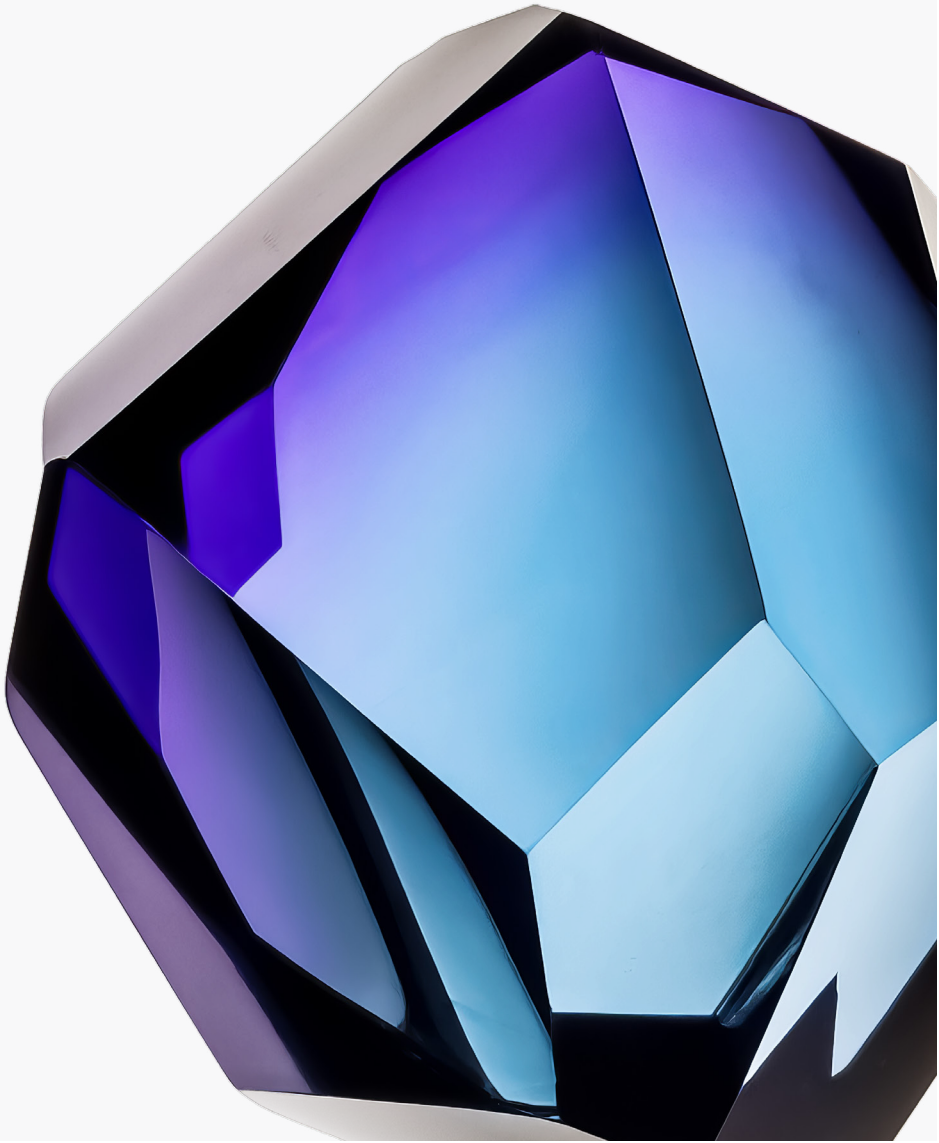
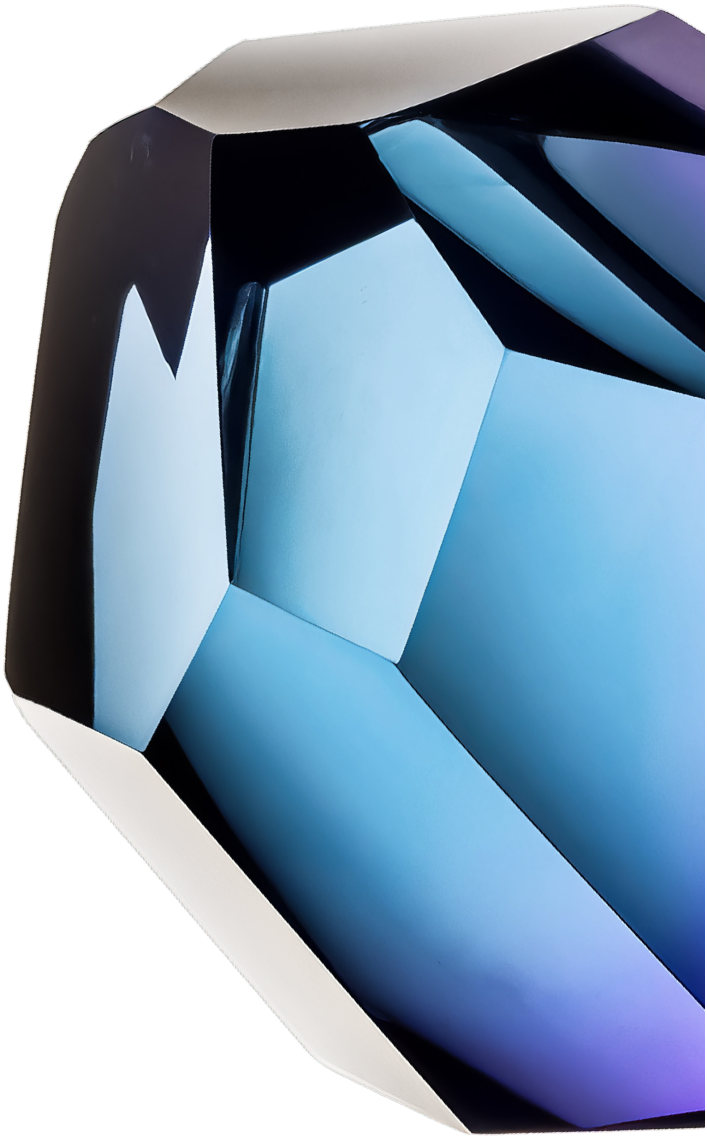
Agents

作者：朱莉娅·维辛格、帕特里克·马洛和弗拉基米尔·武斯科维奇



Google

致谢



审稿人和撰稿人

黄伊凡 薛

艾米丽

奥尔詹·塞尔奇

诺格鲁、塞巴斯

蒂安·里德尔、

萨廷德·巴韦贾、

安东尼奥·古利、

阿南特·纳瓦尔

加里亚

策展人和编辑

安东尼奥·古利、

阿南特·纳瓦尔

加里亚、格雷

斯·莫里森

技术作家

乔伊·海梅克

设计师

迈克尔·兰宁

20242年9月

目录（或内容摘要）

推理、逻辑以及获取外部信息的能力都与生成式AI模型相关联，这种结合引发了Agent的概念。

介绍

人类在混乱的模式识别任务上表现出色。然而，在得出结论之前，他们经常依赖工具——比如书籍、谷歌搜索或计算器——来补充他们的先验知识。就像人类一样，生成式AI模型也可以被训练来使用工具获取实时信息或建议采取现实世界的行动。例如，模型可以利用数据库检索工具来访问特定信息，比如客户的购买历史，从而生成量身定制的购物推荐。或者，基于用户的查询，模型可以发起各种API调用，以向同事发送电子邮件回复或代表您完成金融交易。为此，模型不仅需要访问一系列外部工具，还需要能够以自我指导的方式规划和执行任何任务。这种推理、逻辑和访问外部信息的结合，都与生成式AI模型相关联，这引发了agent的概念，或者说是一个超越生成式AI模型独立能力的程序。本白皮书将更详细地探讨所有这些及相关方面。

20244年9月

什么是Agents？

从最根本的意义上讲，生成式AI agent(智能体)可以被定义为一种应用程序，它通过观察世界并利用其可支配的工具来采取行动，以实现某个目标。智能体具有自主性，可以独立于人类干预而行动，尤其是在被赋予了它们要实现的适当目标或目的时。智能体在实现目标的方法上也可以采取积极主动的态度。即使在没有人类明确指令的情况下，智能体也可以推理出为实现其最终目标而应采取的下一步行动。虽然AI中的智能体概念相当普遍且强大，但本白皮书重点关注的是在发布时生成式AI模型能够构建的特定类型的智能体。

为了理解智能体的内部工作原理，我们首先来介绍驱动智能体行为、动作和决策的基础组件。这些组件的组合可以描述为一种认知架构，通过混合和匹配这些组件，可以实现许多这样的架构。聚焦于核心功能，如图1所示，智能体的认知架构中有三个基本组件。

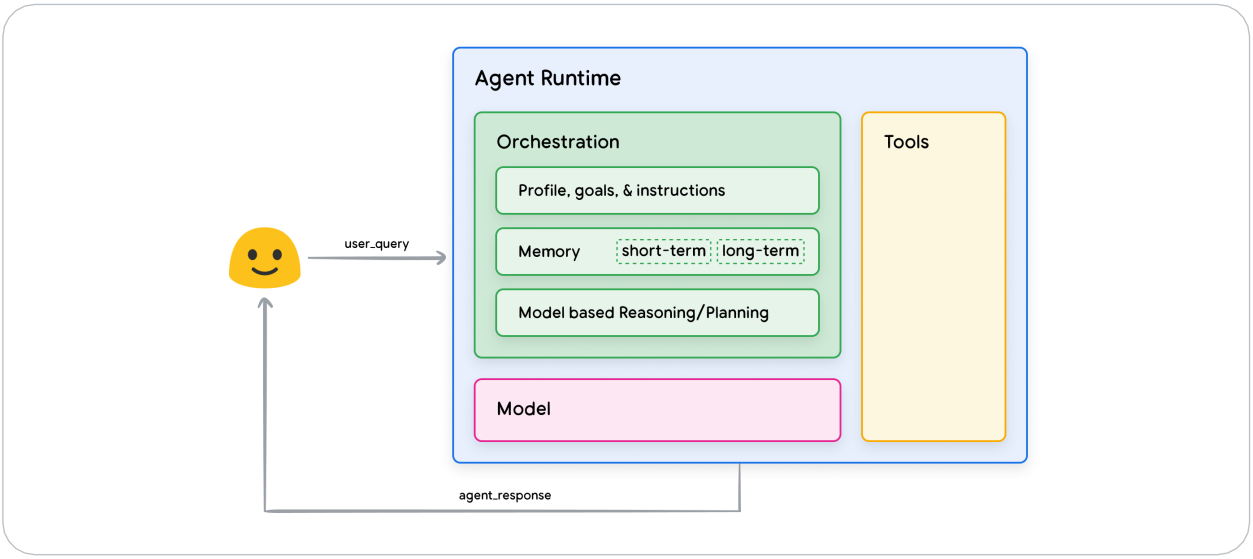


图1. 通用agent架构及其组件

模型

在agent的范围内，模型指的是将作为agent流程的集中决策者使用的语言模型（LM）。agent 使用的模型可以是一个或多个LM，其大小可以是任何规模（小/大），能够遵循基于指令的推理和逻辑框架，如ReAct、思维链（Chain-of-Thought）或思维树（Tree-of-Thoughts）。模型可以是通用的、多模态的，也可以根据特定agent架构的需求进行微调。为了获得最佳的生产效果，您应该选择最适合您所需最终应用的模型，理想情况下，该模型应基于与您计划在认知架构中使用的工具相关的数据签名进行训练。需要注意的是，模型通常不是使用agent的特定配置设置（即工具选择、编排/推理设置）进行训练的。但是，通过提供展示agent能力的示例，包括agnet在各种情境中使用特定工具或推理步骤的实例，可以进一步优化agent任务的模型。

工具

基础模型尽管在文本和图像生成方面表现出色，但仍然受到无法与外部世界交互的制约。*工具*弥合了这一差距，使agents能够与外部数据和服务进行交互，同时解锁了超出基础模型自身能力的更广泛的操作范围。工具可以采取多种形式，复杂程度各不相同，但通常与常见的Web API方法（如GET、POST、PATCH和DELETE）保持一致。例如，工具可以更新数据库中的客户信息或获取天气数据，以影响agents向用户提供的旅行建议。借助工具，agents可以访问和处理现实世界的信息。这使他们能够支持更专业的系统，如检索增强生成（RAG），这大大扩展了agent的能力，超出了基础模型自身所能实现的范围。我们将在下面更详细地讨论工具，但最需要理解的是，工具弥合了agent的内部能力和外部世界之间的差距，解锁了更广泛的可能性。

编排层

编排层描述了一个循环过程，该过程控制着智能体如何获取信息、进行内部推理，并利用该推理来指导其下一步行动或决策。通常，这个循环会持续进行，直到智能体达到其目标或到达某个停止点。编排层的复杂性可能因智能体及其执行的任务而异。一些循环可能只是简单的计算和决策规则，而另一些则可能包含连锁逻辑、涉及额外的机器学习算法，或实现其他概率推理技术。我们将在认知架构部分深入讨论智能体编排层的详细实现。

Agents与模型

为了更清晰地理解Agents和模型之间的区别，请考虑以下情况：

模型（Models）	Agents
知识仅限于他们的训练数据中可获取的内容。	知识通过工具与外部系统的连接得以扩展
基于用户查询进行单一推理/预测。除非模型中明确实现，否则不会进行会话管理 历史记录或连续上下文。（即聊天记录）	管理会话历史（即聊天历史），以便根据用户查询和在编排层做出的决策进行多轮推理/预测。在此上下文中，“一轮”被定义为交互系统与agent之间的交互。（即1个传入事件/查询和1个agents响应）
无原生工具实现。	工具在agent架构中是原生实现的。
未实现原生逻辑层。用户可以以简单问题形式构建提示，或使用推理框架（如CoT、ReAct等）构建复杂提示，以引导模型进行预测。	采用如CoT、ReAct等推理框架，或如LangChain等预构建Agents框架的原生认知架构。

认知架构：智能体如何运作

想象一下，一位厨师在忙碌的厨房中。他的目标是为餐厅的顾客烹制出美味的菜肴，这涉及一系列的规划、执行和调整过程。

- 他们收集信息，比如顾客的点餐内容以及食品柜和冰箱里有哪些食材。
- 他们根据刚刚收集到的信息，进行内部推理，思考可以制作哪些菜肴和风味。
- 他们开始制作这道菜：切菜、混合香料、煎肉。

在流程的每个阶段，厨师都会根据需要进行调整，随着食材的消耗或客户反馈的接收，不断优化他们的计划，并利用之前的结果来确定下一步的行动计划。这种信息收集、计划、执行和调整的循环，描述了厨师为实现目标而采用的独特认知架构。

就像厨师一样，智能体也可以利用认知架构，通过迭代处理信息、做出明智的决策，并根据之前的输出优化后续行动，从而实现最终目标。智能体认知架构的核心是编排层，负责维护记忆、状态、推理和规划。它利用快速发展的提示工程及相关框架来指导推理和规划，使智能体能够更有效地与其环境进行交互并完成任务。在提示工程框架和语言模型任务规划领域的研究正在迅速发展，涌现出各种有前景的方法。虽然这里没有列出所有方法，但以下是本文发表时最受欢迎的几个框架和推理技术：

- **ReAct** 是一个提示工程框架，它为语言模型提供了一种思维过程策略，使其能够根据用户查询进行推理并采取行动，无论是否提供上下文示例。ReAct提示已被证明优于多个当前最佳（SOTA）基线，并提高了人类与LLM的互操作性和信任度。

- 思维链（CoT）是一种提示工程框架，通过中间步骤实现推理能力。CoT有各种子技术，包括自一致性、主动提示和多模态CoT，每种技术都有其优缺点，具体取决于具体应用。
- 思维树（ToT）是一个非常适合探索或战略前瞻任务的提示工程框架。它概括了思维链提示，并允许模型探索各种思维链，这些思维链可作为使用语言模型解决一般问题的中间步骤。

Agents可以使用上述推理技术中的一种，或许多其他技术，为给定的用户请求选择下一个最佳行动。例如，让我们考虑一个被编程为使用*ReAct*框架来为用户查询选择正确行动和工具的agent。事件序列可能大致如下：

1.用户向agent发送查询

2.Agent开始执行ReAct序列

3.Agent向模型发出提示，要求其生成下一个ReAct步骤及其相应的输出：

1 **问题：**用户查询中的输入问题，已随提示提供

2 **思考：**模型对于接下来应该执行的操作的思考

3 **动作：**模型决定接下来应采取的行动

1 这就是可以选择工具的地方

2 例如，动作可以是[航班、搜索、代码、无]中的一种，其中前三个代表模型可以选择的已知工具，最后一个代表“无工具选择”

4 **动作输入：**模型决定向工具提供哪些输入（如果有的话）

5 **观察：**动作/动作输入序列的结果

1 这种想法/行动/动作输入/观察可以根据需要重复*N*次

6 最终**答案：**模型针对原始用户查询提供的**最终答**案

4. ReAct循环结束，并向用户提供最终答案

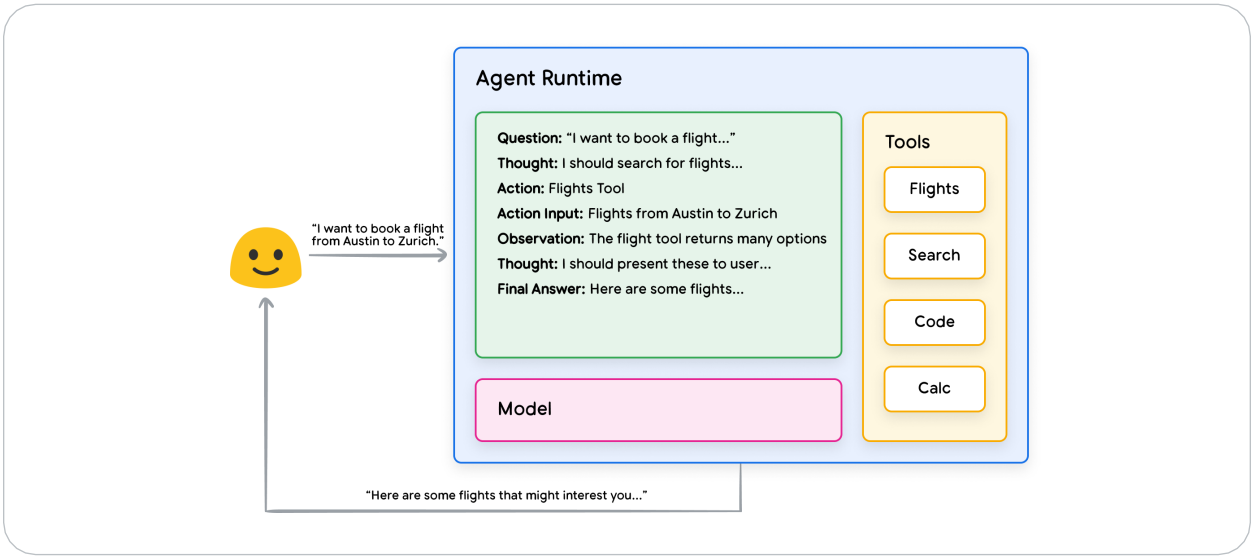


图2:在编排层中采用ReAct推理的示例agent

如图2所示，模型、工具和Agent配置协同工作，根据用户的原始查询为用户提供基于事实的简洁回应。虽然模型可以根据其先验知识猜测答案（即产生幻觉），但它还是选择使用工具（Flights）来搜索实时外部信息。这些额外的信息被提供给模型，使其能够基于真实数据做出更明智的决策，并将这些信息总结后反馈给用户。

总之，agent响应的质量可以直接与模型对这些不同任务的推理和行动能力相关联，包括选择正确工具的能力，以及这些工具的定义有多完善。就像厨师用新鲜食材精心烹制菜肴并关注顾客反馈一样，agents依靠合理的推理和可靠的信息来提供最佳结果。在下一节中，我们将深入探讨agents与新鲜数据连接的各种方式。

工具：我们通往外部世界的钥匙

虽然语言模型在处理信息方面表现出色，但它们缺乏直接感知和影响现实世界的能力。这限制了它们在需要与外部系统或数据交互的场景中的实用性。这意味着，从某种意义上说，语言模型的能力仅限于它从训练数据中学到的知识。然而，无论我们向模型投入多少数据，它们仍然缺乏与外部世界交互的基本能力。那么，我们如何才能使我们的模型具备与外部系统进行实时、上下文感知的交互能力呢？函数、扩展、数据存储和插件都是为模型提供这种关键能力的方法。

虽然它们有着各种各样的名称，但正是工具在我们的基础模型与外部世界之间搭建起了桥梁。这种与外部系统和数据的连接，使得我们的智能体能够执行更多种类的任务，并且更加准确可靠。例如，工具可以让智能体调整智能家居设置、更新日历、从数据库中提取用户信息，或者根据特定指令发送电子邮件。

截至本文发布之日，谷歌模型能够与之交互的主要工具类型有三种：扩展、函数和数据存储。通过为agents配备这些工具，我们释放了它们巨大的潜力，使它们不仅能够理解世界，还能对世界采取行动，为无数新的应用和可能性打开了大门。

扩展

理解扩展最简单的方式就是将其视为以标准化的方式弥合API和agent之间的鸿沟，使agent能够无缝地执行API，而无需考虑其底层实现。假设你构建了一个旨在帮助用户预订航班的agent。你知道你想使用Google Flights API来检索航班信息，但你不确定如何让你的agent调用这个API端点。

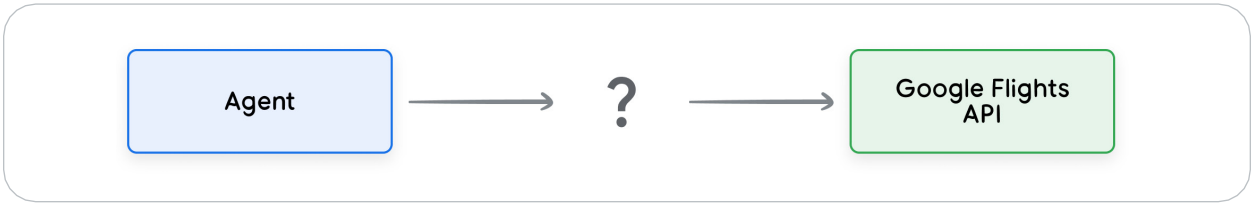


图3. agents如何与外部API进行交互?

一种方法可能是实现自定义代码，该代码将接收传入的用户查询，解析查询以获取相关信息，然后进行API调用。例如，在航班预订用例中，用户可能会说“我想预订*从奥斯汀飞往苏黎世*的航班”。在这种情况下，我们的自定义代码解决方案需要在尝试进行API调用之前，从用户查询中提取“奥斯汀”和“苏黎世”作为相关实体。但是，如果用户说“我想预订飞*往苏黎世*的航班”并且从未提供出发城市，会发生什么情况呢？没有所需数据，API调用将失败，并且需要实现更多代码来捕捉像这样的边缘和极端情况。这种方法不可扩展，并且在任何超出已实现的自定义代码的情况中都可能轻易出现问题。

一种更具弹性的方法是通过使用扩展。扩展通过以下方式弥合了agent和API之间的差距：

1.通过示例教agent如何使用API端点。

2.教导agent需要哪些参数或自变量来成功调用API端点。

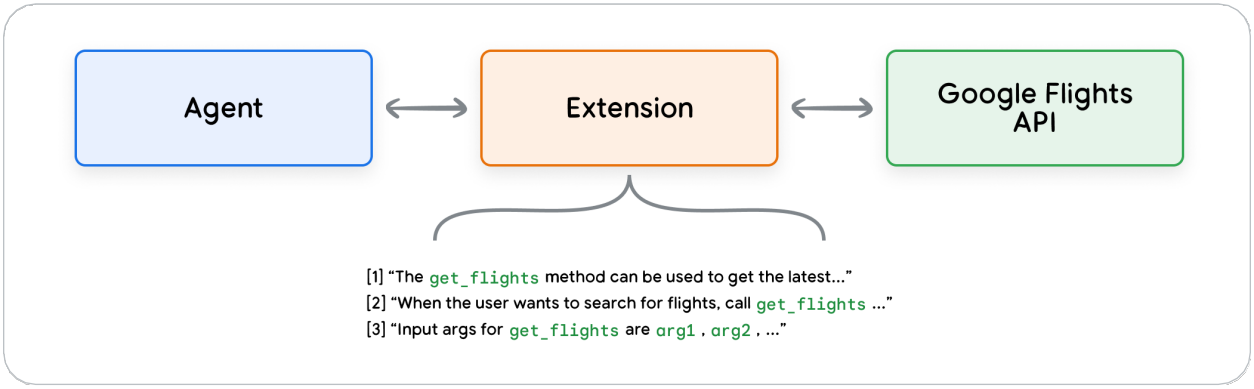


图4. 扩展将agents与外部API连接起来

扩展可以独立于agent进行开发，但应作为agent配置的一部分提供。agent在运行时使用模型和示例来决定哪个扩展（如果有的话）适合解决用户的查询。这突显了扩展的一个关键优势，即它们*内置的示例类型*，使agent能够动态地为任务选择最合适的扩展。

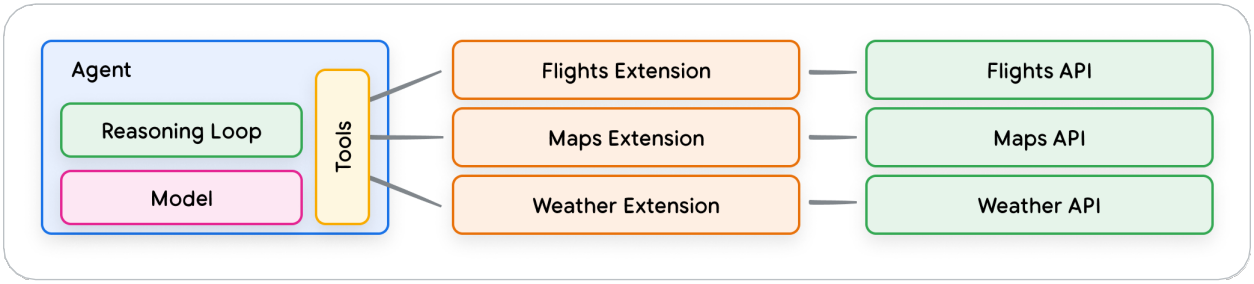


图5. Agents、扩展和API之间的1对多关系

以同样的方式思考，就像软件开发人员在解决用户问题并寻找解决方案时决定使用哪些API端点一样。如果用户想要预订航班，开发人员可能会使用Google Flights API。如果用户想知道离他们位置最近的咖啡店在哪里，开发人员可能会使用Google Maps API。同样地，agent/模型堆栈使用一组已知的扩展来决定哪一个最适合用户的查询。如果你想看到扩展的实际应用，你可以通过转到“设置”>“扩展”，然后启用任何你想测试的扩展，在Gemini应用程序上尝试它们。例如，你可以启用Google Flights扩展，然后向Gemini询问“显示从奥斯汀到苏黎世，下周五起飞的航班。”

示例扩展

为了简化扩展的使用，Google提供了一些现成的扩展，可以快速集成到您的项目中，并使用最少的配置。例如，代码片段1中的代码解释器扩展允许您从自然语言描述生成并运行Python代码。

Python

```
import vertexai
import pprint


PROJECT_ID = "YOUR_PROJECT_ID"
REGION = "us-central1"

vertexai.init(project=PROJECT_ID, location=REGION)


from vertexai.preview.extensions import Extension


extension_code_interpreter = Extension.from_hub("code_interpreter")
CODE_QUERY = """Write a python method to invert a binary tree in O(n) time."""


response = extension_code_interpreter.execute(
  operation_id = "generate_and_execute",
  operation_params = {"query": CODE_QUERY}
  )


print("Generated Code:")
pprint.pprint({response['generated_code']})


# The above snippet will generate the following code.
```
```
Generated Code:
class TreeNode:
  def __init__(self, val=0, left=None, right=None):
    self.val = val
    self.left = left
    self.right = right
```

Continues next page...

Python

```python
def invert_binary_tree(root):
    """
    Inverts a binary tree.
    Args:
        root: The root of the binary tree.
    Returns:
        The root of the inverted binary tree.
    """


    if not root:
        return None


    # Swap the left and right children recursively
    root.left, root.right =
invert_binary_tree(root.right), invert_binary_tree(root.left)


    return root


# Example usage:
# Construct a sample binary tree
root = TreeNode(4)
root.left = TreeNode(2)
root.right = TreeNode(7)
root.left.left = TreeNode(1)
root.left.right = TreeNode(3)
root.right.left = TreeNode(6)
root.right.right = TreeNode(9)


# Invert the binary tree
inverted_root = invert_binary_tree(root)
```

代码片段1：代码解释器扩展可以生成并运行Python代码

总之，扩展为agents提供了一种以无数种方式感知、交互和影响外部世界的方式。这些扩展的选择和调用由示例的使用来指导，所有示例都被定义为扩展配置的一部分。

功能

在软件工程领域，函数被定义为自包含的代码模块，它们完成特定的任务，并可根据需要重复使用。软件开发人员在编写程序时，通常会创建许多函数来完成各种任务。他们还将定义何时调用函数a而不是函数b的逻辑，以及预期的输入和输出。

在agents的世界中，函数的工作方式非常相似，但我们可以将软件开发人员替换为模型。模型可以接收一组已知的函数，并根据其规范决定何时使用每个函数以及函数需要哪些参数。函数与扩展在几个方面有所不同，最显著的是：

1.模型输出一个函数及其参数，但并不进行实时的API调用。

2.函数在*客户端*执行，而扩展在*agent端*执行。

再次以我们的谷歌航班为例，一个简单的函数设置可能如图7中的示例所示。

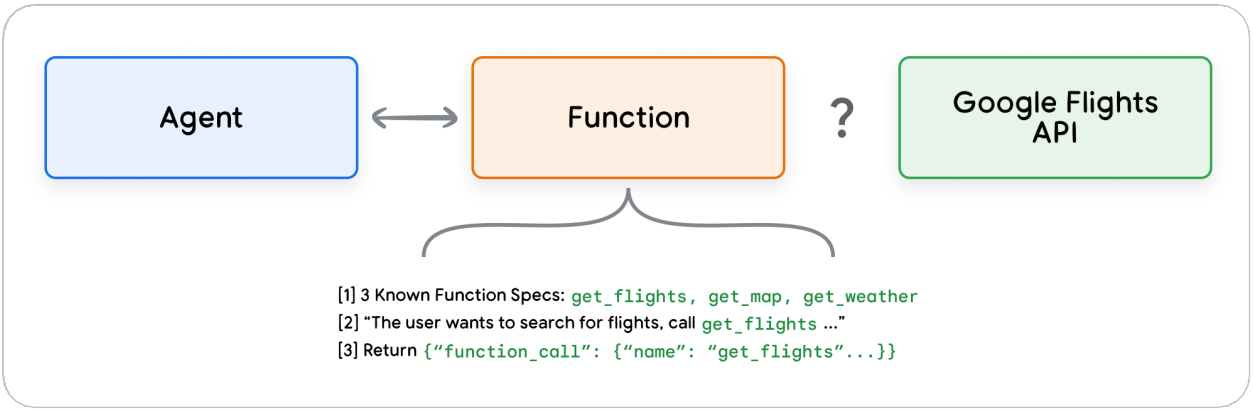


图7. 函数如何与外部API进行交互？

请注意，这里的主要区别在于，无论是函数还是agent，都不会直接与Google Flights API进行交互。那么，API调用实际上是如何发生的呢？

通过函数，调用实际API端点的逻辑和执行从agent中卸载，并返回到客户端应用程序，如下图8和图9所示。这为开发者提供了对应用程序中数据流的更精细的控制。开发者选择使用函数而非扩展的原因有很多，但一些常见的用例包括：

- API调用需要在应用程序堆栈的另一层进行，位于直接agent架构流之外（例如，中间件系统、前端框架等）

- 阻止agent直接调用API的安全或身份验证限制（例如，API未向互联网公开，或agent基础设施无法访问API）

- 阻止agent实时进行API调用的时间或操作顺序约束。（即批量操作、人工介入审核等）

- 需要对agent无法执行的API响应应用额外的数据转换逻辑。例如，考虑一个API端点，它没有提供用于限制返回结果数量的过滤机制。在客户端使用函数为开发者提供了额外的机会来进行这些转换。
- 开发人员希望在agent开发上进行迭代，而无需为API端点部署额外的基础设施（即函数调用可以像API的“存根”一样工作）

如图8所示，虽然这两种方法在内部架构上的差异并不明显，但额外的控制和与外部基础设施的解耦依赖使得函数调用成为开发人员的一个有吸引力的选择。

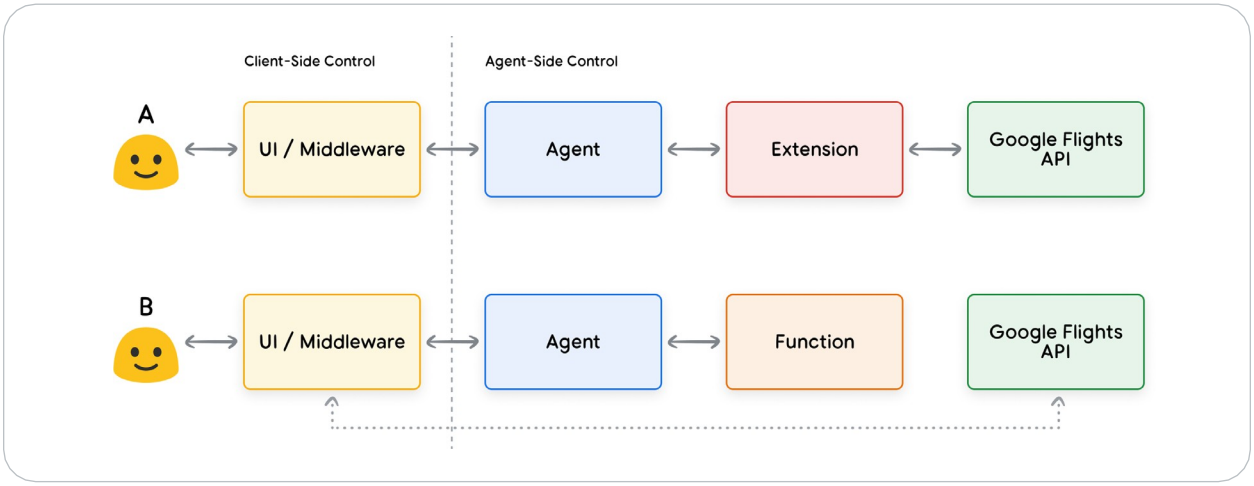


图8. 划分客户端与agent端对扩展和函数调用的控制

使用案例

模型可以用来调用函数，以便为最终用户处理复杂的客户端执行流程，其中agent开发人员可能不希望语言模型来管理API执行（就像扩展的情况一样）。让我们考虑以下示例，其中一个agent正在被训练为旅行礼宾，与想要预订度假旅行的用户进行交互。目标是让agent生成一个城市列表，我们可以在我们的中间件应用程序中使用该列表来下载用户旅行计划的图像、数据等。用户可能会说类似以下的话：

我想和家人一起去滑雪旅行，但我不确定去哪里。

在向模型发出典型提示时，输出可能如下所示：当然，以下是一份你可以考虑

的家庭滑雪旅行城市清单：

- 美国科罗拉多州克雷斯特德比特
- 加拿大不列颠哥伦比亚省惠斯勒
- 瑞士采尔马特

虽然上述输出包含我们所需的数据（城市名称），但其格式并不适合解析。借助函数调用，我们可以训练模型以结构化风格（如JSON）格式化此输出，这样更便于其他系统进行解析。在给定相同的用户输入提示的情况下，函数调用的示例JSON输出可能如代码段5所示。

```
Unset

function_call {
  name: "display_cities"
  args: {
    "cities": ["Crested Butte", "Whistler", "Zermatt"],
    "preferences": "skiing"
    }
}
```

代码段5. 用于显示城市列表和用户偏好的示例函数调用有效负载

这个JSON有效负载由模型生成，然后发送到我们的客户端服务器，以执行我们想要对它进行的任何操作。在这个特定情况下，我们将调用Google Places API来获取模型提供的城市并查找图像，然后将它们作为格式化的丰富内容提供给我们的用户。考虑图9中的这个序列图，它一步步详细地展示了上述交互。

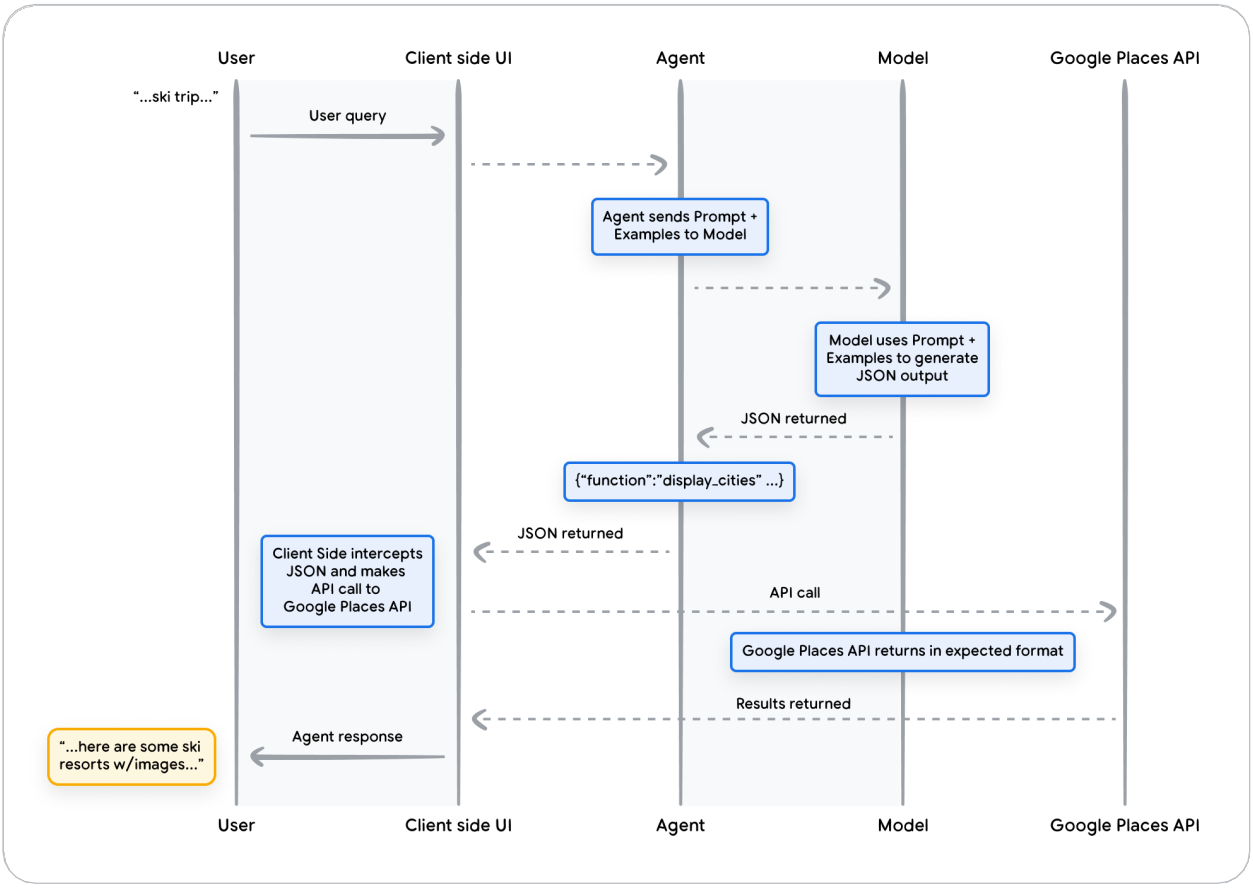


图9. 展示函数调用生命周期的序列图

图9中的示例结果是，模型被用来“填补空白”，提供客户端UI调用Google Places API所需的参数。客户端UI使用模型在返回的Function中提供的参数来管理实际的API调用。这只是Function Calling的一个用例，但还有许多其他场景需要考虑，比如：

- 你想要一个语言模型来建议一个可以在代码中使用的函数，但你不想在代码中包含凭据。因为函数调用并不会实际运行函数，所以无需在包含函数信息的代码中包含凭据。

- 你正在运行可能需要几秒钟以上的异步操作。这些场景非常适合使用函数调用，因为函数调用本身就是一种异步操作。
- 你想在一个与产生函数调用及其参数的系统不同的设备上运行函数。

关于函数，需要记住的一个关键点是，它们旨在为开发者提供更多的控制权，不仅是对API调用的执行，还包括整个应用程序中的整个数据流。在图9中的示例中，开发者选择不向agent返回API信息，因为这对于agent可能采取的未来行动来说并不重要。然而，基于应用程序的架构，将外部API调用数据返回给agent以影响未来的推理、逻辑和行动选择可能是有意义的。最终，由应用程序开发者来选择适合特定应用程序的正确做法。

函数示例代码

为了从我们的滑雪度假场景中获得上述产出，让我们逐步构建每个组件，以便在我们的gemini-1.5-flash-001模型中实现这一目标。

首先，我们将把display_cities函数定义为一个简单的Python方法。

Python

```python
def display_cities(cities: list[str], preferences: Optional[str] = None):
    """Provides a list of cities based on the user's search query and preferences.

    Args:
        preferences (str): The user's preferences for the search, like skiing,
        beach, restaurants, bbq, etc.
        cities (list[str]): The list of cities being recommended to the user.

    Returns:
        list[str]: The list of cities being recommended to the user.
    """

    return cities
```

代码段6. 显示城市列表的Python函数示例。

接下来，我们将实例化我们的模型，构建工具，然后将用户的查询和工具传递给模型。执行下面的代码将得到代码片段底部所示的输出。

Python

```python
from vertexai.generative_models import GenerativeModel, Tool, FunctionDeclaration

model = GenerativeModel("gemini-1.5-flash-001")

display_cities_function = FunctionDeclaration.from_func(display_cities)
tool = Tool(function_declarations=[display_cities_function])

message = "I'd like to take a ski trip with my family but I'm not sure where
to go."

res = model.generate_content(message, tools=[tool])

print(f"Function Name: {res.candidates[0].content.parts[0].function_call.name}")
print(f"Function Args: {res.candidates[0].content.parts[0].function_call.args}")

> Function Name: display_cities
> Function Args: {'preferences': 'skiing', 'cities': ['Aspen', 'Vail',
'Park City']}
```

片段7. 构建一个工具，将用户查询发送给模型，并允许进行函数调用

总之，函数提供了一个直观的框架，使应用程序开发人员能够精细地控制数据流和系统执行，同时有效地利用agent/模型进行关键输入生成。开发人员可以根据具体的应用程序架构需求，有选择地决定是否通过返回外部数据来保持agent的“参与”，或者忽略它。

数据存储

将语言模型想象成一个庞大的藏书库，其中包含其训练数据。但与不断扩充新书的图书馆不同，这个模型库保持不变，只保留最初训练时所获得的知识。这带来了一个挑战，因为现实世界的知识在不断演变。数据存储通过提供对更动态和最新信息的访问，并确保模型的响应始终基于事实和相关性，从而克服了这一局限性。

考虑一个常见场景，即开发人员可能需要向模型提供少量的额外数据，这些数据可能以电子表格或PDF文件的形式呈现。

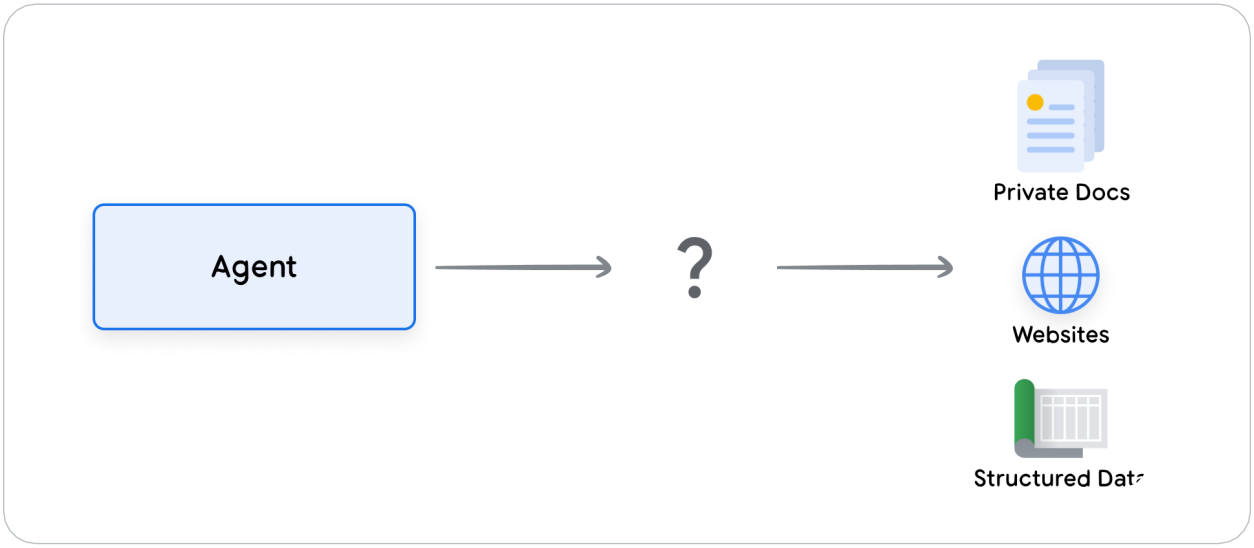


图10. Agent如何与结构化和非结构化数据交互？

*数据存储*允许开发人员以原始格式向agent提供额外数据，从而无需进行耗时的数据转换、模型重新训练或微调。数据存储将传入的文档转换为一组*向量数据库嵌入*，agent可以使用这些*嵌入*来提取所需信息，以辅助其下一步行动或对用户的响应。

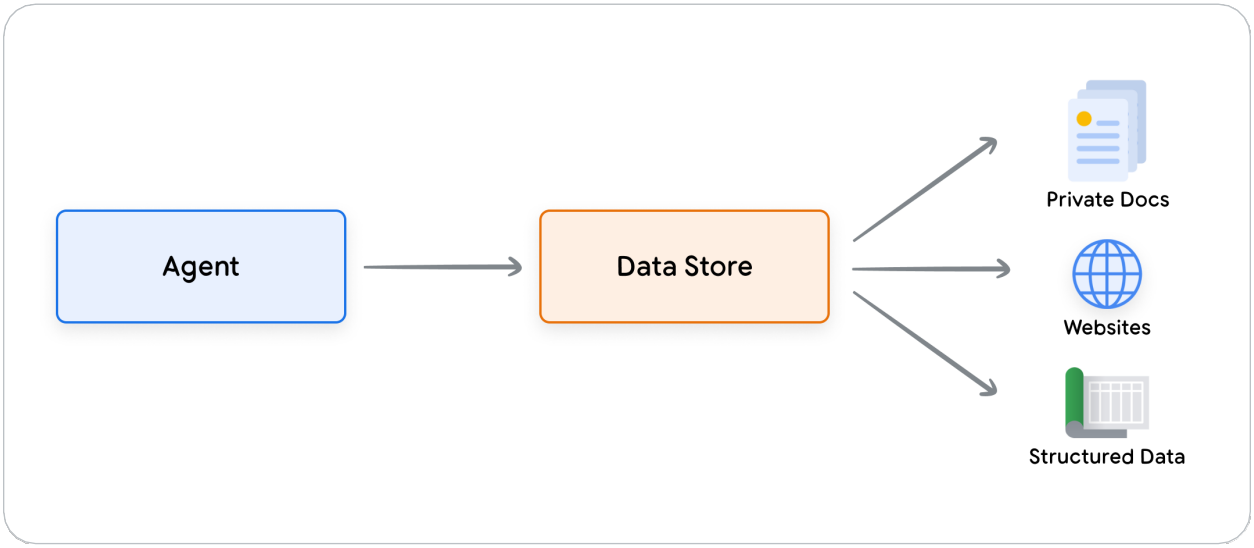


图11. 数据存储将agents与各种类型的新实时数据源连接起来。

实施与应用

在生成式AI agents的上下文中，数据存储通常被实现为开发者希望agent在运行时能够访问的*向量*数据库。虽然我们不会在此深入探讨向量数据库，但需要理解的关键点是，它们以向量嵌入的形式存储数据，这是一种高维向量或所提供数据的一种数学表示。近年来，数据存储在语言模型中使用的一个最典型的例子是检索增强（Retrieval Augmented）的实现

基于生成对抗网络（RAG）的应用。这些应用旨在通过让模型访问各种格式的数据，如：

- 网站内容
- PDF、Word文档、CSV、电子表格等格式的结构化数据。
- HTML、PDF、TXT等格式的非结构化数据。

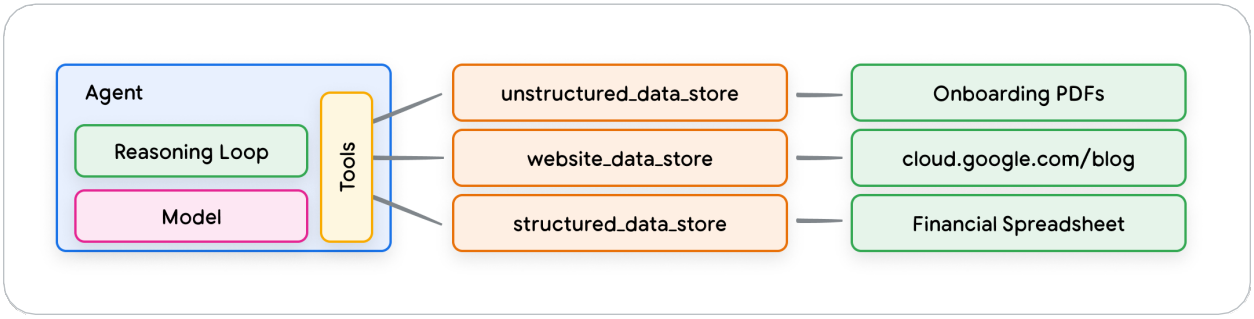


图12展示了agents与数据存储之间的1对多关系，这种关系可以代表各种类型的预索引数据

每个用户请求和agent响应循环的底层流程通常如图13所示。

1.用户查询被发送给嵌入模型，以生成该查询的嵌入向量

2.然后，使用像SCaNN这样的匹配算法，将查询嵌入与向量数据库的内容进行匹配

3.从向量数据库中检索匹配的内容，并以文本格式将其发送回给agent

4.agent接收到用户查询和检索到的内容后，会制定相应的响应或操作

5 向用户发送最终回复

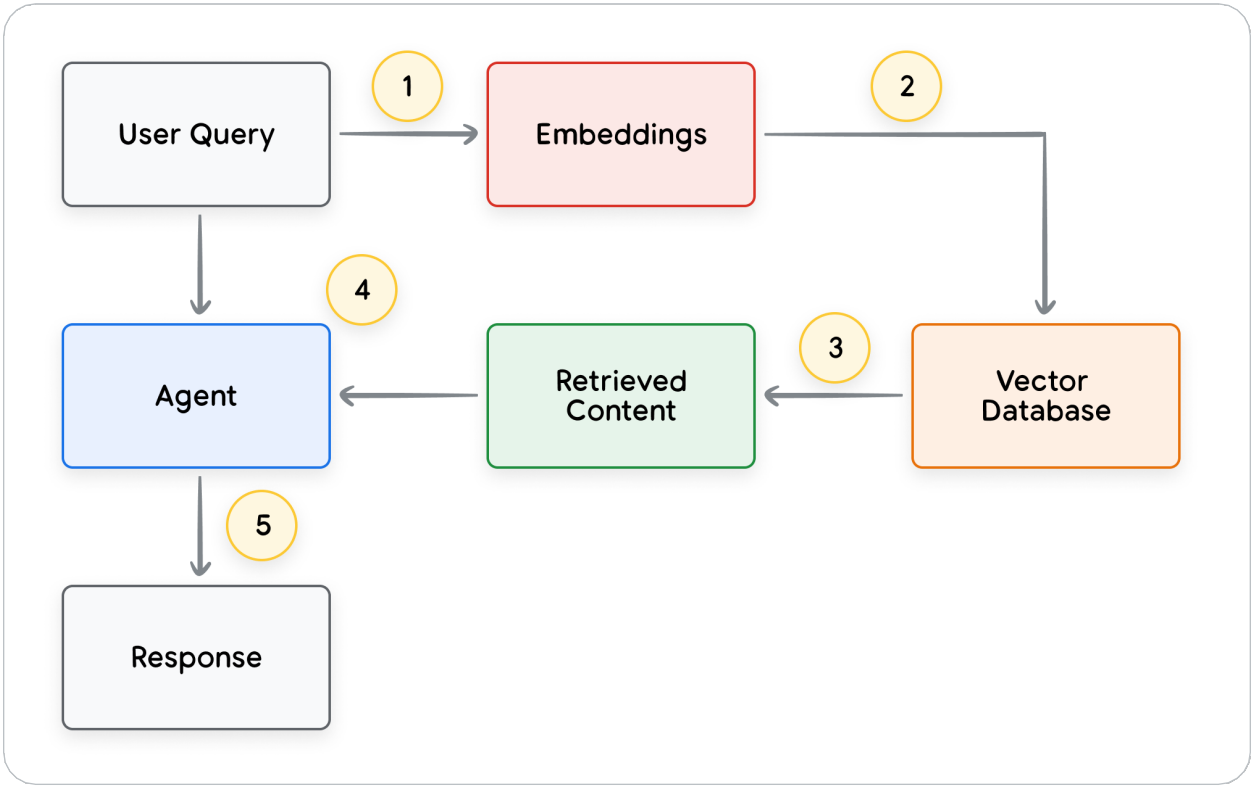


图13. 基于RAG的应用中用户请求和agent响应的生命周期

最终结果是一个应用程序，它允许agent通过向量搜索将用户的查询与已知的数据存储进行匹配，检索原始内容，并将其提供给编排层和模型进行进一步处理。下一步可能是向用户提供最终答案，或者执行额外的向量搜索以进一步细化结果。

图14展示了一个与实现*RAG*（*结合ReAct推理/规划*）的智能体进行交互的示例。

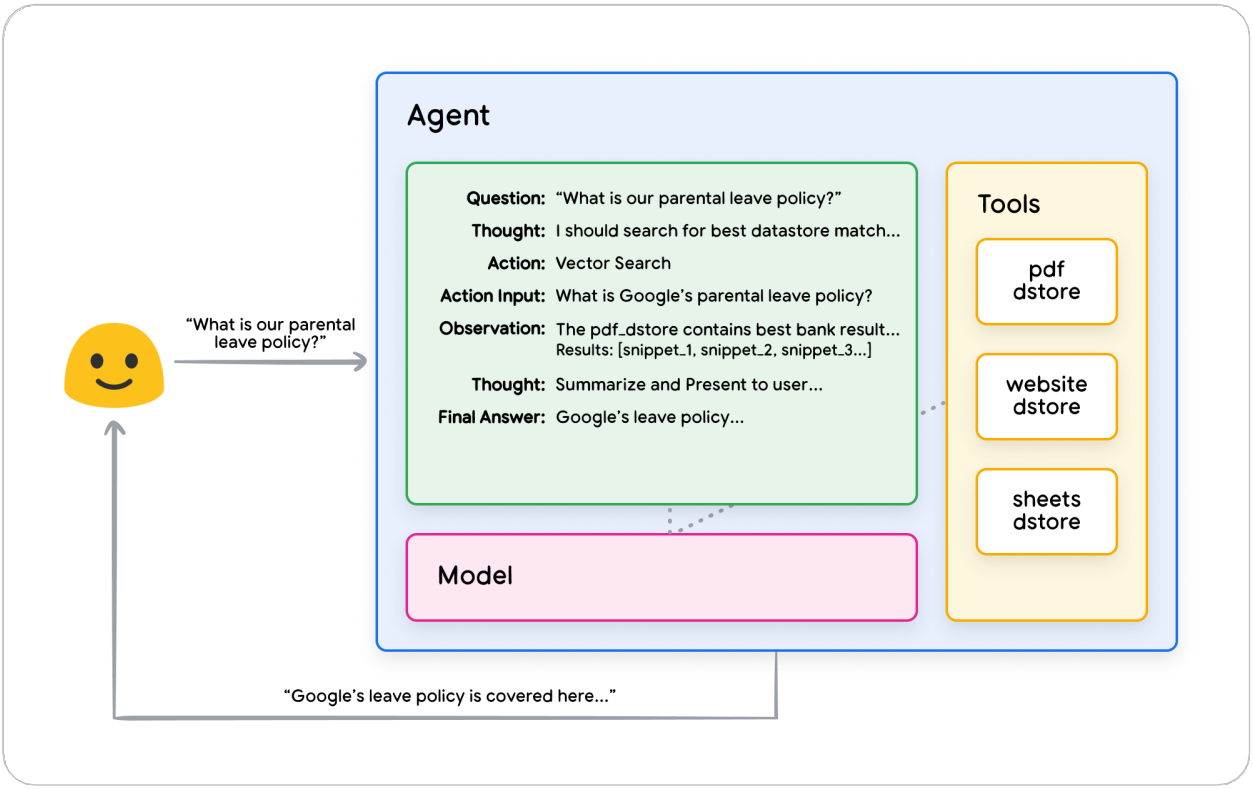


图14. 基于RAG的示例应用，结合ReAct推理/规划

工具回顾

总的来说，扩展、函数和数据存储构成了Agent在运行时可以使用的几种不同工具类型。每种工具都有其特定用途，并且可以根据agent开发者的意愿，选择一起使用或单独使用。

	扩展	函数调用	数据存储
执行	agent端执行	客户端执行	agent端执行
使用案例	• 开发者希望agent能够控制与API端点的交互 • 在利用原生预构建扩展（例如Vefiex搜索、代码解释器等）时非常有用 • 多跳规划和API调用（即下一个agent动作取决于前一个动作/API调用的输出）	• 安全或身份验证限制阻止agent直接调用API • 时间限制或操作顺序限制阻止agent实时进行API调用。（即批处理操作、人工介入审核等） • 未向互联网公开的API，或谷歌系统无法访问	开发者希望使用以下任何数据类型来实现检索增强生成（RAG）： • 来自预先索引的域名和URL的网站内容 • PDF、Word文档、CSV、电子表格等格式的结构化数据。 • 关系型/非关系型数据库 • HTML、PDF、TXT等格式的非结构化数据。

通过针对性学习提升模型性能

有效使用模型的一个关键方面是，它们在生成输出时能够选择正确的工具，尤其是在生产环境中大规模使用工具时。虽然一般训练有助于模型培养这种技能，但现实世界的场景往往需要超出训练数据范围的知识。想象一下，这就像基本烹饪技能与掌握特定菜系之间的区别。两者都需要基础的烹饪知识，但后者需要针对性学习，以获得更细致的结果。

为了帮助模型获取这种特定类型的知识，存在几种方法：

- **上下文学习：**这种方法在推理时提供了一个泛化模型，该模型配备了提示、工具和少量示例，使其能够“即时学习如何以及何时将这些工具用于特定任务”。ReAct框架就是自然语言领域中这种方法的典型例子。
- **基于**检索的**上下**文**学习：**这种技术通过从外部存储器中检索最相关的信息、工具和相关示例，动态地为模型提示填充内容。例如，Vefiex AI扩展中的“示例存储”或前面提到的基于RAG的数据存储架构。
- **基于微调**的**学习：**这种方法涉及在推理之前使用包含特定示例的更大数据集来训练模型。这有助于模型在接收任何用户查询之前，理解何时以及如何应用确认工具。

为了对每种有针对性的学习方法提供更多见解，让我们再次回到烹饪的类比。

- 想象一下，一位厨师从顾客那里收到了一份特定的食谱（即提示）、几种关键食材（即相关工具）以及一些示例菜肴（即少量示例）。基于这些有限的信息以及厨师对烹饪的一般知识，他们需要即兴发挥，准备出最符合食谱和顾客喜好的菜肴。这就是*情境学习*。

- 现在，让我们想象一下，我们的厨师身处一个备有充足食材的厨房（外部数据存储），里面装满了各种食材和烹饪书（示例和工具）。厨师现在能够从厨房中动态选择食材和烹饪书，并更好地进行搭配
  根据顾客的食谱和偏好，厨师可以更明智、更精细地制作菜肴，同时利用*现有*知识*和新知识*。这是一种*基于检索的上下文学习*。

- 最后，让我们想象一下，我们让厨师回到学校学习一种或多种新的菜肴（在更大的特定示例数据集上进行预训练）。这使得厨师能够以更深入的理解来处理未来未见过的客户食谱。如果我们想让厨师在特定菜肴（知识领域）上表现出色，这种方法是完美的。这就是*基于微调*的*学习*。

这些方法在速度、成本和延迟方面各有优缺点。然而，通过在agent框架中结合这些技术，我们可以利用各种优势并最大限度地减少其弱点，从而实现更稳健、适应性更强的解决方案。

使用LangChain快速构建Agents

为了提供一个实际可执行的agent操作示例，我们将使用LangChain和LangGraph库构建一个快速原型。这些流行的开源库允许用户通过将逻辑、推理和工具调用的序列“链接”起来，来构建客户agents，以回答用户的查询。我们将使用我们的gemini-1.5-flash-001模型和一些简单工具来回答用户的多阶段查询，如代码片段8所示。

我们使用的工具是SerpAPI（针对谷歌搜索）和谷歌地点API。在执行完 Snippet 8 中的程序后，您可以在 Snippet 9 中看到示例输出。

Python

```python
from langgraph.prebuilt import create_react_agent
from langchain_core.tools import tool
from langchain_community.utilities import SerpAPIWrapper
from langchain_community.tools import GooglePlacesTool

os.environ["SERPAPI_API_KEY"] = "XXXXX"
os.environ["GPLACES_API_KEY"] = "XXXXX"

@tool
def search(query: str):
    """Use the SerpAPI to run a Google Search."""
    search = SerpAPIWrapper()
    return search.run(query)

@tool
def places(query: str):
    """Use the Google Places API to run a Google Places Query."""
    places = GooglePlacesTool()
    return places.run(query)

model = ChatVertexAI(model="gemini-1.5-flash-001")
tools = [search, places]

query = "Who did the Texas Longhorns play in football last week? What is the
address of the other team's stadium?"

agent = create_react_agent(model, tools)
input = {"messages": [("human", query)]}

for s in agent.stream(input, stream_mode="values"):
    message = s["messages"][-1]
    if isinstance(message, tuple):
        print(message)
    else:
        message.pretty_print()
```

片段8. 基于LangChain和LangGraph的agent示例及其工具

Unset

```
================================ Human Message =================================
Who did the Texas Longhorns play in football last week? What is the address
of the other team's stadium?
================================== Ai Message ==================================
Tool Calls: search
Args:
   query: Texas Longhorns football schedule
================================= Tool Message =================================
Name: search
{...Results: "NCAA Division I Football, Georgia, Date..."}
================================== Ai Message ==================================
The Texas Longhorns played the Georgia Bulldogs last week.
Tool Calls: places
Args:
   query: Georgia Bulldogs stadium
================================= Tool Message =================================
Name: places

{...Sanford Stadium Address: 100 Sanford...}
================================== Ai Message ==================================
The address of the Georgia Bulldogs stadium is 100 Sanford Dr, Athens, GA
30602, USA.
```

片段9. 片段8中程序的输出

虽然这是一个相对简单的agent示例，但它展示了模型、编排和工具这些基础组件如何协同工作以实现特定目标。在最后一节中，我们将探讨这些组件如何在谷歌规模的托管产品（如Vefiex AI agents和生成式Playbook）中协同工作。

配备VefiexAI agents的生产应用

虽然本白皮书探讨了agents的核心组件，但构建生产级应用程序需要将它们与用户界面、评估框架和持续改进机制等其他工具集成。谷歌的*Vekex AI*平台通过提供一个完全托管的环境，涵盖了前面提到的所有基本元素，简化了这一过程。使用*自然语言界面*，开发人员可以快速定义其agents的关键要素——目标、任务指令、工具、任务委托的子agents和示例，从而轻松构建所需的系统行为。此外，该平台还附带了一组开发工具，可用于测试、评估、测量agent性能、调试和提高开发agents的整体质量。这使得开发人员能够专注于构建和完善他们的agents，而基础设施、部署和维护的复杂性则由平台本身管理。

在图15中，我们提供了一个基于Vefiex AI平台构建的agent的示例架构，该平台使用了各种功能，如Vefiex Agent Builder、Vefiex Extensions、Vefiex Function Calling和Vefiex Example Store等。该架构包含了许多生产就绪应用程序所需的各个组件。

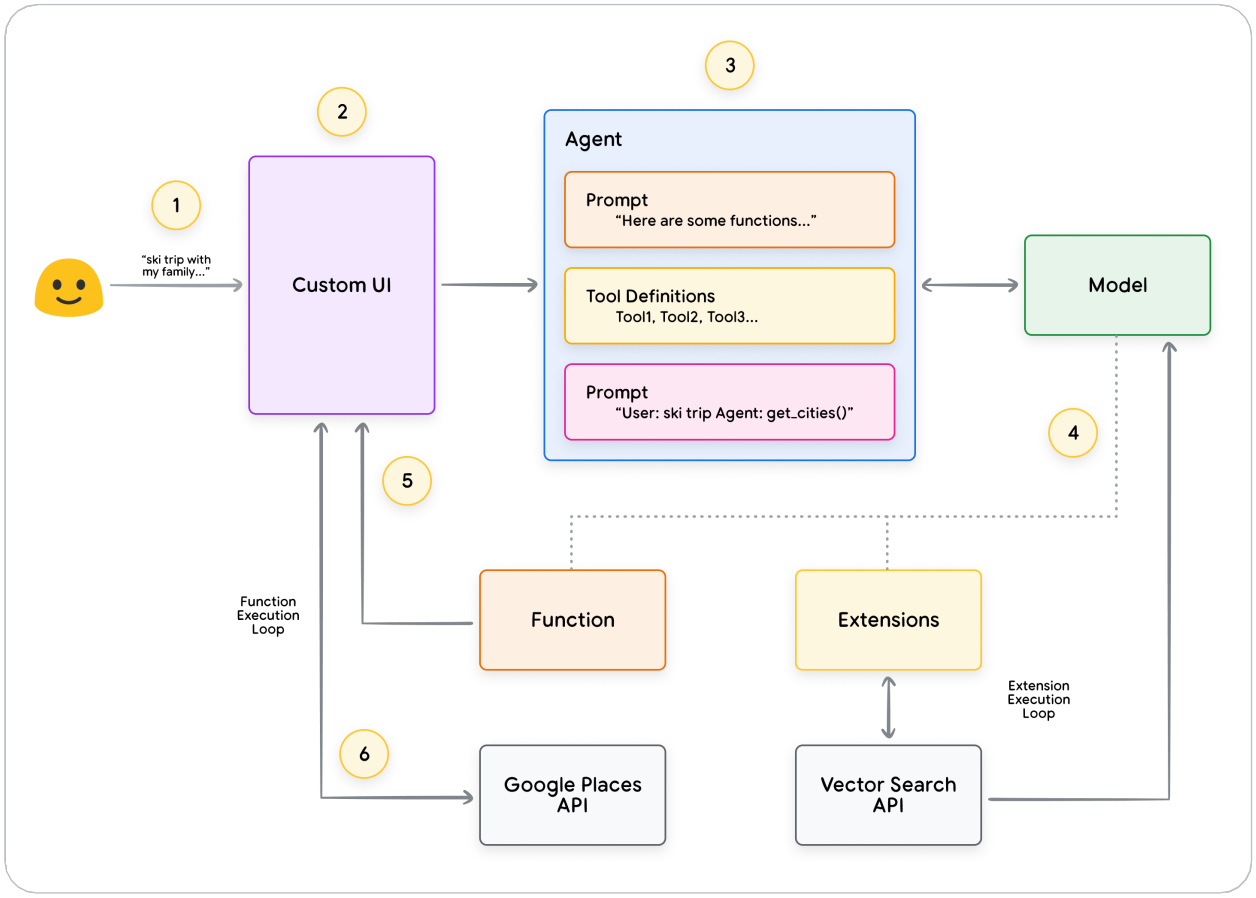


图15. 基于Vefiex AI平台构建的端到端agent架构示例

你可以从我们的官方文档中尝试这个预构建agent架构的示例。

总结

在本白皮书中，我们讨论了生成式AI Agents的基本构建块、它们的组成以及以认知架构的形式实现它们的有效方法。本白皮书的一些关键要点包括：

1. Agents通过利用工具访问实时信息、建议真实世界的操作以及自主计划和执行复杂任务来扩展语言模型的功能。agents可以利用一个或多个语言模型来决定何时以及如何转换状态，并使用外部工具来完成任何数量的复杂任务，这些任务对于模型本身来说是困难或不可能完成的。

2. Agents操作的核心是编排层，这是一种认知架构，用于构建推理、规划、决策并指导其行动。各种推理技术，如ReAct、Chain of Thought和Tree of Thoughts，为编排层提供了一个框架，以接收信息、执行内部推理并生成明智的决策或响应。

3. 工具，如扩展、函数和数据存储，是agents进入外部世界的钥匙，使他们能够与外部系统交互，并访问培训数据之外的知识。扩展提供了agents和外部API之间的桥梁，使API调用的执行和实时信息的检索成为可能。函数通过分工为开发人员提供了更精细的控制，允许agents生成可以在客户端执行的函数参数。数据存储为agents提供了对结构化或非结构化数据的访问，从而支持数据驱动的应用程序。

Agents的未来充满了令人兴奋的进步，而我们才刚刚开始探索其无限可能。随着工具的日益精进和推理能力的不断增强，agent将有能力解决日益复杂的问题。
此外，“agent链”的战略方法将继续获得发展势头。

通过结合专业化的agents——每个agents在特定领域或任务中表现出色——我们可以创建一种“agent专长混合(mixture of agent expert)”的方法，能够在各个行业和问题领域中实现卓越的成果。

重要的是要记住，构建复杂的agent架构需要采用迭代的方法。实验和优化是针对特定业务案例和组织需求寻找解决方案的关键。由于支撑其架构的基础模型具有生成性，因此没有两个agents是相同的。然而，通过利用这些基础组件的各自优势，我们可以创建有影响力的应用程序，扩展语言模型的能力，并推动实现真正的价值。

尾注

1 Shafran, I., Cao, Y. 等人，2022，《ReAct：在语言模型中融合推理与行动》。可访问：hflps://arxiv.org/abs/2210.03629

2 魏佳、王晓等，2023年，《思维链提示引发大型语言模型的推理》。可查阅：https://arxiv.org/pdf/2201.11903.pdf。

3 王X等，2022年，《自我一致性提高了语言模型中的思维链推理能力》。可查阅：https://arxiv.org/abs/2203.11171。

4 刁思等，2023，《大型语言模型的思维链主动提示》。可查阅：hflps://arxiv.org/pdf/2302.12246.pdf.

5 张H等，2023，《语言模型中的多模态思维链推理》。可查阅：hflps://arxiv.org/abs/2302.00923.

6 姚思等，2023，《思维树：利用大型语言模型进行深思熟虑的问题解决》。可查阅：hflps://arxiv.org/abs/2305.10601.

7 龙，X.，2023，《大型语言模型引导的思维链》。可访问：hflps://arxiv.org/abs/2305.08291.

8 谷歌。谷歌Gemini应用程序。可在以下网址获取：hflp://gemini.google.com。

9 Swagger。OpenAPI规范。可在以下网址获取：https://swagger.io/specification/。

10 谢，M.，2022年，《情境学习是如何工作的？理解其与传统监督学习差异的框架》。可访问：https://ai.stanford.edu/blog/understanding-incontext/。

11 谷歌研究。ScaNN（可扩展最近邻）。可在以下网址获取：hflps://github.com/google-research/google-research/tree/master/scann.

12 LangChain。LangChain。可在以下网址获取：https://python.langchain.com/v0.2/docs/introduction/。