DeerFlow-Py

超级智能体协作框架技术文档

解决方案文档 / 架构设计文档 / 详细设计文档 / 实现文档

版本	v0.2.0
日期	2026-07-28
语言	中文
页数	~40页

目 录

第一部分 解决方案文档

1.1 问题定义与背景

随着大语言模型（LLM）能力的飞速发展，单一模型的直接问答已无法满足复杂研究任务的需求。真实世界的研究场景——如市场分析、技术调研、学术综述——需要多步骤推理、多源信息整合、代码验证和迭代修正。ByteDance 开源的 DeerFlow 正是为解决这一需求而生的：它将研究流程建模为多智能体协作的状态机，通过 Planner、Researcher、Coder、Reporter、Critic 等专门化智能体的分工协作，实现端到端的研究自动化。

然而，DeerFlow 原版依赖 LangGraph、LangChain 和 MCP SDK 等多层抽象，对于希望理解内核原理或进行深度定制的开发者而言，这些抽象层反而增加了认知负担。此外，原版缺少独立的测试评估框架（Harness），难以系统性地度量智能体性能。本项目——DeerFlow-Py——旨在用纯 Python 从零重建 DeerFlow 的核心思想，让每一个移动部件（状态机、ReAct 循环、工具调用、MCP 协议、中断恢复、评估器）都可读、可改、可测试。

1.2 设计目标

DeerFlow-Py 的设计围绕以下五大目标展开，每一项都直接影响架构决策和技术选型：

- 零隐藏抽象：不依赖 LangGraph/LangChain，状态机、Agent 运行时、MCP 协议全部从零实现，代码总量约 4900 行，每一行都可追溯。
- 生产可用：支持流式输出、工具调用、JSON 模式、自动重试、token 统计，可直接对接 OpenAI/DeepSeek/Zhipu 等主流 LLM 提供商。
- 可扩展：通过插件系统、中间件、Skills 技能三层扩展机制，第三方可在不修改框架源码的前提下添加自定义 Agent、Tool、Skill 和图节点。
- 可测试：内置 Harness 框架，提供数据集管理、并发执行器、三种评估器（精确匹配/关键事实/LLM-as-Judge）和 FastAPI 服务，支持离线和在线评估。
- 人机协同：支持 Human-in-the-loop 中断恢复，关键决策点可暂停等待人类反馈，通过 Checkpointer 持久化状态实现跨会话恢复。

1.3 解决方案概述

DeerFlow-Py 采用分层架构，自底向上分为五层：核心引擎层、LLM 层、工具层、智能体层和 Harness 层。核心引擎层包含自定义的 StateGraph 状态机（~490 行）和 Channel reducer 系统，实现了 LangGraph 的超步执行、并行 fan-out、条件路由、中断恢复和 checkpointing 等核心能力，但完全不依赖 LangGraph。LLM 层是一个 OpenAI 兼容的异步客户端，支持流式输出、原生函数调用、JSON 模式和 tenacity 自动重试。工具层将 Web

搜索、网页爬虫、Python 沙箱和 MCP 协议统一为 JSON-Schema 描述的可调用工具。

智能体层包含六个专门化 Agent：Coordinator（路由）、Planner（计划拆解）、Researcher（搜索+爬取）、Coder（代码执行）、Reporter（报告生成）和 Critic（评审+修订）。它们通过 research_graph 组合成完整的研究工作流：问题输入后，Planner 生成 JSON 计划，Researcher 按计划搜索和爬取网页，Coder 可选地执行 Python 代码进行数据分析，Reporter 综合所有证据生成 Markdown 报告，Critic 评审报告并决定通过或要求修订。Harness 层提供 FastAPI 服务（/run、/eval、/stream WebSocket）、数据集管理（SimpleQA、GAIA 格式）、并发执行器和三种评估器。

1.4 核心创新点

相较于原版 DeerFlow，本项目在以下五个方面进行了创新或增强：

创新点	原版 DeerFlow	DeerFlow-Py
状态机	依赖 LangGraph（~15000行含依赖）	纯 Python 自研（~490行），零依赖
LLM 层	LangChain ChatOpenAI	直接 httpx async（~322行），无中间层
MCP 协议	@modelcontextprotocol/sdk	纯 Python JSON-RPC（~315行）
记忆系统	仅短期对话窗口	四层记忆：短期+长期+情景+工作记忆
扩展机制	无独立扩展框架	三层扩展：Skills+中间件+插件系统
Harness	无独立测试框架	完整 FastAPI+Runner+Evaluator

其中最关键的创新是四层记忆系统和三层扩展机制。四层记忆系统让智能体不仅能记住当前对话，还能跨会话持久化事实（长期记忆）、回忆相似历史任务（情景记忆，基于 TF-IDF 检索）、并在当前任务中暂存中间结果（工作记忆）。三层扩展机制——Skills 技能、中间件、插件——分别从工作流复用、调用拦截和组件注册三个维度提供扩展能力，使得第三方可以在不触碰框架源码的前提下深度定制行为。

第二部分 架构设计文档

2.1 系统总体架构

DeerFlow-Py 的系统架构遵循经典的分层+管道模式。客户端层（CLI/REST/WebSocket）通过 Harness 层（Runner/Evaluator/Datasets）调度任务，Harness 层将请求转发给核心引擎层（StateGraph/Orchestrator/Middleware），核心引擎驱动智能体层（Coordinator/Planner/Researcher/Coder/Reporter/Critic）协作完成任务，智能体通过工具层（Search/Crawl/Sandbox/MCP）与外部世界交互。整个流程的状态通过 Checkpointer 持久化，记忆系统贯穿所有层。

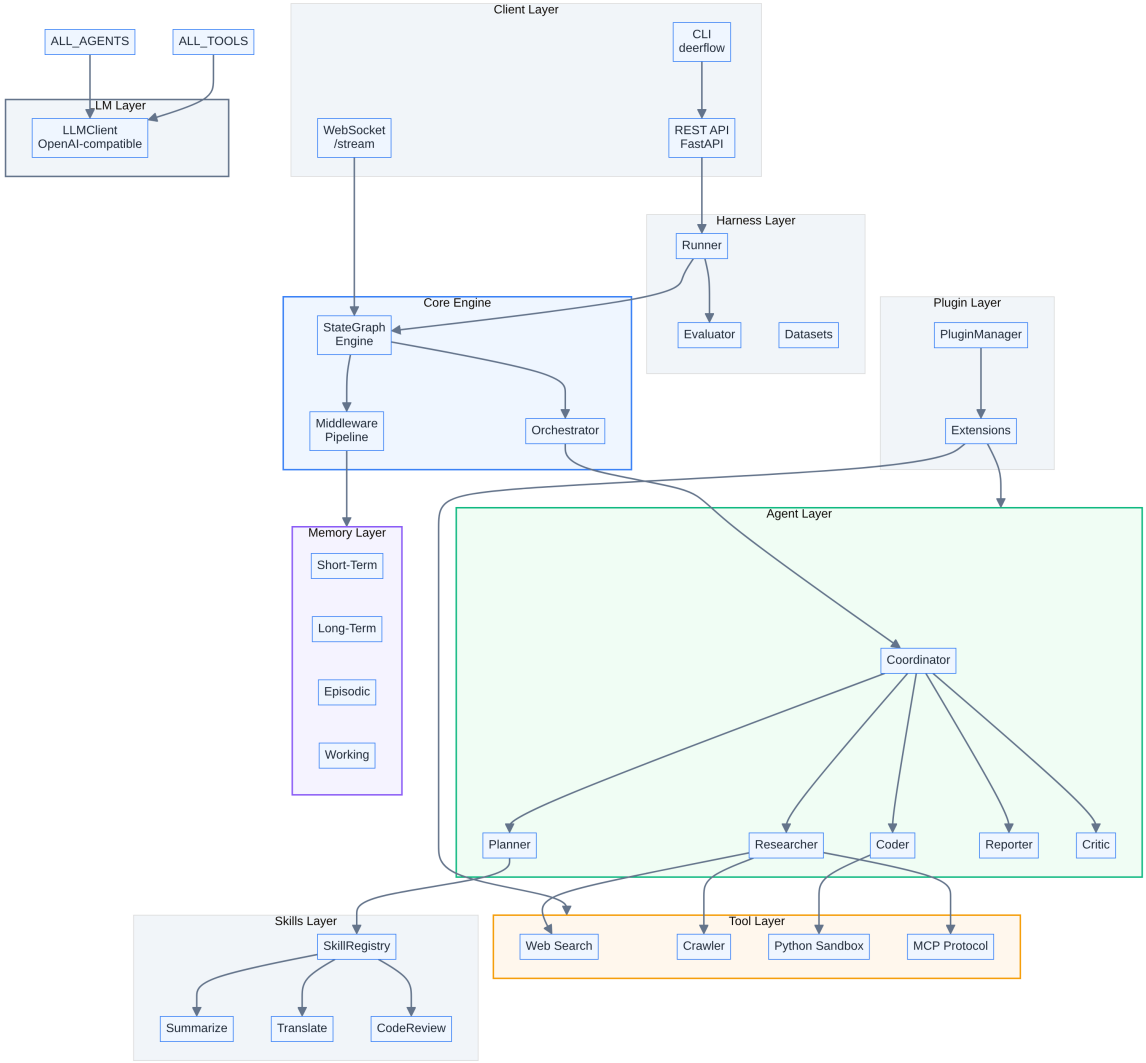

图 2-1 DeerFlow-Py 系统总体架构图

2.2 分层架构设计

系统采用严格的五层分离设计，每一层只依赖其下层接口，不跨层调用。这种设计确保了各层可以独立替换：例如可以将 LLM 层从 OpenAI 换成 Anthropic 而不影响智能体层，或将工具层从本地沙箱换成远程 MCP 服务器而不影响核心引擎。

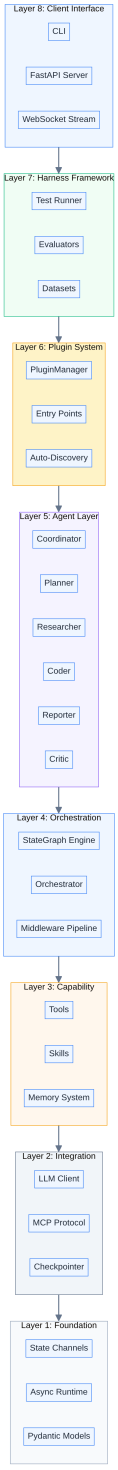

图 2-2 分层架构栈（自底向上）

层	模块	职责	代码量
客户端层	CLI / FastAPI / WebSocket	用户交互、任务提交、流式输出	~300行
Harness 层	Runner / Evaluator / Datasets	测试执行、性能评估、数据管理	~620行
智能体层	6个专门 Agent + research_graph	计划/研究/编码/报告/评审/路由	~440行
核心引擎层	StateGraph / Agent / Memory / Orch.	状态机、ReAct 循环、记忆、编排	~1300行

LLM 层	LLMClient (OpenAI兼容)	异步调用、流式、工具调用、重试	~322行
工具层	Search / Crawl / Sandbox / MCP	搜索、爬虫、代码执行、MCP协议	~900行
扩展层	Skills / Middleware / Plugins	技能复用、调用拦截、插件注册	~800行

从代码量分布可以看出，核心引擎层（~1300行）和工具层（~900行）占据了主体，这反映了框架的设计重心：状态机编排和工具集成是多智能体系统的两大核心能力。扩展层（~800行）是 v0.2.0 新增的，包含记忆系统、Skills、中间件和插件四个子系统。

2.3 组件职责划分

每个组件都有明确的单一职责，组件间通过定义良好的接口通信。StateGraph 只负责状态流转和节点调度，不关心节点内部逻辑；Agent 只负责 ReAct 循环（推理-行动-观察），不关心状态如何持久化；Tool 只负责执行单个操作并返回结果，不关心谁在调用它。这种职责分离使得每个组件都可以独立单元测试——事实上本项目的 24 个测试覆盖了所有核心组件的关键路径。

2.4 技术选型

技术选型遵循「最小依赖、最大可控」原则。所有依赖都是经过仔细评估的：Pydantic v2 用于数据模型验证（性能优于 v1）；httpx 用于异步 HTTP（优于 requests 的同步模型）；tenacity 用于重试逻辑（比手写循环更健壮）；FastAPI 用于 Harness 服务（原生异步、自动文档）；readability-lxml 用于网页正文提取（业界标准）；Playwright 用于图表渲染（跨浏览器一致性）。刻意不使用 LangChain/LangGraph 是为了保持内核透明——每一行代码都可追溯、可调试。

技术	版本	用途	选择理由
Python	3.10+	运行时	async/await 原生支持、类型提示
Pydantic	v2.6+	数据模型	高性能验证、JSON Schema 生成
httpx	0.27+	异步HTTP	原生 async、HTTP/2 支持
FastAPI	0.110+	Harness 服务	原生异步、自动 OpenAPI 文档
tenacity	8.2+	重试机制	指数退避、条件重试
ReportLab	最新	PDF 生成	纯 Python、无外部依赖
Playwright	最新	图表渲染	Mermaid 渲染、截图

第三部分 详细设计文档

3.1 UML 类图设计

下图展示了 DeerFlow-Py 核心类的继承与组合关系。Agent 是所有智能体的抽象基类，定义了 ReAct 循环的骨架（arun 方法）和工具调用逻辑。六个具体智能体（Coordinator/Planner/Researcher/Coder/Reporter/Critic）继承 Agent 并覆盖 run 方法实现各自的工作流。Tool 是工具的抽象基类，FunctionTool 是其便捷子类，通过函数签名自动推断 JSON Schema。StateGraph 是状态机构建器，compile() 后产生不可变的 CompiledGraph 运行时。

图 3-1 核心类 UML 类图

类图中的关键设计模式包括：模板方法模式（Agent.arun 定义 ReAct 骨架，子类覆盖 build_messages 和 inject_context 注入上下文）；策略模式（Channel reducer 的 update 方法，LastValue/AddMessages/AddDocuments 是三种策略）；观察者模式（Middleware 的 before/after 钩子）；工厂模式（build_research_graph 函数组装完整工作流图）；装饰器模式（MiddlewarePipeline 洋葱式包裹 Agent 调用）。

3.2 研究工作流序列图

下图展示了一次完整研究任务的时序流程。用户提交问题后，StateGraph 引擎按超步执行：第一步运行 Planner 生成 JSON 计划；第二步 Researcher 按计划搜索和爬取网页（可能多轮 ReAct 循环）；第三步 Reporter 综合证据生成报告；第四步 Critic 评审报告，若判定为 revise 则回到 Reporter 修订（最多 N 次），若判定为 pass 则结束。整个流程的状态通过 Checkpointer 在每步后持久化，支持中断后从断点恢复。

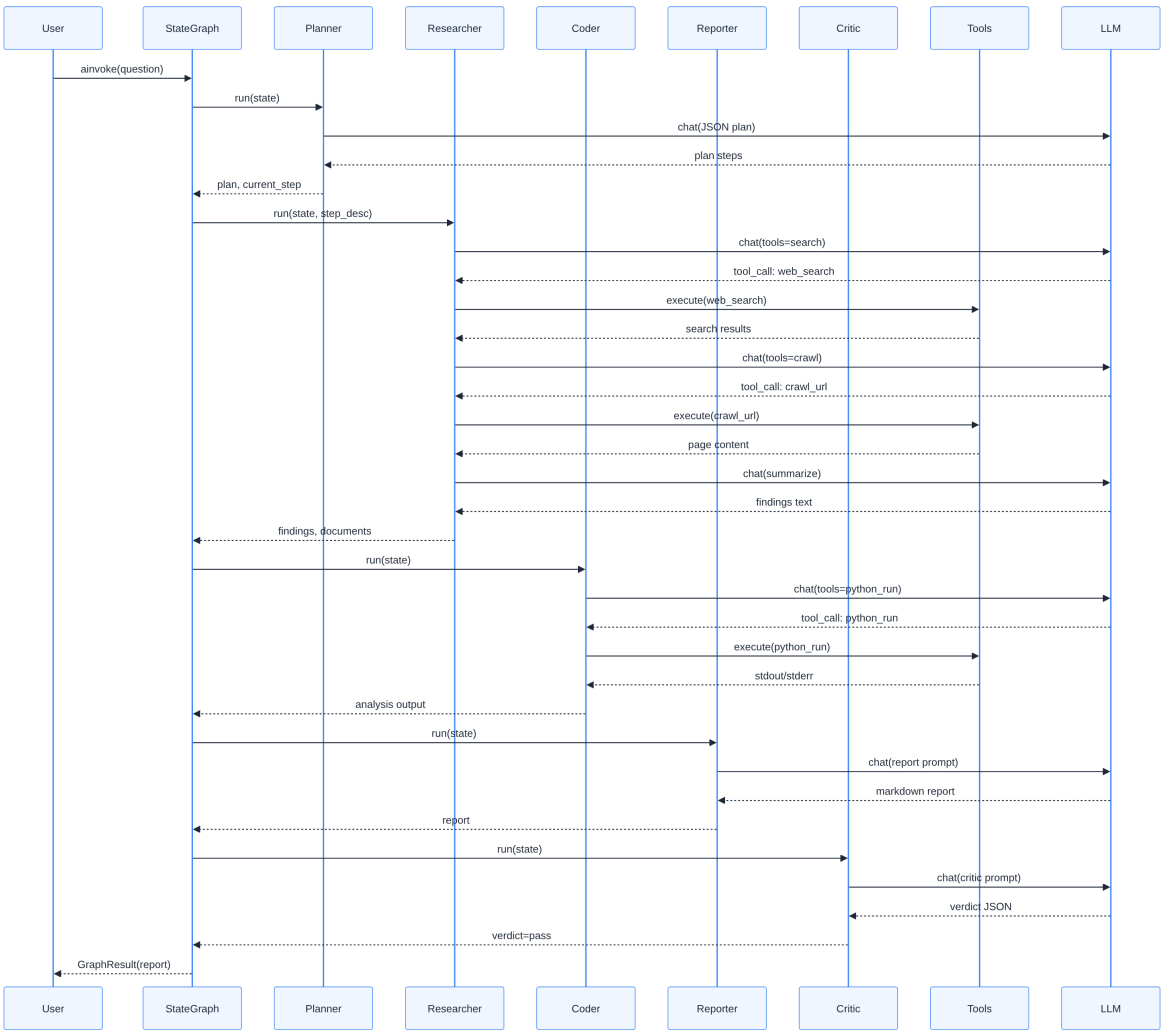

图 3-2 研究工作流序列图

序列图中的关键交互点包括：用户→StateGraph 的 ainvoke 调用携带初始消息；StateGraph→Agent 的 _run_node 内部方法将当前状态传给 Agent.run；Agent→LLMClient 的 chat 调用携带工具 schema；LLMClient 返回的 tool_calls 被 ToolRegistry.execute 并发执行；工具结果作为 tool 消息追加到对话中继续 ReAct 循环。Critic 的 verdict 字段驱动条件路由——这是状态图 add_conditional_edges 的典型应用。

3.3 ReAct 循环流程图

ReAct（Reason-Act-Observe）是 Agent 运行时的核心循环。每轮迭代中，Agent 将对话历史和工具 schema 发送给 LLM；若 LLM 返回 tool_calls，Agent 通过 ToolRegistry 并发执行这些工具，将结果作为 tool 消息追加到对话中，然后进入下一轮迭代；若 LLM 返回纯文本（无 tool_calls），则循环结束，文本即为最终答案。max_iterations 参数防止无限循环。

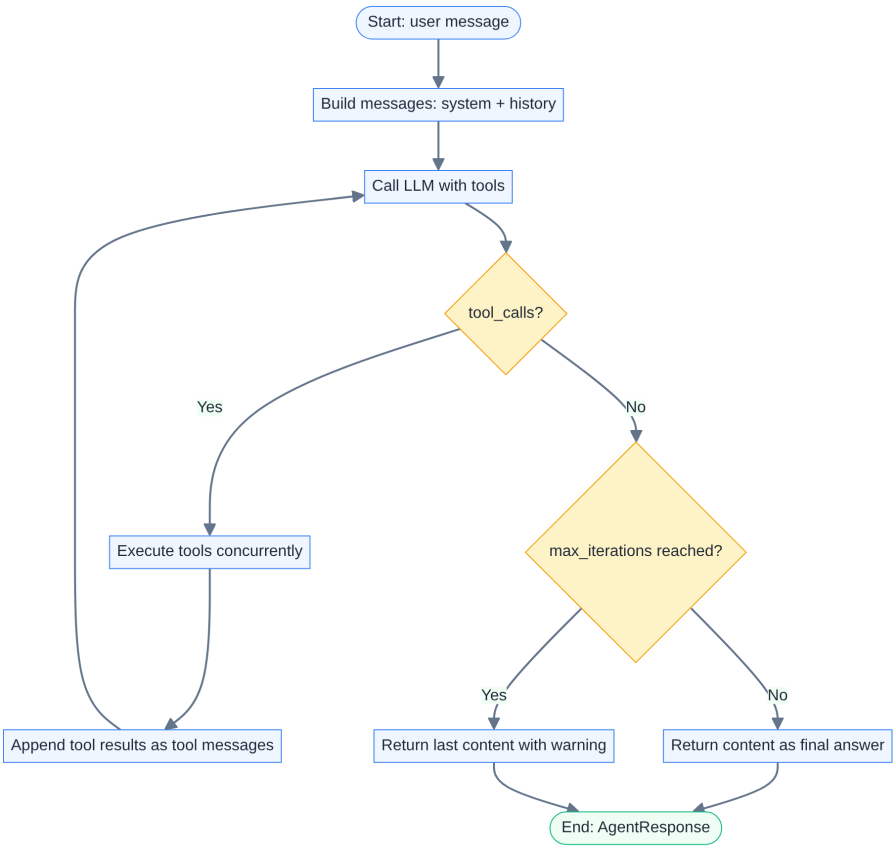

图 3-3 ReAct 循环流程图

ReAct 循环的设计要点包括：工具并发执行（asyncio.gather）减少延迟；每步记录 AgentStep（thoughts/tool_calls/tool_results/latency/usage）用于遥测；工具错误被捕获并转为 ToolResult.error 而非抛出异常，保证循环不中断；LLM 调用通过 tenacity 自动重试瞬时错误（5xx/429）。这种设计使得 Agent 既能自主多步推理，又能在工具失败时优雅降级。

3.4 数据流设计

下图展示了从用户输入到最终报告的数据流转过程。用户问题作为 messages 进入状态图，经过 Planner 后新增 plan 字段（JSON 计划），Researcher 执行后新增 documents 字段（爬取的网页内容），Reporter 执行后新增 report 字段（Markdown 报告），Critic 执行后新增 verdict 和 critique 字段。所有字段通过 Channel reducer 合并——messages 和 documents 是 append-only（去重），其他字段是 last-write-wins。

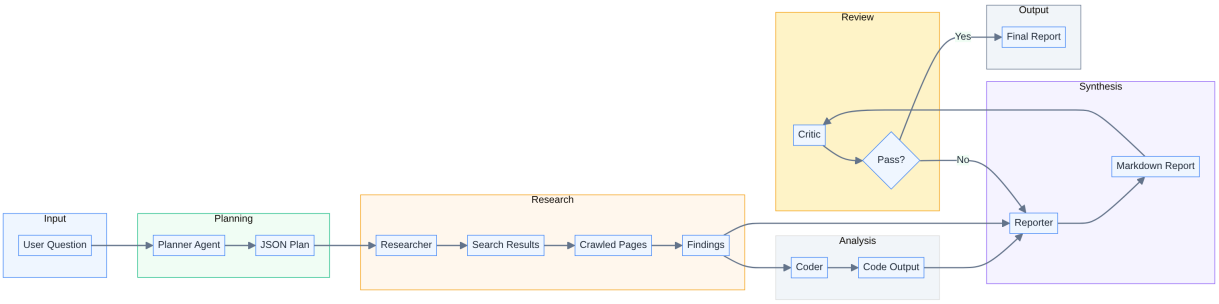

图 3-4 数据流转图

数据流设计的核心是 Channel reducer 系统。每个状态键关联一个 Channel 对象，负责决定如何合并新值与旧值。AddMessages 通道按消息 ID 或内容哈希去重，确保节点重跑（如中断恢复后）不会产生重复消息。AddDocuments 通道按（source, content）哈希去重，避免同一搜索结果被多次加入上下文。LastValue 通道是简单的最后写入胜出，用于 plan、report、verdict 等标量字段。这种设计让状态合并成为声明式操作——节点只需返回部分更新字典，Channel 自动处理合并语义。

3.5 记忆系统架构

v0.2.0 新增的四层记忆系统是本框架区别于原版 DeerFlow 的关键特性。短期记忆（ShortTermMemory）是滑动窗口对话缓冲，保留最近 N 条消息并自动摘要溢出部分。长期记忆（LongTermMemory）是 JSON 文件持久化的键值存储，跨会话保存用户偏好和学到的事实。情景记忆（EpisodicMemory）存储历史任务（Episode），通过 TF-IDF + 余弦相似度检索相似过往经验。工作记忆（WorkingMemory）是当前任务的暂存区，存放计划、中间结果和引用，任务结束后清空。

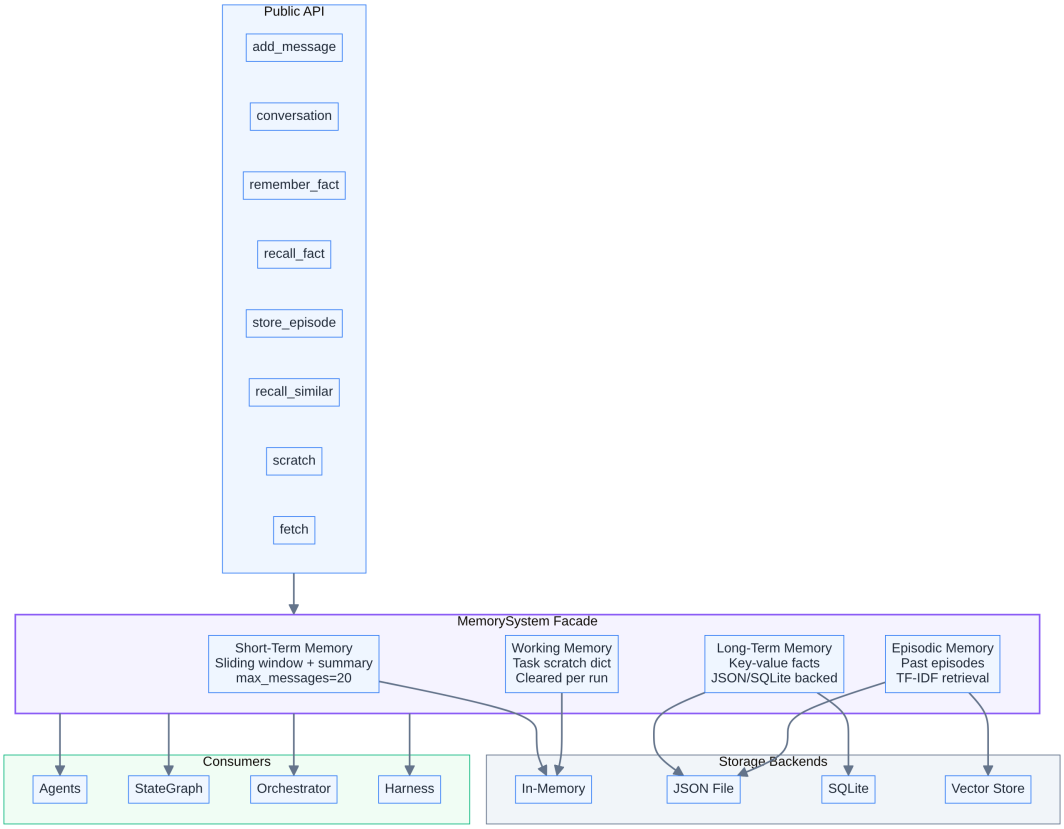

图 3-5 四层记忆系统架构

MemorySystem 是统一门面，提供 remember_fact/recall_fact（长期）、store_episode/recall_similar（情景）、scratch/fetch（工作）等便捷方法。智能体在执行前可调用 recall_similar 检索相似历史任务作为 few-shot 示例，在执行后可调用 store_episode 记录本次经验。这种设计让框架具备持续学习能力——随着使用次数增加，情景记忆库不断丰富，智能体在相似任务上的表现逐步提升。

3.6 插件扩展系统

插件系统让第三方包在不修改框架源码的前提下扩展功能。一个插件是继承 Plugin 基类的Python模块，通过 PluginContext 注册自定义 Agent、Tool、Skill、图节点和中间件。插件发现支持三种方式：entry_points（pyproject.toml 声明，自动加载）、显式路径（传入文件路径或模块名）、目录扫描（自动发现 deerflow_*.py 文件）。

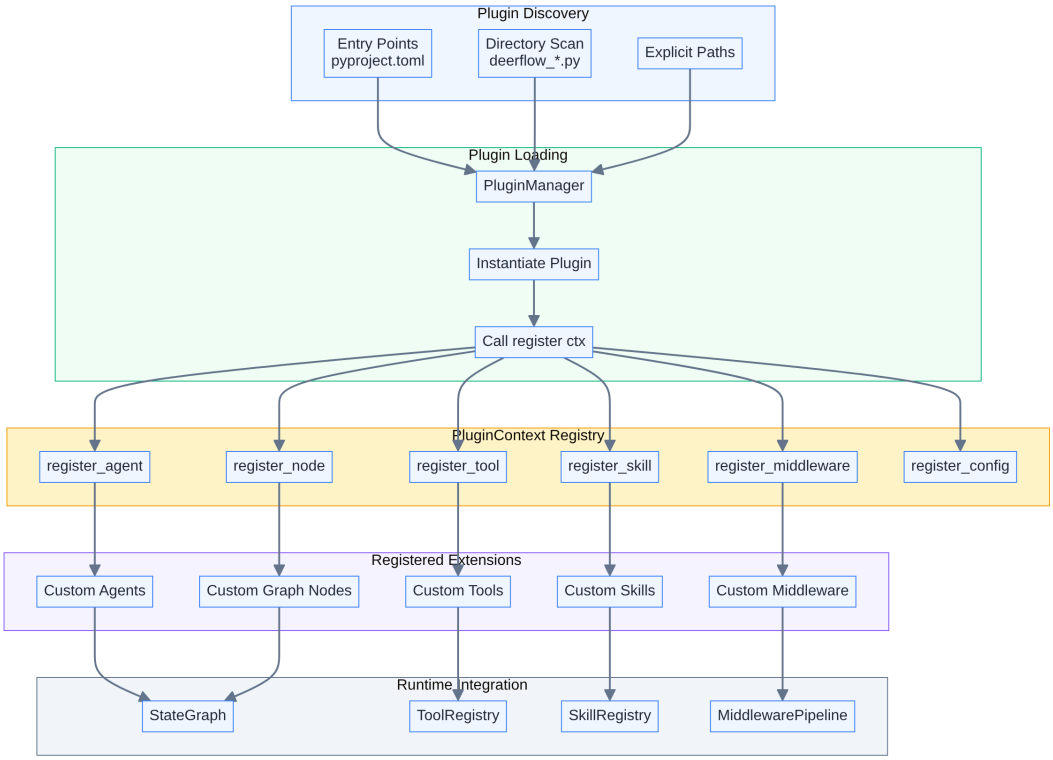

图 3-6 插件扩展系统架构

插件系统的设计借鉴了 Flask 的蓝图和 Django 的应用配置模式。PluginContext 是注册中心，持有 llm、tools、agents、skills、middleware、graph_nodes 等容器。Plugin 基类定义了 register（注册时调用）和 on_unload（卸载时调用）两个生命周期钩子。PluginManager 负责发现、加载、卸载插件，并构建最终的 MiddlewarePipeline 和 StateGraph。这种设计使得企业可以将内部工具封装为私有插件包，通过 pip install 后即自动集成到框架中。

第四部分 实现文档

4.1 状态机引擎实现

StateGraph 是 LangGraph 的纯 Python 等价物，实现了超步执行模型。核心执行循环在 CompiledGraph.ainvoke 中：每次迭代弹出一批就绪节点（入度为 0 的节点），用 asyncio.gather 并发执行，将各节点输出通过 Channel reducer合并到共享状态，然后根据刚完成节点的边计算下一批就绪节点，重复直到到达 END 或无就绪节点。中断通过 GraphInterrupt 异常实现——节点抛出该异常时，图捕获它并将中断信息存入 state['__pending_interrupt__']，通过 Checkpointer 持久化后返回 interrupted=True。恢复时，_resume_frontier 方法读取中断信息，将 resume 值作为中断节点的输出应用到状态，然后计算下一批节点继续执行。

状态机的关键设计决策包括：节点是 async 或 sync 可调用（sync 自动包装为 to_thread）；边可以是静态的（add_edge）或条件的（add_conditional_edges，路由函数返回下一节点名）；并行 fan-out 通过从一个节点引出多条边实现；checkpointing 在每步后自动调用（可插拔 InMemoryCheckpointer 或 JSONFileCheckpointer）；astream 方法是流式版本，每步 yield 一个事件字典，适合 WebSocket 推送。

4.2 Agent 运行时实现

Agent 基类实现了标准 ReAct 循环。arun 方法接收消息列表和上下文，内部循环最多 max_iterations 次：每次调用 LLMClient.chat 传入对话和工具 schema，若返回 tool_calls 则通过 ToolRegistry.execute_many 并发执行，将结果作为 tool 消息追加到对话中继续循环；若返回纯文本则结束循环返回 AgentResponse。每步记录 AgentStep（thoughts/tool_calls/tool_results/latency/usage）用于遥测和 Harness 评估。子类通过覆盖 build_messages 注入系统提示词，覆盖 inject_context 注入来自图状态的动态上下文。

六个具体智能体的实现各有特点：Coordinator 是无工具的单次 LLM 调用，只返回一个智能体名用于路由；Planner 使用 JSON 模式（response_format）强制 LLM 返回结构化计划，解析失败时回退到默认计划；Researcher 配备 web_search 和 crawl_url 工具，通过 inject_context 接收当前计划步骤描述以聚焦搜索；Coder 配备 python_run 沙箱工具，ReAct 循环支持写-跑-读错-修-重跑；Reporter 是无工具的单次调用，从状态中读取 plan 和 documents 作为额外上下文；Critic 使用 JSON 模式返回结构化评审结果，verdict 字段驱动条件路由。

4.3 工具系统实现

Tool 基类定义了 name、description、parameters（JSON Schema）三个属性和 arun 异步方法。FunctionTool 是便捷子类，通过函数签名自动推断参数类型（int→integer，str→string 等），通过 __post_init__ 在构造时自动生成 schema。ToolRegistry 是工具集合，提供 schemas()

方法将所有工具转为 OpenAI function-calling 格式，execute() 方法接收一个 tool_call 字典，解析 name 和 arguments，查找对应工具并执行，返回 ToolResult。execute_many() 用 asyncio.gather 并发执行多个调用。

内置工具包括：DuckDuckGoSearchTool（免费搜索，解析 HTML 结果页）、TavilySearchTool（生产级搜索，需 API key）、WebCrawlerTool（用 readability-lxml 提取正文）、PythonSandboxTool（子进程隔离执行 Python，支持 RLIMIT 和超时）、MCPClient/MCPServer（JSON-RPC 2.0 over stdio 的 MCP 协议实现）。MCP 实现支持 initialize 握手、tools/list 列举、tools/call 调用，可以与 Claude Desktop、Cursor 等标准 MCP 客户端互操作。

4.4 Skills 技能系统

Skill 是比 Tool 更高层的抽象——Tool 是单个函数，Skill 是一个可能调用多个工具、运行多次 LLM 调用的完整工作流。Skill 基类定义了 execute 异步方法和 to_tool_dict 方法（将 Skill 转为 OpenAI 工具描述符，让 LLM 可以像调用 Tool 一样调用 Skill）。内置四个技能：SummarizeSkill（将任意文本总结为 N 个要点）、TranslateSkill（翻译到目标语言）、CodeReviewSkill（审查代码片段的 bug）、OutlineSkill（为主题生成结构化大纲）。SkillRegistry 管理技能集合，default_skills(llm) 工厂函数返回预装四个内置技能的注册表。

Skills 系统的设计灵感来自 deer-flow 的 deep research skill 和 MCP 的 prompts 能力，但推广到了任意工作流。技能是可组合的——一个技能的 execute 方法内部可以调用其他技能。这使得复杂工作流可以通过组合简单技能来构建，而非从头编写。例如，一个 ResearchSkill 可以内部调用 SummarizeSkill 来总结搜索结果，再调用 OutlineSkill 来组织报告结构。

4.5 中间件系统

中间件系统允许在不修改 Agent/Tool 代码的前提下拦截和修改调用。Middleware 基类定义了四个钩子：before_agent/after_agent（包裹 Agent.arun）和 before_tool/after_tool（包裹 Tool.arun）。MiddlewarePipeline 按洋葱模型执行——before_* 钩子按注册顺序执行（先注册先执行），after_* 钩子按逆序执行（后注册先执行）。内置五个中间件：LoggingMiddleware（结构化日志）、TimingMiddleware（延迟追踪）、RateLimitMiddleware（令牌桶限流）、CacheMiddleware（按参数哈希缓存工具结果）、TracingMiddleware（OpenTelemetry 风格 span 追踪）。

中间件的核心价值在于横切关注点的分离。例如，限流逻辑不需要侵入每个工具的实现，只需注册一个 RateLimitMiddleware 即可全局生效。缓存同理——CacheMiddleware 通过对 (tool_name, args) 取哈希作为缓存键，命中时直接返回缓存结果跳过实际执行。这种设计使得性能优化、可观测性、安全控制等横切逻辑可以独立开发和部署。

4.6 插件扩展系统

插件系统让第三方包通过标准 Python 机制扩展框架。Plugin 基类定义了 name、version 属性和 register/on_unload 生命周期方法。PluginContext 是注册中心，提供 register_agent/register_tool/register_skill/register_middleware/register_node 等方法。PluginManager 负责发现和加载插件，支持三种发现方式：entry_points（通过 pyproject.toml 的 [project.entry-points.'deerflow.plugins'] 声明）、load_module（传入模块名）、load_dir（扫描目录中的 deerflow_*.py 文件）。

插件加载后，PluginManager 提供 build_middleware_pipeline() 和 build_graph() 方法，将所有插件注册的中间件和图节点组装成运行时。这种设计使得企业可以将内部工具（如私有数据库查询、内部 API 调用）封装为插件包，通过 pip install 后自动集成。插件的 on_unload 钩子支持热卸载——在运行时动态移除功能而不重启服务。load_all(llm, plugin_dir) 是便捷入口，一次性加载 entry_points 和指定目录的所有插件。

4.7 Harness 测试框架

Harness 是框架的测试评估层，包含四个组件。Datasets 模块定义 TestCase（id/question/reference/category/difficulty/expected_tools）和 Dataset 集合，内置 SimpleQADataset（10 个手工 Q&A;）和 GAIADataset（GAIA 基准格式），支持 to_jsonl/from_jsonl 持久化。Evaluator 模块提供三种评估器：ExactMatchEvaluator（归一化后精确匹配）、ContainsEvaluator（关键事实覆盖率）、LLMJudgeEvaluator（用 LLM 作裁判，返回 0-100 分加理由）。

Runner 是并发执行器，用 asyncio.Semaphore 控制并发度，对每个测试用例调用 graph.ainvoke，计时并提取 trace 中的 token 用量，然后运行评估器打分，汇总成 RunReport（pass_rate/mean_score/mean_latency/total_tokens/by_category）。FastAPI 服务暴露 /run（单次运行）、/eval（数据集评估）、/stream（WebSocket 流式）、/datasets（数据集列表）、/health（健康检查）五个端点。CLI 工具 deerflow-harness 提供 serve（启动服务）和 eval（离线评估）两个子命令。

附录 API 参考与快速上手

A.1 公共 API 一览

模块	主要类/函数	用途
core.graph	StateGraph, CompiledGraph, GraphInterrupt	状态机构建与执行
core.agent	Agent, AgentResponse, AgentStep	ReAct 智能体基类
core.memory	Memory, Checkpointer, InMemoryCheckpointer	短期记忆与持久化
core.advanced_memory	MemorySystem, LongTermMemory, EpisodicMemory	四层记忆系统
core.orchestrator	Orchestrator	多智能体路由编排
llm.client	LLMClient, LLMMessage, LLMResponse	OpenAI 兼容 LLM 客户端
tools	Tool, ToolRegistry, FunctionTool	工具系统与注册表
tools.mcp	MCPClient, MCPServer	MCP 协议客户端/服务端
agents	CoordinatorAgent, PlannerAgent, ...	六个专门化智能体
agents.research_graph	build_research_graph()	研究工作流图构建器
skills	Skill, SkillRegistry, SummarizeSkill, ...	技能系统
middleware	Middleware, MiddlewarePipeline, ...	中间件系统
plugins	Plugin, PluginManager, load_all()	插件扩展系统
harness	Runner, Evaluator, create_app()	测试评估框架

A.2 快速上手

安装与基本使用：

```
# ■■
cd deerflow_py && pip install -e .

# CLI ■■■■ OPENAI_API_KEY■■ key ■■■■ MockLLM■
deerflow "Summarise 2024 LLM agent trends"

# ■■■■
import asyncio
from deerflow import build_research_graph, LLMClient

async def main():
    llm = LLMClient.from_env()
    graph = build_research_graph(llm)
    result = await graph.ainvoke({
        'messages': [{'role': 'user', 'content': 'your question'}]
    })
    print(result.state['report'])

asyncio.run(main())
```

自定义插件示例：

```
# my_plugin.py
from deerflow.plugins import Plugin, PluginContext
from deerflow.middleware import LoggingMiddleware

class MyPlugin(Plugin):
    name = 'my-plugin'
    version = '1.0.0'

    def register(self, ctx: PluginContext) -> None:
        ctx.register_agent('my_agent', MyAgent)
        ctx.register_tool('my_tool', MyTool())
        ctx.register_skill('my_skill', MySkill(ctx.llm))
        ctx.register_middleware(LoggingMiddleware())

# ■■■■
from deerflow import load_all
pm = load_all(llm, plugin_dir='./plugins')
```

A.3 测试验证

项目包含 24 个单元测试，覆盖状态机（通道去重、线性图、条件路由、中断恢复）、工具系统（Schema 推断、注册表分发、沙箱执行与超时）、智能体（ReAct 循环、研究图端到端、Critic 修订循环、流式输出）和 Harness（数据集、评估器、Runner）四大模块。运行命令：python -m pytest tests/ -v，全部 24 个测试通过。

A.4 项目结构

路径	说明
src/deerflow/core/	状态机、Agent、记忆、编排器（~1300行）
src/deerflow/llm/	LLM 客户端（~322行）
src/deerflow/tools/	搜索/爬虫/沙箱/MCP（~900行）
src/deerflow/agents/	六个智能体+研究图（~440行）
src/deerflow/skills/	技能系统（~300行）
src/deerflow/middleware/	中间件系统（~250行）
src/deerflow/plugins/	插件系统（~250行）
harness/	测试评估框架（~620行）
examples/	三个可运行示例
tests/	24个单元测试