大模型原理与应用

第 4 课：Agent 系统 | 工具、记忆、工作流与评测

从提示词走向工具调用与真实执行

看什么

- Prompt、ICL、reasoning 为什么是 agent 底座
- tool、memory、workflow、evaluation 怎样拼成系统
- MCP、模型接入层、agent 框架、产品层分别在做什么
- 为什么我们后面会重点用 PydanticAI 做工具调用练习

Agent 不是单一模型，而是由 prompt、reasoning、tool、memory、workflow 组成的复合系统。

先给这章一个总判断

- agent 不是“更会聊天的模型”
- agent 是“能在环境里反复观察、决策、执行、再更新”的系统
- 所以今天讲 agent，重点不只是模型本身
- 更是接口、工具、状态、流程和评测

什么是 Agent

- agent 会和环境反复交互
- agent 不只输出文本，还要做 action
- agent 要根据 observation 更新下一步决策

从历史上看，大致经历三层：

1. text agent：只在文本环境里行动
2. LLM agent：用大模型生成动作
3. reasoning agent：显式规划、调用工具、分步执行

今天主流的 agent，基本都属于第三层。

一个 agent 系统的标准回路

无论框架名叫什么，核心循环都差不多：

1. 接收用户目标
2. 读取上下文与历史
3. 规划下一步
4. 调用工具或环境动作
5. 观察返回结果
6. 更新状态并继续
7. 给出最终答案或产物

先回到第一层：Prompt 为什么仍然重要

- prompt engineering 是“如何和模型沟通”
- 模板、few-shot、instruction 都是 prompt 的不同形式
- Chain-of-Thought 让模型把隐式推理外显出来

它的工程含义是：

- prompt 不是装饰，而是程序接口
- 许多 agent 框架本质上都在管理 prompt 结构

再往前一步：ICL 为什么是最轻量的适配方式

- GPT-2 已经展示零样本/少样本泛化
- GPT-3 让 few-shot prompting 成为主流工作范式
- 任务说明、示例和目标可以直接塞进上下文

这给开源工具链带来两个关键启发：

- 许多任务可以先不微调，直接靠上下文适配
- agent 的 planner、tool description、memory snippets，本质上都在做 ICL

推理为什么是 agent 的能力底座

- reasoning 是让模型从“像答案”走向“能解题”的关键
- CoT、Zero-shot CoT、Self-consistency 是最经典的推理增强手段
- 大模型很多能力具有 scale-driven emergence

也就是说：

- agent 之所以有用，不只是因为能调用工具
- 更因为它能在工具之间组织多步推理

Test-Time Compute：让模型“想久一点”

- 不是只靠更大参数
- 也可以靠更多 test-time compute 提高效果

主要方法包括：

- best-of-n
- beam / search
- verifier / PRM / ORM
- branching、backtracking、reflection

这和 agent 框架直接相关，因为 agent workflow 本质上就是一种 inference-time scaling。

弱监督与强模型对齐

- 弱监督者也可能有效控制强模型
- 不同任务上的 Performance Gap Recovered 差异很大
- NLP 与 chess 类任务更容易恢复
- reward modeling 类任务更难

这对工具链有现实意义：

- agent 训练不一定总依赖最强人工标注
- 也可以借助弱模型、规则系统、自动反馈构建 supervision pipeline

进入第二层：为什么 agent 必须接工具

- 纯语言模型缺少最新知识
- 容易幻觉
- 数学与精确计算薄弱
- 不知道时间流逝

所以需要把模型和外部工具相连：

- 搜索
- 计算器
- API
- 数据库
- 浏览器
- 代码执行环境

所以今天看 agent，重点已经不只是“prompt 写得好不好”。

更关键的是：

Toolformer 到现代 Tool Use

- 让模型自己学习何时调用工具
- 目标不是手工写死流程
- 而是让 API 调用也成为语言建模的一部分

材料里还点到了：

- ART
- AgentBench
- ToolLLM
- ToolkenGPT
- CogAgent

这说明 tool use 已经从“补丁”变成独立研究方向。

MCP：为什么它现在这么重要

- 过去每个 agent 框架都有自己的一套 tool 接口
- 现在越来越多系统转向 MCP
- 因为大家都希望：工具接一次，多个 agent 都能用

可以把 MCP 理解成：

- agent 世界里的“统一工具插座”

它解决的是：

- 工具怎么发现
- 参数怎么描述
- 权限怎么控制
- 不同客户端怎么复用

先把工程栈理一理

如果把 agent 系统拆开来看，至少有几层：

1. base model layer：基础模型与推理模型
2. model access layer：统一模型接入与路由
3. tool / protocol layer：tool schema、MCP、prompt 约定
4. agent framework layer：单 agent 或多 agent 框架
5. product / runtime layer：IDE、终端、云端执行环境
6. observability / eval layer：追踪、评测、调试

后面这些工具名，其实就是分布在这几层。

VS Code：为什么它正在变成 Agent 运行环境

- 过去 VS Code 更像“编辑器 + 插件市场”
- 现在它越来越像“多 agent 的控制台”

最近几轮更新之后，VS Code 已经支持：

- 本地 agent
- 云端 agent
- agent session 管理
- MCP server 接入
- skills / hooks / browser tools

所以今天很多开发者不是“离开编辑器去用 agent”， 而是直接把 VS Code 当成 agent 工作台。

GitHub Copilot：为什么它最容易先上手

- 它直接内嵌在 VS Code 里
- 上手门槛低
- 对学生最友好

你可以把它分成两层来看：

1. 编辑器内的对话、补全、修改
2. 背景里的 coding agent

对初学者最重要的一点是：

- 你不用先搭一整套 agent 框架
- 就能先体验“提出任务 -> 读代码 -> 改文件 -> 继续迭代”

GitHub Copilot 的新变化

最近一年的重点变化，不只是“更会补全”：

- agent mode 可以在编辑器里多步执行任务
- coding agent 可以在 GitHub 云端后台工作
- VS Code 里可以直接把本地对话委托给后台 agent
- 还能结合 MCP 与更多外部工具

这意味着：

- Copilot 不再只是“代码补全”
- 而是在向真正的 agent 工作流演化

Claude Code：终端里的强执行型 agent

- Claude Code 的特点不是“像聊天”
- 而是“能直接动手”

它擅长做的事包括：

- 读仓库
- 改文件
- 跑命令
- 跑测试
- 提交修改

对学生来说，更容易理解成：

- GitHub Copilot 很像“在编辑器里帮你写”
- Claude Code 更像“在终端里帮你做”

Claude Cowork：把 agent 从“写代码”扩展到“做工作”

- Cowork 的定位比 Claude Code 更宽
- 它不是只服务程序员
- 而是想让 Claude 去完成更一般的多步工作流

官方现在强调的方向包括：

- research synthesis
- document creation
- data extraction
- multi-step workflow execution

所以可以把它理解成：

- Claude Code：偏开发任务
- Cowork：偏通用办公与知识工作任务

Codex：为什么它值得单独讲

- Codex 现在已经不是早年的代码模型名字
- 而是 OpenAI 的 coding agent 产品线

它强调的是：

- 能独立完成较完整的工程任务
- 适合 routine PR、refactor、migration 这类工作
- 支持多 agent / 多任务并行

如果用一句话概括：

- 它不是“帮你补一句代码”
- 而是“替你接下一整段工程工作”

OpenCode：开源 coding agent 的一个代表

- OpenCode 是开源的
- 重点在终端与 IDE 里的 coding agent 体验
- 强调 agent、subagent、permissions、skills

它很值得课堂里讲的原因是：

- 很容易看清一个 agent 系统的真实组成
- 包括主 agent、子 agent、工具权限、任务分发

所以它特别适合拿来解释：

- 一个现代 coding agent 到底是怎么被“编排”出来的

OpenClaw：为什么它也值得提

- OpenClaw 更强调开放编排和多 agent 协作
- 它不是单纯一个“聊天窗口”
- 而是朝着“可扩展 agent 平台”方向走

你可以把它理解成：

- 更强调 orchestrate
- 更强调 sub-agent
- 更强调把不同任务挂到不同执行单元

这类系统的价值在于：

- 帮我们看清 agent 未来不一定只是“单个助手”
- 也可能是一个会分工、会调度、会并发的系统

第三层：先讲最适合上手的框架，为什么是 PydanticAI

- 因为它特别适合教学
- 代码短
- 结构清楚
- 和 Python 生态贴得近

它最核心的价值是：

- 把 agent 写成一种“类型清晰、工具清晰、输出清晰”的 Python 程序

官方自己的说法是：

- 要把 FastAPI 那种感觉带到 GenAI app 和 agent 开发里

PydanticAI 最值得初学者抓住的点

1. Agent 是一个 Python 对象
2. tool 就是 Python 函数
3. 参数 schema 自动生成
4. 输出可以做结构化校验
5. 出错后可以重试和反思

这几件事合在一起，特别适合课堂练习。

因为学生能很快看懂：

- 模型什么时候在说话
- 什么时候在调工具
- 工具参数是怎么传的
- 输出为什么能更稳

PydanticAI 和一般“函数调用”框架的区别

- 很多框架也能做 tool calling
- 但 PydanticAI 的优势是“强约束”

具体体现在：

- 用 Pydantic 校验输入参数
- 用 Pydantic 校验结构化输出
- tool 的 docstring 也能直接变成工具描述
- RunContext 让依赖和上下文更清楚

这意味着：

- 不是只让模型“试着调工具”
- 而是让工具调用更像真正的软件接口

为什么后面的 2-prog 要用 PydanticAI

因为它很适合做一个最小、清楚、可解释的 agent 例子。

我们后面的练习可以很自然地展示：

- 如何注册 tool
- 如何写 system instructions
- 如何让模型自己决定要不要调工具
- 如何把结果组织成结构化输出

这比一上来就上大型框架更适合编程基础较弱的同学。

再看更通用的一组框架

PydanticAI 适合讲“最小闭环”。

但真实工程里， 还会经常遇到另一组问题：

- 怎样更快搭一个 agent
- 怎样做状态化工作流
- 怎样做多 agent 协作

所以还要看：

- LangChain
- LangGraph
- Deep Agents
- CrewAI

LangChain：为什么它仍然是默认入口之一

- LangChain 的优势是上手快
- 模型接入多
- 工具生态广
- 很适合先把一个 agent 跑起来

官方现在的定位也很明确：

- 如果你想快速搭一个 agent
- LangChain 往往是最方便的起点

所以它更像：

- agent 开发里的“通用脚手架”

LangGraph：为什么很多生产系统会走到它

- LangGraph 比 LangChain 更底层
- 更强调状态、节点、边和长生命周期执行
- 特别适合“可控工作流 + agent 行为”混合的系统

更容易懂的说法是：

- LangChain 适合快速搭一个 agent
- LangGraph 适合把 agent 真正做成一个可控系统

它特别适合：

- durable execution
- human-in-the-loop
- stateful workflow
- long-running agent

Deep Agents：LangChain 现在推荐的“更完整 agent”

- LangChain 官方现在会明确区分：
- LangChain
- LangGraph
- Deep Agents

Deep Agents 可以理解成：

- 建在 LangChain + LangGraph 之上的“batteries-included” agent harness

它自带的东西更完整：

- planning
- virtual filesystem
- subagents
- long-term memory

所以如果任务更复杂，它比“裸 LangChain agent”更接近真实 agent 系统

CrewAI：为什么它在多 Agent 这条线上很流行

- CrewAI 的核心表达不是一个 agent
- 而是一组 agent、task、crew、flow

它很强调：

- 多角色协作
- 流程编排
- guardrails
- memory
- human-in-the-loop

所以它适合讲的不是“最小 agent”，而是：

- 多 agent 怎么分工
- 任务怎样串起来
- 一个 business workflow 怎样自动化

LangChain / LangGraph / Deep Agents / CrewAI 怎么区分

可以先用一句最实用的话记住：

- LangChain：先把 agent 搭起来
- LangGraph：把 agent 变成可控工作流
- Deep Agents：把复杂 agent 的常用能力预先装好
- CrewAI：把多 agent 协作和业务流程做成主线

所以它们不是完全互斥，而是站位不同。

模型接入层：LiteLLM 为什么非常有用

- 真正做 agent 系统时，一个麻烦问题是：
- 模型提供商太多
- 接口不完全一样
- 成本、限流、日志、fallback 也都要管

LiteLLM 的价值就在这里：

- 用统一接口接很多模型
- 可以做 proxy / gateway
- 可以做路由、限流、日志、成本追踪、fallback

所以它更像：

- agent 系统里的“模型接入层”或“模型网关”

观测与评测层：Langfuse 为什么迟早会需要

- agent 一旦变长链路，就很难靠肉眼 debug
- 你需要知道：
- 它什么时候调用了哪个工具
- 哪一步开始跑偏
- 哪个 prompt 版本效果更好

Langfuse 这类平台的价值就在于：

- tracing
- observability
- metrics
- evals
- prompt management

也就是说：

到这里，可以把整套工程栈放回一张图

如果把它们放到同一张图里：

- MCP：统一工具接口
- LiteLLM：统一模型入口
- LangChain：快速搭 agent
- LangGraph：编排可控工作流
- Deep Agents：复杂 agent 的完整 harness
- CrewAI：多 agent 协作和 flows
- PydanticAI：类型清楚的 Python agent
- Langfuse：观测、调试、评测

这样就不容易把“框架”“协议”“网关”“观测平台”混为一谈。

现在回到 agent 本身：memory 为什么不能省

- 语言 agent 不该只有短时上下文
- 还需要跨任务保留有用的 causal abstractions
- 无用经验应被遗忘，有用经验应被持久化

这个视角很关键：

- memory 不只是“存聊天记录”
- 而是把过去 trial 抽象成可复用知识

Agent Memory 的几种常见形态

1. short-term memory：当前对话和 scratchpad
2. episodic memory：过去任务的轨迹与反思
3. semantic memory：长期知识库、向量索引、文档库
4. procedural memory：工具说明、策略模板、工作流规则

很多失败的 agent，不是模型不够强，而是 memory 设计太弱。

context 和 memory 不要混

- context 更像当前这一次执行链里的信息传递
- memory 更像系统跨任务保留下来的经验

所以：

- prompt 是当前任务说明
- context 是任务交接
- memory 是历史经验

这几个词如果混了，后面设计 agent 就会越来越乱。

软件 Agent 的关键挑战

- environment 怎么定义
- observation / action 怎么设计
- file localization 怎么做
- planning 和 error recovery 怎么做
- safety 怎么保障

这也说明：

- agent 的难点往往不在模型本身
- 而在环境接口与任务闭环设计

最后再回到 evaluation：为什么不能只看问答分数

- agent evaluation 比普通 LLM eval 更复杂

需要区分：

- close-ended vs open-ended
- verifiable vs non-verifiable
- static vs dynamic
- 单步任务 vs 多步环境任务

Agent 的强弱，必须在 environment 中测。

WebArena / WorkArena 这类基准为什么重要

这类 benchmark 的意义在于：

- 不只看最终答案
- 也看路径是否正确
- 看是否能在真实网页或企业工作流里完成操作

它们评测的能力包括：

- planning
- tool use
- browser interaction
- long-horizon decision making
- robustness

这比普通 benchmark 更接近真实部署。

一条最小可行的 Agent 工程闭环

如果要自己做一个开源 agent，最小闭环通常是：

1. 选一个基础模型
2. 设计 system prompt 与任务模板
3. 接一个或多个工具
4. 给出 memory / state 表示
5. 规定 workflow 与终止条件
6. 设计自动评测
7. 根据失败案例做迭代

少任何一环，系统通常都很脆弱。

为什么“框架”不能替代“设计”

这些材料共同传达了一个现实判断：

- 框架能节省样板代码
- 但不能替代任务建模

真正决定效果的，往往是：

- 环境抽象得是否正确
- 工具接口是否稳定
- prompt 是否清晰
- 记忆是否可控
- 评测是否对准真实目标

所以要警惕“换框架就能变强”的错觉。

最后给一个实际的选型建议

如果目标不同，框架也不同：

- 想最快上手：优先看 GitHub Copilot + VS Code
- 想快速搭一个通用 agent：优先看 LangChain
- 想做强状态控制与长流程：优先看 LangGraph
- 想做更完整的复杂 agent：可以看 Deep Agents
- 想做多 agent 协作与业务流程：可以看 CrewAI
- 想做强执行型 coding agent：优先看 Claude Code / Codex / OpenCode
- 想做开放编排与多 agent：可以看 OpenClaw
- 想写一个结构清楚、适合教学的 tool-using agent：优先看 PydanticAI
- 想把很多工具真正接进来：优先理解 MCP
- 想统一接很多模型：优先看 LiteLLM
- 想把 agent 调试清楚、做评测和 prompt 管理：优先看 Langfuse

框架选型应服从任务，不应反过来。

这部分你该带走什么

- agent 是复合系统，不是单一模型
- prompt、ICL、reasoning 是能力起点
- tool use、memory、workflow、evaluation 决定它能不能真的做事
- MCP 正在变成工具接入标准层
- LiteLLM 和 Langfuse 代表了“模型网关”和“可观测性”这两块基础设施
- LangChain、LangGraph、Deep Agents、CrewAI、PydanticAI 分别代表不同层次的 agent 工程路线
- VS Code 正在变成多 agent 运行环境
- PydanticAI 很适合拿来做第一批 tool-using agent 练习

下一步可以怎么做

1. 在 VS Code 里体验 GitHub Copilot / agent mode
2. 用 Claude Code 或 Codex 感受强执行型 coding agent
3. 用 PydanticAI 写一个最小工具调用 agent
4. 比较无工具、有工具、有结构化输出三种配置

第 6 章收束

下一步：把 Agent 真正跑起来，而不只是停留在概念上