从指令到行动：函数调用与大语言模型智能体架构的崛起与全面解析

第一部分：超越文本生成的范式革命

本节旨在阐明函数调用（Function Calling）技术出现的根本原因，将其定位为克服大型语言模型（LLM）固有局限性所必需的架构演进，而不仅仅是一项功能性补充。它将用户查询中提供的“密室”比喻与学术和行业研究中的正式定义相结合，系统性地揭示这一技术变革的深远意义。

1.1 解构“密室”困境：独立大语言模型的内在局限性

大型语言模型，作为强大的文本生成引擎，其核心能力在于基于其庞大的训练数据集，以概率形式预测下一个最有可能出现的词元 [1]。然而，这种设计也使其陷入了一个概念上的“密室”——它们是功能强大但与现实世界隔离的“大脑” [用户查询]。它们的知识是静态的，被其训练数据的截止日期所束缚，无法主动获取实时信息或执行外部操作 [2]。这种固有的隔离性构筑了一座“信息孤岛”，导致了一系列关键的技术局限性 [用户查询]。

首先，LLM 在处理需要精确性的确定性任务时表现出固有的不可靠性。例如，对于数学计算、日期查询或精确的数据检索，模型的输出本质上是基于其语言模式的“猜测”，而非逻辑推导或计算。这使得它们在这些场景下的结果远不如专业的代码工具可靠 [4]。其次，由于知识的静态性，LLM 无法回答任何关于其训练数据截止日期之后发生的事件的问题，例如实时天气、股票价格或最新的新闻动态 [3]。当面对超出其知识范围的查询时，模型倾向于“幻觉”（Hallucination），即编造看似合理但实际上不准确的答案，这严重影响了其在关键应用中的可信度 [3]。最后，LLM 无法与用户的私有或专有数据进行交互，例如访问个人日历、查询内部数据库或操作企业级软件，这极大地限制了它们在个性化和企业自动化场景中的应用潜力 [4]。

这种从根本上将概率性文本生成与确定性现实世界操作分离开来的架构模式，是理解函数调用技术价值的关键。独立 LLM 的输出是基于其内部语言模型的概率分布，这对于创造性写作或文本摘要等任务是优势，但对于需要事实准确性和与外部世界互动的任务则成为致命缺陷。因此，需要一种新的架构范式，将 LLM 的概率性推理能力与外部世界的确定性执行能力相结合。函数调

用正是实现这种分离与协作的核心机制，它允许 LLM 专注于其最擅长的部分——理解人类语言的模糊意图，同时将需要精确执行的任务委托给专门的、可靠的外部代码。这种策略性的任务划分不仅弥补了 LLM 的短板，更催生了更加可靠、可信和功能强大的 AI 系统[3]。

1.2 作为连接外部世界之桥梁：函数调用的核心原则与术语

函数调用，也被称为“工具使用”（Tool Use）或“API 调用”（API Calling），是一种使大型语言模型能够与外部系统进行交互的核心能力[6]。其根本原则并非让 LLM 直接执行代码，这是一个普遍存在的误解。相反，LLM 的角色是生成一个结构化的、机器可读的指令，该指令详细描述了外部应用程序应该执行哪个函数以及传递什么参数[10]。这个过程将 LLM 的自然语言理解能力与外部代码的实际执行能力解耦，形成了一个强大而可靠的协作框架。

在行业实践和学术研究中，相关术语常被交替使用，但它们指向同一核心概念。函数调用（Function Calling）这一术语由 OpenAI 推广，特指其 API 中允许模型生成调用特定函数指令的功能。工具使用（Tool Use）是 Anthropic 等公司和研究界更广泛使用的术语，它涵盖了更广义的能力，其中函数调用是其最主要的形式[8]。

API 调用（API Calling）则更具体地强调了与外部网络服务接口的交互[14]。尽管存在细微差别，但这三个术语的本质都是一致的：LLM 识别用户意图，并生成一个结构化的输出来调用一个预定义的外部函数[8]。为了保持一致性，本报告将主要使用“函数调用”这一术语，同时承认其他术语的通用性。

这一机制的出现，标志着 AI 应用从单向的文本生成，转向了双向的、与现实世界互动的对话。它赋予了原本封闭的 LLM “手”和“脚”，使其能够突破自身知识的限制，获取实时数据、执行具体操作，从而在真实世界中产生实际影响。

1.3 大语言模型作为“智能任务调度员”：一个概念框架

通过函数调用，大型语言模型的角色发生了根本性的转变：从一个被动的文本生成器，演变为一个主动的、智能的“任务调度员”或“任务分发器”[12]。在这种新的框架下，LLM 的核心认知任务不再是简单地续写文本，而是深度理解用户通过自然语言表达的复杂意图，并将其精确地翻译成一系列结构化的函数调用指令[6]。

整个工作流程构成了一个“LLM 决策、外部代码执行”的闭环协作系统[16]。这个循环的起点是用户的自然语言请求。LLM 作为决策中心，分析该请求，并对照开发者预先提供给它的“工具清单”，判断需要调用哪个或哪些工具来完成任务。随后，它生成一个包含函数名和参数的结构化数据（通

常是 JSON 格式）。这个结构化数据被传递给外部的应用程序代码。应用程序代码负责解析这个指令，并真正地执行相应的函数——可能是一个本地计算，也可能是一次对外部 API 的网络请求。函数执行完毕后，其返回结果被再次发送给 LLM。LLM 在接收到这个新的信息后，可能会决定任务已经完成，并基于此结果生成最终的、人性化的自然语言回复给用户；或者，它也可能判断任务尚未完成，需要进行下一步操作，从而开启新一轮的“决策-执行”循环 [17]。

这个闭环、迭代的协作模式是所有高级 AI 智能体（Agent）行为的基础。它将 LLM 的语言理解和规划能力，与外部代码的精确执行和数据获取能力完美结合，使得 AI 系统能够处理远比单一文本生成任务复杂得多的、需要与现实世界进行多步交互的复杂任务。

第二部分：函数调用的运作机制：端到端流程剖析

本节将对函数调用的端到端工作流程进行一次精细、技术性的剖析。通过代码结构和 API 引用，我们将逐一拆解从工具定义到最终响应生成的每一个关键阶段，揭示其内部的运作逻辑。

2.1 定义工具箱：模式、描述与参数的关键作用

函数调用流程的起点是开发者向 LLM “注册”一个可用的工具清单 [用户查询]。这个过程并非在模型内部进行永久性修改，而是在每次 API 调用时，通过请求体动态地提供一组工具的结构化定义，通常是以 JSON Schema 的形式 [2]。这个定义就像是为 LLM 提供了一份详细的“工具使用说明书”，使其能够理解每个工具的功能和用法。

一个完整、有效的工具定义通常包含以下几个核心部分：

- 名称 **(name)**：一个唯一的、符合编程规范的函数名称，例如 getCurrentWeather。这个名称将由 LLM 在决定调用该工具时返回，并由应用程序代码用于映射到实际的函数实现 [16]。
- 描述 **(description)**：一段清晰、详尽的自然语言描述。这是整个机制中至关重要的部分，因为它直接影响 LLM 的决策过程。描述需要准确说明工具的功能、适用的场景、不适用的场景以及任何重要的限制。例如，“用于获取一个指定地点的实时天气信息” [16]。
- 参数 **(parameters 或 input_schema)**：一个 JSON Schema 对象，用于定义该函数接受的参数。这个对象详细说明了每个参数的名称、类型（如 string, integer）、描述以及是否为必需 (required)。这为 LLM 提供了提取和构建函数参数的精确蓝图 [2]。

为了提升开发效率和代码的健壮性，许多现代开发框架，如 Python 中的 Pydantic 库，允许开发者以更符合语言习惯和类型安全的方式来定义这些模式。这些框架通常可以自动将代码中的类或函数定义转换为符合 API 要求的 JSON Schema 格式，从而简化了工具定义的流程 [14]。

工具描述的质量直接决定了函数调用的可靠性。这不仅仅是为开发者提供文档，更是为 LLM 的推理引擎提供关键的上下文信息。一个模糊或不准确的描述会导致模型在工具选择上出现错误，或者无法正确提取参数。因此，精心撰写工具描述本身就是一种针对模型的“提示工程”（Prompt Engineering）。它要求开发者不仅要从人的角度思考，更要从模型的角度思考，如何用最清晰的语言引导模型做出正确的决策。这种“描述性提示”的技巧是构建高效、可靠的智能体的核心能力之一 [16]。

2.2 从意图到指令：大语言模型如何分析提示并选择工具

当接收到用户的查询后，LLM 会启动其强大的自然语言理解（NLU）引擎，将用户的非结构化文本与开发者提供的结构化工具清单进行匹配 [6]。这个过程可以分解为两个并行的认知任务：意图匹配和参数提取。

首先，模型会分析用户查询的核心意图。例如，对于“波士顿现在天气怎么样？”这个查询，模型会识别出用户的核心需求是“查询天气” [2]。接着，它会遍历所有可用工具的

description 字段，寻找与该意图最匹配的描述。如果某个工具的描述是“获取指定地点的当前天气”，模型就会建立起强关联，并选择该工具作为执行目标。这个匹配过程高度依赖于描述的清晰度和准确性。

在确定了要使用的工具后，模型会进入参数提取阶段。它会再次审视用户的查询，并根据所选工具的 parameters schema 来寻找对应的值。在上述例子中，schema 要求一个名为 location 的字符串参数。模型会从“波士顿现在天气怎么样？”中成功提取出“波士顿”作为 location 参数的值 [8]。如果 schema 中还定义了可选参数，比如

unit（单位），而用户查询中未提供，模型可能会使用 schema 中定义的默认值，或者在某些情况下，如果信息不足以做出决策，它甚至会生成一个追问来向用户澄清，例如“您想用摄氏度还是华氏度来显示温度？”。

整个过程——意图识别、工具匹配、参数提取——都是经典的 NLU 任务。这揭示了一个深刻的观点：函数调用的成功与否，在很大程度上取决于模型底层的语言理解能力，而非其生成代码的能力。一个在 NLU 任务上表现更优异的模型，通常也能更可靠地使用工具，从而构建出更强大的智能体 [2]。

2.3 结构化对话：生成并解析 JSON 函数调用

当 LLM 决定使用一个工具后，它并不会直接生成一句自然语言的回答。取而代之的是，API 的响

应会包含一个特殊的信号，表明其意图是调用一个或多个工具。这个信号在不同平台的 API 中有不同的体现，例如在 OpenAI 的 API 中，响应的 finish_reason 字段会是 tool_calls [23]；而在 Anthropic 的 API 中，

stop_reason 字段会是 tool_use [19]。

伴随这个信号，响应体中会包含一个名为 tool_calls（或类似名称）的结构化对象，通常是一个 JSON 数组 [2]。数组中的每一个元素都代表一个待执行的函数调用。每个元素都清晰地指明了以下信息：

- 工具名称 **(name)**：需要调用的函数的名称。
- 参数 **(arguments)**：一个包含所有提取出的参数及其值的 JSON 字符串。
- 调用 **ID (id)**：一个由模型生成的唯一标识符，用于在后续的交互中将函数执行的结果与这次特定的调用请求关联起来。

此时，控制权从 LLM 转移到了开发者的应用程序代码。应用程序的责任是接收这个 API 响应，首先检查 finish_reason 或 stop_reason 来判断是否需要进行工具调用。如果需要，它就必须解析 tool_calls 数组，遍历其中的每一个调用请求，提取出函数名和参数 JSON 字符串，并将其解析为程序可以使用的原生数据结构（例如 Python 中的字典）[10]。这个解析步骤是连接 LLM 的“指令”和应用程序“行动”的关键环节。

2.4 行动阶段：应用程序代码在执行外部函数中的角色

一个至关重要且常常被误解的核心点是：大型语言模型本身不执行任何代码 [10]。所有的实际操作，无论是本地计算还是与外部世界的交互，都完全由开发者的应用程序代码在自己的运行环境中完成 [10]。LLM 的角色仅限于“提议”或“指令”一个动作，而不是执行它。

应用程序在解析出函数名和参数后，需要一个调度（dispatcher）或映射（mapping）逻辑。这个逻辑的核心作用是将 LLM 返回的字符串形式的工具名称（如 "getCurrentWeather"）与代码库中实际的、可执行的函数（如 def get_current_weather(location, unit):...）关联起来 [10]。一旦找到对应的函数，应用程序就会用解析出的参数来调用它。

这个执行阶段可能涉及多种操作：

- 本地计算：执行一个数学运算、日期格式化或数据转换函数。
- 数据库查询：连接到数据库，执行一条由 LLM 生成的 SQL 查询语句。
- 外部 **API** 调用：向一个第三方服务（如天气预报、股票行情、航班信息 API）发起网络请求。
- 文件系统操作：读取或写入本地文件。
- 硬件控制：在物联网（IoT）场景中，向智能家居设备发送控制命令。

这种架构设计确保了安全性和可控性。因为所有的执行都在开发者的环境中进行，所以开发者可

以实施完整的安全检查、权限控制和错误处理逻辑，防止 LLM 生成的指令导致意外或恶意的行为。

2.5 闭合循环：反馈结果以合成最终响应

当外部函数执行完毕后，其返回值（例如，从天气 API 获取的 {"temperature": 15, "condition": "Cloudy"}）必须被送回给 LLM，以完成整个信息交互的闭环 [2]。

这个反馈过程是通过向 LLM 发起一次新的 API 调用来实现的。在这次调用中，开发者需要在对话历史（messages）中追加一条新的消息。这条消息具有一个特殊的角色（role），通常被定义为 tool（或因 API 而异）。这条消息的内容结构非常关键，它必须包含：

- 函数执行的结果：通常是字符串格式的函数返回值。
- 工具调用 **ID (tool_call_id)**：这个 ID 必须与之前 LLM 在 tool_calls 响应中提供的 ID 完全一致。这个 ID 的作用就像一个回执单号，它告诉 LLM 这个结果是针对哪一次具体的函数调用请求的 [19]。

当 LLM 接收到这条包含工具执行结果的新消息后，它就获得了完成用户原始请求所需的全部上下文。此时，它会利用其强大的语言生成能力，将这个结构化的数据（工具结果）与原始的用户问题结合起来，综合成一句通顺、自然、人性化的最终答复，例如："伦敦当前的天气是 15 摄氏度，多云。" [8]。

这个完整的"请求-决策-执行-反馈-合成"的流程揭示了函数调用工作流的内在特性：它本质上是异步且有状态的。整个过程至少涉及两次对 LLM API 的独立调用，而在这两次调用之间，应用程序需要执行一个可能耗时较长的外部任务。因此，应用程序必须负责维护整个对话的上下文状态，包括消息历史和 tool_call_id，以确保信息的连续性和正确性。这与简单的、无状态的请求-响应模式有根本区别，也为构建更复杂的、需要长期记忆和多步规划的智能体系统奠定了架构基础 [6]。

第三部分：战略优势与变革性能力

本节将深入探讨函数调用技术所带来的深远影响，详细阐述它如何将大型语言模型从新奇的技术展示，提升为能够解决现实世界问题的强大、实用的工具。

3.1 接入动态现实：集成实时数据

函数调用技术最直接、最显著的优势之一，是彻底打破了大型语言模型受限于其静态训练数据的“知识截止日期”[2]。通过调用外部 API，LLM 能够获取并整合实时、动态变化的信息。这意味着，当用户询问当前的天气状况、最新的股票价格、突发新闻头条或实时航班信息时，模型不再依赖其可能已经过时的内部知识，而是可以触发一个相应的工具来获取最新、最准确的数据，并将其融入到回答中[6]。

除了公共的实时数据，函数调用还为 LLM 安全地访问私有和专有数据源打开了大门。企业可以将 LLM 与其内部系统（如客户关系管理（CRM）数据库、库存管理系统、项目管理工具）或用户的个人应用（如日历、待办事项列表）进行集成。这种集成方式的优势在于，敏感数据本身无需作为训练数据提供给模型，也无需离开企业的安全边界。LLM 仅通过定义好的、受控的工具接口来请求所需信息，从而在保护数据隐私和安全的前提下，实现了高度个性化和情境化的交互[4]。例如，一个销售助理可以查询 CRM 获取特定客户的最新沟通记录，一个个人助手可以检查用户的日历以安排会议。

3.2 从“说到”到“做到”：执行真实世界操作与任务自动化

如果说集成实时数据是赋予 LLM “眼睛”和“耳朵”，那么执行真实世界操作则是赋予其“手”和“脚”，这是函数调用带来的最具变革性的能力跃迁。它使得 LLM 的能力从单纯的“信息处理”（说）扩展到了“任务执行”（做）[31]。通过调用相应的 API 或函数，LLM 可以触发实际的动作，例如：发送一封电子邮件、在日历上创建一个会议、向 CRM 系统中添加一条新的销售线索、预订一张机票，甚至控制家中的智能灯光或恒温器[5]。

这种行动能力是实现任务自动化的基石。过去需要用户在多个不同应用程序之间手动切换、点击和输入才能完成的复杂工作流，现在可以通过一个单一的自然语言指令来触发[6]。例如，用户可以说：“帮我总结一下今天上午与 Acme 公司的会议纪要，并根据纪要内容生成一个后续任务列表，然后将任务分配给项目经理，并安排下周三进行一次跟进会议。”一个配备了相应工具（文档处理、任务管理、日历管理）的 AI 智能体可以自主地将这个复杂指令分解为一系列连续的函数调用，并自动完成整个流程，极大地提升了工作效率[32]。

3.3 立足于事实：确保准确性与可靠性

大型语言模型的一个核心挑战是其在处理需要精确性的任务时的不可靠性，以及其产生“幻觉”的倾向[2]。函数调用通过“任务委托”的机制，为解决这一问题提供了强有力的架构保障。对于那些本质上是确定性的任务，如数学运算、逻辑推理或精确的日期计算，让 LLM 依靠其概率性的语言模

型去“心算”是不可靠的。通过定义一个专门的计算器工具或日期处理工具，LLM 可以将这类任务外包给经过严格测试、保证结果百分之百准确的外部代码来完成 [4]。

更重要的是，通过将模型的回答“锚定”在从权威外部来源获取的真实数据上，函数调用显著降低了模型产生幻觉的风险 [3]。当模型的回答不是基于其内部知识的模糊记忆，而是基于刚刚通过 API 调用从事实来源（如公司的知识库、官方的天气服务或实时的金融数据提供商）获取的、可验证的数据时，其回答的准确性和可信度得到了根本性的提升。

这种机制从根本上改变了我们与 LLM 之间的信任关系。过去，我们必须对 LLM 的每一个输出都持怀疑态度，因为它可能是一个概率上看似合理但实际上错误的“幻觉”。现在，当一个由函数调用支持的回答被提供时，它背后有一个可追溯的数据来源。例如，如果一个 AI 助手提供了某个产品的库存数量，应用程序可以记录下这个信息是在特定时间通过调用 get_inventory_status 工具获得的。这就建立了一条清晰的审计链，将信任的基础从 LLM 内部不可见的、复杂的状态，转移到了外部的、可验证的、权威的工具上。这种可验证性使得 LLM 不仅能力更强，而且更加可信赖，从而加速了其在商业、金融、医疗等对准确性要求极高的关键业务流程中的应用 [33]。

3.4 结构化数据提取与自然语言接口

函数调用不仅是与外部世界交互的桥梁，也是一种极其高效的、将非结构化的自然语言转换为结构化数据的机制 [2]。通过定义一个具有特定 JSON Schema 的工具，开发者可以指示模型从一段自由文本中提取关键信息，并严格按照预设的格式进行组织。例如，可以定义一个

extract_contact_info 工具，其参数 schema 要求提取姓名、邮箱和电话号码。当向模型提供一段包含这些信息的文本时，模型会返回一个精确填充了这些字段的 JSON 对象，而无需开发者编写复杂的正则表达式或解析逻辑 [20]。

这一能力可以进一步扩展，为复杂的后端系统（如数据库或 API）构建强大的自然语言接口。用户无需学习复杂的查询语言（如 SQL）或 API 文档，只需用日常语言提问，LLM 就能充当“翻译官”的角色。例如，一个业务分析师可以问：“上个季度我们在欧洲市场的销售额前三的产品是哪些？”LLM 可以将这个问题自动翻译成一条精确的、语法正确的 SQL 查询语句，然后通过一个 execute_sql 工具来执行这条查询 [2]。这种方式极大地降低了技术门槛，使非技术人员也能与复杂的数据系统进行深度交互，从而实现了数据的民主化。

这种交互模式的经济价值在于它极大地降低了用户的“认知摩擦”。许多数字任务要求用户在不同的软件界面之间切换，并将他们的高层次意图（例如“为客户安排一次跟进”）转化为一系列低层次的、繁琐的点击和输入操作 [15]。这个过程消耗了大量的精神资源。函数调用让 LLM 成为一个通用接口，用户只需用自然语言表达他们的最终目标，LLM 就能在后台处理所有底层的、跨系统的函数执行 [6]。用户不再需要关心具体的操作步骤或学习多个复杂的用户界面，聊天窗口成为了他们唯一的、统一的交互入口 [31]。这种认知摩擦的减少直接转化为生产力的提升和效率的飞跃，这

也是企业在工作流程中积极采用此类技术的核心经济驱动力[5]。

第四部分：智能体的架构蓝图

本节将从单个函数调用的机制，过渡到探讨如何通过组织和编排一系列的函数调用来构建能够自主推理和行动的智能体（Agent）系统。我们将深入分析其架构模式，并阐明其与简单函数调用的区别。

4.1 函数调用与智能体系统：对比分析

理解函数调用和智能体系统之间的关系至关重要：前者是基础能力，而后者是基于该能力构建的复杂架构。

- 函数调用（构建模块）：这是一种直接的、通常是单步的交互机制。在一个典型的请求-响应模型中，LLM 根据用户输入决定调用一个预定义的函数。整个流程的控制权和编排逻辑主要由外部的应用程序代码掌握[33]。应用程序负责发起请求、执行函数、反馈结果，LLM 在其中扮演的是一个“按需响应的工具选择器”的角色。
- 智能体系统（复杂架构）：这是一个更复杂的、具备一定自主性的系统。在这个系统中，LLM 扮演着“大脑”或“中央推理引擎”的角色。它不仅仅是响应单个请求，而是为了达成一个更宏大的目标，自主地制定一个多步骤的计划。它会根据计划和当前状态，决定在何时、以何种顺序、使用哪些工具，并且能够处理工具执行过程中可能出现的错误，通常在一个循环（loop）中运行，直到目标达成[12]。

二者之间的核心区别在于自主性、状态管理、复杂性和规划能力。一个简单的函数调用是执行一个明确的命令，而一个智能体则拥有一个目标，并能自主地设计和执行一系列动作来实现这个目标[33]。这种从“执行命令”到“实现目标”的跃迁，是构建真正智能应用的关键。

为了更清晰地展示二者的差异，下表从多个维度进行了对比：

表 **4.1**：简单函数调用与复杂智能体系统的对比

特性	函数调用	大语言模型智能体
核心概念	LLM 与外部世界的直接交互	智能自主系统

架构	简单的请求-响应模型	复杂的模块化架构（大脑、记忆、规划、工具）
操作模式	单步的函数调用	迭代式的推理、规划、行动、观察循环
外部交互	对预定义函数的直接调用	在计划中自主使用工具
状态管理	主要由调用应用程序管理	内部记忆（短期和长期）
自主性水平	较低	较高
复杂性	较低	较高
主要用例	特定动作、数据检索、API 集成	复杂任务、推理、规划、高级助手

这个表格清晰地揭示了，函数调用是实现智能体的必要条件，但智能体系统本身是一个远超单个函数调用的、更为复杂的工程概念。开发者在设计 AI 应用时，需要根据任务的复杂度和对自主性的要求，来决定是构建一个简单的、由外部逻辑驱动的工具集成应用，还是一个由 LLM 驱动的、具备自主规划能力的智能体系统。

4.2 ReAct 循环 (推理 + 行动)：实现多步推理

ReAct (Reason + Act) 是一个极具影响力的智能体架构框架，它通过模拟人类解决问题的思维过程，使 LLM 能够处理需要多步推理和与环境交互的复杂任务 [5]。其核心思想是将任务分解为一个迭代循环，每个循环包含三个关键阶段：

1. 推理 **(Reason)**：在这一阶段，LLM 会进行“内心独白”或“思考”。它会分析当前的目标、已经掌握的信息（来自之前的观察），以及下一步需要做什么才能更接近目标。这个过程通常会生成一段描述其思考过程的文本，例如：“用户想知道巴黎的温度，但我首先需要知道巴黎的经纬度坐标。所以，我的下一步是使用 get_coordinates 工具。”
2. 行动 **(Act)**：基于推理阶段得出的结论，LLM 会生成一个具体的 tool_call 指令来执行计划中的下一步。例如，生成 {"name": "get_coordinates", "arguments": {"city": "Paris"}}。
3. 观察 **(Observe)**：应用程序代码执行该工具调用，并将返回的结果（例如，巴黎的坐标）作为“观察”结果反馈给 LLM。这个结果会成为下一轮循环中“推理”阶段的重要输入信息。

这个“推理-行动-观察”的循环会持续进行，直到 LLM 判断任务已经完成。例如，在获取坐标后，

LLM 在下一轮推理中会说：“现在我已经有了坐标，我需要使用 get_temperature 工具来查询天气。” 然后它会发起第二次行动。这种迭代式的反馈循环，使得智能体能够动态地调整其计划，处理中间步骤的结果，并逐步解决那些无法一步到位的问题，极大地增强了其问题解决能力[12]。

4.3 编排复杂性：链接工具调用

在智能体系统中，单个的工具调用很少能独立解决复杂问题。真正的威力来自于将多个工具调用有效地链接和编排起来，形成复杂的工作流。

- 顺序链接 **(Sequential Chaining)**：这是最基本的多步操作模式，即前一个工具调用的输出成为后一个工具调用的输入。这在存在依赖关系的任务中非常常见。例如，要回答“当前公司市值最高的员工是谁？”，智能体可能需要顺序执行两个工具：首先调用 get_highest_value_stock() 获取股票代码，然后将该股票代码作为参数调用 get_top_shareholder(stock_symbol)[13]。
- 并行链接 **(Parallel Chaining)**：当用户的请求包含多个可以独立执行的子任务时，一个高效的 LLM 可以在一轮决策中生成多个 tool_calls。应用程序接收到这些指令后，可以并行地执行这些函数，从而显著缩短总体的响应时间。例如，当用户问“东京现在是什么天气？巴黎现在几点了？”时，模型可以同时生成对 get_weather(location="Tokyo") 和 get_time(location="Paris") 的调用请求。这种并行处理能力是衡量一个智能体系统效率的重要指标[3]。
- 条件逻辑 **(Conditional Logic)**：更高级的智能体能够在其推理引擎中实现条件判断。它会根据前一个工具调用的结果，来决定下一步应该调用哪个工具。例如，一个客服智能体在处理退款请求时，可以先调用 get_order_status(order_id)。如果返回的状态是“已发货”，则调用 initiate_return_process() 工具；如果返回的状态是“处理中”，则调用 cancel_order() 工具。这种动态决策能力使得智能体能够适应不同的情况，执行更加复杂和智能的工作流[27]。

4.4 构建自主系统：智能体的核心架构

一个典型的自主智能体系统，无论其具体实现如何，通常都包含三个不可或缺的核心组件[43]：

1. 模型 **(大脑)**：这是智能体的认知核心，通常是一个强大的大型语言模型。它负责所有的推理、规划和决策任务，是整个系统的“指挥官”[12]。
2. 工具 **(手脚)**：这是一组预定义的函数或 API，是智能体与外部世界进行交互、获取信息和执行动作的手段。工具集的丰富性和可靠性直接决定了智能体的能力边界[43]。
3. 指令 **(目标/个性)**：这通常是通过系统提示（System Prompt）来设定的。它为智能体定义了其总体的目标、行为准则、约束条件和“个性”。一份精心设计的指令对于引导智能体产生预期中的、可靠的行为至关重要[10]。

为了简化构建这些复杂系统的工程挑战，社区中涌现出了许多开源的编排框架，其中最著名的是 LangChain 和 LangGraph。这些框架提供了一套高级的抽象和工具，用于定义智能体、管理工具集、处理状态记忆以及设计智能体与工具之间的控制流。它们将 ReAct 循环、工具链接等复杂的架构模式封装成易于使用的组件，让开发者可以更专注于业务逻辑的实现，而不是底层的编排细节 [14]。

智能体的兴起预示着人机交互范式的一次深刻变革。传统的软件交互是“命令式”的：用户必须通过一系列精确的点击和指令来告诉软件“如何”完成一个任务。而基于函数调用的智能体系统则开启了“声明式”交互的时代：用户只需告诉智能体他们“想要什么”（目标），而智能体则自主地规划出“如何做”（工具调用的序列）[42]。例如，用户不再需要手动打开日历、查找空闲时间、创建会议、添加参会者等一系列命令式操作，而只需声明一个目标：“下周帮我和张三安排一次会议，并发送邀请。”这种将规划和执行的负担从人类转移到 AI 的范式转变，将极大地改变我们与数字世界的互动方式，使得用户的核心技能从程序化执行转向了有效的目标设定和沟通 [6]。

然而，这种强大的自主性也带来了新的工程挑战。智能体在一个循环中运行，引入了多个潜在的失败点：LLM 可能做出错误的规划决策，可能陷入无限循环，可能误解工具的输出，或者无法优雅地处理来自外部工具的错误 [27]。因此，构建一个鲁棒的智能体系统，不仅需要定义好工具，更需要在其编排层设计出有效的规划、错误处理和状态管理机制。工程的重心从简单的 API 集成转向了复杂的状态系统设计，这需要新的调试和评估工具（如 LangSmith）来确保系统的可靠性和可预测性 [14]。

第五部分：主流工具使用实现方案的比较分析

本节将对三大主流大型语言模型提供商——OpenAI、Google 和 Anthropic——的函数调用（或工具使用）API 进行深入的技术剖析。我们将比较它们在工具定义、调用语法、响应处理、核心能力及开发者体验方面的异同，并提供相应的代码示例。

5.1 OpenAI (GPT 系列)：成熟的行业标准

OpenAI 作为函数调用功能的早期推动者，其 API 设计已成为事实上的行业标准，被广泛采用和模仿。

- **API 结构**：其核心在于 Chat Completions API 中的 tools 和 tool_choice 两个参数。tools 参数接受一个由函数定义对象组成的数组，每个对象都遵循特定的 JSON Schema。tool_choice 参数则提供了对模型行为的精确控制，可以设为 "auto"（默认，由模型决定是否使用工具）、"none"（强制模型不使用任何工具），或指定一个具体的函数名称来强制模型使

用该函数 [22]。

- 响应处理：当模型决定调用工具时，API 响应的 finish_reason 字段会返回 "tool_calls"。同时，在 message 对象中会包含一个 tool_calls 数组。应用程序需要检查这个 finish_reason，然后解析 tool_calls 数组以获取需要执行的函数名、参数和唯一的 tool_call_id [22]。
- 核心特性：OpenAI 的实现以其成熟、稳定和详尽的文档而著称。它对并行函数调用提供了强大的支持，允许模型在单次响应中请求执行多个独立的工具。一个重要的可靠性增强功能是“结构化输出”（Structured Outputs），通过在函数定义中设置 strict: true，可以保证模型生成的参数 JSON 严格符合开发者提供的 schema，极大地减少了解析错误和不一致性 [45]。
- 代码示例 **(Python)**：

```
Python
from openai import OpenAI
import json

client = OpenAI()

# 1. 定义工具
tools =},
            },
            "required": ["location"],
        },
    },
  }
]

messages =

# 2. 发起首次 API 调用
response = client.chat.completions.create(
  model="gpt-4o",
  messages=messages,
  tools=tools,
  tool_choice="auto",
)

response_message = response.choices.message
tool_calls = response_message.tool_calls

# 3. 处理模型响应
if tool_calls:
  messages.append(response_message)  # 将助手的回复（包含 tool_calls）加入历史记录

  # 4. 执行函数
  available_functions = {
```

```
        "get_current_weather": get_current_weather, # 假设 get_current_weather 是一个已定义的
Python 函数
    }
    for tool_call in tool_calls:
        function_name = tool_call.function.name
        function_to_call = available_functions[function_name]
        function_args = json.loads(tool_call.function.arguments)
        function_response = function_to_call(
            location=function_args.get("location"),
            unit=function_args.get("unit"),
        )

        # 5. 将执行结果反馈给模型
        messages.append(
            {
                "tool_call_id": tool_call.id,
                "role": "tool",
                "name": function_name,
                "content": function_response,
            }
        )

    # 6. 发起第二次 API 调用以获取最终回复
    second_response = client.chat.completions.create(
        model="gpt-4o",
        messages=messages,
    )
    print(second_response.choices.message.content)
```

22

5.2 Google (Gemini 系列)：灵活性与生态系统集成

Google Gemini 的函数调用功能在设计上与 OpenAI 高度相似，但提供了更高的灵活性和更紧密的生态系统集成，特别是在开发者体验方面。

- **API 结构**：Gemini API 同样使用 tools 参数来传递工具定义，其结构为 FunctionDeclarations 列表 [26]。
- **工具定义**：Gemini 在工具定义方面提供了极大的灵活性。开发者不仅可以使用标准的

OpenAPI JSON Schema（与 OpenAI 兼容），还可以直接使用带有类型注解和文档字符串（docstrings）的原生 Python 函数来定义工具。Gemini 的 SDK 能够自动从这些 Python 函数中解析并生成所需的 schema，这大大简化了开发流程，提升了代码的可读性和可维护性 [26]。

- 核心特性：Gemini 同样强大地支持并行和顺序函数调用 [26]。其最引人注目的特性之一是提供了一个与 OpenAI 兼容的 API 端点。这意味着已经基于 OpenAI API 构建了应用程序的团队，可以以极低的代码修改成本，平滑地迁移或切换到使用 Gemini 模型，这极大地降低了技术选型的壁垒 [45]。此外，Gemini 与 Google Cloud (Vertex AI) 和 Google Workspace (Docs, Sheets 等) 的深度集成为其在企业级应用和生产力工具场景中提供了独特的优势 [53]。
- 代码示例 **(Python,** 使用原生函数定义**)**：

```python
import google.generativeai as genai

# 假设已配置好 API key
# genai.configure(api_key="YOUR_API_KEY")

# 1. 使用原生 Python 函数定义工具
def get_current_weather(location: str, unit: str = "celsius"):
    """Get the current weather in a given location."""
    #... 函数实现...
    return {"temperature": "22", "condition": "sunny"}

# 2. 创建模型实例并绑定工具
model = genai.GenerativeModel(
    model_name='gemini-1.5-pro-latest',
    tools=[get_current_weather]
)
chat = model.start_chat(enable_automatic_function_calling=True)

# 3. 发起调用，SDK 自动处理循环
response = chat.send_message("What's the weather in San Francisco?")
print(response.text)
```

26

5.3 Anthropic (Claude 系列)：注重推理过程与并行能力

Anthropic 的 Claude 系列模型在工具使用（Tool Use）的实现上，展现了其对模型推理过程透明度和并行处理能力的重视。

- **API** 结构：与 OpenAI 类似，Claude API 使用 tools 参数定义工具，并提供 tool_choice 参数

来强制或引导工具的使用[19]。

- 响应处理：当 Claude 决定使用工具时，API 响应的 stop_reason 字段为 tool_use。其独特的之处在于，content 响应数组可以同时包含类型为 text 和 tool_use 的内容块。这使得模型可以在生成工具调用指令之前，先输出一段“思考”或解释性的文本（例如，“好的，我将为您查询天气”），这种“边想边做”的模式为用户提供了更自然的交互体验和更高的透明度[19]。
- 核心特性：
  - 客户端工具 **vs.** 服务器端工具：Claude 明确区分了两类工具。客户端工具（Client Tools）是由开发者的应用程序在自己的环境中执行的自定义函数。而服务器端工具（Server Tools）则是由 Anthropic 的服务器执行的预置功能，如网页搜索 (web_search) 和网页内容抓取 (web_fetch)，开发者只需声明使用，无需自行实现[19]。
  - 强大的并行工具使用：Claude 对并行调用给予了高度重视，并且可以通过系统提示（System Prompt）进一步鼓励模型进行并行操作，以提升效率[19]。
  - **“思考”过程可见**：部分模型版本支持在响应中返回其详细的推理过程，这对于调试复杂的智能体行为、理解模型决策逻辑非常有价值[56]。
  - 独特的专用工具：Anthropic 提供了一些独特的、预定义的客户端工具，例如 computer_use，它允许模型通过模拟鼠标和键盘操作来与图形用户界面（GUI）进行交互，实现了桌面自动化，这是一个非常前沿和强大的能力[56]。
- 代码示例 **(Python)**：

```Python
import anthropic
import json

client = anthropic.Anthropic()

# 1. 定义工具
tools = [
    #... 与 OpenAI 示例类似的工具定义...
]

# 2. 首次调用
response = client.messages.create(
    model="claude-3-opus-20240229",
    max_tokens=1024,
    messages=,
    tools=tools,
)

# 3. 处理响应
if response.stop_reason == "tool_use":
    tool_use = next(block for block in response.content if block.type == "tool_use")
    tool_name = tool_use.name
    tool_input = tool_use.input
```

```
    tool_use_id = tool_use.id

    # 4. 执行函数
    #... 执行 tool_name(tool_input)...
    tool_result_content = "The weather is 65 degrees and sunny."

    # 5. 反馈结果
    second_response = client.messages.create(
        model="claude-3-opus-20240229",
        max_tokens=1024,
        messages=,
            },
        ],
        tools=tools,
    )
    print(second_response.content.text)
```

19

5.4 技术深度对比：摘要表

为了直观地比较这三个平台的 API 实现，下表总结了关键的技术差异。

表 **5.4**：主流平台函数调用 **API** 技术对比

特性	OpenAI (GPT)	Google (Gemini)	Anthropic (Claude)
工具定义参数	tools	tools	tools
工具定义格式	JSON Schema	JSON Schema, Python Docstrings	JSON Schema
强制工具使用参数	tool_choice	tool_choice	tool_choice
工具调用响应信号	finish_reason: "tool_calls"	返回 FunctionCall 对象	stop_reason: "tool_use"

并行调用支持	强大	强大	强大（可通过提示增强）
保证 **Schema** 遵循	是 (strict: true)	不明确保证	不明确保证
独特功能	成熟的生态系统，Assistants API	Python 原生定义，OpenAI 兼容 API	客户端/服务器端工具，“思考”过程，computer_use 工具
开发者体验	文档完善，社区庞大	灵活，易于从 OpenAI 迁移	注重安全与透明度

这张表格为技术决策者提供了一个清晰的参考框架。选择 OpenAI 意味着选择一个成熟、稳定且拥有庞大社区支持的生态系统。选择 Google Gemini 则能享受到极佳的开发灵活性（特别是对于 Python 开发者）和轻松迁移的便利。而选择 Anthropic Claude 则更适合那些需要高度透明的推理过程、强大的并行处理能力以及独特的桌面自动化等前沿功能的应用场景。

第六部分：实际应用与行业案例研究

本节将理论与实践相结合，通过具体的、跨行业的真实世界案例，展示函数调用技术如何将抽象的 AI 能力转化为具有商业价值的实际应用。

6.1 电子商务：构建智能购物助手

在电子商务领域，函数调用技术正在驱动新一代智能购物助手的诞生，它们能够提供高度个性化、实时且端到端的购物体验。

- 核心应用场景：
  - 实时产品发现与搜索：传统的关键词搜索正在被自然语言对话所取代。用户可以提出复杂的、口语化的查询，例如：“我想找一双适合在公路上跑步的蓝色男士跑鞋，尺码是 42 码，有足弓支撑功能的。” 配备了 searchProducts() 工具的 AI 助手可以精确地解析出颜色、品类、尺码、功能等多个参数，并实时查询产品目录数据库，返回最匹配的结果 [10]。
  - 订单状态查询与物流追踪：用户无需再去翻找邮件或登录复杂的订单系统，只需向助手提问：“我的最新订单到哪里了？” 智能体就会调用 getOrderStatus(customerId) 或 trackShipment(orderId) 等工具，直接从后端系统获取最新的物流信息，并以自然语言

的形式告知用户 [28]。
    - 深度个性化推荐：通过调用工具来获取用户的历史购买记录、浏览行为和愿望清单，AI 助手可以超越简单的“猜你喜欢”，提供更具情境化的推荐。例如，它可以说：“我看到您上个月买了一台咖啡机，这里有几款评价很高的配套咖啡豆推荐给您。” [28]。
    - 端到端交易流程：最高级的购物助手能够引导用户完成从意向到支付的全过程。这个工作流可能涉及多个工具的顺序调用：首先使用 searchProducts() 找到商品，然后调用 checkStockAvailability() 确认库存，接着使用 calculatePrice() 计算包含折扣和运费的总价，最后通过 createCheckoutLink() 生成一个可供用户直接点击支付的链接，整个过程在一次流畅的对话中完成 [27]。

6.2 数据分析：为数据库创建自然语言接口

函数调用技术正在彻底改变人与数据的交互方式，尤其是在“文本到 SQL”（Text-to-SQL）这一关键应用场景中，它使得数据分析的门槛被前所未有地降低 [38]。

- **“文本到 SQL”**工作流：
    1. 用户提问：一个非技术背景的业务人员用自然语言提出一个数据查询需求，例如：“请展示上个季度华北地区销售额最高的五位销售代表及其业绩。” [58]。
    2. **LLM** 理解与翻译：AI 系统被提供了一个 execute_sql_query 工具以及目标数据库的 schema（表结构、字段名、注释等）。LLM 的推理引擎会理解用户的业务问题，并将其翻译成一条语法完全正确的 SQL 查询语句，例如 SELECT representative_name, SUM(sales_amount) FROM sales_records WHERE region = '华北' AND quarter = 'Q2' GROUP BY representative_name ORDER BY SUM(sales_amount) DESC LIMIT 5; [29]。
    3. 工具调用：LLM 生成一个对 execute_sql_query 工具的调用请求，并将上面生成的 SQL 字符串作为参数传递 [38]。
    4. 执行与反馈：应用程序代码接收到这个指令后，安全地连接到数据库，执行该 SQL 查询，获取查询结果（一个数据表）。
    5. 结果合成：查询结果被返回给 LLM，LLM 再将这个结构化的数据表，以清晰的、易于理解的自然语言（甚至可以是图表或摘要）呈现给用户 [38]。
- 核心价值：这种模式实现了数据访问的民主化。它让组织中的每一个人，无论是否掌握 SQL 或其他数据分析工具，都能够通过简单的对话来探索和利用数据，从而做出更明智的、由数据驱动的决策 [58]。

6.3 企业自动化：集成 CRM、日历与内部 API

函数调用是推动企业内部工作流程自动化的核心引擎，它能够将分散在不同系统中的操作串联起

来，形成统一、高效的自动化流程 [6]。

- 核心应用场景：
  - **CRM** 系统集成：销售人员可以在聊天界面中直接下达指令，如："将我与 XYZ 公司的通话记录总结一下，并更新到 Salesforce 的联系人备注中。" AI 智能体可以调用内部的转录服务、摘要模型，最后通过调用 Salesforce API 的工具来完成记录的更新 [6]。
  - 智能会议安排：安排会议的繁琐过程可以被完全自动化。当用户说："下周帮我和王经理找个时间开会讨论项目进展。" 智能体可以触发一系列工具调用：首先通过日历 API 检查双方的空闲时间，找到重叠的时间段，然后创建一个会议邀请，并自动发送给与会者 [6]。
  - 自动化客户支持：一个高级的客服智能体可以处理复杂的客户请求。例如，当收到退货请求时，它可以先调用订单系统 API 检查订单状态，再调用库存系统 API 检查商品是否可退，最后调用财务系统 API 来处理退款，整个过程无需人工干预，极大地提高了服务效率和客户满意度 [5]。

这些应用案例揭示了一个更深层次的趋势：函数调用正在催生一个"长尾"的、超个性化的自动化市场。传统的自动化技术（如 RPA）由于实施成本高，通常只适用于那些高频率、标准化的核心业务流程 [61]。然而，在日常工作中存在大量零散、多变、个性化的"长尾"任务，这些任务用传统方法进行自动化是不经济的。基于 LLM 和函数调用的智能体，由于其能够动态地、即时地组合各种基础工具（如"发送邮件"、"创建文档"、"查询数据"），使得用户可以通过简单的自然语言描述，来即时构建和执行这些一次性或小众的自动化工作流 [27]。这极大地降低了自动化的门槛，使得个人和团队能够将他们独特的、特定的工作流程自动化，释放出巨大的生产力潜力，这是以往的技术所无法企及的 [31]。

第七部分：实施最佳实践与未来展望

本节旨在为开发者提供在实际项目中应用函数调用技术的可行性建议，并探讨该技术未来的发展方向，揭示其在构建下一代人工智能系统中的潜力。

7.1 设计高效的工具：函数描述的艺术

工具设计的质量直接决定了智能体的性能和可靠性。其中，函数描述的撰写是一项需要投入巨大精力的核心任务。

- 清晰与详尽至上：正如前文所述，工具的 description 字段是 LLM 做出决策的主要依据。因此，描述必须极其清晰、详尽且无歧义。它应该全面地解释工具的功能、适用场景、每个参数的含义、潜在的限制以及不适用的情况。一个高质量的描述是确保模型能够准确、可靠地选择和使用工具的最重要因素 [16]。

- 函数的粒度：在设计工具集时，应遵循“单一职责原则”。倾向于创建一系列功能单一、目标明确的小工具，而不是一个功能庞杂的“万能”工具。例如，将“处理用户账户”这个大功能分解为 get_user_profile、update_user_address、reset_user_password 等多个更细粒度的工具。这样做不仅能让每个工具的描述更清晰，也使得 LLM 在进行规划和推理时更容易、更准确地选择合适的工具组合 [16]。
- 命名规范：为函数及其参数选择清晰、具有描述性的名称。例如，get_user_by_email 远比 fetch_data 更能向模型传达其意图。良好的命名本身就是一种有效的沟通方式，能够降低模型理解和使用工具的难度 [16]。

7.2 安全性与可靠性：风险规避与错误处理

将 LLM 与能够执行真实世界操作的工具相连接，在带来强大能力的同时，也引入了新的安全风险和可靠性挑战。

- 防范提示注入 **(Prompt Injection)**：这是最主要的安全威胁之一。恶意用户可能通过精心构造的输入，诱骗或指令 LLM 调用其本不应调用的函数，或者使用恶意的参数（例如，在调用发送邮件的工具时，将收件人改为攻击者，内容改为敏感信息）。构建坚固的“护栏”（Guardrails）是必不可少的 [10]。
- 缓解策略：
  - 输入验证与净化：在将任何用户输入传递给 LLM 之前，必须进行严格的验证和净化，过滤掉已知的恶意模式和指令 [10]。
  - 二次模型审核：可以引入另一个 LLM 作为审核层，专门用于评估用户输入的意图是否具有潜在的恶意性 [10]。
  - 遵循最小权限原则：为工具授予其完成任务所需的最小权限。例如，如果一个工具只需要查询数据库，就应为其配置只读权限的数据库账户，绝不能给予写入或删除权限 [22]。
  - 避免不安全的执行：绝对禁止在未经验证的情况下，对 LLM 返回的参数直接使用 eval() 等动态代码执行函数，这会造成严重的安全漏洞 [10]。
  - 用户确认机制：对于任何会产生重大影响或不可逆后果的操作（如删除数据、进行支付、发送重要邮件），必须在执行前增加一个明确的用户确认步骤 [22]。
- 错误处理：应用程序代码必须具有鲁棒性。它需要能够优雅地处理外部 API 调用失败、网络中断等情况。同时，它也需要验证 LLM 返回的 tool_calls 对象的结构和内容是否符合预期，并为可能发生的瞬时故障实施重试逻辑 [14]。

7.3 工具使用的未来：走向自主工具创建与发现

函数调用技术本身仍在快速演进，其未来的发展方向预示着更加自主和动态的 AI 系统。

- **LLM 作为工具制造者 (LATM)**：这是一个前沿的研究领域，其核心思想是让 LLM 不再仅仅是现有工具的“使用者”，更能成为新工具的“创造者”。当面对一个没有现成工具可以解决的新问题时，一个具备 LATM 能力的 LLM 可以动态地编写一个新的 Python 函数或脚本来解决该问题，并将其添加到自己的工具库中，以备将来使用。这使得智能体的能力可以动态扩展，而无需人工干预 [18]。
- 动态工具发现 **(MCP)**：当前的模式通常要求开发者在对话开始时就提供一个固定的工具列表。未来的智能体可能会更加动态，它们可以在运行时查询一个“工具注册中心”或“工具服务器”，来发现当前环境中可用的工具。模型上下文协议（Model Context Protocol, MCP）就是一个旨在标准化这一过程的开放协议，它允许智能体与工具解耦，使其更具模块化和可扩展性，能够适应不断变化的环境和需求 [10]。
- 智能体流程自动化 **(APA)** 的兴起：我们正在见证一个从机器人流程自动化（Robotic Process Automation, RPA）到智能体流程自动化（Agentic Process Automation, APA）的范式转变。RPA 依赖于人工预先录制或编写的、僵化的工作流脚本。而 APA 则利用 LLM 的推理能力，让智能体能够根据高层次的自然语言指令，自主地理解、规划和编排工作流程。函数调用正是实现这一革命性转变的核心使能技术 [61]。

这一切都指向一个未来：人类开发者与 AI 智能体之间将形成一种深刻的共生关系。当前，是人类定义工具，AI 使用工具。随着 LATM 等技术的发展，AI 将开始能够自己创造简单的工具。然而，这些由 AI 生成的工具，仍然需要依赖于由人类工程师构建的、复杂的、安全的、高性能的基础设施、API 和数据库。未来的工作模式很可能演变为：人类专家负责构建稳定、可靠的底层“原子能力”（如核心业务 API），而 AI 智能体则在这些原子能力之上，充当一个动态的、灵活的编排层，通过即时创造和组合更高层次的“复合工具”来解决具体的、情境化的任务。这种人机协作模式，将人类的深度工程能力与 AI 的快速适应和自动化能力相结合，必将以前所未有的速度推动技术创新和问题解决。

引用的著作

1. Large language model - Wikipedia, 访问时间为 九月 13, 2025, https://en.wikipedia.org/wiki/Large_language_model
2. Function Calling with LLMs - Prompt Engineering Guide, 访问时间为 九月 13, 2025, https://www.promptingguide.ai/applications/function_calling
3. Leveraging LLM function calling to harness real-time knowledge - Fabrity, 访问时间为 九月 13, 2025, https://fabrity.com/blog/leveraging-llm-function-calling-to-harness-real-time-knowledge/
4. Defining and using tools with the LLM Mesh - Dataiku Developer Guide, 访问时间为 九月 13, 2025, https://developer.dataiku.com/latest/tutorials/genai/agents-and-tools/llm-agentic/tools/index.html
5. Transform LLMs into Smart Assistants with Function Calling - Tredence, 访问时间为 九月 13, 2025, https://www.tredence.com/blog/harnessing-the-power-of-function-calling-transforming-llms-into-smart-assistants
6. An introduction to function calling and tool use - Apideck, 访问时间为 九月 13,

2025, https://www.apideck.com/blog/llm-tool-use-and-function-calling
7. Function Calling in Large Language Models (LLMs) | by Mehmet Ozkaya - Medium, 访问时间为 九月 13, 2025, https://mehmetozkaya.medium.com/function-calling-in-large-language-models-llms-8e7712c0e60f
8. Function Calling in LLM - Medium, 访问时间为 九月 13, 2025, https://medium.com/@danushidk507/function-calling-in-llm-e537b286a4fd
9. vercel.com, 访问时间为 九月 13, 2025, https://vercel.com/guides/what-is-an-llm-tool#:~:text=A%20Large%20Language%20Model%20(LLM)%20tool%20is%20a%20way%20to,functions%2C%20APIs%2C%20or%20systems.
10. Function calling using LLMs - Martin Fowler, 访问时间为 九月 13, 2025, https://martinfowler.com/articles/function-call-LLM.html
11. Tool Calling: How LLMs Connect To The Real World | by Paul Fruitful | Medium, 访问时间为 九月 13, 2025, https://medium.com/@fruitful2007/tool-calling-how-llms-connect-to-the-real-world-7c32f197a45d
12. How Do LLMs Handle Function Calls with External Libraries/APIs? : r/AI_Agents - Reddit, 访问时间为 九月 13, 2025, https://www.reddit.com/r/AI_Agents/comments/1ic8lo5/how_do_llms_handle_function_calls_with_external/
13. Function calling in LLM agents - Symflower, 访问时间为 九月 13, 2025, https://symflower.com/en/company/blog/2025/function-calling-llm-agents/
14. Function Calling in Agentic Workflows | by Alex Gilmore | Neo4j Developer Blog | Medium, 访问时间为 九月 13, 2025, https://medium.com/neo4j/function-calling-in-agentic-workflows-92ff33be0975
15. What is Function Calling with LLMs? - Hopsworks, 访问时间为 九月 13, 2025, https://www.hopsworks.ai/dictionary/function-calling-with-llms
16. Function Calling - Hugging Face, 访问时间为 九月 13, 2025, https://huggingface.co/docs/hugs/guides/function-calling
17. What Are Tools in the Scope of LLMs and Why Are They So Important - Alfred Nutile @ DailyAi.Studio, 访问时间为 九月 13, 2025, https://alnutile.medium.com/what-are-tools-in-the-scope-of-llms-and-why-are-they-so-important-f57f76190e58
18. Integrating External Tools with Large Language Models (LLM) to Improve Accuracy - arXiv, 访问时间为 九月 13, 2025, https://arxiv.org/html/2507.08034v1
19. Tool use with Claude - Anthropic, 访问时间为 九月 13, 2025, https://docs.anthropic.com/en/docs/agents-and-tools/tool-use/overview
20. How to implement tool use - Anthropic API, 访问时间为 九月 13, 2025, https://docs.anthropic.com/en/docs/agents-and-tools/tool-use/implement-tool-use
21. arxiv.org, 访问时间为 九月 13, 2025, https://arxiv.org/html/2507.08034v1#:~:text=These%20schemas%20act%20like%20blueprints,the%20LLM's%20internal%20processing%20capabilities.
22. How to use function calling with Azure OpenAI in Azure AI Foundry ..., 访问时间为

九月 13, 2025, https://learn.microsoft.com/en-us/azure/ai-foundry/openai/how-to/function-calling
23. Integrating Large Language Models with External Tools: A Practical Guide to API Function Calls - Complere Infosystem, 访问时间为 九月 13, 2025, https://complereinfosystem.com/integrate-large-language-models-with-external-tools
24. LLM Function Calling Explained: A Deep Dive into the Request and Response Payloads | by James Tang | Medium, 访问时间为 九月 13, 2025, https://medium.com/@jamestang/llm-function-calling-explained-a-deep-dive-into-the-request-and-response-payloads-894800fcad75
25. Function calling with the Gemini API - YouTube, 访问时间为 九月 13, 2025, https://www.youtube.com/watch?v=mVXrdvXplj0
26. Function calling with the Gemini API | Google AI for Developers, 访问时间为 九月 13, 2025, https://ai.google.dev/gemini-api/docs/function-calling
27. Function Calling in LLMs – Real Use Cases and Value? : r/AI_Agents - Reddit, 访问时间为 九月 13, 2025, https://www.reddit.com/r/AI_Agents/comments/1iio39z/function_calling_in_llms_real_use_cases_and_value/
28. Understanding Function Calling in LLMs - Zilliz blog, 访问时间为 九月 13, 2025, https://zilliz.com/blog/harnessing-function-calling-to-build-smarter-llm-apps
29. Function Calling with Open-Source LLMs - BentoML, 访问时间为 九月 13, 2025, https://www.bentoml.com/blog/function-calling-with-open-source-llms
30. Use Cases for Function Calling | Learn - Predictable Dialogs, 访问时间为 九月 13, 2025, https://predictabledialogs.com/learn/function-calling-use-cases
31. Function Calling in LLMs - Stack AI, 访问时间为 九月 13, 2025, https://www.stack-ai.com/blog/function-calling-in-llms
32. LLM Function Calling & API Integration | Practical Guide - Future AGI, 访问时间为 九月 13, 2025, https://futureagi.com/blogs/llm-function-calling-2025
33. LLM Agent vs Function Calling: Key Differences & Use Cases, 访问时间为 九月 13, 2025, https://blog.promptlayer.com/llm-agents-vs-function-calling/
34. Understanding Function Calling in LLMs and Its Difference to RAG - NPi AI, 访问时间为 九月 13, 2025, https://docs.npi.ai/blog/understanding-function-calling-in-llm-and-its-difference-to-rag
35. Function Calling: How LLMs Invoke Real-World APIs (OpenAI & Gemini examples) - Medium, 访问时间为 九月 13, 2025, https://medium.com/@akankshasinha247/function-calling-how-llms-invoke-real-world-apis-openai-gemini-examples-266bdd802c03
36. OpenAI Function Calling Tutorial: Generate Structured Output - DataCamp, 访问时间为 九月 13, 2025, https://www.datacamp.com/tutorial/open-ai-function-calling-tutorial
37. Practical Examples of OpenAI Function Calling | by Cobus Greyling - Medium, 访问时间为 九月 13, 2025, https://cobusgreyling.medium.com/practical-examples-of-openai-function-callin

g-a6419dc38775
38. Build a Question/Answering system over SQL data | 🦜 LangChain, 访问时间为 九月 13, 2025, https://python.langchain.com/docs/tutorials/sql_qa/
39. Querying Databases with Function Calling - arXiv, 访问时间为 九月 13, 2025, https://arxiv.org/html/2502.00032v1
40. 17 Proven LLM Use Cases in E-commerce That Boost Sales in 2025 - Netguru, 访问时间为 九月 13, 2025, https://www.netguru.com/blog/llm-use-cases-in-e-commerce
41. Function-Calling vs Agents - AWS Builder Center, 访问时间为 九月 13, 2025, https://builder.aws.com/content/2sryksE4Ga2hAsUksJZfnT8pJnr/function-calling-vs-agents
42. Building Autonomous Agents with LLMs | TechAhead, 访问时间为 九月 13, 2025, https://www.techaheadcorp.com/blog/building-autonomous-agents-with-llms/
43. A practical guide to building agents - OpenAI, 访问时间为 九月 13, 2025, https://cdn.openai.com/business-guides-and-resources/a-practical-guide-to-building-agents.pdf
44. Build an Agent - LangChain, 访问时间为 九月 13, 2025, https://python.langchain.com/docs/tutorials/agents/
45. Claude Sonnet 4's Tool Calling vs. GPT-4 & Gemini: A Deep Dive - Arsturn, 访问时间为 九月 13, 2025, https://www.arsturn.com/blog/claude-sonnet-4-tool-calling-vs-gpt-4-gemini-a-deep-dive
46. How to use tools in a chain | 🦜 LangChain, 访问时间为 九月 13, 2025, https://python.langchain.com/docs/how_to/tools_chain/
47. LangGraph - LangChain, 访问时间为 九月 13, 2025, https://www.langchain.com/langgraph
48. LLM Workflows: From Automation to AI Agents (with Python) - YouTube, 访问时间为 九月 13, 2025, https://www.youtube.com/watch?v=Nm_mmRTpWLg
49. Function Calling in the OpenAI API, 访问时间为 九月 13, 2025, https://help.openai.com/en/articles/8555517-function-calling-in-the-openai-api
50. A Guide to Function Calling in OpenAI - Mirascope, 访问时间为 九月 13, 2025, https://mirascope.com/blog/openai-function-calling
51. Mastering Function Calling with OpenAI APIs: A Deep Dive | by Kshitij Kutumbe - Medium, 访问时间为 九月 13, 2025, https://kshitijkutumbe.medium.com/mastering-function-calling-with-openai-apis-a-deep-dive-386544298141
52. Intro to Function Calling with the Gemini API & Python SDK - Colab - Google, 访问时间为 九月 13, 2025, https://colab.research.google.com/github/GoogleCloudPlatform/generative-ai/blob/main/gemini/function-calling/intro_function_calling.ipynb
53. Comparing OpenAI vs Claude vs Gemini: Which AI API Is Best for Developers? - Djamware, 访问时间为 九月 13, 2025, https://www.djamware.com/post/689e8836a378ff6175921d4a/comparing-openai-vs-claude-vs-gemini-which-ai-api-is-best-for-developers
54. OpenAI GPT 4.1 vs Claude 3.7 vs Gemini 2.5: Which Is Best AI? - YourGPT, 访问时

间为 九月 13, 2025, https://yourgpt.ai/blog/updates/openai-gpt-4-1-vs-claude-3-7-vs-gemini-2-5
55. Web fetch tool - Anthropic API, 访问时间为 九月 13, 2025, https://docs.anthropic.com/en/docs/agents-and-tools/tool-use/web-fetch-tool
56. Computer use tool - Anthropic API, 访问时间为 九月 13, 2025, https://docs.anthropic.com/en/docs/agents-and-tools/tool-use/computer-use-tool
57. Claude Function Calling Made Dead Simple (Anthropic Tool Use) - YouTube, 访问时间为 九月 13, 2025, https://www.youtube.com/watch?v=2HsmNeT8TCg
58. LLM for Data Analysis: Tools, Costs, and Implementation Guide - Binadox, 访问时间为 九月 13, 2025, https://www.binadox.com/blog/llm-for-data-analysis-tools-costs-and-implementation-guide/
59. Building a Data Analytics Agent using LLMs - YouTube, 访问时间为 九月 13, 2025, https://www.youtube.com/watch?v=Ts4ovDipCgA
60. Data Analysis Using LLM(GPT-3.5) | Retail data analytics using langchain, 访问时间为 九月 13, 2025, https://mariamkilibechir.medium.com/data-analysis-using-llm-chatgpt-data-analysis-retail-data-analytics-using-langchain-d770b6e8e2c0
61. WorkflowLLM: Enhancing Workflow Orchestration Capability of Large Language Models, 访问时间为 九月 13, 2025, https://arxiv.org/html/2411.05451v1
62. How to build automation workflow with function calling on local LLM? : r/flowise - Reddit, 访问时间为 九月 13, 2025, https://www.reddit.com/r/flowise/comments/1d2yx37/how_to_build_automation_workflow_with_function/