本地代码仓的小型项目的自动化训练数据生成系统 提交版

目录

1. 背景与目标

系统概述

- 本设计文档旨在构建一套面向本地代码仓的小型项目的自动化训练数据生成系统，用于支持专有模型在以下能力上的训练：
  - 基于真实代码上下文的代码理解与问答能力
  - 基于已有架构的需求驱动式系统设计与方案生成能力
- 与通用代码数据集不同，本系统通过解析逐个**组件（Components）**运行，强调：
  - 业务逻辑与代码绑定：通过结合代码仓README.md重新从业务角度理解docstring，将组件的docstring描述转换成业务语言。
  - 依据证据链的生成内容：带Context（代码仓内部）和WIKI（外部查询）的内容反复多次（可设置最大循环次数）理解组件，生成的数据有证据链可依赖。
  - 递归语义聚合：采用自下而上的方式，从原子函数到文件，再到模块和全局架构，逐级构建代码仓知识图谱。
  - 可持续增量生成：主程序脚本main.py意外中断后，重新运行会检测已经处理的组件，生成数据（`output/`）及日志都采用追加模式，避免已生成内容重复执行。
  - 中心化 Agent 编排：基于 Orchestrator 模式，实现 Reader（分析）、Searcher（检索）、Writer（生成）的协同闭环。

系统总体架构

Code block

```
RepoGen
  └── agent
        ├── llm # 包含主流LLM接口类
        ├── base.py # BaseAgent基类
        ├── readmefilter.py # Readme内容过滤Github仓库Agent
        ├── task # 完成生成任务的Agents系统
            ├── orchestrator.py # 组织r-s-w逻辑Agent
            ├── reader.py # 阅读理解Agent
            ├── searcher.py # 查询WIKI和下载仓库(单一)Agent
            └──  writer.py # 任务执行Agent
        └── wiki # 用于理解Repo生成WIKI.md的Agents系统
            ├── build_repo_wiki.py # 单独直接构造Pipeline
            ├── recursive_system.py # 递归方式实现Pipeline
            ├── state.py # 共享agents状态
            └── agents
                ├── context_manager.py # readme摘要Agent
                ├── atomic_analyzer.py # docstring理解Agent
                ├── architect.py # 架构理解Agent
                └── wiki_builder.py # 合成WIKI.md Agent
  └── data_process
        ├── repo_downloader.py # 按照过滤要求下载Github仓库
        └── repo_tree.py # 生成repo文件目录结构
  └── dependency_analyzer
        ├── ast_parser.py # AST解析代码仓，生成组件
        ├── filter_components_by_cis.py # 根据图结构过滤重要性，保留前 60-80% 的重要的组件，可覆盖原dependency_graph.json
        └── topo_sort.py # 将组件构成图
  └── visualizer # task Agents Pipeline 处理组件流程可视化
  └── fiter_readme.py # 离线测试 agent/readmefilter.py是否正确
  └── generate_wiki # wiki Agents Pipeline 入口
  └── run_ast_parser.py # 生成dependency_graph.json
  └── run_repo_tree.py # 生成 repo_tree.json
  └── main.py # task agents pipeline 入口
```

2. 训练数据设计

- 主要路径

Code block

```
repo_dir: "data/raw_test_repo"
out_dir: "data/meta_test_repo"

repo_tree_dir: "data/meta_test_repo/repo_tree.json"
repo_dependency_dir: "data/meta_test_repo/dependency_graph.json"
repo_docstrings_dir: "data/meta_test_repo/dependency_graph.json"
```

```
repo_readme_dir: "data/raw_test_repo/README.md"

wiki_dir: "data/meta_test_repo/repo_wiki"

rag_dir: "data/meta_test_repo/repo_wiki/WIKI.md"
qa_out_dir: "output/qa_datasets.jsonl"
design_out_dir: "output/design_datasets.jsonl"

```

元数据

- 通过执行 `data/process/repo_downloader.py`，从Github上下载按照过滤条件筛选的代码仓，构成训练原始数据。逐一将代码仓放入 `data/` （后续步骤以data/raw_test_repo为示例）。
- 通过执行 `run_ast_parser.py`， `run_repo_tree.py` 分别生成**依赖图**和**目录树**的json文件，获取**组件**（**Components**）及其依赖信息、代码仓文件目录结构。
- （可选）通过执行 `filter_components_by_cis.py`，按照图统计特征（出入度，连通性等指标）筛选出60%-80%重要组件，忽略20%-40%辅助性组件，代替 `run_ast_parser.py` 使用。
- 元数据存储于 `data/meta_test_repo/` 路径下，包括：
  - Readme: `data/raw_test_repo/raw_test_repo/README.md`
  - dependency_graph.json: 单条 `models.product.Item` 组件，内容如下：

    Code block

    ```
    "models.product.Item": {
      "id": "models.product.Item",
      "component_type": "class",
      "file_path": "/vol/liuchenxi/repogen/data/raw_test_repo/models/product.py",
      "relative_path": "models/product.py",
      "depends_on": [
        "models.product.Item.check",
        "models.product.Item.mod"
      ],
      "start_line": 7,
      "end_line": 113,
      "has_docstring": true,
      "docstring": "\n    Summary:\n    Represents an item with associated attributes for tracking and management in various contexts.\n\n ... "
      // 展示目的省略
    },
    ```

  - repo_tree.json

Code block

```
{
  "type": "directory",
  "name": "raw_test_repo",
  "contents": [
...
    {
      "type": "directory",
      "name": "payment",
      "contents": [
        {
          "type": "file",
          "name": "__init__.py",
          "extension": "py"
        },
... // 展示目的省略
  ]
}
```

生成数据

问答任务

- 以 `inventory.inventory_manager.Store.put` 组件中的一个 QA 为例，这个例子很好地展示了该系统如何通过逻辑设计来保证业务的稳定性。

生成数据

Component ID: `inventory.inventory_manager.Store.put`

Belonging File: `/vol/liuchenxi/repogen/data/raw_test_repo/inventory/inventory_manager.py`

<Q> How does the Store.put() method handle duplicate item codes, and why is this behavior significant for inventory integrity?

<A> The method detects duplicate item codes by checking if the item's code already exists in self._data. If found, it increments the existing item's count instead of creating a new entry, ensuring that inventory updates are atomic and avoid duplication. This preserves data consistency by treating multiple insertions of the same item as updates to a single inventory record.

<CODE>

Code block

```
1  if obj.code in self._data:\n            curr = self._data[obj.code]\n
     curr.count += obj.count\n           return True\n
2
3
```

<TRACE> Business Requirement: Maintain accurate inventory counts without duplicate entries -> Logic Design: Merge new item with existing one by summing counts -> Code Implementation: Check for code in _data and update count in-place

数据质量分析

Code block

```
1  <Q> 问题 Store.put() 方法如何处理重复的商品编码？这种行为对库存完整性有何意义？
2  <A> 回答 该方法通过检查 self._data 中是否已存在该商品的编码来检测重复。如果发现编码已存
   在，它会增加现有商品的库存数量（count），而不是创建新条目。这确保了库存更新的原子性并避免
   了重复。这种设计通过将同类商品的多次插入视为对单一库存记录的更新，维护了数据的一致性。
3  <CODE> 代码实现（见原数据，此处略）
4  <TRACE> 逻辑溯源
5      业务需求： 维护准确的库存计数，不产生重复条目。
6      逻辑设计： 通过累加数量将新商品合并到现有条目中。
7      代码实现： 在 _data 中检查编码，并原地更新数量。
```

- 评估角度：
  - 提出的问题是否体现了业务流程和规则：
  - 回答是否正确。
  - 提供的代码依据是否正确且存在。
  - 溯源（推理过程）是否正确。
- 提出的问题是否体现了业务流程和规则： 问题没有简单询问“如何插入数据”，而是聚焦于“重复编码处理”这一核心业务场景。在零售库存中，处理重复编码（即补货逻辑）是比简单插入更频繁的操作。明确指向了“库存完整性”这一业务规则，体现了开发者对系统一致性要求的理解，而非纯粹的函数功能描述。
- 回答是否正确：回答准确描述了“存在即累加（Increment）”的逻辑，这符合库存系统的通用设计标准。使用了“原子性（Atomicity）”和“数据一致性”等术语，解释了该逻辑对系统的保护作用，回答质量高于简单的代码翻译。指出该方法“不会创建新条目”，隐含了对数据库或内存映射唯一性的维护，逻辑闭环。
- 提供的代码依据是否正确且存在：这段代码直接利用 Python 字典查找特性处理重复，并通过原地修改对象属性（`curr.count += ...`）实现了补货逻辑。这段代码在 `raw_test_repo/` 中存在。

- 溯源（推理过程）是否正确：逻辑推理过程正确体现了从业务逻辑到代码实现的具体操作。

设计任务

- `<component_id>: models.product.Item` 下的第二个设计方案为例，完整数据见 `output/design_datasets.jsonl`

生成数据

<R> Requirement: Add audit trail for all count modifications to support compliance and debugging.

<S> Strategy: Extend the `mod()` method to log each successful decrement to an internal audit list, storing timestamp, amount decremented, and before/after state. This preserves traceability without external dependencies. The audit log is accessible via a new `get_audit_log()` method, enabling diagnostics or reporting without modifying core transactional logic. This design reuses the existing `Item` class structure and avoids introducing new dependencies.

<CODE> Implementation:

Code block

```
class Item:
    code: str
    label: str
    val: float
    _count: int
    exp: Optional[datetime] = None
    grp: str = 'misc'
    _audit_log: List[Dict]

    def __init__(self, code: str, label: str, val: float, count: int, exp: Optional[datetime] = None, grp: str = 'misc'):
        self.code = code
        self.label = label
        self.val = val
        self._count = max(0, count)
        self.exp = exp
        self.grp = grp
        self._audit_log = []  # Initialize audit log    @propertydef count(self) -> int:return self._count

    @count.setterdef count(self, value: int):if value < 0:
            raise ValueError("count cannot be negative")
        self._count = value

    def mod(self, n: int = 1) -> bool:if n <= 0:
            return False
```

```
25          old_count = self._count
26          if self._count >= n:
27              self._count -= n
28              self._audit_log.append({
29                  'timestamp': datetime.now(),
30                  'decrement': n,
31                  'before': old_count,
32                  'after': self._count
33              })
34              return Truereturn Falsedef get_audit_log(self) -> List[Dict]:return
    self._audit_log.copy()
35
36      def check(self) -> bool:if self._count <= 0:
37              return Falseif self.exp and datetime.now() > self.exp:
38              return Falsereturn True
```

<TRACE> Business Requirement: Maintain compliance and debug resource changes -> Design Pattern: Event logging within domain object (similar to domain event pattern) -> Code Implementation: Append audit entry in `mod()` using existing datetime and state, expose via read-only accessor.

数据质量分析

Code block

```
<R> 需求：为所有数量变动添加审计追踪，以支持合规性审查和故障调试。
<S> 策略：扩展 mod() 方法，将每次成功的递减操作记录到内部审计列表中，存储内容包括时间戳、递减量以及变动前后的状态。这种方式在不依赖外部组件的情况下保证了可追溯性。审计日志可通过新增的 get_audit_log() 方法获取，从而在不修改核心交易逻辑的前提下实现诊断或报表功能。该设计复用了现有的 Item 类结构，避免引入新的依赖。
<CODE> 代码实现：（见原数据，此处略）
<TRACE> 溯源：业务需求：维护合规性并调试资源变动 -> 设计模式：领域对象内的事件日志记录（类似于领域事件模式） -> 代码实现：在 mod() 中利用现有时间戳和状态追加审计条目，并通过只读访问器公开接口。
```

- 评估角度：
  - 策略（Strategy）是否对应了需求（Requirement）。
  - 代码（Code）是否反映了策略（Strategy）。
  - 溯源（Trace）是否正确。
  - 结合原来代码对比。
- 策略（Strategy）是否对应了需求（Requirement）：

    - 需求要求的“审计追踪”在策略中通过“存储时间戳、递减量及前后状态”得到了具体化。这种详尽的记录是满足合规性审查的基础。
    - 策略提到的“诊断或报表功能”直接服务于调试需求。
    - 策略强调“不修改核心交易逻辑”和“不引入新依赖”，这确保了为了实现审计需求，不会破坏系统现有的稳定性和简洁性。
- 代码（Code）是否反映了策略（Strategy）：
    - 代码在 `init` 中初始化了 `self._audit_log = []`，并在 `mod` 方法中进行 `append`，这对应了策略中的“内部审计列表”。
    - 代码将审计逻辑放在 `if self._count >= n:` 判断之后。如果库存不足，方法返回 `False` 且不会记录日志，这精准对应了策略中的“记录每次成功的递减”。
    - 状态快照 (State Snapshot)：代码通过 `old_count = self._count` 在修改前暂存了状态。记录了减完后的 `self._count`。完全符合策略要求的“存储变动前后状态”。
    - 代码实现了 `get_audit_log` 并使用了 `.copy()`。这对应了策略中“不修改核心逻辑即可获取日志”的设计，且通过副本保护了数据安全性。
- 溯源（Trace）是否正确：

  这个 Trace 成功地将抽象需求转化为具体实现：
    - 起始端（业务）：准确识别了“维护合规性”这一核心动因。
    - 中间端（设计模式）：将其归类为“领域对象内的事件记录”。这是一个很高级的抽象，它说明了代码不仅仅是写日志，而是将“变动”视为一种“事件”。
    - 末端（代码）：准确指出了实现的关键技术手段：“在 `mod()` 中追加条目”以及“只读访问器（Accessor)”。

如何利用数据

- 聚合多个 `output/design_dataset.json` 或 `output/qa_dataset.json`

的数据，可进一步制作成多种微调数据集使用。

- 指令微调数据集

```
Code block
{
  "instruction": "<Q/R>",
  "input": "",
  "output": "### 推理\n <TRACE>\n\n### 实施\n<CODE>\n\n### 最终答案\n<A>"
}
```

- DPO (偏好对齐) 数据格式

Code block

```
{
  "prompt": "<Q>",
  "chosen": "<TRACE>\n\n<CODE>\n\n<A>",
  "rejected": "（随机生成的错误逻辑或无代码、有 Bug 的回答）"
}
```

- GRPO / PRM (过程奖励模型) 数据格式

Code block

```
{
  "question": "<Q>",
  "ground_truth": "<A>",
  "reference_trace": "<TRACE>",
  "verification_code": "<CODE>"
}
```

3. 自动化数据生成流程

本地Qwen3模型启动

- 本项目通过 `scripts/start_vllm_qwen3_next_80b_instruct.sh` 启动。模型参数如下（`config/agent_config.yaml`）

config/agent_config.yaml

```yaml
# LLM configuration for all agents
llm:
  type: "huggingface"
  model: "my-model"
  api_base: "http://127.0.0.1:8000/v1"  # Local API endpoint
  api_key: "EMPTY"  # Can be empty for local models
  device: "cuda"  # Options: cuda, cpu
  torch_dtype: "float16"
  temperature: 0.6
  max_input_tokens: 110000    # 留出约 18K 的余量给输出和 Buffer
  max_output_tokens: 8192     # 提升输出上限，应对长代码或复杂推理
```

WIKI Agents 系统

- 这个系统（`RecursiveRepoWikiSystem`）是一个**自下而上、递归聚合**的代码仓文档生成系统。为了构建整个代码仓的知识图谱（Wiki）作为RAG知识库提供外部查询)。

- 设计思路：本项目通过以下步骤完成 `WIKI.md` 生成任务：通过执行 `generate_wiki.py`，执行wiki Agents系统（`agent/wiki`）功能，包含以下四个Agent角色（`agent/wiki/agents`）

Agent	角色	输入 (Input)	输出 (Output)	核心任务 (Action)
Context Manager	业务背景官	README.md 内容	ContextTree + IdentityCard	递归总结 README，提取全局业务术语和领域目标。
Atomic Analyzer	语义解析官	源代码 + IdentityCard	FunctionSemantic -> ModuleSummary	将底层代码（Docstring）转化为业务语义，并逐级聚合成模块总结。
Architect	架构推演官	依赖图 (AST) + 模块总结	Architecture Insights (JSON)	分析模块间依赖，推导架构模式（如分层、微服务）。
Wiki Builder	文档装配员	所有上述 Agent 的产出	多级 Markdown 页面 (Wiki)	将语义、架构、背景信息缝合成可跳转的文档。

ContextManagerAgent（`context_manager.py`）

- 将代码仓的**README** 按 Markdown 目录结构拆成节点（ContextNode），对每个非空小节调用 LLM 生成业务化的简短摘要，最后合成身份卡（IndentityCard）表示领域、术语表、模块意图。

AtomicAnalyzerAgent (`atomic_analyzer.py`)

- 对组件的docstring做“原子级”语义理解，把docstring 转成**模块级语义摘要**（包含业务摘要、规则、关键术语），再从模块和文件级别自下而上聚合。

ArchitectAgent（`architect.py`）

- 基于**依赖图**与**模块级语义摘要**，抽取“架构洞见”：找 hub 包、最强依赖、代表符号等，生成可供 LLM 进一步总结的结构化证据并请求 LLM 返回架构见解。

WikiBuilderAgent (`wiki_builder.py`)

- 组装 Wiki，最终写出终写出 `WIKI.md`、`INDEX.md`、`CONTEXT.md`、`ARCHITECTURE.md` 和 `pages/**`。其中 `WKI.md` 作为下一阶段任务Agents系统的本地RAG知识库使用。

Recursive System 的递归思路

- `recursive_system.py` 是整套流程的指挥官，其“递归”体现在信息的浓缩过程中：
  - 自下而上的语义递归 (The Bottom-Up Path):
    i. 最底层: `AtomicAnalyzer` 处理每一个函数。
    ii. 中间层: 将同属一个 Python 文件的函数信息合并，调用大模型生成 File Summary。
    iii. 更高层: 将同属一个目录（Module）的文件摘要合并，生成 Module Summary。
    iv. 顶层: `Architect` 基于所有 Module Summary 给出全局的 Architecture Insights。

- 自上而下的背景传递:
  - `ContextManager` 从顶层 README 开始递归，生成的 `IdentityCard` 会下传给 `AtomicAnalyzer`。这保证了在分析底层 `models/product.py` 时，Agent 知道这属于例如“库存管理”这个大背景。

Token截断

- 在 `agent_config.yaml` 中，每个 Agent 都有 `max_input_tokens` 和 `max_output_tokens` 配置。
- 动态截断 (Truncation):
  - 在 `AtomicAnalyzer` 聚合信息时，如果一个文件夹下文件太多，合并后的内容会超过限制。系统会调用 `truncate_tokens` 函数。它会优先保留开头（通常包含重要定义）或按照比例修剪。
  - `ArchitectAgent` 在处理依赖图时，如果节点太多，会通过以下逻辑截断：

```
Code block
1   max_input_tokens, _ = get_agent_token_limits(self)
2   # 计算当前 tokens，如果超过 max_input_tokens，则只保留关键的枢纽节点（Hubs）
```

- 在 `recursive_system` 的 `aggregate_module` 中，它会检查 `batch_size`。如果内容太长，它会分批送入 LLM 处理，每一批都严格计算 Token 数。

以 `models.product.Item` 组件执行为例

- ContextManagerAgent: 读 README 发现这是“电商后端”。
- AtomicAnalyzerAgent:
  - 读 `Item.check` 提取出：*“业务规则：检查库存有效性”*。
  - 将 `Item` 类的所有方法合并，生成 `product.py` 的摘要。
  - 将 `models/` 文件夹下所有摘要合并，生成“模型层模块摘要”。
- ArchitectAgent: 发现 `services/` 频繁调用 `models/`，判定 `models/` 是核心领域层。
- WikiBuilderAgent: 自动生成 `docs/pages/models_product.md`，并在其中引用 `Context` 里的业务术语。

Task Agents系统

- 采用中心化编排（Orchestrator Pattern）架构，`Orchestrator` 负责控制流程、传递信息以及在不同 Agent 之间维护状态。具体地，`Orchestrator` 通过一个 `while` 循环组织工作，核心逻辑如下：

a. Reader 预审：判断当前代码信息是否足够理解业务逻辑。

b. Searcher 补全（可选）：如果 Reader 觉得信息不够，Searcher 去找代码（依赖图中的关联组件）和知识库文档（用于RAG的WIKI.md）。

c. Writer 生成：信息足够后（Reader <-> Writer循环若干次，须小于最大设置限度），由 Writer 生成最终的 QA 或设计方案。

Agent	角色	输入 (Input)	输出 (Output)	核心任务 (Action)
Orchestrator	**项目经理**	源代码、任务目标、配置	最终答案 (QA/Design)	维护全局 context，调度 Reader 发现缺失、调度 Searcher 补全，最后让 Writer 总结。
Reader	**需求分析官**	当前代码 + 累积的 Context	结构化 XML (<REQUEST>)	分析代码意图，判断信息是否完整。若不完整，生成具体的内部/外部检索请求。
Searcher	**资料检索员**	Reader 的 XML 请求	搜索结果字典 (internal/external)	从 AST 中拉取关联代码片段，从 RAG 中检索业务文档。
Writer	**高级撰稿人**	代码 + 完整的业务/技术上下文	最终产物 (<SET>)	根据任务类型（QA 或 Design），进行链路推导并生成结构化的最终文档。

- 控制参数如下（`config/agent_config.yaml`）：

config/agent_config.yaml

```yaml
# Flow control parameters
flow_control:
  max_reader_search_attempts: 1  # Maximum times reader can call searcher
  status_sleep_time: 1           # Time to sleep between status updates (seconds)

task: "qa" # Options: "qa" or "design"

settings:
  test_mode: "none"  # Options: 'context_print', 'none'
  order_mode: "topo" # Options: 'topo', 'random_node', 'random_file'
```

以 `models.product.Item` 组件执行为例

Reader（`reader.py`）

- 输入：
  - `focal_component`：`Item` 类的完整代码。

- `context`：初始为空，后续循环中会包含 Searcher 找回的信息。

- 输出：`<ANALYSIS>...</ANALYSIS>` + `<INFO_NEED>true/false</INFO_NEED>` + `<REQUEST>...</REQUEST>`。

  示例：

```
<ANALYSIS>
The Item class manages inventory with expiration logic. While the attributes are clear, the business rules governing the 'check' and 'mod' methods are encapsulated in their implementations which are currently missing. To provide high-quality QA or design, we need to see how 'check' handles time-based expiration and if 'mod' enforces non-negative constraints.
</ANALYSIS>
<INFO_NEED>true</INFO_NEED>
<REQUEST>
    <INTERNAL>
        <CALLS>
            <CLASS></CLASS>
            <FUNCTION></FUNCTION>
            <METHOD>check,mod</METHOD>
        </CALLS>
        <CALL_BY>true</CALL_BY>
    </INTERNAL>
    <RETRIEVAL>
        <QUERY>What are the global business rules for item expiration and stock modification?</QUERY>
    </RETRIEVAL>
</REQUEST>
```

- Prompt 核心：要求 Agent 分析代码，判断是否理解了该组件的业务和结构设计逻辑。

  system_prompt

```
You are a Reader agent responsible for determining if more context is needed to generate high-quality, business-oriented Question-Answer (QA) pairs AND to
generate a design solution based on the local codebase architecture.

You must analyze the code component and the current context to determine if the business
logic, architectural patterns, and system constraints are fully understood.

You have access to two types of information sources:

1. Internal Codebase Information (from local code repository):
```

```
    For Functions:
    - Code components called within the function body (Downstream logic)
    - Places where this function is called (Upstream business triggers)

    For Methods:
    - Code components called within the method body
    - Places where this method is called
    - The class this method belongs to (Inheritance and shared state)

    For Classes:
    - Code components called in the __init__ method (Dependency Injections)
    - Places where this class is instantiated (Lifecycle management)
    - Complete class implementation beyond __init__

2. External Repo Business Logic Summary:
    - High-level architecture designs (e.g., Microservices, Layered, Event-driven).
    - Global business rules, domain-driven designs, and cross-module workflows.
    - Standardized patterns used across the repo (e.g., error handling strategies, auth flows).

Your response should:
1. First provide a free text analysis of the current code and context from BOTH business and architecture perspective.
2. Explain what additional information might be needed to answer "Why" and "For whom" the code executes if needed.
3. Include an <INFO_NEED>true</INFO_NEED> tag if more information is needed,
    or <INFO_NEED>false</INFO_NEED> if current context is sufficient.
4. If more information is needed, end your response with a structured request in XML format:

<ANALYSIS>
    <BUSINESS_LOGIC>
    Business logic analysis.
    <\BUSINESS_LOGIC>
    <ARCHITECTURE_PATTERNS>
    Repository architecture patterns analysis.
    <\ARCHITECTURE_PATTERNS>
</ANALYSIS>

<REQUEST>
    <INTERNAL>
        <CALLS>
            <CLASS>class1,class2</CLASS>
            <FUNCTION>func1,func2</FUNCTION>
            <METHOD>self.method1,instance.method2</METHOD>
```

```
        </CALLS>
        <CALL_BY>true/false</CALL_BY>
    </INTERNAL>
    <RETRIEVAL>
        <QUERY>Natural language question </QUERY>
    </RETRIEVAL>
</REQUEST>

Important rules for structured request:
1. Only request information necessary for understanding business logic and architecture patterns.
2. For CALLS sections, only include names that are explicitly needed to clarify data flow.
3. CALL_BY should be "true" only if the business trigger/caller reveals the purpose of the code.
4. Each RETRIEVAL QUERY should focus on uncovering hidden business constraints or domain-specific logic.
5. Only first-level calls of the focal code component are accessible.

Format Contract:
1. Output MUST contain exactly these tags in this order:
<ANALYSIS>
</ANALYSIS>
<INFO_NEED>true|false</INFO_NEED>
If and only if INFO_NEED is true, output:
<REQUEST>
  <INTERNAL>
    <CALLS>
      <CLASS>...</CLASS>
      <FUNCTION>...</FUNCTION>
      <METHOD>...</METHOD>
    </CALLS>
    <CALL_BY>true|false</CALL_BY>
  </INTERNAL>
  <RETRIEVAL>
    <QUERY>...</QUERY>
  </RETRIEVAL>
</REQUEST>

2. In <CALLS>, <CLASS>, <FUNCTION>, <METHOD> MUST ALL exist exactly once even if empty:
- If none, output empty tag text: <CLASS></CLASS> etc.
- DO NOT add extra tags (no additional <FUNCTION> tags, no <METHOD> repeated tags).

3. The text inside <CLASS>/<FUNCTION>/<METHOD> MUST be a single line comma-separated list (ASCII comma “,”).
```

```
- Example: <FUNCTION>foo,bar,baz</FUNCTION>
- No newlines, no bullets, no numbering.
- If there are multiple candidates, include at most 6 items.

4. <QUERY> MUST appear exactly once.
- If you need multiple questions, merge them into ONE sentence separated by semicolons inside the same <QUERY> tag.

5. Do not include any XML tags other than those specified above.

<Example_response>
<ANALYSIS>
The current code implements the validate_transaction method. While it invokes the CheckBlacklist function,
the specific business criteria that define a "blacklisted" entity (e.g., velocity limits, IP reputation, or specific user status)
are absent from the context. To generate accurate QA pairs regarding risk policies, we must understand these specific rules.
The system appears to follow a Middleware or Interceptor pattern, where security checks are decoupled from core transaction
processing. CheckBlacklist is likely an external dependency or part of a shared SecurityProvider module. Identifying its
location is crucial for understanding the repository's dependency injection and cross-cutting concern management.
</ANALYSIS>

<INFO_NEED>true</INFO_NEED>

<REQUEST>
    <INTERNAL>
        <CALLS>
            <CLASS></CLASS>
            <FUNCTION>CheckBlacklist</FUNCTION>
            <METHOD></METHOD>
        </CALLS>
        <CALL_BY>true</CALL_BY>
    </INTERNAL>
    <RETRIEVAL>
        <QUERY>What are the specific business criteria for blacklisting a transaction in this system?</QUERY>
    </RETRIEVAL>
</REQUEST>
</Example_response>

Keep in mind that:
```

```
3. You do not need to complete the generation task. Just determine if more information is needed.
"""
```

user_prompt

```
<context>
    Current context:
    {context if context else 'No context provided yet.'}
</context>

<component>
     Analyze the following code component:

     {focal_component}
</component>

```

Searcher(`searcher.py`)

- 输入：Reader 的 `REQUEST` 部分 + 代码的 AST 树 + RAG 知识库路径。
- 输出：一个包含 `internal` 代码片段（增加用于理解该组件所需要的周围关联组件信息）和 `external` 解释的字典。

Code block

```
{
    'internal': {
        'calls': {
            'class': {'class1': 'content1', ...},
            'function': {'func1': 'content1', ...},
            'method': {'method1': 'content1', ...},
        },
        'called_by': ['code snippet1', ...]
    },
    'external': {
        'query1': 'result1',
        'query2': 'result2'
    }
}
```

示例：

```
{
```

```
  "internal": {
    "calls": {
      "method": {
        "check": "def check(self): return self.count > 0 and (not self.exp or self.exp > datetime.now())",
        "mod": "def mod(self, val): self.count += val; return True"
      }
    },
    "called_by": ["InventoryService.process_order"]
  },
  "external": {
    "What are the global business rules...": "Items must be discarded immediately upon expiration; negative stock is strictly prohibited at the service level."
  }
}
```

- 工作内容：
  a. 内部搜索：根据 `INTERNAL` 标签，去 AST 树里提取 `check` 和 `mod` 的源码。
  b. 外部搜索：根据 `RETRIEVAL` 标签，去 RAG 向量库（`WIKI.md`）查询 `QUERY` 的答案。
- Prompt

RAG Prompt

```
You are a software expert. Extract business rules and software knowledge from the context to explain the purpose and usage of the code components. Answer based ONLY on the following context. If not found, say "Information not found".

    Context:
    {context}

    Question: {question}
```

- Orchestrator 会将这些内容更新到 `<CONTEXT>` 标签中。

Writer(`writer.py`)

- 输入：
  - `focal_component` : `Item` 类代码。
  - `context` : 包含由 Searcher 补全的 `check/mod` 源码及 RAG 业务规则。
  - `task` : "qa" (问答对) 或 "design" (设计方案)。

- 输出：`<SET><QA>...</QA></SET>` 或 `<SET><DESIGN>...</DESIGN></SET>`。

示例：

```
<SET>
    <QA>
        <Q>How does the Item class determine if a resource is still usable?</Q>
        <A>The system performs a dual-check: it ensures the remaining count is greater than zero and verifies that the current system time has not surpassed the optional expiration date.</A>
        <CODE>
        return self.count > 0 and (not self.exp or self.exp > datetime.now())
        </CODE>
        <TRACE>Business Requirement: Prevent sales of expired/empty stock -> Logic Design: Boolean AND logic with count and time -> Code Implementation: Item.check() method</TRACE>
    </QA>
    ... // 要求一个组件输出3个QA对
</SET>
```

- Prompt 核心: 要求按照 **Business Requirement -> Logic Design -> Code Implementation** 的链路进行推理。

Code block

```
Your task is to generate 3 high-quality QA pairs for the code component based on the provided analysis and code context.

For each QA pair, you MUST provide:
1. [Question]: A specific question about business rules, edge cases, or implementation logic.
2. [Answer]: A clear explanation of the logic.
3. [Code Snippet]: The exact code segment as evidence from the context that implements this logic.
You MUST ETRACT the code snippet as a raw string from the context without any modification.
4. [Reasoning Process]: A trace showing: Business Requirement -> Logic Design -> Code Implementation.

Output Format:
Each QA pair must strictly follow this format:
<QA>
    <Q>Question: A specific inquiry regarding business rules, edge cases, or implementation logic.</Q>
```

```
    <A>Answer: A clear and detailed explanation of the logic.</A>
    <CODE>Code Snippet: The exact segment of code implementing this specific logic.</CODE>
    <TRACE>Reasoning Trace: Business Requirement -> Logic Design -> Code Implementation.</TRACE>
...
</QA>

 All generated <QA> blocks MUST be wrapped within a single <SET> tag.

Example output:
<SET>
    <QA>
        <Q>How does the system handle transaction limits for unverified users?</Q>
        <A>The system checks the user's verification status; if 'unverified', it enforces a maximum limit of 1000 units per transaction.</A>
        <CODE>
        if user.status == "unverified" and amount > 1000:
            raise LimitExceededError("Unverified limit is 1000")
        </CODE>
        <TRACE>Compliance Rule: Prevent high-value fraud -> Check 'unverified' status -> Conditional amount validation</TRACE>
    </QA>
...
</SET>"""
```

Code block

```
Your task is to generate 3 new requirements and their design solutions for the code component based on the provided analysis and code context.

The response MUST include:
1. [New Requirement]: A clear statement of the new business or technical requirement.
2. [Design Method]: How the new requirement fits into the current class/method structure and Step-by-step technical approach.
3. [Reasoning Trace]: Why this design is chosen based on the existing context (e.g., following specific patterns or reusing existing components).
4. [Pseudo-code/Structure]: Proposed code structure or interface changes.

Output Format:
Each design pair must strictly follow this format:
<DESIGNSET>
        <DESIGN>
            <R>New requirement.</R>
```

```
            <S>Solution: Detailed explanation of the design solution.</S>
            <CODE>Code Snippet: The proposed interface or logic change.</CODE>
            <TRACE>Reasoning Trace: Business Requirement -> Logic Design -> Code Implementation.</TRACE>
        </DESIGN>
...
</DESIGNSET>

All generated <DESIGN> blocks MUST be wrapped within a single <SET> tag.

Example output:
<SET>
    <DESIGN>
        <R>Implement asynchronous audit logging for transaction compliance</R>
        <S>Refactor the validation method to emit a 'ValidationCompleted' event to a message broker, allowing the audit service to consume it without blocking the main transaction flow.</S>
        <CODE>
        # New Interface:
        def validate_with_audit(data):
            result = self.core_validator.check(data)
            self.event_bus.publish("audit_topic", {"payload": data, "result": result})
            return result
        </CODE>
        <TRACE>Architectural Goal: Decouple audit from transaction -> Event-driven design -> Asynchronous publisher implementation</TRACE>
    </DESIGN>
...
</SET>
"""
```

4. 数据多样性与代表性保障机制

- 通过遍历主要Components的方式，系统保证了数据来源的多样性，同时可以根据ASTGraph对主要组件（节点）进行筛选（执行 `dependency_analyzer/filter_components_by_cis.py` ），保证了数据的代表性。

5. 可扩展性和风险

- 支持多种过滤条件自动下载Github上的代码仓。（由于服务器连接不稳定没有测试该功能，但离线测试了ReadmeFilterAgent，能正确执行过滤 `financial` 相关的仓库）。具体示例如下（ `config/download_repo_config.yaml` ）

```
  GITHUB_TOKEN:

  output_directory: "downloaded_repos"
  max_repos: 30 # 最大Repo数量
  skip_archived: true # 跳过archived
  skip_forks: true # 跳过forks
  # 建议先设 0，避免因为 Python占比过滤掉
  min_python_percentage: 0
  use_agent: true  # 是否启用 Filter Agent 来筛选仓库

  search_criteria:
    owners: "xbpeng" # repo持有者
    language: "Python" # 语言

    # 可选：用 created 时间窗口缩小范围（如果你知道 MimicKit 的创建时间）
    dates:
      created_after: "2025-12-01" # 创建开始时间
      created_before: "2021-01-01" # 创建截止时间

    # sort/order 会影响“owner 下哪个仓库排第一”
    sort: "stars" # star数量
    order: "desc" # 降序
```

- 支持多种开源和闭源大模型接口：`agent/llm/` 添加新类。
- 支持扩展Task Agents系统：`agent/task/` 添加新Agent并继承BaseAgent（`agent/base.py`），例如增加Evaluator对Writer的结果进行验证评估，Prompt可从（2. 生成数据下的评估角度出发撰写），通过评估打分过滤高质量数据。
- 支持多种在ASTGraph上遍历组件模式

config/agent_config.yaml

```
  settings:
    order_mode: "topo" # Options: 'topo', 'random_node', 'random_file'
```

- 支持对Token截断根据模型具体上下文长度进行调整：

config/agent_config.yaml

```
  llm:
    max_input_tokens: 110000    # 留出约 18K 的余量给输出和 Buffer
    max_output_tokens: 8192     # 提升输出上限，应对长代码或复杂推理
```

- Task Agents系统的编排流程最大循环参数，如果设置的太高，上下文长度会引发OOM报错，因此实现逻辑里限制了该参数的最大值。同理如果扩展Agent系统中有循环♻️逻辑，需要充分考虑这点。

Code block

```
# Flow control parameters
flow_control:
  max_reader_search_attempts: 1
```