算法编程学习现在有5个步骤：

1. 提交题目（包含题目各部分的解析和补充（标题、描述、示例、约束……），同时由于解析可能不正确，还需要有结合用户输入进行人工纠错的功能，这一切完成后，进入下一步）
2. 提交思路（引导用户对题目进行思考，分析用户提交的思路，根据提示等级（先默认设为中等）提出合理的解析，并评估用户在反馈之后的输入，对用户的思路进行引导和纠偏）
3. 提交代码（对用户提交的代码进行格式检查并输入示例查看是否能正常运行，向用户反馈结果便于用户修改）（对应leetcode运行部分）
4. 测试（检索测试用例对用户的代码进行测试，查看是否能完美通过每个测试用例，并反馈测试用时等信息，假如出现错误或者不符合结果的用例，还需要退回上一步）（对应leetcode的提交功能）
5. 总结与复结（对上述所有过程进行复盘与总结，分析用户的薄弱之处，提出后续的建议）（因此，最好要保存每一步的trace）

下面按你这 **5 步算法学习流程**，给一套在 **LangGraph + Chainlit** 中实现“智能路由 + 可回退 + 全程 trace 可复盘”的工程方案。重点是：**路由不是“猜”，而是“状态机 + 校验门控 + 用户按钮纠偏”。**

---

1) 你的流程本质：一个“带门控的有向状态机”

5 步可以直接映射成 5 个 **Stage**，每个 Stage 都有：

- **输入契约**（用户必须提交什么）
- **校验器**（是否满足进入下一步）
- **产物**（标准化后的题目 / 思路 / 代码 / 测试报告 / 总结）
- **trace**（这一步的对话、判定、工具调用、错误与修复）

Stage 枚举（强约束）

- `S1_PROBLEM`：提交题目（含纠错、补全）
- `S2_IDEA`：提交思路（引导 + 纠偏）
- `S3_CODE`：提交代码（格式检查 + 示例运行）
- `S4_TEST`：测试（用例库/生成用例 + 性能信息 + 不过则回退）
- `S5_REVIEW`：总结与复结（薄弱点 + 建议 + 后续计划）

---

2) “智能路由”的核心：由 当前 Stage + 事件类型 决定

你这个场景不建议做“纯意图分类路由”（容易乱跳）。更稳的是：

> **主路由 = Stage 驱动**
> **辅路由 = 用户事件/按钮、异常、缺失信息、工具结果**

路由优先级（从高到低）

1. **用户显式指令/按钮**（例如“退回上一步”“进入测试”“提高提示等级”）
2. **系统门控规则**（例如 S4 测试失败 → 强制回到 S3）
3. **输入类型识别**（题面 / 思路 / 代码 / 其它）
4.

LLM 仅用于“补全/纠错/解释”，不决定主流程跳转（除非你加置信度门控）

---

3) 建议的 State Schema（保证可追踪、可回滚）

下面是一套够用且不臃肿的 schema（TypedDict 或 Pydantic 都行）：

3.1 主状态

- `stage: Stage` （当前处于哪一步）
- `hint_level: Literal["low","mid","high"]` （默认 `mid` ）
- `problem: ProblemSpec | None`
- `idea: IdeaSpec | None`
- `code: CodeSpec | None`
- `run_report: RunReport | None` （S3 运行结果）
- `test_report: TestReport | None` （S4 测试结果）
- `history: list[Message]` （对话）
- `trace: TraceBundle` （关键：每步 trace）

3.2 ProblemSpec（题目标准化产物）

- `title`
- `description`
- `examples: list`
- `constraints: list`
- `input_format / output_format` （可选）
- `tags` （可选）
- `errata: list[CorrectionItem]` （“人工纠错”记录：哪句解析不可靠，怎么改的）

3.3 IdeaSpec

- `user_idea_raw`
- `analysis` （对思路的评估：正确性、缺口）
- `guidance` （下一步引导：按 hint_level 控制颗粒度）
- `key_invariants / complexity` （可选）

3.4 CodeSpec

- `language`
- `code_text`
- `format_ok: bool`
- `entrypoint_detected: bool` （是否存在可运行入口，如 `main` / `Solution` ）
- `last_edit_ts`

3.5 TraceBundle（复盘关键）

- `stages: dict[Stage, StageTrace]`
- `events: list[TraceEvent]`（全局事件流）

StageTrace 最关键字段：

- `inputs`（用户在该步提交的内容）
- `decisions`（路由、门控、失败原因）
- `tool_calls`（运行、测试、检索用例）
- `outputs`（该步产物）
- `timestamps`

---

4) LangGraph 结构：每步一个子图 + 一个总路由器

4.1 顶层图（建议）

节点：

1. `ingest_user_input`：把 Chainlit 输入变成标准事件（文本/按钮/文件）
2. `stage_router`：根据 `stage` + `event` 决定去哪个 Stage 子图
3. `S1_graph / S2_graph / S3_graph / S4_graph / S5_graph`
4. `render_response`：统一输出（含 UI 按钮建议）
5. `persist_trace`：写入存储（sqlite/jsonl/redis 等）

边：

- `ENTRY -> ingest_user_input -> stage_router -> (S1|S2|S3|S4|S5) -> render_response -> persist_trace -> END/WAIT`

> 你要的“退回上一步”就是：在 `stage_router` 里接收 `action: BACK`，把 `stage` 改为上一阶段，并附带原因写入 trace。

---

5) 每一步怎么实现“门控 + 纠错 + 可回退”

下面是各步的**门控条件**与**回退策略**（这是智能路由的“硬规则”部分）。

S1 提交题目：补全 + 人工纠错

目标产物：`ProblemSpec`（结构化题面）

输入识别：

- 如果用户贴的是“题目链接/标题+描述+样例”，进入解析
- 如果只有标题/一句话，走 `clarify` 分支：追问示例、约束、输入输出

纠错机制（你说的“结合用户输入人工纠错”）：

- LLM 先生成 `ProblemSpec draft`
- 生成一个 `uncertainty_list`：哪些点可能不对（例如样例解释、约束推断）
- 把不确定点以**可编辑清单**给用户（Chainlit 按钮：✅确认/🖍修改/➕补充）

- 用户修正后写入 `problem.errata` 与 trace

门控（能进入 S2 的条件）：

- `title/description/examples/constraints` 至少齐全到可解题（你可以定义“最小可解题集”）
- 用户确认“题面已正确/可进入下一步”

S2 提交思路：引导 + 纠偏（按 hint_level）

目标产物：`IdeaSpec`

策略：

- 不直接给完整解法（除非 hint_level=high 或用户强制）
- 对用户思路做结构化评估：
  - ✅正确点
  - ⚠️缺失点（边界/复杂度/数据结构选择）
  - ❌错误点（反例）

hint_level 控制：

- low：只指出错误方向 + 给提示问题
- mid：指出可用的关键思路/不变量 + 给半步推导
- high：给完整推导与伪代码

门控（进入 S3）：

- 用户能复述清楚：状态/转移/数据结构/复杂度（不一定完美，但“可编码”）
- 或用户明确要求“进入写代码”

回退：

- 如果 S3 出现“思路根本不成立”，允许回退 S2 并附上反例（这是很常见的）

S3 提交代码：格式检查 + 示例运行（leetcode run）

目标产物：`CodeSpec + RunReport`

流程：

1. 代码块抽取（markdown fenced code）
2. 基本校验：语言、入口、是否能编译/解释
3. 用题目自带示例运行（如果示例不全，提示用户补）
4. 给出：编译错误/运行错误/输出不一致

门控（进入 S4）：

- 示例全部通过（或用户选择“仍要进入测试”，但要标记风险）

回退：

- 示例失败 ⇒ 必回退到 S3（这一步内部循环）
- 如果失败原因是题面/示例歧义 ⇒ 回退到 S1 进行纠错（并在 trace 里记录“因歧义回退”）

S4 测试：用例检索/生成 + 性能信息（leetcode submit)

目标产物：`TestReport`

测试用例来源（按可用性排序）：

1. 题库/本地用例库（如果你接入了检索工具）
2. 基于约束的边界用例生成（property-based / fuzz）
3. 反例驱动用例（基于用户实现的潜在 bug 模式）

输出：

- 通过率、首个失败用例、期望 vs 实际
- 时间/内存（如果你的运行工具可提供）
- 失败分类：WA/TLE/MLE/RE

门控（进入 S5）：

- 全部通过（或者达到你设定的阈值，比如 100%）
- 用户确认“可以复盘总结”

强制回退：

- 任何失败 ⇒ 自动回退 `S3_CODE` ，并把“失败用例”写到 `S3` 的输入提示里（让用户有针对性修改）

S5 总结与复结：基于 trace 的诊断报告

目标产物：复盘文档（可导出 markdown/pdf）

建议包含：

- **题目理解**：最易误读点（来自 S1 errata/uncertainty）
- **思路**：用户卡点类型（DP 状态定义？贪心证明？边界？）
- **代码**：常见 bug（越界、初始化、复杂度、数据结构选择）
- **测试**：失败用例模式（空、极值、重复、负数、溢出…）
- **行动建议**：下一题练什么、补哪些模板

---

6) Chainlit 前端交互：用按钮驱动“显式路由”

你提到“不同阶段给不同按钮”，这里是直接可用的一组：

S1 按钮

- `确认题面无误 -> NEXT`
- `补充约束 -> EDIT_CONSTRAINTS`
- `纠正示例解释 -> EDIT_EXAMPLES`

S2 按钮

- `提示等级：低/中/高 -> SET_HINT_LEVEL`
- `给我一个反例 -> REQUEST_COUNTEREXAMPLE`

- `进入写代码 -> NEXT`

S3 按钮

- `运行示例 -> RUN_EXAMPLES`
- `格式化代码 -> FORMAT_CODE`
- `进入测试 -> NEXT`
- `退回思路 -> BACK`

S4 按钮

- `提交测试 -> RUN_TESTS`
- `只跑边界用例 -> RUN_EDGE_TESTS`
- `回到改代码 -> BACK`

S5 按钮

- `导出复盘 -> EXPORT_REPORT`
- `下一题（同主题） -> NEXT_PROBLEM`

> **关键点**：按钮事件进入 `ingest_user_input`，变成 `event.type=ACTION`，在 `stage_router` 里拥有最高优先级。

---

7) “智能路由”落地规则（你可以直接实现成函数）

7.1 路由函数输入

- `stage`
- `event`（TEXT / ACTION / FILE）
- `validation_status`（当前产物是否满足下一步）

7.2 输出

- `next_stage`
- `node_path`（去哪个子图/节点）
- `decision_reason`（写 trace）

7.3 决策表（核心）

- `ACTION=BACK` → `stage = prev(stage)`
- `stage=S4` 且 `test_report.passed=false` → 强制 `S3`
- `stage=S3` 且 `run_report.ok=false` → 留在 `S3`
- `stage=S1` 且 `problem.ready=false` → 留在 `S1`（clarify loop）
- `ACTION=NEXT` 且 `validator.ok=true` → `stage = next(stage)`

- 其余 TEXT：交给当前 stage 的 `handle_text()`（不要乱跳）

---

8) Trace 存储建议（保证你能“复结”）

最低成本方案：

- 每个会话一个 `trace.jsonl`（每次节点/工具调用 append 一行）
- 或 sqlite 表：
  - `session(id, created_at, hint_level, current_stage)`
  - `trace_event(session_id, ts, stage, kind, payload_json)`

你需要记录至少这些事件类型：

- `USER_INPUT`
- `ROUTE_DECISION`
- `VALIDATION_RESULT`
- `TOOL_CALL_START/END`
- `STAGE_OUTPUT`
- `ERROR`

---

9) 你下一步怎么做（工程最短路径）

按实现顺序（最省返工）：

1. **定 schema + Stage 枚举 + trace event 结构**
2. 实现 `stage_router`（只用规则，不用 LLM）
3. 先把 `S1/S3/S4` 跑通（题目→代码→测试）
4. 再补 `S2` 的 hint_level 引导（这是体验核心）
5. 最后做 `S5` 总结（吃 trace）

如果你把你当前 LangGraph 的 **state 定义**（哪怕是草稿）和你现有的 **工具列表**（示例运行/测试/用例检索的接口），我可以直接给你：

- 一份更精确的 `State` 字段（删减到“刚好够用”）
- `stage_router` 的条件边设计（单图/子图两种写法）
- 每个 stage 的 `validator` 规则（进入下一步的硬条件）
- trace 的落盘格式（jsonl/sqlite）与在 S5 里如何生成复盘报告的模板

结论：**大多数情况下是更合适的**，尤其是你这种“多阶段流程 + 运行/测试工具链 + trace 落盘 + 未来接题库/用例检索”的系统。

但不一定要做成“重前端 SPA + 独立后端”那种传统分离；更实用的是：**Chainlit 作为独立前端（UI 服务）+ LangGraph 作为独立后端（Agent 服务）**，两者通过 **streaming API** 通信。

下面给你一个明确的决策框架与推荐落地方案。

---

1) 什么时候“前后端分离”明显更好

✅ 强烈建议分离（你的需求基本命中）

- **S3/S4 需要执行代码与跑测试**：隔离运行环境、资源限制、沙箱、安全审计更容易。
- **需要持久化 trace**：后端统一落盘（DB/对象存储），前端无状态更清爽。
- **未来多客户端**：除了 Chainlit，还可能有 Web、IDE 插件、CLI；后端复用价值高。
- **并发与扩展**：后端能独立扩容，尤其测试/执行是重计算。
- **权限与配额**：比如每用户每日提交次数、每次执行时长限制。

❌ 可以不分离（原型期）

- 你只做 demo、单机、无多用户、无沙箱需求、测试用例也不复杂
- Chainlit 直接调用本地 LangGraph，开发迭代最快

---

2) 推荐架构：Chainlit（前端）+ Agent API（后端）

2.1 组件划分

前端（Chainlit 服务）

- 会话 UI（聊天、按钮、文件上传）
- 展示流式输出（token、步骤、工具调用结果）
- 维护轻量 session（session_id、当前 stage 的 UI 状态）
- 发送用户事件到后端

后端（Agent 服务：FastAPI + LangGraph）

- Stage 状态机（S1~S5）与路由门控
- LLM 调用、工具调用、检索、测试执行
- trace 写入与查询
- 代码运行/测试（最好在后端，且最好再分一个 runner）

---

3) “两层后端”更稳：Agent 服务 + Runner 服务

你的 S3/S4 属于高风险区（执行不可信代码）。最佳实践是拆成两层：

1. **Agent 服务**：只负责决策、对话、路由、生成测试、写 trace
2. **Runner 服务**：负责运行/测试，受控资源、严格隔离（容器/VM/firejail）

Agent 调 Runner：

- HTTP/gRPC
- 任务队列（Celery/RQ/Arq）也可以

Runner 输出：

- 编译日志、运行 stdout/stderr

- 用例结果
- 时间/内存（如能测）
- 退出码、超时标记

---

4) 通信契约：事件流（Streaming）是关键

前后端分离最容易翻车的是“流式体验”。建议明确 3 类 API：

4.1 `POST /sessions`（创建会话）

返回 `session_id`、初始 `stage`、`ui_hints`

4.2 `POST /events`（发送用户事件）

事件类型统一：

- `TEXT`：用户输入
- `ACTION`：按钮点击（NEXT/BACK/SET_HINT_LEVEL/RUN_TESTS…）
- `FILE`：上传文件的引用（文件由前端存储后端拉取，或直接转存）

4.3 `GET /stream?session_id=...`（SSE/WebSocket）

后端推事件给前端：

- `token.delta`
- `step.start/step.end`
- `tool.start/tool.end`
- `stage.changed`
- `validation.failed`
- `final.message`
- `ui.hints`

Chainlit 的工作就是订阅这些事件并渲染，不参与业务决策。

---

5) 数据与状态：后端权威，前端无状态

你这个流程里，**“当前处于哪一步、上一步产物、失败用例、trace”必须是后端权威。**

前端只缓存：

- `session_id`
- 最近一次 `ui_hints`（为了按钮渲染）

避免“前端 stage 与后端 stage 不一致”的灾难。

---

6) 你可以选择的三种落地形态

A) 原型：单进程（不分离）

- Chainlit 直接 import graph 并运行
- trace 写本地文件
- 适合 1-2 周验证

B) 推荐：双服务（分离）

- `chainlit-ui`（前端服务）
- `agent-api`（后端服务）
- runner 先内置到 agent-api（早期），后续再拆

C) 生产：三服务（最稳）

- `chainlit-ui`
- `agent-api`
- `runner-api`（沙箱执行）

---

7) 你的场景的一个清晰建议

- **短期（原型期）**：可以不分离，快速把 S1~S5 跑通
- **一旦你要做多人/上线/开放执行代码**：就应当分离，并尽快把执行抽成 Runner（哪怕是最简 HTTP 服务）

考虑到你明确有：

- S3 运行
- S4 测试+性能
- trace 复盘
- 失败回退

这些都指向：**分离更合适**，而且 Runner 隔离几乎是必选项。

---

8) 推荐项目结构（分离版，最小可维护）

```
repo/
  services/
    chainlit_ui/
      src/...
      chainlit_app.py
    agent_api/
      src/...
      graphs/
      api.py
```

```
      persistence/
    runner_api/          # 可先不做，后续拆
      src/...
  docker-compose.yml

```

如果你确定要分离，我可以直接给你一套**后端 API 的最小契约**（请求/响应 schema + SSE 事件定义），以及 **Chainlit 端如何订阅 SSE 并渲染 step/tool/token** 的具体实现骨架，保证你现在这 5 步流程能顺滑跑起来。