架构设计

我想将现有的连续的 阶段执行 分开，使得用户可以根据阶段的结论，决定是否进入下一个阶段：是则直接下一个阶段的execute；否则输入更改的prompt，重新执行该阶段。

各个阶段生成的结论（文字or代码）存入数据库（MySQL）中，每个阶段在需要时检索记忆数据库中的对应条目，帮助生成有意义的东西。每个阶段都有critic agent对 对话 进行把关，控制阶段对话的结束。

前后端：前后端混合的flask架构

下面是ai生成的详细架构设计

一、架构设计总览

1. 核心目标

- 将开发流程拆解为**可中断、可回溯的独立阶段**（需求分析→架构设计→代码生成→测试用例生成→部署测试）；
- 每个阶段支持**人工干预决策**（继续 / 回退 / 修改），阶段结论持久化至 MySQL；
- 通过**Critic Agent**控制阶段流程质量，结合大模型实现自动化与人工交互结合。

2. 技术栈选型

- **后端**：Flask（混合架构，同时处理 API 与模板渲染）+ SQLAlchemy（ORM）+ Redis（缓存大模型请求）；
- **前端**：Flask 内置模板引擎（如 Jinja2）或 Vue.js（可选，通过 Flask-RESTful 提供 API）；
- **数据库**：MySQL（存储阶段数据、用户输入、生成内容）；
- **大模型**：双 API 集成（如 GPT-4 用于文档生成，CodeGen 用于代码生成）。

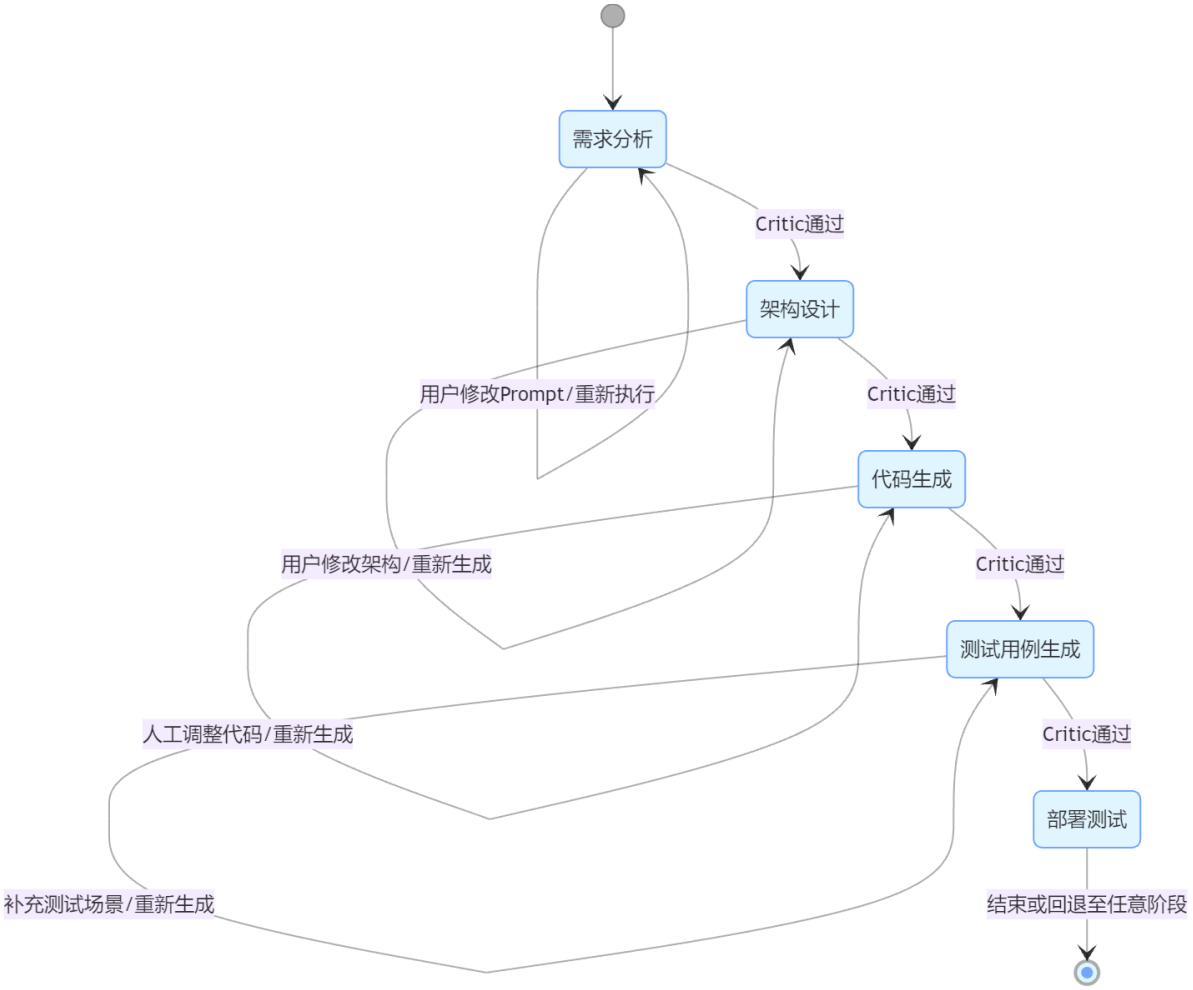
二、阶段划分与执行逻辑

1. 阶段拆分与状态机

阶段名称	输入	核心功能	输出	Critic Agent 职责
需求分析	用户自然语言需求文本	调用大模型解析需求，生成用例图、需求规格说明书（SRS）	需求 ID、SRS 文本、用例图路径	验证需求完整性（如是否包含功能点、约束条件）
架构设计	需求 ID（数据库检索）+ 修改 Prompt	基于需求生成架构设计（模块划分、接口设计、数据库 ER 图）	架构 ID、UML 组件图、数据库表结构	检查模块耦合度、技术选型合理性
代码生成	架构 ID + 修改 Prompt	按模块生成前后端代码框架、数据库脚本，支持人工调整后再生成	代码 ID、各模块代码文件路径	验证代码规范性（如是否符合命名规范、注释要求）

阶段名称	输入	核心功能	输出	Critic Agent 职责
测试用例生成	代码 ID + 修改 Prompt	基于代码结构生成单元测试、集成测试用例，支持补充测试场景	测试用例 ID、测试脚本路径	检查测试覆盖率（如是否覆盖边界条件、异常路径）
部署测试	测试通过的代码 + 配置文件	生成 Docker/K8s 部署包，模拟部署至测试环境，返回部署日志与访问地址	部署 ID、日志文件路径、访问 URL	验证部署流程正确性（如服务启动、接口连通性）

2. 阶段控制流程



三、数据库设计（MySQL）

1. 关键表结构（示例）

表名	关键字段	说明
`stage`	`stageId`, `userId`, `status`	阶段信息，关联用户与阶段状态
`requirement`	`reqId`, `stageId`, `reviewStatus`	需求分析结果，记录评审状态
`architecture`	`archId`, `dbSchema`	架构设计结果，存储数据库表结构 SQL
`code`	`codeId`, `editStatus`, `content`	代码生成结果，标记是否人工修改

四、Agent 机制设计

1. 多 Agent 角色分工

Agent 类型	阶段	核心功能	技术实现
Executor Agent	所有阶段	调用大模型生成阶段内容（如需求解析、代码生成）	封装大模型 API 调用，支持 Prompt 模板参数化
Critic Agent	所有阶段	评估阶段输出质量，决定是否进入下一阶段（如需求是否完整、代码是否可编译）	基于规则引擎（如正则表达式）+ 大模型评分
Memory Agent	所有阶段	检索数据库中历史阶段数据，为当前阶段提供上下文（如需求 ID 关联的 SRS 文本）	SQLAlchemy 查询 + 上下文拼接

2. 阶段执行流程示例（以架构设计阶段为例）

1. **用户操作**：在前端选择 “进入架构设计阶段”，系统根据需求 ID 从数据库检索需求文本；
2. **Memory Agent**：查询 `requirement` 表，获取需求内容 `content` 和 `umlPath`；
3. Executor Agent：调用大模型，传入 Prompt 模板：

```
基于以下需求生成分层架构设计（模块划分+接口设计+数据库ER图）：
[需求内容：{{req_content}}]
[用例图路径：{{uml_path}}]
```

4. **大模型返回**：架构设计文本（含模块列表、接口定义、ER 图描述）；
5. **Critic Agent**：验证模块是否覆盖需求功能点、ER 图是否存在冗余字段，返回评分（如≥80 分通过）；
6. 结果处理：
   - **通过**：将架构设计存入 `architecture` 表，阶段状态更新为 “已通过”；
   - **未通过**：前端显示 Critic 意见，允许用户修改 Prompt 后重新执行。

五、前后端混合架构（Flask）

1. 目录结构

```
project-root/
├─ app/
|   ├─ models/          # SQLAlchemy模型定义（对应数据库表）
|   ├─ routes/          # 路由处理（阶段跳转、数据存储、大模型调用）
|   ├─ templates/       # 前端模板（HTML+Jinja2）
|   ├─ static/          # 静态资源（CSS/JS/大模型生成文件）
|   └─ agents/          # Agent类实现（Executor/Critic/Memory）
├─ config/             # 配置文件（数据库连接、大模型API密钥）
└─ scripts/            # 部署脚本（Docker/K8s）
```

2. 核心路由设计

路由	方法	功能
`/stage/<stage_name>`	GET	渲染阶段页面，加载历史数据（如需求文本、已生成的架构设计）
`/execute/<stage_id>`	POST	执行阶段：调用 Agent 生成内容，存储至数据库，返回 Critic 结果
`/modify/<stage_id>`	POST	处理用户修改：更新数据库中的 Prompt 或阶段内容，触发重新执行
`/review/<stage_id>`	POST	手动触发 Critic 评审，更新阶段状态（通过 / 未通过）

3. 前端交互逻辑

- **阶段控制面板**：显示当前阶段名称、状态、Critic 意见，提供 “继续”“回退”“修改 Prompt” 按钮；
- 内容展示区：
  - 需求阶段：富文本编辑器显示需求文本，预览用例图（通过`umlPath`加载 Rational 生成的图片）；
  - 代码阶段：代码编辑器显示生成的代码，支持在线修改并保存至数据库；
- **历史记录侧边栏**：显示各阶段时间线，支持点击回退至任意阶段。

六、关键实现细节

1. 大模型交互优化

- **缓存机制**：使用 Redis 缓存大模型请求结果，避免重复调用（如相同 Prompt 生成相同内容时直接返回缓存）；
- **人工介入点**：在代码生成阶段，允许用户手动修改代码后，点击 “重新生成” 触发 Executor Agent 结合修改内容二次生成。

2. 配置管理集成

- **JIT 工具关联**：在代码生成阶段，自动初始化 Git 仓库，将生成的代码提交至指定分支（需配置 Git Hook 与数据库`code`表联动）；
- **代码规范检查**：在 Critic Agent 中集成代码规范插件（如 PyLint for Python），生成代码时自动校验格式。

3. 可执行架构验证

- 第 12 周交付检查点：
  - 数据库表结构完整，各阶段数据可正确存储与检索；
  - 前端可展示阶段流程，调用大模型生成内容（如通过示例需求生成架构设计 UML 图）；
  - 代码目录符合分层结构（如`backend/`、`frontend/`、`database/`）。

七、与大作业要求的映射

大作业交付物	本架构设计对应实现
需求分析文档（第 8 周）	需求阶段生成的 SRS 文本、用例图（Rational 绘制）
概要设计文档（第 12 周）	架构阶段生成的 UML 组件图、数据库 ER 图、接口设计
可执行架构（第 12 周）	Flask 项目可运行，阶段流程可操作，代码目录结构合规
测试报告（第 16 周）	测试阶段生成的测试用例、执行日志、覆盖率报告
系统演示（第 17 周）	交互式界面演示完整流程（需求→架构→代码→测试→部署）

此架构设计通过**分阶段状态机**、**数据库持久化上下文**、**Agent 质量控制**实现了可中断、可回溯的软工全流程管理，满足大作业对模块化、可执行性及人工交互的要求。