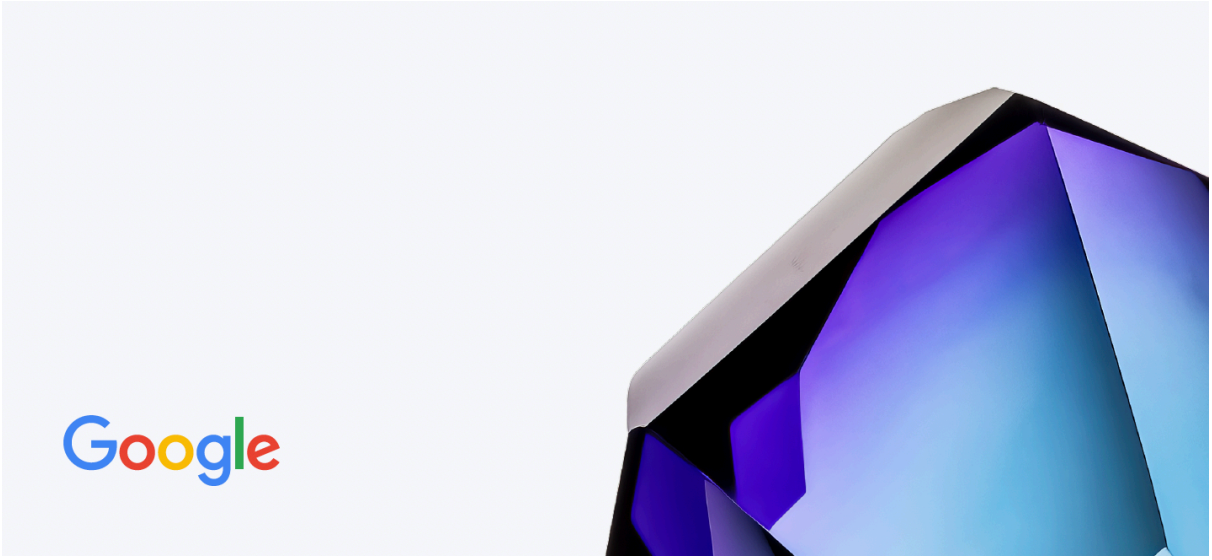
# 上下文工程：会话与记忆

***Context Engineering: Sessions, Memory***

**作者：Kimberly Milam 和 Antonio Gulli**

**翻译：Google notebooklm**

**英文版链接：**
https://drive.google.com/file/d/1JW6Q_wwvBjMz9xzOtTldFfPiF7BrdEeQ/view



> **有状态和个性化的 AI 始于上下文工程。**

# 引言 (Introduction)

本白皮书探讨了会话和记忆在构建**有状态、智能 LLM 智能体**中的关键作用，旨在赋能开发人员创造更强大、更个性化和更持久的 AI 体验。为了让大型语言模型（LLM）能够记忆、学习和个性化交互，开发人员必须在它们的**上下文窗口内动态组装和管理信息**，这一过程被称为**上下文工程（Context Engineering）**。

**有状态且个性化的人工智能始于上下文工程（Context Engineering）**。这些核心概念在以下白皮书中进行了总结：

- **上下文工程：动态地组装和管理** LLM **上下文窗口内的信息**，以实现有状态、智能代理的过程。
- **会话（Sessions）**：代理与用户进行**完整对话的容器**，包含对话的**按时间顺序排列的历史记录**和代理的**工作记忆**。
- **记忆（Memory）**：**长期持久化**的机制，用于**跨多个会话捕获和整合关键信息**，为 LLM 代理提供**连续和个性化的体验**。

# 上下文工程 (Context Engineering)

LLM 本质上是**无状态的**。除了它们的训练数据之外，它们的推理和感知能力仅限于**单个 API 调用**的“**上下文窗口**”内提供的信息。这提出了一个根本性的问题，因为 AI 智能体必须配备操作指令以识别可以采取哪些行动、进行推理的证据和事实数据，以及定义当前任务的即时对话信息。为了构建可以记忆、学习和个性化交互的**有状态、智能智能体**，开发人员必须为对话的每一轮构建此上下文。这种 LLM 信息的动态组装和管理被称为**上下文工程**。

上下文工程代表了对传统**提示工程（Prompt Engineering）精心设计最优的、通常是静态的系统指令**。相反，上下文工程处理**整个有效载荷（payload）**，根据用户、对话历史记录和外部数据动态构建**状态感知（state-aware）提示**。它涉及策略性地选择、总结和注入不同类型的信息，以**最大化相关性同时最小化噪音**。**外部系统**——例如 **RAG 数据库**、**会话存储**和**记忆管理器**——管理着大部分上下文。智能体框架必须协调这些系统，将上下文检索并组装到最终的提示中。

可以将上下文工程视为智能体的备料（mise en place）——即厨师在烹饪前收集和准备所有食材的关键步骤。如果只给厨师食谱（提示），他们可能会用手头随机的食材做出还过得去的饭菜。但是，如果首先确保他们拥有所有正确、高质量的食材、**专业的**

**工具** 以及对呈现风格的清晰理解，他们就可以可靠地做出出色、定制化的结果。上下文工程的目标是确保模型**不多不少，只拥有完成其任务最相关的信息**。

上下文工程管理着复杂有效载荷的组装，其中可能包括各种组件：

- 指导推理的上下文（Context to guide reasoning）定义了智能体的基本推理模式和可用操作，决定了其行为：
  - **系统指令（System Instructions）**：定义智能体人设、能力和限制的**高级指令**。
  - **工具定义（Tool Definitions）**：智能体可用于与外界交互的 API 或函数的模式。
  - **少样本示例（Few-Shot Examples）**：通过上下文内学习（in-context learning）指导模型推理过程的精选示例。
- 证据和事实数据（Evidential & Factual Data）是智能体进行推理的实质性数据，包括预先存在的知识和针对特定任务动态检索的信息；它作为智能体响应的“证据”：
  - **长期记忆（Long-Term Memory）**：关于用户或主题的**持久知识，跨多个会话收集**。
  - **外部知识（External Knowledge）**：从数据库或文档中检索的信息，通常使用**检索增强生成（RAG）**。
  - **工具输出（Tool Outputs）**：工具返回的数据或结果。
  - **子智能体输出（Sub-Agent Outputs）**：已被委派特定子任务的专业智能体返回的结论或结果。
  - **人工制品（Artifacts）**：与用户或会话关联的**非文本数据**（例如文件、图像）。
- 即时对话信息（Immediate conversational information）将智能体定位于当前的交互中，定义了即时任务：
  - **对话历史记录（Conversation History）**：当前交互的**逐轮记录**。
  - **状态 / 暂存器（State / Scratchpad）**：智能体用于即时推理过程的临时、正在进行中的信息或计算。
- **用户提示（User's Prompt）**：需要解决的即时查询。

上下文的动态构建至关重要。例如，**记忆不是静态的**；它们必须在用户与智能体交互或摄取新数据时被**选择性地检索和更新**。此外，有效的推理通常依赖于**上下文内学习**（LLM 从提示中的演示中学习如何执行任务的过程）。当智能体使用与当前任务相关

的**少样本示例**而不是依赖硬编码示例时，上下文内学习会更有效。同样，RAG 工具会根据用户的即时查询检索外部知识。

构建上下文感知智能体的最关键挑战之一是管理**不断增长的对话历史记录**。理论上，具有大上下文窗口的模型可以处理大量的文本记录；但在实践中，随着上下文的增长，**成本和延迟会增加**。此外，模型可能会遭受"**上下文腐烂（context rot）**"的困扰，这是一种现象，即随着上下文的增长，模型关注关键信息的能力会减弱。上下文工程通过采用**动态改变历史记录**的策略——例如**总结、选择性修剪或其他压缩技术**——来直接解决这个问题，以**保留关键信息同时管理整体令牌计数**，最终带来更健壮和个性化的 AI 体验。

这种实践在智能体的**操作循环**中表现为一个**连续的周期**，用于对话的每一轮：

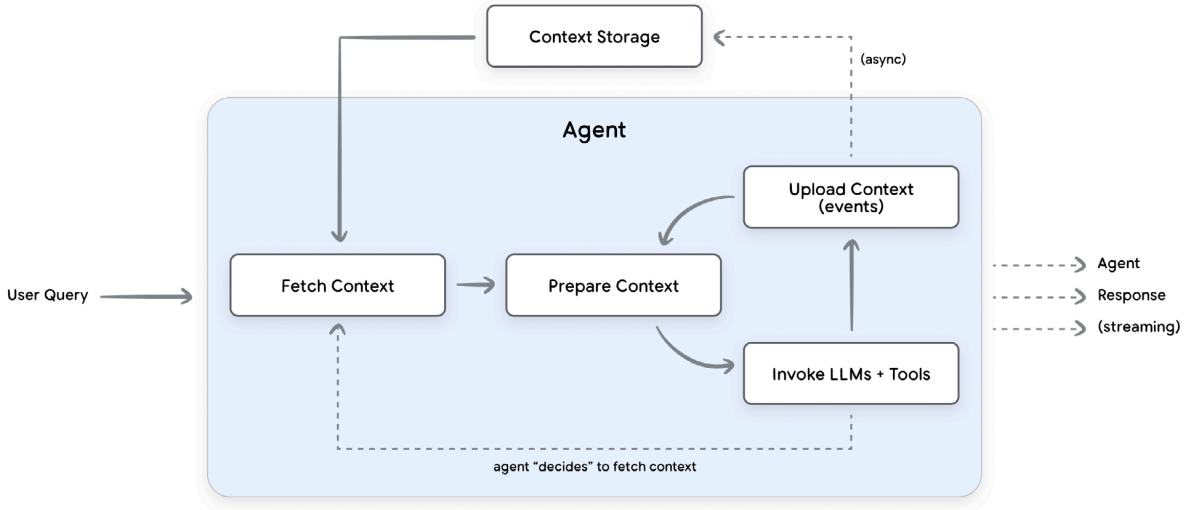



图 1. 智能体（Agent）上下文管理流程图

1. **获取上下文（Fetch Context）**：智能体首先检索上下文——例如用户记忆、RAG 文档和最近的对话事件。对于动态上下文检索，智能体将使用用户查询和其他元数据来识别要检索的信息。

2. **准备上下文（Prepare Context）**：智能体框架动态地为 LLM 调用构建完整的提示。尽管单个 API 调用可能是异步的，但**准备上下文是一个阻塞的、"热路径"过程**。在上下文准备好之前，智能体无法继续。

3. **调用 LLM 和工具（Invoke LLM and Tools）**：智能体迭代地调用 LLM 和任何必要的工具，直到为用户生成最终响应。工具和模型输出被附加到上下文中。

4. **上传上下文（Upload Context）**：在本轮中收集到的新信息被上传到持久存储中。这通常是一个"**后台**"过程，允许智能体在记忆整合或其他后处理异步发生时完成执行。

在这个生命周期的核心是两个基本组成部分：会话和记忆。**会话**管理着**单个对话的逐轮状态**。相比之下，**记忆**提供了**长期持久性**的机制，**跨多个会话捕获和整合关键信息**。

可以将会话视为你用于特定项目的**工作台或办公桌**。当你工作时，桌子上摆满了所有必要的工具、笔记和参考资料。一切都**可立即访问，但也是临时且特定于手头任务的**。项目完成后，你不会将整个凌乱的桌子都塞进存储空间。相反，你开始创建记忆的过程，记忆就像一个**有组织的档案柜**。你会查看桌上的材料，丢弃草稿和冗余的笔记，并仅将**最关键、已定稿的文件归档**到带标签的文件夹中。这确保了档案柜对于未来所有项目来说仍然是一个**干净、可靠、高效的事实来源**，而不会被工作台的短暂混乱所干扰。这个类比直接反映了一个有效智能体的运作方式：**会话充当单个对话的临时工作台，而智能体的记忆是精心组织的档案柜**，允许它在未来的交互中回忆起关键信息。

基于对上下文工程的这一高级概述，我们现在可以探讨两个核心组成部分：会话和记忆，从会话开始。

# 会话 (Sessions)

上下文工程（Context Engineering）的一个基础元素是会话，它封装了单一、持续对话的即时对话历史和工作记忆。每个会话都是一个自包含的记录，与特定的用户绑定。会话允许**智能体**在单个对话的范围内保持上下文并提供连贯的响应。一个用户可以有多个会话，但每个会话都作为特定交互的独立、不相关的日志运行。每个会话都包含两个关键组成部分：按时间顺序排列的历史记录（事件）和**智能体**的工作记忆（状态）。

**事件（Events）状态（state）**——一个结构化的“工作记忆”或暂存器。它保存与当前对话相关的临时结构化数据，例如购物车中有哪些商品。

随着对话的进行，**智能体**会将额外的事件附加到会话中。此外，它可能会根据**智能体**中的逻辑来改变状态。事件的结构类似于传递给 Gemini API 的 Content 对象列表，其中每个具有 `role` 和 `parts` 的项目代表对话中的一轮——或一个事件。生产**智能体**的执行环境通常是无状态的，这意味着请求完成后它不会保留任何信息。因此，其对话历史必须保存到持久存储中，以维持持续的用户体验。虽然内存存储适用于开发，但生产应用程序应利用强大的数据库来可靠地存储和管理会话。例如，您可以将对话历史存储在像 Agent Engine Sessions3 这样的托管解决方案中。

```
# 片段 1：对 Gemini 的多轮调用示例
contents = [
  {
    "role": "user",
```

```
        "parts": [ {"text": "What is the capital of France?"} ]
    },
    {
        "role": "model",
        "parts": [ {"text": "The capital of France is Paris."} ]
    }
]
response = client.models.generate_content(
    model="gemini-2.5-flash",
    contents=contents
)
```

## 框架和模型之间的差异 (Variance across frameworks and models)

虽然核心思想相似，但不同的**智能体**框架以不同的方式实现会话、事件和状态。**智能体**框架负责维护 LLM 的对话历史和状态，使用此上下文构建 LLM 请求，以及解析和存储 LLM 响应。

**智能体**框架充当您的代码与 LLM 之间的通用翻译器。作为开发人员，您处理框架针对每个对话轮次的统一内部数据结构，而框架则处理关键任务：将这些结构转换为 LLM 所需的精确格式。这种抽象非常强大，因为它将您的**智能体**逻辑与您正在使用的特定 LLM 解耦，从而防止供应商锁定。

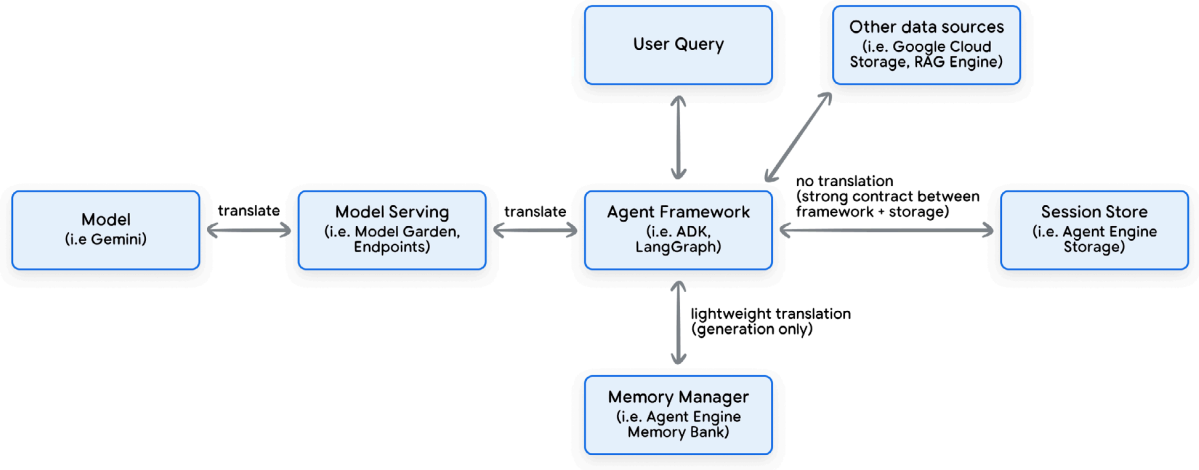



图 2：智能体（Agent）上下文管理流程图

最终目标是生成 LLM 可以理解的“请求”。对于 Google 的 Gemini 模型，这是一个 `List[Content]`。每个 `Content` 对象是一个简单的类似字典的结构，包含两个键：`role`，它

定义了说话者（“user”或“model”）；以及 `parts`，它定义了消息的实际内容（文本、图像、工具调用等）。框架在进行 API 调用之前，会自动处理将数据从其内部对象（例如 ADK Event）映射到 `Content` 对象中相应的 `role` 和 `parts` 的任务。实质上，框架为开发人员提供了一个稳定的内部 API，同时在幕后管理着不同 LLM 复杂多样的外部 API。

ADK 使用一个明确的 Session 对象，其中包含一个 Event 对象列表和一个单独的状态对象。Session 就像一个文件柜，其中一个文件夹用于对话历史记录（事件），另一个用于工作记忆（状态）。LangGraph 没有正式的“会话”对象。相反，**状态**就是会话。这个包罗万象的状态对象包含对话历史记录（作为 Message 对象列表）和所有其他工作数据。与传统会话的仅追加日志不同，LangGraph 的状态是可变的。它可以被转换，并且像历史压缩这样的策略可以修改记录。这对于管理长对话和令牌限制非常有用。

## 多智能体系统中的会话 (Sessions for multi-agent systems)

在多**智能体**系统中，多个**智能体**进行协作。每个**智能体**专注于一个更小、更专业的任务。为了使这些**智能体**有效地协同工作，它们必须共享信息。如框图所示，系统架构定义了它们用于共享信息的通信模式。该架构的一个核心组成部分是系统如何处理会话历史记录——所有交互的持久日志。

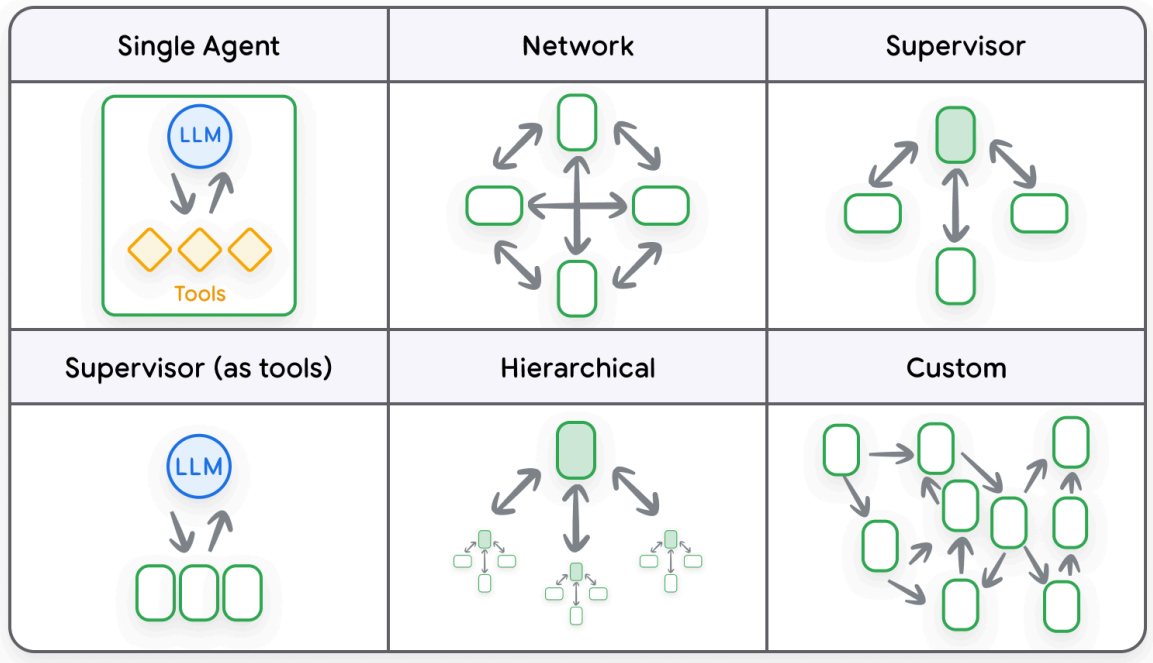



图 3：不同的多智能体架构模式

在探索用于管理此历史记录的架构模式之前，区分它与发送给 LLM 的上下文至关重要。将会话历史记录视为整个对话的永久、未经删节的脚本。另一方面，**上下文**是为单个轮次精心制作并发送给 LLM 的信息有效载荷。**智能体**可以通过从历史记录中仅选择相关摘录，或通过添加特殊格式（例如指导性前言）来构建此上下文，以引导模型的响应。本节的重点是**智能体**之间传递了哪些信息，而不一定是发送给 LLM 的上下文。

**智能体**框架使用两种主要方法之一来处理多**智能体**系统的会话历史记录：所有**智能体**都贡献到一个日志的**共享统一历史记录**，或每个**智能体**维护自己视角的**单独个体历史记录**。这两种模式之间的选择取决于任务的性质以及**智能体**之间所需的协作方式。

对于**共享统一历史记录**模型，系统中的所有**智能体**都从相同、单一的对话历史记录中读取，并将所有事件写入其中。每个**智能体**的消息、工具调用和观察都按时间顺序附加到一个中央日志中。这种方法最适用于需要单一事实来源的紧密耦合协作任务，例如多步解决问题流程，其中一个**智能体**的输出是下一个**智能体**的直接输入。即使是共享历史记录，子**智能体**也可能会在将其传递给 LLM 之前对其进行处理。例如，它可以过滤掉相关事件的子集，或添加标签以识别哪个**智能体**生成了哪个事件。

如果您使用 ADK 的 LLM 驱动委托来将任务移交给子**智能体**，则子**智能体**的所有中间事件都将写入与根**智能体**相同的会话中。

```
# 片段 2：跨多个智能体框架的 A2A 通信
from google.adk.agents import LlmAgent
# 子智能体可以访问 Session 并向其中写入事件。
sub_agent_1 = LlmAgent(...)
# 可选地，子智能体可以将最终响应文本（或结构化输出）保存到指定的 state 键。
sub_agent_2 = LlmAgent(
  ...,
  output_key="..."
)
# 父智能体。
root_agent = LlmAgent(
  ...,
  sub_agents=[sub_agent_1, sub_agent_2]
)
```

在**单独个体历史记录**模型中，每个**智能体**维护自己的私有对话历史记录，并像黑箱一样对其他**智能体**运行。所有内部流程——例如中间思想、工具使用和推理步骤——都

保留在**智能体**的私有日志中，对其他**智能体**不可见。通信仅通过明确的消息发生，其中**智能体**分享其最终输出，而不是其过程。

这种交互通常通过实现**智能体即工具（Agent-as-a-tool）**智能体到智能体（A2A）**协议**来实现。对于“**智能体**即工具”，一个**智能体**像调用标准工具一样调用另一个**智能体**，传递输入并接收最终的、自包含的输出。对于“**智能体**到**智能体**（A2A）协议”，**智能体**使用结构化协议进行直接消息传递。

我们将在下一部分更详细地探讨 A2A 协议。

## 跨多个智能体框架的互操作性 (Interoperability across multiple agent frameworks)

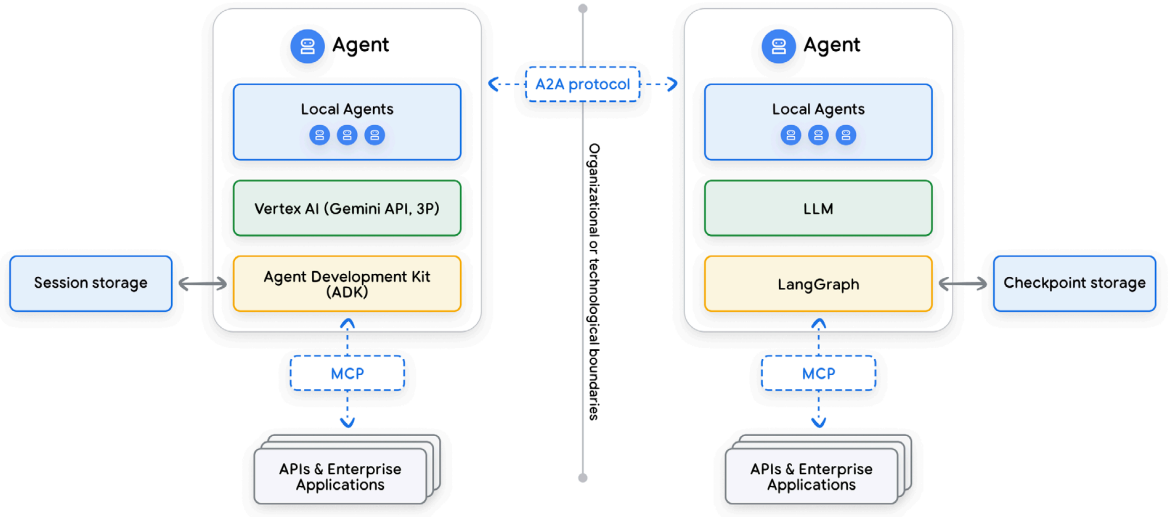



图 4：跨使用不同框架的多个代理之间的 A2A 通信机制

框架对内部数据表示的使用给多**智能体**系统带来了一个关键的架构权衡：将**智能体**与 LLM 解耦的抽象，同时也将其与其他使用不同**智能体**框架的**智能体**隔离开来。这种隔离在持久层得到固化。会话的存储模型通常将数据库模式直接耦合到框架的内部对象，从而创建了一个僵化、相对不可移植的对话记录。因此，使用 LangGraph 构建的**智能体**无法原生解释由基于 ADK 的**智能体**持久化的不同 Session 和 Event 对象，使得无缝的任务移交变得不可能。协调这些隔离**智能体**之间协作的一种新兴架构模式是**智能体到智能体（A2A）通信**。虽然此模式使**智能体**能够交换消息，但它未能解决共享丰富上下文状态的核心问题。每个**智能体**的对话历史都编码在其框架的内部模式中。因此，任何包含会话事件的 A2A 消息都需要一个翻译层才能有用。一种更稳健的互操作性架构模式是将共享知识抽象到一个与框架无关的数据层，例如**记忆（Memory）**。与保存原始、特定于框架的对象（如 Events 和 Messsages）的会话存储不同，记忆层旨在保存经过处理的、规范的信息。关键信息——如摘要、提取的实体和事实——从对话中提取出来，通常存储为字符串或字典。记忆层的数据结构不

与任何单个框架的内部数据表示耦合，这使得它能够充当通用的、公共的数据层。此模式允许异构**智能体**通过共享共同的认知资源来实现真正的协作智能，而无需定制翻译器。

# 会话的生产环境考量 (Production Considerations for Sessions)

将**智能体**迁移到生产环境时，其会话管理系统必须从一个简单的日志演变为一个稳健的企业级服务。关键考量因素分为三个关键领域：**安全和隐私、数据完整性**和**性能**。像 Agent Engine Sessions 这样的托管会话存储专为满足这些生产要求而设计。

安全和隐私 (Security and Privacy)

保护会话中包含的敏感信息是不可协商的要求。**严格隔离**是最关键的安全原则。会话归单个用户所有，系统必须强制执行严格隔离，以确保一个用户永远无法访问另一个用户的会话数据（即通过 ACLs）。对会话存储的每个请求都必须针对会话所有者进行身份验证和授权。处理个人身份信息（PII）的最佳实践是**在会话数据写入存储之前对其进行编辑**。这是一项基本的安全措施，可大幅降低潜在数据泄露的风险和“爆炸半径”。通过使用 Model Armor9 等工具确保敏感数据永远不会持久化，您可以简化对 GDPR 和 CCPA 等隐私法规的遵守，并建立用户信任。

数据完整性和生命周期管理 (Data Integrity and Lifecycle Management)

生产系统需要明确的规则来管理会话数据随时间的存储和维护方式。会话不应永久存活。您可以实施**生存时间（TTL）**策略来自动删除不活跃的会话，以管理存储成本并减少数据管理开销。这需要明确的数据保留政策，定义会话在存档或永久删除之前应保留多长时间。此外，系统必须保证操作以确定性顺序附加到会话历史记录中。保持事件正确的按时间顺序排列对于对话日志的完整性至关重要。

性能和可扩展性 (Performance and Scalability)

会话数据位于每个用户交互的“热路径”上，因此其性能是首要关注的问题。读取和写入会话历史记录必须极其快速，以确保响应式的用户体验。**智能体**运行时通常是无状态的，因此在每个轮次的开始都需要从中央数据库中检索整个会话历史记录，这会产生网络传输延迟。为了减轻延迟，减少传输数据的大小至关重要。一个关键的优化是在将历史记录发送给**智能体**之前对其进行**过滤或压缩**。例如，您可以删除当前对话状态不再需要的旧的、不相关的函数调用输出。下一节将详细介绍几种压缩历史记录的策略，以有效地管理长上下文对话。

# 管理长上下文对话：权衡和优化 (Managing long context conversation: tradeoffs and optimizations)

在简化的架构中，会话是用户和**智能体**之间对话的不可变日志。然而，随着对话规模的扩大，对话的令牌使用量会增加。现代 LLM 可以处理长上下文，但仍存在限制，特别是对于延迟敏感的应用程序：

1. **上下文窗口限制：** 每个 LLM 都有一个它可以一次处理的最大文本量（上下文窗口）。如果对话历史记录超过此限制，API 调用将失败。

2. **API 成本（$）：** 大多数 LLM 提供商根据您发送和接收的令牌数量收费。较短的历史记录意味着更少的令牌和更低的每轮成本。

3. **延迟（速度）：** 向模型发送更多文本需要更长的处理时间，从而导致用户响应时间变慢。压缩可以保持**智能体**的快速和响应能力。

4. **质量：** 随着令牌数量的增加，由于上下文中的额外噪音和自回归错误，性能可能会变差。

管理与**智能体**的长时间对话可以比作一位精明的旅行者为长途旅行打包行李箱。行李箱代表**智能体**有限的上下文窗口，而衣服和物品则是对话中的信息片段。如果您只是试图将所有东西都塞进去，行李箱就会变得太重和杂乱无章，从而难以快速找到您需要的东西——就像上下文窗口超载会增加处理成本并减慢响应时间一样。另一方面，如果您打包得太少，您可能会冒着遗漏护照或保暖外套等必需品的风险，从而危及整个旅程——就像**智能体**可能会丢失关键上下文，导致不相关或不正确的答案一样。旅行者和**智能体**都在类似的限制下运行：成功取决于您能携带多少，而在于只携带您需要的。

压缩策略（Compaction strategies）会缩短长对话历史记录，压缩对话以适应模型的上下文窗口，从而降低 API 成本和延迟。随着对话变长，每次发送给模型的历史记录可能会变得过大。压缩策略通过在尝试保留最重要上下文的同时智能地修剪历史记录来解决此问题。

那么，如何知道从会话中删除哪些内容而不会丢失有价值的信息呢？策略范围从简单的截断到复杂的压缩：

- **保留最后 N 轮：** 这是最简单的策略。**智能体**只保留对话中最近的 N 轮（一个“滑动窗口”），并丢弃所有更旧的内容。
- **基于令牌的截断（Token-Based Truncation）：** 在将历史记录发送给模型之前，**智能体**会计算消息中的令牌，从最新的消息开始倒推。它会包含尽可能多的消息，但不超过预定义的令牌限制（例如 4000 个令牌）。所有更旧的内容都会被简单地截断。
- **递归摘要（Recursive Summarization）：** 对话的较旧部分被 AI 生成的摘要取代。随着对话的增长，**智能体**会定期使用另一个 LLM 调用来总结最旧的消息。然后，此摘要被用作历史记录的浓缩形式，通常前缀于更近期的、逐字的消息。

例如，您可以使用 ADK 应用程序的内置插件来限制发送给模型的上下文，从而在 ADK 中保留最后 N 轮。这不**会**修改存储在会话存储中的历史事件：

```
# 片段 3：使用 ADK 仅保留最后 N 轮的会话截断
from google.adk.apps import App
from google.adk.plugins.context_filter_plugin import ContextFilterPlugin

app = App(
    name='hello_world_app',
    root_agent=agent,
    plugins=[
  # Keep the last 10 turns and the most recent user query.
        ContextFilterPlugin(num_invocations_to_keep=10),
    ],
)
```

鉴于复杂的压缩策略旨在降低成本和延迟，关键是**在后台异步执行昂贵的操作（如递归摘要）并持久化结果**。“在后台”确保客户端无需等待，“持久化”确保昂贵的计算不会被过度重复。通常，**智能体**的记忆管理器负责生成和持久化这些递归摘要。**智能体**还必须记录哪些事件包含在压缩摘要中；这可以防止原始的、更冗长的事件不必要地发送给 LLM。

此外，**智能体**必须决定何时需要压缩。触发机制通常分为几个不同的类别：

- **基于计数的触发器**（即令牌大小或轮次计数阈值）：一旦对话超过某个预定义的阈值，就会进行压缩。这种方法对于管理上下文长度通常“足够好”。
- **基于时间的触发器：** 压缩不是由对话的大小触发，而是由缺乏活动触发。如果用户停止交互一段设定时间（例如 15 或 30 分钟），系统可以在后台运行压缩作业。
- **基于事件的触发器**（即语义/任务完成）：当**智能体**检测到特定任务、子目标或对话主题已结束时，它会决定触发压缩。

例如，您可以使用 ADK 的 `EventsCompactionConfig` 在配置的轮次后触发基于 LLM 的摘要：

```
# 片段 4：使用 ADK 进行摘要的会话压缩
from google.adk.apps import App
from google.adk.apps.app import EventsCompactionConfig

app = App(
```

```
    name='hello_world_app',
    root_agent=agent,
    events_compaction_config=EventsCompactionConfig(
        compaction_interval=5,
        overlap_size=1,
    ),
)
```

记忆生成是从冗长且嘈杂的数据源中提取持久知识的广泛能力。在本节中，我们介绍了从对话历史中提取信息的一个主要示例：**会话压缩**。压缩提炼了整个对话的逐字记录，提取关键事实和摘要，同时丢弃了对话填充物。在压缩的基础上，下一节将更广泛地探讨记忆生成和管理。我们将讨论创建、存储和检索记忆的各种方式，以建立**智能体**的长期知识。

# 记忆 (Memory)

记忆和会话共享着深刻的共生关系：**会话是生成记忆的主要数据源，而记忆是管理会话大小的关键策略**。一个记忆是从对话或数据源中提取的有意义信息的快照。它是一种浓缩的表示，可以保留重要的上下文，使其对未来的交互有用。一般来说，记忆会**跨会话持久保存**，以提供连续和个性化的体验。

作为一个专门的、解耦的服务，“记忆管理器”为多智能体互操作性奠定了基础。记忆管理器通常使用**与框架无关的数据结构**，例如简单的字符串和字典。这使得基于不同框架构建的智能体能够连接到单个记忆存储，从而创建一个任何连接的智能体都可以利用的共享知识库。

*注：有些框架也可能将会话或逐字对话称为“短期记忆”。对于本白皮书而言，记忆被定义为提取出的信息，而非逐轮对话的原始对话记录。*

存储和检索记忆对于构建复杂且智能的智能体至关重要。一个健壮的记忆系统通过解锁以下几项关键能力，将一个基本的聊天机器人转变为一个真正智能的智能体：

- **个性化（Personalization）**：最常见的用例是记住用户偏好、事实和过去的交互，以定制未来的响应。例如，记住用户最喜欢的运动队或在飞机上首选的座位，可以创造更有帮助和更个性化的体验。
- **上下文窗口管理（Context Window Management）**：随着对话变长，完整的历史记录可能会超过 LLM 的上下文窗口。记忆系统可以通过创建摘要或提取关键事实来压缩此历史记录，从而在不发送数千个令牌的情况下保留上下文。这降低了成本和延迟。

- **数据挖掘和洞察（Data Mining and Insight）**：通过分析许多用户存储的记忆（以聚合、保护隐私的方式），您可以从噪音中提取洞察。例如，零售聊天机器人可能会识别出许多用户正在询问特定产品的退货政策，从而标记出一个潜在的问题。
- **智能体自我改进和适应（Agent Self-Improvement and Adaptation）**：智能体通过创建关于自身表现的**程序性记忆**（procedural memories）来从以前的运行中学习——记录哪些策略、工具或推理路径带来了成功的成果。这使智能体能够建立一套有效的解决方案手册，从而随着时间的推移适应和改进其问题解决能力。

在 AI 系统中创建、存储和利用记忆是一个协作过程。堆栈中的每个组件——从最终用户到开发人员的代码——都扮演着独特的角色：

1. **用户（The User）**：提供记忆的原始源数据。在某些系统中，用户可以直接提供记忆（即通过表单）。

2. **智能体（开发人员逻辑）（The Agent (Developer Logic)）**：配置如何决定记住什么以及何时记住，协调对记忆管理器的调用。在简单的架构中，开发人员可以实现逻辑，使得记忆*总是*被检索并且*总是*被触发生成。在更高级的架构中，开发人员可能会实现**记忆即工具（memory-as-a-tool）**，由智能体（通过 LLM）决定何时应该检索或生成记忆。

3. **智能体框架（Agent Framework）（例如 ADK, LangGraph）**：提供记忆交互的结构和工具。框架充当管道。它定义了开发人员的逻辑如何访问对话历史记录并与记忆管理器交互，但它本身并不管理长期存储。它还定义了如何将检索到的记忆塞入上下文窗口。

4. **会话存储（Session Storage）（即 Agent Engine Sessions, Spanner, Redis）**：存储会话的逐轮对话。原始对话将被摄取到记忆管理器中以生成记忆。

5. **记忆管理器（Memory Manager）（例如 Agent Engine Memory Bank, Mem0, Zep）**：处理记忆的存储、检索和压缩。存储和检索记忆的机制取决于所使用的提供者。这是一个专门的服务或组件，它接受智能体识别的潜在记忆并处理其整个生命周期。

- **提取（Extraction）**：从源数据中提炼关键信息。
- **整合（Consolidation）**：策划记忆以合并重复的实体。
- **存储（Storage）**：将记忆持久保存到持久数据库。
- **检索（Retrieval）**：获取相关的记忆，为新的交互提供上下文。

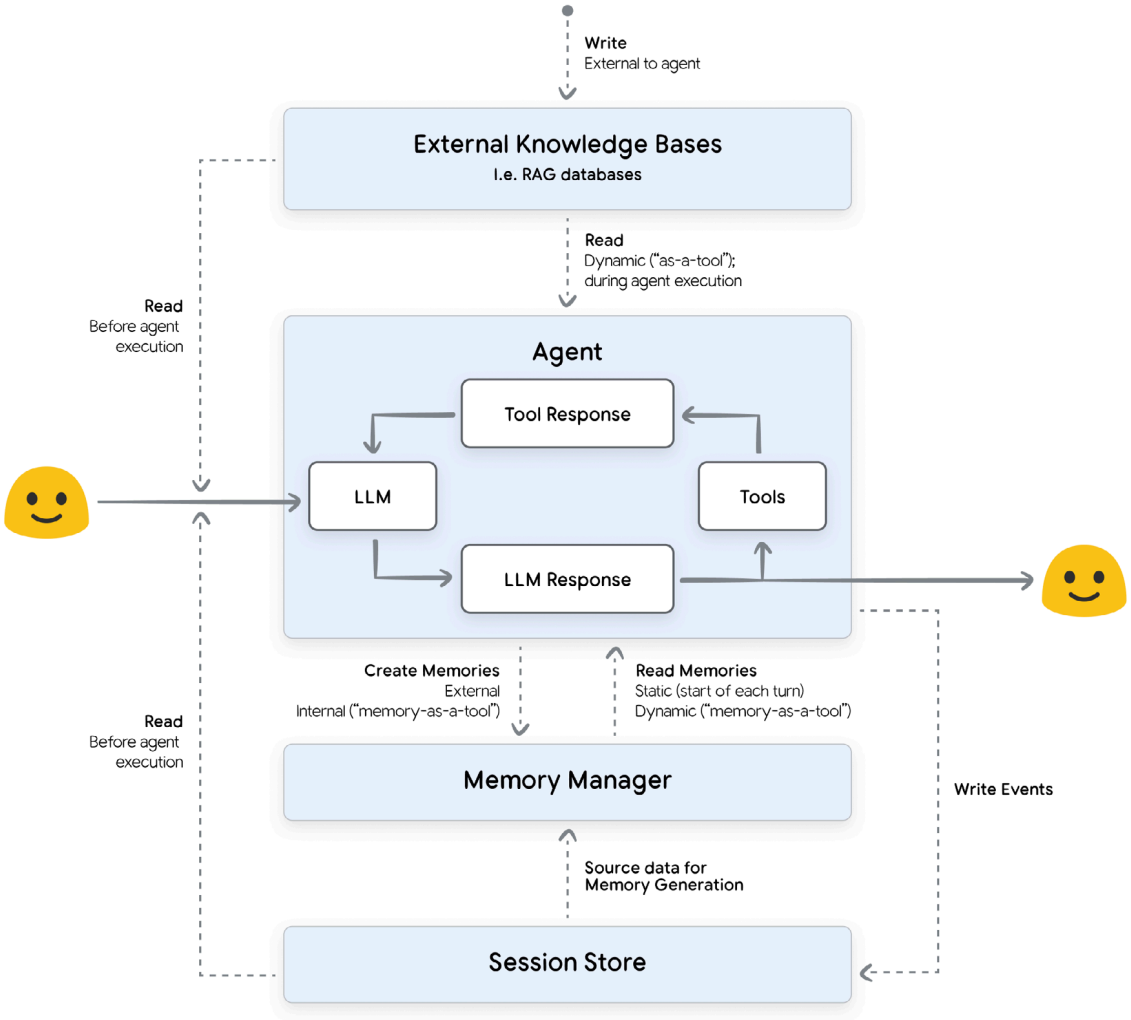



*图 5：会话、记忆和外部知识之间的信息流*

职责划分确保了开发人员可以专注于智能体独特的逻辑，而不必构建用于记忆持久化和管理的复杂底层基础设施。重要的是要认识到记忆管理器是一个**主动系统**，而不仅仅是一个被动的向量数据库。虽然它使用相似性搜索进行检索，但其核心价值 在于它能够随着时间的推移智能地**提取、整合和策划**记忆。托管记忆服务（如 Agent Engine Memory Bank）处理记忆生成和存储的整个生命周期，让您可以专注于智能体的核心逻辑。

这种检索能力也是记忆经常被拿来与另一个关键架构模式：**检索增强生成（RAG）**进行比较的原因。然而，它们是建立在不同的架构原则之上的，因为 RAG 处理静态的外部数据，而记忆策划动态的、用户特定的上下文。它们履行着两个不同且互补的角色：**RAG 使智能体成为事实方面的专家，而记忆使其成为用户方面的专家**。下表分解了它们的高层差异：

| RAG 引擎 (RAG Engines) | 记忆管理器 (Memory Managers) |
| --- | --- |

| | |
|---|---|
| **主要目标**：将外部的、事实性的知识注入上下文。 | **主要目标**：创建个性化和有状态的体验。智能体记住事实，随着时间的推移适应用户，并保持长期运行的上下文。 |
| **数据源**：静态的、预先索引的外部知识库（例如 PDF、维基、文档、API）。 | **数据源**：用户和智能体之间的对话。 |
| **隔离级别**：通常是共享的。知识库通常是一个全局的、只读的资源，所有用户都可以访问，以确保一致的、事实性的答案。 | **隔离级别**：高度隔离。记忆几乎总是按用户范围划分，以防止数据泄漏。 |
| **信息类型**：静态、事实性和权威性。通常包含领域特定数据、产品细节或技术文档。 | **信息类型**：动态且（通常）用户特定。记忆来源于对话，因此存在固有的不确定性。 |
| **写入模式**：批量处理；通过离线的、管理性操作触发。 | **写入模式**：事件驱动处理；以某种节奏触发（即每轮或在会话结束时）或记忆即工具（智能体决定生成记忆）。 |
| **读取模式**：RAG 数据几乎总是“即工具”检索。当智能体决定用户的查询需要外部信息时进行检索。 | **读取模式**：有两种常见的读取模式：* 记忆即工具：当用户的查询需要有关用户（或某些其他身份）的额外信息时进行检索。* 静态检索：记忆在每轮开始时总是被检索。 |
| **数据格式**：自然语言“块”（chunk）。 | **数据格式**：自然语言片段或结构化配置文件。 |
| **数据准备**：分块和索引：源文档被分解成更小的块，然后转换为嵌入并存储以供快速查找。 | **数据准备**：提取和整合：从对话中提取关键细节，确保内容没有重复或矛盾。 |

理解它们之间差异的一个有用方法是，将 RAG 视为智能体的**研究图书管理员**，将记忆管理器视为其**私人助理**。

研究图书管理员（RAG）在一个巨大的公共图书馆工作，图书馆里装满了百科全书、教科书和官方文件。当智能体需要一个既定事实——比如产品的技术规格或历史日期——它就会咨询图书管理员。图书管理员从这个静态的、共享的、权威的知识库中检索信息，以提供一致的、事实性的答案。图书管理员是世界事实方面的专家，但他们对提问的用户一无所知。

相比之下，私人助理（记忆）跟随智能体，并携带一本私密笔记本，记录与特定用户交互的每一个细节。这本笔记本是动态的、高度隔离的，包含个人偏好、过去的对话和不断变化的目标。当智能体需要回忆用户最喜欢的运动队或上周项目讨论的背景时，它会求助于助理。助理的专长不在于全球事实，而在于用户本身。

最终，一个真正智能的智能体需要两者。RAG 为其提供了关于世界的专家知识，而记忆则为其提供了对其所服务用户的专家理解。

下一节将通过检验记忆的核心组成部分来解构记忆的概念：它存储的信息类型、其组织模式、其存储和创建机制、其范围的战略定义，以及它对多模态与文本数据的处

理。

# 记忆类型 (Types of memory)

智能体的记忆可以根据信息的存储方式和捕获方式进行分类。这些不同类型的记忆协同工作，以创建对用户及其需求的丰富、上下文理解。在所有类型的记忆中，规则是：**记忆是描述性的，而不是预测性的**。

一个"记忆"是记忆管理器返回并被智能体用作上下文的一个**原子上下文片段**。虽然确切的模式可能有所不同，但单个记忆通常由两个主要组成部分组成：**内容**和**元数据**。

**内容（Content）与框架无关的**，使用任何智能体都可以轻松摄取的简单数据结构。内容可以是结构化或非结构化数据。**结构化记忆**包括通常以通用格式（如字典或JSON）存储的信息。它的模式通常由开发人员定义，而不是特定的框架。例如：`{"seat_preference": "Window"}`。**非结构化记忆**是自然语言描述，它捕获了更长交互、事件或主题的本质。例如："用户偏爱靠窗的座位"。

元数据（Metadata）提供关于记忆的上下文，通常存储为简单的字符串。这可以包括记忆的唯一标识符、记忆"所有者"的标识符，以及描述记忆内容或数据源的标签。

## 信息类型 (Types of information)

*除了基本结构之外，记忆还可以根据它们所代表的知识的基本类型进行分类。这种区别对于理解智能体如何使用记忆至关重要，它将记忆分为源自认知科学的两个主要功能类别：**陈述性记忆**（"知道是什么"）和**程序性记忆**（"知道如何做"）。*

- *陈述性记忆（Declarative memory）**是智能体关于事实、数据和事件的知识。它是智能体可以明确陈述或"声明"的所有信息。如果记忆是对"是什么"问题的回答，那就是陈述性的。此类别涵盖了一般世界知识（语义）和特定的用户事实（实体/情景）。
- *程序性记忆（Procedural memory）**是智能体关于技能和工作流程的知识。它通过隐含地演示如何正确执行任务来指导智能体的操作。如果记忆有助于回答"如何做"的问题——例如预订旅行的正确工具调用序列——它就是程序性的。

## 组织模式 (Organization patterns)

记忆创建后，接下来的问题是如何组织它。记忆管理器通常采用以下一种或多种模式来组织记忆：**集合（Collections）**、**结构化用户配置文件（Structured User Profile）**"**滚动摘要**"**（Rolling Summary）**。这些模式定义了单个记忆之间以及记忆与用户之间的关系。

**集合** 模式将内容组织成针对单个用户的多个独立、自然语言的记忆。每个记忆都是一个不同的事件、摘要或观察，尽管对于一个单一的高级主题，集合中可能存在多个记

忆。集合允许存储和搜索与特定目标或主题相关的大量、结构较松散的信息。

- *结构化用户配置文件（Structured user profile）**模式将记忆组织成关于用户的一组核心事实，就像一个不断用新的、稳定的信息更新的联系人卡片。它旨在快速查找基本的事实信息，如姓名、偏好和账户详情。

与结构化用户配置文件不同，**“滚动”摘要（“rolling” summary）**模式将所有信息整合到一个单一的、不断演变的记忆中，代表了整个用户-智能体关系的自然语言摘要。记忆管理器不会创建新的、单独的记忆，而是持续更新这个主文档。这种模式经常用于压缩长会话，在保留重要信息的同时管理总令牌数。

## 存储架构 (Storage architectures)

此外，存储架构是一个关键的决策，它决定了智能体检索记忆的速度和智能程度。架构的选择定义了智能体是擅长查找概念上相似的想法、理解结构化关系，还是两者兼而有之。

记忆通常存储在**向量数据库**和/或**知识图谱**中。向量数据库有助于查找与查询概念上相似的记忆。知识图谱将记忆存储为实体及其关系的网络。

- *向量数据库（Vector databases）**是最常见的方法，它允许基于语义相似性而非精确关键词进行检索。记忆被转换为嵌入向量，数据库查找与用户查询最接近的概念匹配。这在检索上下文和意义是关键的非结构化、自然语言记忆时表现出色（即“原子事实”）。
- *知识图谱（Knowledge graphs）**用于将记忆存储为实体（节点）及其关系（边）的网络。检索涉及遍历此图以查找直接和间接连接，从而允许智能体推理不同事实是如何链接的。它非常适合结构化的、关系型查询和理解数据中的复杂连接（即“知识三元组”）。

您还可以通过用向量嵌入丰富知识图谱的结构化实体，将这两种方法结合起来形成**混合方法**。这使得系统能够同时执行关系搜索和语义搜索。这提供了图谱的结构化推理和向量数据库的细微概念搜索，提供了两者的最佳选择。

## 创建机制 (Creation mechanisms)

我们还可以根据记忆是如何创建的来对其进行分类，包括信息是如何得出的。**显式记忆（Explicit memories）**是在用户给出直接命令让智能体记住某事时创建的（例如，“记住我的周年纪念日是 10 月 26 日”）。另一方面，**隐式记忆（implicit memories）**是在智能体从对话中推断和提取信息而没有直接命令时创建的（例如，“我的周年纪念日是下周。你能帮我找一个送给我伴侣的礼物吗？”）。

记忆还可以通过记忆提取逻辑是位于智能体框架的**内部**还是**外部**来区分。**内部记忆（Internal memory）**指的是直接构建在智能体框架中的记忆管理。它便于入门，

但通常缺乏高级功能。内部记忆可以使用外部存储，但生成记忆的机制是智能体内部的。

- *外部记忆（External Memory）**涉及使用一个专用于记忆管理的独立、专业服务（例如 Agent Engine Memory Bank, Mem0, Zep）。智能体框架向此外部服务进行 API 调用，以存储、检索和处理记忆。这种方法提供了更复杂的功能，如语义搜索、实体提取和自动摘要，将复杂的记忆管理任务卸载给一个专用工具。

## 记忆范围 (Memory scope)

您还需要考虑记忆描述的是谁或什么。这对于您使用哪个实体（即用户、会话或应用程序）来聚合和检索记忆有影响。

- *用户级（User-Level）**范围是最常见的实现，旨在为每个个体创建连续的、个性化的体验；例如，“用户偏爱中间座位”。记忆与特定的用户 ID 绑定，并持久保存在他们所有的会话中，允许智能体建立对其偏好和历史的长期理解。
- *会话级（Session-Level）**范围旨在压缩长对话；例如，“用户正在购买 2025 年 11 月 7 日至 2025 年 11 月 14 日期间纽约和巴黎之间的机票。他们偏爱直飞和中间座位”。它创建了从单个会话中提取的洞察的持久记录，允许智能体用一组简洁的关键事实替换冗长、占用令牌的对话记录。至关重要的是，这种记忆不同于原始的会话日志；它仅包含从对话中处理过的洞察，而不是对话本身，并且其上下文隔离于该特定会话。
- *应用程序级（Application-level）**范围（或全局上下文），是应用程序的所有用户都可以访问的记忆；例如，“代号 XYZ 指的是项目……”。此范围用于提供共享上下文、广播系统范围的信息或建立通用知识的基线。应用程序级记忆的一个常见用例是程序性记忆，它为智能体提供“操作指南”；这些记忆通常旨在帮助所有用户的智能体进行推理。至关重要的是，这些记忆必须清除所有敏感内容，以防止用户之间的数据泄漏。

## 多模态记忆 (Multimodal memory)

*"多模态记忆（Multimodal memory）"是一个关键概念，描述了智能体如何处理非文本信息，如图像、视频和音频。关键在于区分记忆派生自的数据（其来源）和记忆存储为的数据（其内容）。*

- *来自多模态来源的记忆（Memory from a multimodal source）**是最常见的实现。智能体可以处理各种数据类型——文本、图像、音频——但它创建的记忆是从该来源派生出的文本洞察。例如，智能体可以处理用户的语音备忘录来创建记忆。它不存储音频文件本身；相反，它转录音频并创建一个文本记忆，例如：“用户对最近的运输延迟表示沮丧”。

- *具有多模态内容的记忆（Memory with Multimodal Content）**是一种更高级的方法，记忆本身包含非文本媒体。智能体不只是描述内容；它直接存储内容。例如，用户可以上传一张图片并说“记住这个设计作为我们的标志”。智能体创建了一个直接包含图像文件，并链接到用户请求的记忆。

大多数当前的记忆管理器侧重于处理多模态来源，同时生成文本内容。这是因为为特定记忆生成和检索非结构化二进制数据（如图像或音频）需要专门的模型、算法和基础设施。将所有输入转换为一个通用的、可搜索的格式：文本，要简单得多。

例如，您可以使用 Agent Engine Memory Bank 从多模态输入生成记忆。输出记忆将是从内容中提取的文本洞察：

```
# 片段 5：Example memory generation API call for Agent Engine Memory Bank
from google.genai import types
client = vertexai.Client(project=..., location=...)
response = client.agent_engines.memories.generate(
  name=agent_engine_name,
  direct_contents_source={
     "events": [
          {
         "content": types.Content(
            role="user",
          parts=[
            types.Part.from_text(
                 "This is context about the multimodal input."
     ),
            types.Part.from_bytes(
               data=CONTENT_AS_BYTES,
               mime_type=MIME_TYPE
            ),
            types.Part.from_uri(
               file_uri="file/path/to/content",
               mime_type=MIME_TYPE
            )
    ])}]},
  scope={"user_id": user_id}
)
```

下一节将探讨记忆生成的机制，详细介绍两个核心阶段：从源数据中提取新信息，以及随后将该信息与现有记忆库进行整合。

## 记忆生成：提取和整合 (Memory Generation: Extraction and Consolidation)

记忆生成将原始对话数据自主地转化为结构化的、有意义的洞察，其功能类似于一个由 LLM 驱动的 **ETL（提取、转换、加载）管道**，旨在提取和压缩记忆。记忆生成的 ETL 管道将记忆管理器与 RAG 引擎和传统数据库区分开来。

记忆管理器无需开发人员手动指定数据库操作，而是使用 LLM 智能地决定何时添加、更新或合并记忆。这种**自动化**是记忆管理器的核心优势；它抽象了管理数据库内容、链接 LLM 调用和部署后台服务进行数据处理的复杂性。

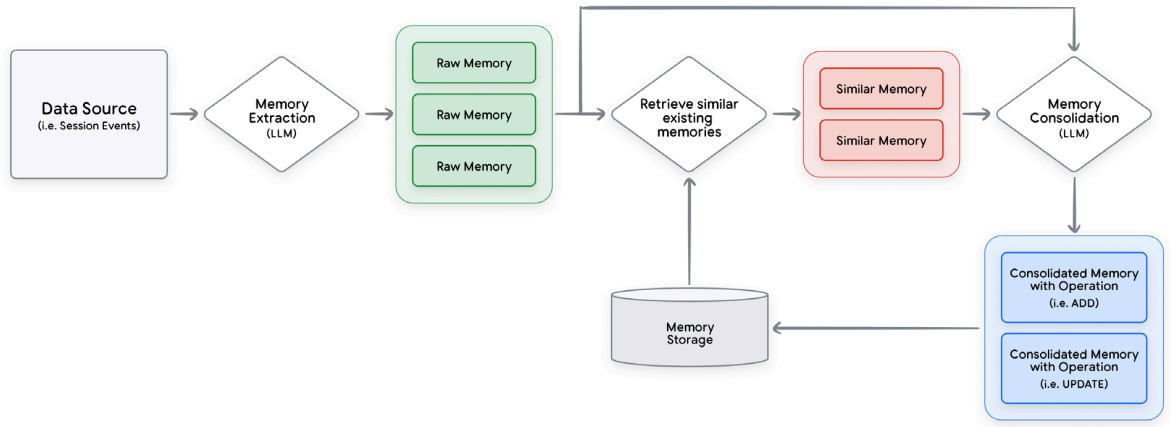



*图 6：记忆生成的高级算法流程，它从新数据源中提取记忆并与现有记忆进行整合*

虽然具体的算法因平台而异（例如 Agent Engine Memory Bank, Mem0, Zep），但记忆生成的高级流程通常遵循以下四个阶段：

1. **摄取（Ingestion）**：流程始于客户端向记忆管理器提供原始数据源，通常是对话历史记录。

2. **提取和过滤（Extraction & Filtering）**：记忆管理器使用 LLM 从源数据中提取有意义的内容。关键在于，这个 LLM 不会提取所有内容；它只会捕获符合预定义主题定义的信息。如果摄取的数据不包含任何与这些主题匹配的信息，则不会创建记忆。

3. **整合（Consolidation）**：这是最复杂的阶段，记忆管理器在此处理冲突解决和去重。它执行一个“自我编辑”过程，使用 LLM 将新提取的信息与现有记忆进行比较。为了确保用户的知识库保持连贯、准确并随着新信息而演变，管理器可以决定：

- 将新洞察**合并**到现有记忆中。
- 如果现有记忆现在已失效，则将其**删除**。

。如果主题是新颖的，则**创建**一个全新的记忆。

4. **存储（Storage）**：最后，新的或更新的记忆被持久化到持久存储层（如向量数据库或知识图谱），以便在未来的交互中可以检索到。

一个托管记忆管理器，如 Agent Engine Memory Bank，完全自动化了此管道。它们提供了一个单一、连贯的系统，将对话噪音转化为结构化知识，使开发人员能够专注于智能体逻辑，而不是自己构建和维护底层数据基础设施。例如，使用 Memory Bank 触发记忆生成只需要一个简单的 API 调用：

```
# 片段 6：使用 Agent Engine Memory Bank 生成记忆
from google.cloud import vertexai
client = vertexai.Client(project=..., location=...)
client.agent_engines.memories.generate(
  name="projects/.../locations/...reasoningEngines/...",
  scope={"user_id": "123"},
  direct_contents_source={
   "events": [...]
  },
  config={
   # Run memory generation in the background.
   "wait_for_completion": False
  }
)
```

记忆生成的流程可以比作一个辛勤的园丁照料花园。提取就像收到新的种子和树苗（来自对话的新信息）。园丁不会只是随意地将它们扔到地块上。相反，他们通过拔除杂草（删除冗余或冲突的数据）、修剪过长的枝条以改善现有植物的健康（细化和总结现有记忆），然后小心翼翼地将新树苗种在最佳位置，来执行**整合**。这种持续、周到的策划确保了花园随着时间的推移保持健康、有组织，并继续繁荣发展，而不是变成一片杂乱、无法使用的混乱。这种异步过程在后台发生，确保花园随时为下一次访问做好准备。

现在，让我们深入探讨记忆生成的两个关键步骤：提取和整合。

## 深入探讨：记忆提取 (Deep-dive: Memory Extraction)

记忆提取的目标是回答一个基本问题：“这次对话中的哪些信息足够有意义，可以成为记忆？” 这不是简单的摘要；它是一个有针对性的、智能的过滤过程，旨在将信号（重要事实、偏好、目标）与噪音（寒暄、填充文本）分开。

“有意义”不是一个普遍的概念；它完全由智能体的目的和用例定义。客户支持智能体需要记住的内容（例如订单号、技术问题）与个人健康教练需要记住的内容（例如长期目标、情绪状态）有着根本的不同。因此，**定制保留哪些信息是创建真正有效智能体的关键**。

记忆管理器的 LLM 通过遵循一套精心构建的、程序化的护栏和指令来决定提取什么，这些护栏和指令通常嵌入在一个复杂的系统提示中。这个提示通过向 LLM 提供一组**主题定义**来定义“有意义”的含义。通过**基于模式和模板的提取（schema and template-based extraction）**，LLM 会被赋予一个预定义的 JSON 模式或一个使用 LLM 功能（如结构化输出）的模板；LLM 被指示使用对话中相应的信息来构建 JSON。或者，通过**自然语言主题定义（natural language topic definitions）**，LLM 由主题的简单自然语言描述进行指导。

通过**少样本提示（few-shot prompting）**，LLM 通过示例被“展示”要提取的信息。提示包括几个输入文本示例和应该提取的理想、高保真记忆。LLM 从示例中学习所需的提取模式，使其对于难以用模式或简单定义描述的自定义或细微主题非常有效。

大多数记忆管理器开箱即用，通过寻找常见主题（如用户偏好、关键事实或目标）来工作。许多平台还允许开发人员定义自己的自定义主题，从而根据其特定领域定制提取过程。例如，您可以通过提供自己的主题定义和少样本示例来定制 Agent Engine Memory Bank 认为有意义并需要持久化的信息：

```
# 片段 7：定制 Agent Engine Memory Bank 认为有意义需要持久化的信息
from google.genai.types import Content, Part
# See https://cloud.google.com/agent-builder/agent-engine/memory-bank/set-up for more information.
memory_bank_config = {
  "customization_configs": [{
    "memory_topics": [
      { "managed_memory_topic": {"managed_topic_enum": "USER_PERSONAL_INFO" }},
      {
     "custom_memory_topic": {
          "label": "business_feedback",
          "description": """Specific user feedback about their experience at the coffee shop. This includes opinions on drinks, food, pastries, ambiance, staff friendliness, service speed, cleanliness, and any suggestions for improvement."""
        }
      }
```

```
    ],
   "generate_memories_examples": {
    "conversationSource": {
     "events": [
      {
       "content": Content(
          role="model",
                parts=[Part(text="Welcome back to The Daily Grind! We'd lo
ve to hear your feedback on your visit.")])
      },{
         "content": Content(
          role="user",
                parts=[Part(text= "Hey. The drip coffee was a bit lukewarm t
oday, which was a bummer. Also, the music was way too loud, I could barel
y hear my friend.")])
       }]
    },
    "generatedMemories": [
     {"fact": "The user reported that the drip coffee was lukewarm."},
     {"fact": "The user felt the music in the shop was too loud."}
    ]
  }
  }]
}
agent_engine = client.agent_engines.create(
   config={
      "context_spec": {"memory_bank_config": memory_bank_config }
   }
)
```

尽管记忆提取本身不是“摘要”，但算法可能会纳入摘要来提炼信息。为了提高效率，许多记忆管理器将对话的**滚动摘要**直接纳入记忆提取提示中。这种浓缩的历史提供了从最近的交互中提取关键信息所需的上下文。它消除了在每一轮中重复处理完整、冗长对话以保持上下文的需要。

一旦信息从数据源中提取出来，就必须通过整合来更新现有的记忆库，以反映新的信息。

## 深入探讨：记忆整合 (Deep-dive: Memory Consolidation)

在从冗长的对话中提取记忆之后，**整合**应将新信息整合到一个连贯、准确和不断演变的知识库中。它可以说是记忆生命周期中最复杂的阶段，将简单的事实集合转化为对用户的精选理解。如果没有整合，智能体的记忆很快就会变成一个嘈杂、矛盾且不可靠的、捕获到的所有信息的日志。这种“自我策划”通常由 LLM 管理，正是它将记忆管理器提升到了一个简单数据库之上的高度。

整合解决了源于对话数据的基本问题，包括：

- **信息重复（Information Duplication）**：用户可能在不同对话中以多种方式提及相同的事实（例如，“我需要一张去纽约的机票”，后来又说“我正计划去纽约旅行”）。一个简单的提取过程会创建两个冗余的记忆。
- **信息冲突（Conflicting Information）**：用户的状态会随着时间而变化。如果没有整合，智能体的记忆将包含矛盾的事实。
- **信息演变（Information Evolution）**：一个简单的事实可能会变得更加微妙。最初的记忆“用户对营销感兴趣”可能会演变为“用户正在领导一个专注于第四季度客户获取的营销项目”。
- **记忆相关性衰减（Memory Relevance Decay）**：并非所有记忆都会永远有用。智能体必须进行**遗忘**——主动删除旧的、过时的或低置信度的记忆，以保持知识库的相关性和效率。遗忘可以通过指示 LLM 在整合过程中服从较新的信息来实现，也可以通过“生存时间”（TTL）自动删除来实现。

整合过程是一个 LLM 驱动的工作流程，它将新提取的洞察与用户现有的记忆进行比较。首先，工作流程会尝试检索与新提取的记忆相似的现有记忆。这些现有记忆是整合的候选者。如果现有记忆被新信息所矛盾，它可能会被删除。如果它被增强，它可能会被更新。

其次，LLM 会被呈现现有记忆和新信息。它的核心任务是分析它们，并确定应该执行哪些操作。主要操作包括：

- **更新（UPDATE）**：用新的或纠正的信息修改现有记忆。
- **创建（CREATE）**：如果新洞察完全新颖且与现有记忆无关，则创建一个新记忆。
- **删除/失效（DELETE / INVALIDATE）**：如果新信息使旧记忆完全不相关或不正确，则删除或使其失效。

最后，记忆管理器将 LLM 的决定转化为更新记忆存储的事务。

# 记忆出处 (Memory Provenance)

*经典的机器学习格言“垃圾进，垃圾出”对于 LLM 来说更为关键，因为结果往往是“垃圾进，自信的垃圾出”。要让智能体做出可靠的决策，并让记忆管理器有效地整合记*

*忆，它们必须能够批判性地评估其自身记忆的质量。这种可信度直接来源于记忆的**出处（provenance）**——其起源和历史的详细记录。*

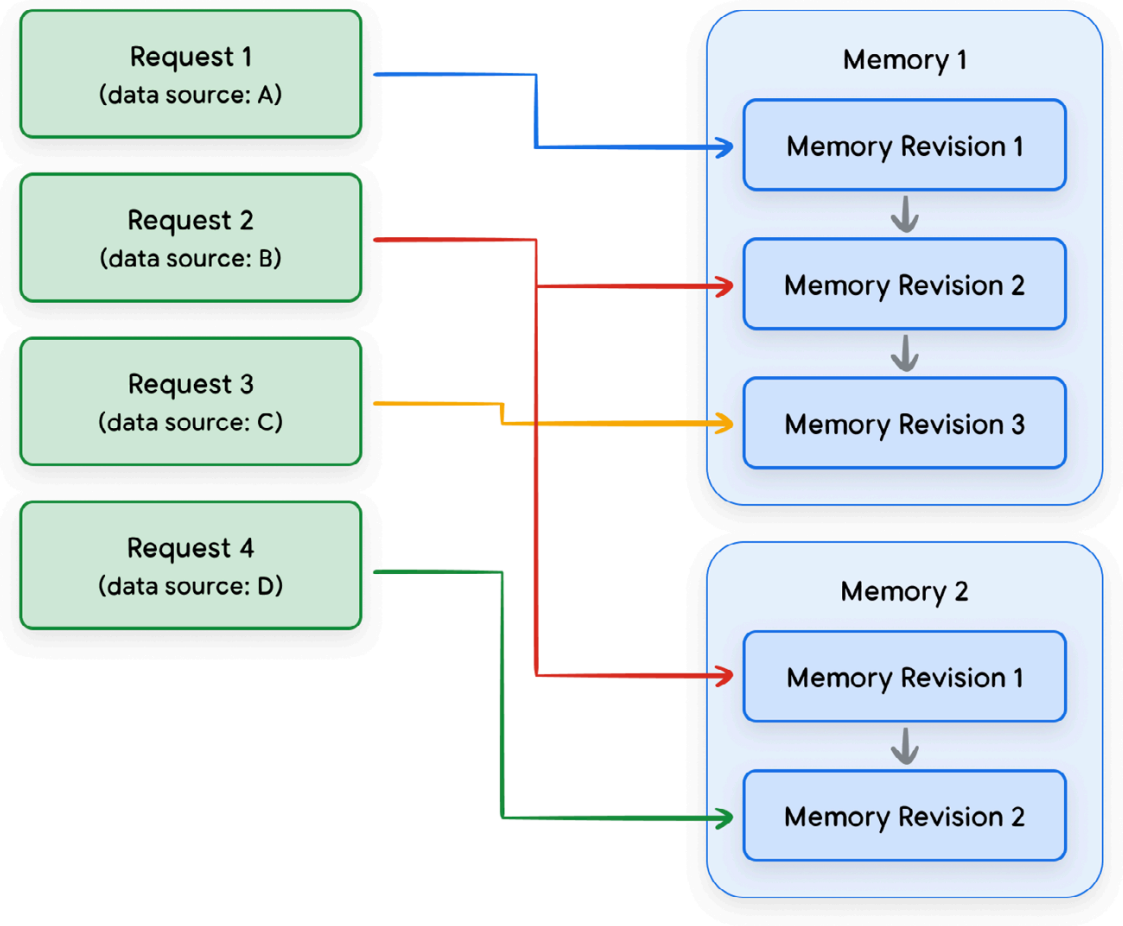



*图 7：数据源和记忆之间的信息流。一个记忆可以来源于多个数据源，而一个数据源可能贡献给多个记忆。*

记忆整合的过程——将来自多个来源的信息合并到单个、不断演变的记忆中——产生了跟踪其血缘关系的需要。如上图所示，单个记忆可能是多个数据源的混合，而单个来源可能被分割成多个记忆。

为了评估可信度，智能体必须跟踪每个来源的关键细节，例如其**起源（来源类型）年龄（"新鲜度"）**。这些细节至关重要，原因有二：它们决定了每个来源在记忆整合过程中的权重，并告知智能体在推理过程中应该在多大程度上依赖该记忆。

- *来源类型（The source type）**是决定信任的最重要因素之一。数据源分为三个主要类别：
- **引导数据（Bootstrapped Data）**：从内部系统（如 CRM）预加载的信息。这种高信任度数据可用于初始化用户的记忆，以解决冷启动问题，即为智能体从未交互过的用户提供个性化体验的挑战。

- **用户输入（User Input）**：这包括明确提供的数据（例如通过表单，这是高信任度的）或从对话中隐式提取的信息（通常不太值得信赖）。
- **工具输出（Tool Output）**：从外部工具调用返回的数据。通常不鼓励从工具输出生成记忆，因为这些记忆往往是脆弱和过时的，使这种来源类型更适合短期缓存。

## 在记忆管理期间考虑记忆血缘关系 (Accounting for memory lineage during memory management)

这种动态的、多源的记忆方法在管理记忆时产生了两个主要的操作挑战：冲突解决和删除派生数据。

记忆整合不可避免地会导致冲突，即一个数据源与另一个数据源冲突。记忆的出处允许记忆管理器为其信息来源建立信任等级。当来自不同来源的记忆相互矛盾时，智能体必须在**冲突解决策略**中使用此等级。常见策略包括优先考虑最受信任的来源、倾向于最新的信息，或寻找多个数据点的佐证。

管理记忆的另一个挑战发生在删除记忆时。一个记忆可以派生自多个数据源。当用户撤销对一个数据源的访问时，派生自该来源的数据也应被移除。删除被该来源"触及"的每个记忆可能过于激进。一种更精确但计算成本更高的方法是仅使用剩余的、有效的来源从头开始重新生成受影响的记忆。

除了静态出处细节之外，对记忆的**置信度（confidence）**必须演变。通过佐证，例如当多个受信任的来源提供一致的信息时，置信度会增加。然而，一个高效的记忆系统还必须通过**记忆修剪（memory pruning）**来主动策划其现有知识——这是一个识别并"遗忘"不再有用的记忆的过程。这种修剪可以由几个因素触发：

- **基于时间的衰减（Time-based Decay）**：记忆的重要性会随时间下降。两年前关于一次会议的记忆可能不如上周的记忆相关。
- **低置信度（Low Confidence）**：从微弱推断创建且从未被其他来源佐证的记忆可能会被修剪。
- **不相关性（Irrelevance）**：随着智能体对用户获得更复杂的理解，它可能会确定一些旧的、琐碎的记忆与用户当前的目标不再相关。

通过将反应性整合管道与前瞻性修剪相结合，记忆管理器确保了智能体的知识库不仅仅是迄今为止所说的所有内容的不断增长的日志。相反，它是一个经过策划的、准确且相关的用户理解。

## 在推理期间考虑记忆血缘关系 (Accounting for memory lineage during inference)

除了在策划记忆库内容时考虑记忆的血缘关系外，在**推理时**也应考虑记忆的可信度。智能体对记忆的置信度不应是静态的；它必须根据新信息和时间的流逝而演变。通过佐证，例如当多个受信任的来源提供一致的信息时，置信度会增加。反之，随着时间的推移，旧记忆变得过时，置信度会下降（或衰减），当引入矛盾信息时也会下降。最终，系统可以通过归档或删除低置信度的记忆来“遗忘”。这种动态置信度分数在推理时至关重要。记忆及其置信度分数不会向用户展示，而是被注入到提示中，使 LLM 能够评估信息可靠性并做出更细致的决策。

整个信任框架服务于智能体的内部推理过程。记忆及其置信度分数通常不会直接向用户展示。相反，它们被注入到系统提示中，允许 LLM 权衡证据、考虑其信息可靠性，并最终做出更细致和更值得信赖的决策。

## 触发记忆生成 (Triggering memory generation)

尽管一旦触发生成，记忆管理器就会自动化记忆提取和整合，但智能体仍必须决定何时应该尝试记忆生成。这是一个关键的架构选择，平衡了数据新鲜度与计算成本和延迟。此决策通常由智能体的逻辑管理，它可以采用几种触发策略。记忆生成可以基于各种事件启动：

- **会话完成（Session Completion）**：在多轮会话结束时触发生成。
- **轮次节奏（Turn Cadence）**：在特定轮次后运行该过程（例如每 5 轮）。
- **实时（Real-Time）**：在每一轮后生成记忆。
- **显式命令（Explicit Command）**：在用户直接命令时激活该过程（例如“记住这个”）。

触发器的选择涉及成本和保真度之间的直接权衡。频繁生成（例如实时）确保记忆高度详细和新鲜，捕获对话的每个细微差别。然而，这会带来最高的 LLM 和数据库成本，如果不当处理，可能会引入延迟。不频繁生成（例如在会话完成时）更具成本效益，但存在创建保真度较低的记忆的风险，因为 LLM 必须一次总结更大的对话块。您还需要小心，不要让记忆管理器多次处理相同的事件，因为这会引入不必要的成本。

### 记忆即工具 (Memory-as-a-Tool)

一种更复杂的做法是允许智能体自行决定何时创建记忆。在这种模式中，记忆生成被暴露为一个工具（即 `create_memory`）；工具定义应界定应被视为有意义的信息类型。然后，智能体可以分析对话，并在识别出有意义的信息需要持久化时，自主决定调用此工具。这将识别“有意义信息”的责任从外部记忆管理器转移到了智能体本身（因此也转移到了作为开发人员的您）。

例如，您可以使用 ADK 来做到这一点，将您的记忆生成代码打包成一个**工具**，当智能体认为对话值得持久化时，它会决定调用该工具。您可以将会话发送给 Memory Bank，Memory Bank 将从对话历史中提取和整合记忆：

```
# 片段 8：ADK 智能体使用自定义工具触发记忆生成。Memory Bank 将提取并整合记忆。
from google.adk.agents import LlmAgent
from google.adk.memory import VertexAiMemoryBankService
from google.adk.runners import Runner
from google.adk.tools import ToolContext

def generate_memories(tool_context: ToolContext):
    """Triggers memory generation to remember the session."""
    # Option 1: Extract memories from the complete conversation history usi
ng the
    # ADK memory service.
    tool_context._invocation_context.memory_service.add_session_to_memo
ry(
        session)

# Option 2: Extract memories from the last conversation turn.
    client.agent_engines.memories.generate(
        name="projects/.../locations/...reasoningEngines/...",
        direct_contents_source={
            "events": [
              {"content": tool_context._invocation_context.user_content}
            ]
        },
        scope={
            "user_id": tool_context._invocation_context.user_id,
            "app_name": tool_context._invocation_context.app_name
        },
        # Generate memories in the background
        config={"wait_for_completion": False}
    )
    return {"status": "success"}

agent = LlmAgent(
    ...,
    tools=[generate_memories]
)
```

```
runner = Runner(
    agent=agent,
    app_name=APP_NAME,
    session_service=session_service,
    memory_service=VertexAiMemoryBankService(
        agent_engine_id=AGENT_ENGINE_ID,
        project=PROJECT,
        location=LOCATION
    )
)
```

另一种方法是利用**内部记忆（internal memory）**，由智能体主动决定要从对话中记住什么。在此工作流程中，智能体负责提取关键信息。然后，这些提取的记忆可以选择性地发送到 Agent Engine Memory Bank，以便与用户现有的记忆进行整合：

```
# 片段 9：ADK 智能体使用自定义工具从对话中提取记忆并触发与 Agent Engin
e Memory Bank 的整合。与片段 8 不同，智能体负责提取记忆，而非 Memory
Bank。
def extract_memories(query: str, tool_context: ToolContext):
    """Triggers memory generation to remember information.

Args:
      query: Meaningful information that should be persisted about the user.
    """
    client.agent_engines.memories.generate(
        name="projects/.../locations/...reasoningEngines/...",
        # The meaningful information is already extracted from the conversati
on, so we
        # just want to consolidate it with existing memories for the same user.
        direct_memories_source={
            "direct_memories": [{"fact": query}]
        },
        scope={
            "user_id": tool_context._invocation_context.user_id,
            "app_name": tool_context._invocation_context.app_name
        },
        config={"wait_for_completion": False}
    )
```

```
    return {"status": "success"}

agent = LlmAgent(
    ...,
    tools=[extract_memories]
)
```

**后台操作 vs. 阻塞操作 (Background vs. Blocking Operations)**

记忆生成是一项昂贵的操作，需要 LLM 调用和数据库写入。对于生产中的智能体，记忆生成**几乎应该总是作为后台过程异步处理**。

在智能体向用户发送响应后，记忆生成管道可以并行运行，而不会阻塞用户体验。这种解耦对于保持智能体的快速和响应能力至关重要。阻塞（或同步）方法，即用户必须等待记忆被写入后才能收到响应，将导致用户体验慢得令人无法接受且令人沮丧。这要求记忆生成必须发生在架构上与智能体核心运行时分离的服务中。

# 记忆检索 (Memory Retrieval)

有了记忆生成机制，您的焦点就可以转移到关键的**检索**任务上。智能的检索策略对于智能体的性能至关重要，它包括关于应该检索哪些记忆以及何时检索它们的决策。

检索记忆的策略很大程度上取决于记忆是如何组织的。对于结构化用户配置文件，检索通常是查找完整配置文件或特定属性的简单过程。然而，对于记忆集合，检索是一个复杂得多的搜索问题。目标是从大量的非结构化或半结构化数据中发现最相关、概念上相关的信息。本节讨论的策略旨在解决记忆集合的这种复杂检索挑战。

记忆检索搜索当前对话中最相关的记忆。有效的检索策略至关重要；提供不相关的记忆可能会使模型感到困惑并降低其响应质量，而找到完美的上下文片段则可能带来非常智能的交互。核心挑战是在严格的延迟预算内平衡记忆的“有用性”。

高级记忆系统超越了简单的搜索，并通过多个维度对潜在记忆进行评分以找到最佳匹配。

- **相关性（Relevance）（语义相似性）**：此记忆与当前对话在概念上的关联程度如何？
- **时新性（Recency）（基于时间）**：此记忆是多久前创建的？
- **重要性（Importance）（显著性）**：此记忆的整体关键程度如何？与相关性不同，记忆的“重要性”可以在生成时定义。

仅依赖基于向量的相关性是一个常见的陷阱。相似性分数可能会浮现出概念上相似但陈旧或琐碎的记忆。最有效的策略是结合所有三个维度的分数的**混合方法**。

对于准确性至关重要的应用程序，可以使用**查询重写（query rewriting）**、**重排序（reranking）**或专门的检索器（specialized retrievers）**等方法来改进检索。然而，这些技术计算成本高昂，会增加显著的延迟，使其不适用于大多数实时应用程序。对于需要这些复杂算法且记忆不会迅速过时的场景，**缓存层**可以是一个有效的缓解措施。缓存允许检索查询的昂贵结果被临时存储，从而绕过后续相同请求的高延迟成本。

通过**查询重写**，可以使用 LLM 来改进搜索查询本身。这可能涉及将用户的模糊输入重写为更精确的查询，或将单个查询扩展为多个相关的查询，以捕获主题的不同方面。虽然这显著提高了初始搜索结果的质量，但它在流程开始时增加了额外的 LLM 调用延迟。

通过**重排序**，初始检索使用相似性搜索获取一组广泛的候选记忆（例如，前 50 个结果）。然后，LLM 可以重新评估和重新排序这个较小的集合，以产生更准确的最终列表。

最后，您可以使用**微调**来训练一个专门的检索器。然而，这需要访问标记数据并可能显著增加成本。

最终，最佳的检索方法始于更好的记忆生成。确保记忆库是高质量且没有不相关信息的，是保证任何检索到的记忆集都会有帮助的最有效方式。

## 检索时机 (Timing for retrieval)

检索的最终架构决策是**何时检索记忆**。一种方法是**主动检索（proactive retrieval）**，其中记忆在每轮开始时自动加载。这确保了上下文始终可用，但对于不需要记忆访问的轮次会引入不必要的延迟。由于记忆在单个轮次中保持静态，因此可以有效地缓存它们以减轻这种性能成本。

例如，您可以使用内置的 `PreloadMemoryTool` 或自定义回调在 ADK 中实现主动检索：

```
# 片段 10：使用内置工具或自定义回调在 ADK 中每轮开始时检索记忆
# Option 1: Use the built-in PreloadMemoryTool which retrieves memories with similarity search every turn.
agent = LlmAgent(
    ...,
    tools=[adk.tools.preload_memory_tool.PreloadMemoryTool()]
)

# Option 2: Use a custom callback to have more control over how memories are retrieved.
def retrieve_memories_callback(callback_context, llm_request):
```

```
    user_id = callback_context._invocation_context.user_id
    app_name = callback_context._invocation_context.app_name

response = client.agent_engines.memories.retrieve(
        name="projects/.../locations/...reasoningEngines/...",
        scope={
            "user_id": user_id,
            "app_name": app_name
        }
    )
    memories = [f"* {memory.memory.fact}" for memory in list(response)]
    if not memories:
        # No memories to add to System Instructions.
        return
    # Append formatted memories to the System Instructions
    llm_request.config.system_instruction += "\nHere is information that you
have about the user:\n"
    llm_request.config.system_instruction += "\n".join(memories)
    agent = LlmAgent(
     ...,
     before_model_callback=retrieve_memories_callback,
    )
```

或者，您可以使用**反应式检索（reactive retrieval）**（“记忆即工具”），其中智能体被赋予一个工具来查询其记忆，自行决定何时检索上下文。这更高效和健壮，但需要额外的 LLM 调用，增加了延迟和成本；然而，记忆只在必要时才被检索，所以延迟成本的发生频率较低。此外，智能体可能不知道是否存在相关信息可供检索。但是，可以通过让智能体了解可用的记忆类型（例如，如果您使用自定义工具，可以在工具的描述中说明）来缓解这种情况，从而对何时查询做出更明智的决定。

```
# 片段 11：配置您的 ADK 智能体以决定何时使用内置或自定义工具检索记忆
# Option 1: Use the built-in LoadMemory.
agent = LlmAgent(
    ...,
    tools=[adk.tools.load_memory_tool.LoadMemoryTool()],
)

# Option 2: Use a Custom tool where you can describe what type of inform
```

```
ation  # might be available.
def load_memory(query: str, tool_context: ToolContext):
    """Retrieves memories for the user.

The following types of information may be stored for the user:
    * User preferences, like the user's favorite foods.
    ...
    """
    # Retrieve memories using similarity search.
    response = tool_context.search_memory(query)
    return response.memories

agent = LlmAgent(
    ...,
    tools=[load_memory],
)
```

# 使用记忆进行推理 (Inference with Memories)

一旦检索到相关的记忆，最后一步就是将它们策略性地放置到模型的上下文窗口中。这是一个关键过程；记忆的放置可以显著影响 LLM 的推理、影响运营成本，并最终决定最终答案的质量。

记忆主要通过附加到**系统指令（system instructions）**或注入到**对话历史（conversation history）**中来呈现。在实践中，**混合策略**通常是最有效的。将系统提示用于应始终存在的稳定、全局记忆（如用户配置文件）。否则，使用对话注入或记忆即工具来处理仅与对话的即时上下文相关的瞬态、情景记忆。这平衡了对持久上下文的需求与即时信息检索的灵活性。

## 系统指令中的记忆 (Memories in the System Instructions)

使用记忆进行推理的一个简单选择是将其附加到系统指令中。此方法通过将检索到的记忆与前导语一起直接附加到系统提示中，将其框定为整个交互的基础上下文，从而保持对话历史的整洁。例如，您可以使用 Jinja 动态地将记忆添加到您的系统指令中：

```
# 片段 12：使用检索到的记忆构建您的系统指令
from jinja2 import Template

template = Template(""" {{ system_instructions }}}
```

```
<MEMORIES>
Here is some information about the user:
{% for retrieved_memory in data %}
* {{ retrieved_memory.memory.fact }}
{% endfor %}
</MEMORIES>
""")

prompt = template.render(
    system_instructions=system_instructions,
    data=retrieved_memories
)
```

将记忆包含在系统指令中赋予了记忆**很高的权威性**，将上下文与对话清晰地分离，并且是稳定、“全局”信息（如用户配置文件）的理想选择。然而，存在**过度影响**的风险，即智能体可能会试图将每个话题都与其核心指令中的记忆联系起来，即使不合适。

*这种架构模式引入了几个限制。首先，它要求智能体框架支持在每次 LLM 调用之前动态构建系统提示；此功能并非总是易于支持。此外，该模式与“记忆即工具”不兼容，因为系统提示必须在 LLM 决定调用记忆检索工具之前最终确定。最后，它对非文本记忆处理不佳。大多数 LLM 只接受文本作为系统指令，这使得将图像或音频等多模态内容直接嵌入到提示中变得具有挑战性。*

## 对话历史中的记忆 (Memories in the Conversation History)

在这种方法中，检索到的记忆直接注入到逐轮对话中。记忆可以放置在完整对话历史之前，或紧邻最新用户查询之前。

然而，这种方法可能会产生**噪音**，增加令牌成本，并且如果检索到的记忆不相关，可能会使模型感到困惑。它的主要风险是**对话注入**，即模型可能会错误地将记忆视为对话中实际说过的话。您还需要更仔细地处理注入到对话中的记忆的视角；例如，如果您使用“user”角色和用户级记忆，记忆应以第一人称视角编写。

将记忆注入对话历史的一个特例是通过工具调用检索记忆。记忆将作为工具输出的一部分直接包含在对话中。

```
# 片段 13：将记忆作为工具检索，它直接将记忆插入对话中
def load_memory(query: str, tool_context: ToolContext):
    """Loads memories into the conversation history..."""
```

```
    response = tool_context.search_memory(query)
    return response.memories

agent = LlmAgent(
    ...,
    tools=[load_memory],
)
```

# 程序性记忆 (Procedural memories)

本白皮书主要关注**陈述性记忆（declarative memories）**，这一重点反映了当前商业记忆的格局。大多数记忆管理平台也都是为这种陈述性方法设计的，擅长提取、存储和检索“是什么”——事实、历史和用户数据。

然而，这些系统并未设计用于管理**程序性记忆**，后者是改进智能体工作流程和推理的机制。存储“如何做”不是一个信息检索问题；它是一个推理增强问题。管理这种“知道如何做”需要一个完全独立且专业的算法生命周期，尽管其高级结构相似：

1. **提取（Extraction）**：程序性提取需要专门的提示，旨在从成功的交互中提炼出可重用的策略或“手册”，而不是仅仅捕获一个事实或有意义的信息。

2. **整合（Consolidation）**：虽然陈述性整合合并相关事实（“是什么”），但程序性整合策划工作流程本身（“如何做”）。这是一个主动的逻辑管理过程，专注于将新的成功方法与现有的“最佳实践”相结合，修补已知计划中的有缺陷步骤，并修剪过时或无效的程序。

3. **检索（Retrieval）**：目标不是检索数据来回答问题，而是检索一个计划来指导智能体如何执行复杂的任务。因此，程序性记忆可能与陈述性记忆具有不同的数据模式。

智能体这种“自我演变”逻辑的能力自然会让人联想到一种常见的适应方法：**微调（fine-tuning）**——通常通过人类反馈强化学习（RLHF）来实现。虽然这两个过程都旨在改善智能体的行为，但它们的机制和应用有着根本的不同。微调是一个相对缓慢的离线训练过程，会改变模型权重。程序性记忆提供了一种快速的**在线适应**，通过动态地将正确的“手册”注入到提示中，通过**上下文内学习**来指导智能体，而无需任何微调。

# 测试和评估 (Testing and Evaluation)

现在您有了一个启用记忆的智能体，您应该通过全面的质量和评估测试来验证您的启用记忆的智能体的行为。评估智能体的记忆是一个多层次的过程。评估要求验证智能体是否记住了正确的事情（质量），它是否能在需要时找到这些记忆（检索），以及使

用这些记忆是否真的有助于它完成目标（任务成功）。虽然学术界侧重于可复现的基准，但行业评估则侧重于记忆如何直接影响生产智能体的性能和可用性。

- *记忆生成质量指标（Memory generation quality metrics）**评估记忆本身的内容，回答以下问题："智能体是否记住了正确的事情？" 这通常通过将智能体生成的记忆与手动创建的"黄金集"理想记忆进行比较来衡量。
- **精确率（Precision）**：在智能体创建的所有记忆中，有多少百分比是准确且相关的？ 高精确率可以防止"过于热衷"的记忆系统用不相关的噪音污染知识库。
- **召回率（Recall）**：在它应该从来源中记住的所有相关事实中，它捕获了多少百分比？ 高召回率确保智能体不会遗漏关键信息。
- **F1-分数（F1-Score）**：精确率和召回率的调和平均值，提供了一个单一的、平衡的质量衡量标准。
- *记忆检索性能指标（Memory retrieval performance metrics）**评估智能体在正确时间找到正确记忆的能力。
- **Recall@K**：当需要记忆时，是否在检索到的前 'K' 个结果中找到了正确的记忆？ 这是检索系统准确性的主要衡量标准。
- **延迟（Latency）**：检索处于智能体响应的"热路径"上。整个检索过程必须在严格的延迟预算内执行（例如，低于 200 毫秒），以避免降低用户体验。
- *端到端任务成功指标（End-to-End task success metrics）**是最终的测试，回答以下问题："记忆是否真的有助于智能体更好地完成其工作？" 这通过评估智能体在使用其记忆执行下游任务时的性能来衡量，通常使用 LLM"裁判"将智能体的最终输出与黄金答案进行比较。裁判确定智能体的答案是否准确，有效地衡量了记忆系统对最终结果的贡献程度。

评估不是一次性事件；它是持续改进的引擎。上述指标提供了识别弱点和系统地增强记忆系统所需的数据。这个迭代过程涉及建立基线、分析故障、调整系统（例如，优化提示、调整检索算法）和重新评估以衡量变化的影响。

虽然上述指标侧重于质量，但生产就绪性也取决于性能。对于每个评估领域，衡量底层算法的延迟及其在负载下扩展的能力至关重要。在"热路径"上检索记忆可能有一个严格的、亚秒级的延迟预算。生成和整合虽然通常是异步的，但必须有足够的吞吐量来跟上用户需求。最终，一个成功的记忆系统必须是智能、高效且健壮的，以用于实际应用。

# 记忆的生产考量 (Production considerations for Memory)

除了性能之外，将启用记忆的智能体从原型过渡到生产需要关注企业级的架构问题。这一转变引入了对可扩展性、弹性和安全性的关键要求。生产级系统不仅必须设计得智能，还必须具备企业级的健壮性。

为了确保用户体验永远不会被计算成本高昂的记忆生成过程阻塞，健壮的架构必须将记忆处理与主应用程序逻辑解耦。虽然这是一种事件驱动的模式，但它通常是通过对专用记忆服务的直接、非阻塞 API 调用来实现的，而不是通过自行管理的消息队列。流程如下：

1. **智能体推送数据（Agent pushes data）**：在发生相关事件（例如会话结束）后，智能体应用程序向记忆管理器发出**非阻塞 API 调用**，“推送”原始源数据（如对话记录）以供处理。

2. **记忆管理器在后台处理（Memory manager processes in the background）**：记忆管理器服务立即确认请求，并将生成任务放入其内部、托管的队列中。然后，它全权负责异步的繁重工作：进行必要的 LLM 调用以提取、整合和格式化记忆。管理器可能会延迟处理事件，直到经过一定时间的不活动期。

3. **记忆被持久化（Memories are persisted）**：该服务将最终的记忆——这可能是新条目或对现有条目的更新——写入专用的、持久的数据库。对于托管记忆管理器，存储是内置的。

4. **智能体检索记忆（Agent retrieves memories）**：然后，主智能体应用程序可以在需要为新的用户交互检索上下文时，直接查询此记忆存储。

这种基于服务、非阻塞的方法确保了记忆管道中的故障或延迟不会直接影响面向用户的应用程序，使系统更具弹性。它还为在线（实时）生成（适用于对话新鲜度）和离线（批量）处理（适用于从历史数据填充系统）之间的选择提供了依据。

随着应用程序的增长，记忆系统必须在没有故障的情况下处理高频事件。鉴于并发请求，当多个事件尝试修改相同的记忆时，系统必须**防止死锁或竞态条件**。您可以使用事务性数据库操作或乐观锁来缓解竞态条件；但是，当多个请求试图修改相同的记忆时，这可能会引入排队或限制。健壮的**消息队列**对于缓冲大量事件和防止记忆生成服务被压垮至关重要。

记忆服务还必须对瞬时错误具有弹性（**故障处理**）。如果 LLM 调用失败，系统应使用带指数退避的重试机制，并将持久性故障路由到死信队列进行分析。

对于全球性应用，记忆管理器必须使用具有内置**多区域复制**的数据库，以确保低延迟和高可用性。客户端复制是不可行的，因为整合需要数据的单一、事务一致的视图来防止冲突。因此，记忆系统必须在内部处理复制，向开发人员呈现一个单一的、逻辑上的数据存储，同时确保底层知识库是全球一致的。

托管记忆系统，如 Agent Engine Memory Bank，应该帮助您解决这些生产考量，从而让您可以专注于核心智能体逻辑。

### 隐私和安全风险 (Privacy and security risks)

记忆源自并包含用户数据，因此它们需要**严格的隐私和安全控制**。一个有用的类比是，将系统的记忆视为由专业档案管理员管理的**安全企业档案**，档案管理员的工作是保护公司的同时保留有价值的知识。

该档案的首要规则是**数据隔离（data isolation）**。正如档案管理员绝不会将来自不同部门的机密文件混在一起一样，记忆必须在用户或租户级别进行严格隔离。服务于一个用户的智能体绝不能访问另一个用户的记忆，这需要使用限制性访问控制列表（ACLs）来强制执行。此外，用户必须对其数据拥有程序化控制权，并明确选择退出记忆生成或请求从档案中删除其所有文件。

在归档任何文档之前，档案管理员会执行关键的安全步骤。首先，他们会细致地检查每一页以**编辑敏感的个人身份信息（PII）**，确保在不产生责任的情况下保存知识。其次，档案管理员经过培训，能够发现和丢弃伪造或故意误导的文档——这是对记忆投毒（memory poisoning）的防范。同样，系统必须在将信息提交到长期记忆之前验证和清理信息，以防止恶意用户通过提示注入破坏智能体的持久知识。系统必须包括像 Model Armor 这样的防护措施，以在将信息提交到长期记忆之前验证和清理信息。

此外，如果多个用户共享同一组记忆，存在**数据泄露风险**，例如程序性记忆（用于教智能体如何做某事）。例如，如果一个用户的程序性记忆被用作另一个用户的示例——就像在全公司范围内共享备忘录一样——档案管理员必须首先执行严格的**匿名化**，以防止敏感信息泄漏到用户边界之外。

# 总结 (Conclusion)

本白皮书探讨了上下文工程（Context Engineering）这一学科，重点关注其两个核心组成部分：**会话（Sessions）记忆（Memory）**。从一个简单的对话轮次到一段持久的、可操作的智能信息，这一过程由这种实践所管辖，它涉及将所有必要信息——包括对话历史记录、记忆和外部知识——动态地组装到 LLM 的上下文窗口中。整个过程依赖于两个不同但相互关联的系统之间的相互作用：即时会话和长期记忆。

**会话**管理着“当下”，充当单个对话的低延迟、按时间顺序排列的容器。它的主要挑战是性能和安全性，要求低延迟访问和严格隔离。为了防止上下文窗口溢出和延迟，您必须使用提取技术，如 基于令牌的截断（token-based truncation）**或**递归摘要（recursive summarization）来压缩会话历史记录或单个请求有效载荷中的内容。此外，安全性至关重要，要求在会话数据持久化之前对 **PII（个人身份信息）进行编辑（redaction）**。

**记忆**是长期个性化的引擎，也是跨多个会话实现持久性的核心机制。它超越了 RAG（检索增强生成）（它使**智能体**成为**事实**方面的专家），使**智能体**成为**用户**方面的专家。记忆是一个**主动的、由 LLM 驱动的 ETL 管道**——负责**提取（extraction）、整合（consolidation）和检索（retrieval）**——它从对话历史记录中提炼出最重要的信息。通过**提取**，系统将最关键的信息提炼成关键的记忆点。在此之后，**整合**会策划新信息并将其与现有语料库集成，解决冲突并删除冗余数据，以确保知识库的连贯性。为了保持流畅的用户体验，**智能体**做出响应后，记忆生成必须作为**异步后台进程运行**。通过跟踪 出处（provenance）**并采用针对**记忆投毒（memory poisoning）等风险的防护措施，开发人员可以构建值得信赖、具有适应性、并能真正与用户一起学习和成长的助理。