DeepCodeResearch 多场景复杂代码生成 Agent 系统技术方案

团队名称 ： 用 AI 打败 AI

团队成员 ：

YAP HOW CHIEN

KHAW QI FANG

CHEE YUNG CHIAN

1. 项目背景与目标

1.1 赛题背景

赛题 3「复杂代码生成 DeepCodeResearch」要求选手设计并实现一个能够完成复杂代码生成任务的 Agent 系统，核心要求包括：

- 能够先对多源技术文档做深度自主研究，再进行系统设计与代码实现
- 支持 PDF / PPTX / DOCX / TXT / MD 等多种文档格式输入，具备多文档 RAG 能力
- 能够完成 repo-level 代码生成，而不是单文件/片段级别
- 在实现过程中具备 自我调试、自我修复能力，而不依赖人工频繁干预
- 框架本身需具备可扩展性、可插拔工具体系、良好观测性，适合工程落地

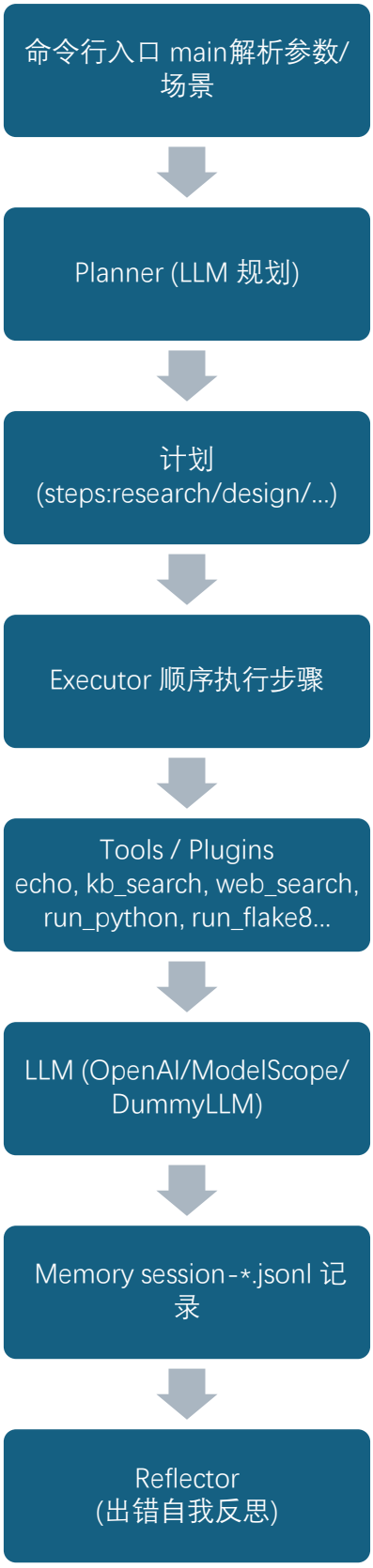
1.2 本方案目标

本方案基于 MS-Agent 的架构思想，自主实现了一套简化但工程完整的 Agent 运行时——**DeepCodeResearch**。目标是：

1. 提供一个**通用的多场景复杂代码生成 runtime**，而不仅仅是一个针对某个单一项目的脚本；
2. 在同一 runtime 下，统一支撑多种复杂任务，例如：
    - Python 代码审查与自动修复 Agent 项目；
    - 图书管理 REST API 服务；
    - 日志分析工具；
    - 股票数据分析脚本等；
3. 完整覆盖「**研究 → 设计 → 代码生成 → 测试 → 自动修复 → 文档撰写**」的工程闭环；
4. 通过 JSON + Markdown 报告、长期记忆，方便评审与后续 CI/平台接入。

2. 总体方案概述

2.1 系统总体架构图



2.2 多场景 Agent Runtime

DeepCodeResearch 提供一个统一入口 main.py，通过参数 --scenario 支持三类场景：

- code_review：复杂的 Python 代码审查与自动修复 Agent 场景
- code_scratch：从零实现某个复杂业务需求的代码生成场景（REST API、数据分析、日志工具等）
- doc_research：只做 Deep Research 和报告，不生成代码的纯文档研究场景

三种场景共用同一套：

- Planner（规划器）
- Executor（执行器）
- Memory（记忆管理）
- RAG 知识库
- 工具/插件体系
- 自动修复组件
- 报告系统

仅通过轻量的 scenario 提示与流程差异，决定是否生成代码仓库、是否运行静态分析与自动修复。

2.3 与 MS-Agent 的对齐

本方案在架构上与 MS-Agent 的核心理念对齐：

- **Planner / Executor 解耦**：Planner 负责根据 goal 和场景生成 JSON 结构化计划；Executor 负责逐步执行、调用工具、记录记忆
- **Memory**：所有关键行为写入 JSONL，便于回放与分析
- **Plugins + Hooks**：通过 plugins/目录与 HookManager 支持工具和生命周期事件的可插拔扩展
- **通用工具调用协议**：通过 ToolRegistry + MCPClient 演示了一个简化版的 MCP/HTTP 工具封装，支持 web_search 等远程工具。

3. 系统架构设计

3.1 模块划分

核心模块位于 deepcoderesearch/ 目录：

- config.py：AgentConfig / LLMConfig，从 code_review_agent.yml 加载项目级配置
- llm.py：抽象 LLM 接口，支持 OpenAI/ModelScope（兼容 OpenAI 协议）以及 DummyLLM
- planner.py：负责将自然语言 goal 转换为 JSON 计划（steps[]），包含 step_type / tools / role
- executor.py：顺序执行计划步骤，内部集成 RAG 检索、工具调用、自我反思逻辑
- memory.py：MemoryManager，将事件写入 memory/session-*.jsonl
- rag/document_loaders.py：多格式文档（PDF/PPTX/DOCX/TXT/MD）加载
- rag/knowledge_base.py：轻量 RAG 知识库，支持可选的向量检索
- tools.py：Tool / ToolRegistry 抽象，兼容 OpenAI function 调用格式
- hooks.py：HookManager，支持 pre_planning、pre_step、post_step、on_error 等钩子
- autofix.py：自动修复模块，基于测试失败与静态分析 issues 生成修改方案并应用
- codegen.py：代码仓库生成模块，负责结合 LLM 输出和保底模板生成 demo repo
- reflection.py：Reflector 模块，执行出错时给出 LLM 级别的错误分析与修复建议
- mcp_client.py：简化版 MCP/HTTP 客户端，用于 web_search 等远程工具调用

3.2 插件与工具体系

plugins/ 下通过插件机制扩展工具与 Hook：

- default_tools.py
    - 工具：echo、kb_search
    - Hooks：pre_step / post_step，打印每步执行日志

- code_runner.py
    - 工具：run_python，在临时文件 _agent_snippet.py 中执行 Python 代码，用于运行 unittest
- code_quality.py
    - 工具：run_flake8 / run_pylint / run_bandit / run_mypy
    - 提供统一 issues schema，便于自动修复
- web_search.py
    - 工具：web_search，通过 MCPClient 调用外部 HTTP 搜索服务，支持设置认证头

插件通过 plugins/__init__.py 自动发现并调用各自的 register(tools, hooks) 方法，无需改动核心代码。

3.3 RAG 与文档管理

- 支持按文件后缀自动选择 loader
- 文档在导入后被切分为 DocumentChunk，并可选地通过 _build_openai_embedder() 为每个 chunk 生成向量
- 检索策略:
    - 优先使用向量余弦相似度
    - 无 embedding 时回退到词重叠策略
- Executor 每个步骤会从 KnowledgeBase 和 kb_search 工具获取上下文，喂给 LLM 进行推理

4. 执行流程与三种场景

4.1 code_review 场景（代码审查与修复）

4.1.1 流程概览

1. **规划阶段**
    - 使用 Planner（scenario=code_review）生成多步 JSON 计划
    - 计划覆盖：文档研究 → 需求归纳 → 架构设计 → 代码生成 → 测试/自检 → 文档撰写

- 每个步骤标注 step_type、建议工具、Agent 角色

2. **人工确认**
    - 在终端打印完整计划
    - 通过 (Y/n) 询问是否继续执行并生成代码仓库
3. **执行阶段**
    - Executor 逐步执行：
        - 从 KnowledgeBase 检索相关文档片段
        - 根据 tools 字段调用 kb_search / web_search 等工具
        - 调用 LLM 根据上下文完成当前步骤
    - 记录每步耗时 duration_seconds 与输出摘要
4. **代码仓库生成**
    - 根据 --coding-task 生成任务 slug，创建对应 outputs/<slug>/ 目录；
    - 调用 generate_demo_repo：
        - 先让 LLM 返回包含 project_name 和 files[] 的 JSON
        - JSON 不合法时，将原始输出保存
          在 README.generated_raw_output.md
        - 然后 _ensure_core_demo_files 覆盖/补足一个稳定可跑的 demo（含单元测试）
5. **静态分析与自动修复**
    - 依据 code_review_agent.yml 中的 static_tools 配置，决定启用 flake8/pylint/bandit/mypy
    - 使用线程池并行运行各工具，统计每个工具的运行时间和 issue 数量
    - 汇总 issues 为统一 schema
    - 如果 autofix.static_issues=true 且有问题，则调
      用 attempt_autofix_from_issues 进行自动修复
6. **测试与基于测试的修复**
    - 使用 run_python 工具，在 demo 仓库下执行内联 unittest 脚本
    - 如果测试失败且 autofix.test_failures=true：

        - 调用 Reflector 分析失败原因
        - 调用 attempt_autofix 基于 stdout/stderr 自动修复代码
        - 再次运行测试并记录结果
7. **报告生成**
    - 生成 JSON 报告 reports/run-<session_id>.json
    - 生成 Markdown 报告 reports/run-<session_id>.md，包含：
        - 每个步骤的耗时、类型、角色
        - 每个静态工具的耗时和问题数
        - 测试结果摘要
        - 自动修复对应的配置和修改文件列表

4.2 code_scratch 场景（通用复杂代码生成）

- 适合从零实现业务系统/工具：
    - 例如：图书管理 REST API（FastAPI + SQLite）
    - 股票数据分析脚本
    - 日志分析工具等
- Planner 会倾向于生成更偏“项目实现”的步骤
- 流程仍包含 research / design / code / test / doc
- 生成 demo repo + 测试，并运行测试
- 静态分析工具默认不强制启用，仅保留 static_analysis 字段展示扩展能力

4.3 doc_research 场景（纯文档研究）

- 用于只做 Deep Research 和报告、不生成代码的场景
- Planner 使用 scenario=doc_research，并在执行前过滤只保留 research / design / doc 步骤
- 不生成代码仓库，不运行静态分析与测试
- 仍然生成 JSON + Markdown 报告，集中展现研究结论和设计分析

5. 报告与观测性

每次运行都会生成：

- JSON 报告：完整记录 goal、coding_task、LLM 配置、plan、静态分析结果、测试结果、autofix 行为
- Markdown 报告：面向人类阅读的精简版本，包含步骤耗时表、静态工具耗时表、测试日志片段、autofix 概况

这些报告可以：

- 作为赛题评审直观了解系统能力的证据
- 作为企业 CI/平台对接时的机器可读 API 输出

6. 多任务覆盖能力展示

本方案在同一 runtime 下运行了多种 goal + coding_task 组合，并保留了对应的：

- outputs/<task_slug>/ demo 仓库
- reports/run-*.json / .md 运行报告

典型任务包括：

1. 代码质量类：
    - 「实现一个集成 flake8、pylint、bandit、mypy 的 Python 代码审查与自动修复 Agent 项目」（code_review 场景）
2. 应用类：
    - 「实现一个支持增删改查、分页和关键字搜索的图书管理 REST API（FastAPI + SQLite）」（code_scratch）
    - 「实现一个命令行日志分析工具，从 log 文件中统计不同错误等级和类型的出现频率」（code_scratch）
3. 数据/分析类：
    - 「从公开 API 拉取指定股票历史价格，计算简单移动平均和波动率，并生成图表和文字分析报告」（code_scratch）
4. 研究类：
    - 「系统性调研多模态 Deep Research Agent 架构，输出设计与优化建议」（doc_research）

通过查看这些任务对应的 outputs & reports，可以证明：

- 同一套 Planner/Executor/RAG/Tools 架构在不同任务上都能工作；
- 并非硬编码针对某一个项目，而是通用的复杂代码生成 runtime。

7. 工程实现亮点与评估

7.1 工程亮点

- **与 MS-Agent 思想对齐的架构解耦**：清晰拆分 Planner / Executor / Memory / RAG / Tools / Hooks
- **真实工具集成**：接入 flake8/pylint/bandit/mypy，输出统一 issues schema 供 LLM 修复使用
- **自动修复闭环**：不仅能发现问题，还能基于静态分析和测试失败自动修复代码
- **场景化设计**：通过 scenario 参数支持 code_review / code_scratch / doc_research 三类典型工程场景
- **报告与观测性**：支持 JSON+Markdown 双格式报告，记录每一步耗时和工具表现，适合 CI 与平台集成
- **多任务验证**：同一系统在多个真实难度的 coding_task 上运行，并保留结果作为证据

7.2 可扩展空间

后续可以进一步扩展：

- 引入更精细的 **Skill 抽象**，将“静态分析+修复”、“测试+修复”封装为可复用高阶技能
- 在 doc_research 场景中输出更结构化的 JSON 研究结果，供 code_scratch 场景直接消费，实现“研究结果驱动代码生成”的自动串联
- 增加 CI 入口脚本，支持对指定仓库目录运行本 Agent 的审查+自动修复，并根据 issue 数或测试结果控制退出码

8. 结语

DeepCodeResearch 并不是简单的“摆在 README 里的设计理念”，而是一个真正能够跑起来、多场景、多任务适配的复杂代码生成 Agent 运行时：

- 它能从多文档输入出发，自主规划步骤、调用工具、生成代码仓库
- 能在真实静态分析工具和单元测试的反馈下，进行 LLM 驱动的自动修复

- 能在同一套架构下处理多种不同类型的复杂 coding_task