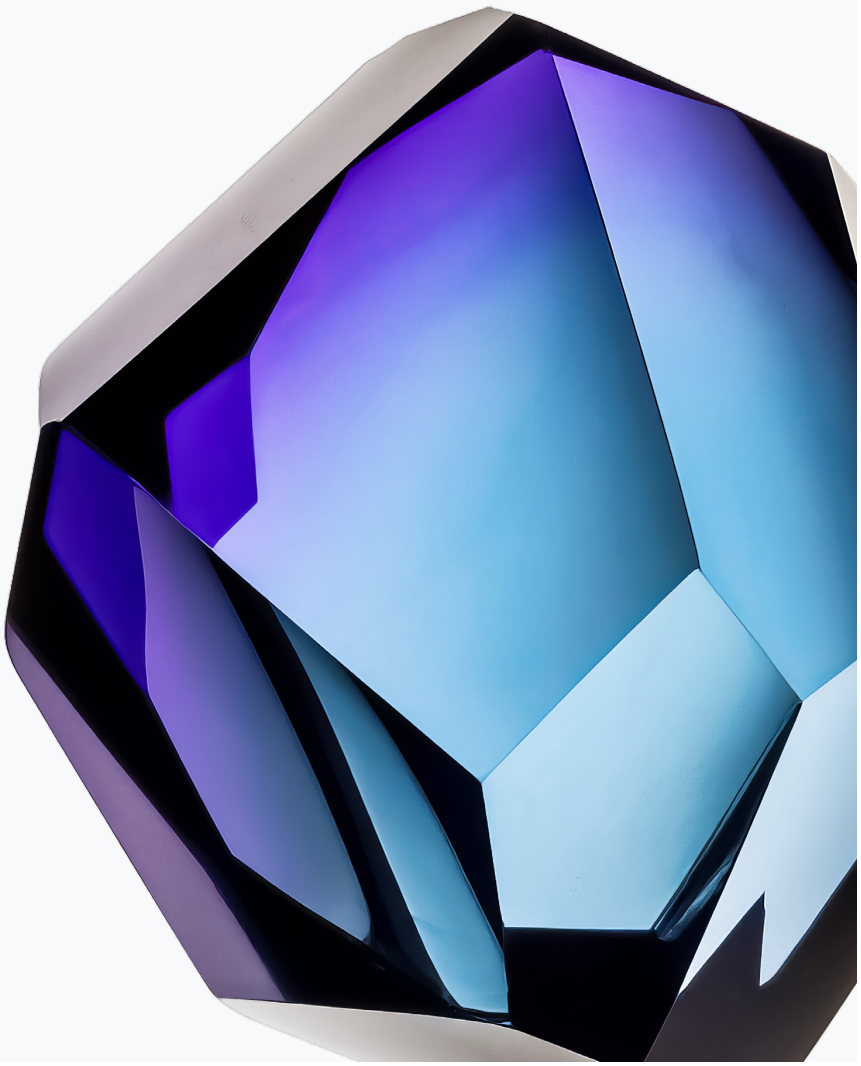
智能体白皮书

作者: Julia Wiesinger, Patrick Marlow and Vladimir Vuskovic



Google

致谢 (Acknowledgements)

审阅者与贡献者 (Reviewers and Contributors)

- Evan Huang
- Emily Xue
- Olcan Sercinoglu
- Sebastian Riedel
- Satinder Baveja
- Antonio Gulli
- Anant Nawalgaria

策划与编辑 (Curators and Editors)

- Antonio Gulli
- Anant Nawalgaria
- Grace Mollison

技术作者 (Technical Writer)

- Joey Haymaker

设计师 (Designer)

- Michael Lanning

目录 (Table of contents)

这种将推理、逻辑以及访问外部信息的能力都连接到一个生成式 AI 模型上的做法，引出了智能体 (agent) 的概念。

1、引言 (Introduction)

人类在处理复杂的模式识别任务方面具有很强的能力。然而，通常人们会借助工具，如书籍、Google 搜索或计算器等，来弥补自己已有知识的不足。

就像人类一样，生成式 AI 模型也可以被训练使用工具来访问实时信息或建议现实世界的操作。例如，模型可以利用数据库检索工具来访问特定信息，如客户的购买历史，从而生成针对客户量身定制的购物推荐。或者，根据用户的查询，模型可以进行各种 API 调用，以向同事发送电子邮件回复或代表你完成金融交易。

要做到这一点，模型不仅需要访问一套外部工具，还需要具备以自主方式规划和执行任何任务的能力。这种推理、逻辑以及访问外部信息的能力，所有这些都连接到一个生成式 AI 模型上，从而引出了智能体 (agent) 的概念，或者说是一个超越「生成式 AI 模型独立能力范围」的程序。本白皮书将更详细地探讨所有这些方面及相关内容。

2、什么是智能体？ (What is an agent?)

在其最基本的形式中，生成式 AI 智能体可以被定义为一个应用程序，它试图通过观察世界并使用其可支配的工具对其采取行动来达成一个目标。智能体是自主的，可以独立于人类干预而行动，特别是当它们被赋予了旨在实现的恰当目标或目的时。智能体在其实现目标的方法上也可以是主动的。即使在没有来自人类的明确指令集的情况下，智能体也可以推理「下一步应该做什么」来实现其最终目标。虽然 AI 中智能体的概念相当普遍且强大，但这篇侧重于「生成式 AI 模型」在本白皮书发布时能够构建的特定类型的智能体。

为了理解智能体的内部运作，让我们首先介绍驱动智能体行为（behavior）、行动（action）和决策（decision making）的基础组件。这些组件的组合可以被描述为一个认知架构 (cognitive architecture)，并且可以通过混合和匹配这些组件来实现许多这样的架构。关注核心功能，一个智能体的认知架构中有三个基本组件，如图 1 所示。

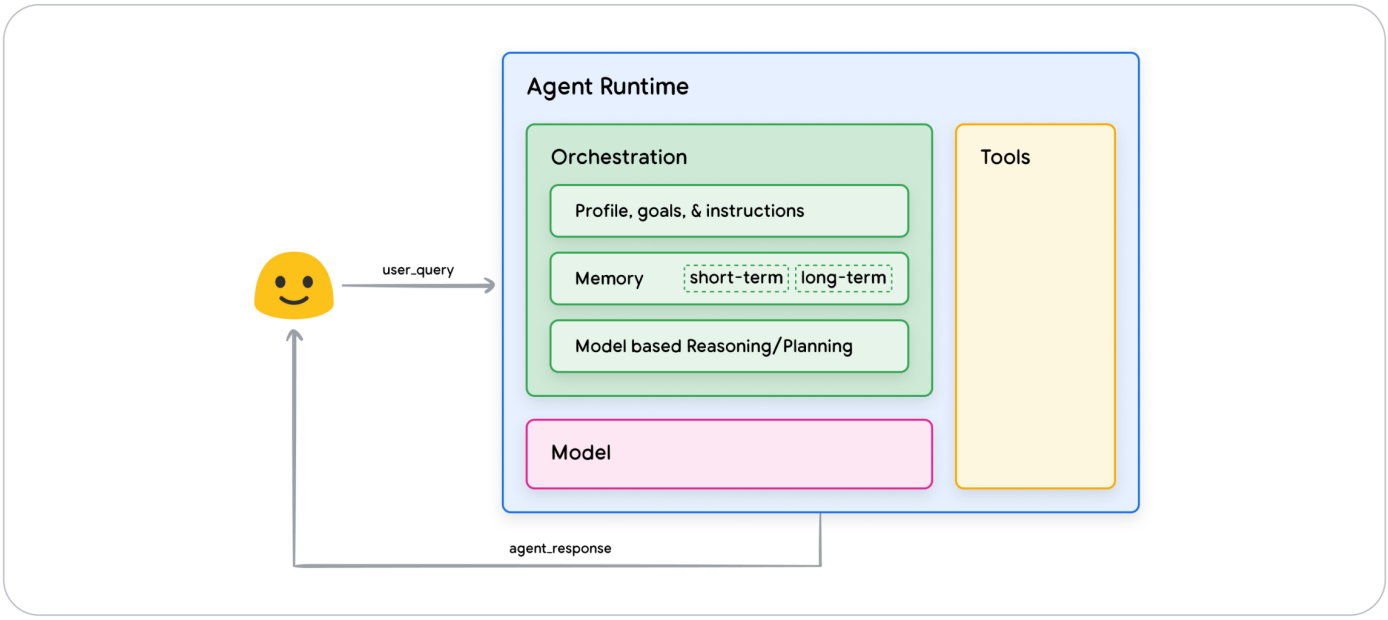


图 1. 通用智能体架构和组件

2.1 模型 (The model)

在智能体的范畴内，模型指的是将被用作智能体流程的中心决策者的语言模型 (LM)。智能体使用的模型可以是一个或多个任何规模（小/大）的 LM，它们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、多模态的或根据特定智能体架构的需求进行微调的。为了获得最佳的生产结果，你应该利用最适合「你期望的最终应用」的模型，并且理想情况下，该模型已经在与你计划在认知架构中使用的工具相关的数据特征上进行了训练。需要注意的是，模型通常没有使用智能体的特定配置设置（即工具选择、编排/推理设置）进行训练。然而，可以通过提供展示智能体能力的示例来进一步为智能体的任务优化模型，包括智能体在各种上下文中使用特定工具或推理步骤的实例。

2.2 工具 (The tools)

基础模型尽管在文本和图像生成方面表现出色，但仍然**受限于它们无法与外部世界互动**。工具弥合了这一差距，使智能体能够与外部数据和服务互动，同时解锁了超越底层模型本身能力的更广泛行动范围。工具可以采取多种形式，并具有不同程度的复杂性，但通常与常见的 Web API 方法（如 GET、POST、PATCH 和 DELETE）保持一致。例如，一个工具可以更新数据库中的客户信息，或者获取天气数据来影响智能体向用户提供的旅行建议。通过工具，智能体可以访问和处理现实世界的信息。这使它们能够支持更专门化的系统，如检索增强生成 (Retrieval Augmented Generation, RAG)，这显著扩展了智能体的能力，超越了基础模型自身所能达到的水平。我们将在下文中更详细地讨论工具，但**最重要的是要理解工具弥合了智能体内部能力与外部世界之间的鸿沟，从而解锁了更广泛的可能性**。

2.3 编排层 (The orchestration layer)

编排层描述了一个周期性过程，它管理智能体如何接收信息、执行一些内部推理，并使用该推理来指导其下一步行动或决策。通常，这个循环将持续进行，直到智能体达到其目标或一个停止点（stopping point）。编排层的复杂性可能根据智能体及其执行的任务而有很大差异。一些循环可以是带有决策规则的简单计算，而另一些可能包含链式逻辑，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更详细地讨论智能体编排层的具体实现。

2.4 智能体 vs. 模型 (Agents vs. models)

为了更清晰地理解智能体和模型之间的区别，请参考下表：

模型 (Models)	智能体 (Agents)
知识仅限于其训练数据中可用的内容。	知识通过与外部系统的连接（通过工具）得到扩展。
基于用户查询的单次推理/预测。除非为模型明确实现，否则没有会话历史或连续上下文的管理。(即聊天历史)	管理会话历史（即聊天历史），以允许基于用户查询和在编排层中做出的决策进行多轮推理/预测。在此上下文中，“轮次 (turn)” 定义为交互系统与智能体之间的一次交互。(即 1 个传入事件/查询和 1 个智能体响应)
没有原生工具实现。	工具在智能体架构中原生实现。
没有实现原生逻辑层。用户可以将提示构造成简单问题，或使用推理框架 (CoT, ReAct 等) 构成复杂提示来指导模型进行预测。	原生认知架构使用像 CoT、ReAct 这样的推理框架，或其他像 LangChain 这样的预构建智能体框架。

2.5 认知架构：智能体如何运作（Cognitive architectures: How agents operate）

想象一下繁忙厨房里的厨师。他们的目标是为餐厅顾客制作美味的菜肴，这涉及规划、执行和调整的某种循环。

- 他们收集信息，比如顾客的订单以及储藏室和冰箱里有哪些食材。
- 他们根据刚收集到的信息，进行一些内部推理，思考可以制作哪些菜肴和风味组合。
- 他们采取行动制作菜肴：切菜、混合香料、煎肉。

在过程的每个阶段，厨师都会根据需要进行调整，随着食材耗尽或收到顾客反馈而完善他们的计划，并使用先前结果的集合来确定下一步的行动计划。这种信息接收、规划、执行和调整的循环描述了厨师为达到目标所采用的独特认知架构。

就像厨师一样，智能体可以使用认知架构，通过迭代处理信息、做出明智决策以及根据先前输出来优化后续行动来达到其最终目标。智能体认知架构的核心在于编排层，它负责维护记忆、状态、推理和规划。它利用快速发展的提示工程领域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互动并完成任务。语言模型的提示工程框架和任务规划领域的研究正在迅速发展，产生了各种有前途的方法。虽然不是详尽无遗的列表，但以下是本出版物发布时可用的一些最流行的框架和推理技术：

- **ReAct**，一个提示工程框架，为语言模型提供了一种思维过程策略，用以对用户查询进行推理 (Reason) 和采取行动 (Act)，无论有无上下文示例。ReAct 提示已被证明优于几种 SOTA 基线，并提高了 LLM 的人类互操作性和可信赖性。
- **思维链 (Chain-of-Thought, CoT)**，一个提示工程框架，通过中间步骤启用推理能力。CoT 有多种子技术，包括自洽性 (self-consistency)、主动提示 (active-prompt) 和多模态 CoT，每种技术根据具体应用各有优缺点。
- **思维树 (Tree-of-thoughts, ToT)**，一个提示工程框架，非常适合探索或战略性前瞻任务。它泛化了思维链提示，并允许模型探索各种作为通用问题解决中间步骤的思维链。

智能体可以利用上述推理技术之一，或许多其他技术，来为给定的用户请求选择最佳的下一步行动。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智能体。事件序列可能如下所示：

1. 用户向智能体发送查询
2. 智能体开始 ReAct 序列
3. 智能体向模型提供一个提示，要求它生成下一个 ReAct 步骤之一及其对应的输出：
   a. **问题 (Question):** 来自用户查询的输入问题，随提示提供
   b. **思考 (Thought):** 模型关于下一步应该做什么的想法
   c. **行动 (Action):** 模型关于下一步采取什么行动的决定
      i. 这是工具能选择行动的集合
      ii. 例如，一个行动可以是 [Flights, Search, Code, None] 中的一个，其中前 3 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”
   d. **行动输入 (Action input):** 模型决定向工具提供哪些输入（如果有）
   e. **观察 (Observation):** 行动/行动输入序列的结果
      i. 这个思考/行动/行动输入/观察可以根据需要重复 N 次
   f. **最终答案 (Final answer):** 模型提供给原始用户查询的最终答案
4. ReAct 循环结束，最终答案被返回给用户

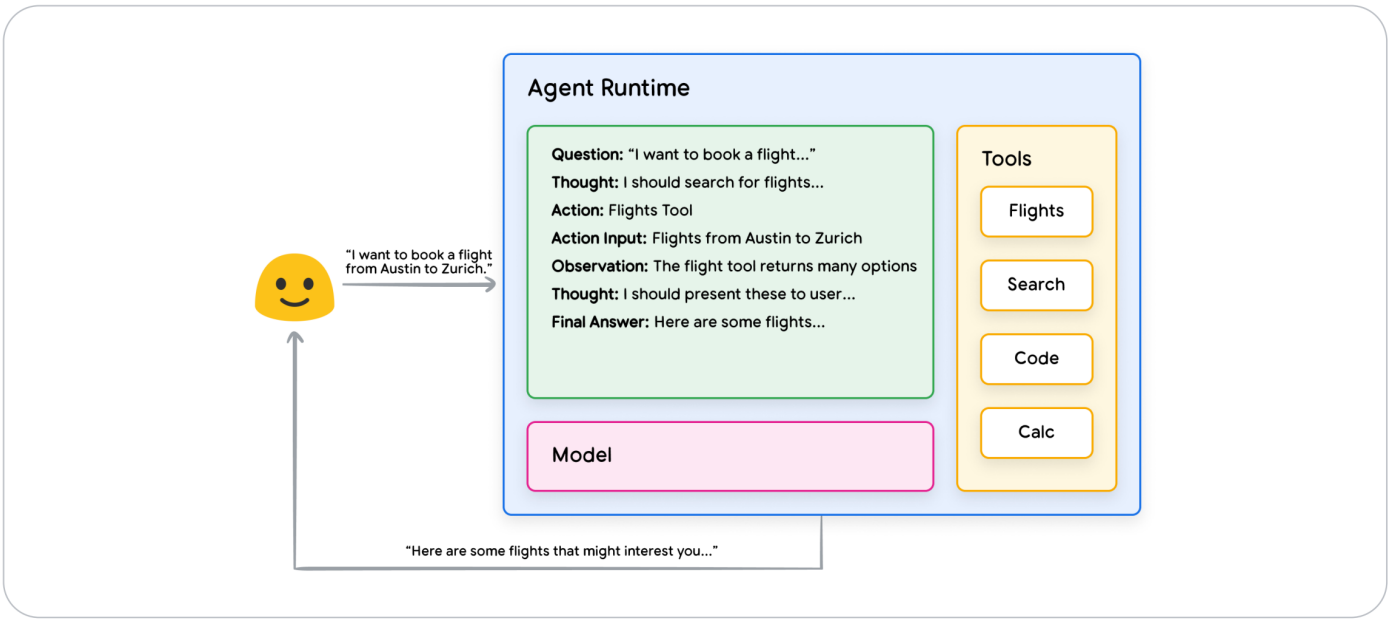


图 2. 具有 ReAct 推理的智能体在编排层中的示例

如图 2 所示，模型、工具和智能体配置协同工作，根据用户的原始查询向用户提供一个有根据的、简洁的响应。虽然模型可以根据其先验知识猜测（幻觉）出一个答案，但它却使用了一个工具 (Flights) 来搜索实时的外部信息。这些额外的信息被提供给模型，使其能够基于真实的、事实性的数据做出更明智的决策，并将这些信息总结后反馈给用户。

总而言之，智能体响应的质量直接关系到模型对这些不同任务进行推理和行动的能力，包括选择正确工具的能力，以及该工具被定义得有多好。就像厨师用新鲜食材精心制作菜肴并关注顾客反馈一样，智能体依赖于合理的推理和可靠的信息来提供最佳结果。在下一节中，我们将深入探讨智能体连接新鲜数据的各种方式。

3、工具：我们通往外部世界的钥匙 (Tools：Our keys to the outside world)

虽然语言模型擅长处理信息，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情况下的实用性。这意味着，在某种意义上，语言模型的优劣取决于它从训练数据中学到的东西。但无论我们向模型输入多少数据，它们仍然**缺乏与外部世界互动的基本能力**。

那么，我们如何赋予我们的**模型与外部系统进行实时、上下文感知交互的能力**呢？**函数 (Functions)、扩展 (Extensions)、数据存储 (Data Stores) 和插件 (Plugins) 都是为模型提供这种关键能力的方式**。

虽然它们有许多名称，但**工具是在我们的基础模型和外部世界之间创建链接的东西。这种与外部系统和数据的链接使我们的智能体能够执行更广泛的任务**，并且具有更高的准确性和可靠性。例如，工具可以使智能体能够调整智能家居设置、更新日历、从数据库中获取用户信息，或根据一组特定指令发送电子邮件。

截至本出版物发布之日，Google 模型能够与之交互的主要有三种工具类型：扩展 (Extensions)、函数 (Functions) 和数据存储 (Data Stores)。**通过为智能体配备工具，我们解锁了巨大的潜力，使它们不仅能够理解世界，还能对其采取行动，为无数新的应用和可能性打开了大门。**

3.1 扩展 (Extensions)

理解扩展最简单的方法是将其视为一种以标准化方式弥合 API 和智能体之间空隙的桥梁，允许智能体无缝执行 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智能体。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智能体调用这个 API 端点。

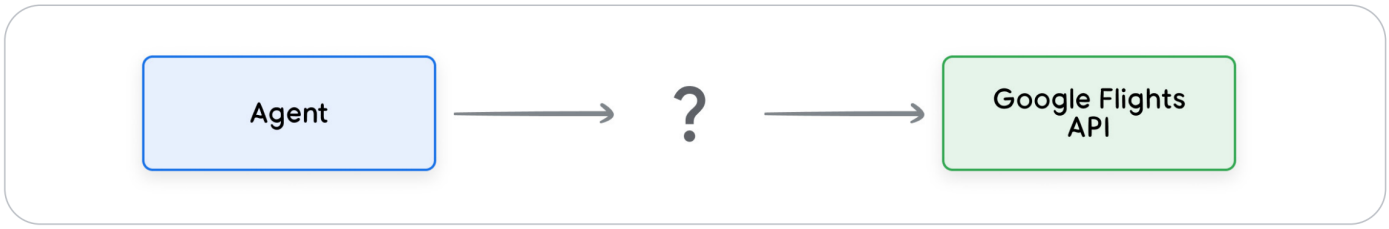


图 3. 智能体如何与外部 API 交互?

一种方法是实现自定义代码，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「我想预订一张从奥斯汀到苏黎世的航班。」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「我想预订一张去苏黎世的航班」并且从未提供出发城市会怎样？API 调用将在没有所需数据的情况下失败，并且需要实现更多代码来捕获此类边缘和角落情况。这种方法不可扩展，并且在任何超出已实现自定义代码范围的情况下都很容易崩溃。

一个更具弹性的方法是使用扩展 (Extension)。扩展通过以下方式弥合了智能体和 API 之间的空隙：

1. 通过示例教智能体如何使用 API 端点。
2. 教智能体成功调用 API 端点需要哪些参数。

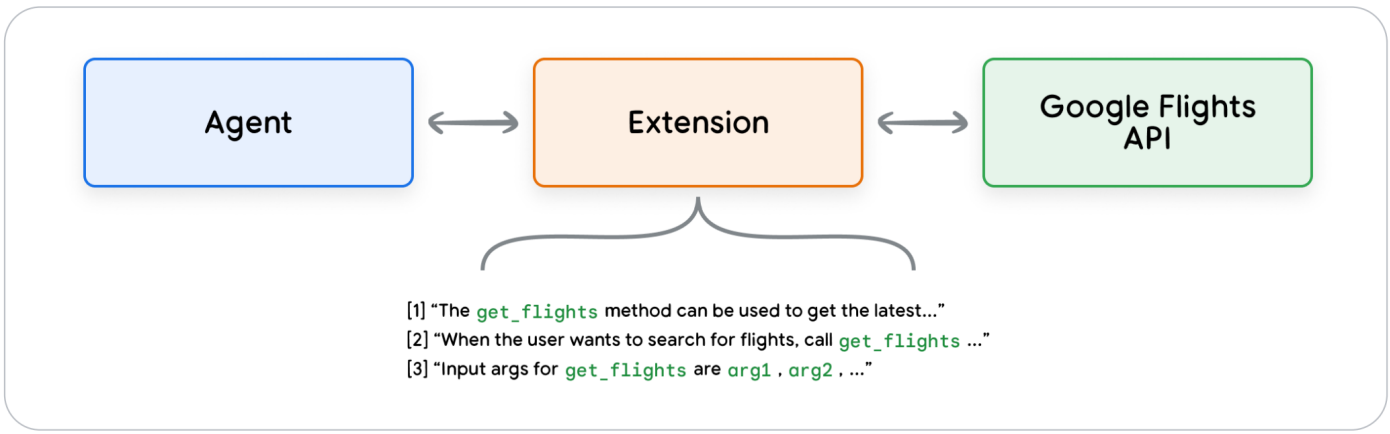


图 4. 扩展连接智能体与外部 API

扩展可以独立于智能体制作，但应作为智能体配置的一部分提供。智能体在运行时使用模型和示例来决定哪个扩展（如果有）适合解决用户的查询。这突显了扩展的一个关键优势，即其内置的示例类型，允许智能体动态选择最适合任务的扩展。

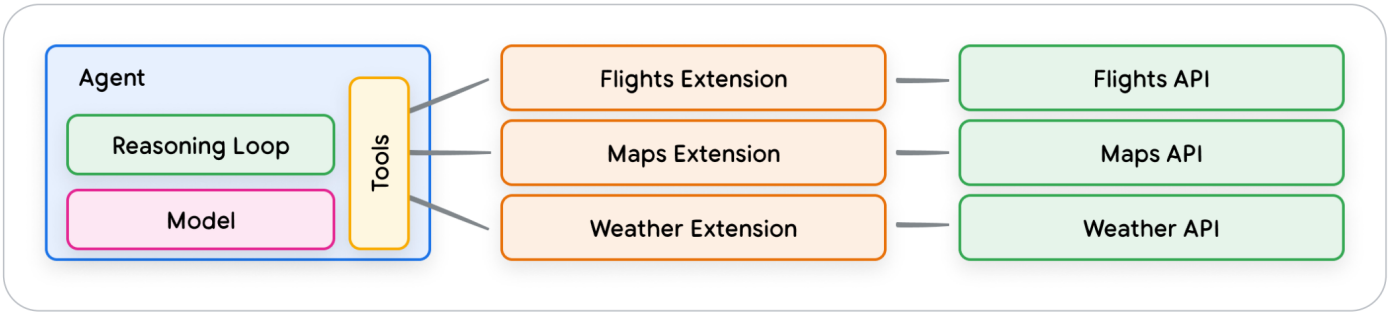


图 5. 智能体、扩展和 API 之间的一对多关系

可以将其想象成软件开发人员在为用户问题寻找解决方案时决定使用哪些 API 端点的方式。如果用户想要预订航班，开发人员可能会使用 Google Flights API。如果用户想知道最近的咖啡店相对于他们的位置在哪里，开发人员可能会使用 Google Maps API。以同样的方式，智能体/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。

如果你想亲身体验扩展，可以在 Gemini 应用程序中尝试，方法是转到设置 (Settings) --> 扩展 (Extensions)，然后启用任何你想测试的扩展。例如，你可以启用 Google Flights 扩展，然后问 Gemini Show me flights from Austin to Zurich leaving next Friday.

示例扩展 (Sample Extensions)

为了简化扩展的使用，Google 提供了一些开箱即用的扩展，可以快速导入到你的项目中，并且只需最少的配置即可使用。例如，代码片段 1 中的代码解释器 (Code Interpreter) 扩展允许你根据自然语言描述生成并运行 Python 代码。

Python

```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

    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

总而言之，**扩展为代理提供了一种以多种方式感知、交互和影响外部世界的方法**。这些扩展的选择和调用是通过使用示例来指导的，所有这些示例都被定义为扩展配置的一部分。

3.2 函数 (Functions)

在软件工程领域，函数被定义为完成特定任务的自包含代码模块，并且可以根据需要重用。当软件开发人员编写程序时，他们通常会创建许多函数来执行各种任务。他们还将定义何时调用函数 a 与函数 b 的逻辑，以及预期的输入和输出。

函数在智能体世界中的工作方式非常相似，但我们可以用模型替换软件开发人员。模型可以接受一组已知的函数，并根据其规范决定何时使用某个函数以及该函数需要哪些参数。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**不**进行实时的 API 调用。
2. 函数在**客户端 (client-side)** 执行，而扩展在**智能体端 (agent-side)** 执行。

再次使用我们的 Google Flights 示例，一个简单的函数设置可能看起来像图 7 中的示例。

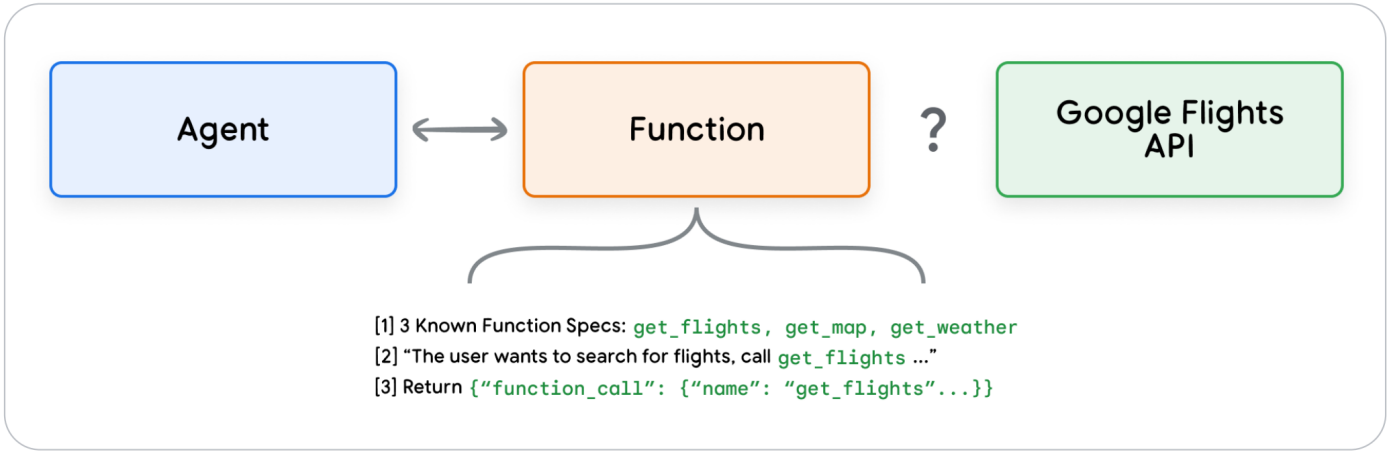


图 7. 函数如何与外部 API 交互?

请注意，这里的主要区别在于函数和智能体都**不**直接与 Google Flights API 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 端点的逻辑和执行被从智能体中移除，回到了客户端应用程序，如图 8 和图 9 所示。**这为开发人员提供了对应用程序中数据流更精细的控制**。开发人员可能选择使用函数而不是扩展的原因有很多，但一些常见的用例是：

- API 调用需要在应用程序堆栈的另一层进行，在直接的智能体架构流程之外（例如，中间件系统、前端框架等）
- 安全或身份验证限制阻止智能体直接调用 API（例如，API 未暴露给互联网，或智能体基础设施无法访问）
- 阻止智能体实时进行 API 调用的时间或操作顺序限制。（即批处理操作、人工审核环节等）

- 需要对 API 响应应用额外的智能体无法执行的数据转换逻辑。例如，考虑一个不提供限制返回结果数量的过滤机制的 API 端点。在客户端使用函数为开发人员提供了进行这些转换的额外机会。
- 开发人员希望在不部署 API 端点的额外基础设施的情况下迭代智能体开发（即函数调用可以充当 API 的“存根 (stubbing)”）

虽然两种方法之间的内部架构差异如图 8 所示是微妙的，但额外的控制和与外部基础设施解耦的依赖性使得函数调用成为开发人员的一个有吸引力的选择。

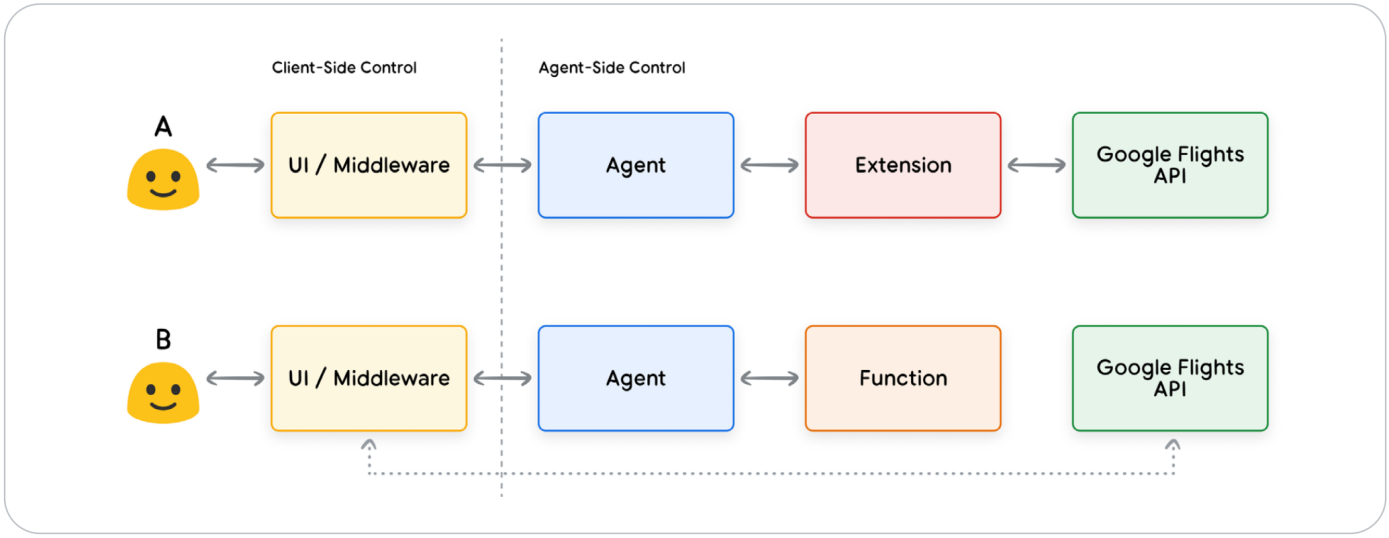


图 8. 描述扩展和函数调用的客户端与智能体端控制

使用案例 (Use cases)

模型可以被用来调用函数，以处理最终用户复杂的、客户端执行流程，其中智能体开发人员可能不希望语言模型管理 API 执行（就像扩展的情况一样）。让我们考虑以下示例，其中一个智能体被训练为旅行礼宾，与想要预订度假旅行的用户互动。目标是让智能体生成一个城市列表，我们可以在我们的中间件应用程序中使用该列表来下载图像、数据等，用于用户的旅行规划。用户可能会说这样的话：

我想和家人去滑雪旅行，但不确定去哪里。

在对模型的典型提示中，输出可能如下所示：

> 当然，这里有一些你可以考虑的家庭滑雪旅行城市列表：
>
> - *Crested Butte, Colorado, USA*
> - *Whistler, BC, Canada*
> - *Zermatt, Switzerland*

虽然上面的输出包含了我们需要的数据（城市名称），但格式对于解析来说并不理想。通过函数调用，我们可以教模型以结构化样式（如 JSON）格式化此输出，这对于另一个系统解析更为方便。给定来自用户的相同输入提示，来自函数的**示例 JSON 输出**可能看起来像代码片段 5。

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

代码片段 5. 用于显示城市列表和用户偏好的示例函数调用负载（payload）

这个 JSON 负载由模型生成，然后发送到我们的客户端服务器，以执行我们想用它做的任何事情。在这个特定案例中，我们将调用 Google Places API，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富内容提供回我们的用户。考虑图 9 中的序列图，它逐步详细显示了上述交互。

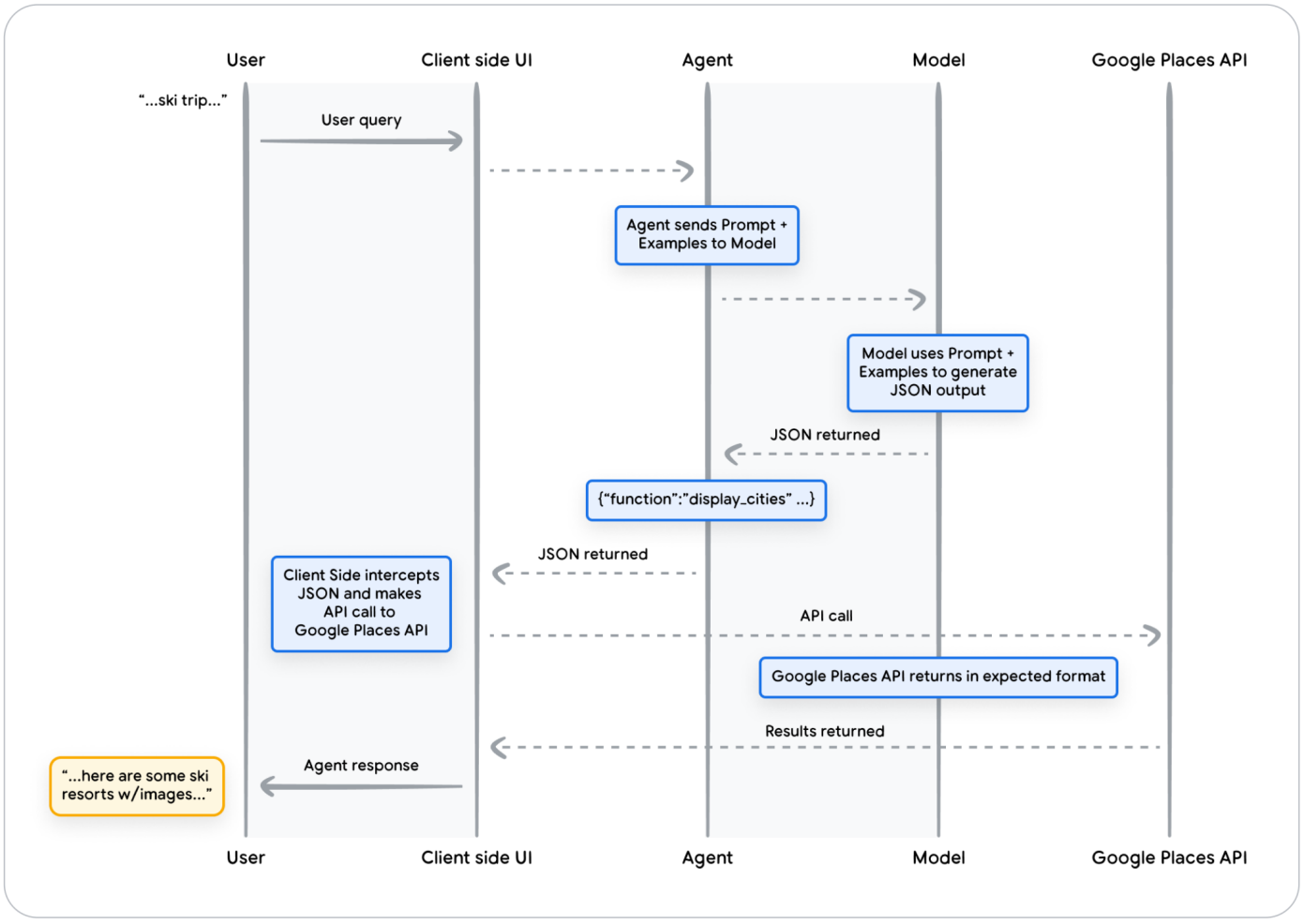


图 9. 展示函数调用生命周期的序列图

图 9 中示例的结果是，模型被用来“填空”，提供客户端 UI 调用 Google Places API 所需的参数。客户端 UI 使用模型在返回的函数中提供的参数来管理实际的 API 调用。这只是函数调用的一个用例，但还有许多其他场景需要考虑，例如：

- 你希望一个语言模型为你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。因为函数调用不会运行函数，所以在提供函数信息时，你不需要在代码中包含凭据。
- 你正在运行可能需要几秒钟以上的异步操作。这些场景非常适合函数调用，因为它是一个异步操作。
- 你想在与生成函数调用及其参数的系统不同的设备上运行函数。

关于函数的一个关键点需要记住：它们旨在为开发者提供更多控制，不仅是对 API 调用的执行，还有整个应用程序的数据流。在图 9 的例子中，开发者选择不将 API 信息返回给智能体，因为这些信息对于智能体可能采取的未来行动并不相关。 然而，根据应用程序的架构，将外部 API 调用数据返回给智能体可能是合理的，以便影响未来的推理、逻辑和行动选择。 最终，具体应用程序应该如何选择，取决于应用程序开发者。

函数示例代码 (Function sample code)

为了从我们的滑雪度假场景中实现上述输出，让我们构建每个组件，使其与我们的 gemini-1.5-flash-001 模型一起工作。

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

Python

```
from typing import Optional

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

```

代码片段 6. 用于显示城市列表的示例 python 方法函数。

接下来，我们将实例化我们的模型，构建工具 (Tool)，然后将用户的查询和工具传递给模型。执行下面的代码将导致代码片段底部的输出。

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']}
```

代码片段 7. 构建一个工具，将其与用户查询一起发送给模型，并允许函数调用发生

总而言之，函数提供了一个直接的框架，使应用程序开发人员能够对数据流和系统执行进行细粒度控制，同时有效地利用智能体/模型进行关键输入生成。开发人员可以通过返回外部数据来选择性地让智能体保持「在循环中」，或者根据特定的应用程序架构要求省略它。

3.3 数据存储 (Data stores)

想象一个语言模型，它像一个巨大的图书馆，包含着它的训练数据。但与不断购入新书的图书馆不同，这个图书馆保持静态，只拥有它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 (Data Stores) 通过提供对更动态和最新信息的访问来解决这一限制，并确保模型的响应保持基于事实和相关性。

考虑一个常见的场景，开发人员可能需要向模型提供少量额外数据，也许是以电子表格或 PDF 的形式。

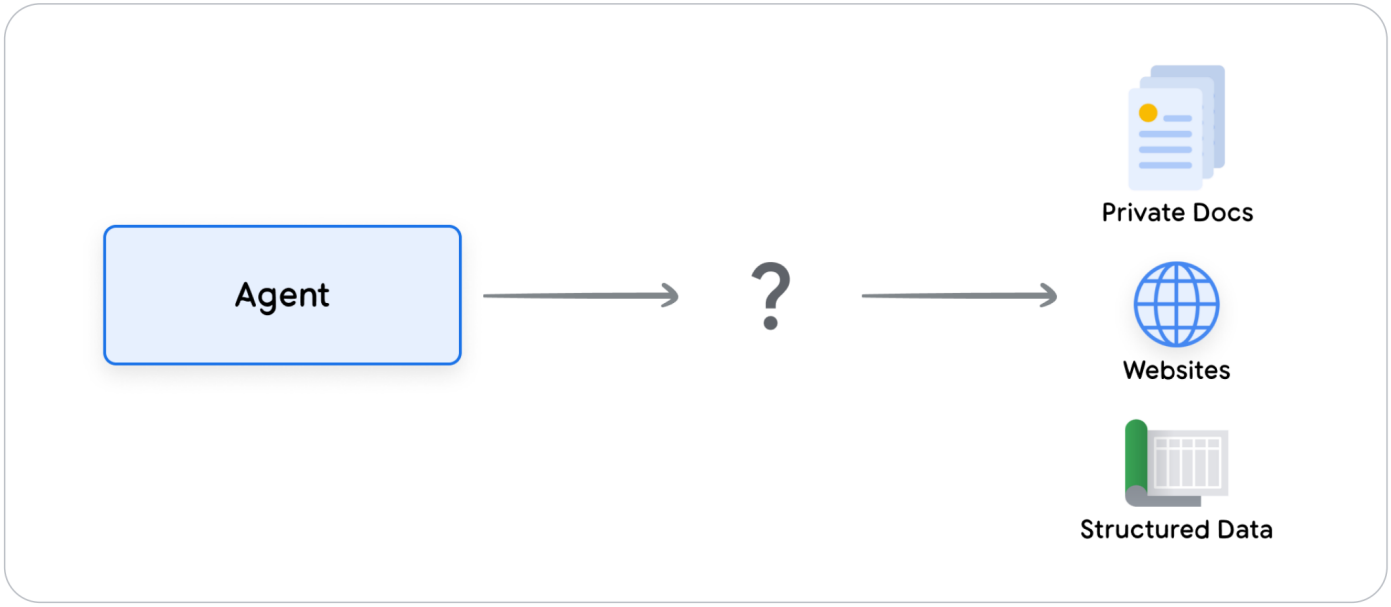


图 10. 智能体如何与结构化和非结构化数据交互?

数据存储允许开发人员以其原始格式向智能体提供额外数据，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（vector database embeddings），智能体可以使用这些嵌入来提取所需信息，以补充其下一步行动或对用户的响应。

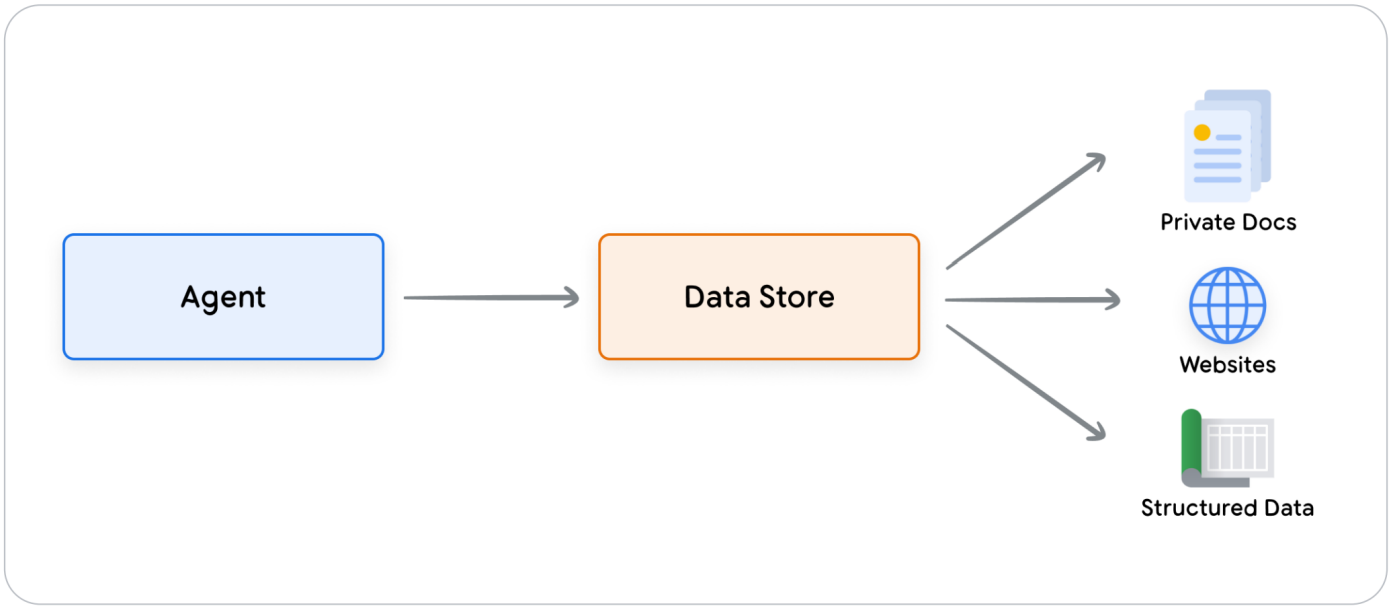


图 11. 数据存储将智能体连接到各种新的实时数据源

实现与应用

在生成式 AI 智能体的背景下，数据存储通常实现为开发人员希望智能体在运行时能够访问的向量数据库 (vector database)。虽然我们不会在这里深入探讨向量数据库，但关键点是要理解它们以向量嵌入 (vector embeddings) 的形式存储数据，这是一种所提供数据的高维向量或数学表示。近年来，语言模型使用数据存储最普遍的例子之一是检索增强生成 (Retrieval Augmented Generation, RAG) 的实现。

基于 RAG 的应用程序旨在通过让模型访问各种格式的数据来扩展模型知识的广度和深度，超越其基础训练数据，例如：

- 网站内容
- 结构化数据，格式如 PDF、Word 文档、CSV、电子表格等。
- 非结构化数据，格式如 HTML、PDF、TXT 等。

图 12. 智能体和数据存储之间的一对多关系，可以表示各种类型的预索引数据

每个用户请求和智能体响应循环的底层过程通常建模如图 13 所示。

1. 用户查询被发送到一个嵌入模型 (embedding model) 以生成查询的嵌入。
2. 然后使用匹配算法（如 ScaNN）将查询嵌入与向量数据库的内容进行匹配。
3. 匹配的内容以文本格式从向量数据库中检索出来，并发送回智能体。
4. 智能体接收用户查询和检索到的内容，然后制定响应或行动。
5. 最终响应被发送给用户。

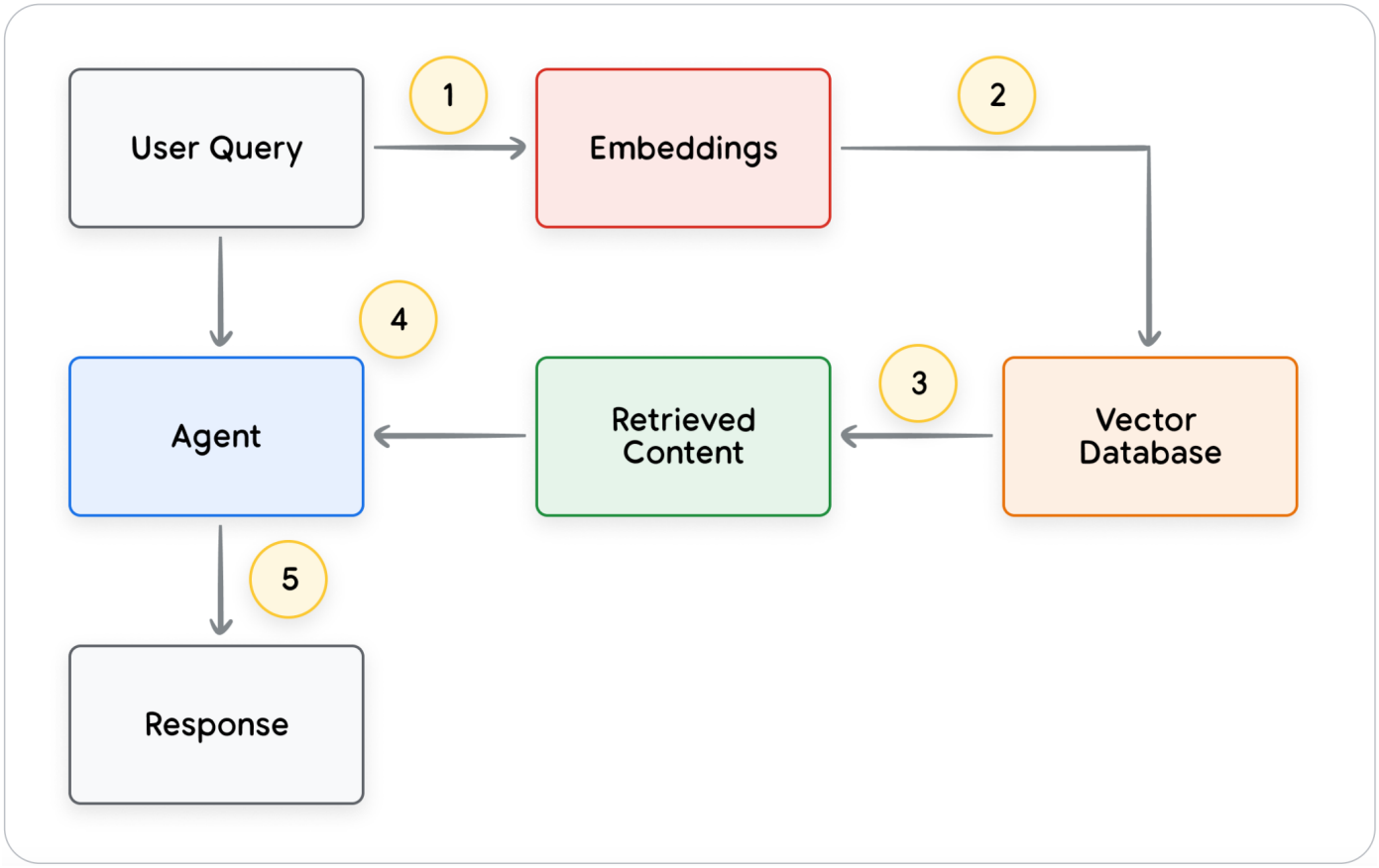


图 13. RAG 应用中用户请求和智能体响应的生命周期

最终结果是一个应用程序，它允许智能体通过向量搜索将用户查询匹配到已知的数据存储，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。下一步行动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。

一个实现了 RAG 并带有 ReAct 推理/规划的智能体的示例交互可以在图 14 中看到。

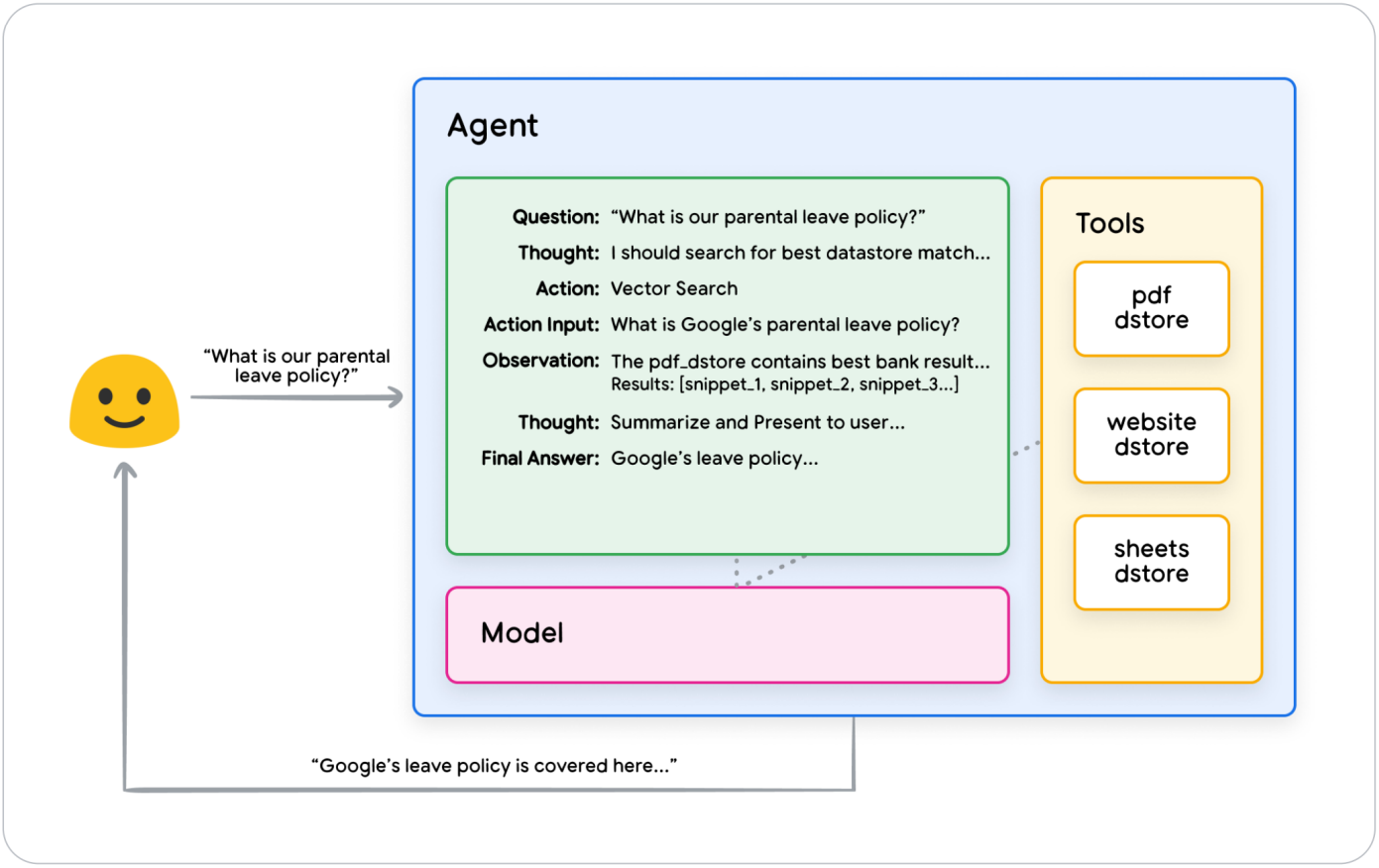


图 14. 带有 ReAct 推理/规划的示例 RAG 应用

3.4 工具回顾 (Tools recap)

总而言之，**扩展**、**函数**和**数据存储**构成了智能体在运行时可用的几种不同工具类型。每种都有其自身的用途，并且可以根据智能体开发人员的判断一起使用或独立使用。

	扩展 (Extensions)	函数调用 (Function Calling)	数据存储 (Data Stores)
执行	智能体端执行 (Agent-Side Execution)	客户端执行 (Client-Side Execution)	智能体端执行 (Agent-Side Execution)
使用案例	• 开发人员希望智能体控制与 API 端点的交互 • 在利用原生预构建扩展时很有用（例如 Vertex Search, Code Interpreter 等） • 多跳规划和 API 调用（即下一个智能体行动取决于前一个行动/API 调用的输出）	• 安全或身份验证限制阻止智能体直接调用 API • 时间限制或操作顺序约束阻止智能体实时进行 API 调用。(即批处理操作，人工审核环节等) • 未暴露给互联网或 Google 系统无法访问的 API	开发人员希望实现检索增强生成 (RAG)，使用以下任何数据类型： • 来自预索引域和 URL 的网站内容 • 结构化数据，格式如 PDF, Word Docs, CSV, Spreadsheets 等 • 关系型/非关系型数据库 • 非结构化数据，格式如 HTML, PDF, TXT 等

4、通过定向学习（targeted learning）增强模型性能

有效使用模型的一个关键方面是它们在生成输出时选择正确工具的能力，尤其是在生产中大规模使用工具时。虽然通用训练帮助模型发展这项技能，但现实世界的场景通常需要超出训练数据的知识。可以将此想象为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，但后者需要针对更细微结果的定向学习。

为了帮助模型获得这种类型的特定知识，存在几种方法：

- **上下文学习 (In-context learning):** 此方法在推理时为通用模型提供提示、工具和少量示例 (few-shot examples)，使其能够“即时 (on the fly)”学习如何以及何时为特定任务使用这些工具。ReAct 框架是这种自然语言方法的示例。

- **基于检索的上下文学习 (Retrieval-based in-context learning):** 此技术通过从外部存储器中检索最相关的信息、工具和相关示例来动态填充模型提示。Vertex AI 扩展中的“示例存储 (Example Store)”或前面提到的基于数据存储的 RAG 架构就是这种方法的示例。
- **基于微调的学习 (Fine-tuning based learning):** 此方法涉及在推理之前使用更大的特定示例数据集训练模型。这有助于模型在收到任何用户查询之前理解何时以及如何应用某些工具。

为了对每种定向学习方法提供更多见解，让我们回顾一下我们的烹饪类比。

- 想象一下，一位厨师收到了一个特定的食谱（提示）、一些关键食材（相关工具）和来自顾客的一些示例菜肴（少量示例）。基于这些有限的信息和厨师的通用烹饪知识，他们需要“即时”弄清楚如何准备最符合食谱和顾客偏好的菜肴。这就是**上下文学习**。
- 现在让我们想象一下，我们的厨师在一个厨房里，拥有一个储备充足的食品储藏室（外部数据存储），里面装满了各种食材和食谱（示例和工具）。厨师现在能够从食品储藏室动态选择食材和食谱，并更好地匹配顾客的食谱和偏好。这使得厨师能够利用现有知识和新知识创造出更信息丰富、更精致的菜肴。这就是**基于检索的上下文学习**。
- 最后，让我们想象一下，我们送厨师回学校学习一种新的菜系或一系列菜系（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的理解来处理未来未见过的顾客食谱。如果我们希望厨师在特定菜系（知识领域）方面表现出色，这种方法是完美的。这就是**基于微调的学习**。

这些方法中的每一种在速度、成本和延迟方面都提供了独特的优势和劣势。然而，通过在智能体框架中结合这些技术，我们可以利用各种优势并最小化它们的劣势，从而实现更健壮和适应性更强的解决方案。

5、使用 LangChain 快速开始智能体（Agent quick start with LangChain）

为了提供一个智能体在实际操作中的真实可执行示例，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、推理和工具调用序列“链接”在一起来构建客户智能体，以回答用户的查询。我们将使用我们的 `gemini-1.5-flash-001` 模型和一些简单的工具来回答来自用户的多阶段查询，如代码片段 8 所示。

我们使用的工具是 SerpAPI（用于 Google 搜索）和 Google Places API。在代码片段 8 中执行我们的程序后，你可以在代码片段 9 中看到示例输出。

Python

```
import os
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
from langchain_google_vertexai import ChatVertexAI

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()
```

代码片段 8. 基于 LangChain 和 LangGraph 的带工具示例智能体

输出

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

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

```

代码片段 9. 代码片段 8 中程序的输出

虽然这是一个相当简单的智能体示例，但它演示了模型、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 Google 规模的托管产品（如 Vertex AI 智能体和生成式 Playbooks）中结合在一起。

6、使用 Vertex AI 智能体的生产应用

虽然本白皮书探讨了智能体的核心组件，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附加工具集成。Google 的 Vertex AI 平台通过提供一个包含前面提到的所有基本要素的全托管环境来简化这一过程。使用自然语言界面，开发人员可以快速定义其智能体的关键要素，即目标、任务指令、工具、用于任务委托的子智能体以及示例，以轻松构建所需的系统行为。此外，该平台还配备了一套开发工具，允许测试、评估、衡量智能体性能、调试以及提高已开发智能体的整体质量。这使得开发人员能够专注于构建和优化他们的智能体，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 Vertex AI 平台构建的智能体示例架构，使用了诸如 Vertex Agent Builder、Vertex Extensions、Vertex Function Calling 和 Vertex Example Store 等多种特性。该架构包含了创建应用程序所需的各种组件。

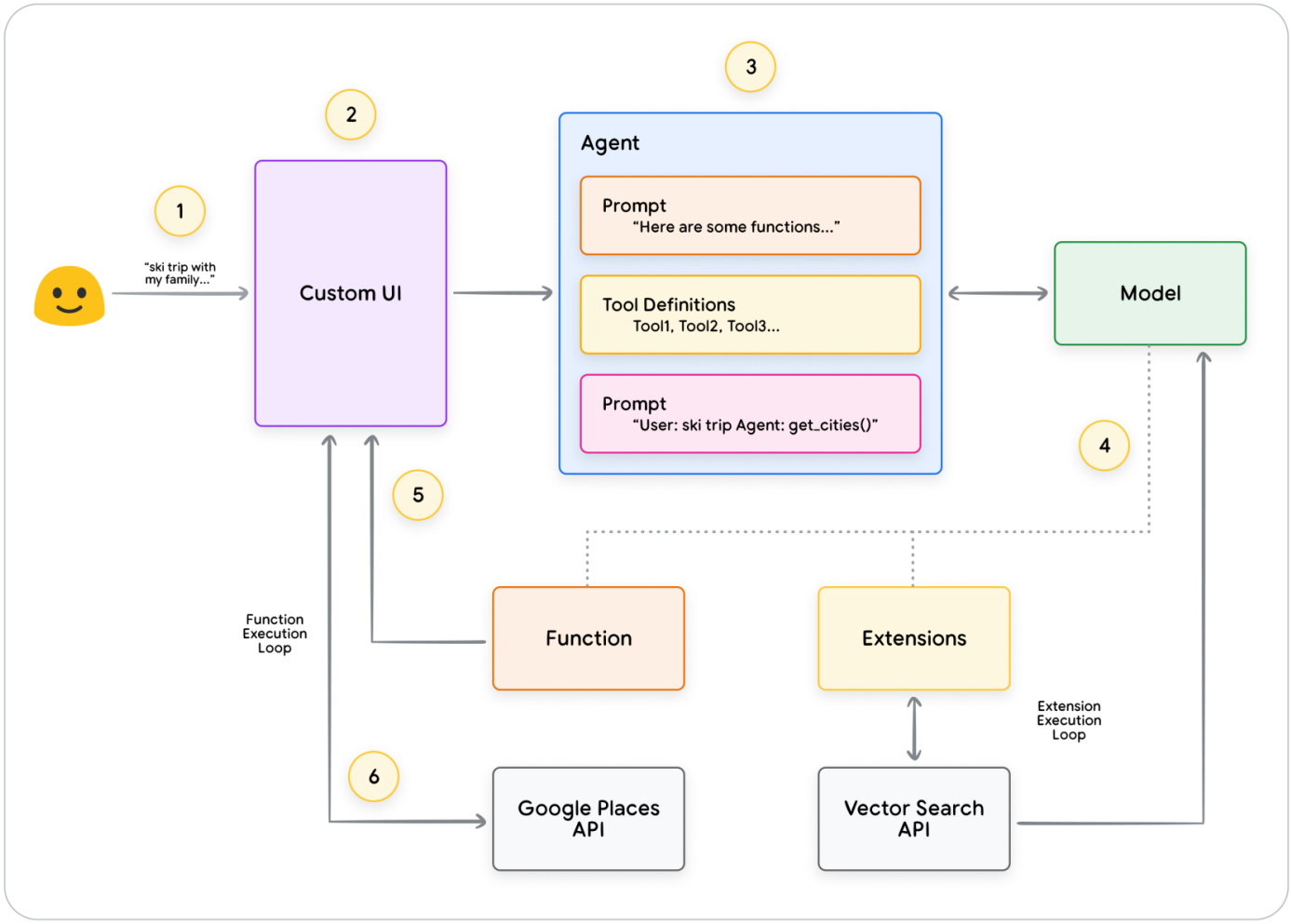


图 15. 在 Vertex AI 平台上构建的示例端到端智能体架构

你可以从我们的官方文档中尝试这个预构建智能体架构的示例。

7、总结 (Summary)

在本白皮书中，我们讨论了生成式 AI 智能体的基础构建模块、它们的组成以及以认知架构形式有效实现它们的方法。本白皮书的一些关键要点包括：

1. 智能体通过利用工具访问实时信息、建议现实世界行动以及自主规划和执行复杂任务，扩展了语言模型的能力。智能体可以利用一个或多个语言模型来决定何时以及如何在状态之间转换，并使用外部工具来完成模型自身难以或不可能完成的任何数量的复杂任务。
2. 智能体操作的核心是编排层，这是一个构建推理、规划、决策制定并指导其行动的认知架构。各种推理技术，如 ReAct、思维链和思维树，为编排层提供了一个框架，用以接收信息、执行内部推理，并生成明智的决策或响应。

3. 工具，例如扩展 (Extensions)、函数 (Functions) 和数据存储 (Data Stores)，充当智能体通往外部世界的钥匙，允许它们与外部系统交互并访问其训练数据之外的知识。扩展在智能体和外部 API 之间提供了一座桥梁，实现了 API 调用和实时信息的检索。函数通过分工为开发人员提供了更细致的控制，允许智能体生成可以由客户端执行的函数参数。数据存储为智能体提供了对结构化或非结构化数据的访问，从而实现了数据驱动的应用。

智能体的未来拥有激动人心的进展，而我们才刚刚触及其表面。随着工具变得更加复杂，推理能力得到增强，智能体将有能力解决日益复杂的问题。此外，「智能体链 (agent chaining)」的战略方法将继续获得发展势头。

通过组合专门的智能体，其中每个智能体在特定领域或任务中表现出色，从而我们可以创建一个「混合智能体专家 (mixture of agent experts)」的方法，能够在各种行业和问题领域提供卓越的结果。

重要的是要记住，构建复杂的智能体架构需要迭代的方法。实验和改进是为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的生成性质，没有两个智能体是完全相同的。然而，通过利用这些基础组件各自的优势，我们可以创建有影响力的应用程序，扩展语言模型的能力，并驱动现实世界的价值。

尾注 (Endnotes)

1. Shafran, I., Cao, Y. et al., 2022, 'ReAct: Synergizing Reasoning and Acting in Language Models'. **https://arxiv.org/abs/2210.03629**
2. Wei, J., Wang, X. et al., 2023, 'Chain-of-Thought Prompting Elicits Reasoning in Large Language Models'. **https://arxiv.org/pdf/2201.11903.pdf**.
3. Wang, X. et al., 2022, 'Self-Consistency Improves Chain of Thought Reasoning in Language Models'. **https://arxiv.org/abs/2203.11171**.
4. Diao, S. et al., 2023, 'Active Prompting with Chain-of-Thought for Large Language Models'. **https://arxiv.org/pdf/2302.12246.pdf**.
5. Zhang, H. et al., 2023, 'Multimodal Chain-of-Thought Reasoning in Language Models'. **https://arxiv.org/abs/2302.00923**.
6. Yao, S. et al., 2023, 'Tree of Thoughts: Deliberate Problem Solving with Large Language Models'. **https://arxiv.org/abs/2305.10601**.
7. Long, X., 2023, 'Large Language Model Guided Tree-of-Thought'. **https://arxiv.org/abs/2305.08291**.

8. Google. 'Google Gemini Application'. **http://gemini.google.com**.
9. Swagger. 'OpenAPI Specification'. **https://swagger.io/specification/**.
10. Xie, M., 2022, 'How does in-context learning work? A framework for understanding the differences from traditional supervised learning'. **https://ai.stanford.edu/blog/understanding-incontext/**.
11. Google Research. 'ScaNN (Scalable Nearest Neighbors)'. **https://github.com/google-research/google-research/tree/master/scann**.
12. LangChain. 'LangChain'. **https://python.langchain.com/v0.2/docs/introduction/**.