亚马逊云科技

亚马逊云科技
AGENTIC AI 应用构建
实践指南

目录
CONTENTS

导读

迎接 Agentic AI 时代的技术革命

在人工智能飞速发展的今天，Agentic AI 正成为下一代生成式 AI 技术的核心范式。它超越了传统 AI 的被动响应模式，通过赋予 AI 自主规划、记忆、工具调用及协作能力，使其能够像人类一样主动理解、分解并解决复杂问题。从提升企业效率的垂类 Agent 到多 Agent 协同的生态系统，Agentic AI 正在重塑人机交互的边界，并为各行各业带来前所未有的智能化变革。

本文旨在系统性地梳理 Agentic AI 的技术框架、应用场景及落地路径。第 1 章深入解析 Agentic AI 的核心技术原理，包括其依赖的规划、记忆、工具调用等能力，并分类探讨了不同自主程度的 Agent 技术栈，以及介绍了 Agent 的应用形态，包括通用 Agent 与垂类 Agent；第 2 章提供实践指南，从专用的 Agent（Amazon Q），到开箱即用的托管服务（如 Amazon Bedrock Agents），再到完全自建 Agent 的方案设计，进行了详细的实践介绍；第 3 章则通过公开的客户案例，验证 Agentic AI 在优化流程、加速决策中的实际价值；第 4 章则是对本文的总结和展望，需要特别指出的是，本文在 2025 年 6 月发布，而 Agentic AI 的技术发展非常迅速，对于更新的 Agentic AI 的技术动向，可以访问亚马逊云科技的主页来了解最新的信息。

无论您是技术开发者、企业决策者还是 AI 研究者，本文将帮助您理解 Agentic AI 的底层逻辑，掌握其落地方法论，并预见这一技术如何驱动未来的智能化浪潮。让我们共同探索 Agentic AI 如何成为人类能力的延伸，开启人机协作的新篇章。

Agentic AI 概述

1

1.1 Agentic AI 是什么

Agentic AI 的概念由 AI Agent（也称为智能体）发展而来。AI Agent 技术并非一个全新的概念，其理论基础可以追溯到 20 世纪 90 年代初的人工智能研究。早期的 Agent 系统主要基于规则引擎、专家系统和符号逻辑构建，受限于当时计算能力和算法的局限性，只能在特定的、结构化的环境中发挥作用。大型语言模型（LLM）的出现引发了 Agent 技术的根本性变革，从面向过程的架构转变为面向目标的架构。现代 Agent 不再局限于单一规则或算法，而是能够理解自然语言指令，进行复杂推理，并通过灵活调用各类工具来实现目标。

在当今大语言模型的环境下，Agentic AI 是利用 AI 技术来推理、规划和代表人类或系统完成任务的自主软件系统。它们可以从头到尾执行各种操作，比如进行代码评审、编撰研究报告、处理申报、规划行程或管理企业应用程序。这些 Agent 具有反复思考的能力，包括评估结果、调整方法并继续朝着既定目标努力。它们不仅回答问题，还通过探索和细化的过程来解决问题。

大语言模型凭借其强大的自然语言处理能力，为 Agent 提供了以下关键能力：

通用语言的理解能力	大语言模型能够理解和处理自然语言指令，使 Agent 能够理解人类用自然语言表达的复杂任务描述，而无需进行专门的编程以及定义规则。
强大的推理能力	大语言模型通过预训练获得的广泛知识和推理能力，使 Agent 能够在问题描述处于模糊不清的情况下，进行因果推理，并在新的上下文环境中应用已经构建的知识。
自主规划能力	结合思维链（Chain-of-Thought）、树状思维（Tree-of-Thoughts）等技术，大语言模型驱动的 Agent 能够将复杂任务自行分解为子任务，并对这些子任务进行有序规划和执行。
工具使用能力	现代大语言模型具备了函数调用（Function Calling）的能力，能够识别何时需要调用函数，并正确构造 API 调用参数，使用外部工具来完成任务，从而极大的拓展了 Agent 的能力边界。

由于技术的进步（如基础大语言模型具有增强的推理能力）、不断提升的性价比以及能够安全可靠的集成数据的基础设施，从而推动了 Agentic AI 的飞跃发展。借助先进的开发工具，简化了部署 Agent 的过程，使得各种规模的企业现在都可以实施这些 Agentic AI 系统。根据 Gartner 的预测，到 2028 年，企业软件应用中将有 33% 集成 Agentic AI 功能（来源：Gartner 研究报告《2025 年顶级战略技术趋势》，2024 年 10 月发布），日常工作决策中的 15% 将由 Agentic AI 系统自主完成（来源：Gartner 研究报告《顶级战略技术趋势：Agentic AI——用户体验的演变》，2025 年 2 月发布）。

1.2 Agentic AI 的技术原理

现代 Agentic AI 是一种智能系统，能够感知环境、做出决策并采取行动以实现特定目标。与传统的 AI 系统相比，Agent 强调自主性、交互性和持续学习能力。当前的 Agent 系统通常由四个核心组成部分构建：大型语言模型作为推理引擎、规划能力、记忆机制和工具使用能力。这四个组件相互协作，形成了一个完整的 Agent 系统架构，使其能够理解复杂指令、制定执行计划、保持上下文连贯性，并与外部系统和资源交互。

1.2.1 大型语言模型作为推理引擎

大型语言模型是现代 Agentic AI 系统的核心，作为推理引擎驱动 Agent 的智能行为。大语言模型在 Agentic AI 架构中的核心功能体现在自然语言理解从而将非结构化的自然语言转化为系统可理解的意图和目标、推理与决策（进行逻辑分析、因果推断和假设验证）、预训练阶段获取的广泛世界知识的应用、对话历史维护从而理解上下文并进行连贯的多轮对话。

1.2.2 规划能力（Planning）

规划能力是 Agentic AI 系统的关键组成部分，使其能够分解复杂问题、设计解决方案路径并有条理地执行多步骤任务，展现出很高的灵活性和适应性。在实现规划能力的过程中，通常都会利用到思维链（Chain of Thought, CoT）技术，这是一种提示词技术，引导大语言模型生成一系列中间的推理步骤，而不是直接产生最终答案。这种技术模拟人类的逐步思考和推理过程，极大提升了复杂任务的解决质量。

通过在 Agent 的编排提示中嵌入类似的思维链指令，可以显著提升 Agent 在复杂任务处理中的表现。在 Agentic AI 系统中，思维链规划主要应用于以下场景：

- 任务拆解：将复杂任务拆解为有序的子任务，便于逐步依次执行。
- 推理验证：通过展示中间推理过程，验证推理的正确性和完整性。
- 决策透明度：使 Agent 的决策过程可见，增强可解释性和可信度。

ReAct（Reasoning and Acting）框架是目前常用的 Agent 规划框架之一。该框架通过思维链的方式，引导模型将复杂问题进行拆分，一步一步地进行推理（Reasoning）和行动（Action，比如工具调用、信息检索等），同时还引入了观察（Observation）环节，在每次行动（Action）之后，都会先观察（Observation）当前现状并了解环境变化，根据观察结果进行下一步新的思考和推理（reasoning），从而形成闭环反馈机制。这种把推理（Reasoning）和行动（Action）结合起来，通过交替进行思考和行动，在处理复杂任务时展现出更高的灵活性和适应性。通过使用 ReAct 框架，可以让大语言模型根据当前状况进行推理，然后采取行动与外界环境互动。

1.2.3 记忆机制（Memory）

记忆机制是 Agentic AI 系统的关键组成部分，使其能够存储和检索过去的交互信息、知识和经验，从而实现上下文感知的连续对话和长期学习。通过记忆机制，使 Agent 能够维持长时间的一致性互动。记忆机制分为短期记忆和长期记忆。

短期记忆主要负责维护当前对话或任务中的上下文信息，是 Agent 保持对话连贯性的基础。短期记忆通常在内存和提示词中存储对话历史数据，从而记录用户的输入信息和 Agent 响应的完整或摘要信息。随着对话进行，提示词的上下文窗口会面临容量限制。为了有效管理会话上下文，通常会保留最近 N 轮对话并丢弃较早的内容，或者保留对话中的关键信息并压缩或删除不重要的细节，也可以定期生成对话历史摘要，用摘要替代详细历史。

长期记忆扩展了 Agent 的能力范围，使其能够存储和检索超出单次会话范围的信息，形成持久化的知识库。长期记忆系统在实现中，可以将信息以结构化形式进行存储，比如存储在关系型数据库或者 NoSQL 数据库中，便于高效检索。也通常会选择将文本信息转换为语义向量并存储在向量数据库中，以及支持相似性搜索。检索增强生成（RAG：Retrieval-Augmented Generation）是结合长期记忆和大语言模型的重要技术，使 Agent 能够处理超出大语言模型的训练数据范围的问题，在生成响应前先检索相关的知识，并基于检索到的信息生成更准确和详细的答案，从而提高准确性并减少幻觉。

1.2.4 工具使用（Tools）

工具使用能力是现代 Agentic AI 系统的关键特性，使 Agent 能够超越纯文本交互的限制，与外部系统和资源进行交互，执行相关操作并访问实时信息。这一能力将 Agent 从简单的对话系统转变为能够完成实际任务的智能助手。Agentic AI 系统通过调用外部提供的 API 接口来完成对各种工具的使用。Agent 需要理解 API 的端点（endpoint）、参数、认证要求和响应格式，以及能够从用户输入中提取必要的参数，并确保其有效性。从而正确的构造 API 请求，包括 URL、头部和请求体。在 Agent 系统中调用 API 时，可以直接构造和发送 API 请求以及处理响应，也可以通过中间层（如 MCP：Model Context Protocol）间接与 API 交互，增加安全性和灵活性。

1.3 Agent 的分类

1.3.1 基于交互模式分类

根据交互模式可以把 Agent 分为两大类：单 Agent（Single Agent）和多 Agent（Multi-Agent）系统。这种分类方式主要是基于 Agent 之间的协作方式以及任务分配方式而得出的。

单 Agent 是指以单个 Agent 为核心处理单元的系统架构。在这种架构中，所有的任务处理、决策制定和外部交互均由同一个 Agent 完成。单 Agent 拥有完整的功能集，包括自然语言理解、知识检索、工具调用和响应生成等能力，可以独立完成从用户输入到最终输出的全部流程。单 Agent 架构的优势在于决策路径简短从而提供较快的响应速度，简单的架构带来较低的配置和维护成本，相对固定的行为模式使得测试和优化更加简便，单一模型的调用带来较少的资源消耗，单一决策流程便于追踪和监控。不过单 Agent 架构也存在一定的局限性，比如在处理跨多个专业领域的复杂任务时效率较低，功能的不断增加会导致提示词变得更加复杂，单 Agent 难以同时精通多个专业领域，协调多个工具使用时的能力有限，以及无法像多 Agent 系统那样提供多维度的思考和解决方案。因此单 Agent 架构适用的场景包括：特定领域的任务场景（比如客户服务或者产品推荐等），任务步骤清晰且相对固定的应用场景，需要快速响应且不需要复杂协作的场景等。

多 Agent 系统是指由多个协作的 Agent 共同组成的复杂系统架构。在这种架构中，每个 Agent 负责不同的任务或领域，并针对特定领域优化，通过协作共同完成复杂的问题解决过程。多 Agent 系统通常采用“主管 - 协作者”（Supervisor-Collaborator）模型，不同的协作者并行处理不同的任务。多 Agent 的优势主要在于把复杂问题分解以后交给更专业的、不同的 Agent 执行，提高解决效率。因为多个 Agent 可并行工作，因此系统整体处理效率得到提高。多 Agent 的挑战则在于每个 Agent 交互都计为单独的 API 调用而导致的成本和延迟增加，多 Agent 交互时的跟踪、监控以及问题排查变得更加复杂。多 Agent 系统架构适用的场景包括需要多种专业知识协作的任务（比如综合性咨询服务）、需要多个步骤和决策点的复杂业务流程、需要多角度思考的场景（比如风险评估等）。

1.3.2 基于自主程度的分类

除了基于交互模式的分类外，Agentic AI 系统还可以根据自主程度进行分类，这种分类方式侧重于 Agent 的决策自由度、学习能力和适应性。主要区分为两类：Agentic Workflow（Agent 工作流）和 Autonomous Agent（自主 Agent）。

Agentic Workflow 是一种预定义包含了步骤序列工作流的 Agentic AI 系统，工作流中的每个步骤、决策点和分支逻辑都被事先定义，每个步骤具有特定的责任，Agent 仅在预设的决策框架内做出选择，其决策能力和适应能力有限。在这种模式中，整个工作流程预先设计好，通过编排机制来实现，Agentic AI 系统按照设定的流程执行任务，类似于 " 自动化的流水线 "。其执行结果高度一致，并具有较高的可预测，但是对未预见到的情况的适应能力有限。Agentic Workflow 适用的场景包括具有明确步骤和决策点的流程（比如订单处理和申请审批等）、需要可预测性和可审计性的场景（比如金融服务和医疗保健等）、任务复杂度相对较低的应用、以及需要低延迟和高效率的场景等。

Autonomous Agent（自主 Agent）是一种具有高度自主性的 Agent 系统，能够根据设定的高级目标自行规划、决策并执行复杂任务。与 Agentic Workflow 不同，Autonomous Agent 不依赖预定义的工作流，而是根据当前情境决定行动方案，感知环境变化，并从交互和反馈中学习，不断改进决策过程。Autonomous Agent 通过将大型语言模型作为推理引擎，结合规划能力、记忆机制和工具使用能力，实现更灵活、更适应性强的任务处理方式。Autonomous Agent 适合的应用场景包括需要考虑多种因素和权衡不同选择的场景（比如投资顾问和战略规划）、环境或要求经常变化从而需要灵活适应新情况的场景、需要创造性思维和非常规解决方案的问题、以及需要理解并响应复杂人类意图的交互场景等。

1.4 Agentic AI 的技术栈

Agentic AI 技术栈与标准大语言模型应用的技术栈有所不同，它由多个不同的技术组件组成。

1.4.1 Agent 框架（Agent Framework）

Agent Framework 指的是一个软件框架或工具集（比如 LangChain），用于构建和管理 Agent。这些框架为开发者提供了一套结构化的方法来创建能够自主执行任务、做出决策和与环境交互的 Agentic AI 系统。Agent Framework 的目标是简化 Agentic AI 应用的开发过程，使开发者能够更容易地创建复杂的、自主的 AI 系统。它们通常与大语言模型集成，利用这些模型的强大能力来实现更高级的认知功能。它提供的主要功能包括：

- **Agent 结构：**提供创建和定义 Agent 的基本结构，包括 Agent 的目标、知识库、决策机制等。
- **任务管理：**允许定义和管理 Agent 需要执行的任务，包括任务分解、优先级排序等。
- **环境交互：**提供与外部环境（如 API、数据库、其他系统）交互的接口。
- **记忆和状态管理：**管理 Agent 的短期和长期记忆，以及当前状态。
- **决策引擎：**通常基于大语言模型的输出，实现 Agent 的决策逻辑。
- **工具集成：**允许集成各种工具和 API，扩展 Agent 的能力。
- **多 Agent 协作：**支持多个 Agent 之间的通信和协作。
- **监控和日志：**提供监控 Agent 行为和性能的工具。
- **安全和控制机制：**实现安全检查和控制措施，确保 Agent 的行为在预期范围内。
- **可扩展性：**支持根据需求扩展和定制 Agent 的功能。

1.4.2 Agent 托管（Agent Hosting）

Agent Hosting 是 Agentic AI 技术栈中的关键中间层，承担着将 Agent 部署至本地服务器或云端基础设施的关键任务，实现内外部系统的便捷调用与无缝访问。当前 Agentic AI 生态体系中，大部分 Agent 框架局限于 Python 脚本或 Jupyter Notebook 等开发环境，难以作为独立服务稳定运行。推动 Agentic AI 从开发测试迈向实际生产应用，其核心在于构建可靠且具备高扩展性的服务体系，以满足多元化业务场景的实际需求。在选型 Agent Hosting 解决方案时，需系统性权衡应用场景适配性、业务拓展需求、安全合规要求，以及与现有系统的兼容性等核心要素。从行业发展来看，也逐步出现更多针对不同领域的专业化 Agent 的托管方案，涵盖通用场景优化与垂直行业定制化服务。

1.4.3 模型服务（Model Serving）

在 Agentic AI 技术栈中，Model Serving 是大语言模型从研发成果转化为实际应用价值的核心桥梁。作为 Agentic AI 的智慧中枢，大语言模型需要通过推理引擎进行部署和调用，而这一过程就是 Model Serving。其本质是通过推理引擎将大语言模型封装成可通过 API 访问的服务单元，开发者只需按照接口规范发送输入请求，便能即时获取模型推理结果，整个过程屏蔽了底层复杂的硬件配置、环境搭建及运维管理工作，极大降低了 Agentic AI 应用开发的技术门槛与资源投入。

Model Serving 作为 Agentic AI 技术栈的基础层，通过多样化的服务形式与技术框架，为不同需求的用户提供了丰富的选择。从闭源商业模型的前沿能力，到云上托管模型的灵活多元方案，再到本地推理服务对数据安全与性能优化的保障，每种模式都各有所长，其选择直接决定了 Agentic AI 的能力边界与应用成效。开发者需依据数据敏感性、业务实时性及成本预算等核心需求，来选择合适的服务模式。

1.4.4 记忆（Memory）管理

在 Agentic AI 技术栈中，通过引入记忆管理的技术，Agent 可以回忆用户的名字、偏好、之前的任务目标等，能够对历史对话进行总结，还可以根据历史表现调整行为，甚至保留整个历史记忆提高个性化服务。记忆并非一个孤立的技术组件，而是与 Agent 体系中的规划和工具使用深度交织，形成一个协同系统。规划和工具使用的有效性往往取决于 Agent 记忆的质量和可访问性。Agent 的架构设计必然要求一种模块化方法，其中记忆、规划和工具是既独特又协作的模块。这种模块化对于管理复杂性至关重要，并使得每个组件的专门化开发成为可能。对记忆管理可以选择不同的技术，包括：

传统数据库 （SQL/NoSQL）

传统数据库，包括关系型数据库（SQL）和非关系型数据库（NoSQL），可以作为 Agentic AI 记忆系统的一部分，用于存储结构化和半结构化数据，例如用户画像、交互日志以及适合关系模型或文档模型的事实数据 。其优势在于这些数据库技术成熟，对于事务性数据处理稳健可靠，以及易于理解的查询语言 ，能够作为结构化、实时数据的可靠来源 。挑战则在于如何有效地将传统数据库与更偏向 AI 原生的记忆技术（如向量数据库和知识图谱）相结合，以创建一个统一且易于访问的知识池。

向量数据库与语义搜索

向量数据库已成为现代 AI 记忆，特别是长期记忆的核心组成部分。它们将信息存储为向量嵌入（vector embedding），从而实现基于概念相似性而非仅仅关键词匹配的语义搜索。其优势在于能够从大型数据集中高效检索语义相似的信息，并且能很好地处理非结构化文本。

知识图谱（Knowledge Graphs）

知识图谱将信息存储为实体和关系，为知识表示和推理提供了一个结构化框架 。它们使 Agent 能够理解上下文、进行推断、填补不完整数据中的空白，并对相互关联的数据执行复杂的、基于实体关系的推理 。知识图谱为记忆提供了一种与向量数据库互补的方法，在需要显式结构化知识和多步关系推理（而非仅仅语义相似性）的场景中表现出色。向量数据库擅长查找“相似”事物，而知识图谱擅长理解事物之间“如何”连接。

混合记忆系统（Hybrid Memory Systems）

是指结合不同的记忆技术（例如关系数据库与图数据库），以便利用它们各自的优势，并创建更全面、更多功能的记忆架构 。其基本原理在于，没有任何单一的记忆技术能够完美适用于所有类型的信息或所有 Agent 任务。

Agentic AI 记忆系统的实现和管理，受益于一系列高级框架和专用工具的支撑。这些工具旨在简化开发流程，提供标准化的记忆操作接口，并优化性能。比如类似 LangChain 这类 Agent Framework，为 Agentic AI（包括其记忆组件）的开发提供了抽象层和实用工具集。它们提供用于管理聊天历史、连接各种记忆后端（如向量存储、数据库）以及实现不同记忆策略的模块。而特定的记忆工具则包括各种支持向量嵌入的数据库（比如 PostgreSQL 的 pgvector 插件，Amazon OpenSearch 的 vector search collections 等）。随着 Agentic AI 技术栈的发展，目前也出现了专门的记忆系统和服务，比如 Mem0 等，提供智能的、多层级的记忆（用户、会话、Agent 状态）和自适应个性化来增强 AI 助手的能力。

专用框架（比如 LangChain）和专业记忆服务（比如 Mem0）的出现，标志着 Agentic AI 领域的成熟，正从临时的记忆解决方案转向更标准化和优化的方法。这些工具的存在，表明了对构建 Agent 记忆的可重用组件的普遍需求。

1.4.5 沙箱环境（Sandbox）

Agent 正经历从执行简单自然语言处理 (NLP) 任务到能够规划、推理、使用工具并在动态环境中交互的同时维持记忆的复杂系统的演变 。这种演进的核心之一是 Agent 生成和执行代码，以及与操作系统资源（如命令行、浏览器、文件系统和应用程序编程接口 (API)）进行交互的能力。这种能力的增强与潜在风险的增加成正比。Agent 的自主性以及与敏感环境交互的能力，如果不受控制，可能导致严重的安全漏洞，如恶意软件执行、数据窃取或系统受损 。因此，Agentic AI 能力的提升直接关联到潜在攻击面和风险的扩大。沙箱技术（Sandbox）通过提供隔离环境 ，限制这些自主行为所带来的潜在危害，对 Agent 的执行进行风险管理。

Agent 沙箱是一个隔离的、受控的环境，专为运行 Agent 生成的不可信代码或限制 Agent 与系统资源交互而设计 。它提供了一个具有严格定义边界的安全外围 。沙箱技术不仅仅是一项功能特性，更是安全和负责任地部署 Agent 的基础要求，特别是当这些 Agent 获得更多自主权并能访问关键系统和数据时 。OWASP LLM 十大风险清单强调了诸如“过度代理权”之类的风险，而沙箱技术正是直接针对这些风险的缓解措施 。

沙箱的核心原则是创建一个隔离的环境，将 Agent 的行为限制在其中，以防止对主机系统或其他受保护资源的意外或恶意交互 。隔离可以在不同层面实现，包括操作系统层面（比如利用容器技术）、应用程序层面（比如基于 Jupyter 内核的沙箱）、甚至硬件级虚拟化层面（比如亚马逊开源的 Firecracker 技术），每种层面在安全强度、性能和复杂性方面都有不同的权衡 。硬件级虚拟化的 Firecracker 技术，为每个虚拟机运行完整的操作系统内核（针对启动和运行速度进行了优化），被认为是比容器更强的隔离性和安全边界 ，因为每个 microVM 都有自己的内核，适用于运行高度敏感或不受信任的代码。比如目前广泛使用的 E2B Sandbox 的沙箱环境调度框架就使用了 Firecracker 作为 Agent 自动生成的不可信代码的执行环境。

1.5 Agent 技术延展

随着人工智能技术的迅猛发展，Agentic AI 技术也诞生了多种前沿的领域，包括 MCP（Model Context Protocol）、Computer Use 技术、Browser Use 技术以及 Agent2Agent（A2A）技术，它们正在重塑 AI 与数字世界的交互方式。

1.5.1 MCP（Model Context Protocol）

Model Context Protocol（MCP）是由 Anthropic 公司于 2024 年 11 月开源的一种新标准，旨在建立大模型与数据存储系统之间的统一连接协议，包括内容存储库、业务工具和开发环境。MCP 的核心目标是帮助前沿 AI 模型产生更相关、更有价值的响应。MCP 的基本理念虽然技术性很强，但核心思想非常简单：为 Agentic AI 提供一种与工具、服务和数据连接的一致方式，无论它们存在于何处或如何构建。

MCP 解决了 Agentic AI 应用开发中的关键痛点：

- **统一接口：**MCP 正式化了 Agentic AI 与环境的通信方式。开发者不再需要硬编码 API 调用或将所有内容塞入冗长的提示中，而是可以使用清晰的界面将模型逻辑与外部系统分离。
- **简化集成：**传统上，每个数据源都需要维护单独的连接器。而使用 MCP，开发者现在可以针对一个标准协议进行构建。
- **上下文保持：**随着生态系统的成熟，AI 系统将在不同工具和数据集之间移动时保持上下文，用更可持续的架构替代当前分散的集成方式。

MCP 技术已经在各个领域展现出强大的应用潜力，并且其应用范围正在迅速扩大，从基本的文件系统交互扩展到复杂的专业工具集成。这种标准化的协议使 Agent 能够更加无缝地连接到各种服务和工具，为用户提供更加智能和个性化的体验。

1.5.2 Computer Use 技术

Computer Use 技术代表了 Agentic AI 技术的突破，使 AI 能够像人类一样操作计算机，通过视觉理解屏幕内容，并通过鼠标和键盘交互执行复杂任务。Computer Use Agent（CUA）是一种能够像人类一样使用计算机的 AI 模型，可以搜索网络信息、填写表格和点击按钮。这些 Agent 结合了先进的视觉模型和推理能力，能够通过查看用户屏幕的截图，理解界面内容和布局。进而将光标移动到特定位置，进行具体的操作。

Computer Use 技术为各类应用场景开辟了新的可能性，比如：

- **流程自动化：**Computer Use 技术通过使自动化更加智能和直观，大幅提高 RPA（机器人流程自动化）的灵活性。
- **日常任务辅助：**可以进行开放式的信息检索，研究，甚至可以帮助用户寻找产品的最佳价格或制定旅行规划。
- **企业效率：**可以自动化许多日常工作，例如填写表单、进行数据录入或进行软件测试。这种能力使得 AI 能够处理多步骤的复杂流程，从而提高工作效率。

1.5.3 Browser Use 技术

Browser Use 技术是 Agentic AI 技术的另一个研究领域，专注于使 Agentic AI 能够有效地导航和与网络进行交互。这一技术使 Agentic AI 能够像人类一样浏览网页、填写表格、提取信息，甚至完成复杂的网络任务，比如能够从网站中提取和处理结构化数据，为数据收集、内容分析和研究任务提供强大支持。在 Browser Use 技术中，开源的自动化测试工具 Playwright 提供了重要的能力。具体来说，Playwright 在 Browser Use 中的作用包括：

- **基础设施：**Browser Use 本质上是将 Playwright 与 AI 能力结合的一个库，其核心技术是 " 在 Playwright 基础上包装了 AI 智能，并赋予大语言模型控制浏览器的能力。
- **跨浏览器支持：**Playwright 为 Browser Use 提供了一个统一的 API，可以自动化 Chromium（Chrome、Edge）、Firefox 和 WebKit（Safari）等主流的浏览器，使 AI Agent 能够在不同浏览器环境中工作。
- **DOM 交互能力：**通过 Playwright，Browser Use 能够扫描网页并提取所有交互元素（按钮、输入字段、链接、表单等），然后为 AI Agent 提供这些元素的结构化表示，使 AI 能够理解并与之交互。

1.5.4 Agent2Agent（A2A）协议

Agent2Agent（A2A）协议是由谷歌在 2025 年 4 月发起的开放标准，旨在实现不同 Agent 系统之间的通信和互操作性。核心目标是允许基于不同框架或由不同供应商构建的 Agent 发现彼此的能力，协商交互模式，并互相协作完成任务。其主要特点包括：

- **Agent 发现：**Agent 可以通过 Agent Cards（JSON 格式）发现彼此的能力，从而使得 Agent 可以定位到适合执行某个特定任务的其他 Agent。
- **任务管理：**Agents 之间的通信主要是为了完成终端用户的请求。对于长时间运行的任务，Agent 会对状态进行流式更新和实时协作。
- **协作：**Agents 可以相互发送消息来传递上下文、回复或用户指令。
- **用户体验协商：**通过“部件（parts）”（这是一个完整形成的内容片段，如生成的文本或图像），客户端和远程 Agent 可以协商正确的格式和用户界面功能，并明确包含对用户 UI 功能的协商，如 iframe、视频、Web 表单等。

A2A 协议目前尚处于早期发展阶段，在标准化程度、功能实现、生态系统等方面还有待进一步完善，需要业界持续的努力和探索，以实现更成熟和完善的 Agent 间通信机制。

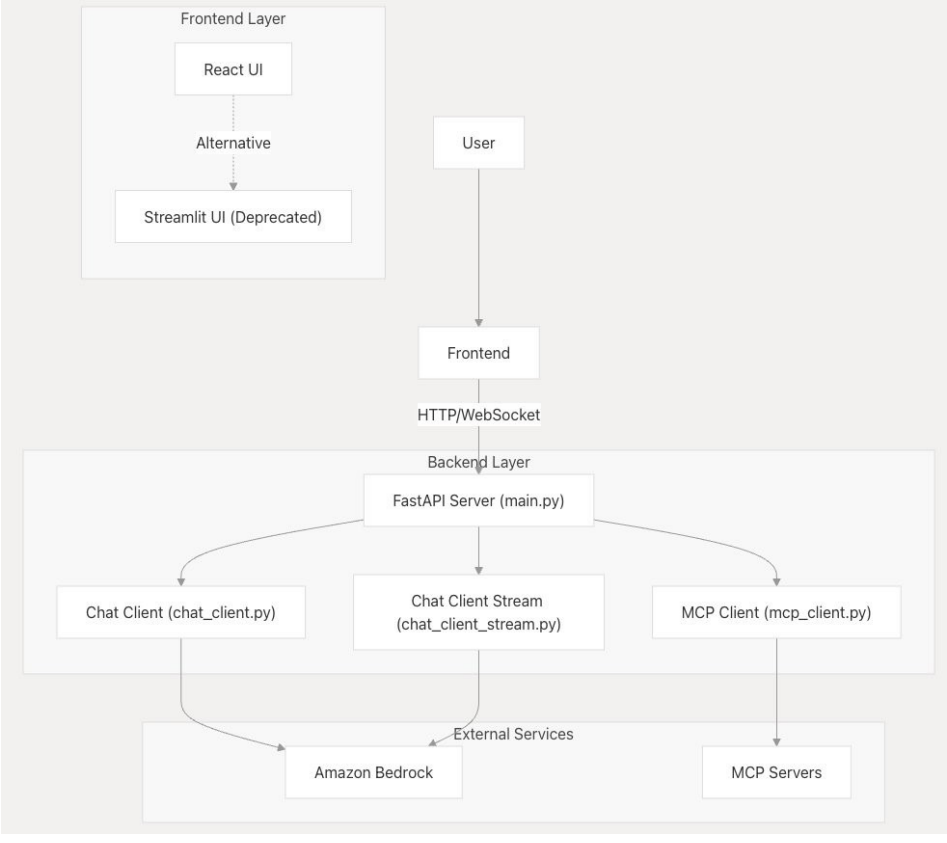
1.6 Agentic AI 的应用形态

Agentic AI 主要有两种应用形态：通用 Agent 和垂类 Agent。

1.6.1 通用 Agent

通用 Agent 指的是能够执行各种通用任务的 Agent 系统，它不局限于特定领域，而是能够处理多种类型的请求和任务。通用 Agent 系统通常都会包含 Agent 框架（Agent Framework）、Agent 托管环境（Agent Hosting）、沙箱环境（Sandbox）、记忆（Memory）管理以及 MCP 的集成。

通用型个人助手是一种典型的通用 Agent，也成为 Agent 技术最普及的应用形式之一。这类系统能够理解自然语言指令，并通过与各种外部工具和服务的集成，帮助用户完成从简单到复杂的多种任务。以亚马逊云科技提供的、基于 Amazon Bedrock（这是使用基础模型构建和扩展生成式人工智能应用程序的全托管服务）和 MCP 的通用 Agent 样例（https://github.com/aws-samples/demo_mcp_on_amazon_bedrock/tree/main）为例，我们可以深入了解其技术架构和工作原理，从而了解通用 Agent 的技术场景。该样例实现了一个功能强大的通用型个人助手 Agent 系统。其总体架构如下图。

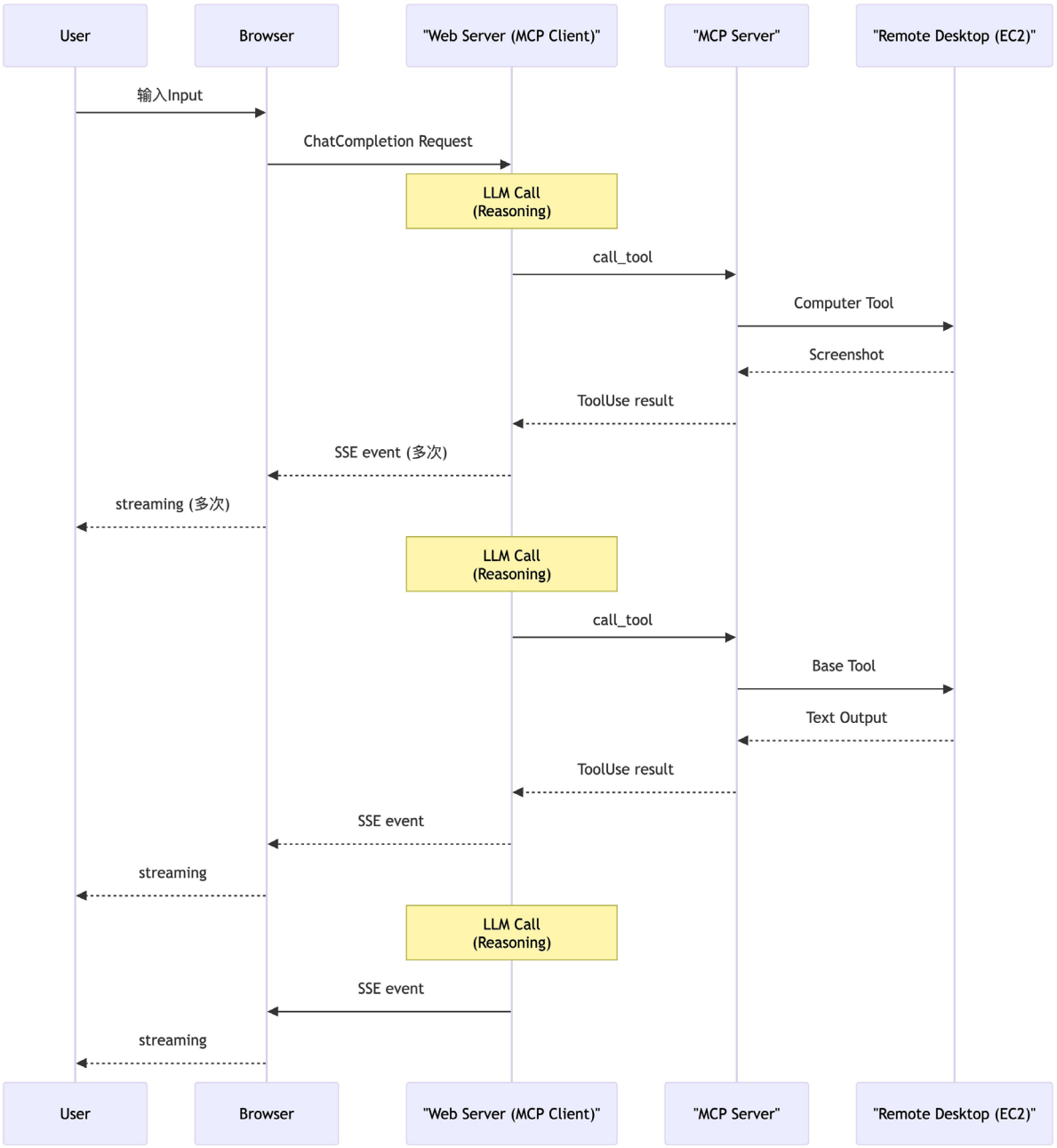


该样例实现的工作流程如下：

- 用户输入处理：通用 Agent 系统接收并解析用户的自然语言指令。
- 任务规划：通过 Amazon Bedrock 调用基础模型（如 Anthropic Claude 或 Amazon Nova），对用户意图进行理解，从而制定执行计划。
- 工具选择与调用：通过 MCP 协议，系统动态选择并调用适当的工具来完成特定子任务。
- 信息整合：将从各个工具获取的信息整合，形成连贯的解决方案。
- 响应生成：生成自然语言响应并呈现给用户。
- 上下文维护：保存会话状态和上下文信息，以支持持续交互。

我们可以通过该通用 Agent 处理复杂任务，比如当用户请求“帮我生成一份 2024 年中国企业出海行业市场分析报告，需要制作成精美的 HTML, 你可以使用工具获取尽可能丰富和准确的数据和信息”时，该 Agent 系统的处理流程为：

1. 理解任务需求，确定需要收集信息，制定执行计划。
2. 通过 MCP 调用搜索工具获取相关信息。
3. 利用 Computer Use 工具访问特定网页获取详细资料和图像。
4. 使用系统工具保存图片和生成 HTML 文件。
5. 整合信息生成结构化报告，并以 HTML 格式保存到用户指定位置。



通这一过程展示了通用 Agent 如何通过工具组合解决复杂任务，而无需为每个特定任务开发专门的解决方案。

1.6.2 垂类 Agent

与通用 Agent 不同，垂类 Agent（Vertical Agent）专注于特定领域的任务和服务，为特定行业或功能量身定制的智能系统，基于特定领域的数据、工作流程和标准构建，能够精确地执行复杂的现实世界任务。其主要特点包括：

- 领域专长：有些垂类 Agent 会利用特定领域的大语言模型，能够有效解决复杂的特定领域挑战。
- 精准性：与通用 Agent 相比，垂类 Agent 在特定任务上表现更加精准。
- 效率：针对特定领域优化，使处理速度更快，资源利用更高效。
- 深度集成：与行业特定工具和系统深度集成，提供无缝工作流程。

我们借助 Cline（https://github.com/cline/cline）来了解垂类 Agent 的技术架构。Cline 是一款面向 VS Code 的编程扩展，属于编程领域的垂类 Agent（Coding AI Agent），它能够通过文件操作、终端命令执行、浏览器交互等多种工具辅助软件开发，同时具备严格的安全控制机制以确保用户环境安全。其架构设计采用消息传递机制实现扩展主程序与 Webview 的高效通讯，并支持通过统一的 API 使用多种大语言模型提供服务。Cline 支持通过 MCP 拓展自定义工具功能，增强灵活性和扩展能力。Cline 采用了前置上下文加载的策略，通过“计划模式（Plan）”和“执行模式（Act）”的分离，实现了更高效的代码生成。在计划模式下，Cline 与用户交互，收集上下文信息，比如使用 read_file 或 search_files 来获取更多关于任务的背景信息。一旦获得了足够多的关于用户请求的背景信息，就可以制定一个详细的计划来完成任务，包括架构设计和执行计划。在计划模式下，无法更改代码库，而是让用户聚焦于理解需求并制定实施计划。在执行模式下，则基于制定的计划，可以使用 Cline 的所有构建功能，对代码进行编写。Cline 使用了一个名为 Memory Bank 的结构化文档系统，并结合上下文管理，来实现 Agent 系统中的记忆功能。该文档系统使得 Cline 可以在不同会话之间维护上下文，从而将 Cline 从无状态的助手转变为一个持久的开发伙伴，能够随着时间的推移有效地“记住”您的项目细节。

构建 Agentic AI 应用

2

通常来说，我们在构建 Agentic AI 应用时，可以先从简单的提示指令开始尝试，避免过度设计和不必要的复杂功能。只有当简单方法不奏效时，再考虑更复杂的多步骤方案。在构建 Agentic AI 系统时，应专注在三个核心部分：Agent 所处的环境、Agent 能使用的工具和给 Agent 的基本指令。可以先实现一个最低功能的版本，然后根据实际效果和反馈逐步改进完善。

我们应该让每个 Agent 都需要清楚自己“能做什么”和“不能做什么”，明确为 Agentic AI 指定它的任务是什么，并给它详细的说明和指导。同时为 Agent 说明目标是什么，以及清晰的成功标准，从而让 Agent 知道在什么情况下被认为是完成了任务。

构建时，从“我要怎么写代码解决这个问题”转变为“我要怎么指导 Agent 来解决这个问题”，这就像从亲自做到指导别人做的思维转变。使用自然、简洁的语言与它交流，避免过于技术化的表达。避免给 Agent 处理过度复杂的格式或极长的代码计算。设计 Agent 应用时，应当考虑透明度，让终端用户能够看到 Agent 的思考和决策过程，从而打造一个人与 Agent 之间顺畅的交流界面。

在选择 Agentic Workflow（Agentic 工作流）或者 Autonomous Agent（自主 Agent）构建时，需要综合考虑任务的确定性，对错误的容忍度，灵活性的需求，复杂性以及用户交互的多样性等因素。在选择时，可以参考下表所示的建议。

考虑因素	倾向于 Agentic 工作流	倾向于自主 Agent
任务确定性	高度确定	开放性、探索性
错误容忍度	低（任务关键型）	相对较高
灵活性需求	低（遵循固定路径）	高（需要自适应）
复杂性	可分解为明确步骤	需要推理和规划
用户交互	最小化或结构化	会话式、适应性强

在亚马逊云科技，无论是通过定制开发、直接可用的解决方案、还是两者的组合，都可以满足各种构建 AI Agent 应用的需求。为了有效的构建 Agentic AI 应用，在亚马逊云科技上提供了三种主要的方式：专用 Agent、全托管的 Agent 以及完全自己管理的模式。

2.1 专用 Agent：Amazon Q

专用 Agent 是可直接部署并使用的 Agent，同时也可以进一步定制以满足特定业务和场景的需求。通过结合客户自己的数据和应用系统，这些直接可用的 Agent 能够自动执行重复性任务，比如代码创建或者内容生成、快速回答问题、以及采取相关的行动从而解决问题。对于企业客户（包括企业专业人士和企业开发人员）来说，如果希望以最小的技术投入立即部署 Agentic AI 应用，则可以借助亚马逊云科技提供的 Amazon Q Business 和 Amazon Q Developer，快速测试和部署 Agentic AI 应用，或者进一步定制以满足客户业务的具体需求。

2.1.1 Amazon Q Developer

Amazon Q Developer 是功能强大的 AI 助手，用于构建、操作和转换软件，具有管理数据和 AI/ML 的高级功能。Amazon Q Developer 不仅能辅助编程，还能帮助开发人员和 IT 专业人员完成所有任务，从编程、测试和部署，到故障排除、执行安全扫描和修复、现代化应用程序、优化亚马逊云资源以及创建数据工程管道。Amazon Q Developer 常见的使用场景包括代码生成与优化，代码安全扫描，代码审核，生成测试脚本，运维助手，遗留系统现代化等。

2.1.1.1 代码生成与优化：加速开发流程

在现代软件开发中，开发者经常需要编写大量重复性的样板代码，这不仅消耗了宝贵的开发时间，更阻碍了团队将精力投入到真正的业务创新上。Amazon Q Developer 能够深度理解开发者的意图并自动生成高质量的代码实现。例如，当开发者需要在 Amazon Lambda 上构建一个新的 REST API 端点来处理用户认证请求时，只需在集成开发环境（IDE）中输入简单的自然语言指令，如 " 创建一个接受 POST 请求的 /api/authenticate 端点，返回 JWT 令牌 "。系统会智能分析项目上下文，自动生成包含完整 API 实现、相关配置文件和单元测试的代码。同时，开发者可以调用 Amazon Q Developer Agentic coding 的能力，自动执行单元测试，如果无法通过，则可以让 Amazon Q Developer 尝试自己修复。

常见 Agent 编程场景有：

- **设计稿生成 UI 交互界面：**通过 MCP 实现 Amazon Q Developer 与 Figma 设计工具的对接，自动获取设计稿中的视觉元素、布局信息和交互规范，使用 Agent 自动生成对应的前端代码，包括组件结构、样式定义和交互逻辑。
- **单元测试代码生成：**Agent 能够感知当前代码环境和项目结构，自动生成覆盖核心功能的单元测试用例，并自动执行测试验证代码质量。当测试失败时，系统会智能分析失败原因并自动修复相关问题，确保代码的健壮性。
- **Bug 修复：**基于错误日志、堆栈跟踪和代码上下文，Agent 能够自动定位问题根源，生成修复方案并实施代码修改，大幅减少开发者在调试和问题排查上的时间投入。

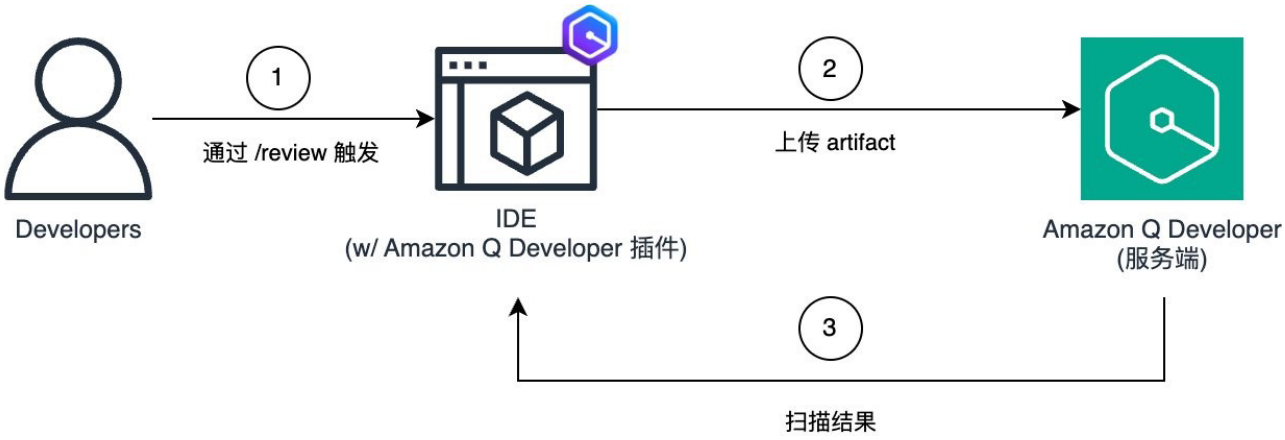
2.1.1.2 代码安全扫描：构建安全防护屏障

代码安全漏洞是威胁应用程序稳定性和用户数据安全的重要风险因素。一个看似微小的安全缺陷可能导致数据泄露、系统入侵或业务中断等严重后果。因此，在开发阶段及早识别和修复安全漏洞至关重要，这不仅能够显著降低后期修复成本，更能从根本上提升应用程序的安全防护水平。

Amazon Q Developer 正是基于这一需求，内置了功能强大的 Amazon Q Detector Library，一个汇集了丰富安全检测规则和代码质量标准的专业知识库。该工具专门为帮助开发者在亚马逊云科技上构建安全、高效的应用程序而设计，能够自动识别潜在的安全风险点。

开发者可以通过简单的操作启动代码安全扫描：在 Amazon Q Developer 的 Visual Studio Code 或 JetBrains IDE 插件中，只需在对话输入框中输入 `/review` 指令，即可以针对当前打开的单个代码文件进行精准扫描，也可以对整个代码仓库进行全面检查。

代码扫描的执行流程如下图：



1 **扫描触发**

开发者通过 `/review` 指令启动扫描，根据需要选择扫描当前激活文件或整个代码仓库。

2 **云端分析**

Amazon Q Developer IDE 插件将待扫描文件安全传输至服务端，利用内置的 Amazon Q Detector Library 进行深度安全分析。

3 **结果展示**

系统返回详细的扫描报告，智能地将发现的问题按照 Critical（严重）、High（高危）、Medium（中危）、Low（低危）、Info（信息）五个风险等级进行分类排列。

扫描完成后，开发者可以在 IDE 中直观查看识别出的安全漏洞详情，Amazon Q Developer 还会提供针对性的修复建议，帮助开发者快速解决问题。值得一提的是，该代码扫描功能不仅限于本地开发环境，还完美集成到 GitLab 和 GitHub 等主流代码托管平台中，能够在 Pull Request（代码合并请求）阶段自动触发安全检查，确保每一次代码变更都经过严格的安全审核，真正实现安全开发的全流程覆盖。

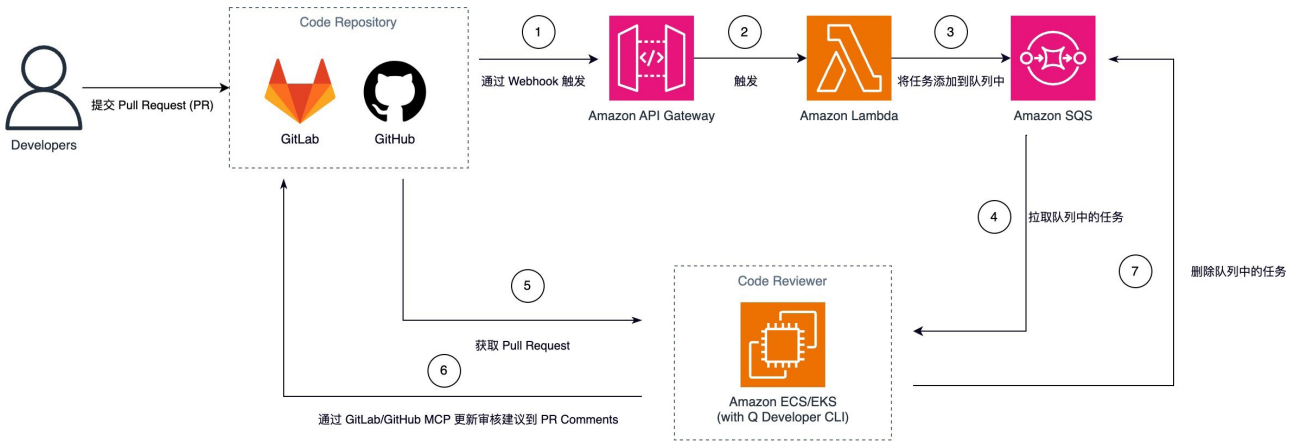
2.1.1.3 代码审核：提升代码可维护性

在协作开发环境中，代码质量的一致性和可维护性直接影响着项目的长期成功。代码审核作为软件开发流程中的关键环节，是指开发者完成功能开发并向 Git 仓库提交代码合并请求后，由其他团队成员在代码正式合并前进行全面检查和评估的质量保障过程。这一实践不仅是技术层面的把关，更是团队协作和知识传承的重要桥梁。

代码审核的核心价值体现在以下几个关键维度：

- **统一编码风格，提升代码可读性：**审核过程能够有效发现并纠正代码风格不一致的问题，例如 Python 函数命名不符合 PEP8 规范、缺少异常处理机制等，确保代码风格在整个项目中保持统一标准。
- **识别性能优化机会：**经验丰富的审核者往往能够发现潜在的实现优化点，比如在 Python 开发中建议使用列表推导式替代传统的 for 循环，从而显著提升代码执行效率。
- **减少技术债务积累：**通过持续的代码审核实践，团队能够及时发现和解决设计缺陷，避免技术债务像财务债务一样产生“利息”，防止后期维护和重构成本的增长。

开发者可以通过 Amazon Q Developer CLI 与 GitLab/GitHub 等代码仓库联动，实现代码审核自动化，以下是开发者构建自动化代码审核的参考架构：



1

PR 提交触发

开发者向 GitLab 或 GitHub 代码仓库提交 Pull Request（代码合并请求），通过 Webhook 机制自动触发后续的审核流程。

2

API 网关接收

Amazon API Gateway 接收来自代码仓库的 Webhook 请求，作为整个审核系统的统一入口点。

3

代码审核任务入列

Amazon Lambda 函数被触发执行，将代码审核任务添加到 Amazon SQS 队列中。

4

获取审核队列中的任务

Amazon ECS/EKS 中的运行程序获取 Amazon SQS 中的任务信息，这些任务信息中包含代码仓库的地址，PR 信息，代码审核标准等等。

5

获取代码内容

系统从 GitLab 或 GitHub 仓库中获取具体的 Pull Request 代码变更内容，代码审核程序触发 Amazon Q Developer CLI 对代码进行深度分析。

6

结果反馈

审核完成后，系统通过 GitLab 或 GitHub 的 MCP（Model Context Protocol）将详细的审核结果以 PR Comments 的形式直接反馈给开发者，包括发现的问题、修复建议和代码质量评分。

7

删除队列信息

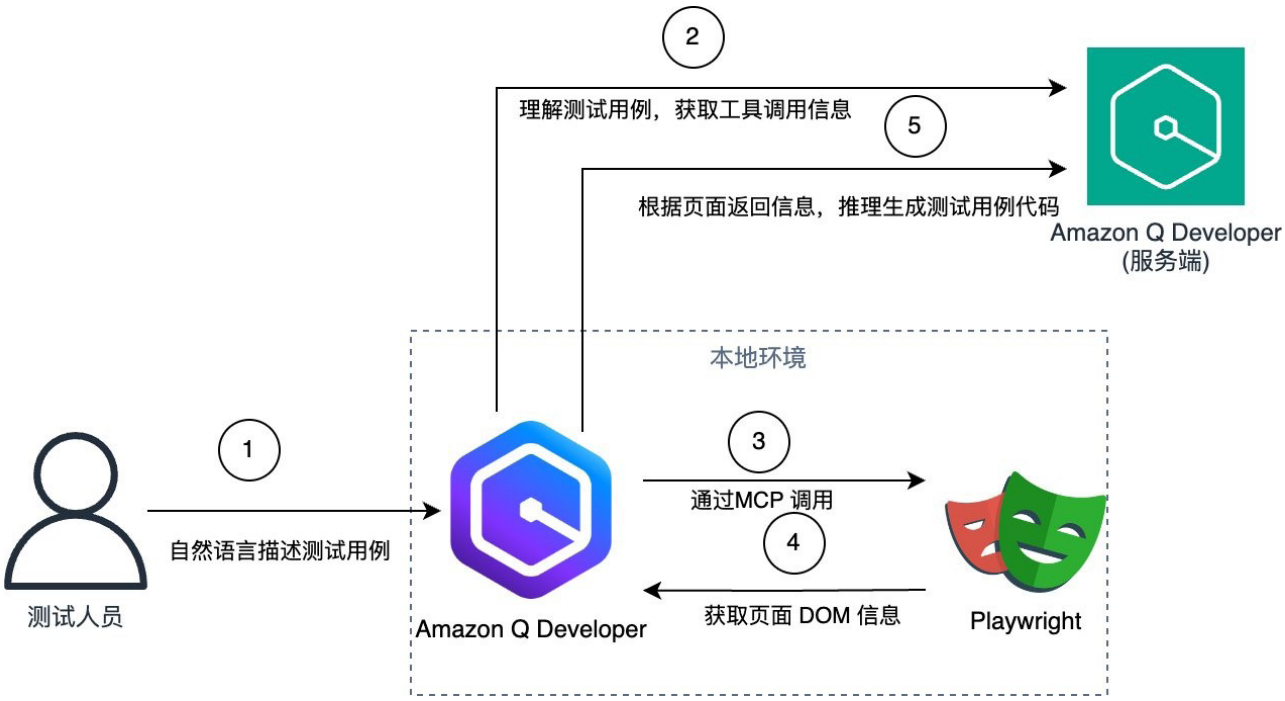
代码审核程序删除 Amazon SQS 队列中的信息，确保每个审核任务不被重复触发。

2.1.1.4 端到端测试脚本自动生成：提升软件质量

在敏捷开发和持续迭代的背景下，端到端测试是防范代码变更导致现有功能回退的重要防线。新功能的快速上线往往伴随着意想不到的副作用——看似无关的代码修改可能会破坏已有的核心业务流程。然而，传统的端到端测试用例编写需要开发者掌握复杂的测试框架语法，维护工作繁重。面对快速迭代的开发节奏，如何高效地生成 端到端测试用例成为开发团队面临的重要挑战。

Amazon Q Developer CLI 支持 MCP，通过与 Playwright 的集成，为开发者提供了通过智能 Agent 快速生成端到端测试用例的强大能力，将测试脚本的生成过程从手工编码转变为自然语言驱动的自动化流程。

以下是 Amazon Q Developer 智能生成端到端测试用例的详细流程：



1

自然语言测试需求描述

测试人员使用自然语言向 Amazon Q Developer CLI 描述需要生成的测试用例需求，无需掌握复杂的 Playwright 测试框架语法。

2

需求理解与工具调用

Amazon Q Developer 服务端理解测试用例描述，获取相关的工具调用信息。

3

MCP 协议集成调用

Amazon Q Developer 通过 MCP 与 Playwright 集成，自动打开浏览器，按照自然语言描述实现操作自动化。

4

页面 DOM 信息获取

Playwright 自动打开目标网页并获取页面的 DOM 结构信息，并返回给 Amazon Q Developer。

5

智能测试代码生成

Amazon Q Developer 服务端基于页面返回的 DOM 信息和用户的测试需求，通过智能推理生成完整的端到端测试用例代码，包括用户操作模拟、异步处理逻辑、错误捕获机制和断言验证等。

2.1.1.5 亚马逊云科技运维助手：降低运维难度提升运维效率

传统的运维工作往往需要运维人员具备深厚的技术功底和丰富的实战经验，在面对紧急故障时容易因为操作复杂而延误最佳处理时机。Amazon Q Developer CLI 作为智能化的运维助手，彻底改变了这一现状，通过自然语言交互的方式大幅降低了亚马逊云科技上的运维技术门槛。

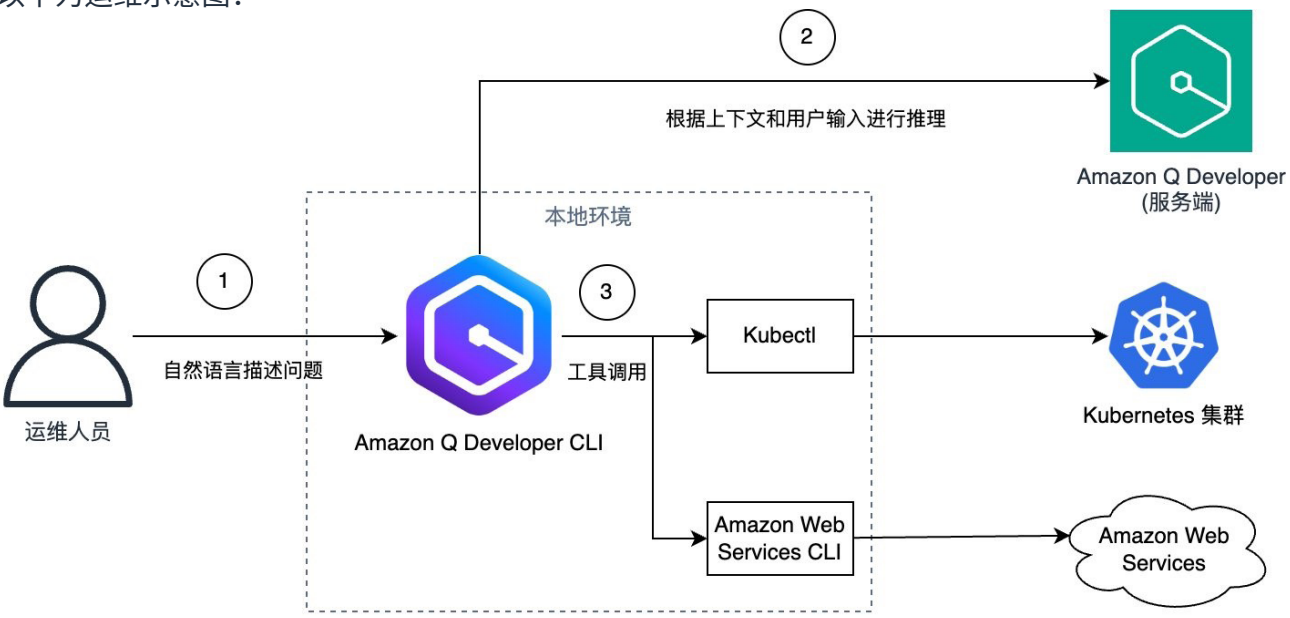
Amazon Q Developer CLI 内置了强大的工具调用能力，对亚马逊云科技生态系统拥有深度的理解和全面的知识覆盖。运维人员无需记忆复杂的命令行语法或 API 参数，只需用自然语言描述需求，Q Developer CLI 就会自动调用亚马逊云服务的操作指令并执行相关的命令。

典型的运维场景包括：

- **智能故障诊断：**当系统出现异常时，运维人员可以直接描述问题现象，Amazon Q Developer CLI 会自动分析相关的亚马逊云服务状态、日志信息和监控指标，快速定位问题根源并提供解决方案。
- **自动化基础设施部署：**通过自然语言描述基础设施需求，系统能够自动生成相应的 Infrastructure as Code（IaC）代码，如 CloudFormation 模板或 Terraform 配置文件。
- **成本优化建议：**基于当前的资源使用情况和历史数据，智能分析并提供个性化的成本优化建议，包括资源右调、预留实例购买建议、存储类型优化等，帮助企业有效控制云成本。

除了深度集成亚马逊云服务外，Amazon Q Developer CLI 还具备对超过 250 种常见运维工具的熟练掌握，涵盖网络诊断工具（如 curl、netstat）、版本控制系统（如 git）、容器编排工具（如 kubectl）等运维工作中的核心工具。

以下为运维示意图：



Amazon Q Developer CLI 运维操作的详细流程：

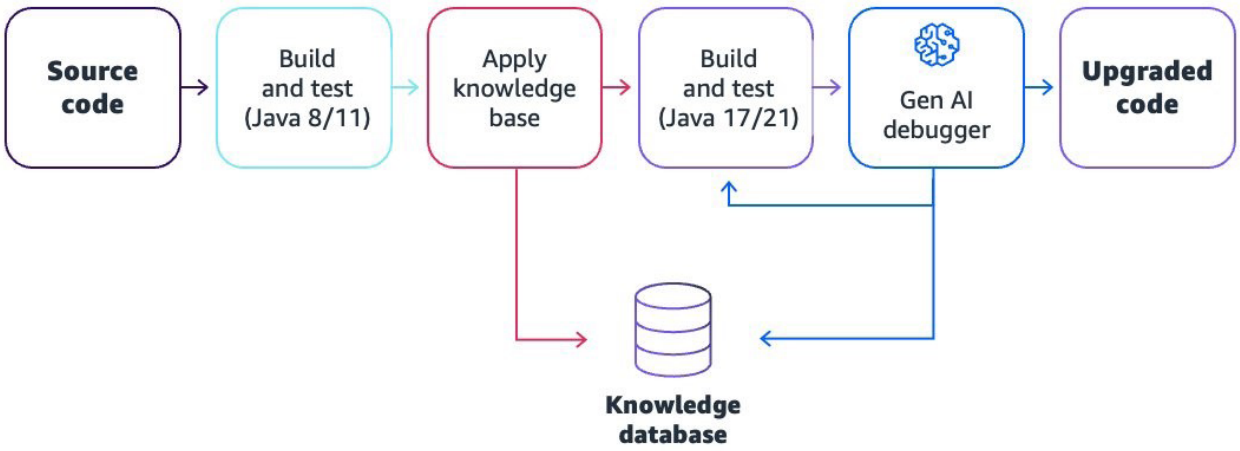
1. **自然语言问题描述：**运维人员使用自然语言向 Amazon Q Developer CLI 描述遇到的运维问题或需要执行的操作任务。
2. **智能推理分析：**Amazon Q Developer 服务端基于用户的自然语言输入和当前的上下文环境信息进行智能推理，理解用户的真实意图并制定解决方案和工具调用。
3. **工具调用执行：**Amazon Q Developer CLI 在本地环境中根据推理结果智能选择并调用合适的工具。

Amazon Q Developer CLI 会持续监控和分析每次工具调用的执行结果。基于返回结果进行再次推理：如果问题已成功解决，则自动生成操作过程总结，帮助运维人员了解完整的处理流程；如果仍需进一步处理，Q Developer CLI 会基于当前状态制定策略，执行后续的工具调用操作。

2.1.1.6 遗留系统现代化：简化迁移和升级

软件维护阶段通常涉及系统升级、重构与安全加固等任务，传统方式往往耗时耗力。Amazon Q Developer 通过自动化能力，显著提升了维护效率，降低了技术债务。

Amazon Q Developer 的 Java Transformation 功能内置 Java 升级的知识库，可根据用户需求自动完成升级语言版本、更新依赖项和配置、修改不兼容语法等任务。借助 Amazon Q Developer，企业能够加速系统现代化进程，显著降低维护成本与风险，有效控制技术债务，提升可维护性。



例如，一家电视操作系统厂商借助 Amazon Q Developer，将两个合计超过 13 万行的 Java 8 系统升级至 Java 17，仅耗时 2 天，而传统方式预计需四周。升级过程中，系统不仅完成了版本迁移，还显著减少了安全漏洞和技术债务。

2.1.2 Amazon Q Business

Amazon Q Business 是强大的生成式 AI 助手，可用于查找信息、获取业务洞察和采取行动。它能够整合企业组织内各种数据源的数据，获得业务洞察并回答相关的问题。它可以回答问题、生成内容、解决问题，并在各种应用程序中执行操作。Amazon Q Business 在设计之初就考虑到了安全性和隐私性，使企业组织能够安全地使用生成式 AI。它保留了所有数据的用户级别的访问控制，同时 Amazon Q Business 绝不会使用客户数据来改进底层模型。

我们知道生成式 AI 有能力提升生产力。但在这些生成式 AI 应用和 Agent 真正在工作中发挥作用之前，它们必须了解企业组织的数据、客户、运营和业务。而且它们必须是安全的。然而，当今许多 Agent 既不能轻易进行定制化，也无法满足企业所需的数据隐私和安全要求。这正是亚马逊云科技推出 Amazon Q Business 的原因，它是强大的生成式 AI 助手，能帮助员工比以往更有效率、更具创造力、更依据数据做决策。

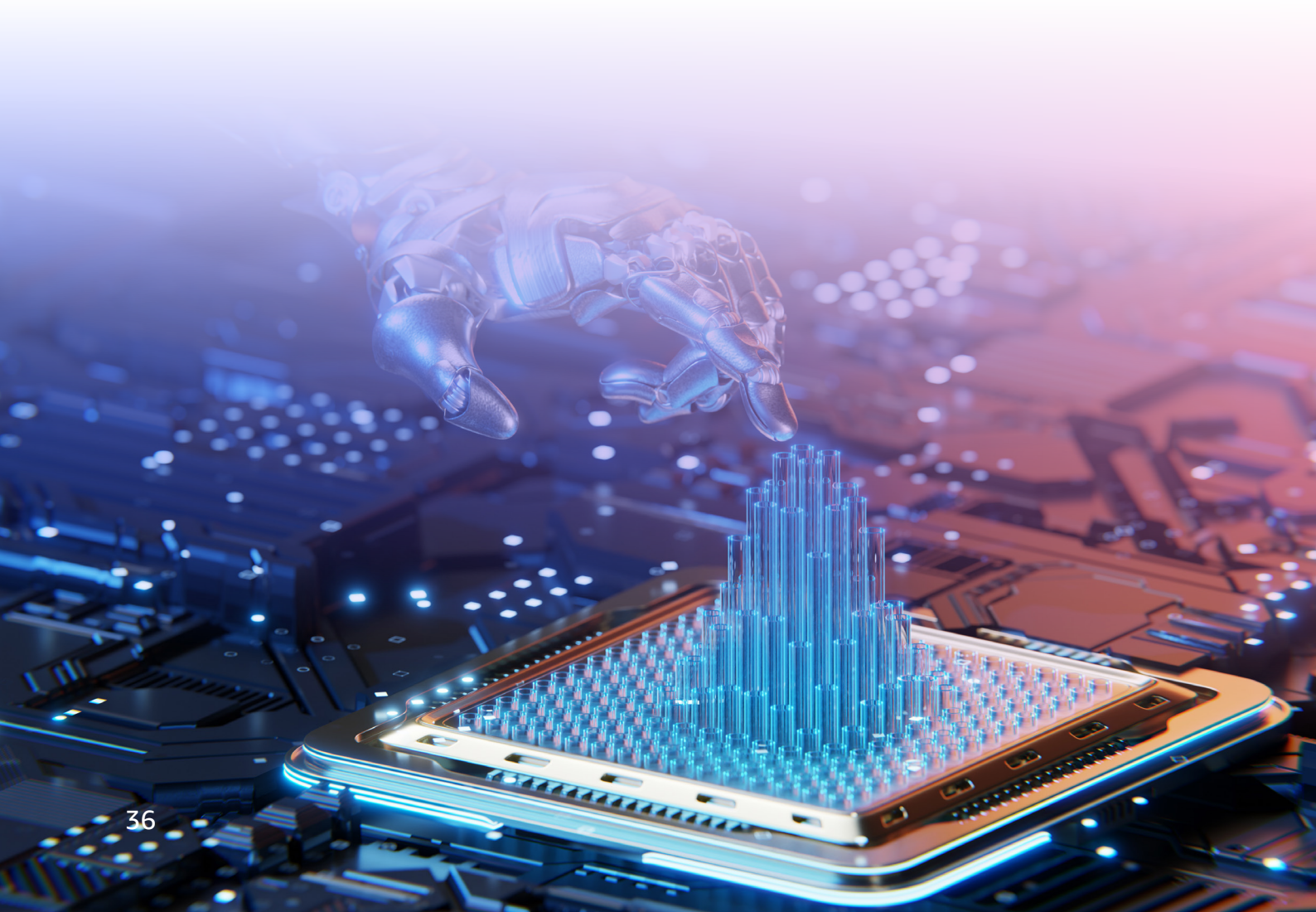
通过 Amazon Q Business，企业可以轻松地将企业数据统一起来，帮助员工发现信息、分析数据、创建内容并更快地采取行动。我们还让这些步骤的自动化变得更加简单。

Amazon Q Business 包含的特性有：

- 提供 40 多个完全托管的连接器（connector），只需几次点击，即可整合企业所有资源内容，从企业现有的应用程序和文档库中导入数据，包括 Salesforce、Slack、SharePoint、Asana、Zendesk、Amazon S3、Dropbox、Jira、Confluence、Github、ServiceNow、Microsoft Exchange、Google Workspace 等应用，让您能够以统一集中的方式访问所有数据。
- 保持已有的访问控制：Amazon Q Business 在设计时就考虑到安全性和隐私性，它能理解并遵循您现有的身份、角色和权限。如果用户在不使用 Amazon Q Business 的情况下无权访问某些数据，使用 Amazon Q Business 时也同样无法访问。
- Amazon Q Business 不会使用企业的数据来为他人改进底层模型。
- 管理员能够轻松应用护栏来自定义和控制响应，确保用户针对他们提出的问题而获得相关且适当的回答。

有很多场景可以使用 Amazon Q Business，比如：

- 跨应用程序执行操作：通过 Amazon Q Business 的操作功能，可以与包括 Jira、ServiceNow、Salesforce 和 PagerDuty 在内的主流第三方应用程序无缝交互，提升工作效率。用户可以直接在 Amazon Q Business 中执行多种操作，无需在不同系统间切换，从而节省宝贵时间。
- 提取关键洞察：Amazon Q Business 统一了对结构化数据（数据库、数据仓库）和非结构化数据（文档、维基、邮件）的访问，使用户能够在单一应用程序中获得全面的见解，更快地做出明智决策。
- 加速内容创作：从市场营销到销售和工程团队的所有人都可以使用 Amazon Q Business 的生成式 AI 聊天界面来提高生产力和创造力。它可用于创建电子邮件文案、生成博客草稿和摘要、编写销售话术等。帮助各部门员工更快地创建内容。
- 将 Amazon Q Business 的数据集成到企业现有的应用程序中：软件供应商可以通过简单的 API 操作即可把他们的软件集成到 Amazon Q Business 创建的索引中，以增强其自身生成式 AI 助手的结果。最终企业可以控制哪些软件供应商的应用程序可以访问其数据，且索引保持细粒度的权限控制。



2.2 全托管的 Agent 服务：Amazon Bedrock 和 Amazon Bedrock Agents

Amazon Bedrock 是一项完全托管的服务，通过单个 API ，我们可以调用来自 Anthropic、DeepSeek、Meta、Mistral AI、Stability AI、Amazon 等领先人工智能公司的高性能基础模型（FM），并提供通过安全性、隐私性和负责任的人工智能构建生成式人工智能应用程序所需的一系列广泛功能。使用 Amazon Bedrock，我们可以轻松试验和评估适合当前场景的基础模型，通过微调和检索增强生成（RAG）等技术利用企业自己的数据对其进行定制化，构建执行任务的 Agent，从预订旅行和处理保险索赔，到创建广告活动和管理库存，所有这些都可以快速构建和上线。由于 Amazon Bedrock 是无服务器的，因此我们也不用管理任何基础设施，并且可以使用已经熟悉的亚马逊云服务将生成式人工智能功能安全地集成和部署到应用程序中。在 Agentic AI 的技术栈中，Amazon Bedrock 提供了模型服务（Model Serving）的能力。

Amazon Bedrock Agents 是亚马逊云科技提供的全托管服务，其目的是帮助企业快速构建、部署和管理能够完成复杂任务的 Agentic AI 应用。在 Agentic AI 技术栈中，它同时提供了 Agent 托管（Agent Hosting）和 Agent 框架（Agent Framework）的能力。作为 Amazon Bedrock 服务家族的重要组成部分，Amazon Bedrock Agents 专注于简化 Agentic AI 应用的开发流程，帮助开发人员能够创建能与企业系统、API 和数据源交互的 Agent 。

Amazon Bedrock Agents 使用大语言模型的推理能力，将用户请求分解为多个步骤，协调和执行复杂任务。它能够处理从 API 调用、知识库查询到代码解释等多种操作，提供端到端的 Agent 应用开发体验。它提供了视觉化界面和简化的开发流程，减少代码编写需求，从而帮助降低开发门槛。通过预先构建的组件和模板加速开发过程，大幅减少开发周期。通过它内置的安全控制和合规机制，确保 Agent 应用符合企业的安全标准。Amazon Bedrock Agents 建立在 Amazon Bedrock 的基础之上，继承了 Amazon Bedrock 访问多种基础模型的能力，同时专注于提供 Agentic AI 特有的功能。它扩展了基础模型的能力边界，使模型能够与外部系统交互并执行实际操作，而不仅限于文本生成。

2.2.1 Amazon Bedrock Agents 的核心组件

2.2.1.1 基础模型选择与集成

Amazon Bedrock 支持多种领先的基础模型，包括 Anthropic 的 Claude 系列、Mistral AI 的模型、DeepSeek R1 、Meta 的 Llama 3 以及 Amazon 的 Nova 系列模型等。这些模型在能力、规格和专长上各有差异，适合不同的使用场景和预算要求。

在选择具体的模型时需要考虑：

- 上下文窗口大小：从较小的 2K tokens 到较大的 200K tokens 不等，影响 Agent 处理信息的能力
- 响应速度：轻量级模型（如 Amazon Bedrock Claude 3.7 Haiku）适合需要快速响应的场景，重量级模型（Amazon Bedrock Claude 4 Opus）适合复杂任务。如果要平衡响应速度和推理效果，则可以考虑中等规模的模型，比如 Amazon Bedrock Claude 3.7 Sonnet。
- 成本效益：模型复杂度与调用成本成正比，需根据任务价值选择适当模型
- 特定能力：不同模型在编码、推理、创意生成等方面表现各异
- 多语言支持：中文等非英语语言的支持程度的差异

在模型的选择上，我们建议的最佳实践包括：

- 根据任务复杂性分级使用模型，从而兼顾响应时间、推理效果以及成本。比如使用轻量级模型处理简单查询，使用重量级模型处理复杂任务规划。或者借助 Amazon Bedrock 智能提示路由（Amazon Bedrock Intelligent Prompt Routing）将提示自动路由到模型系列中的不同基础模型，从而帮助您优化响应质量和成本。智能提示路由功能可以在不影响准确性的情况下将成本降低多达 30%。
- 构建模型评估框架，基于业务 KPI 而非纯技术指标选择模型。
- 在模型表现和成本之间寻找平衡，关注业务影响而非模型参数大小。

2.2.1.2 动作组（Action Groups）

动作组是 Amazon Bedrock Agents 的核心功能之一，它允许 Agent 调用企业 API 和服务来执行具体操作。通过动作组，Agent 可以查询数据库、更新记录、调用微服务、触发工作流程等，从而实现与企业系统的无缝集成。在构建和配置动作组的过程中，我们建议的最佳实践包括：

- 提供清晰、详细的自然语言描述，帮助模型理解何时和如何调用函数
- 设计适当粒度的函数，避免过于宽泛或过于细化
- 实现严格的参数验证和错误处理，增强调用可靠性
- 为关键操作添加用户确认流程，提高安全性

2.2.1.3 知识库（Knowledge Bases）

Amazon Bedrock Agents 可以与 Amazon Bedrock Knowledge Bases 无缝集成，实现检索增强生成（RAG）功能。这使 Agent 能够访问企业专有信息，基于真实、最新的数据提供准确回答，有效减少幻觉问题。Amazon Bedrock 知识库的核心功能包括文档索引，向量存储，基于元数据和权限控制信息访问以及提供信息来源和证据链等。在对知识库进行优化时，我们建议的策略包括：

- 文档分割与处理：采用适当粒度和重叠策略分割文档
- 检索参数配置：调整相似度阈值、结果数量等参数
- 查询优化：优化检索查询构造，提高相关性
- 元数据增强：添加有意义的元数据，提升过滤精度
- 对知识库的内容进行同步与更新，以确保信息时效性的时候，我们可以利用 Amazon EventBridge 触发自动同步流程。同时建议实现增量更新策略，提高更新效率。

2.2.1.4 流程编排（Orchestration）

Amazon Bedrock Agents 提供强大的流程编排能力，使 Agent 能够规划和执行复杂的多步骤任务。它采用 ReAct（Reasoning and Acting）方法作为默认编排策略，结合推理（思考过程）和行动（工具使用），通过推理得出要执行的动作，并对执行动作后的结果进行观察，并根据观察调整后续规划。也提供了推理过程的可见性和可解释性。

除了缺省的编排策略，Amazon Bedrock Agents 也支持自定义编排策略，包括修改默认 ReAct 行为，添加中间验证步骤、实现特定领域的推理逻辑和决策规则以及优化性能关键场景和减少中间交互。在对 Agent 的流程编排中，我们可以应用的优化技巧包括：

- 提供清晰的指令和示例，引导 Agent 规划能力
- 分解复杂任务为可管理的子任务
- 设计适当的回退和错误恢复机制
- 优化提示工程，提高编排效率

2.2.1.5 代码解释器（Code Interpreter）

Amazon Bedrock Agents 的代码解释器功能使 Agent 能够生成和执行代码，分析复杂问题，处理数据转换，创建可视化效果等。代码在安全隔离的环境中运行，提供强大的计算能力同时确保安全性。其核心能力包括动态生成 Python 代码解决用户问题、执行数据分析、创建图表和数据可视化，并支持多种文件格式（CSV、JSON、Excel 等）。通过这些核心能力，我们可以把代码解释器应用在不同的场景下，比如复杂数据分析与探索、报告生成与数据可视化、数值计算与模拟、文件格式转换与处理等。在执行代码的过程中，需要考虑安全因素，Amazon Bedrock Agents 提供了隔离环境（Agentic AI 技术栈中的沙箱环境），并在该隔离环境中执行代码以防止恶意操作，以及限制资源使用，避免计算资源滥用。

2.2.1.6 记忆管理（Memory Management）

Amazon Bedrock Agents 提供记忆管理功能，使 Agent 能够保留和利用之前交互的上下文，提供更连贯、个性化的用户体验。对于短期记忆来说，Amazon Bedrock Agents 能够单次会话中的交互历史、理解上下文引用和代词指代、跟踪会话状态和进度以及在复杂多轮对话中保持一致性。对于长期记忆来说，则可以把用户会话等信息保存在 Amazon Bedrock 知识库中。在具体的记忆实现策略中，可以利用唯一记忆 ID 区分不同用户，配置记忆保留期限（最长 30 天），使用摘要技术压缩记忆从而避免上下文窗口限制，以及实现选择性记忆从而只保留重要信息。

2.2.1.7 会话处理（Session Handling）

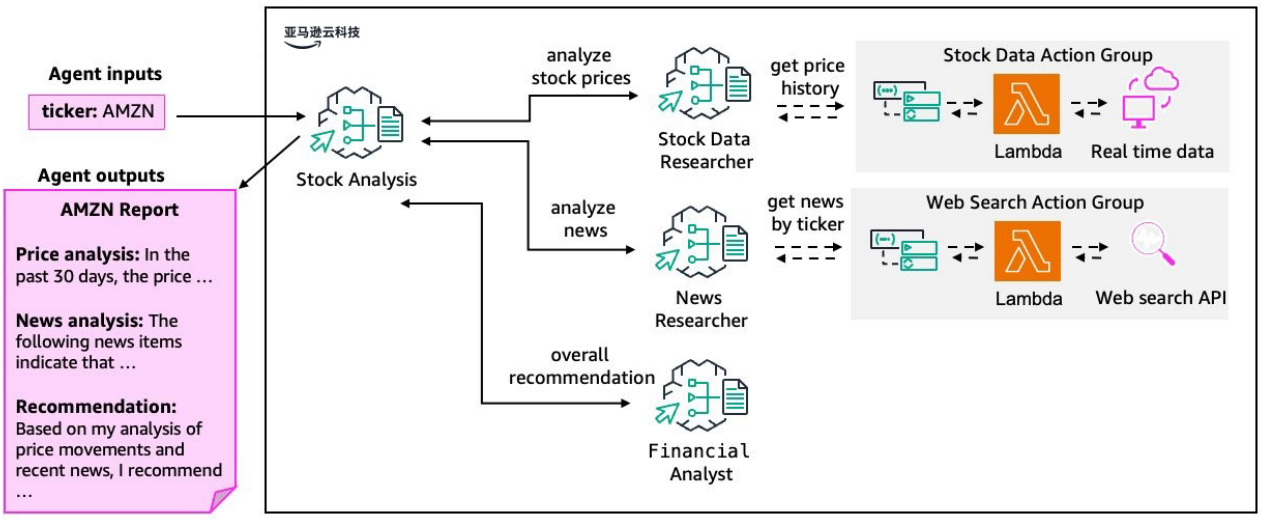
Amazon Bedrock Agents 的会话处理功能提供了灵活的状态管理和会话控制机制，支持复杂交互流程和上下文保留。对会话状态进行管理，比如管理会话参数和配置、传递仅对 Action Groups 可见的信息、提供对模型可见的上下文信息等。在会话管理中，则会使用唯一会话标识符跟踪对话、配置会话超时和终止策略、实现状态转换和会话分支处理、支持上下文窗口管理和压缩等。在会话处理中，我们一般的最佳实践则包括设计简洁明了的会话流程、提供明确的会话开始和结束提示以及优化上下文传递从而避免不必要的信息冗余。

2.2.1.8 安全与监控

Amazon Bedrock Agents 提供全面的安全控制和监控功能，确保 Agent 应用符合企业安全标准和合规要求。这些安全控制机制包括利用 Amazon Bedrock Guardrails 来过滤有害内容并控制主题边界、通过 IAM 权限管理实现精细化访问控制和身份验证、对数据传输和存储进行加密保护、Amazon Bedrock Agents 为代码解释器的执行而提供安全隔离的执行环境。在 Agent 执行过程中的监控层面来说，可以利用 Amazon Bedrock Agents 提供的 Trace 功能来追踪 Agent 的决策和执行流程，以及使用 Amazon CloudWatch 监控性能指标和异常事件。Amazon Bedrock Agents 在日志里详细记录了 Agent 的操作和交互历史，从而供用户进行分析。

2.2.2 多 Agent 协作案例分析

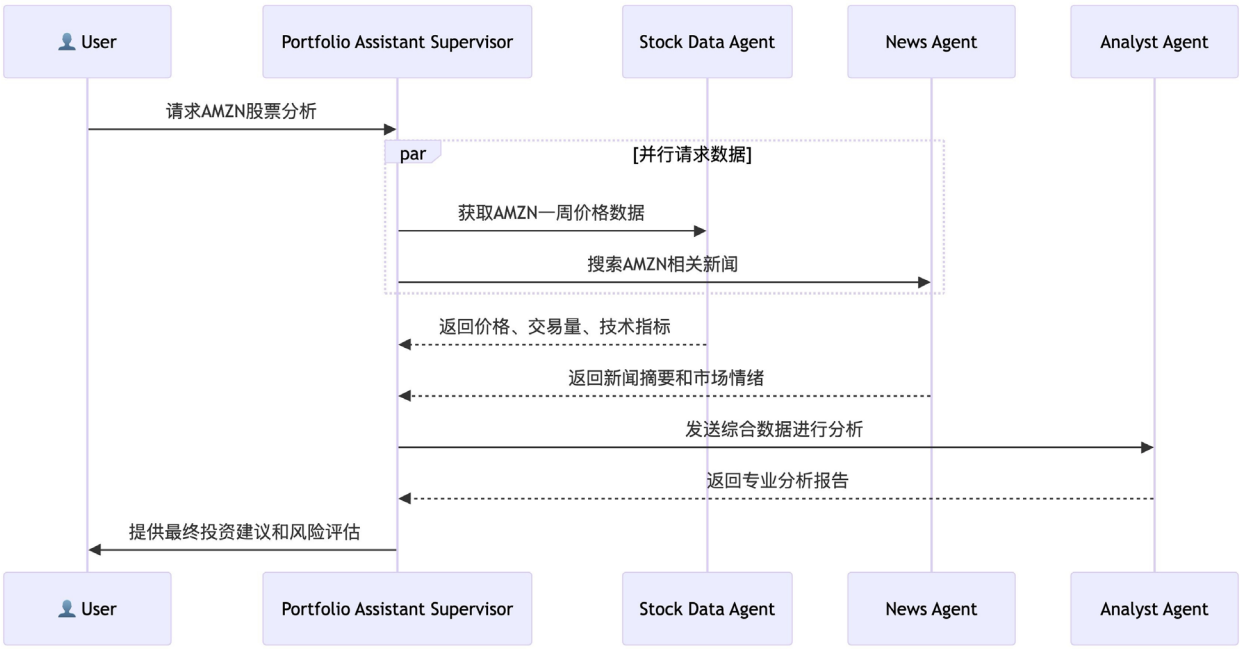
我们以投资组合助手来说明如何使用 Amazon Bedrock Agent 开发多 Agents 的 Agentic AI 系统。在这个场景中，投资组合助手会对给定的潜在股票投资进行分析，并提供包含投资考虑因素的报告。该系统通过多个子 Agent 协作完成股票分析、新闻研究和投资建议生成。 该应用的架构图如下所示。



其中包含四个 Agent：

- 新闻代理（News Agent）：搜集和分析相关新闻，提供市场情绪和趋势洞察
- 股票数据代理（Stock Data Agent）：获取和分析股票历史数据、价格趋势和技术指标
- 分析师代理（Analyst Agent）：综合新闻和股票数据，生成专业分析报告
- 投资组合助手监督代理（Portfolio Assistant Supervisor）：协调各子代理，生成最终投资建议

这四个 Agent 的协作流程示意图如下所示：



在设计该多 Agent 的时候，我们会遵循以下原则，以构建更强大、更灵活的 Agentic AI 解决方案。

1. 单一职责原则：每个 Agent 应专注于特定领域或功能，比如新闻、股票数据等不同领域使用不同的 Agent，避免能力过于宽泛导致的效率低下。
2. 松耦合设计：这些 Agent 之间通过标准化接口通信，减少直接依赖，提高系统灵活性。
3. 可扩展性考虑：设计时考虑未来 Agent 的添加和功能扩展，避免架构僵化。
4. 故障隔离：单个 Agent 的故障不应影响整个系统运行，实现优雅降级。

针对该多 Agent 的代码实现中的关键步骤包括：

步骤 1：导入必要模块和初始化

```
# 基于 portfolio_assistant_agent/main.py 的官方实现
import sys
from pathlib import Path
import argparse
import boto3
sys.path.append(str(Path(__file__).parent.parent.parent.parent))

from src.utils.bedrock_agent import Agent, SupervisorAgent, Task, Guardrail, region, account_id

bedrock_client = boto3.client("bedrock")
```

步骤 2：创建子代理

```
# 创建新闻代理
    news_agent = Agent.create(
      name="news_agent",
      role="Market News Researcher",
      goal="Fetch latest relevant news for a given stock based on a ticker.",
      instructions="Top researcher in financial markets and company announcements.",
      tool_code=f"arn:aws:lambda:{region}:{account_id}:function:web_search",
      tool_defs=[
        {
          "name": "web_search",
          "description": "Searches the web for information",
          "parameters": {
            "search_query": {
                "description": "The query to search the web with",
                "type": "string",
                "required": True,
              },
              "target_website": {

  "description": "The specific website to search including its domain name. If not provided, the most relevant website will be used",
                "type": "string",
                "required": False,
              },
```

```
                    "topic": {
                        "description": "The topic being searched. 'news' or 'general'. Helps narrow the search when news is the focus.",
                        "type": "string",
                        "required": False,
                    },
                    "days": {
                        "description": "The number of days of history to search. Helps when looking for recent events or news.",
                        "type": "string",
                        "required": False,
                    },
                },
            }
        ],
    )

    # 创建股票数据代理
    stock_data_agent = Agent.create(
        name="stock_data_agent",
        role="Financial Data Collector",
        goal="Retrieve accurate stock trends for a given ticker.",
        instructions="Specialist in real-time financial data extraction.",
        tool_code=f"arn:aws:lambda:{region}:{account_id}:function:stock_data_lookup",
        tool_defs=[


  {  # lambda_layers: yfinance_layer.zip, numpy_layer.zip
            "name": "stock_data_lookup",
            "description": "Gets the 1 month stock price history for a given stock ticker, formatted as JSON",
            "parameters": {
                "ticker": {

 "description": "The ticker to retrieve price history for",
                    "type": "string",
                    "required": True,
                }
            },
        }
    ],
)

    # 创建分析师代理
```

```
        analyst_agent = Agent.create(
            name="analyst_agent",
            role="Financial Analyst",
            goal="Analyze stock trends and market news to generate insights.",
            instructions="Experienced analyst providing strategic recommendations. You take as input the news
summary and stock price summary.",
        )
```

步骤 3：创建投资组合助手监督代理

```
    # 创建监督代理
        portfolio_assistant = SupervisorAgent.create(
            "portfolio_assistant",
            role="Portfolio Assistant",
            goal="Analyze a given potential stock investment and provide a report with a set of investment
considerations",
            collaboration_type="SUPERVISOR",
            instructions="""
                Act as a seasoned expert at analyzing a potential stock investment for a given
                stock ticker. Do your research to understand how the stock price has been moving
                lately, as well as recent news on the stock. Give back a well written and
                carefully considered report with considerations for a potential investor.
                You use your analyst collaborator to perform the final analysis, and you give
                the news and stock data to the analyst as input. Use your collaborators in sequence, not in parallel.""",
            collaborator_agents=[
                {

    "agent": "news_agent",
                    "instructions": """
                                    Use this collaborator for finding news about specific stocks.""",
                },
                {
                    "agent": "stock_data_agent",
                    "instructions": """
                                    Use this collaborator for finding price history for specific stocks.""",
                },
                {
                    "agent": "analyst_agent",
                    "instructions": """
                                    Use this collaborator for taking the raw research and writing a detailed report and
investment considerations.""",
```

```
        },
    ],
    collaborator_objects=[news_agent, stock_data_agent, analyst_agent],
    guardrail=no_bitcoin_guardrail,
    llm="us.anthropic.claude-3-5-sonnet-20241022-v2:0",
```

步骤 4：执行多代理协作分析

```
# 调用投资组合助手进行多代理协作任务
request = "what's AMZN stock price doing over the last week and relate that to recent news"

result = portfolio_assistant.invoke(
    request, enable_trace=True, trace_level=args.trace_level
)
print(f"Final answer:\n{result}")
```

在这投资组合助手系统中，Amazon Bedrock Agents 会使用 Supervisor Agent 协调多个专业 Agent，基于任务驱动多个 Agent 进行执行和协作。该投资组合助手支持动态股票代码输入和分析，并提供完整的追踪和日志功能。有关该投资组合助手更详细的代码，可以查看：https://github.com/awslabs/amazon-bedrock-agent-samples/tree/main/examples/multi_agent_collaboration/portfolio_assistant_agent

2.2.3 Amazon Bedrock Inline Agent

Amazon Bedrock Inline Agent 是一种动态 AI 助手解决方案，使得开发人员可以在运行时动态配置和调整 AI 助手的行为。与传统的静态 Agent 相比，Inline Agent 提供了更大的灵活性和适应性，使其能够根据实际需求实时调整 Agent 指令和参数，也支持代码解释器、知识库和自定义操作组，从而实现无需重新部署即可测试新配置。我们以一个实例来说明 Amazon Bedrock Inline Agent 的使用方式。

```
# 1. 基础设置
import boto3

bedrock_rt_client = boto3.client("bedrock-agent-runtime")

# 2. 配置参数
model_id = "anthropic.claude-3-sonnet-20240229-v1:0"
sessionId = "custom-session-id-123"
request_params = {
   "instruction": """You are a helpful AI assistant helping employees
                  with their questions.""",
   "foundationModel": model_id,
   "sessionId": sessionId,
   "endSession": False,
   "enableTrace": True
}

# 3. 添加工具（代码解释器）
code_interpreter_tool = {
   "actionGroupName": "UserInputAction",
   "parentActionGroupSignature": "AMAZON.CodeInterpreter"
}
request_params["actionGroups"] = [code_interpreter_tool]

# 4. 添加知识库（可选）
kb_config = {
   "knowledgeBaseId": "your-kb-id",
   "description": "Knowledge base description",
   "retrievalConfiguration": {
      "vectorSearchConfiguration": {
         "numberOfResults": 3,
```

```
            "overrideSearchType": "HYBRID"
        }
    }
}
request_params["knowledgeBases"] = [kb_config]

# 5. 调用 Inline Agent
response = bedrock_rt_client.invoke_inline_agent(**request_params)
```

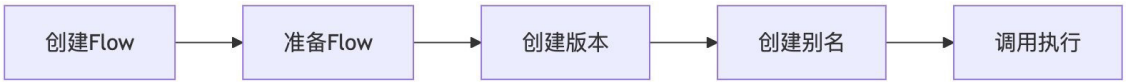
更细节的代码实现请参考：https://aws-samples.github.io/amazon-bedrock-samples/agents-and-function-calling/bedrock-agents/features-examples/15-invoke-inline-agents/inline-agent-api-usage/

下表显示了与传统预配置的 Bedrock Agent 的区别。

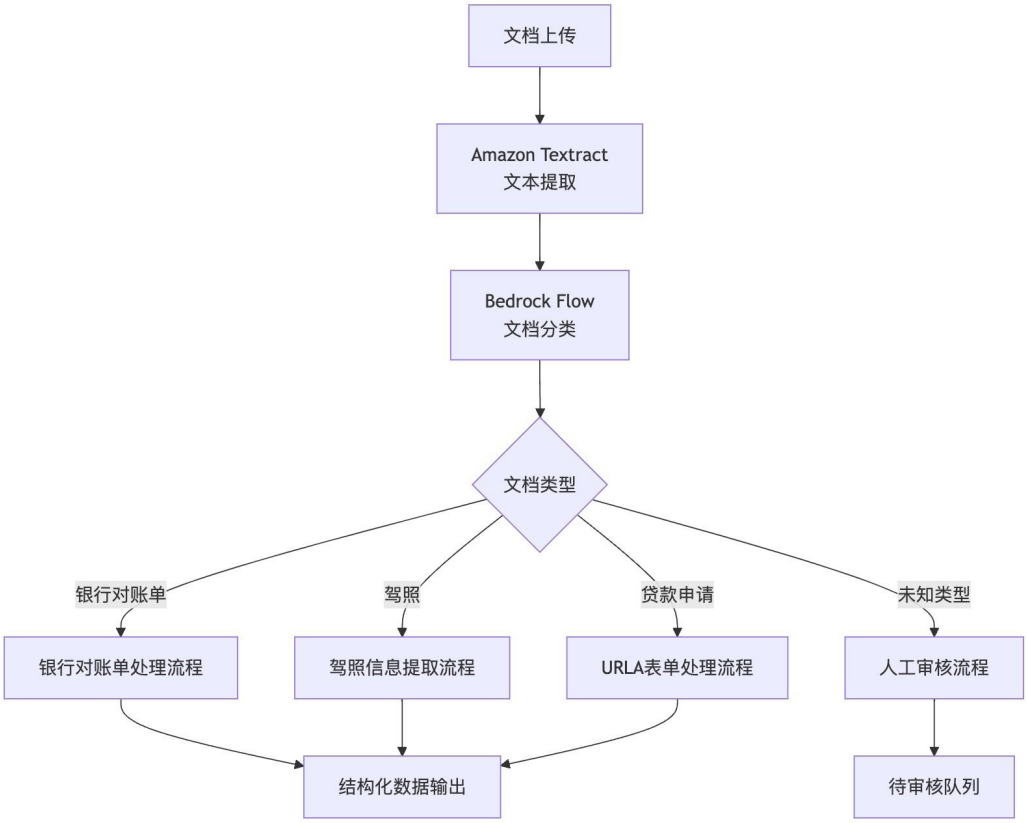
维度	传统预配置 Agent	Inline Agents
部署方式	预先配置，需要部署	运行时动态配置，无需部署
功能扩展	固定模块，修改需重新部署	动态添加 / 移除，即时生效
使用场景	生产环境，固定业务流程	原型开发，动态业务场景
维护成本	高（配置复杂，流程长）	低（配置简单，快速迭代）
适用场景	企业级应用，稳定服务	开发测试，定制化需求

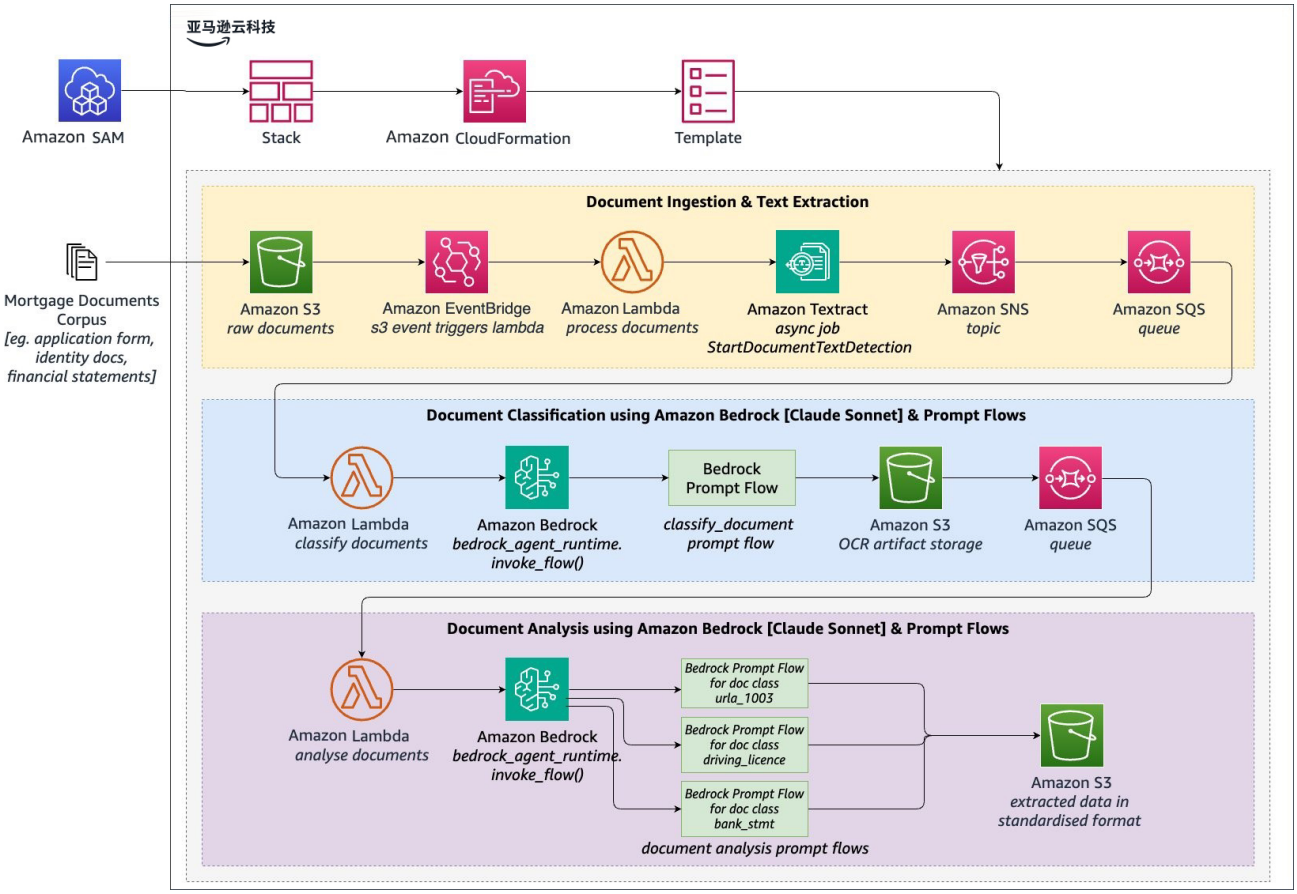
2.2.4 Amazon Bedrock Flow

Amazon Bedrock Flows（原 Amazon Bedrock Prompt Flows）是亚马逊云科技提供的完全托管服务，使企业能够通过直观的可视化构建器而快速创建、测试和部署生成式 AI 工作流。该服务将基础模型、提示词、亚马逊云科技的服务和自定义逻辑无缝连接，构建端到端的 AI 解决方案。Amazon Bedrock Flows 的主要组件包括节点（Nodes）比如输入节点、提示词节点和输出节点，以及用于定义节点之间的数据流连接（Connections）。使用 Amazon Bedrock Flows 的流程如下图所示。



这些功能共同构成了一个完整的工作流管理系统，帮助开发者减少开发时间和工作量。我们以一个智能文档处理的例子来说明如何使用 Amazon Bedrock Prompt Flows 构建 Agentic AI 应用。该智能文档处理的应用主要针对企业在制造业、金融和医疗保健等行业中面临的大量文档处理的需求，如财务报告、合同、病历等文档的自动化处理。具体工作流程如下：





实现该场景的解决方案架构图如下所示：

该解决方案具有以下特点：

- 完全基于 serverless 架构，具有高度可扩展性
- 支持多种文档类型，如贷款申请、银行对账单、驾驶证等
- 易于扩展，可以通过添加新的 Prompt Flow 支持新的文档类型
- 成本效益高，处理 1000 页文档的成本低于 25 美元
- 实现了文档处理的自动化，显著减少了人工处理时间和错误

这个解决方案特别适合需要处理大量文档的企业，可以帮助他们提高效率、降低成本、减少错误，并能随业务需求的变化快速调整文档处理流程。详细方案以及部署代码，请参考博客：https://aws.amazon.com/blogs/machine-learning/automate-document-processing-with-amazon-bedrock-prompt-flows-preview/

2.3 完全自己构建的 Agent

如果您打算不借助任何的 Agent 托管服务，而是完全自己构建 Agent 应用，则可以借助亚马逊云科技提供的各种不同的工具来实现 Agentic AI 的技术栈。

2.3.1 AI Agent 开源框架：Strands Agents

Strands Agents 是由亚马逊云科技开源的一个 AI Agent 开发框架，它采用模型驱动的方法，使得开发者能够通过几行代码就快速构建和运行 AI Agent。与其他需要开发者定义复杂工作流的框架不同，Strands Agents 简化了 Agent 的开发过程，充分利用了现代大语言模型的推理、规划、调用工具和自我反思的能力。该框架可以从简单的会话助手场景扩展到复杂的自主工作流场景，既适用于本地开发环境，也适合生产环境的部署。Strands Agents 与亚马逊云科技的生态深度集成，在亚马逊云科技上部署优势明显。并且通过 Strands Agents 开发的 Agentic AI 应用程序可以在亚马逊云科技全球区域运行，也可以在亚马逊云科技的中国区运行。目前，亚马逊云科技内部的多个团队已经将 Strands Agents 用于生产环境，包括 Amazon Q Developer、Amazon Glue 和 VPC Reachability Analyzer 等。

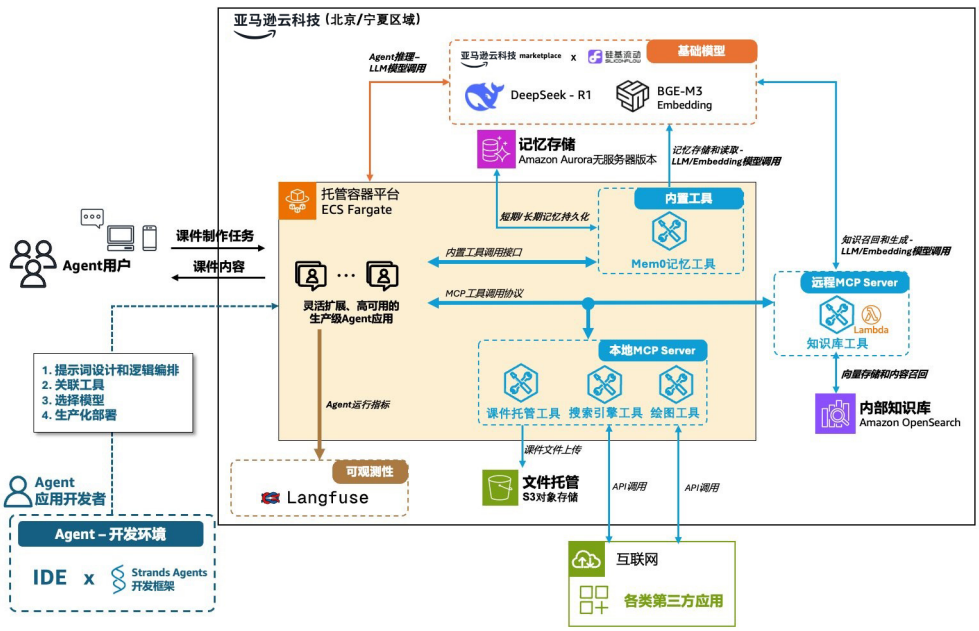
Strands Agents 的核心功能包括：

- **轻量级与灵活性。**Strands Agents 的核心是一个简单而强大的代理循环（agentic loop），它使模型能够自主地规划下一步行动并执行工具。这种设计既简单有效，又完全可定制，让开发者能根据需要调整代理的行为。
- **多模型支持。**Strands Agents 支持多种模型提供商，使开发者可以根据需求选择合适的基础模型，包括 Amazon Bedrock 模型（默认使用 Claude 3.7 Sonnet），Anthropic API，Llama API，Ollama 以及通过 LiteLLM 支持的其他模型。对于亚马逊云科技中国区来说，也支持通过亚马逊云科技 marketplace 调用硅基流动的 DeepSeek 的模型（https://awsmarketplace.amazonaws.cn/marketplace/pp/prodview-65lo53ldx6wda）。
- **丰富的工具集和 MCP 能力支持。**Strands Agents Tools 提供了一系列预构建的工具，可以直接与 Agent 集成使用。这些工具包括文件操作（读取、写入和编辑文件）、Shell 集成（安全地执行和交互式操作系统命令）、HTTP 客户端（进行 API 请求，支持各种认证方法）、Python 执行（运行 Python 代码片段，保持状态持久性）、数学工具（执行高级计算，具有符号数学功能）、无缝

访问亚马逊云科技的服务、图像处理（生成和处理图像）、环境管理（安全处理环境变量）以及思考工具（允许模型多次推理与自我反思）。原生支持 MCP，可以直接使用已有的 MCP server 作为 Agent 的工具扩展。

- **多 Agent 和自主 Agent 系统。**Strands Agents 支持构建复杂的多 Agent 系统，使多个 Agent 能够协同工作，解决更复杂的问题，比如通过工作流工具来编排多个 Agent 之间的任务，提供图工具来创建 Agent 关系网络，以及利用集群（Swarm）工具来协调并行问题的解决。
- **生产部署能力。**Strands Agents 生产环境提供了完整的工具集，比如通过集成如 LangFuse 等工具提供可观测性，从而进行追踪和调试。以及支持多种部署选项，包括 Amazon Lambda、Amazon Fargate、Amazon EC2 等多种部署架构。
- **安全和隐私。**Strands Agents 将安全和隐私作为设计原则之一，提供保护数据和确保安全执行的机制。

我们以一个实际案例，展示如何使用 Strands Agents 构建一个课件制作助手，从而帮助用户搜索各种素材、制作课件并提供用户下载课件。在该应用中，通过 Strands Agents 开发的 Agent 部署到适用于 Amazon ECS 的 Amazon Fargate 中，作为容器化服务，使用 Amazon Aurora Serverless 作为记忆管理的数据库，以及 Amazon OpenSearch 作为托管的 RAG 解决方案从而提供知识检索服务。通过中国区 Marketplace 来调用硅基流动的 DeepSeek 大模型或者海外区 Amazon Bedrock 上托管的其他领先大模型。并使用 Strands Agents 作为系统的中枢，协调管理网络搜索、文件系统、数据库及其他工具服务，为上层应用提供基础能力支持。整体架构图如下所示。



Strands Agents SDK 是这个系统的核心引擎，它提供了以下关键能力：

1. 支持多种模型

Strands Agents SDK 支持多种模型，只需要初始化不同的 Model Provider，就可以支持 OpenAI 兼容接口的模型（比如硅基流动在亚马逊云科技中国区 Marketplace 上提供的 DeepSeek 大模型）和 Amazon Bedrock 上的模型。以下这个核心示例代码中，演示了如何使用这两类模型来开发 Agent：

```
def _get_model(self, model_id, thinking, thinking_budget, max_tokens=1024, temperature=0.7):
    """ 根据提供商获取适当的模型 """
    if self.model_provider == 'openai':
        return OpenAIModel(
            client_args={
                "api_key": self.api_key,
                "base_url": self.api_base
            },
            model_id=model_id,
            params={
                "max_tokens": max_tokens,
                "temperature": temperature,
            }
        )
    elif self.model_provider == 'bedrock':
        # 创建自定义 boto3 会话
        session = boto3.Session(
            aws_access_key_id=self.env['AWS_ACCESS_KEY_ID'],
            aws_secret_access_key=self.env['AWS_SECRET_ACCESS_KEY'],
            region_name=self.env['AWS_REGION']
        )

        return BedrockModel(
            model_id=model_id,
            boto_session=session,
            cache_tools="default",
            cache_prompt="default",
            max_tokens=max_tokens,
            temperature=temperature,
            additional_request_fields=additional_request_fields,
        )
```

2.MCP 工具集成机制

Strands Agents SDK 提供了 MCP Client 能力，支持包括 stdio，streamable http，sse，自定义传输层四种类型的 MCP Server 集成。以下这个核心示例代码中，展示了 stdio，streamable http，sse 三种传输层类型的 MCP Server 如何连接。

```
async def connect_to_server(self, server_id: str, command: str = "",
                            server_script_path: str = "",
                            server_script_args: List[str] = [],
                            server_script_envs: Dict = {},
                            server_url: str = "",
                            http_type: str = 'stdio', token: str = ""):
    """ 使用 Strands MCP 客户端连接到 MCP 服务器 """

    if server_url:
        # 基于 HTTP 的服务器
        if http_type == 'sse':
            headers = {"Authorization": f"Bearer {token}"} if token else None
            mcp_client = MCPClient(lambda: sse_client(server_url, headers=headers))
        elif http_type == 'streamable_http':
            headers = {"Authorization": f"Bearer {token}"} if token else None
            mcp_client = MCPClient(lambda: streamablehttp_client(server_url, headers=headers))
    else:
        # 基于 Stdio 的服务器
        params = StdioServerParameters(
            command=command,
            args=server_script_args,
            env=server_script_envs
        )
        mcp_client = MCPClient(lambda: stdio_client(params))

    # 启动服务器
    mcp_client.start()
```

为了提高响应速度，系统会在第一次用户登陆访问时，把 MCP 工具预先加载激活，并存入该用户 session 的 MCP Clients 连接池中，当用户在一定时间内未有任何动作后，再从 MCP Clients 连接池卸载，释放资源。在以下这个核心示例代码中，展示了如何加载 MCP 工具并存放到用户 session 下的 MCP Clients 连接池中：

```
async def initialize_user_servers(session: UserSession):
    """ 初始化用户特有的 MCP 服务器 """
    user_id = session.user_id
    server_configs = await get_user_server_configs(user_id)
    global_server_configs = get_global_server_configs()

    # 合并全局和用户的服务器配置
    server_configs = {**server_configs, **global_server_configs}

    for server_id, config in server_configs.items():
        if server_id in session.mcp_clients:
            continue

        try:
            mcp_client = StrandsMCPClient(name=f"{session.user_id}_{server_id}")

            await mcp_client.connect_to_server(
                server_id=server_id,
                command=config.get('command'),
                server_url=config.get('url', ""),
                http_type="sse" if is_endpoint_sse(config.get('url', "")) else "streamable_http",
                token=config.get('token', None),
                server_script_args=config.get("args", []),
                server_script_envs=config.get("env", {})
            )

            session.mcp_clients[server_id] = mcp_client

        except Exception as e:
            logger.error(f" 用户 {session.user_id} 初始化服务器 {server_id} 失败 : {e}")
```

Agent 真正使用的是 MCP 服务器上的工具， 以下这个示例代码，展示了如何从连接池中取出 MCP Clients，并通过 list_tools_sync() 获取 MCP 服务器上的 MCP 工具，其已经抽象成了 AgentTool 对象，可以当做一个 python 工具直接被 Strands Agents 使用。

```
def get_tools(self, server_id: str) -> List[AgentTool]:
    """ 从特定 MCP 服务器获取工具 """
    if server_id not in self.active_clients:
        logger.error(f" 服务器 {server_id} 未激活 ")
        return []

    try:
        mcp_client = self.active_clients[server_id]
        tools = mcp_client.list_tools_sync()
        logger.info(f" 从服务器检索到 {len(tools)} 个工具 : {server_id}")
        return tools

    except Exception as e:
        logger.error(f" 从服务器 {server_id} 获取工具失败 : {e}")
        return []
```

3. 模型驱动的自主型 Agent

Strands Agents 是模型驱动为优先的 Agent 开发框架，我们创建一个自主型 Agent，只需要重点关注工具，系统提示词，上下文，模型，以下核心示例代码展示了该设计理念。在以下核心示例代码中，我们创建一个单 Agent，为其提供 MCP 工具和内置的长短期记忆管理工具 (mem0_memory)，指定模型，和上下文管理器 conversation_manager（用于管理最长的消息长度，避免 context 过长），以及系统提示词，就能很简单的创建一个具有自主推理并使用各种外部工具，以及长短期记忆和上下文管理的 Agent。

```
async def _create_agent_with_tools(self, model_id, messages, mcp_clients=None,
                                   mcp_server_ids=None, system_prompt=None,
                                   thinking=True, thinking_budget=4096,
                                   max_tokens=1024, temperature=0.7):
    """ 创建带有 MCP 工具的 Strands 代理 """

    # 创建 MCP 工具
    tools = await self._create_mcp_tools(mcp_clients, mcp_server_ids)

    # 添加内置长短期记忆工具 mem0_memory
    tools += [mem0_memory]
    # 获取模型
    model = self._get_model(model_id, thinking=thinking,
                            thinking_budget=thinking_budget,
                            max_tokens=max_tokens, temperature=temperature)

    # 创建 Agent
    agent = Agent(
        model=model,
        messages=messages,
        conversation_manager=SlidingWindowConversationManager(
            window_size=window_size,
        ),
        system_prompt=system_prompt or "You are a helpful assistant.",
        tools=tools
    )

    return agent
```

4. 流式响应处理

Strands Agents SDK 同时提供了封装之后的流式输出事件如（Agent 输出，工具调用结果），以及模型原生的流式输出事件，其中模型原生的流式输出事件统一转换成了 Amazon Bedrock converse stream API 响应格式。我们可以非常灵活的使用不同的事件类型，定制我们给前端 UI 的流式响应事件。以下示例代码，展示了如何定义一个 Agent，并且通过 Agent 的 stream_async 方法，取出响应事件，将工具调用的过程实时展示给前端 UI。

```
self.agent = await self._create_agent_with_tools(
        messages=history_messages,
        model_id=model_id,
        mcp_clients=mcp_clients,
        mcp_server_ids=mcp_server_ids,
        system_prompt=system_prompt,
        thinking=thinking,
        thinking_budget=thinking_budget,
        max_tokens=max_tokens,
        temperature=temperature
    )

    response = self.agent.stream_async(prompt)
    async for event in self._process_stream_response(stream_id,response):
        yield event
        # Handle tool use in content block start
        if event["type"] == "block_start":
            block_start = event["data"]
            if "toolUse" in block_start.get("start", {}):
                current_tool_use = block_start["start"]["toolUse"]
                tool_calls.append(current_tool_use)
                logger.info("Tool use detected: %s", current_tool_use)

        if event["type"] == "block_delta":
            delta = event["data"]
            if "toolUse" in delta.get("delta", {}):
                # 取出最近添加的 tool, 追加 input 参数
                current_tool_use = tool_calls[-1]
                if current_tool_use:
                    current_tooluse_input += delta["delta"]["toolUse"]["input"]
                    current_tool_use["input"] = current_tooluse_input

        # Handle tool use input in content block stop
        if event["type"] == "block_stop":
            if current_tooluse_input:
                # 取出最近添加的 tool, 把 input str 转成 json
                current_tool_use = tool_calls[-1]
                if current_tool_use:
                    current_tool_use["input"] = json.loads(current_tooluse_input)
                    current_tooluse_input = ''

        # Handle message stop and tool use
```

```
    if event["type"] == "toolResult":
            new_event = {}
            toolUseId = event['toolUseId']
            if toolUseId not in tool_results_dict:
                tool_results_dict[toolUseId] = event['data']
                # output tool results for UI
                tool_results_serializable = [[tool,{"tool_name":tool['name'],"tool_result":tool_results_dict.
get(tool['toolUseId'])}] for tool in tool_calls
                                        if tool_results_dict.get(tool['toolUseId']) and tool['toolUseId'] not in sent_results_
history ]
                # tool_results = [item for pair in zip(tool_calls, tool_results_serializable) for item in pair]
                tool_results = [item for pair in tool_results_serializable for item in pair]
                new_event = {'type':'result_pairs','data':{'stopReason':'tool_use','tool_results':tool_results}}
                yield new_event
                sent_results_history[toolUseId] = toolUseId

        if event["type"] == "message_stop":
            # Save the system to session
            self.system = system
            yield event
```

如果需要了解该课件制作助手更详细的信息和代码，可以查看：https://github.com/aws-samples/sample_agentic_ai_strands

2.3.2 沙箱环境

沙箱本质上是提供一个相对隔离的安全的运行环境，在 Agent 应用中，使用沙箱技术最主要的场景是执行大语言模型生成的代码，因为生成的代码具有不可预知性，因此建议使用安全隔离度高的运行环境。

在实践当中，如果只是用沙箱执行大语言模型生成的代码，并能接受以下情况的发生，则可以采用基于容器的沙箱隔离技术：

- 运行时容器可能导致的底层 Linux 内核崩溃或者挂住，从而影响整个宿主机上运行的其他容器或者进程的后果
- 跨容器攻击的发生
- 不需要向用户展现 Agent 系统的执行过程，比如 Compute Use 以及 Browser use 等技术。

否则需要使用更加严格的隔离技术，比如当前广泛应用的沙箱技术：E2B Sandbox 开源项目。作为最佳实践，建议将 E2B Sandbox 部署在亚马逊云科技上（可以参考 github 开源代码：https://github.com/aws-samples/sample-e2b-on-aws/ 了解如何把E2B Sandbox部署在亚马逊云科技上），从而获得独特的价值：

- 更专业的技术支持。E2B Sandbox 是基于亚马逊云科技开源的 Firecracker MicroVM 而构建。
- 更好的性能以及更低的部署成本。E2B Sandbox 的 Firecracker MicroVM 的部署方式，可以在虚拟机里部署 Firecracker MicroVM（即所谓的嵌套虚拟化），也可以是在物理机（Baremetal）里部署 Firecracker MicroVM。当前，亚马逊云科技上支持的是物理机部署 Firecracker MicroVM。该部署方式的性能比虚拟机里部署 Firecracker MicroVM 的性能更佳。同时在相同的硬件资源配置下，相比于虚拟机里部署 Firecracker MicroVM，在亚马逊云科技的物理机上运行数量更多的 Firecracker MicroVM，从而极大的降低了成本。

2.4 选择适合不同场景的方案

在详细介绍了 Amazon Q（专用 Agent）、Amazon Bedrock Agents（全托管 Agent）和 Strands Agents（完全自建 Agent）的原理、能力和典型应用场景后，我们对三种方案进行比较。

方案	部署方式	适用团队	主要特性与最佳场景
Amazon Q	SaaS	业务用户、开发者	• 开箱即用的生产力工具 • 适合代码生成、业务数据问答 • 零代码、直接在亚马逊控制台 /CLI/ 开发工具中使用
Amazon Bedrock Agents	全托管（Fully-Managed）	AI 开发者、业务团队	• 快速集成企业 API/ 知识库 • 自动化文档处理、RAG、多模态场景 • 无需管理底层基础设施 • 适合需要快速上线、合规性要求高但定制化需求中等的场景
Strands Agents	自托管（Self-Managed）	技术专家、AI 开发者	• 支持多模型以及自己部署的大模型 • 可运行在 Amazon SageMaker、Amazon EKS、Amazon EC2、Amazon Lambda 等基础设施上 • 企业有特定合规、治理、模型选择 • 对框架代码有极致的调优诉求

在选型时，我们建议：

- 如果业务需求是提升开发和业务生产力，追求开箱即用、零代码集成，或者需要面向特定岗位（如开发、客服、BI 分析）的智能助手，建议选择 Amazon Q。 适合开发辅助、运维助手、业务问答等场景。
- 如果希望快速集成企业 API、知识库，自动化处理业务流程，对安全合规有较高要求，但对定制化要求适中，建议选择 Amazon Bedrock Agents。适合保险理赔、客户服务、企业知识管理等场景。
- 如果追求极致的灵活性、需要自有模型的支持、同时希望自定义工具链，并期望对框架有极致的优化，并且团队具备一定的 AI 开发和运维能力，选择 Strands Agents。 构建高度定制化、复杂的 Agentic AI 系统，如自主研究助手、跨系统自动化、生产级 AI 平台等。

应用 Agentic AI 技术进行构建的客户案例

3

3.1 金蝶国际软件集团利用 Agent 技术优化 ERP 系统智能提单和指标分析流程

金蝶国际软件集团有限公司（以下简称金蝶）是全球知名的企业管理云 SaaS 公司，成立于 1993 年，已为世界范围内超过 740 万家企业、政府组织提供企业管理云产品及服务。为进一步提升海外用户的 ERP 使用体验，金蝶采用亚马逊云科技的生成式 AI 服务，利用 Amazon Bedrock Agents 构建智能提单系统和智能指标分析系统，2 个月完成生成式 AI 功能落地，以自然语言交互简化用户提单，优化智能指标查询业务流程。

3.1.1 业务痛点

设想一位刚入职不久的财务专员，要为部门新采购的办公设备进行报销，由于设备类型、采购方式、供应商条款、单据种类等不同，可能涉及系统中数十个单据，以及数百个页签，面对 ERP 系统庞杂的界面和功能，财务专员一时无从下手，如果参考百科全书式的 ERP 系统用户手册一步步按章操作会非常低效。再设想一位企业高管，在外与客户进行商务沟通时，想要即时了解上个季度某子公司某项产品的研发投入和同比增长情况，而系统并无现成数据参考，则会希望有一个自助式的查询工具，能够便捷高效、随时随地了解具体各类业财数据。

3.1.2 构建智能提单和指标分析系统

要在金蝶的业务场景中应用生成式 AI 精准匹配客户需求，依靠基础模型本身远远不够，还需要集成公司内部系统、API 以及外部系统的数据。Agentic AI 的技术出现后，可以从专有数据源检索所需信息并提供准确且相关的响应，精准捕捉用户意图，十分适合前面提到的业务场景。

金蝶利用亚马逊云科技 Amazon Bedrock Agents 对其面向海外客户的提单系统进行优化，构建智能提单系统，将原有 ERP 中基于 GUI （Graphical User Interface，图形用户界面）的表单界面提升为 LUI （Language User Interface，自然语言用户界面）。用户通过自然语言描述后，生成式 AI 能够帮助分析和推导其意图，并给出操作提示，用户按照提示即可简单的发起报销、借款、出差等各类单据提交。

除智能提单外，Agents 还可应用于智能指标分析场景。金蝶应用亚马逊云科技 Amazon Bedrock Agents 面向海外客户构建智能指标分析系统，企业管理者可与线上专属 AI 助理进行自然语言对话，获得即时、精准的指标数据查询，避免了在繁杂的 BI 报表中寻找数据的烦恼。在海外业务场景中，与用户确认意图后，金蝶 AI 助理自主拆解任务，并检索企业知识库，匹配推荐相关度最高的指标。企业管理者就像拥有了 7x24 小时的 1v1 专属财务助手，轻松查询各类财务指标对比分析，重塑了企业的分析和决策流程。

3.1.3 采用 Agent 解决方案以后达成的业务成果

面向海外客户的智能提单系统落地后，想要发起 ERP 提单流程的用户，再也不用在 ERP 系统上百个应用、每个应用数百个页签的复杂界面前无所适从，仅仅通过自然语言对话描述，就能自动发起报销、借款、出差等各类单据提交，显著简化了用户与系统的交互方式，极大提升了用户提单体验。

采用生成式 AI 技术后，通过面向海外客户的智能指标分析系统，企业管理者能够随时随地查询企业业财状态和数据，且查询的准确率显著提升。在金蝶的实践中，针对简单场景的查询，第一轮答复准确率超过 90%；针对复杂场景的查询，例如带有指标趋势计算的聚合逻辑，第一轮答复准确率为 70% ~ 80% ；借助 Agents 模式与用户进行对话后，如补充具体数据、来源细节后，第二轮答复准确率可达 100%。通过将生成式 AI 技术融入了企业的业务流程，优化了企业使用 ERP 系统的方式和过程，进一步提升企业运营效率。

3.2 Formula 1 利用 AI Agent 加速比赛日各种问题的根因分析

一级方程式赛车（F1®）在赛事期间需要处理和解决各种复杂的技术问题，这些问题的诊断和解决可能需要长达 3 周的时间。通过利用 Amazon Bedrock Agents 和 Amazon Bedrock Knowledge Bases，F1 开发了智能根因分析（RCA：Root Cause Analysis）助手，从而实现了用自然语言交互式进行问题的诊断。

3.2.1 F1 的业务痛点

一级方程式赛车赛事是分秒必争的战场，运营效率至关重要。在赛事直播期间，F1 的 IT 工程师必须快速诊断各类关键问题——比如某个 API 的网络性能下降。这会波及依赖该 API 数据的下游服务，包括提供每场比赛直播 / 点播内容和实时遥测数据的 F1 TV 等产品。定位问题根源并防止复发往往需要耗费大量精力。由于赛事日程和变更冻结期的限制，完成关键问题的诊断、测试和解决可能长达 3 周，需要开发、运维、基础设施和网络等多团队协同调查。

3.2.2 构建智能根因分析助手

F1 与亚马逊云科技的技术团队共同合作，在搭建了日志系统的基础之上，使用 Amazon Bedrock 构建了基于 Agent 的 RAG 知识库，从而支持该智能根因分析助手。Amazon Bedrock 是一项完全托管的服务，供了领先的基础模型和 RAG 工作流程功能。F1 使用 Amazon Bedrock Knowledge Bases 来索引和检索转换后的日志数据。Amazon Bedrock Knowledge Bases 会自动将文本分成块，创建嵌入，并构建向量索引以实现高效检索。

Amazon Bedrock Agents 与 Amazon Bedrock Knowledge Bases 无缝集成，为终端用户提供了一个集成的、针对用户问题的回答和响应。智能根因分析助手自动创建执行计划，去调用相关的工具（比如查询数据库，提交 Jira 工单等工具）和查询知识库。当 Agent 接收到知识库返回的文档以及从工具 API 返回的结果时，则把这些返回的数据合并起来传给大语言模型，从而得到最终的响应。

3.2.3 根因分析助手带来的业务价值

通过使用该智能根因分析助手，工程师团队能够在 3 天内定位到问题的根本原因，并实施解决方案，包括在比赛周末进行部署和测试。该系统不仅节省了解决问题的时间，还将问题路由到正确的团队进行解决，使团队能够专注于其他高优先级任务，如开发新产品以提升比赛体验。通过使用生成式 AI 技术，工程师可以在 5-10 秒内收到特定查询的响应，并将问题初始分类时间从超过一天减少到不到 20 分钟，端到端的问题解决时间最多减少了 86%。

总结和展望

4

Agentic AI 代表了人工智能领域的一次重大飞跃，它不再局限于被动响应指令，而是能够主动感知环境、制定目标、规划行动并自主执行复杂任务。通过结合大型语言模型（LLM）、多模态感知、强化学习和自动化工作流技术，Agentic AI 能够以更接近人类决策的方式运作，从而在各行各业都展现出来巨大潜力。

在前面的章节中，我们探讨了 Agentic AI 的核心概念，深入描述了在亚马逊云科技上构建和部署 Agentic AI 应用的最佳实践。亚马逊云科技提供了一系列强大的工具和服务，如 Amazon Bedrock，开源的 Strands Agents 等，这些服务共同构成了一个完整的 Agentic AI 开发生态系统。通过利用亚马逊云科技高效的生成式 AI 服务、可扩展的计算能力、安全的数据存储和网络等基础设施，企业和开发者能够快速地构建高可用、可扩展的 Agentic AI 应用。

Agentic AI 的发展势头不可阻挡，其未来的应用场景也将更加广泛。Agentic AI 不仅是技术的进步，更是人机关系的一次重塑。它将在未来深刻地改变我们的工作方式、商业模式和社会结构。对于企业和开发者而言，现在正是探索 Agentic AI 的最佳时机。通过结合亚马逊云科技的先进工具和最佳实践，我们可以构建出更智能、更自主的 AI 系统，推动社会进入真正的智能化时代。未来已来，而 Agentic AI 将成为这场变革的核心驱动力。

同时，也要看到，本文于 2025 年 6 月发布，而 Agentic AI 技术正在高速发展中，技术的迭代和更新也非常迅速，如果要了解有关 Agentic AI 的最新技术和方案，请关注亚马逊云科技的官网了解最新信息。

亚马逊云科技