F, Chen X, Panichella S, et al. Bugs in large language models generated code: An empirical study[J]. arXiv preprint arXiv:2403.08937, 2024.

[2] Wang Z, Li Y, Jiang Y, et al. Towards understanding the characteristics of code generation errors[C]//Proceedings of the International Conference on Software Engineering (ICSE). IEEE, 2025.

[3] Dou Y, Zhang H, Li Y, et al. What is wrong with your code generated by large language models?[J]. Science China Information Sciences, 2026.

[4] Chen Y, Wang T, Li J, et al. Fixing code generation errors for large language models[J]. arXiv preprint arXiv:2409.00676, 2024.

[5] Liu S, Zhang Q, Zhao X, et al. Chain of targeted verification questions to improve code reliability in large language models[J]. arXiv preprint arXiv:2405.13932, 2024.

[6] Zhou K, Wang R, Chen L, et al. Uncovering systematic failures of large language models in verifying code correctness[J]. arXiv preprint arXiv:2508.12358, 2025.

[7] Chen M, Tworek J, Jun H, et al. Evaluating large language models trained on code[C]//Advances in Neural Information Processing Systems (NeurIPS). 2021.

[8] Austin J, Odena A, Nye M, et al. Program synthesis with large language models[J]. arXiv preprint arXiv:2108.07732, 2021.

[9] Hendrycks D, Basart S, Kadavath S, et al. Measuring coding challenge competence with APPS[J]. Advances in Neural Information Processing Systems (NeurIPS), 2021.

[10] Li Y, Jiang Y, Wang Z, et al. LBPP: A harder benchmark for code generation models[C]//Findings of the Association for Computational Linguistics: EMNLP. 2024.

附录 A：全量分题矩阵（baseline 与 plus）

A.1 baseline → t5

Q	qid	type	baseline	t1	t2	t3	t4	t5
0	0	stdin	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
1	3	stdin	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
2	4	stdin	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
3	6	stdin	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
4	11	stdin	0.0625	0.0000	0.0000	0.3125	0.1250	0.0000
5	13	stdin	0.0000	0.0625	0.0000	0.0625	0.0625	0.1250
6	17	stdin	0.5625	0.4375	0.9375	0.4375	0.8750	1.0000
7	20	stdin	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
8	25	stdin	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
9	27	stdin	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
10	28	stdin	1.0000	0.9375	1.0000	0.9375	1.0000	1.0000
11	29	stdin	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
12	35	stdin	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
13	43	stdin	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
14	53	stdin	0.8125	0.8125	0.8750	0.8125	1.0000	1.0000
15	54	stdin	0.9375	0.7500	0.9375	0.8750	0.8125	0.9375
16	57	stdin	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
17	62	stdin	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
18	64	stdin	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
19	130	functional	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

A.2 qwen3_coder_plus → t4

Q	qid	type	plus	plus_t1	plus_t2	plus_t3	plus_t4
0	0	stdin	0.0000	0.0000	0.0000	0.0000	0.0000
1	3	stdin	1.0000	0.8750	1.0000	0.8750	0.8125
2	4	stdin	0.0000	0.0000	0.0000	0.0000	0.0000
3	6	stdin	1.0000	1.0000	1.0000	1.0000	1.0000
4	11	stdin	1.0000	1.0000	1.0000	1.0000	1.0000
5	13	stdin	0.5000	0.3125	0.4375	0.3750	0.1875

Q	qid	type	plus	plus_t1	plus_t2	plus_t3	plus_t4
6	17	stdin	1.0000	1.0000	1.0000	0.5000	1.0000
7	20	stdin	1.0000	1.0000	1.0000	1.0000	1.0000
8	25	stdin	0.1250	0.0000	0.0625	0.0625	0.0625
9	27	stdin	1.0000	1.0000	1.0000	1.0000	1.0000
10	28	stdin	1.0000	1.0000	1.0000	1.0000	1.0000
11	29	stdin	0.0000	0.0000	0.0625	0.0000	0.0000
12	35	stdin	1.0000	1.0000	1.0000	1.0000	1.0000
13	43	stdin	0.0625	0.3125	0.1875	0.8125	0.2500
14	53	stdin	0.9375	1.0000	1.0000	0.8125	0.8750
15	54	stdin	1.0000	1.0000	1.0000	1.0000	1.0000
16	57	stdin	1.0000	1.0000	1.0000	1.0000	1.0000
17	62	stdin	0.0000	0.0000	0.0000	0.0000	0.0000
18	64	stdin	0.1250	0.0000	0.1875	0.0625	0.0625
19	130	functional	0.0000	0.0000	0.0000	0.0000	0.0000

附录 B：加速方法

1. Linux 环境通常比 Windows 更快，尤其在多进程调度与 I/O 上。
2. tokenizer 本地化缓存可减少重复加载开销。
3. 进程数需结合 CPU 核心与服务端并发能力联合调优，避免 API error。