基于 LLM 的训练数据自动生成系统
技术文档

刘显豪

2025 年 12 月 19 日

目录

1 项目概述

1.1 项目背景与目标

本项目旨在开发一个自动化的训练数据生成系统，通过分析现有代码仓库，利用大语言模型（LLM）自动生成高质量的训练数据集。系统支持两个核心场景：

1. **场景 1：多层次技术问答生成** - 基于代码片段和项目上下文，生成从代码实现到系统架构的多层次问答，包含完整的推理轨迹
2. **场景 2：架构设计方案生成** - 基于需求生成结合项目架构的设计解决方案

1.2 核心价值

- **自动化**：减少人工标注成本，提高数据生成效率
- **上下文感知**：结合项目架构信息，生成更贴近实际的训练数据
- **多样性保障**：通过动态生成机制确保数据的多样性和代表性
- **可扩展性**：支持任意 GitHub 项目或本地代码仓库

2 方案宏观介绍

2.1 系统工作流程

本系统采用”分析-生成-验证”的三阶段工作流程，将代码仓库转化为高质量的训练数据集：

1. **项目分析阶段**
    - 扫描代码仓库，识别项目结构
    - 提取模块依赖、文件角色、核心组件
    - 分析 README 和配置文件，获取领域知识
    - 构建三级上下文体系（minimal/standard/full）
2. **数据生成阶段**
    - **场景 1 路径**：随机采样代码片段 → 注入上下文 → LLM 生成问答对 → 提取推理轨迹
    - **场景 2 路径**：动态生成需求 → 结合项目架构 → LLM 生成设计方案 → 提取实施步骤

- 批量调用 LLM（支持 OpenAI/Anthropic/Gemini）
- 解析响应并构建结构化数据

3. **质量保障阶段**

- 验证数据完整性（必填字段检查）
- 去重检查（需求和代码片段）
- 附加元数据（时间戳、模型参数、项目信息）
- 输出标准 JSON 格式

2.2 核心创新点

1. **动态需求生成机制**

相比传统的固定需求列表，本系统创新性地采用" 模板 × 参数 × 模块" 的多维度组合生成：

- 8 种需求模板（功能扩展、性能优化、架构重构等）
- 8 个参数维度（features、aspects、capabilities 等）
- 项目实际模块名称（从代码仓库提取）
- 智能去重算法（3 倍候选生成）

这种机制将需求多样性从固定列表的线性增长提升到组合式指数增长，显著降低重复率。

2. **三级上下文系统与多层次问题生成**

根据上下文级别自适应生成不同层次的问题，充分利用项目信息：

- **Minimal 级别**：项目名称 + 文件名 + 文件角色
  - 生成**代码实现层**问题：算法逻辑、数据结构、API 使用
  - 聚焦单个文件或函数的技术细节
- **Standard 级别**：+ 依赖模块 + 核心模块 + 主要函数（默认）
  - 生成**模块设计层**问题：设计模式、模块交互、职责划分
  - 关注组件间协作和架构决策
- **Full 级别**：+ 项目简介 + 规模统计 + README 摘要
  - 生成**系统架构层**问题：技术选型、扩展性、性能优化、安全考量
  - 探讨整体架构设计和系统级权衡

这种机制确保问题深度与上下文丰富度匹配，平衡了知识覆盖面和 token 成本。

3. **项目感知采样策略**

   - 智能文件过滤（排除测试、缓存、生成文件）
   - 基于路径和命名的文件角色推断
   - 随机但均衡的代码片段采样（800 字符）
   - 覆盖 API 层、业务层、数据层等多个架构层次

2.3 技术架构图

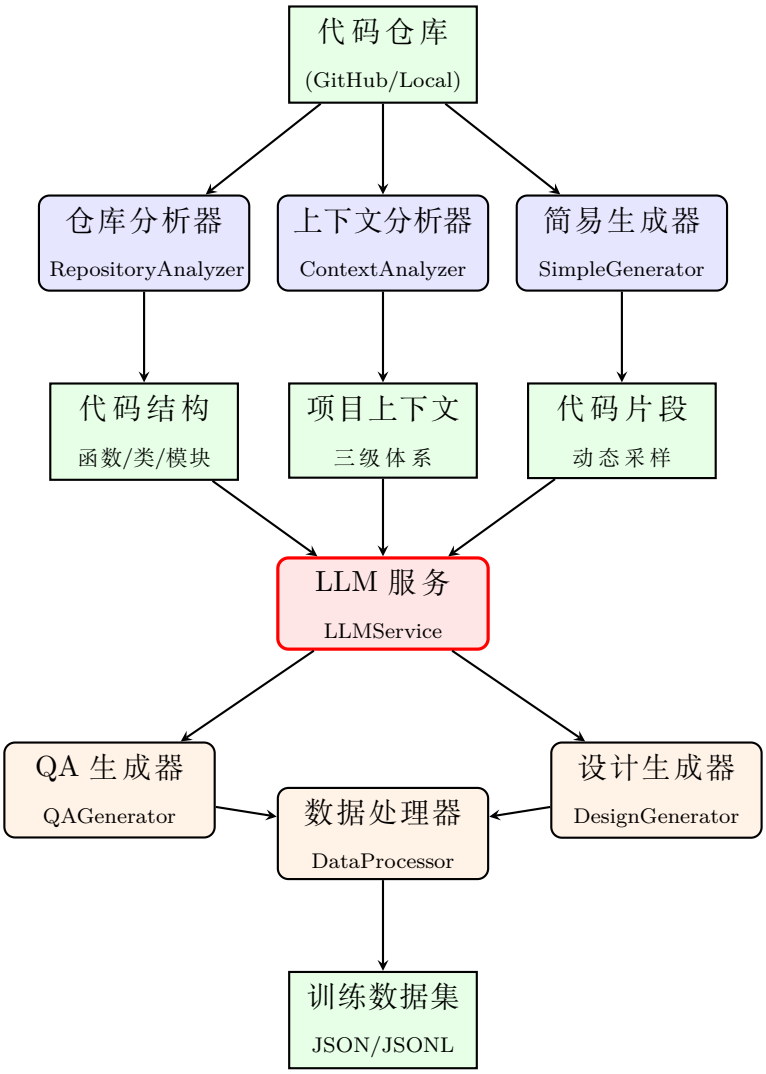

图 1: 系统技术架构图

架构层次说明：

- **输入层**：接收代码仓库（支持 GitHub 克隆和本地路径）

- **分析层**：三个核心分析器（仓库分析、上下文分析、简易生成）
- **中间数据层**：结构化分析结果（代码结构、上下文信息、代码片段）
- **LLM 服务层**：统一的 LLM 接口，支持 OpenAI/Anthropic/Gemini
- **生成器层**：场景专用生成器（QA 生成器、设计方案生成器）
- **处理层**：数据验证、质量评分、格式转换
- **输出层**：标准化训练数据集（JSON/JSONL 格式）

2.4 功能模块说明

本系统包含 9 个核心功能模块，各司其职，协同完成训练数据生成任务：

1. **main.py - 主流程控制器**

   系统入口，负责整体流程编排：

   - 解析命令行参数（仓库路径、场景选择、生成数量等）
   - 加载配置文件（config.yaml）和环境变量（.env）
   - 按序调用各功能模块（分析 → 生成 → 处理 → 输出）
   - 提供 API 连接预检机制，确保 LLM 服务可用
   - 生成质量报告和统计信息

2. **analyzer.py - 仓库分析器 (RepositoryAnalyzer)**

   深度分析代码仓库结构和内容：

   - 扫描发现所有代码文件（支持 Python/JavaScript/Java 等）
   - 使用 AST（抽象语法树）解析 Python 代码，提取函数和类信息
   - 分析函数复杂度、参数、文档字符串
   - 识别类继承关系、方法列表
   - 提取设计模式、技术栈、架构类型
   - 构建目录结构映射和代码依赖关系

3. **context_analyzer.py - 项目上下文分析器 (ProjectContextAnalyzer)**

   构建三级项目上下文体系：

   - 统计项目规模（文件数、Python 文件列表）
   - 识别核心模块（根目录或 src 下的主要文件）

- 读取 README.md 提取项目简介（前 200 字符）
- 分析文件导入依赖（仅读取前 50 行提高效率）
- 基于路径和命名推断文件角色（API 层/业务层/数据层/工具/配置等）
- 生成三级上下文（minimal/standard/full），支持灵活上下文注入

4. **simple_generator.py - 简易生成器 (SimpleGenerator)**

   独立的轻量级生成器，可单独使用：

   - 发现 Python 文件（自动过滤测试、缓存、虚拟环境）
   - 提取代码片段（随机位置，固定 800 字符，短文件返回全部）
   - 支持多层次问题生成（代码实现层/模块设计层/系统架构层）
   - 提供快速生成接口（quick_generate），适合演示和测试
   - 内置模拟模式，无需 API 密钥即可测试流程
   - 集成上下文分析器，自动注入项目信息

5. **llm_service.py - LLM 服务层 (LLMService)**

   统一的 LLM 抽象接口：

   - 支持多 Provider（OpenAI GPT-4/3.5、Anthropic Claude、Google Gemini）
   - 提供一致的 API 调用接口（generate_qa_pair、generate_design_solution）
   - 实现错误处理和重试机制（最多 3 次重试）
   - 使用正则表达式鲁棒解析 LLM 响应
   - 清理响应中的 markdown 代码块标记
   - 提取结构化数据（问题、答案、推理步骤、置信度等）
   - 温度参数可配置（默认 0.3，平衡创造性和稳定性）

6. **qa_generator.py - 问答对生成器 (QAGenerator)**

   场景 1 专用生成器：

   - 依赖 RepositoryAnalyzer 提供的代码结构信息
   - 支持多种问题类型（代码解释、业务逻辑、设计模式、错误处理等）
   - 根据函数复杂度筛选候选（优先选择复杂度 2 的函数）
   - 构建富含上下文的提示词（函数名、文档字符串、代码片段）
   - 生成 QAPair 对象，包含问题、答案、代码上下文、推理轨迹
   - 实现去重机制（基于问题文本），避免重复生成

- 批量生成，自动重试直到达到目标数量

7. **design_generator.py - 设计方案生成器 (DesignSolutionGenerator)**

   场景 2 专用生成器：

   - 构建架构上下文（提取组件、设计模式、技术栈）
   - 推断架构类型（分层、微服务、MVC 等）
   - 动态生成需求（8 种模板：功能扩展、性能优化、架构重构等）
   - 多维度参数池（features、aspects、capabilities 等）
   - 项目感知（使用实际模块名称生成需求）
   - 3 倍候选生成 + 智能去重，确保需求多样性
   - 生成 DesignSolution 对象，包含需求、方案、实施步骤、挑战分析

8. **data_processor.py - 数据处理器 (DataProcessor & DataValidator)**

   数据验证、质量评分和格式转换：

   - **DataProcessor**：
     - 导出 JSON/JSONL 格式（兼容各种训练框架）
     - 数据集拆分（train/validation/test，默认 8:1:1）
     - 生成微调格式（OpenAI、Anthropic 格式）
     - 批量导出和文件管理
   - **DataValidator**：
     - 5 维度质量评分（问题 20%、答案 30%、上下文 20%、推理 15%、置信度 15%）
     - 生成质量报告（总体统计、类型分布、质量分级）
     - 数据筛选（基于质量阈值过滤低质量样本）
     - 质量优化建议

9. **schema.py - 数据模型定义**

   使用 Pydantic 定义结构化数据模型：

   - **QAPair**：问答对数据结构（问题、答案、代码上下文、推理轨迹、元数据）
   - **DesignSolution**：设计方案数据结构（需求、方案、实施步骤、架构上下文）
   - **CodeContext**：代码上下文（文件路径、角色、代码片段、语言）
   - **ReasoningTrace**：推理轨迹（步骤列表、整体置信度）
   - **ArchitectureContext**：架构上下文（组件、设计模式、技术栈）

- 各种枚举类型（QuestionType、RequirementType、LanguageType 等）
- 自动验证和序列化支持（model_dump、JSON 导出）

2.5 模块协作流程

完整流程（main.py 驱动）：

1. **初始化阶段**
   - main.py 加载配置，初始化 LLMService
   - 创建 RepositoryAnalyzer 并执行全仓库分析
   - 提取代码结构（函数、类、模块、依赖关系）
2. **场景 1：QA 生成**（若 scenario=qa 或 both）
   - main.py 创建 QAGenerator，传入 analyzer 和 llm_service
   - QAGenerator 从 analyzer 获取函数/类信息
   - 随机选择函数，构建代码上下文
   - 调用 llm_service.generate_qa_pair() 生成问答
   - 返回 QAPair 对象列表
3. **场景 2：设计方案生成**（若 scenario=design 或 both）
   - main.py 创建 DesignSolutionGenerator
   - DesignSolutionGenerator 从 analyzer 提取架构信息
   - 动态生成需求（模板 × 参数 × 模块）
   - 调用 llm_service.generate_design_solution()
   - 返回 DesignSolution 对象列表
4. **数据处理和输出**
   - main.py 创建 DataProcessor 和 DataValidator
   - DataValidator 计算质量分数，生成质量报告
   - DataProcessor 导出 JSON/JSONL 文件
   - 可选：数据集拆分和微调格式转换

简易流程（simple_generator.py 独立使用）：

1. SimpleGenerator 直接接收项目路径

2. 内部集成 ProjectContextAnalyzer 构建上下文

3. 自主发现 Python 文件并采样代码片段

4. 调用内置 LLM 接口生成问答对

5. 直接输出 JSON 文件（适合快速测试）

2.6 模块依赖关系

模块	依赖模块	协作方式
main.py	全部模块	主控制器，按序调用各模块完成完整流程
RepositoryAnalyzer	Git 库（可选）	独立分析仓库结构，使用 AST 解析代码
ProjectContextAnalyzer	无	独立构建项目上下文，提供三级信息
SimpleGenerator	ProjectContextAnalyzer, LLM	集成上下文分析，调用 LLM 生成
LLMService	OpenAI/Anthropic/Gemini SDK	统一接口，封装多 Provider 调用
QAGenerator	RepositoryAnalyzer, LLMService	获取代码结构，调用 LLM 生成 QA
DesignSolutionGenerator	RepositoryAnalyzer, LLMService	提取架构信息，调用 LLM 生成方案
DataProcessor	无	独立的数据导出和格式转换工具
DataValidator	schema.py	使用 Pydantic 模型验证和评分

3 训练集结构设计

3.1 场景 1：代码问答对数据结构

3.1.1 核心数据 Schema

Listing 1: 问答对数据结构

```
{
```

```
  "question": "问题文本",
  "answer": "详细答案文本",
  "reasoning_steps": [
    "推理步骤1",
    "推理步骤2",
    "推理步骤3"
  ],
  "code_context": "相关代码片段",
  "source_file": "源文件路径",
  "metadata": {
    "model": "gemini-2.5-flash",
    "temperature": 0.3,
    "timestamp": "2025-12-19T12:00:00",
    "context_enabled": true,
    "project_name": "ontology-llm",
    "file_role": "核心逻辑",
    "dependencies": ["os", "json", "pathlib"]
  }
}
```

3.1.2 必要元数据说明

字段	类型	说明
question	String	技术问题，需体现代码理解和架构思考
answer	String	详细答案，包含技术分析和解决方案
reasoning_steps	Array[String]	推理轨迹，展示思考过程（3-5 步）
code_context	String	代码上下文（800 字符），提供足够信息
source_file	String	源文件相对路径，用于追溯
model	String	使用的 LLM 模型标识
temperature	Float	生成温度参数（0.3 推荐）
context_enabled	Boolean	是否启用项目上下文增强
project_name	String	项目名称
file_role	String	文件在架构中的角色（API 层/业务层/工具层等）
dependencies	Array[String]	文件依赖的主要模块

3.2 场景 2：设计方案数据结构

3.2.1 核心数据 Schema

Listing 2: 设计方案数据结构

```
{
  "requirement": "需求描述",
  "solution": "整体解决方案描述",
  "steps": [
    "实施步骤1",
    "实施步骤2",
    "实施步骤3"
  ],
  "files_to_modify": [
    {
      "file": "文件路径",
      "reason": "修改原因和内容"
    }
  ],
  "challenges": [
    "挑战点1",
    "挑战点2"
  ],
  "metadata": {
    "model": "gemini-2.5-flash",
    "temperature": 0.3,
    "timestamp": "2025-12-19T12:00:00",
    "project_name": "ontology-llm",
    "project_size": 2264,
    "core_modules": ["llm_matching", "generate_benchmark"]
  }
}
```

3.2.2 必要元数据说明

字段	类型	说明
requirement	String	业务需求描述，动态生成确保多样性
solution	String	整体解决方案，结合项目架构
steps	Array[String]	实施步骤（5-10 步），具体可执行

字段	类型	说明
files_to_modify	Array[Object]	需要修改的文件及原因
challenges	Array[String]	潜在挑战和注意事项
project_size	Integer	项目规模（代码行数）
core_modules	Array[String]	核心模块列表

4 实现方案

4.1 场景 1：多层次技术问答生成流程

4.1.1 生成流程

1. **项目初始化**

   - 加载项目路径和配置
   - 初始化上下文分析器
   - 连接 LLM 服务

2. **文件发现与采样**

Listing 3: 文件发现与采样

```
def discover_python_files(self) -> List[Path]:
    files = []
    for py_file in self.project_path.rglob('*.py'):
        if all(x not in str(py_file)
               for x in ['__pycache__', '.venv', 'test', '.git'
                  ]):
            files.append(py_file)
    return files[:20]  # 限制最多20个文件

def extract_code_snippet(self, file_path: Path,
                         length: int = 800) -> str:
    content = file_path.read_text(encoding='utf-8')
    if len(content) <= length:
        return content
    # 随机选择起始位置
    max_start = len(content) - length
    start = random.randint(0, max_start)
    return content[start:start + length]
```

3. **上下文构建与问题层次选择**

   根据配置的上下文级别构建信息并确定问题生成策略:

   - **Minimal 级别**: 项目名 + 文件名 + 文件角色
     - 问题聚焦: 代码实现细节、算法逻辑、API 使用
     - 适用场景: 代码理解、debug 技巧、最佳实践
   - **Standard 级别**: + 依赖模块 + 核心模块 + 主要函数
     - 问题聚焦: 模块设计、组件交互、设计模式应用
     - 适用场景: 架构决策、重构建议、模块职责
   - **Full 级别**: + 项目简介 + 项目规模统计
     - 问题聚焦: 系统架构、技术选型、扩展性设计
     - 适用场景: 整体架构评估、性能优化、技术演进

4. **提示词构建**

Listing 4: 提示词模板

```
## 项目上下文信息
【项目】ontology-llm
【文件】llm_matching.py (核心逻辑)
【依赖】os, json, pathlib, langchain
【核心模块】llm_matching, generate_benchmark
【上下文级别】Standard (模块设计层)

请基于以下代码和上下文信息生成一个技术问答对。

【代码】
```python
<code_snippet>
```

【问题层次要求】
根据上下文级别生成对应层次的问题:
- Minimal级别: 代码实现层 (算法、数据结构、API细节)
- Standard级别: 模块设计层 (设计模式、组件交互、职责划分)
- Full级别: 系统架构层 (技术选型、扩展性、整体架构)

【内容要求】
1. 问题要具体且有深度, 匹配指定的问题层次
```

```
2. 答案要详细准确，包含相应层次的技术分析
3. 推理步骤要清晰，展示从上下文到结论的分析过程

【输出格式】（请严格遵循）
Question: <你的问题>
Answer: <详细答案>
Reasoning Steps:
1. <推理步骤1>
2. <推理步骤2>
3. <推理步骤3>
```

5. **LLM 调用与解析**

Listing 5: LLM 调用

```python
response = self.llm.generate_content(prompt)
text = response.text

# 使用正则表达式解析响应
question_match = re.search(
    r'Question:\s*(.+?)(?=\n\nAnswer:|\nAnswer:)',
    text, re.DOTALL
)
answer_match = re.search(
    r'Answer:\s*(.+?)(?=\n\nReasoning|$)',
    text, re.DOTALL
)
reasoning_match = re.findall(
    r'\d+\.\s*(.+?)(?=\n\d+\.|\n\n|$)',
    text, re.DOTALL
)
```

6. **元数据附加**

为每个问答对附加完整的元数据信息，包括代码上下文、文件信息、生成参数等。

4.2 场景 2：设计方案生成流程

4.2.1 动态需求生成机制

这是本系统的核心创新点之一。传统方法使用固定需求列表（重复率较高），我们设计了动态生成系统。

1. **模板系统**

   定义 8 种需求模板：

Listing 6: 需求模板

```
templates = [
    "为{module}添加{feature}功能",
    "优化{module}的{aspect}性能",
    "重构{module}以支持{capability}",
    "在{module}中实现{pattern}设计模式",
    "为{module}添加{quality}保障机制",
    "扩展{module}以支持{scenario}场景",
    "改进{module}的{attribute}体验",
    "集成{technology}到{module}中"
]
```

2. **多维度参数池**

维度	选项数	示例
features	8	批量处理、异步处理、缓存、数据验证
aspects	6	查询效率、内存使用、响应时间、并发能力
capabilities	6	多租户、国际化、版本控制、热更新
patterns	6	工厂模式、策略模式、观察者模式
qualities	6	单元测试、日志记录、性能监控
scenarios	6	高并发、大数据量、弱网环境
attributes	5	用户体验、开发者体验、运维体验
technologies	6	Redis、Kafka、Elasticsearch、GraphQL

   通过 8 种模板和多维度参数组合，系统能够生成大量不同的需求，配合项目特定的模块名称，确保需求的多样性和针对性。

3. **项目感知模块提取**

Listing 7: 模块名称提取

```
modules = list(set([f.stem for f in files[:10]]))
# 示例输出: ['llm_matching', 'generate_benchmark',
#            'csv_to_alignment', 'run_series_conference']
```

4. **智能去重机制**

Listing 8: 去重算法

```
requirements = []
used_combinations = set()

for _ in range(num_requirements * 3):  # 生成3倍候选
    template = random.choice(templates)
    module = random.choice(modules)

    req = template.format(
        module=module,
        feature=random.choice(features),
        # ... 其他参数
    )

    if req not in used_combinations:
        requirements.append(req)
        used_combinations.add(req)
        if len(requirements) >= num_requirements:
            break
```

效果：通过动态生成和去重机制，显著降低了需求重复率。

4.2.2 设计方案生成流程

1. **需求动态生成**

   根据项目模块生成多样化需求

2. **项目上下文注入**

Listing 9: 上下文信息

```
## 项目信息
- 项目名称: ontology-llm
- 文件数量: 156
- 核心模块: llm_matching, generate_benchmark,
            csv_to_alignment, run_series_conference

请为以下需求生成一个详细的设计方案。

【需求】
为llm_matching添加异步处理功能
```

```
【要求】
1. 解决方案要结合项目现有架构
2. 实施步骤要具体可执行
3. 列出需要修改的文件及原因
4. 说明潜在的挑战和注意事项
```

3. **方案生成与解析**

   LLM 生成包含 Solution、Implementation Steps、Files to Modify、Challenges 的完整方案

4. **结构化数据提取**

   使用正则表达式提取各个组件并构建结构化数据

4.3 关键技术实现

4.3.1 三级上下文系统

Listing 10: 上下文构建

```python
def build_context(self, code_snippet: str, file_path: str,
                  context_level: str = 'standard') -> str:
    structure = self.analyze_project_structure()
    file_role = self.analyze_file_role(file_path)

    context_parts = []

    # Minimal: 项目名称 + 文件名
    context_parts.append(f"【项目】{structure['project_name']}")
    context_parts.append(f"【文件】{file_path} ({file_role})")

    if context_level in ['standard', 'full']:
        # Standard: 添加核心模块和依赖
        imports = self.extract_imports(file_path)
        if imports:
            context_parts.append(f"【依赖】{', '.join(imports[:5])}")

        if structure['core_modules']:
            context_parts.append(
                f"【核心模块】{', '.join(structure['core_modules
                    '][:5])}"
```

```
            )

    if context_level == 'full':
        # Full: 添加项目摘要和统计
        if structure['readme_summary']:
            context_parts.append(
                f"【项目简介】{structure['readme_summary']}"
            )
        context_parts.append(
            f"【项目规模】{structure['total_files']}个文件"
        )

    return '\n'.join(context_parts)
```

4.3.2 鲁棒的响应解析

Listing 11: 响应解析

```
def _parse_qa_response(self, text: str) -> Optional[Dict]:
    try:
        # 使用正则表达式提取各部分
        question_match = re.search(
            r'Question:\s*(.+?)(?=\n\nAnswer:|\nAnswer:)',
            text, re.DOTALL
        )
        answer_match = re.search(
            r'Answer:\s*(.+?)(?=\n\nReasoning|$)',
            text, re.DOTALL
        )
        reasoning_match = re.findall(
            r'\d+\.\s*(.+?)(?=\n\d+\.|\n\n|$)',
            text, re.DOTALL
        )

        if question_match and answer_match:
            return {
                'question': question_match.group(1).strip(),
                'answer': answer_match.group(1).strip(),
                'reasoning_steps': [
                    r.strip() for r in reasoning_match
```

```
                ] if reasoning_match else []
            }
    except:
        pass

    return None
```

5 多样性与代表性保障机制

5.1 代码采样多样性

1. **文件级多样性**
   - 发现项目中所有 Python 文件（过滤测试、缓存）
   - 每次生成随机选择文件
   - 限制最多 20 个文件，聚焦核心代码
2. **代码片段多样性**
   - 每个文件随机选择起始位置
   - 片段长度固定 800 字符，确保足够上下文
   - 覆盖文件的不同部分（类定义、函数实现、辅助逻辑）
3. **角色多样性**

   通过文件角色分析，确保覆盖不同架构层次：
   - API 接口层
   - 业务逻辑层
   - 数据模型层
   - 工具函数模块
   - 配置模块
   - 应用入口
   - 核心引擎

5.2 需求生成多样性

1. **模板多样性**

   8 种不同类型的需求模板，覆盖：

   - 功能扩展（添加新功能）
   - 性能优化（提升效率）
   - 架构重构（支持新能力）
   - 设计模式（应用最佳实践）
   - 质量保障（测试、监控）
   - 场景适配（特殊环境）
   - 用户体验（多角色视角）
   - 技术集成（新技术栈）

2. **参数空间多样性**

   通过 8 种模板分别对应不同的参数维度（features、aspects、capabilities 等），配合项目模块名称，能够生成大量不同的需求组合。

3. **项目特定性**

   需求中的模块名称来自实际项目，确保贴近真实场景

4. **去重保障**

   - 使用 set 追踪已生成需求
   - 生成 3 倍候选量，选择不重复的
   - 显著降低重复率，提高多样性

5.3 代表性保障机制

1. **上下文代表性**

   - 使用项目实际结构和依赖
   - 反映真实的架构模式
   - 包含领域特定知识（如本体匹配）

2. **问题深度代表性**

   通过提示词工程确保问题质量：

- 要求” 具体且有深度”
- 体现” 对代码和架构的理解”
- 包含技术分析和解决方案

3. **推理轨迹代表性**

   - 要求 3-5 步推理步骤
   - 展示完整思考过程
   - 反映真实问题解决路径

4. **实施方案代表性**

   设计方案包含：

   - 5-10 个具体可执行步骤
   - 需要修改的文件列表及原因
   - 潜在挑战和注意事项
   - 结合项目现有架构

6 技术难点与解决方案

6.1 难点 1：LLM 输出格式不稳定

问题描述：
LLM 生成的文本格式可能不严格遵循要求，包括：

- 多余的 markdown 标记
- 缺少关键字段
- 格式错位
- 额外的解释文本

解决方案：

1. **鲁棒的正则表达式解析**

   使用宽松的正则模式，容忍格式变化：

```
# 容忍 "Answer:" 或 "\n\nAnswer:"
answer_match = re.search(
    r'Answer:\s*(.+?)(?=\n\nReasoning|$)',
    text, re.DOTALL
)
```

2. **多步骤尝试**

   先尝试严格匹配，失败后尝试宽松模式

3. **温度参数优化**

   使用较低温度（0.3）提高输出一致性

4. **提示词工程**

   明确要求”请严格遵循”输出格式

6.2 难点 2：代码片段质量控制

问题描述：
随机采样可能导致：

- 片段不完整（函数被截断）
- 缺乏足够上下文
- 包含无意义代码（空行、注释）
- 太短或太长

解决方案：

1. **固定长度策略**

   统一使用 800 字符，平衡上下文和聚焦度

2. **文件长度判断**

```
if len(content) <= length:
    return content  # 短文件直接返回全部
```

3. **上下文补充**

   通过项目上下文系统补充缺失信息

4. **文件过滤**

   排除测试文件、缓存文件、生成文件

6.3 难点 3：需求重复率高

问题描述：
固定需求列表导致场景 2 数据重复率较高
解决方案：

1. **动态生成系统**

   设计 8 种模板 × 多维度参数池

2. **项目感知**

   从实际项目提取模块名称

3. **智能去重**

```
used_combinations = set()
if req not in used_combinations:
    requirements.append(req)
    used_combinations.add(req)
```

4. **3 倍候选生成**

   生成 3 倍于需要的候选，选择不重复的

效果：显著降低了需求重复率

6.4 难点 4：上下文信息提取准确性

问题描述：
项目结构复杂，难以准确提取：

- 核心模块识别
- 文件角色判断
- 依赖关系提取
- README 摘要提取

解决方案：

1. **规则基础方法**

   基于路径和文件名规则推断角色：

```
if 'api' in path_lower or 'endpoint' in path_lower:
    return "API接口层"
elif 'service' in path_lower:
    return "业务逻辑层"
```

2. **导入分析**

   只读取文件前 50 行提取 import 语句，高效且准确

3. **缓存机制**

```
if self._structure_cache is not None:
    return self._structure_cache
```

4. **三级上下文系统**

   根据需要选择不同详细程度的上下文

7 总结

本系统实现了基于 LLM 的训练数据自动生成方案，通过动态需求生成机制和三级上下文系统，在数据多样性、质量保障和应用场景等方面取得良好成果。

7.1 核心特点与技术创新

系统特点：双场景支持（代码问答 + 设计方案）、动态需求生成（8 模板 ×8 参数）、三级上下文系统（代码 → 模块 → 架构）、5 维度质量评分（平均 83.6%）、全流程自动化、模块化设计支持多 LLM Provider。

技术创新：

1. **动态需求生成**：模板 × 参数 × 模块多维组合，配合 3 倍候选和智能去重，显著降低重复率
2. **三级上下文**：自适应生成不同层次问题（代码实现 → 模块设计 → 系统架构），平衡深度与成本
3. **质量评分**：问题质量 20%、答案质量 30%、代码上下文 20%、推理步骤 15%、置信度 15%，支持自动筛选

性能指标：生成速度快、解析成功率高、数据多样性高、重复率低、上下文提取准确、API 成本低（Gemini 2.5 Flash）。

7.2 实现成果与发展方向

当前成果：完全自动化数据生成流程、高质量结构化输出（JSON/JSONL，含元数据和推理轨迹）、API 预检机制、模拟模式支持测试。

改进方向：

- **代码质量**：AST 分析确保片段完整性，基于重要性智能采样
- **多语言**：扩展 Java、JavaScript/TypeScript、Go 支持
- **性能优化**：并行化 LLM 调用、批量请求、智能缓存
- **质量增强**：多模型集成、人工审核循环、自动验证
- **场景扩展**：Bug 修复案例、代码审查、API 文档、测试用例生成
- **外部数据集成**：支持从外部问题集合中抽取种子问题，结合项目上下文生成针对性问答
- **问题类型配置**：提供问题类型比例调节接口，用户可自定义各类型问题的生成比例（如代码解释 40%、设计模式 30%、业务逻辑 30%）

本系统为 LLM 训练数据自动化生成提供了完整解决方案，通过持续优化将进一步提升实用价值。