Anthropic：Agents 的设计之道与年度总结

特工少女 特工宇宙 2024年12月23日 08:18 浙江

在过去的一年里，Anthropic 在构建 LLM 和 agents 这件事情上，与多个行业的数十个团队有过合作。

他们发现，最成功的方案并不是使用复杂的框架或专门的软件包。相反，**它们都是使用简单、可组合的模块来构建的。**

Anthropic：研发 Claude 的主打安全可靠的 AI 公司，核心团队均曾任职于 OpenAI。

这篇文章将分享 Anthropic 从与客户共建 agents 的过程中学到的经验，并为开发者们提供如何构建有价值的 agents 的相关建议。

什么是 agents?

"Agent" 可以有多种不同的定义方式。一些客户将 agents 定义为一个完全自主的系统，它们能够长期独立运行，使用各种工具来完成复杂的任务；另一些客户则把 agents 当作更符合规范性的并遵循预定义的工作流。

在 Anthropic，他们将所有这些不同的形式都归类为 **agentic systems（智能系统）**，但在 **workflows（工作流）**和 **agents（智能体）**之间做出了一个重要的架构区分：

⚡ Workflows 是通过预先定义好的代码路径来编排大模型和工具的系统。

⚡ Agents 是由大模型动态规划自身处理流程和工具使用，并能够自主控制如何完成任务的系统。

接下来将详细探讨这两种 agentic systems。在附录 1（"实践中的 agents"）中将介绍在某些领域中，客户发现使用这类 agentic systems 特别有价值。

何时（以及何时不）使用 agents

当使用 LLMs 构建应用时，建议寻找尽可能简单的解决方案，并且仅在必要时再增加任务复杂性。这可能意味着根本不构建 agentic systems. agentic systems 通常会以延迟和成本来换取更好的性能，需要考虑这种权衡。

何时以及如何使用框架

有许多框架可以使 agentic systems 更容易实现，比如：

⚡ LangChain 的 LangGraph；

⚡ Amazon Bedrock 的框架；

链接：https://aws.amazon.com/cn/bedrock/agents/

⚡ Rivet，一个通过拖拉拽的 GUI LLM 的工作流构建工具；

⚡ Vellum，一个用于构建和测试复杂工作流的 GUI 工具。

这些框架通过简化诸如调用大模型、快速编写和解析相关工具插件、链式调用等标准化的底层任务，使得开发者能够轻松上手。

但是，它们往往会创建额外的“抽象层”，一定程度上遮蔽了底层的 Prompt 和响应内容，从而增加调试的难度。同时，**这些框架还容易让开发者在一些本可以用更简单方案解决的场景中，不自觉地引入不必要的过度设计。**

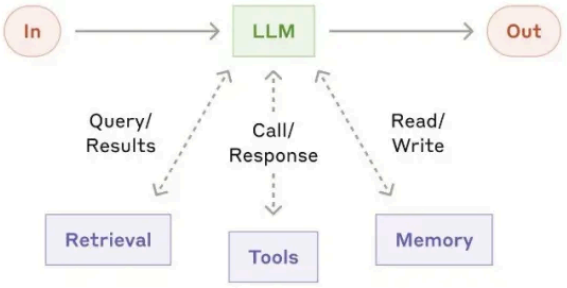
我们建议开发人员从直接使用 LLM API 开始：只需几行代码即可实现许多常用的模式。**如果确实想要使用框架，要确保了解底层的代码。**对底层内容的错误的假设和理解，是客户出错的常见来源。

Anthropic 官方手册：https://github.com/anthropics/anthropic-cookbook/tree/main/patterns/agents

构建 blocks、workflows 和 agents

本节将探讨在生产中看到的 agentic systems 的常见模式。我们会从基础构建模块——增强型大模型（augmented LLM）开始，逐步提升复杂度，从简单的组合工作流到自主的 agents 系统。

agentic ststems 的基本构建模块是经过增强的大语言模型，这种增强包括检索能力、工具调用和记忆功能。如今的模型能够主动运用这些能力——自主生成搜索查询、选择合适的工具，并决定保留哪些信息。



The augmented LLM

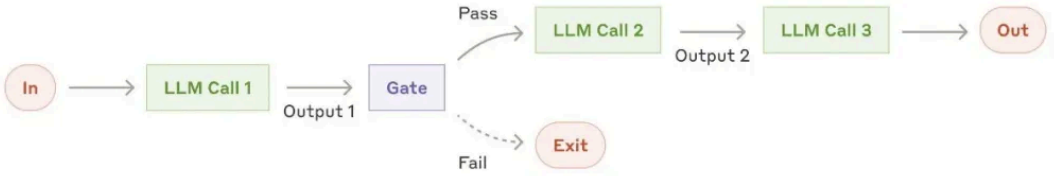
在实现过程中，建议重点关注两个方面：根据具体使用场景定制这些能力，并确保为大模型提供简单且文档完善的接口。虽然实现这些增强功能的方法有很多，但其中一种方式是使用 Anthropic 最近发布的模型上下文协议（MCP，Model Context Protocol），通过简单的 client implementation，开发者可以借助该协议接入各种第三方工具生态。

MCP：https://www.anthropic.com/news/model-context-protocol

对于下文内容，将假设每个 LLM 都可以使用这些增强功能。

Workflow: Prompt chaining（提示链）

提示链将一个任务拆解为一系列步骤，每个 LLM 都会处理前一个步骤的输出。你可以在任何中间步骤中添加一个“程序性检查”（见下图中的"gate"），以确保流程按预期进行。



The prompt chaining workflow

适用场景：非常适合任务可以轻松、明晰地拆解为固定子任务的情形。主要目的是通过使每个 LLM 的调用变得更容易，从而在回复速度和更高的准确性之间进行一些权衡。

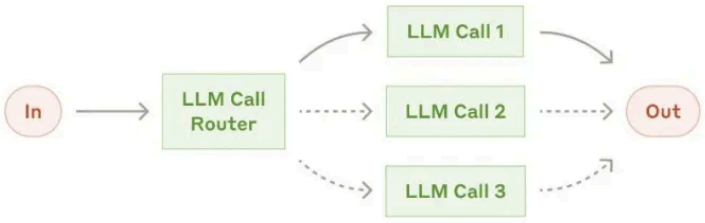
适用示例：

⚡ 生成市场营销文案，然后将其翻译成不同语言。

⚡ 撰写文档大纲，检查大纲是否符合某些标准，然后根据大纲撰写文档。

Workflow: Routing（路由）

Routing 会对输入进行分类并引导至后续相应的专门任务。此工作流允许分离你关注的部分，并对其构建更专业的 Prompt。如果没有此工作流，针对一种输入的优化可能会损害其他输入的性能。



The routing workflow

适用场景：Routing 在处理复杂任务时表现良好，这些任务具有明确的类别，适合分别处理，并且分类可以通过 LLM 或更传统的分类模型/算法进行准确处理。

适用示例：

⚡ 将不同类型的客户的问题（一般问题、退款请求、技术支持等）路由到不同的下游流程、Prompt 和工具中。

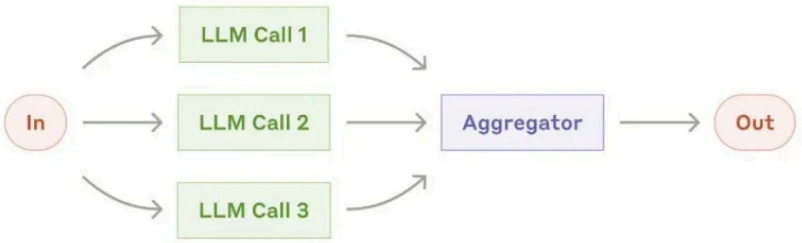
⚡ 将简单/常见问题的路由到较小模型（如 Claude 3.5 Haiku），将困难的/不寻常的问题路由到更强大的模型（如 Claude 3.5 Sonnet），以优化成本和速度。

Workflow: Parallelization（并行）

LLMs 有时可以分别同时完成一项任务，并汇总其输出。此类工作流体现在两个关键：

Sectioning（任务拆解）：将任务分解为并行运行的独立子任务。

Voting（投票）：多次运行同一任务以获得不同的输出。



The parallelization workflow

适用场景：当分割的子任务可以并行化以加速处理，或者当需要从多个角度进行尝试以获得更可靠的结果时，Parallelization 是有效的。对于具有多重考虑的复杂任务，通常把每个要考虑的因素都通过单独的 LLM 调用进行处理时，LLM 的表现更佳。

适用示例：

Sectioning（任务拆解）：

⚡ 安全防护，其中一个模型处理用户的查询，而另一个模型用于筛查不当内容或请求。这通常比让同一个 LLM 同时处理安全防护和核心响应要有效得多。

⚡ 自动化评估，用于评估 LLM 在特定 Prompt 下的表现，每个 LLM 用于评估模型表现的不同方面。

Voting（投票）：

⚡ 检查代码中的漏洞，多个不同的 Prompt 检查代码，如果发现问题则标记。

⚡ 评估某一内容是否不当，多个 Prompt 用于评估不同的方面，或者设置不同的投票阈值，以平衡测试的准确性。

Workflow: Orchestrator-workers（协调者-工作者）

在此工作流中，中央 LLM 动态地分解任务，将其分配给 worker LLMs，并综合考虑它们的结果。

The orchestrator-workers workflow

适用场景：适合无法预测所需子任务的复杂任务（例如，在写代码的过程中，需要更改的文件数量和每个文件内部的更改，可能依赖于整体任务本身）。虽然它的流程图跟 Parallelization 很像，但关键区别在于其更灵活——子任务不是预先定义的，而是由 Orchestrator 根据特定输入确定的。

适用示例：

⚡ 每次需要对多个文件进行复杂更改的代码辅助向产品。

⚡ 涉及从多个来源收集和分析信息以寻找可能相关信息的搜索任务。

Workflow: Evaluator-optimizer（评估器-优化器）

在此工作流程中，一个 LLM 负责生成响应，而另一个 LLM 提供评估和反馈，从而形成循环。

The evaluator-optimizer workflow

适用场景：当我们有明确的评估标准，或迭代过程中的价值可以被衡量时，此工作流特别有效。比较适合的两个标志是，首先，当人类表达反馈时，LLM 的响应可以显著改善；其次，LLM 能够提供此类反馈。这类似于人类在撰写精炼的文档时，可能经历的迭代写作过程。

适用示例：

⚡ 文学翻译，其中翻译 LLM 可能最初无法捕捉到的细微差别，但负责评估的 LLM 可以提供有用的改善建议。

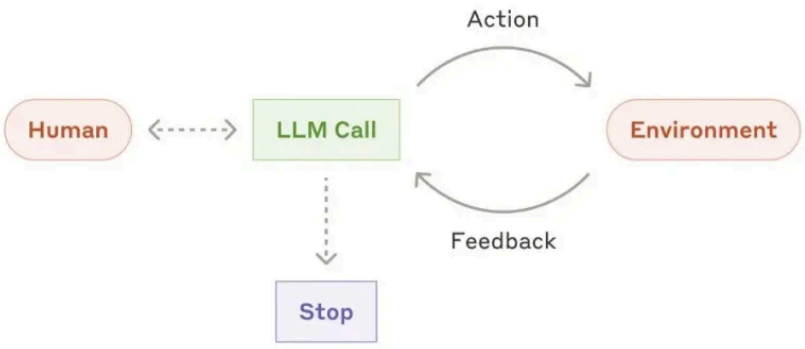
⚡ 需要多轮搜索和分析以收集全面信息的复杂搜索任务，负责评估的 LLM 决定是否需要进一步搜索。

Agents

随着 LLM 在理解复杂输入、进行推理和规划、使用工具以及从错误中纠错等关键能力的成熟，agents 在生产中逐渐兴起。

agents 的工作始于来自用户的命令或互动讨论。一旦任务明确，agents 就可以独立规划和行动，其中可能会反问人类，以获取进一步的信息或判断。在执行任务过程中，agents 在每一步（如工具调用或代码执行）获得环境的 "ground truth"（“真实情况”）来评估任务进展至关重要。因此，agents 可以在遇到障碍时暂停，以获取人类的反馈。任务通常在完成时终止，但也常常包括停止条件（如最大迭代次数）以保持控制。

agents 可以处理复杂任务，但它们的实现往往相对简单。它们通常只是基于环境反馈循环使用工具的 LLM。因此，清晰而周全地设计工具集及其文档至关重要。附录 2（"Prompt Engineering your Tools"）中详细介绍了工具开发的最佳实践。



Autonomous agent

适用场景：agents 可以用于难以或不可能预测所需的步骤数量，并且无法规定好固定路径的开放式问题。LLM 可能会执行多个循环，因此您必须对其决策有一定的信任感。agents 的自主性使其在可信环境中执行任务时特别理想。

agents 的自主性也意味着更高的成本以及不断累积错误的潜在可能，建议在沙盒环境中进行广泛的测试，并设置适当的安全防护。

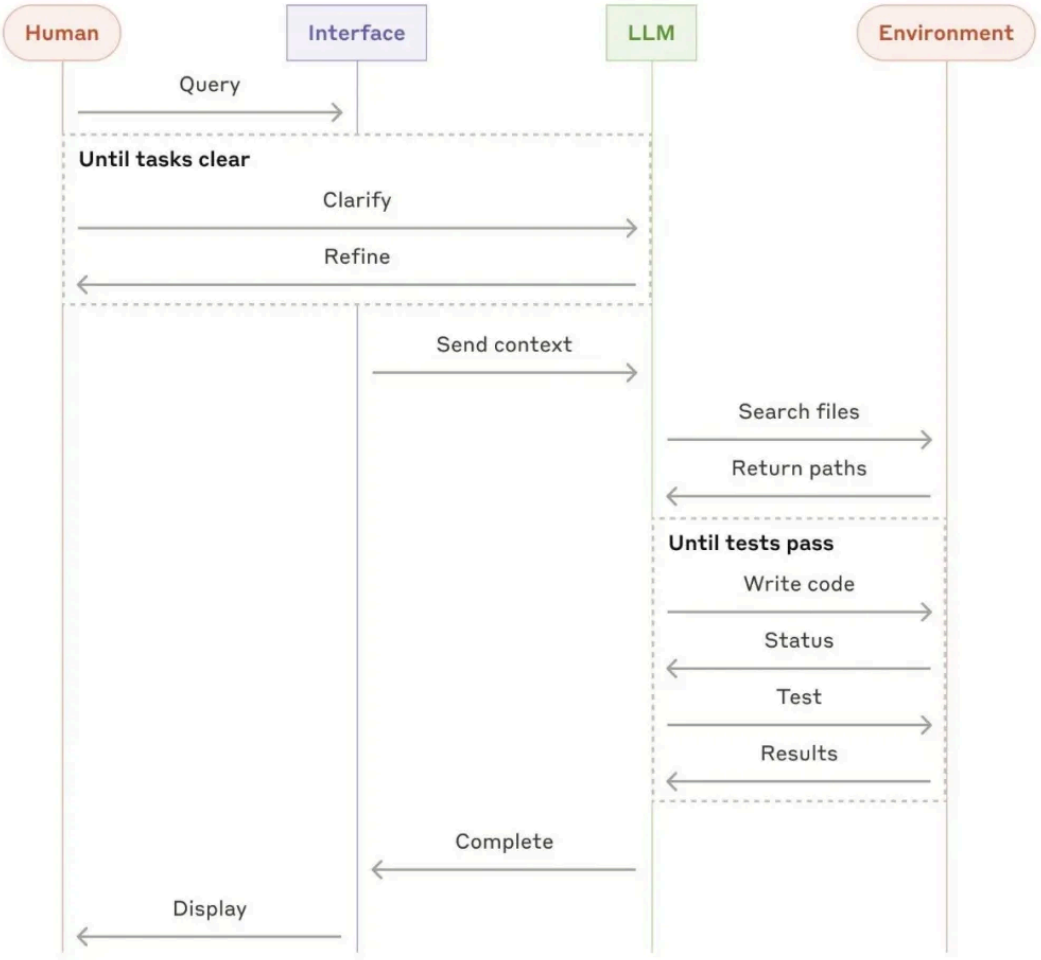
适用示例（以下示例来自 Anthropic 官方实现）：

⚡ 解决 SWE-bench（一个 AI 评估基准，用于评估模型完成软件工程的能力）的编程 agents，该项目涉及根据任务的描述对多个文件进行编辑；

SWE-bench：https://www.anthropic.com/research/swe-bench-sonnet

⚡ “computer use” 参考手册，，其中 Claude 使用一个计算机来完成任务。

“computer use” reference implementation：https://github.com/anthropics/anthropic-quickstarts/tree/main/computer-use-demo



High-level flow of a coding agent

组合和定制这些范式

这些构建范式并不是严格规定好的。它们是开发人员可以搭建和组合以适应不同场景的常见模式。与任何 LLM 功能一样，成功的关键，是衡量性能并迭代落地。重申一下：**只有在明显能改善结果时，才应考虑增加复杂度。**

总结

在大模型领域落地成功并不在于构建最复杂的系统，**而在于为需求构建合适的系统。**从简单的 Prompt 开始，通过全面评估进行优化，仅在简单解决方案不足时添加多步骤的 agentic systems。

在构建 agents 时，我们努力遵循三个核心原则：

⚡ 在 agents 设计中保持简单。

⚡ 通过明确展示 agents 的规划步骤来优先考虑透明度。

⚡ 通过全面的工具文档和测试，精心设计 agent-computer interface（ACI）接口。

框架可以帮助您快速入手，但在进入生产阶段时，请毫不犹豫地减少“抽象层”的使用，而尽量使用基本组件进行构建。通过遵循这些原则，就可以创建不仅强大而且可靠、可维护且受用户信任的 agents.

附录一：实践中的 agents

与客户的合作过程中揭示了两个特别有前景的 AI agents 应用，展示了上述模式的实际价值。这两个应用都表明，agents 在需要对话和行动的任务中提升了重大价值，具有明确的成功标准，能够实现反馈循环，并整合了有价值的人类监督。

A. 客户支持（Customer support）

客户支持通过工具集成将熟悉的聊天机器人界面与增强的功能结合起来。这对于开放式 agents 来说是自然的场景，因为：

1. 遵循对话流程，交互自然，同时需要访问外部的信息和执行动作；

2. 可以集成工具以提取客户数据、订单历史和知识库文章；

3. 如退款或更新工单等操作可以用增加代码节点的方式处理；

4. 可以通过用户预先定义的理想解决效果，明确衡量是否 agents 解决了该问题。

有几家公司通过基于使用量的定价模型证明了这种方法的可行性，该模型仅对成功的解决方案收费，这展示了对其 agents 有效性的信心。

B. 编程辅助（Coding agents）

软件开发领域显示出 LLM 的显著潜力，功能从代码补全演变为自主解决问题。agents 特别有效的原因是：

1. 代码问题的解决可以通过自动化测试进行验证；

2. 代理可以使用测试结果作为反馈，对流程方案进行迭代；

3. 问题定义明确且结构化；

4. 输出质量可以客观测量。

在我们自己的实践过程中，agents 现在能够仅基于 pull request description（拉取请求描述）就能解决实际的 GitHub 问题。然而，尽管自动化测试有助于验证功能，人类审查在确保流程方案符合更广泛的系统要求方面，仍然至关重要。

附录二：插件工具的提示词工程（读懂此部分需要一定编程基础）

无论构建哪种 agentic systems，工具插件都可能是 agents 的重要组成部分。工具使 Claude 能够通过在我们的 API 中规定其确切的定义，从而与外部服务和 API 进行交互。

Claude can now use tools：https://www.anthropic.com/news/tool-use-ga

当 Claude 响应时，如果它计划调用工具，它将在 API 响应中包含工具的使用模块。工具的定义和规范应与整体的 Prompt 设计，获得同样的关注。在这一简短的附录中，描述了如何对工具进行 Prompt Engineering.

通常有几种方法可以指定相同的 actions. 例如，可以通过编写 diff（diff 是一个用于比较文件内容的工具和格式，通常用于显示两个文件之间的差异。它主要用于版本控制系统中，以便开发者可以看到代码的变化，diff 通常显示出哪些行被添加、删除或修改）或通过重写整个文件来实现指定文件的编辑。对于结构化输出，可以在 markdown 或 JSON 中返回代码。在软件工程中，这些差异只是表面的，可以无损地从一种格式转换为另一种格式。然而，某些格式对于 LLM 来说比其他格式更难编写。编写 diff 要求在新代码写入之前知道 chunk header 中有多少行正在更改。相比于 Markdown，在 JSON 中编写代码需要对换行符和引号进行额外的转义。

对决定工具格式的建议如下：

⚡ 给模型足够的 tokens 用于“思考”，以避免它走入死胡同。

⚡ 保持格式与在互联网上自然出现的文本格式相近。

⚡ 确保没有 formatting "overhead"，例如必须准确计数数千行代码，或者对其编写的任何代码进行字符串转义。

一个经验是在人机界面 (HCI) 上投入多少精力，就要投入同样多的精力来创建良好的 agent-computer 界面 (ACI)。以下是关于如何执行此操作的一些想法：

⚡ 设身处地为模型着想。根据描述和参数，如何使用这个工具是否显而易见？好的工具定义通常包括示例用法、边界情况、输入格式要求以及与其他工具的明确界限。

⚡ 如何更改参数名称或描述以使任务更加明显？可以将此视作为团队中的初级开发人员编写易读的说明文档那样。当使用许多类似的工具时，这一点尤其重要。

⚡ 测试模型如何使用您的工具，可以运行多个示例输入，以查看模型犯了哪些错误，然后进行迭代。

在为 SWE-bench 构建 agents 时，Anthropic 实际上花费了比整体 Prompt 设计更多的时间来优化的工具插件。例如，Anthropic 发现在 agents 移出根目录后，模型会在使用相对文件路径的工具时出错。为了解决这个问题，将工具更改为始终需要绝对文件路径，并且发现该模型完美地使用了这种方法。

声明：本文为「宇宙编辑部」翻译，未经许可禁止转载。

特工宇宙 Agent Universe

特工少女 内容运营

关注 AI+泛娱乐、AI+教育领域

欢迎交流，备注来意，加群请私信

Agent Universe，这里有关于Agent的一切。

- AGENT - UNIVERSE

扫码加入特工宇宙

社群矩阵👉

上一篇
深度丨AI 原生应用的定义、生态图谱，与五维评估框架

下一篇
AI Agents 24 年回顾 - 五大发展趋势