AgentSM：面向智能体式文本到 SQL 的语义记忆

Asim Biswal[△°]、Chuan Lei[*◇]、Xiao Qin[*□]、Aodong Li[°]、
Balakrishnan Narayanaswamy[°]、Tim Kraska[°]
◦亚马逊云科技 △加州大学伯克利分校 ◇甲骨文公司 □*Snowflake* 公司
abiswal@berkeley.edu,chuan.lei@oracle.com,xiao.qin@snowflake.com
{aodongli,muralibn,timkrask}@amazon.com

arXiv:2601.15709v1 [cs.AI] 22 Jan 2026

摘要

近年来，基于大语言模型的文本到 SQL 技术不断进步，在 BIRD 和 Spider 等公开基准测试上取得了显著提升。然而，这些系统难以扩展到现实的企业环境中，因为这类环境具有庞大而复杂的模式、多种 SQL 方言，以及成本高昂的多步推理过程。新兴的智能体方法展现出自适应推理的潜力，但往往效率低下且不稳定——反复与数据库交互、生成不一致的输出，偶尔还会无法生成有效答案。为应对这些挑战，我们提出智能体语义记忆（AgentSM），这是一种构建并利用可解释语义记忆的文本到 SQL 智能体框架。AgentSM 不依赖原始草稿板或向量检索，而是将先前的执行轨迹——或经整理生成的轨迹——作为结构化程序加以记录，直接引导后续推理。这种设计能够系统地复用推理路径，使智能体能够高效、可靠地扩展到更大的模式、更复杂的问题和更长的轨迹。与最先进系统相比，在 Spider 2.0 基准测试上，AgentSM 将平均令牌使用量和轨迹长度分别减少了 25% 和 35%，从而提升了效率。它还提高了执行准确率，在 Spider 2.0 Lite 基准上达到 44.8% 的最先进准确率。

1 引言

文本到SQL旨在通过将自然语言任务转换为可执行的SQL查询，实现与结构化数据的交互。这项能力在企业数据分析和商业智能领域尤为重要，因为非技术用户需要从规模庞大、结构复杂的数据库中提取洞见，却无需掌握SQL或深入了解数据模式。

近年来，大语言模型（LLM） [37], 提示策略 [21], 和后训练技术 [34]的进步，推动了文本到SQL在BIRD [16] 和Spider [35]等基准测试上的性能显著提升。然而，现有的大多数文本到SQL系统仍难以在真实的企业环境中扩展应用；深层嵌套模式、多样的SQL方言以及特定领域的业务逻辑等挑战，都会导致准确率和效率下降。

为了在更贴近现实的条件下评估文本到 SQL 系统，近期推出了 Spider 2.0 基准测试 [14]，其中包含复杂的多方言数据库和极长的上下文。传统的文本到 SQL 方法——包括基于向量的模式检索 [15], 、候选生成与多数投票 [21], ，以及自一致性解码 [13, 27]——单独应用于这一基准测试时都表现不佳，暴露出它们在泛化和扩展能力方面的根本局限。这些缺陷促使人们开始探索。人们对智能体式文本到 SQL 方法的兴趣日益浓厚 [7, 26, 30], ，智能体会与数据库进行迭代交互，以检查模式、验证部分查询，并适应方言差异。

尽管智能体系统展现出更强的适应大型复杂模式的能力，但计算成本与准确率之间存在难以权衡的问题，使这类系统难以扩展应用。智能体往往会产生大量**冗余**，在同一数据库上针对不同查询反复执行相同的**探索步骤**，从而推高执行成本和延迟。此外，它们还容易出现**规划方差**，即初始推理路径不理想，导致结果不一致或执行失败。而且，这类系统本身具有迭代特性，因此会带来**较高的计算成本和延迟**，因为每个步骤都会消耗额外的令牌和时间。在步骤数、令牌数或延迟预算等实际约束下，这些低效之处会显著降低总体准确率和效率 [17]。

为了解决这些局限，我们提出了智能体语义记忆（AgentSM），这是一个面向文本到SQL的可扩展、稳定且高效的智能体框架。AgentSM 不再将每个查询视为孤立的交互，而是引入结构化语义记忆，用于捕获并复用以往执行中的轨迹。每条轨迹都会添加语义标注后存储，以支持后续检索和推理。当新查询针对一个已知数据库到来时，AgentSM 会复用以往轨迹中的相关部分，从而消除冗余探索，确保行为更加一致。此外，频繁共同出现的工具序列会自动组合成复合工具，从而缩短轨迹并提高执行效率。

除文本到SQL外，AgentSM 还为其他智能体数据任务提供了可泛化的基础，包括信息抽取、数据清理和数据转换；在这些任务中，结构化轨迹复用对于实现可扩展的高质量性能至关重要。

总而言之，我们的贡献如下：(1) 我们提出了 AgentSM，这是一种智能体框架，能够捕获并利用结构化语义记忆，应用于企业级文本到SQL。(2) 我们设计了一种结构化语义记忆，以可解释、可检索的格式编码过往轨迹。该记忆使智能体能够检索相关的过往经验，从而提升效率并增强推理一致性。(3) 我们为 AgentSM 引入了复合工具，以简化决策、降低延迟，并缓解复杂多步查询生成过程中智能体规划和工具使用中的幻觉。这些工具减少了智能体轮次和令牌使用量。(4) 我们进行了大量实验，结果表明，AgentSM 在 Spider 2.0 Lite

*本研究工作在亚马逊云科技完成。

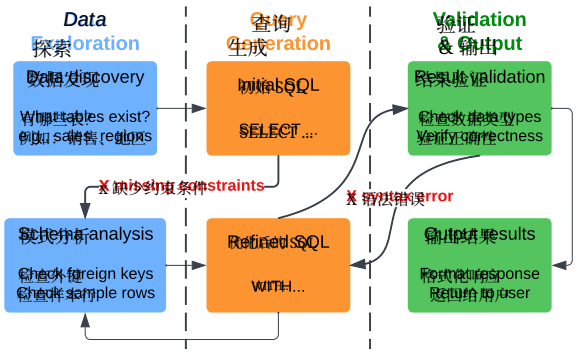


图 1：文本转 SQL 智能体的标准工作流程由一系列步骤组成，在数据探索、查询生成和答案验证之间交替进行。

基准测试上达到了 44.8% 的最先进准确率。消融研究进一步表明，AgentSM 能有效将平均轨迹长度缩短 25%，并将执行准确率提高 35%。

2 背景与问题表述

2.1 智能体式文本到 SQL

现代的 ReAct 智能体 [33] 遵循标准的推理与行动步骤模式。在每一步中，智能体都会观察任务的当前状态，推理下一步适当的行动，并执行该行动，而这通常涉及工具使用。例如，代码智能体会观察程序的中间状态或执行输出，推理需要进行的修改或后续步骤，并通过生成和执行代码采取行动，逐步推进至最终解决方案。

图 1 展示了智能体式文本到 SQL 系统的标准工作流程，通常分为三个阶段：数据探索、SQL 查询生成与执行，以及响应验证。在数据探索阶段，智能体会探索并理解数据库模式，找出完成任务所需的相关表、列和值。在企业环境中，模式信息通常会预先解析并存储在文件中，智能体可以参考这些信息，并执行简单的探索性 SQL 查询来理解数据。随后，智能体尝试生成 SQL 查询来回答给定任务，通常会分步生成，再综合得出最终查询。如果没有遇到错误，智能体可以选择验证响应，也可以直接输出响应。如果发生错误，例如语法违规或查询结果与预期不符，智能体可以根据错误的性质修改 SQL，或返回数据探索阶段以获取更多上下文。

我们发现了这些标准工作流程中的三个关键机遇，它们为我们提出的解决方案提供了启发。

2.1.1 重复探索。 对智能体推理模式的分析表明，在同一数据库的不同查询中，数据探索本质上具有重复性 。事实上，Liu 等人 [17] 指出，智能体在 BIRD 基准 [16], 上的轨迹具有重复性，报告称不同轨迹所占比例不到 10–20%。

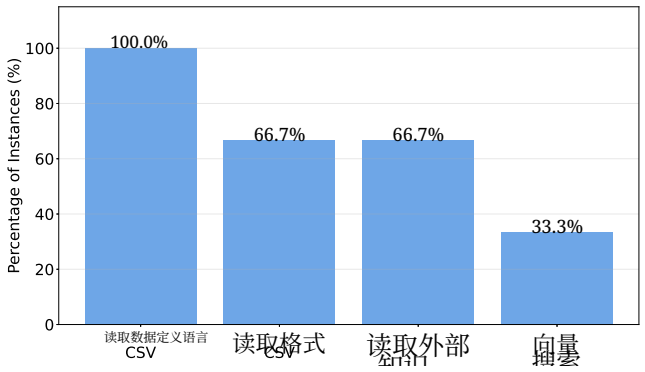


图 2：Spider 2 [14]，报告中，智能体在 ‘firebase’ 数据库上轨迹的前七个步骤分布情况，展示了采取各项特定操作的实例百分比。

我们在通用文本到SQL场景中的智能体身上也观察到了相同的行为。在初始步骤中，智能体会一贯地执行一系列基础探索操作。它首先读取模式文件或执行 PRAGMA 查询，以了解数据库模式，包括表结构、列名和数据类型。如果有外部知识文件，智能体会读取这些文件，以获取有关数据库或任务的更多上下文。如果智能体需要额外帮助来处理特别庞大的模式，就会借助向量搜索等工具，或执行更多 SQL 查询，以了解候选表的内容。

机遇 ***1.*** 复用先前轨迹，以消除冗余探索，并提高针对同一数据库的新问题的处理效率。

2.1.2 策略选择。 传统文本到 SQL 系统已逐渐趋向高度精确的流水线式方案，对每个问题都采用固定的步骤序列 [21, 27]。类似地，智能体式文本到 SQL 方法仍高度依赖提示工程 [7], , 通过精心设计的指令来定义智能体的工作流程和工具使用，以确保一致性。

然而，对所有查询都强制采用单一主导策略，在准确率和效率方面都可能并非最优。标准化流程对于常见情况可能有效，但往往无法处理分布外查询。例如，图 2 显示，在 Spider 2.0 的 ‘firebase’ 数据库中，智能体仅在 30% 的问题上选择执行向量搜索，这可能是因为该数据库具有嵌套模式，且智能体通过先前的探索步骤积累了足够的先验知识。在这类情况下，遵循僵化的策略可能会引入无关上下文、造成不必要的计算并浪费步骤，而这些资源原本可以用于采取更有针对性的操作来解决给定查询。

机遇 ***2.*** 根据每项任务和数据库的特点动态调整轨迹，而不是依赖固定、千篇一律的策略。

2.1.3 降低方差。 文本到SQL生成中的智能体行为在不同运行之间天然存在方差。中间步骤中的微小偏差（包括无关紧要的语法错误）可能会使

推理过程偏离正轨，导致智能体放弃正确的推理路径，转而采取另一条路径。这种不稳定性会导致查询生成不一致，进而使最终结果出现差异 [31]。

即使将智能体的模型温度设为零，也无法可靠地强制实现确定性行为。此外，即使能够实现严格的确定性设置，这也会阻止智能体探索其他推理路径，限制其为此前未见过或复杂的任务发现解决方案的能力 [36]。常见的文本到SQL策略，例如候选生成和查询优化，明确依赖受控的模型方差，以提高生成正确 SQL 的可能性。然而，一旦在推理流程中引入新的上下文，后续步骤就可能出现不可预测的偏离，即使在确定性设置下也是如此。

机会***3.*** 优化工具复杂度与轨迹长度之间的权衡，以尽量降低方差并提高准确率和稳健性。

2.2 问题陈述

本文旨在开发一种智能体式文本到SQL系统，该系统接收自然语言查询 q、数据库 D 和一组工具 U，并生成优化后的推理轨迹

$$\tau(q, D, u) = \arg \max_{\tau \in T(q,D,\mathcal{U})} \mathrm{Acc}(\tau)$$

其中， $T(q, D, u)$表示由工具使用和中间推理步骤构成的可执行推理轨迹空间，而 $\mathrm{Acc}(\tau)$ 则衡量生成的 SQL 查询的准确率。

3 方法

图 3 展示了 AgentSM框架的整体概览。该框架由两个智能体组成，并采用轨迹读取方法和精心设计的工具，以提升智能体在文本到 SQL 任务中的表现。

3.1 AgentSM架构

智能体。我们在框架中引入了两个主要智能体：规划智能体和模式链接智能体。受 ReAct框架启发，智能体在观察、推理和行动循环之间交替进行。在我们的实现中，两个智能体都是编码智能体，通过编写 Python 代码来采取行动。然而，与以往方法不同， [14], 我们不限制智能体可编写的函数或代码，不将其限定在有限的预定义行动空间内。相反，我们提供了多种获准使用的库，供智能体用于数据探索和分析，包括 Pandas 和 NumPy。

作为系统核心推理组件的**规划智能体** 会构建高层执行计划，迭代发出 SQL 查询，进行推理和数据整理，并在生成最终答案前优化查询。不同于将 SQL 生成工作交由专门智能体的多智能体流水线[29],，我们有意将 SQL 生成整合到规划智能体的推理循环中。这一设计确保规划智能体能够直接利用探索过程中获得的模式和数据理解，避免控制权移交给专门子智能体时常见的上下文丢失。此外，通过让规划智能体生成并执行 SQL，AgentSM 避免了不必要的智能体间通信开销和延迟。

模式链接智能体由规划智能体管理，其作用是在规划智能体需要深入检查候选表时，针对任务进行细粒度的数据探索。它可以使用专用工具，包括向量搜索工具，该工具会在预先计算的模式信息索引上执行相似性搜索，以筛选出与给定问题相关的表和列。模式链接智能体的操作预算较少（例如，5 个步骤），可用于执行基本查询以解读嵌套结构、探查候选列并验证潜在的连接路径，然后将其发现（即将问题词元映射到模式元素）返回给规划智能体。将深入探索委派给单独的智能体，可以避免规划智能体的推理偏离方向，并确保其上下文中只保留相关信息。

尽管先前的工作提出了复杂的多智能体系统（例如，引入查询修正器、分解器或评估器 [7]），我们仍刻意让 AgentSM 的架构保持简洁，仅包含两个紧密协作的智能体。这一决策基于实证发现 [9,28]：过度的智能体分解会带来通信开销、造成记忆碎片化，并增大行为方差。在文本到 SQL 任务中，上下文知识往往高度局部化：要理解模式的嵌套层级，需要在多个探索与推理周期中维持持久状态。因此，引入更多智能体就必须配备共享记忆 [4, 24]，以避免上下文碎片化。然而，共享记忆会在检索效率、一致性管理和错误传播方面带来不小的挑战，这些问题留待未来工作解决。

工具。 AgentSM 中的每个智能体都配备了一组经过精心筛选的工具，旨在兼顾灵活性、稳健性与效率。规划智能体配备了用于轨迹检索、输出验证和结果保存的工具。轨迹检索工具可让规划智能体在开始新任务前查看语义相似问题的推理轨迹，帮助其避免冗余的探索步骤。验证和保存工具可确保最终的 SQL 及结果格式一致，并能够正确持久化。另一方面，模式链接智能体配备了专用于细粒度模式探索的工具，尤其是向量搜索工具，它可以从预先计算的索引中检索语义相关的表和列。将此功能限定给模式链接智能体，可以避免两个智能体之间的重复搜索。

两个智能体共用一组通用工具，用于探索本地目录，并在 BigQuery 、 Snowflake 和 SQLite 等受支持的后端上执行 SQL。为提高执行的稳健性，我们在 SQL 执行工具中加入了自我优化机制。当查询失败或返回空结果时，SQL 执行工具会根据收到的反馈自动尝试修正。这一机制降低了单次执行错误导致推理轨迹中断的可能性。

除了这些基本工具之外，我们还引入了复合工具，这类特殊工具将经常在连续步骤中共同出现的单一用途工具的逻辑组合起来。第 3.3 节介绍了这些复合工具，它们简化了规划，减少了不必要的工具使用，并提高了不同运行之间的一致性。

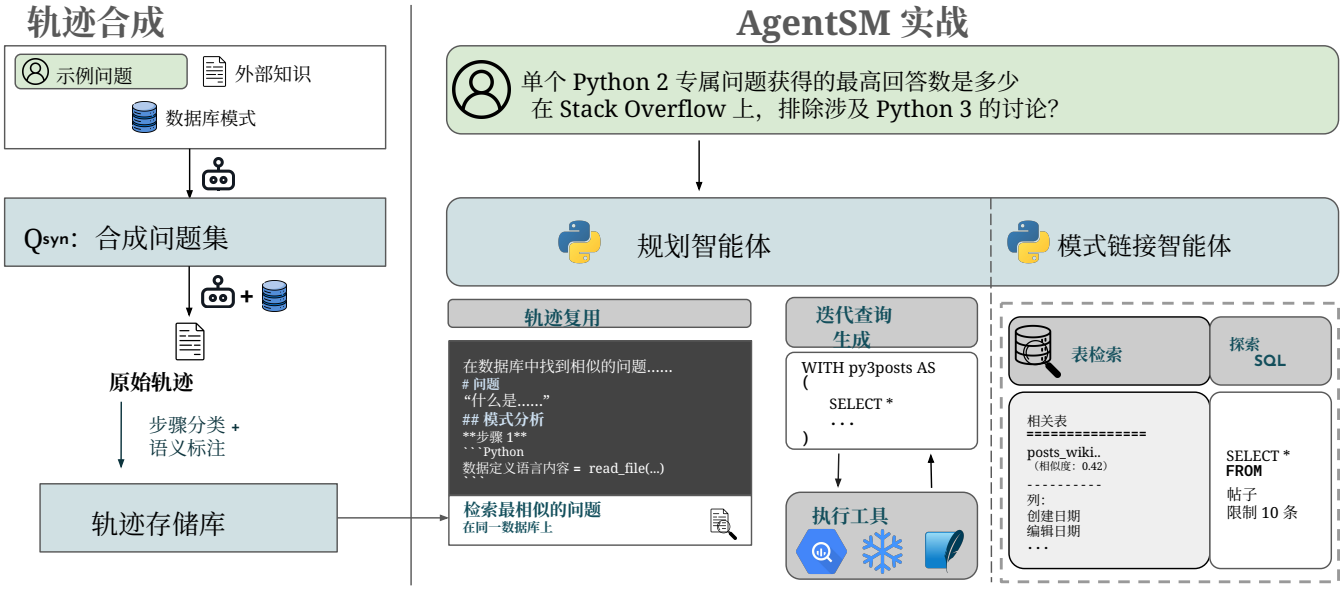


图 3：AgentSM架构概览，该架构利用结构化语义记忆中的轨迹。

3.2 轨迹合成与检索

合成问题探索。 我们发现一个关键事实：数据探索本质上具有可复用性。一旦智能体检查过模式、查看过列并执行过探索性查询，这些交互轨迹对于在同一数据库上处理后续问题仍然很有价值。至关重要的是，这一特性也适用于合成问题，即那些可以离线提出并执行、且专门设计用于促使数据探索的问题。通过在轨迹存储库中丰富此类包含大量探索内容的轨迹，AgentSM 有效减少了重复探查，并提升了智能体处理真实查询的表现。

算法 1 概述了生成完整合成问题集 Q_{syn} 的过程。首先，我们确保查询分布中的每个数据库至少对应一个合成问题（第 2–4 行）。随后，我们根据各数据库在查询集中的分布比例，在不同数据库间分配额外的合成问题，直至达到固定的问题总预算 n （第 6 行）。这种分配策略会自然地侧重于大型、复杂的模式，因为额外的探索轨迹在这些模式上最有帮助（第 7–11 行）。

对于每个选定的模式 S，我们向大语言模型提供该模式、外部知识文件以及一组现有问题，并提示其生成多样化的候选问题，涉及不同的运算符、表和列。每个模式生成的问题数量遵循完整查询集的相同分布。用于生成候选问题的提示可在代码仓库 [1] 中找到。

随后，智能体仅使用 SQL 和文件读取工具处理每个合成问题 $q \in Q_{\text{syn}}$，生成一条轨迹 $T(q)$，并将其存储以供复用。虽然这些合成问题的答案不会被评估，但生成的轨迹提供了密集且多样的探索轨迹。当智能体

算法1：合成问题生成。GenQ(S, K, Q) 使用大语言模型，根据数据库模式 S、外部知识 K 和现有问题 Q，为 $S \in$ D 生成一个新问题。Allocation(S, n, p) 根据查询分布 p(Q) 和剩余全局预算 n，为 S 返回一个整数预算 k，表示最多可生成的问题数量。输入：数据库 D，目标数量 n，查询分布 p(Q) 输出：合成问题集 Q_{syn} [1] $Q_{\text{syn}} \leftarrow \emptyset$; [2] 对每个模式 $S \in$ D 执行 [3] $q \leftarrow$GenQ(S, K, Q); [4] $Q_{\text{syn}} \leftarrow Q_{\text{syn}} \cup \{q\}$; [5] 当 $|Q_{syn}| < n$ 时执行 [6] $k \leftarrow$ Allocation$(S, n, p(\text{Q}))$; [7] 对 $i \leftarrow 1$ 到 k 执行 [8] 如果 $|Q_{syn}| = n$ 则 [9] 跳出 [10] $q \leftarrow$ GenQ(S, K, Q); [11] $Q_{\text{syn}} \leftarrow Q_{\text{syn}} \cup \{q\}$; [12] **返回** Q_{syn};

未来遇到新的查询时，这些轨迹能够提供丰富的上下文，让它利用过去的探索，而不是重复探索。

步骤分类。智能体工作流并非线性，而是由一系列不同的状态组成，智能体在执行过程中会进入和退出这些状态。实际中，文本到SQL智能体会在图 1 所示的三个主要阶段之间交替进行：（1）探索数据库以了解数据；（2）制定并执行候选

[1] https://tinyurl.com/xcey6h33

方法	平均步数	准确率（%）
无轨迹	22.62	25
朴素	22.62	25
Markdown	16.50	50
JSON	15.12	50

表 1：在不同轨迹格式下，Spider 2.0 Lite 部分问题的平均轨迹长度（以步骤计）和执行准确率。与不使用任何轨迹相比，读取原始、非结构化轨迹并不能带来提升。相比之下，无论采用 Markdown 还是 JSON 格式的结构化表示，都能显著减少平均步骤数并提高总体执行准确率。

查询，以及（3）验证结果或生成答案。先前的工作中也观察到了类似的阶段划分 [17]。

将轨迹复用建立在这些阶段之上至关重要：探索步骤通常可以在不同问题之间泛化；执行步骤体现了特定数据库推理；验证步骤则凸显了错误纠正和输出格式化模式。步骤分类有助于避免向智能体呈现由不同工具调用混杂而成的嘈杂序列，否则会令复用失效。

为了对步骤分类，我们采用轻量级的基于正则表达式的文本模式匹配，而不是依赖步骤在轨迹中的位置。虽然探索通常发生在早期，但智能体在遇到执行错误或上下文缺失时，可能会在之后重新进入探索阶段。因此，我们根据工具使用和查询模式来识别意图。例如，文件列表和读取操作会被归为探索，而以 'WITH'（公用表表达式）开头的复杂 SQL 查询几乎总会被识别为主要查询执行。我们也探索了基于大语言模型的步骤分类，发现其准确率相当，但成本和延迟显著更高。

每个分类后的步骤都会被解析并保存到单独的轨迹文件中。在轨迹读取过程中，智能体只选择性地加载与相关阶段对应的轨迹片段，以供参考。例如，在数据探索阶段，智能体会检索并读取相关的探索轨迹，其中包含基本的模式分析和外部知识读取操作，从而能够高效复用先前上下文，避免执行冗余步骤。

轨迹结构。 智能体轨迹通常是冗长且复杂的序列，其中交错包含文本、代码、推理轨迹和执行输出。直接读取原始轨迹会引入大量噪声，并加剧中间信息丢失 效应，即有价值的信息被埋藏在冗长的智能体调试日志和中间输出之中。

为了帮助智能体更有效地提取有用知识，我们对每条轨迹施加结构化表示。具体而言，我们以 Markdown 格式存储智能体轨迹，并使用轻量级大语言模型（例如 Claude Haiku 4.5）自动生成具有语义意义的标题。该结构将轨迹划分为易于理解的部分，从而同时提升检索效率和可读性。

如表 1 所示，无论是以 Markdown 还是 JSON 表示，结构化轨迹都能显著提升执行准确率，并在 20 个基准测试问题的样本中，相比非结构化日志大幅减少平均步骤数。我们采用 Markdown 格式，因为它与外部知识文件保持一致，且便于人类阅读。步骤分类和轨迹结构生成的实现细节可在代码仓库中找到[2]。

轨迹选择。 在建立了用于存储智能体轨迹的结构化表示后，接下来我们介绍如何在推理过程中检索和复用这些已存储的轨迹。在为合成问题 Q_{syn} 合成并保存轨迹后，我们首先将检索范围限制为与给定问题 q使用同一数据库的轨迹。这一筛选缩小了搜索空间，使其聚焦于相关轨迹：

$$Q_d = \{q' \in Q_{\text{syn}} \mid \text{db}(q') = \text{db}(q)\} \tag{1}$$

在这个筛选后的子集中 Q_d，我们随后根据关联问题 q'，选择语义相似度最高、与给定问题 q最相关的轨迹 T：

$$q^* = \arg\max_{q' \in Q_d} \text{sim}(q', q) \tag{2}$$

$$T = \text{traj}(q^*) \tag{3}$$

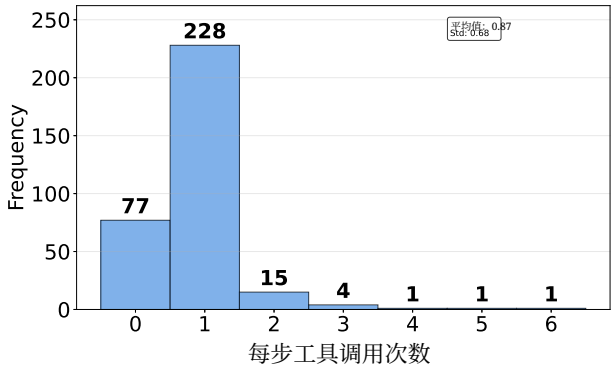
这一策略对于数据探索复用十分有效，因为语义相似的问题在初始探索步骤中往往会探查同一组表。智能体在进行更深入的搜索之前，自然会优先考虑模式与问题在语义上相似的表。因此，根据问题相似度选择轨迹能够有效实现轨迹复用。通过更细粒度地对齐问题与存储的轨迹，可以进一步提升检索质量。例如，模式级检索可以识别与问题相关的表，并将轨迹检索限定在涉及相同表的轨迹中。或者，轨迹级对齐可以利用智能体的初始计划检索相似的先前执行，并在执行之前调整推理策略。我们将这些方向留待未来工作，因为细粒度检索可能会遗漏准确推理所必需的相关上下文。

3.3 复合工具

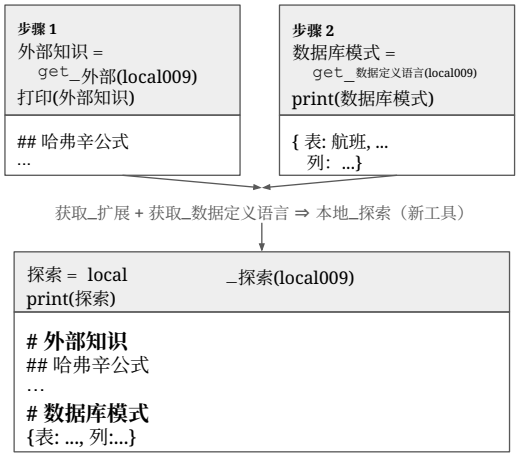
复合工具构建。 在多条轨迹中，我们观察到某些工具对总是按顺序连续出现。图 4b 展示了当智能体在探索阶段开始时需要访问外部知识文件和数据库模式时，get_ext → get_ddl 模式经常出现。因此，这些工具可以组合成一个名为 local_exploration 的复合工具。

如图 4a 所示，智能体很少在单个推理步骤中连续调用多个工具，而是每个步骤只调用一次，即使这类调用序列具有很强的规律性和确定性。这种模式会导致效率低下：智能体将宝贵的步骤耗费在执行重复的探索流程上，而非专注于更高层次的推理和问题求解。

[2]https://tinyurl.com/492ymhcn



(a) 327 个步骤中每步工具调用次数的分布



(b) 复合工具构建。

图 4：工具使用与复合工具构建分析。(a) 显示，大多数步骤使用零个或一个工具，这促使我们进行组合。(b) 展示了如何将频繁共同出现的工具合并为更高层级的复合工具。

为减轻这一开销，我们引入复合工具，将经常共同出现的工具序列组合成单个更高层次的操作。形式上，我们构建一个复合工具：

$$C = \langle t_1, t_2, \ldots, t_k \rangle \quad \text{if} \quad \frac{f(t_1, t_2, \ldots, t_k)}{N} \geq \tau$$

其中，$t_i \in$T 表示工具集，$f(\cdot)$ 统计包含子序列 (t_1，t_2，……，t_k) 的轨迹数量，N表示轨迹总数，τ表示选定的支持度阈值。

在实践中，AgentSM 遵循若干启发式规则，在保持模块化的同时构建有效的复合工具。首先，仅当工具在同一推理阶段（例如数据探索或验证）中共同出现时，才会将它们组合起来，以确保复合工具在语义上保持连贯。其次，为了保留工具在不同上下文中的复用灵活性，不会将跨多个阶段出现的工具纳入组合。最后，为避免过度聚合，我们限制复合工具的最大规模，防止所有工具坍缩为一个单体算子。

随后，大语言模型会为每个复合工具指定一个描述性名称并生成自然语言摘要，供智能体在提示过程中引用。在推理时，智能体会直接使用这些复合工具，以单个高级操作取代重复的多步探索，从而为推理留出更多轨迹预算。

4 评估

本节中，我们评估 AgentSM 在 Spider 2 上的有效性。Spider 2 是智能体文本到SQL系统广泛认可的基准测试。我们的评估旨在回答以下关键问题：(1) 轨迹合成和检索如何影响智能体的效率和准确率？(2) 优化后的工具设计如何提升智能体的效率和准确率？

4.1 评估设置

数据集。 Spider 2.0 [14] 是一个大规模基准测试，包含 547 个文本到 SQL 工作流问题，用于模拟企业级数据分析所面临的挑战。与传统的文本到 SQL 基准测试不同，Spider 2.0 要求解决涉及超长上下文、嵌套模式和多种 SQL 方言的任务。Spider 2.0 中的数据库通常包含数千列，并部署在 BigQuery 和 Snowflake 等生产系统中。

因此，Spider 2.0 是智能体方法的天然工作负载，智能体必须通过迭代规划并发出查询来得出最终答案。我们选择该基准测试来评估我们的方法，因为我们的框架专为在此类复杂环境中运行而设计，而这类环境需要迭代推理和工具使用。

指标。 我们使用原始基准测试作者提供的评估函数，将执行输出与标准输出匹配的比例作为执行准确率。按照原始基准测试的做法，我们采用这一指标。我们分别报告 BigQuery、Snowflake 和 SQLite 问题的准确率，以及总体结果。

我们还报告智能体轨迹的平均长度（以推理步骤数衡量），以及输入和输出令牌数。此外，我们还报告单次执行的平均端到端延迟。

实验设置。 我们使用 Claude 3-7 [1] 和 Claude 4 Sonnet [2] 模型评估我们的方法。包括 LangChain [12]，和 s molagents [8]，等多种智能体框架日益流行。尽管我们的方法与框架无关，但在评估中，我们以 smolagents 为基础实现智能体式文本到 SQL 系统。

对于所有评估的系统，我们使用 SentenceTransformers 的 MiniLM-L6-v2 模型 [23] 生成嵌入，并使用 FAISS [10] 库进行向量相似性搜索。我们在由 8× 个 A100 80GB GPU 组成的集群上，使用 vLLM [11] 运行 ArcticSQL-7B。

基线。 在 Spider 2.0 Lite 的主要实验中，我们将 AgentSM 与两个基线进行比较：SpiderAgent [14]，以及使用 smolagents [8]框架实现、配备第 2 节所述基础 SQL 执行工具的标准编码智能体（CodingAgent）。我们使用 Claude 3-7 [1] 和 Claude 4 Sonnet [2] 作为底层大语言模型，对这两个基线进行评估。其表现

方法	执行准确率 (%)				平均长度			平均延迟 (s)
	BigQuery	Snowflake	SQLite	总体	步骤	输入词元	输出词元	
SpiderAgent（claude-3-7-sonnet）	37.6	12.6	40.7	28.7	18.9	200K	4K	363.2
SpiderAgent（claude-4-sonnet）	-	-	-	27.8*	-	-	-	-
CodingAgent（claude-4-sonnet）	27.3	11.1	42.2	24.7	18.1	200K	4K	325.5
AgentSM（claude-3-7-sonnet）	40.5	33.3	42.2	38.4	16.8	299K	5K	226.4
AgentSM（claude-4-sonnet）	**52.2**	**35.0**	**51.9**	**44.8**	16.4	300K	5K	247.1
AgentSM（claude-4-sonnet，金标准表）	62.0	44.0	79.2	57.6	15.8	268K	5K	194.3

表2：Spider 2.0 Lite 数据集上各方法的执行准确率、步骤/令牌长度和延迟对比。 *此结果直接引自原始 SpiderAgent 论文。

其他最先进系统的表现可在 Spider 2.0 官方排行榜[3]上查看。

4.2 主要结果

表2列出了各方法在包含 547 个问题的完整 Spider 2.0 Lite 基准测试上的执行准确率。我们分别报告 BigQuery、Snowflake 和 SQLite 实例的执行准确率，以及总体准确率。

准确率。AgentSM 大幅优于此前的智能体基线 SpiderAgent [14]，，在每种 SQL 方言上的准确率均更高，执行准确率总体提升了 14.1%。AgentSM 还比使用相同智能体框架[8] 和模型 [2] 实现的标准编码智能体高出 20% 以上。根据我们的评估，在论文提交时，AgentSM 将以 44.8% 的总体准确率位列 Spider 2.0 Lite 排行榜第一。值得注意的是，我们在未使用 Qwen3 [3] 或 OpenAI 的 o3 [19]，等强大的推理模型的情况下取得了这一结果，而其他顶尖系统都依赖此类模型。

在所有 SQL 方言中，Snowflake 数据库上的问题对智能体而言最具挑战性。该平台包含更多数据库规模更大且具有嵌套模式的问题。仅针对该平台，我们的方法使用较旧的模型 Claude 3-7 Sonnet 时，表现优于 Claude-4 Sonnet。

效率。 为了了解 AgentSM 如何提升智能体效率，我们重新审视标准的智能体工作流（即 CodingAgent），并分析我们的方法如何重塑智能体在三个关键阶段中的推进过程：数据库探索、查询执行和答案验证。我们从 Spider 2.0 Lite 数据集中随机抽取 75 个问题，并根据推理轨迹将智能体轨迹中的每个步骤归类为这三个阶段之一
以及执行的查询。我们汇总各实例的阶段标注，以衡量每条轨迹中用于探索与执行的比例。

如图 5 所示，AgentSM 从探索阶段转向执行阶段的时间显著提前，并以更少的总体步骤完成任务。相比之下，未使用合成轨迹的标准编码智能体会反复发起探索性查询，并将大部分步骤花在分析模式上，而非编写或验证 SQL 查询。借助结构化语义记忆，AgentSM 减少了冗余的数据探索，保持了稳定的推理效率，并能从容扩展到更大的数据库和更复杂的问题。

[3]https://spider2-sql.github.io/

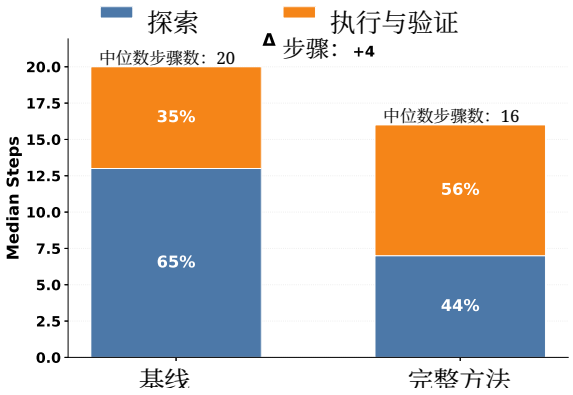


图 5：轨迹构成：中位数步骤按探索与执行和验证划分。

4.3 消融研究

4.3.1 轨迹读取与复合工具的有效性。我们针对 AgentSM 的两个关键组件——合成轨迹和复合工具——开展消融研究，以报告它们各自对准确率和效率的影响。

表3报告了 Spider 2.0 Lite 中一组 75 个样例在启用和未启用轨迹读取时的平均轨迹长度和准确率。我们观察到，启用轨迹读取后，智能体的平均轨迹长度缩短了 25%，准确率则提高了 35%。这与第 4.2 节报告的结果一致。

复合工具对智能体轨迹长度和准确率的影响同样显著。不使用复合工具时，智能体在所有样例中所需的步骤数最多，但总体执行准确率却最低。

观察到的准确率提升自然源于平均轨迹长度的缩短，这可归因于两个因素：(1) 较长的轨迹通常对应着本身更复杂的问题；(2) 随着轨迹长度增加，智能体在解决问题前更有可能达到最大步数阈值。AgentSM 通过减少用于数据探索的步骤，有效缓解了这些问题，使智能体能够将更多步骤用于推理，并在达到最大步数前迭代优化最终查询。

4.3.2 金标准表的影响。我们进一步在 Spider 2.0 Lite 上使用金标准表（由 Spider 2.0 提供）评估 AgentSM。使用金标准表和列时，我们的方法在该问题子集（见表2）上取得了**55.3%** 的执行

方法	执行准确率		步骤			平均长度			延迟	
	执行准确率 (%)	Δ_{EX}	总计	Avg	Δ_{Steps}	输入词元	输出词元	Δ_{Input}	时间 (秒)	Δ_{Time}
无轨迹读取	17.3	−34.7	1527	20.36	+4.37	360K	5.6K	+8K	348.6	+96.2
无复合工具	16.2	−35.8	1557	20.76	+4.77	398K	5.8K	+46K	523.7	+271.3
完整方法	**52.0**	–	**1167**	**15.99**	–	**352K**	**5.2K**	–	**252.4**	–

表3：在抽样的 75 个问题上，合成轨迹和组合工具对执行准确率、步骤数、令牌使用量及延迟的影响。各项变化值均相对于完整方法计算。移除任一组件都会增加令牌使用量和延迟，同时大幅降低执行准确率。

准确率。结果表明，与我们方法的基础配置相比，执行准确率有了显著提升。这凸显出，在这些复杂的数据库场景中，模式链接仍然是一个重大障碍，因为当数据和给定问题含义不明确时，智能体往往难以识别正确的表和列。

4.4 错误分析

我们对完整的 Spider 2.0 Lite 评估进行了详细的错误分析，以更好地了解 AgentSM 尚存的失败案例。在 245 个错误实例中，34 % 出现在 BigQuery 上，22% 出现在 SQLite 上，44% 出现在 Snowflake 上，后者的相对错误率最高。在 Snowflake 上，5% 的失败源于超出步骤预算，30% 源于嵌套模式导致的模式链接错误，其余则源于查询生成过程中的逻辑错误或特定方言的语法错误。

我们进一步考察结构化语义记忆如何影响智能体的成功率。当存在相关轨迹时，AgentSM 可以复用以往的推理步骤，为新查询提供有用的上下文。这对模式链接尤其有效，因为以往的轨迹往往会指出有用的表和连接模式，帮助智能体快速识别大型模式中的相关部分。相比之下，对于需要复杂数学运算或精细 CTE 推理的查询，其效果有限。这类任务通常需要更深入的推理或问题分解，而这些难以从以往轨迹中轻易推断出来。

最后，我们观察到 AgentSM 的性能因领域而异。常用数据库（如 city、weather、census）的准确率达到 60–78%，而异构数据库（如 github_repos、idc）的准确率则较低，为 14–40 %。这些结果表明，AgentSM 在模式规律一致且推理结构反复出现的领域表现出色，但面对高度专业化的领域数据源时仍面临挑战。

5 相关工作

文本到***SQL***。近期的文本到SQL研究涵盖了监督微调、提示工程以及基于强化学习的解决方案。DIN-SQL [22] 和 MAC- SQL [29] 等监督式系统采用包含结构化解码、模式链接和分解组件的流水线来生成 SQL 查询。基于提示的方法 [7, 21] 在少样本或零样本设置下利用大语言模型，通常还会结合模块化流水线，其中包括候选生成、多数投票和执行验证。近年来，采用组相对策略优化（GRPO）的强化学习技术通过设计奖励来提升模型在文本到 SQL 任务中的表现 [5]。Agentic- Data 等多智能体系统则探索结合规划、探索和验证智能体，逐步推导答案 [26]。这些技术稳步提升了模型在 Spider [35] 和 BIRD [16] 基准测试上的总体表现，但它们通常假设模式干净且对齐良好，因此难以应对企业环境中的歧义、嵌套查询和关系复杂性。

编码智能体。 近期关于编码智能体的研究探讨了大语言模型如何使用网络搜索和代码等外部工具，通过迭代推理来完成任务。ReACT [32] 和 Re- flexion [25] 等方法凸显了规划、执行和反馈阶段对复杂任务的有效性。与这些方法一致，智能体文本到SQL系统将查询生成分解为交替进行工具执行和推理的多个步骤，使模型能够检查模式、验证部分查询，并迭代优化最终答案。SpiderAgent [14]将这一趋势正式化为面向 Spider2.0 的初步解决方案。近期研究 [18] 还通过记忆引导的优化扩展了这一通用策略。尽管这些方案效果显著，但大多依赖智能体的全新状态，或维护某种任务内记忆；而我们的方案则利用任务间记忆，在不同示例之间复用数据库探索推理。

智能体记忆。 关于智能体记忆的研究大致可分为两类：面向行动的记忆，侧重于持久保存智能体状态；以及面向知识的记忆，侧重于持久保存智能体通过交互获得的知识。最基本的方法是将语言模型的上下文窗口视为类似便笺的工作记忆，供智能体记录笔记。MemGPT [20] 通过实现模仿操作系统内存管理的记忆层次结构，应对上下文窗口有限这一挑战，从而有效地衔接了前述两种方法。最近，Mem0[6]采用图表示来组织记忆，并以标准图知识库结构维护记忆：由节点（实体）和边（实体之间的关系）构成。这些记忆形式能够在为智能体提供的信息中表示的结构有限。

6 结论

本文介绍了 AgentSM，这是一种框架，能够让智能体在同一数据库中跨相关的文本到SQL任务复用推理步骤。AgentSM 利用数据探索中固有的重复性，合成、存储并检索结构化轨迹，从而大幅减少冗余探索并提升执行效率。此外，我们的解决方案配备了语义记忆，增强了可扩展性，使智能体能够处理更大的模式和更复杂的查询。因此，AgentSM 在智能体轮次、令牌使用量和延迟方面均显著提升了效率。在 Spider 2.0 Lite 基准测试中，AgentSM 的执行准确率达到 44.8%。

参考文献

[1] Anthropic。2025年。Claude 3.7 Sonnet。https://www.anthropic.com/claude。大语言模型。[2] Anthropic。2025年。Claude 4 Sonnet。https://www.anthropic.com/claude。大语言模型。[3] Jiawei Bai、Wei Zhang、Yifei Li 等。2025年。Qwen3 技术报告。arXiv 预印本 arXiv:2503.06749（2025年）。https://arxiv.org/abs/2503.06749[4] Xiaohe Bo、Zeyu Zhang、Quanyu Dai、Xueyang Feng、Lei Wang、Rui Li、Xu Chen 和 Ji-Rong Wen。2024年。基于大语言模型的反思式多智能体协作。载于《神经信息处理系统进展》，A. Globerson、L. Mackey、D. Belgrave、A. Fan、U. Paquet、J. Tomczak 和 C. Zhang（编），第 37 卷。Curran Associates, Inc.，138595–138631。https://proceedings.neurips.cc/paper_files/paper/2024/file/ fa54b0edce5eef0bb07654e8ee800cb4-Paper-Conference.pdf [5] Kai Cheng、Xiaojun Zhou 和 Weidong Chen。2024年。通过执行引导的奖励建模，利用强化学习实现文本到SQL。arXiv 预印本 arXiv:2403.12345（2024年）。[6] Prateek Chhikara、Dev Khant、Saket Aryan、Taranjeet Singh 和 Deshraj Yadav。2025年。Mem0：构建具备可扩展长期记忆的生产就绪型 AI 智能体。arXiv:2504.19413 [cs.CL] https://arxiv.org/abs/2504.19413[7] Minghang Deng、Ashwin Ramachandran、Canwen Xu、Lanxiang Hu、Zhewei Yao、Anupam Datta 和 Hao Zhang。2025年。ReFoRCE：具备自我优化、共识强制和列探索能力的文本到SQL智能体。arXiv: 2502.00675 [ ] 2502.00675 . cs.CL https://arxiv.org/abs/ 。[8] 2025 Hugging Face。。SmolAgents。https://github.com/huggingface/smolagents 用于构建语言模型智能体的轻量级框架。[9] Mourad Gridach、Jay Nanavati、Khaldoun Zine El Abidine、Lenon Mendes 和 Christina Mack。2025年。用于科学发现的智能体式人工智能：进展、挑战与未来方向综述。arXiv:2503.08979 [cs.CL] https://arxiv.org/ abs/2503.08979 [10] Jeff Johnson、Matthijs Douze 和 Hervé Jégou。2019年。借助 FAISS 实现十亿级相似性搜索。载于《IEEE大数据汇刊》。IEEE。https://doi.org/10.1109/TBDATA.2019.2921572[11] Woosuk Kwon、Zhuohan Li、Siyuan Zhuang、Ying Sheng、Lianmin Zheng、Cody Hao Yu、Joseph E. Gonzalez、Hao Zhang 和 Ion Stoica。2023年。借助 PagedAttention 实现大语言模型服务的高效内存管理。载于 ACM SIGOPS 第 29 届操作系统原理研讨会论文集。[12] LangChain。2025年。LangChain。https://github.com/langchain-ai/langchain 大语言模型应用框架。[13] Dongjun Lee、Choongwon Park、Jaehyuk Kim 和 Heesoo Park。2024年。MCS-SQL：利用多重提示和多项选择实现文本到SQL生成。arXiv:2405.07467 [cs.CL] https://arxiv.org/abs/2405.07467[14] Fangyu Lei、Jixuan Chen、Yuxiao Ye、Ruisheng Cao、Dongchan Shin、Hongjin Su、Zhaoqing Suo、Hongcheng Gao、Wenjing Hu、Pengcheng Yin 等。2024年。Spider 2.0：评估语言模型处理现实世界企业级文本到SQL工作流的能力。arXiv 预印本 arXiv:2411.07763（2024年）。[15] Haoyang Li、Shang Wu、Xiaokang Zhang、Xinmei Huang、Jing Zhang、Fuxin Jiang、Shuai Wang、Tieying Zhang、Jianjun Chen、Rui Shi、Hong Chen 和 Cuiping Li。2025年。OmniSQL：大规模合成高质量文本到SQL数据。《VLDB 会议录》18卷11期（2025年9月），4695–4709。https://doi.org/10.14778/3749646. 3749723[16] Jinyang Li、Binyuan Hui、Ge Qu、Jiaxi Yang、Binhua Li、Bowen Li、Bailin Wang、Bowen Qin、Ruiying Geng、Nan Huo 等。2024年。大语言模型已经能充当数据库接口了吗？用于大规模数据库关联文本到SQL任务的大型基准测试。《神经信息处理系统进展》36（2024年）。[17] Shu Liu、Soujanya Ponnapalli、Shreya Shankar、Sepanta Zeighami、Alan Zhu、Shubham Agarwal、Ruiqi Chen、Samion Suwito、Shuo Yuan、Ion Stoica、Matei Zaharia、Alvin Cheung、Natacha Crooks、Joseph E. Gonzalez 和 Aditya G. Parameswaran。2025年。支持我们的 AI霸主：重新设计数据系统，使其智能体优先。arXiv: 2509.00997 [ ] 2509.00997 . cs.AI [18] 2024 https://arxiv.org/abs/ 。Zihan Liu、Junhao Xu、Dongjie Guo 等。。CodeAct：基于大语言模型、通过工具使用与反思实现自主编程。arXiv 预印本 arXiv: *2402.16055* 2024 2402.16055 . ( )。https://arxiv.org/abs/ 。[19] 2025 3 OpenAI。。OpenAI o 推理模型。https://platform.openai.com/docs/ models。大语言模型。[20] Charles Packer、Sarah Wooders、Kevin Lin、Vivian Fang、Shishir G. Patil、Ion Stoica 和 Joseph E. Gonzalez。2024年。MemGPT：迈向作为操作系统的大语言模型。arXiv:2310.08560 [cs.AI] https://arxiv.org/abs/2310.08560[21] Mohammadreza Pourreza、Hailong Li、Ruoxi Sun、Yeounoh Chung、Shayan Talaei、Gaurav Tarlok Kakkar、Yu Gan、Amin Saberi、Fatma Ozcan 和 Sercan O. Arik。2024年。CHASE-SQL：文本到SQL中的多路径推理与偏好优化候选项选择。arXiv:2410.01943 [cs.LG] https://arxiv.org/abs/2410. 01943[22] Mohammadreza Pourreza 和 Davood Rafiei。2023年。DIN-SQL：通过自我纠正进行文本到SQL的分解式上下文学习。arXiv:2304.11015 [cs.CL] https://arxiv.org/abs/2304.11015

[23] Nils Reimers 和 Iryna Gurevych。2019年。Sentence-BERT：使用孪生 BERT 网络的句子嵌入。载于*2019*年自然语言处理经验方法会议论文集。计算语言学协会。https://www.sbert.net/ [24] Alireza Rezazadeh、Zichao Li、Ange Lou、Yuying Zhao、Wei Wei 和 Yujia Bao。2025年。协作记忆：大语言模型智能体中的多用户记忆共享与动态访问控制。arXiv:2505.18279 [cs.MA] https://arxiv.org/abs/2505. 18279[25] Noah Shinn、Joseph Labash 和 Ashwin Gopinath。2023年。Reflexion：采用语言强化学习的语言智能体。载于神经信息处理系统进展（*NeurIPS*）大语言模型研讨会。https://arxiv.org/abs/2303.11366。Ji Sun、[26] 2025 Guoliang Li、Peiyao Zhou、Yihui Ma、Jingzhe Xu 和 Yuan Li。年。AgenticData：面向异构数据的智能体数据分析系统。arXiv: 2508.05002 [ ] 2508.05002 。cs.DB https://arxiv.org/abs/ 。[27] Shayan Talaei、Mohammadreza Pourreza、Yu-Chen Chang、Azalia Mirhoseini 和 Amin Saberi。2024年。CHESS：用于高效 SQL 合成的上下文驾驭。arXiv: 2405.16755 [ ] 2405.16755 。cs.LG https://arxiv.org/abs/ 。[28] Khanh-Tung Tran、Dung Dao、Minh-Duong Nguyen、Quoc-Viet Pham、Barry O’Sullivan 和 Hoang D. Nguyen。2025年。多智能体协作机制：大语言模型综述。arXiv:2501.06322 [cs.AI] https://arxiv.org/abs/2501.06322[29] Bing Wang、Changyu Ren、Jian Yang、Xinnian Liang、Jiaqi Bai、LinZheng Chai、Zhao Yan、Qian-Wen Zhang、Di Yin、Xing Sun 和 Zhoujun Li。2025年。MAC-SQL：用于文本到 SQL 的多智能体协作框架。arXiv:2312.11242 [cs.CL] https://arxiv.org/abs/2312.11242。Yihan Wang、Peiyu Liu 和 Xin Yang。2025年。[30] LinkAlign：面向真实世界大规模多数据库文本到 SQL 的可扩展模式链接。arXiv: 2503.18596 [ ] 2503.18596 。cs.CL https://arxiv.org/abs/ 。[31] Junde Wu、Jiayuan Zhu、Yuyuan Liu、Min Xu 和 Yueming Jin。2025年。智能体推理：利用智能体工具增强大语言模型推理的精简框架。arXiv:2502.04644 [cs.AI] https://arxiv.org/abs/2502.04644[32] Shunyu Yao、Jeffrey Zhao、Dian Yu、Nan Du、Izhak Shafran、Tom Griffiths、Yuan Cao 和 Karthik Narasimhan。2022年。ReAct：在语言模型中协同推理与行动。载于神经信息处理系统进展（*NeurIPS*）。https://arxiv.org/abs/2210.03629[33] Shunyu Yao、Jeffrey Zhao、Dian Yu、Nan Du、Izhak Shafran、Karthik Narasimhan 和 Yuan Cao。2023年。ReAct：在语言模型中协同推理与行动。arXiv:2210.03629 [cs.CL] https://arxiv.org/abs/2210.03629[34] Zhewei Yao、Guoheng Sun、Lukasz Borchmann、Zheyu Shen、Minghang Deng、Bohan Zhai、Hao Zhang、Ang Li 和 Yuxiong He。2025年。Arctic-Text2SQL-R1：简洁奖励与文本到SQL 的强推理能力。arXiv:2505.20315 [cs.CL] https://arxiv.org/abs/2505.20315[35] Tao Yu、Rui Zhang、Kai Yang、Michihiro Yasunaga、Dongxu Wang、Zifan Li、James Ma、Irene Li、Qingning Yao、Shanelle Roman、Zilin Zhang 和 Dragomir Radev。2019年。Spider：用于复杂跨领域语义解析和文本到 SQL 任务的大规模人工标注数据集。arXiv:1809.08887 [cs.CL] https://arxiv.org/abs/1809.08887[36] Bingxi Zhao、Lin Geng Foo、Ping Hu、Christian Theobalt、Hossein Rahmani 和 Jun Liu。2025年。基于大语言模型的智能体推理框架：从方法到场景的综述。arXiv: 2508.17692 [ ] 2508.17692 。cs.AI [37] 2025 https://arxiv.org/abs/ 。Glenn Zorpette。年。大语言模型正在以指数级速度进步。https://spectrum.ieee.org/large-language-model-performance