

火龙果科技

IT 人员的学习中心 (uml.org.cn)

基于大模型的 Agent 技术应用开发

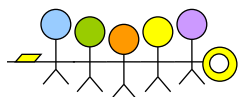
—— 从原理、框架到工程实践的完整指南 ——



科技追踪

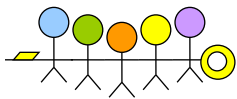
Technical Trackin

编制日期：2026 年 9 月

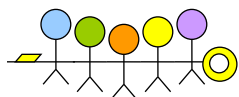


目录

目录	2
第 1 章 企业级 AI Agent 智能体原理	4
1.1 从 LLM 到 Agent: 范式的跃迁	4
1.2 AI Agent 的核心组成	5
1.3 企业级 Agent 的核心特征	5
1.4 智能体的分类	6
第 2 章 基于大模型的 Agent 技术框架	6
2.1 框架全景与分层	7
2.2 LangGraph: 图状编排与有状态流程	8
2.3 CrewAI: 基于角色的团队协作	8
2.4 AutoGen、LlamaIndex 与 Semantic Kernel	9
2.5 框架选型建议	9
第 3 章 如何基于 LangGraph 和 CrewAI 构建多 Agent 智能体	10
3.1 多 Agent 的三种编排模式	10
3.2 用 CrewAI 构建角色化团队	10
3.3 用 LangGraph 构建有状态编排	11
3.4 混合架构: CrewAI + LangGraph + A2A	11
第 4 章 AI 智能体系统的架构设计与模式应用	12
4.1 上下文工程 (Context Engineering) 基础理论	12
4.2 AI 智能体的记忆系统	13
4.3 智能体设计模式	14
4.4 基于多模态构建 Agent	14
第 5 章 基于低代码平台开发 Agent 智能体	15
5.1 低代码 Agent 平台概览	15
5.2 基于字节 Coze 构建智能体	16
5.3 基于开源 Dify 构建智能体	16
5.4 Agent 实战: 企业专属领域的智能客服	16
第 6 章 Agent 智能体协议和 Skill	17
6.1 MCP 原理与实战: 高效 Agent 开发	17
6.2 A2A 协议原理和应用案例分析	18
6.3 Agent Skills 核心技术	19
第 7 章 OpenClaw、Claude Code、Manus 智能体原理	19
7.1 Manus: 把文件系统当作无限记忆	20
7.2 OpenClaw: 为“永不停止”设计的激进压缩	20
7.3 Claude Code: 用缓存换精度的无损方案	20
7.4 OpenAI: 标准化的会话管理工具集	21
7.5 四种方案的取舍对比	21
第 8 章 Manus 产品分析与 OpenManus 源代码分析	22
8.1 Manus 产品分析	22
8.2 OpenManus 架构概览	23
8.3 核心模块源码分析	23
8.4 典型应用场景	24
第 9 章 Harness 工程实践方法	24
9.1 为什么需要 Harness	25



9.2 Hamess 的四大支柱.....	25
9.3 五大核心概念（五支柱详述）	26
9.4 行业实践案例.....	26
9.5 构建你的第一个 Agent Hamess（六步法）	26
9.6 总结：三层工程的演进.....	27



第 1 章 企业级 AI Agent 智能体原理

生成式人工智能（Generative AI）正在向代理式人工智能（Agentic AI）演进。大语言模型（LLM）不再只是被动的问答引擎，而是演变为具备自主推理、规划与执行能力的智能体（Agent）。本章从智能体的定义出发，剖析其核心组成与在企业级场景下的运行机理。

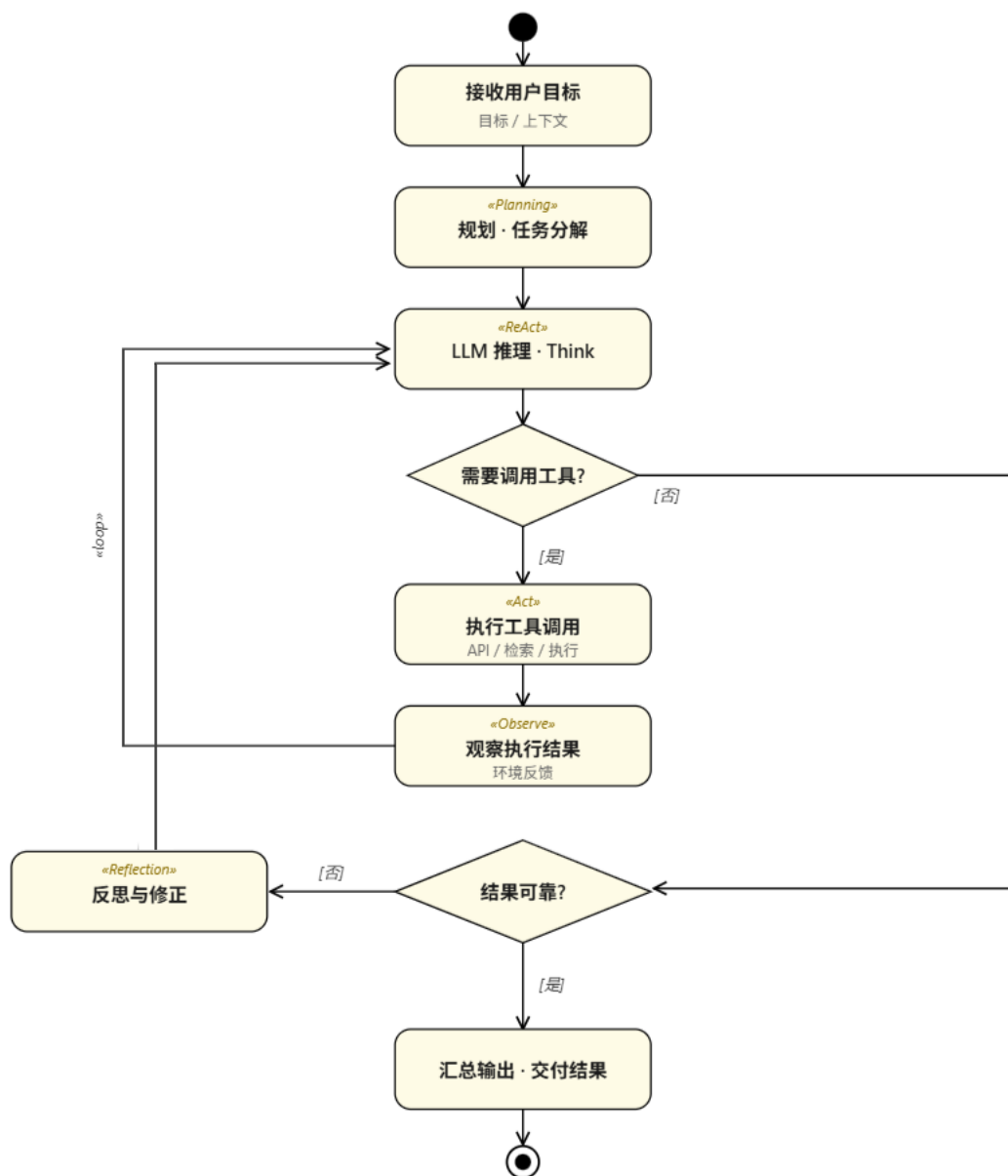
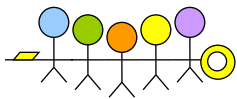


图 1 企业级 AI Agent 核心组成与 ReAct 闭环（UML 活动图）

1.1 从 LLM 到 Agent: 范式的跃迁

传统 LLM 应用遵循“输入—输出”的单轮模式：用户提问，模型生成回答。这种方式在开放域对话中表现优异，但一旦面对需要多步推理、外部信息获取和动作执行的真实业务，就会暴露出三大短板——没有记忆、无法行动、缺乏目标导向。



Agent（智能体）则是将 LLM 与“记忆、规划、工具调用、反思”等能力组合而成的闭环系统。它能够把一个抽象目标自主拆解为可执行的步骤序列，并在执行过程中根据环境反馈不断修正，直至任务完成。

业界常把这一演进概括为三个阶段的范式转移：

- **Prompt Engineering（提示工程）**：关注“说什么”，解决单次对话的指令质量；
- **Context Engineering（上下文工程）**：关注“知道什么”，动态组装模型可用的上下文；
- **Harness Engineering（harness 工程）**：关注“在什么环境里做事”，通过系统级约束与状态管理让 Agent 在生产环境中可靠运行。

1.2 AI Agent 的核心组成

一个典型的企业级 Agent 由五个核心模块构成，合称“Agent = LLM + 记忆 + 规划 + 工具 + 反思”：

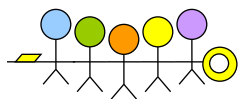
模块	职责	关键能力
LLM（大脑）	语言理解与推理决策，生成行动意图	指令遵循、推理、代码生成
规划（Planning）	将目标分解为子任务并安排执行顺序	任务分解、思维链 CoT、反思 ReAct
记忆（Memory）	保存与召回历史信息与领域知识	短期记忆、长期记忆、向量检索
工具（Tools）	连接外部世界，读写数据与执行动作	API、检索、代码执行、浏览器
反思/评估	对执行结果进行校验与自我修正	自我批评、验证循环、重试

这五个模块通过“感知—思考—行动—观察”（Perceive–Think–Act–Observe）的循环持续运转：Agent 接收用户输入与上下文，由 LLM 推理出下一步动作，调用工具作用于环境，再观察执行结果，据此决定继续、修正或结束。ReAct（Reason + Act）正是这一循环最具代表性的实现范式。

1.3 企业级 Agent 的核心特征

与个人实验性 Agent 相比，企业级智能体对可靠性、可控性与可运维性提出了更高要求：

1. **目标驱动与自主性**：能在明确边界内自主规划路径，而非固定脚本。
2. **工具连接能力**：通过标准化方式接入企业数据库、业务系统、知识库与 SaaS。
3. **状态与记忆持久化**：跨会话保持上下文与用户偏好，崩溃后可恢复。
4. **安全与合规**：权限边界、数据隔离、审计日志，满足企业治理要求。
5. **可观测与可控**：完整的追踪、成本统计、人工在环（Human-in-the-Loop）审批。



1.4 智能体的分类

按能力与形态，Agent 可分为以下几类：

- **单智能体 (Single Agent)**：一个 LLM 主导，配合工具完成线性任务，适合目标明确、流程简单的场景。
- **多智能体 (Multi-Agent)**：多个专业 Agent 协作分工，通过编排器或点对点通信完成复杂任务。
- **工作流型 (Workflow)**：以预定义的图/流程编排为主，确定性高、可审计，适合规则稳定的业务。
- **自主型 (Autonomous)**：以 LLM 动态决策为主，灵活性强，但可控性较低，需更强的护栏。

实践中的经验法则：先用单智能体 + 工作流解决确定性问题，只有当任务确实需要“专业分工、动态路由、并行协作”时，才引入多智能体架构，避免过度工程化。

第 2 章 基于大模型的 Agent 技术框架

围绕上述原理，业界涌现出丰富的 Agent 开发框架。本章梳理主流框架的定位与技术栈，帮助在不同场景下做出合理选型。

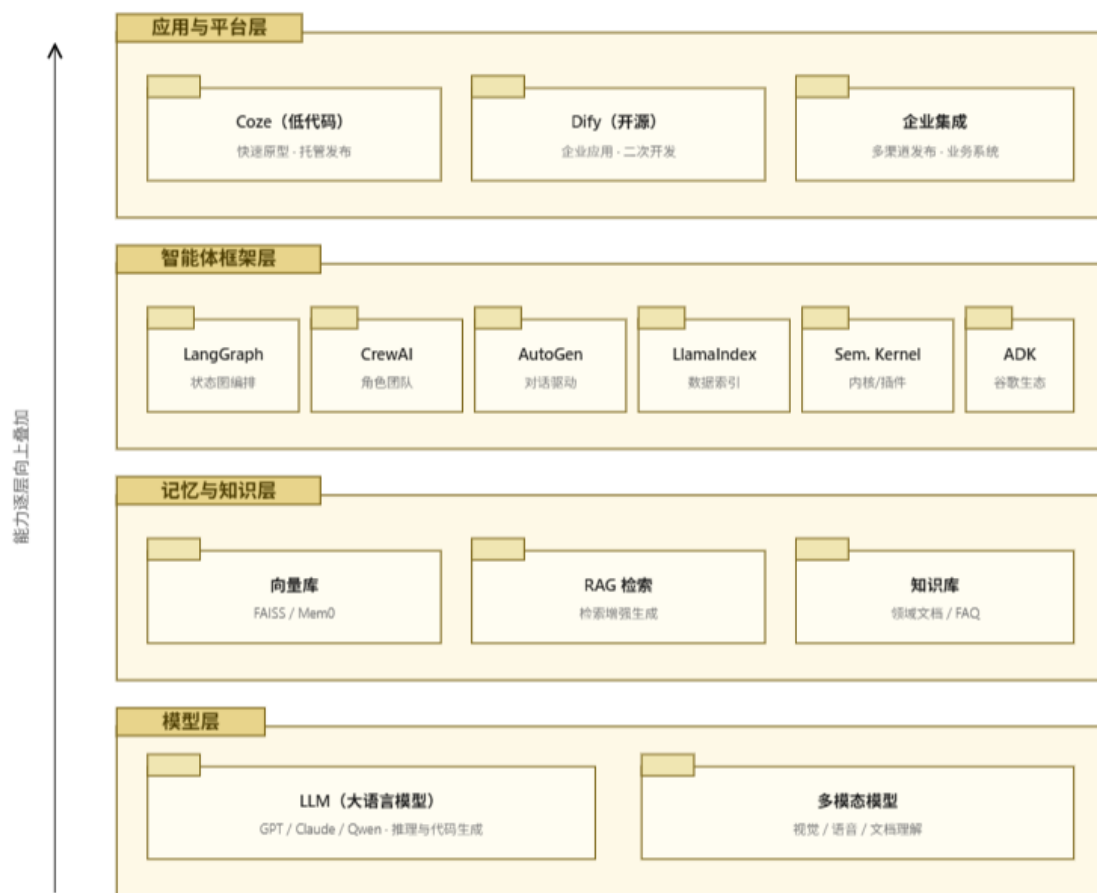
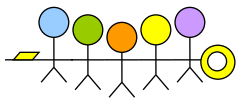
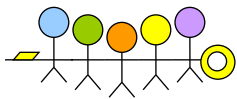


图 2 Agent 技术栈四层架构 (UML 包图)

2.1 框架全景与分层

从底层到上层，Agent 技术栈通常可分为四层：模型层（LLM/多模态模型）、记忆与知识层（向量库、RAG）、智能体框架层（编排与运行）、应用与平台层（低代码、企业集成）。框架的差异主要体现在“编排范式”上：

框架	核心范式	定位
LangChain / LangGraph	链式调用 + 状态图	通用编排，图状工作流与有状态多 Agent
CrewAI	角色扮演团队	基于角色的协作式多 Agent，声明式
AutoGen (微软)	对话驱动	多 Agent 通过对话协作，支持人机协同
LlamaIndex	数据索引 + Agent	RAG 与数据接入见长，Agent 能力持续增强
Semantic Kernel (微软)	内核/插件	面向 .NET/Python 的企业级内核



框架	核心范式	定位
Google ADK	Session + Agent	谷歌生态，Session/Memory 原生抽象
OpenAI Agents SDK	会话管理工具集	标准化的会话状态与工具调用

2.2 LangGraph：图状编排与有状态流程

LangGraph 是 LangChain 生态中面向“有状态多步骤工作流”的编排框架。它采用节点（Node）+ 边（Edge）的有向图模型，开发者显式定义流程、条件路由与跨调用的持久状态。

适用场景：需要基于中间结果做决策、在复杂对话中维护上下文、处理分支/循环逻辑的 Agent。其 StateGraph 允许把对话状态、检索结果、工具输出统一保存在 TypedDict 状态对象中，并支持断点续跑与检查点（Checkpoint）持久化。

```
from langgraph.graph import StateGraph, END
from typing import TypedDict, Annotated, List
import operator

class AgentState(TypedDict):
    messages: Annotated[List, operator.add]
    research: str
    analysis: str

def research_node(state):
    return {"research": researcher.run(state["messages"])}

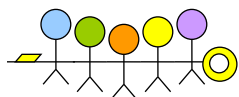
# 构建图: research -> analysis -> END
workflow = StateGraph(AgentState)
workflow.add_node("research", research_node)
workflow.set_entry_point("research")
workflow.add_edge("research", "analysis")
workflow.add_edge("analysis", END)
app = workflow.compile()
```

2.3 CrewAI：基于角色的团队协作

CrewAI 以“角色（Role）+ 任务（Task）+ 团队（Crew）”为核心抽象，让多个专职 Agent 围绕共同目标协作。每个 Agent 被赋予角色、目标与背景故事，CrewAI 自动处理任务委派与 Agent 间通信，适合快速搭建研究、写作、分析类流水线。

```
from crewai import Agent, Task, Crew

researcher = Agent(role="研究专员", goal="搜集 {topic} 的权威资料",
                   backstory="资深行业研究员", tools=[search_tool])
writer = Agent(role="撰稿人", goal="撰写清晰的分析报告", tools=[writing_tool])
```



```
task1 = Task(description="研究 {topic}", agent=researcher)
task2 = Task(description="撰写报告", agent=writer, context=[task1])

crew = Crew(agents=[researcher, writer], tasks=[task1, task2], verbose=True)
result = crew.kickoff(inputs={"topic": "AI Agent 市场趋势"})
```

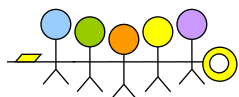
2.4 AutoGen、LlamaIndex 与 Semantic Kernel

- **AutoGen:** 以“对话即编排”为核心，Agent 之间通过消息对话驱动协作，原生支持 Human-in-the-Loop，适合需要人审的多 Agent 场景。
- **LlamaIndex:** 从数据索引/RAG 起家，擅长将企业私有数据接入 LLM，其 Agent 模块（LlamaIndex Agent）可与工作流结合，适合知识密集型企业应用。
- **Semantic Kernel:** 微软推出的企业级 AI 编排内核，以“技能（Skill）/插件”为中心，天然对接 Azure OpenAI 与企业身份体系，适合 .NET 技术栈。

2.5 框架选型建议

选型应遵循“从简单到复杂”的原则，优先用单 Agent + 工作流解决确定性问题：

- **快速原型 / 角色化流水线：**首选 CrewAI，声明式定义角色与任务，上手最快；
- **复杂状态、条件分支、需持久化：**选用 LangGraph，获得精细的流程控制；
- **强数据/RAG 场景：**LlamaIndex 或结合 LangChain 生态；
- **微软 / Azure 企业栈：**Semantic Kernel + AutoGen；
- **混合架构（当前主流）：**用 CrewAI 快速定义 Agent，再用 LangGraph 在其上做动态路由与生产级编排。



第3章 如何基于 LangGraph 和 CrewAI 构建多 Agent 智能体

多智能体系统的价值在于“关注点分离”——每个 Agent 专注自己最擅长的任务，并通过编排协作完成单体难以胜任的复杂工作。本章结合两种主流编排模式，给出可落地的构建方法。



图3 多 Agent 的三种编排模式 (UML 活动图)

图3 多 Agent 三种编排模式 //活动图

3.1 多 Agent 的三种编排模式

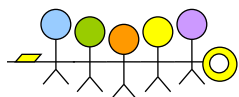
模式	机制	适用场景
分层编排 (Manager)	管理器按上下文动态路由到下一 Agent	任务差异大、需灵活调度
顺序流水线 (Pipeline)	Agent 按固定顺序依次处理	可预测、依赖明确的任务
共识/投票 (Consensus)	多个 Agent 提案并投票达成一致	高风险、质量关键的决策

实践中的常见反模式需要规避：Agent 数量过多导致协调开销爆炸、职责边界模糊、缺少失败处理、把本可串行的任务强行并行、忽略 N 个 Agent 带来的 N 倍 Token 成本。经验法则是**从 2-3 个 Agent 起步，确有必要时才增加**。

3.2 用 CrewAI 构建角色化团队

CrewAI 的 Pipeline 模式适合任务依赖清晰的场景。下面的示例构建一个“研究—分析—写作”三级流水线，后一任务通过 context 依赖前一任务的产出：

```
# 定义专职 Agent
```



```
researcher = Agent(role="Research Specialist", goal="Gather info on {topic}",
                   backstory="Expert researcher", tools=[search_tool, scrape_tool])
analyst = Agent(role="Data Analyst", goal="Synthesize findings into insights",
                tools=[analysis_tool])
writer = Agent(role="Technical Writer", goal="Create compelling reports",
               tools=[writing_tool])

# 任务通过 context 建立依赖链
research_task = Task(description="Research {topic} thoroughly", agent=researcher,
                     expected_output="Research findings with sources")
analysis_task = Task(description="Analyze findings", agent=analyst,
                     context=[research_task])
writing_task = Task(description="Write executive summary", agent=writer,
                    context=[research_task, analysis_task])

crew = Crew(agents=[researcher, analyst, writer],
            tasks=[research_task, analysis_task, writing_task], verbose=True)
result = crew.kickoff(inputs={"topic": "AI market trends"})
```

CrewAI 还支持 **Manager 模式**（由管理 Agent 动态分派子任务）与 **Hierarchical 流程**，可根据复杂度灵活选择。

3.3 用 LangGraph 构建有状态编排

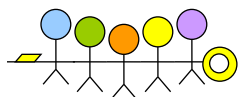
对于需要条件分支、循环、状态持久化的场景，LangGraph 的图模型更为合适。每个节点是一个 Agent 或一个工具调用，边定义流转规则，条件边（`add_conditional_edges`）实现动态路由：

```
class AgentState(TypedDict):
    input: str
    research: str
    analysis: str
    output: str

def research_node(state): return {"research": researcher.run(state["input"])}
def analysis_node(state): return {"analysis": analyst.run(state["research"])}
def writing_node(state): return {"output": writer.run(state["analysis"])}

workflow = StateGraph(AgentState)
workflow.add_node("research", research_node)
workflow.add_node("analysis", analysis_node)
workflow.add_node("writing", writing_node)
workflow.set_entry_point("research")
workflow.add_edge("research", "analysis")
workflow.add_edge("analysis", "writing")
workflow.add_edge("writing", END)
app = workflow.compile()
result = app.invoke({"input": "Analyze AI market trends"})
```

3.4 混合架构：CrewAI + LangGraph + A2A



生产级系统常将两者结合：用 CrewAI 快速定义角色与任务，再用 LangGraph 在其上做动态路由、状态管理与生产级控制。在跨框架/跨组织的场景中，还可通过 ****A2A 协议**** 将不同框架的 Agent 互联：

- **ADK 控制室**：顶层编排器，负责规划并在失败时重新规划；
- **LangGraph 规划器**：分析意图、路由请求、生成报告的状态机；
- **CrewAI 执行团队**：按角色调用工具的专职 Agent；
- **A2A (JSON-RPC 2.0)**：上述组件间的通信桥梁，实现异构框架互通。

这种“关注点分离 + 框架灵活性 + 可扩展性”的组合，代表了 2026 年构建生产级多 Agent 系统的主流实践。

第 4 章 AI 智能体系统的架构设计与模式应用

架构设计决定了 Agent 能处理多复杂的任务、能跑多长时间、能保持多高的精度。本章围绕上下文工程、记忆系统、设计模式与多模态四条主线展开。

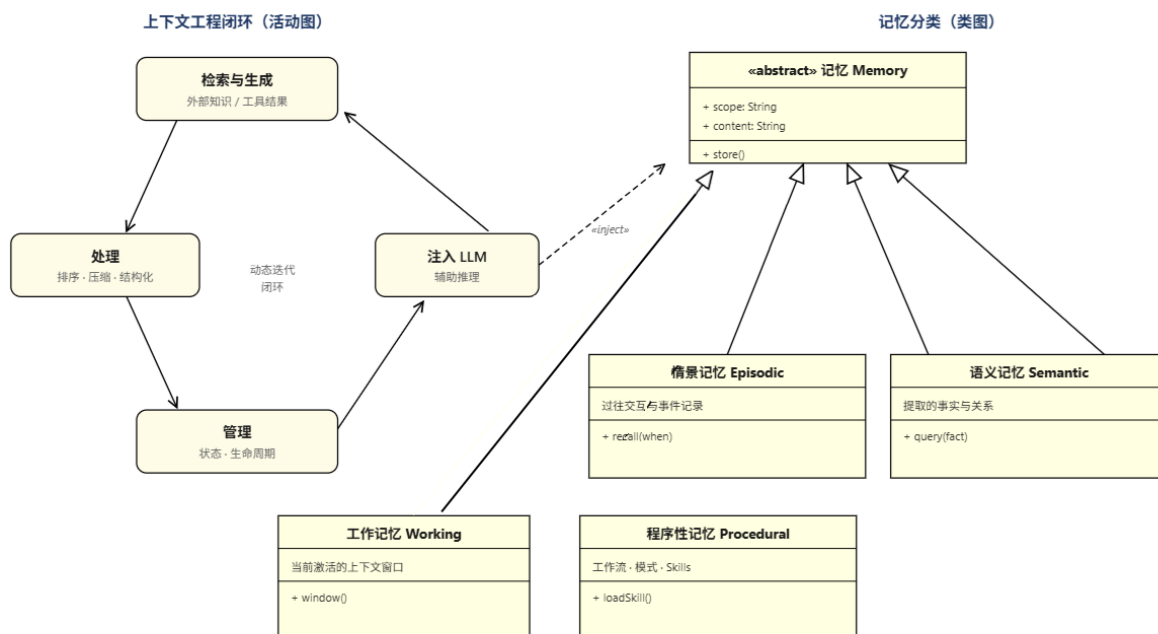
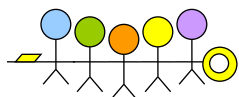


图 4 上下文工程闭环与记忆系统 (UML 活动图 + 类图)

4.1 上下文工程 (Context Engineering) 基础理论

上下文工程是一种系统化构建、处理并动态管理上下文信息，以驱动 LLM 实现精准推理与决策的架构方法。它在性能、成本与准确性之间寻找最佳平衡。

4.1.1 上下文的构成



上下文工程由**输入、记忆、输出**三大部分组成：

- **输入**：系统指令、外部知识检索、工具定义、全局状态、用户输入；
- **记忆**：短期记忆与长期记忆，用于存储和召回历史信息；
- **输出**：工具执行结果与模型生成的结构化内容。

核心工作流是一个动态迭代闭环：检索与生成 → 处理（优先级排序、结构化转化、压缩） → 管理（状态更新、生命周期控制）。

4.1.2 狭义 vs 广义上下文工程

- **狭义**：特指对短期记忆（会话历史）的压缩、卸载、摘要等运行时处理，解决上下文窗口限制与 Token 成本；
- **广义**：还包括模型选择、Prompt 优化、知识库构建、工具集构建等推理前的上下文优化手段。

4.1.3 三大核心策略

策略	做法	特点
上下文缩减	保留预览内容 / 总结摘要	降 Token，但有信息丢失
上下文卸载	原文存外部存储，上下文只保留引用	干净、可恢复、按需加载
上下文隔离	拆分为多个子 Agent，各自独立上下文	类似微服务，避免共享大上下文

主流框架提供了对应的工程实现：Google ADK 通过 EventsCompactionConfig 设置压缩策略；LangChain 通过 SummarizationMiddleware 中间件；AgentScope 的 AutoContextMemory 则自动应用 6 种渐进式压缩策略。

4.2 AI 智能体的记忆系统

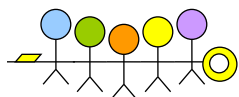
4.2.1 记忆的分类

记忆系统使 Agent 能在单次对话中保持连贯（短期记忆），并跨会话记住用户偏好与领域知识（长期记忆）。按 2026 年 Mem0 的生产级分类法，记忆可分为四类：

- **情景记忆（Episodic）**：过往交互与事件记录，可查询“某时发生了什么”；
- **语义记忆（Semantic）**：提取的事实与关系，如用户偏好、领域知识、项目状态；
- **程序性记忆（Procedural）**：如何做事—— workflow、模式、Agent Skills；
- **工作记忆（Working）**：当前激活的上下文窗口。

4.2.2 短期与长期记忆的双向交互

各框架（Google ADK、LangChain、AgentScope）实现细节不同，但都遵循相同架构模式：



6. 推理前加载：根据当前 query 从长期记忆加载相关信息；
7. 上下文注入：检索到的信息注入短期记忆辅助推理；
8. 记忆更新：推理完成后将短期记忆写回长期记忆；
9. 信息处理：长期记忆模块结合 LLM + 向量化进行提取与检索。

实现组件：长期记忆通常是独立第三方组件，提供完整的 Record（写入）与 Retrieve（检索）能力，常见产品包括 Mem0、Zep、Memos、ReMe 等，Agent 框架通过 API 集成。

4.2.3 记忆管理的最佳实践

生产系统需警惕“记忆晋升（Memory Promotion）”风险——把会话级观察提升为持久存储时，应设置权限闸门：Session → 用户级（允许偏好/草稿）、Session → 租户级（仅允许已验证事实）、Session → 全局级（禁止运行时写入）。否则会出现“静默护栏漂移”与“先例投毒”。

此外，压缩是反应式的（上下文增长后再缩小），而外部记忆是预防式的（多数信息从不进入上下文）。二者结合——会话内压缩 + 跨会话外部记忆——是当前公认的参考架构。

4.3 智能体设计模式

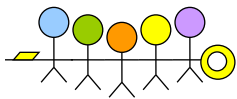
借鉴软件工程与业界实践，Agent 系统常用以下设计模式：

模式	说明
ReAct（推理-行动）	Think → Act → Observe 循环，边推理边调用工具
Plan-and-Execute	先整体规划，再逐步执行并校验
Reflection / Critique	Agent 自我批评或引入批评者 Agent 修正输出
Tool Use / Function Calling	以结构化方式调用外部工具与 API
RAG（检索增强生成）	从知识库检索相关片段增强上下文
Orchestrator-Worker	编排器分发、Worker 执行的分层架构
HITL（人工在环）	在高风险决策点引入人工审批
Progressive Disclosure	按需加载技能与权限，最小权限原则

一个高价值的工程洞察：仅通过改进 Harness（增加自我验证循环与循环检测、不改变模型），编码 Agent 在 Terminal Bench 2.0 上的得分可从 52.8% 跃升至 66.5%。这说明**设计模式与工程约束本身就能显著提升 Agent 表现**。

4.4 基于多模态构建 Agent

多模态 Agent 将视觉、语音、文档等非文本模态纳入感知与行动闭环，是操作真实世界的关键



能力。

- **视觉感知**：通过多模态 LLM 理解屏幕截图、GUI 布局、文档图表，如 OpenClaw 的“视觉—思考—行动”循环；
- **语音交互**：ASR + TTS 实现自然语音对话，适用于客服、助手场景；
- **文档多模态**：解析 PDF、表格、图片中的结构化信息，结合 RAG 使用；
- **跨模态工具**：图像生成、视频理解、OCR 等工具作为 Agent 的可调用能力。

多模态让 Agent 从“只读文本”升级为“能看、能听、能操作图形界面”，是通用计算机控制（Computer Use）类 Agent 的核心技术基础。

第 5 章 基于低代码平台开发 Agent 智能体

低代码平台大幅降低了 Agent 的构建门槛，使非技术人员也能快速搭建并发布智能体。本章对比主流平台，并以企业智能客服为例给出实战思路。

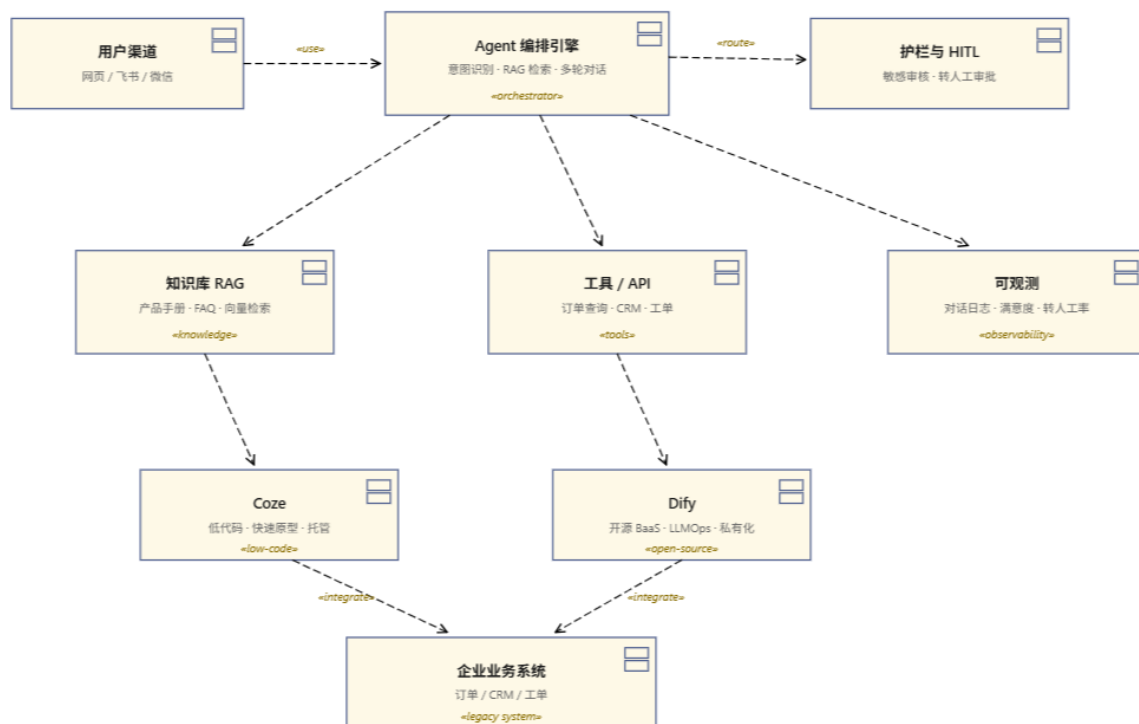
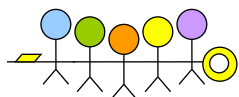


图 5 企业智能客服系统架构（UML 组件图）

5.1 低代码 Agent 平台概览

平台	定位	架构/模型	适用场景
Coze（扣子）	轻量低代码，快速构建	云端托管、封闭源码	原型、聊天机器人、



平台	定位	架构/模型	适用场景
			自动化
Dify	开源企业级 AI 平台	开源 BaaS、LLMOps	企业应用、复杂工作流、二次开发
FastGPT	知识库问答专长	开源、知识库为核心	企业内部知识管理、私有化

5.2 基于字节 Coze 构建智能体

Coze（扣子）是字节跳动推出的低/零代码 Agent 开发平台，主打低门槛、高灵活、快速落地。

- **核心能力：**拖拽式可视化编排、丰富的插件生态、知识库与数据库支持、多模型接入（GPT 系列、Claude 等）、工作流模式。
- **开发方式：**拖拽式零代码+低代码，界面简洁直观，学习难度低，非技术人员可快速上手。
- **适用场景：**电商、零售、企业服务等行业的客服、办公、营销类智能体；中小企业快速验证，中大型企业分场景定制。

构建流程通常为：创建 Bot → 配置人设与提示词 → 添加插件/工作流 → 上传知识库 → 调试 → 发布到多渠道（网页、飞书、微信等）。

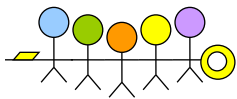
5.3 基于开源 Dify 构建智能体

Dify 是开源低代码企业级 AI 平台，定位于中小企业轻量化落地与快速知识库构建，是国内开源 Agent 领域的标杆。

- **核心能力：**支持 80+ 大模型无缝接入、RAG 流水线优化、可视化编排、数据隔离与精细化权限管理、高并发。
- **架构特点：**后端即服务（BaaS）+ LLMOps，社区版永久免费，支持私有化部署，企业版提供商业支持与 SLA。
- **适用场景：**对数据隐私有要求、追求低成本落地的企业，如知识库问答、智能客服、业务数据分析。

Dify 的典型优势在于“开箱即用的生产级能力”——从知识库构建、工作流编排到 API 发布形成完整闭环，且支持通过 API 与现有业务系统集成。

5.4 Agent 实战：企业专属领域的智能客服



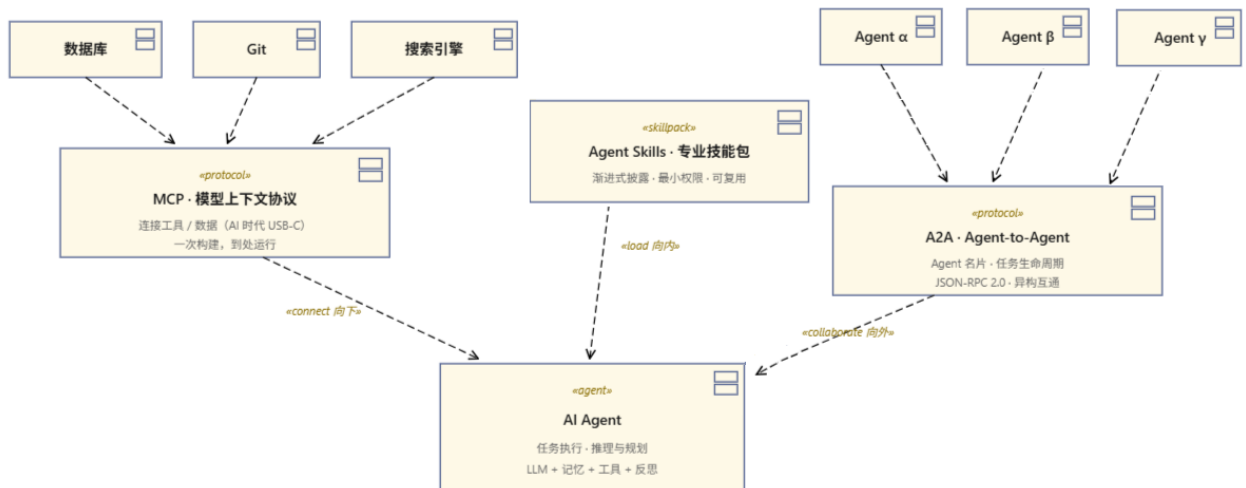
以“某企业产品智能客服”为例，演示基于低代码平台的端到端构建流程：

- 10.需求定义：明确服务范围（产品咨询、售后、退换货）、支持渠道、转人工规则。
- 11.知识库构建：整理产品手册、FAQ、售后政策，切片向量化入库，设置实时更新。
- 12.Agent 编排：配置欢迎语、意图识别、RAG 检索、多轮追问；对超出权限的请求路由至人工。
- 13.工具接入：对接订单查询、CRM、工单系统 API，实现“查订单—办售后”闭环。
- 14.护栏与安全：敏感信息脱敏、回复审核、速率限制、置信度低时转人工。
- 15.评测与上线：准备评测集验证准确率，接入可观测（对话日志、满意度、转人工率），灰度发布。

该实战的核心要点是：****知识库质量决定上限，工具闭环决定能力，护栏与 HITL 决定可靠性****。低代码平台让原型到上线周期从数周缩短至数天。

第 6 章 Agent 智能体协议和 Skill

随着 Agent 生态爆发，MCP、A2A 与 Agent Skills 三项标准技术分别解决了“模型如何感知世界、Agent 如何协作、Agent 如何习得专业能力”三个层面的问题，共同构成现代 AI 基础设施的基石。

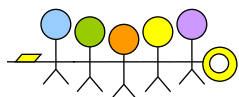


MCP = 模型 ↔ 工具的统一接口 | A2A = Agent ↔ Agent 的协作协议 | Skills = 专业能力的沉淀与复用

图 6 MCP · A2A · Agent Skills 三层技术栈 (UML 组件图)

6.1 MCP 原理与实战：高效 Agent 开发

模型上下文协议（MCP）由 Anthropic 于 2024 年 11 月开源，是 AI 时代的“USB-C 接口”，标准化了单一模型与本地或远程数据源、工具、资源的连接方式，解决“模型如何感知世界”的问



题。其愿景是“一次构建，到处运行”。

6.1.1 核心架构（Client-Host-Server）

- **MCP Host:** 用户交互的 AI 应用（如 Claude Desktop、IDE），为 LLM 提供运行时并发起连接；
- **MCP Client:** 嵌入 Host 的协议适配器，维护与 Server 的 1:1 连接，负责 JSON-RPC 消息翻译、权限协商；
- **MCP Server:** 通往外部世界的桥梁，封装数据源/工具集，定义 AI 能力边界与数据安全防线。

MCP 基于 JSON-RPC 2.0，支持 stdio、SSE 等传输层，标准化三类能力：****资源（Resources，只读数据）****、****工具（Tools，可执行函数）****、****提示词（Prompts，标准化模板）****。

6.1.2 从工具调用到代码执行

面对大规模工具集带来的上下文窗口爆炸，MCP 引入****动态发现 + 代码执行（Code Mode）****：****Agent** 获得代码沙箱与 MCP 客户端接口，按需查询并加载工具定义。这种“渐进式披露”在某些场景可将 Token 消耗从 150,000 降至 2,000（降幅 98.7%），同时增强数据隐私——中间原始数据留在本地，仅聚合结果回传 LLM。

实战：开发者只需为某数据源（如 PostgreSQL、Git、Google Drive）编写一次 MCP Server，即可被任何支持 MCP 的客户端（Claude Desktop、Cursor 等）即插即用。企业应将重心从“为每个应用写 API 包装器”转向“集中构建通用 MCP Server”。

6.2 A2A 协议原理和应用案例分析

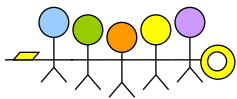
Agent-to-Agent（A2A） 协议由 Google 于 2025 年 4 月推出并捐赠给 Linux 基金会，是 Agent 的“通用翻译器 / 互联网协议”，标准化不同架构、不同归属的 Agent 之间的发现、能力协商与协作，解决“代理如何与社会协作”的问题。

6.2.1 核心机制

- **Agent Card（代理名片）：**托管于 `/.well-known/agent.json` 的标准化 JSON 元数据，含身份、服务端点、能力声明、认证要求、输入输出模态，支持动态发现；
- **Task 有状态生命周期：**Submitted → Working → Input-Required → Completed / Failed / Canceled，通过 SSE/Webhook 支持长时异步任务；
- **松散耦合与委托：**不共享内部 Prompt 与记忆，通过能力声明动态匹配协作伙伴。

6.2.2 应用案例

在“零售库存补货”场景中，ADK 控制室作为顶层编排器，通过 A2A 将任务委托给 LangGraph 规划器，再由 CrewAI 执行团队调用工具完成采购——不同框架的 Agent 通过 A2A（JSON-



RPC 2.0) 桥接, 实现异构系统互通。这正是“MCP 向下连接工具、A2A 向外连接同伴”分工的生动体现。

6.3 Agent Skills 核心技术

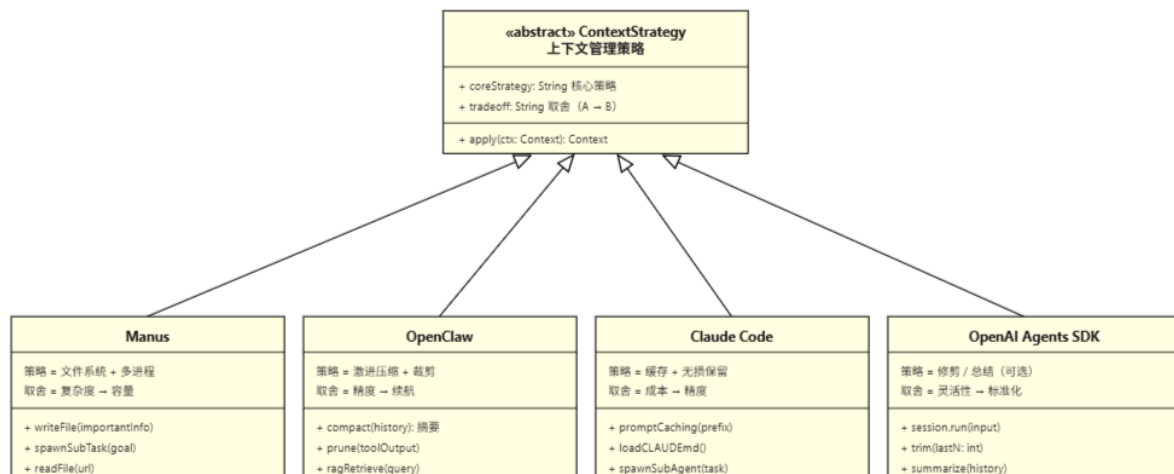
Agent Skills (代理技能) 作为认知方法论层, 通过标准化指令集与渐进式披露 (Progressive Disclosure) 机制, 将领域专家的隐性知识编码为 Agent 可执行的显性流程, 解决“代理如何习得专业行为”的问题。

- **与工具的区别:** 工具是执行接口, Skills 是专业知识、行为规范与经验示例的组合——“能力插件”或“技能包”;
- **渐进式披露:** 按需加载技能与权限, 遵循最小权限原则, 避免一次性塞满上下文;
- **可复用生态:** 以 Markdown 等标准化形式定义, 支持社区贡献与共享 (如 OpenClaw 的 ClawHub 技能市场)。

三者的关系可概括为: ****Agent 负责完成任务, A2A 负责 Agent 间协作, MCP 负责模型与工具的统一连接, Skills 负责专业能力的沉淀与复用****——它们并非竞争, 而是互补的技术栈。

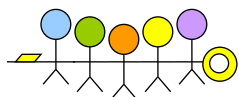
第 7 章 OpenClaw、Claude Code、Manus 智能体原理

本章拆解四款代表性 Agent 产品的记忆与架构设计, 揭示其在“上下文管理”这一核心取舍上的不同哲学。



Manus 换容量 · OpenClaw 换续航 · Claude Code 换精度 · OpenAI 换标准化

图 7 四大产品的上下文管理策略取舍 (UML 类图)



7.1 Manus: 把文件系统当作无限记忆

Manus 的设计思路最为独特——它不在上下文窗口内做文章，而是**把易失的上下文转化为持久的存储**。Manus 把文件系统当成 Agent 的“外挂大脑”，模型主动把重要信息写入文件，需要时再读回。

- **信息卸载：**从上下文中移除整段网页内容，仅保留 URL，需要时重新访问；
- **任务分解：**复杂目标拆分为多个独立子任务，各自创建进程，结果写文件，进程间通过读写文件协作——类似微服务架构；
- **适用场景：**复杂、长周期、需与外部环境深度交互的任务（多网页调研、跨步骤自动化）。

其本质是**用复杂度换容量**，从根本上绕开上下文限制，但系统复杂度也是四者中最高的。

7.2 OpenClaw: 为“永不停止”设计的激进压缩

OpenClaw 是 2026 年初发布的开源、本地优先 Agent 框架（MIT 许可），定位为“真正会做事的 AI”，能操作计算机、执行跨应用复杂任务。

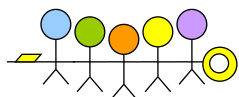
- **架构：**调度中心（Gateway 网关 + Agent 执行器），关注点分离，解耦消息通信、AI 思考与工具执行；
- **多模态控制：**截图 → 多模态 LLM 识别 UI 元素 → 模拟键鼠操作，形成“视觉—思考—行动”循环，摆脱 API 依赖；
- **记忆与技能：**本地 .jsonl 追加记录历史，技能以 Markdown 文件定义，通过 ClawHub 社区共享；
- **上下文策略：**激进的 Compaction（有损摘要）+ Pruning（裁剪工具输出）+ RAG 混合检索，宁可丢细节也要保续航。

OpenClaw 通过 **Tailscale 虚拟组网 + Node 系统**实现跨设备（Mac/Windows/iOS/Android）控制，并内置 Cron 调度实现主动行为。其设计哲学是“用精度换续航”，适合资源受限下的超长自治任务。

7.3 Claude Code: 用缓存换精度的无损方案

Claude Code（Anthropic）走了一条不压缩、不摘要的路，充分利用模型的长上下文能力与 Prompt Caching 机制，尽可能保留原始信息。

- **Prompt Caching：**把项目结构、库定义、代码规范等公共前缀缓存，后续交互无需重新处理，显著降低大上下文的成本与延迟；
- **CLAUDE.md：**系统级持久化指令，每次启动自动加载，确保一致性指导不被压缩遗忘；



- **权限与 Hooks:** 默认只读，通过快照可撤销，生命周期关键点注入自定义脚本（安全扫描、lint、策略）；
- **子 Agent 隔离:** 子 Agent 获得全新上下文+作用域任务描述，输出以结构化摘要回传，防止上下文污染。

Claude Code 的优势在代码场景——修改代码要求高精度，无损方案保证第 20 步仍记得第 3 步的上下文。代价是对模型原生窗口与算力要求高，即**用成本换精度**。

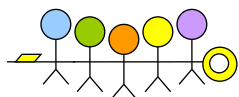
7.4 OpenAI: 标准化的会话管理工具集

OpenAI 的方案更像“官方最佳实践工具箱”，通过 Agents SDK 提供标准化的会话状态管理（Session 对象，反复调用 `session.run()` 自动处理历史）。提供**修剪模式**（保留最近 N 轮，零延迟但有遗忘）与**总结模式**（压缩摘要，保留长期但引入延迟与“上下文中毒”风险）两种策略，配合原生记忆功能，是**用灵活性换标准化**的成熟方案。

7.5 四种方案的取舍对比

产品	核心策略	换取	适用
Manus	文件系统 + 多进程	复杂度 → 容量	长周期、外部交互
OpenClaw	激进压缩 + 裁剪	精度 → 续航	资源受限、超长自治
Claude Code	缓存 + 无损保留	成本 → 精度	代码修改等高精度
OpenAI	修剪 / 总结可选	灵活性 → 标准化	标准多轮对话

核心启示：**上下文管理不是可选优化，而是 Agent 架构的核心设计决策**；文件系统是被低估的记忆介质；有损压缩与无损缓存代表了两种哲学，需按任务特性选型。



第 8 章 Manus 产品分析与 OpenManus 源代码分析

本章先分析 Manus 产品形态，再深入其开源实现 OpenManus 的架构与核心模块，做到“知其然更知其所以然”。

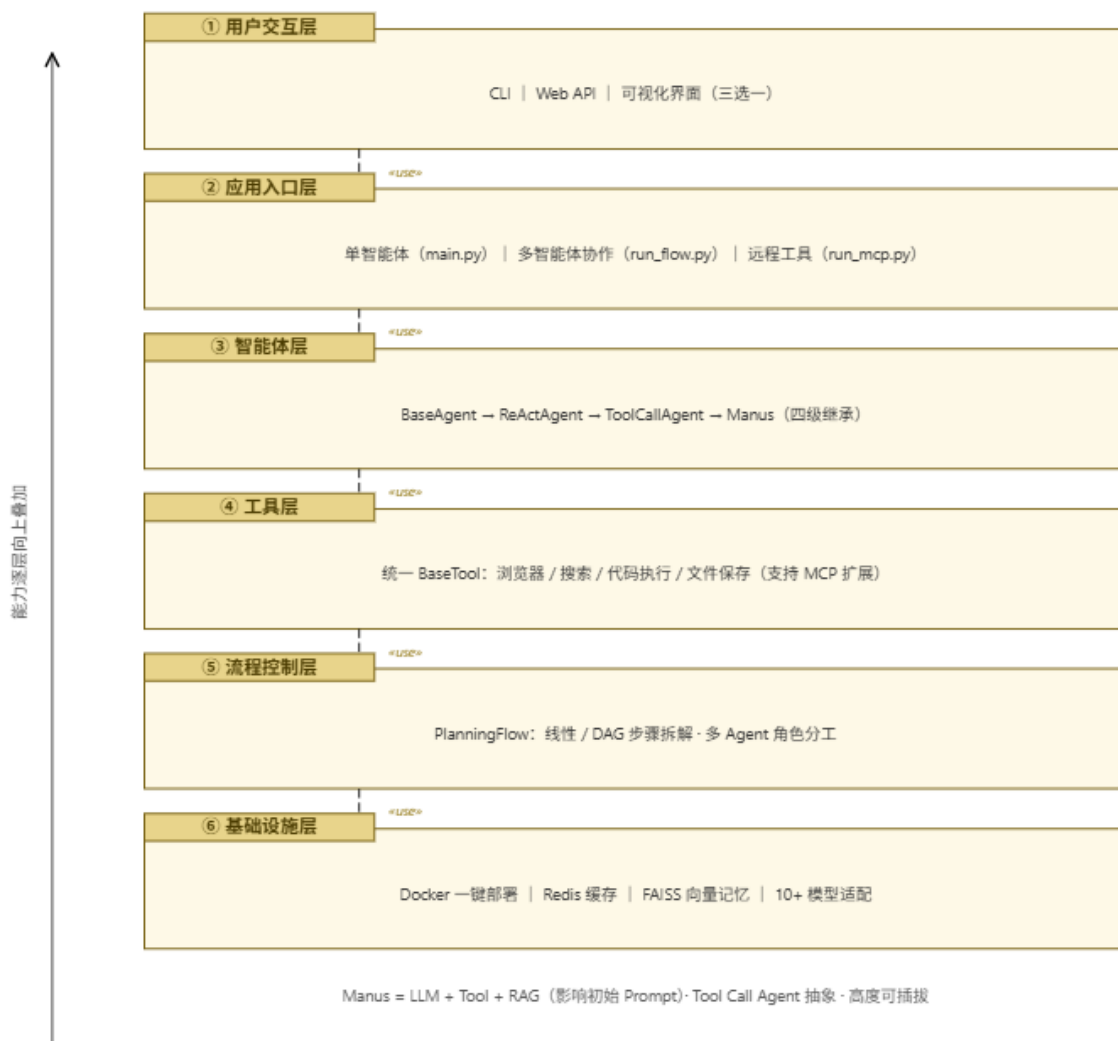
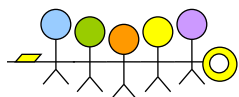


图 8 OpenManus 六层架构 (UML 包图)

8.1 Manus 产品分析

- **定位：**云端托管、零门槛的“数字员工”SaaS 平台，开箱即用、无需部署、稳定可靠；
- **核心能力：**任务驱动的通用 Agent，深度集成 MCP，支持浏览器自动化、代码执行、文件处理；
- **优势：**云端运行、关机也能执行、按用量付费；
- **局限：**隐私数据在云端、定制化相对较低、用量成本。

Manus 的典型工作模式是：用户给出自然语言目标 → 自动规划 → 调用工具（浏览器、代码、文件） → 中间结果写文件 → 汇总交付。其“文件即记忆”的设计让它能处理跨越数十个网页、



多步骤的复杂任务。

8.2 OpenManus 架构概览

OpenManus 是仿照 Manus 的开源项目（MetaGPT 团队维护），定位为开箱即用的多智能体协作平台，可将自然语言需求直接转化为可执行工作流。其核心公式可概括为：****Manus = LLM + Tool + RAG（影响初始 Prompt）****。

OpenManus 采用****六层架构****，逐级叠加推理—行动—工具调用能力：

层级	组成
用户交互层	CLI、Web API、可视化界面三选一
应用入口层	单智能体（main.py）、多智能体协作（run_flow.py）、远程工具（run_mcp.py）
智能体层	BaseAgent → ReActAgent → ToolCallAgent → Manus 四级继承
工具层	统一 BaseTool 接口，浏览器/搜索/代码执行/文件保存，可通过 MCP 扩展
流程控制层	PlanningFlow 自动拆解线性或 DAG 步骤，支持多 Agent 角色分工
基础设施层	Docker 一键部署，Redis 缓存，FAISS 向量记忆，适配 10+ 模型

8.3 核心模块源码分析

8.3.1 Orchestrator（编排器）

编排器负责把用户目标转化为执行计划并驱动流程。PlanningFlow 自动将复杂任务拆分为线性步骤或 DAG（有向无环图），支持多智能体按角色分工，如“项目经理 → 产品经理 → 执行专员”的层级结构。

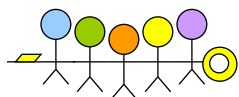
8.3.2 Agents（智能体继承体系）

Agent 层采用经典的四级继承链，逐级叠加能力：

- **BaseAgent**：定义 Agent 的基本接口与生命周期；
- **ReActAgent**：引入 ReAct 的推理—行动循环；
- **ToolCallAgent**：赋予工具调用能力；
- **Manus**：顶层实现，组合完整推理、行动与工具调用。

这种继承设计符合开闭原则，便于扩展自定义 Agent。

8.3.3 Memory（记忆）与 Tools（工具）



- **记忆**：高频知识放 Redis 缓存，长期记忆基于 FAISS 向量库，实现跨会话的语义检索；
- **工具**：统一 BaseTool 接口，内置浏览器、搜索、Python 代码执行、文件保存等 12 类工具，全部可插拔替换，并支持通过 MCP 协议接入外部服务；
- **LLM 适配**：同时支持 DeepSeek、Claude、Qwen 等 10+ 模型，便于切换与成本优化。

8.4 典型应用场景

- **数据采集与可视化**：自然语言 → 自动写爬虫 → 清洗数据 → 生成可视化报告并导出 HTML/PPT；
- **电商运营**：输入“给三款新品写多平台文案” → 市场分析、竞品价格抓取、本地化文案、合规检测全自动完成。

OpenManus 的设计精髓在于“**Tool Call Agent**”的清晰抽象与高度可插拔的工具/模型体系，使其成为学习通用 Agent 架构的优秀范本。

第 9 章 Harness 工程实践方法

Harness Engineering（Harness 工程）是一门设计“包裹在 Agent 周围的系统、约束与反馈循环”的学科，使 Agent 在生产环境中表现可靠。正如其名（马具）——LLM 是那匹强大的马，Harness 则是缰绳、马鞍与马衔，将能量转化为生产力。

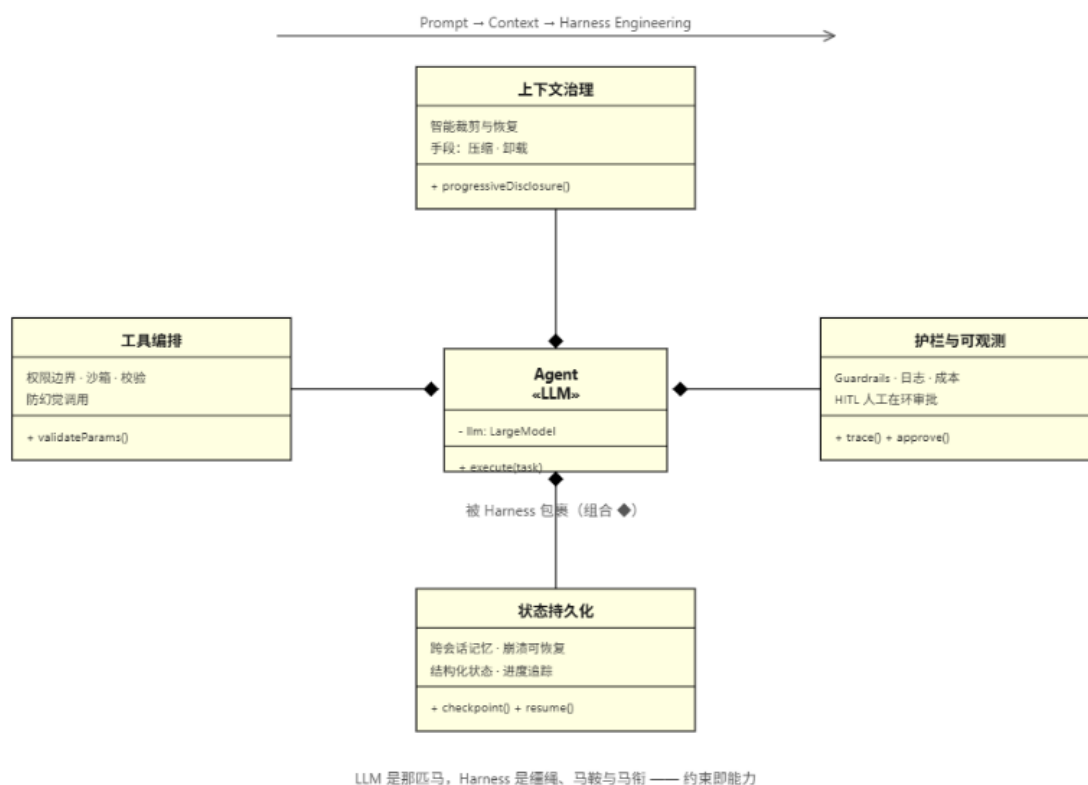
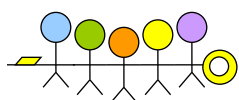


图 9 Harness Engineering 四大支柱 (UML 类图)

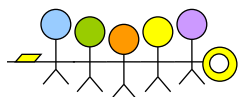
9.1 为什么需要 Harness

没有 Harness 的 Agent 只是一个演示原型: 原生模型会话间没有记忆、会犯下自信的错误、无边界的工具访问危险、规模会放大错误。Harness Engineering 系统化地解决这些问题, 核心挑战是“让 Agent 在多个上下文窗口中取得一致进展”, 解决方案在于结构化的环境、进度追踪构件与清晰的会话间状态管理。

9.2 Harness 的四大支柱

若将 LLM 比作 CPU, Harness 就是操作系统, 由四大支柱构成:

支柱	职责	手段
上下文治理	智能裁剪与恢复, 而非塞满上下文	压缩、卸载、渐进式披露
工具编排	验证参数、沙箱执行、防幻觉调用	权限边界、沙箱、确定性校验
状态持久化	跨会话记忆、崩溃后可恢复	结构化状态、进度追踪、检查点
渐进式披露	按需加载技能与权限	最小权限原则、动态发现



9.3 五大核心概念（五支柱详述）

- **工具编排（Tool Orchestration）**：定义可用工具、调用方式与权限，明确 Agent 能做什么、不能做什么、什么需人工预批准。
- **护栏与安全约束（Guardrails）**：确定性规则在多层运行——权限边界、验证检查（linters/类型检查/测试）、架构约束、速率限制。关键洞察：**更多约束通常带来更高可靠性**。
- **错误恢复与反馈循环**：自动化重试（带升级策略）、自我验证循环、回滚机制、循环检测。增加自我验证与循环检测曾将编码 Agent 得分从 52.8% 提升至 66.5%。
- **可观测性（Observability）**：记录每个动作、追踪 Token 与成本、记录决策点、发现异常，区分研究原型与生产系统。
- **人工在环检查点（HITL）**：在错误代价最高的高杠杆决策点引入人工审批，而非微观管理。

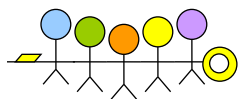
9.4 行业实践案例

产品	Harness 实现
OpenAI Codex	AGENTS.md 机器可读指令 + 可重复隔离环境 + CI 机械不变性，百万行代码无手写
Claude Code	默认只读 + 权限批准 + Hooks 生命周期注入 + CLAUDE.md 持久上下文
Cursor	.cursor/rules 基于 Markdown 的配置，版本受控、按文件模式、始终开启

这些案例共同说明：****约束即能力****——良好的 Harness 不仅防止出错，还通过“在正确时间提供正确上下文、工具与约束”让 Agent 更强。

9.5 构建你的第一个 Agent Harness（六步法）

16. **定义范围**：明确 Agent 能做什么与绝不能做什么，形成权限清单。
17. **创建配置文件**：AGENTS.md / CLAUDE.md / .cursorrules，记录规范、目录、测试命令与架构约束。
18. **设置反馈循环**：至少实现“写代码—跑测试—修复”的自验证循环。
19. **添加护栏**：限制目录访问、提交前 lint、封禁破坏性命令，从严格起步逐步放开。
20. **部署可观测性**：记录动作、工具调用、Token 使用，用数据诊断改进。
21. **设计人工检查点**：明确哪些动作需人工批准，把人类判断放在错误代价最高的点。



9.6 总结：三层工程的演进

Prompt Engineering → Context Engineering → Harness Engineering 的演进，对应 AI 应用从“单次对话”到“可靠生产系统”的成熟路径。真正的工程价值，往往来自对这些层级的正确理解与组合使用——**选对模型、管好上下文、套好 Harness**，方能构建出可落地、可扩展、可信赖的企业级 Agent 系统。