结构化提示语言：面向大语言模型的声明式上下文管理

Wen G. Gong
wen.gong.research@gmail.com

2026年2月22日

摘要

我们提出**SPL**（结构化提示语言），这是一种受 SQL 启发的声明式语言，将大语言模型视为生成式知识库，并将其上下文窗口视为受限资源。SPL 提供显式的 `WITH BUDGET/LIMIT` 词元管理、自动查询优化器、类似于 SQL `EXPLAIN ANALYZE`的 `EXPLAIN` 透明性，以及在单一声明式框架中原生集成检索增强生成（RAG）和持久化记忆。**SPL-flow** 将 SPL 扩展为弹性智能体流水线，采用三层提供商回退策略（Ollama → OpenRouter → 自愈式重试），且对 `.spl` 脚本完全透明。

五项扩展展示了这一范式的广度：(1) *Text2SPL*（多语言自然语言→SPL 翻译）；(2) 模型混合（MoM）路由，在运行时将每个 `PROMPT`分派给领域专家模型；(3) 逻辑分块，一种针对超出单个上下文窗口的文档的智能策略——通过 SPL 现有的 CTE 语法即可自然表达，无需新增结构；它将大型查询分解为 MapReduce 流水线，使注意力计算成本从 $O(N^2)$降低至 $O(N^2/k)$，并可在云端（并行）或本地硬件（顺序执行）上以相同方式运行；(4) *SPL-flow*，具备弹性三层提供商回退机制的声明式智能体编排层；(5) 基准测试，用于并行多模型比较，并自动持久化胜出模型。

我们提供正式的 EBNF 文法、两个可通过 pip 安装的 Python 软件包（`spl-llm`, `spl-flow`），并与 Prompty、DSPy 和 LMQL 进行比较。SPL 平均可减少 65% 的提示词冗余，在执行前显示跨模型档位的 $68\times$ 成本差异，并且无需修改，同一 `.spl` 脚本既可在 OpenRouter 上以 0.002 美元运行，也可在本地 Ollama 实例上以零边际成本运行。

Keyw词语： 大语言模型、提示工程、声明式语言、上下文
上下文管理、词元优化、检索增强生成

1 引言

大语言模型（LLMs）的出现 [Brown等人，2020年，Vaswani等人， 2017]作为通用推理引擎，在软件工程领域带来了前所未有的挑战：如何系统地管理上下文窗口——这个固定大小的输入缓冲区决定了大语言模型生成响应时能够考虑哪些信息。上下文窗口的大小从早期模型的 4K 词元到近期架构的 100 多万词元不等，但挑战的本质始终相同：上下文是一种受限资源，必须在相互竞争的需求之间得到高效分配。

当前大语言模型提示工程的状况，与 SQL 之前的数据库编程有着惊人的相似之处。在前 SQL 时代（20世纪60年代），程序员编写命令式的过程式代码来遍历数据结构——每家数据库供应商都有自己的访问方法，而且没有标准方式来表达需要什么数据，而不是如何 检索数据。Codd 的关系模型 [Codd, 1970] 以及随后 SQL 的开发 [Chamberlin and Boyce,

1974] 通过引入一种声明式语言，将逻辑查询与物理执行分离，从而解决了这一混乱局面。

如今，提示工程师面临着类似的问题：

- 手动计算词元：开发人员手动计算提示词是否能放入上下文窗口，往往只能反复试错。
- 临时截断：当上下文超出限制时，工程师会采用粗略的截断策略，缺乏系统性的优化。
- 缺乏可见性：目前没有标准方法来检查词元在提示词不同部分中的分配情况。
- 供应商锁定：提示词代码通常与特定的大语言模型 API 绑定，导致迁移成本高昂。
- 缺乏可组合性：与 SQL 的公用表表达式（CTE）和视图不同，提示词是单体结构，难以模块化复用。

我们提出**SPL（结构化提示语言）**[1]，，这是一种声明式查询语言，将 SQL 成功的原则——声明式规范、自动优化、可组合性、可移植性和透明性——应用于大语言模型上下文管理领域。

与 DSPy 等提示内容优化框架的一个关键区别在于：SPL 管理词元预算——即限制大语言模型可接收多少上下文的容器——而非提示词内容本身。包含哪些信息由开发者决定；如何在预算范围内容纳这些信息，以及超出预算时如何压缩，则由 SPL 负责。

SPL 的核心概念创新在于，将大语言模型视为生成式知识库，而不只是将文本映射为文本的函数：它不仅存储知识，还会主动合成知识。表1概述了 SQL 与 SPL 的对应关系；第2节将详细阐述这一框架及其影响。

表1：SQL 与 SPL 的对应关系：从静态知识库到生成式知识库。

维度	**SQL（1970年）**	**SPL（2026）**
知识库	静态（行/表）	生成式（训练权重）
查询结果	检索数据	合成内容
受限资源	磁盘I/O、内存	令牌预算（上下文窗口）
优化目标	尽量减少输入/输出	尽量减少词元，最大限度提高相关性
`SELECT`	检索现有数据	从来源收集上下文
`GENERATE`	N/A	根据上下文创建新内容
`EXPLAIN`	显示查询执行计划	显示令牌分配计划
CTE / 视图	编写复杂查询	编写复杂提示词
事务	ACID 保证	内存持久化 + 缓存
提供商可移植性	Oracle、Postgres、MySQL	OpenRouter、Claude、GPT、Llama

1.1 贡献

本文作出以下贡献：

- 语言设计：我们给出了 SPL 的完整语法和形式文法（扩展巴科斯-瑙尔范式）。SPL 是一种用于大语言模型上下文管理的声明式语言，支持显式令牌预算、可组合的 CTE、用户定义函数，以及集成的检索增强生成（RAG [Lewis 等人， 2020]）和记忆。

[1]另一个与本文无关、但同样使用缩写“SPL”的项目采用基于 XML 的模板标记来编写智能体指令[Eshraw， 2025]；它是一种模板语言，而非查询语言。本文的 VS Code 扩展位于`https://github.com/digital-duck/SPL-LLM`。

- 上下文即资源模型：我们将大语言模型上下文窗口形式化为一种受限资源，并提出一种令牌预算分配算法，其原理类似于关系数据库中的查询优化。

- EXPLAIN 机制：我们引入透明的执行计划，让开发者在执行前检查大语言模型查询并推理令牌分配。
- 参考实现：我们提供一个完整、可通过 pip 安装的 Python 软件包（`spl-llm`），其中包含手写的递归下降解析器、优化器和与提供商无关的执行引擎。

- 对比分析：我们从关键维度系统地比较 SPL 与现有的提示词管理方法（Prompty、DSPy、LMQL）。
- SPL 生态系统：我们介绍五项扩展，展示 SPL 是一个应用平台，而非仅仅是研究原型：(1) Text2SPL（多语言自然语言接口）、(2) 模型混合（MoM）路由、(3) 逻辑分块（针对长上下文的声明式映射-归约，由 SPL 原生 CTE 语法提供支持）、(4) SPL-flow（具备韧性的智能体编排），以及 (5) `BENCHMARK`，具备自动优胜者选择功能。

2 大语言模型作为生成式知识库

SPL 的概念基础是将大语言模型视为生成式知识库。这种视角并非纯粹的理论观点：每天构建人工智能应用的从业者发现，“大语言模型即数据库”这一心智模型——将其视为一个可供查询的系统，其响应取决于放入其上下文窗口的内容，而其隐含的“表”则是训练权重——是理解大语言模型实际运作方式最有效的框架。SPL 将从业者的这种直觉转化为正式且可执行的表达。

2.1 从检索到合成

传统信息系统遵循检索并呈现范式：用户提出查询，系统搜索其索引，并返回匹配的文档。随后，用户必须根据检索到的内容综合理解。

大语言模型打破了这一范式。收到查询后，大语言模型并非只是检索存储的信息，而是通过结合以下内容综合出全新的响应：

- 参数化知识：训练过程中编码在模型权重中的信息
- 上下文知识：当前提示词中提供的信息（即上下文窗口）
- 推理：涌现的多步推理能力 SPL 的双子句架构（`SELECT` + `GENERATE`）直接体现了这种区分：

- `SELECT` 填充上下文知识（上文第 2 项），并可选择依据推理框架（第 3 项）进行结构化

- `GENERATE` 触发完整的综合过程，调动全部三个来源：模型权重中已编码的参数化知识（第 1 项）、由 `SELECT` 汇集的上下文知识（第 2 项），以及涌现推理（第 3 项）

这就是为什么 SPL 不仅仅是“提示模板语言”——它是生成式知识库的查询语言，其中“数据库”（大语言模型权重）能够生成其“表”（训练数据）中从未存在过的输出。

2.2 与 SQL 的二元性

在 SQL 中，`SELECT` 语句既指定要检索的内容也 生成结果。在 SPL 中，我们将这两项职责分开：

- `SELECT` 规定要收集哪些上下文（检索阶段）
- `GENERATE` 指定要根据该上下文创建什么（合成阶段）

这种分离使优化器能够将上下文收集与生成视为具有不同资源特征的独立阶段，从而实现更好的预算分配。

SQL 从业者须知：在 SQL 中， `SELECT` 是既收集又生成最终输出的语句。在 SPL 中，这一输出角色属于 `GENERATE`——触发综合并交付响应的子句。SPL 的 `SELECT` 类似于 SQL 子查询或 `WITH`子句：它为生成步骤组装上下文，但自身不产生输出。

2.3 对软件工程的启示

将大语言模型视为生成式知识库，对软件工程实践有若干启示：

- 提示工程变为查询工程：技能组合从“精心编写欺骗模型的文本”转向“声明需要何种上下文以及何种输出”。
- 可观测性成为可能：正如 SQL `EXPLAIN` 能够实现数据库性能调优，SPL `EXPLAIN` 能够实现提示词性能调优。
- 测试变得结构化：SPL 查询可在任何大语言模型调用之前进行静态验证（解析、分析），并估算其资源使用。
- 版本控制因此变得切实可行：`.spl` 文件便于人类阅读、比较差异和审查——而嵌入在应用程序代码中的提示字符串则不然。

3 背景与动机

3.1 提示工程问题

随着大语言模型在软件系统中日益成为核心，提示工程也逐渐发展为一门关键学科。开发人员必须构建输入序列，以有效利用模型的能力，同时满足上下文窗口大小这一硬性约束，以及成本和延迟方面的软性约束。

当前的提示工程实践几乎完全是命令式的：开发人员编写 Python（或 JavaScript 等）代码来拼接字符串、统计词元、运用启发式方法进行截断，并发起 API 调用 [Wei 等人， 2022]。这种命令式方法存在若干有据可查的问题：

- 紧密耦合：提示词构造逻辑与应用逻辑交织在一起，导致提示词难以测试、纳入版本控制和共享。
- 缺乏预算纪律：如果没有明确的预算分配，开发人员往往要等到运行时（通常是在生产环境中）才会发现词元超支。
- 优化盲点：命令式代码模糊了令牌分配的结构，使人无法判断预算是否得到了最优使用。
- 重复造轮子：每个项目都从头重新实现常见模式（RAG 上下文注入、对话历史管理、响应缓存）。

这些模式在 Data-Copilot [Gong,2024], （一款由 RAG 驱动的对话式分析应用）的开发过程中被直接遇到，并促使我们设计 SPL。

3.2 SQL 成功的经验

五十多年来，SQL 一直主导着数据库交互，因为它体现了一系列被证明具有普遍价值的原则：

- 声明式：指定你想要什么，而不是如何实现。优化器负责处理执行策略。
- 可优化：查询规划器可以重新排列连接、选择索引并实现并行处理——这在命令式导航中是不可能的。

- 可组合：CTE、视图、子查询和函数支持以模块化方式构建复杂查询。
- 可移植：同一条 SQL 可以在 Oracle、PostgreSQL、MySQL、SQLite 以及数百种其他引擎上运行（方言略有差异）。
- 透明： `EXPLAIN` 会揭示执行计划，使开发者能够据此进行优化。

SPL 将这些原则逐一应用于大语言模型上下文管理，具体内容将在以下各节中详述。

3.3 将 TokenBudget 视为受限资源

我们提出一种形式化模型，将大语言模型上下文窗口视为受约束资源分配问题。给定一个包含 B 个词元的上下文窗口，SPL 查询必须在以下各项之间分配此预算：n 上下文来源 s_1、…、s_n、生成任务 g以及安全缓冲区 β：

$$\sum_{i=1}^{n} \text{tokens}(s_i) + \text{tokens}(g) + \beta \leq B \tag{1}$$

每个来源 s_i 都有：

- 一个估算大小 $\hat{t}_i$ （优化前的词元数）
- 可选的限制 L_i （ `LIMIT` 子句）
- 一个优先级 p_i （根据来源类型得出：记忆 $>$ 缓存 $>$ 检索增强生成 $>$ 上下文）
- 一个压缩比 $c_i \in (0, 1]$ （超出预算时应用）

优化器的任务是找到分配，$d_i \leq \min(\hat{t}_i, \ldots)$ 使其满足方程 1，同时最大化信息效用。这类似于关系数据库中的查询优化器：在生成正确结果的前提下，最小化 I/O 成本。

4 SPL 语言

4.1 设计理念

SPL 遵循三项设计原则：

- 熟悉 SQL：借助数百万 SQL 开发者的肌肉记忆。任何会编写 `SELECT ... FROM ... WHERE` 的人都能快速学会 SPL。
- 预算优先：每条 SPL 查询都会明确声明其令牌预算。这促使开发者预先考虑上下文分配，避免运行时出现意外。
- 来源无关：上下文来源（用户输入、检索增强生成结果、记忆）通过统一接口访问，因此无论数据来自何处，查询都可以组合。

4.2 核心语法

基本的 SPL 语句是 `PROMPT` 查询，其结构如下：

```
PROMPT answer_question
WITH BUDGET 8000 tokens
USING MODEL claude-sonnet-4-5

SELECT
    system_role("You are a knowledgeable assistant"),
    context.question AS question LIMIT 200 tokens,
    rag.query("relevant docs", top_k=5) AS docs LIMIT 3000 tokens,
    memory.get("history") AS history LIMIT 500 tokens

WHERE
    docs.relevance > 0.7
```

```
ORDER BY
    docs.relevance DESC

GENERATE
   detailed_answer(question, docs, history)
```

清单 1：SPL 查询的基本结构

关键子句包括：
`PROMPT <name>` 为查询命名，以便重复使用和引用。
`WITH BUDGET <n> tokens` 声明整个操作的总词元预算（输入 +输出）。

`USING MODEL <model>` 指定目标大语言模型，从而支持针对特定模型的词元计数和成本估算。

`SELECT` 从异构来源收集上下文，每个来源都可设置可选的 `LIMIT` 上限。
`WHERE` 根据条件筛选上下文条目（例如，针对 RAG 结果设置相关性阈值）。
`ORDER BY` 对上下文条目排序，控制它们在组装后的提示中的显示顺序。
`GENERATE` 指定生成任务及其输出参数。
`STORE RESULT IN memory.<key>` 将结果持久化到记忆存储，以供后续查询。

4.3 内置上下文源

SPL 提供四种内置上下文来源类型：

表 2：SPL 内置上下文来源。

来源	语法	描述
系统角色	`system_role("...")`	设置系统/指令提示
上下文	`context.<field>`	引用外部输入参数-ters
RAG	`rag.query("...", top_k=N)`	从向量存储进行语义检索
记忆	`memory.get("key")`	从持久化存储中进行键值检索存储

4.4 通用表表达式（CTE）

与 SQL 类似，SPL 通过 `WITH` 子句支持 CTE，从而能够用更简单的构建块组合出复杂提示词：

```
PROMPT analyze_and_recommend WITH BUDGET 12000 tokens
```

```
SELECT
    system_role("You are a product recommendation expert"),
    compressed_profile AS profile,
    relevant_docs AS docs,
    memory.get("purchase_history") AS history LIMIT 1000 tokens

GENERATE
    product_recommendations(profile, docs, history)


STORE RESULT IN memory.last_recommendations;
```

列表 2：用于可组合提示构建的 CTE

CTE 有两个用途：(1) 对上下文来源进行令牌受限预处理；(2) 在查询中创建可复用命名组件。

4.5 用户定义函数

SPL 支持 `CREATE FUNCTION` 可复用提示组件：

```
CREATE FUNCTION compress_bio(bio, max_tokens)
RETURNS text
AS $$
    SELECT bio LIMIT max_tokens tokens
$$;
```

列表 3：用户定义函数

函数封装了常见的上下文处理模式（压缩、格式化、提取），可从任意 `SELECT` 或 `GENERATE` 子句中调用。

4.6 EXPLAIN 机制

`EXPLAIN` 语句会生成详细的令牌分配计划，而不会执行查询：

```
EXPLAIN PROMPT answer_question;
```

列表 4：用于执行计划透明度的 EXPLAIN

这会生成如下输出：

```
Execution Plan for: answer_question
============================================================
Budget: 8,000 tokens | Model: claude-sonnet-4-5

Token Allocation:
+-- __system_role__              20 tokens  (  0.2%)
+-- history                     500 tokens  (  6.2%)  [from memory]
+-- docs                      3,000 tokens  ( 37.5%)  [cache MISS]
+-- question                    200 tokens  (  2.5%)
+-- Output Budget             2,000 tokens  ( 25.0%)
\-- Buffer                    2,280 tokens  ( 28.5%)
                              ----------
Total                         5,720 / 8,000 tokens (71.5%)

Estimated Cost: $0.0412
```

EXPLAIN 输出提供：(1) 所有上下文来源的词元分配树状视图，(2) 各来源消耗的预算百分比，(3) 各来源的缓存状态，(4) 适用时显示的压缩注释，以及 (5) 根据目标模型定价估算的成本。

这一机制与 SQL 的 EXPLAIN / EXPLAIN ANALYZE 直接类似，后者会显示查询执行计划，包括连接策略、索引使用情况和预估行数。对于大语言模型查询而言，“执行计划”就是令牌分配策略。

4.7 形式文法

我们为 SPL 提供了完整的 EBNF 文法。该文法是上下文无关的，可由递归下降解析器以 LL(1) 方式解析。完整文法（包括 WHERE、ORDER BY、 CREATE FUNCTION、 EXECUTE、表达式和终结符的所有产生式）见附录 F，也可在项目代码库中通过 docs/grammar.ebnf https://github.com/digital-duck/SPL 获取。核心 PROMPT 语句结构如下：

清单 5：SPL 语法——核心产生式（完整 EBNF 见附录 F）

```
program     = statement (";" statement)* ";"? EOF ;
statement   = prompt_stmt | create_func_stmt | explain_stmt | execute_stmt
   ;

prompt_stmt = "PROMPT" IDENT
              ("WITH" "BUDGET" INTEGER "TOKENS")?
              ("USING" "MODEL" (IDENT | STRING))?
              ("CACHE" "FOR" INTEGER IDENT)?
              cte_block?
              "SELECT" select_item ("," select_item)*
              ("WHERE" condition (("AND" | "OR") condition)*)?
              ("ORDER" "BY" order_item ("," order_item)*)?
              "GENERATE" IDENT "(" arg_list? ")"
              ("WITH" generate_opt ("," generate_opt)*)?
              ("STORE" "RESULT" "IN" "memory" "." IDENT)? ;
```

5 上下文作为受限资源

5.1 信息论表述

我们将 SPL 查询建模为一个资源分配问题。设 Q 为一个 SPL 查询，其总预算为 B，输出预算为 O，上下文来源为 $S = \{s_1, \ldots, s_n\}$。可用于上下文的输入预算为：

$$B_{\text{input}} = B - O - \beta \tag{2}$$

其中 β 为安全缓冲区（通常为 B的 5–10%）。

每个上下文来源 s_i 都有：

- 原始词元数： t_i =个词元(s_i)
- 用户指定上限： L_i （来自 LIMIT 子句；若未指定则为 ∞ ）
- 有效分配： $a_i = \min(t_i, L_i)$

预算约束为：

$$\sum_{i=1}^{n} a_i \leq B_{\text{input}} \tag{3}$$

当 $\sum a_i > B_{\text{input}}$ （超出预算）时，优化器会进行压缩。作为一种闭式近似，均匀比例缩放可得：

$$a_i' = a_i \cdot \frac{B_{\text{input}}}{\sum_{j=1}^{n} a_j} \quad \text{(proportional scaling, simplified)} \tag{4}$$

实现的过程（算法 1）通过最大优先贪心策略对其进行细化：按照分配大小降序排列来源，并将每个来源压缩至多达其分配量的 50%，直至消除超额部分。这会优先压缩最大的来源，从而完整保留较小的固定大小项（例如系统角色、简短问题）。请注意，压缩完全依据大小进行；优先级排序（记忆 > 缓存 > 检索增强生成 > 上下文）仅决定执行顺序，不影响分配保护。此设计有意为之：若将压缩与优先级挂钩，高优先级来源便可能占用其全部分配量，牺牲低优先级来源，甚至导致某些来源类型的词元数量为零。按大小压缩则可确保所有 来源都按比例为上下文做出贡献——较大的记忆条目压缩幅度大于简短的系统角色，但两者都会保留。基于优先级的压缩保护是未来工作的一个自然延伸（第 10 节）。

5.2 成本模型

SPL 的成本模型与关系代数的成本模型类似。在 SQL 中，优化器会估算 I/O 成本（读取的磁盘页、索引查找）。在 SPL 中，优化器会估算：

$$\text{Cost}(\mathcal{Q}) = \text{cost\_per\_input\_token}(m) \cdot \sum a_i + \text{cost\_per\_output\_token}(m) \cdot O \tag{5}$$

其中 m 是目标模型。这使 `EXPLAIN` 能够在执行前显示预估的美元成本，让开发人员能够就模型选择和预算分配做出明智的决定。

5.3 执行顺序优化

SPL 优化器还会按照以下优先级层级，确定解析上下文来源的顺序：

- 缓存命中：先前计算出的结果（零延迟、零成本）
- 记忆：SQLite 键值查找（亚毫秒级、零成本）
- 检索增强生成：FAISS/ChromaDB 向量搜索（毫秒级、成本极低）
- 上下文：外部输入参数（延迟可变）
- 大语言模型调用：生成步骤（延迟和成本最高）

这种排序通过先执行成本最低的操作来最大限度地降低延迟，类似于 SQL 优化器优先选择索引扫描而非全表扫描的做法。优先级仅影响执行顺序——它不会保护来源免受压缩影响，压缩完全由分配大小决定（参见第 5 节）。

6 SPL 引擎

6.1 架构

SPL 引擎遵循经典的编译器流水线架构，其结构与 SQL 查询引擎相似。该流水线包含五个阶段：**词法分析器**、**解析器**、**语义分析器**、**优化器**和**执行器**。

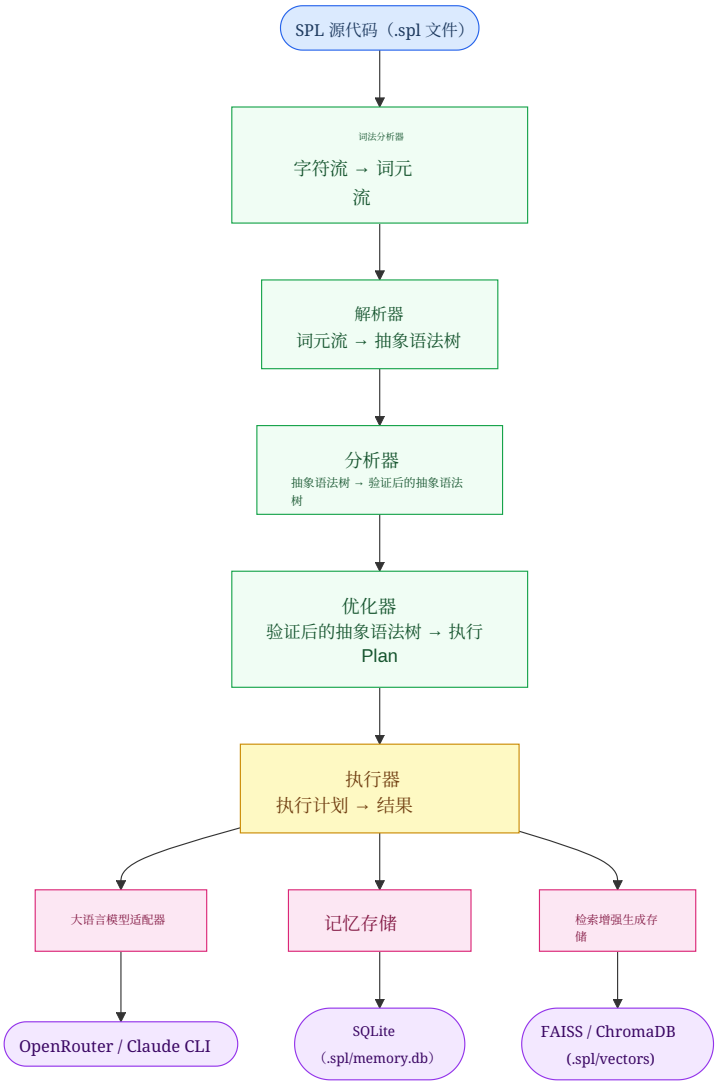

图 1：SPL 引擎流水线架构。编译器阶段（蓝色输入、绿色流水线）将 SPL 源代码转换为执行计划，供执行器（黄色）使用；执行器再将任务分发至三个后端子系统（粉色）：大语言模型适配器（OpenRouter / Claude CLI）、SQLite 记忆存储，以及同时支持 FAISS 和 ChromaDB 的向量检索增强生成存储（紫色）。

6.1.1 词法分析器

词法分析器逐字符扫描，生成带类型的词元流。它处理：SQL 风格关键字（不区分大小写）、字符串字面量（单引号和双引号）、数值字面量（整数和浮点数）、用于限定引用的点表示法（`context.field`）、用于函数体的 `$$` 分隔符，以及 `--` 单行注释。关键字集合包含 30+ 保留字（表 3）。

表 3：SPL 关键字类别。

类别	关键字
结构	PROMPT, SELECT, GENERATE, WITH, AS, FROM
预算	BUDGET, TOKENS, LIMIT, OUTPUT
模型	USING, MODEL, TEMPERATURE, FORMAT
过滤	WHERE, AND, OR, NOT, IN, ORDER, BY, ASC, DESC
函数	CREATE, FUNCTION, RETURNS
控制	EXPLAIN, EXECUTE, PARAMS, STORE, RESULT, CACHE, FOR
内置项	system_role, context, rag, memory

6.1.2 解析器

该解析器是手写的递归下降解析器——这是有意为之的架构选择，而非使用解析器生成器（ANTLR、Lark），原因有三：

- 零依赖：该解析器没有外部依赖项，可缩小攻击面并减少安装阻力。
- 完全控制：错误消息可以引用 SPL 特有的概念（“SELECT 之后应为 GENERATE 子句”），而不是通用的解析错误。
- 可形式化：文法产生式与解析器方法之间的一一对应，使该实现成为语法 LL(1) 性质的直接证明，适合纳入形式语言规范。

解析器为每种语法结构生成带有数据类节点的带类型抽象语法树（表 4）。

表 4：核心抽象语法树节点类型。

节点类型	描述
`PromptStatement`	带有预算、模型、SELECT、GENERATE 的顶层查询子句
`SelectItem`	带有可选别名和令牌的单一上下文来源限制
`CTEClause`	公用表表达式（命名子查询）
`GenerateClause`	带参数的生成规范
`CreateFunctionStatement`	用户定义函数定义
`ExplainStatement`	EXPLAIN 查询
`ExecuteStatement`	参数化查询执行

6.1.3 语义分析器

分析器会对抽象语法树执行验证遍次：

- 名称解析：验证所有引用的别名（在 `GENERATE` 参数、 `WHERE` 条件中）是否对应已定义的 `SELECT` 项目或 CTE。

- 预算算术：当 `LIMIT` 子句的总和超过总预算时发出警告。
- 类型检查：验证上下文来源是否使用正确语法（例如，`rag.query` 需要一个字符串参数）。
- 重复检测：报告重复的提示名称、别名或 CTE 定义。

6.1.4 优化器

优化器将验证后的抽象语法树转换为一个 `ExecutionPlan`——由一系列有序步骤组成，并为其分配令牌预算。优化算法（算法 1）按以下步骤执行：

算法 1 SPL 令牌预算优化器

要求： B：总预算，O：输出预算，S：SELECT 项集合
确保： 带有词元分配的执行计划

$B_{\text{输入}} \leftarrow B - O - \beta$
对于每个 $s_i \in S$ **执行**
  $\hat{t}_i \leftarrow$估算_词元(s_i)
  $a_i \leftarrow$最小值$(\hat{t}_i, L_i)$ ▷ 应用 LIMIT 上限
结束循环
如果 $\sum a_i > B_{\text{input}}$ **then**
  按 S 依据 a_i 降序排序 ▷ 优先压缩最大的项
  超额 $\leftarrow \sum a_i - B_{\text{in put}}$
  对每个 s_i 按排序顺序**执行**
    减少 $\leftarrow$ 最小$(a_i \times 0.5$,超额$)$
    $a_i \leftarrow a_i -$减少
    超额 $\leftarrow$ 超额 $-$ 减少
    如果 超出 ≤ 0 **，则**
      跳出
    结束如果
  结束循环
结束如果
按优先级排序：记忆 $>$ 缓存 $>$ 检索增强生成 $>$ 上下文
返回 ExecutionPlan(S, 分配, 成本估算)

6.1.5 执行器

执行器会逐步处理执行计划：

- 解析各个上下文来源（从记忆中加载、执行检索增强生成查询、读取参数）
- 通过截断应用令牌限制
- 检查提示缓存（基于哈希，并设有 TTL）
- 组装最终提示字符串，其中包含系统角色、上下文部分和生成指令
- 调用大语言模型适配器
- 可选择通过 `STORE RESULT IN memory.<key>` 存储结果
- 返回包含内容、令牌使用量、耗时和成本的结构化结果

6.2 大语言模型适配器接口

SPL 通过抽象适配器接口实现供应商无关性：

清单 6：大语言模型适配器接口（Python）

```
class LLMAdapter(ABC):
    @abstractmethod
```

```
    async def generate(self, prompt: str, model: str,
```

参考实现包含四种适配器，涵盖本地和云端部署模式：

- Ollama 适配器（本地）：封装 Ollama REST API（`localhost:11434`），可完全在本地运行开源权重模型（Llama、Mistral、Phi、通义千问等）。除硬件成本外，该适配器无需额外费用，也不需要互联网连接，因此适用于对隐私敏感的场景和气隙隔离部署。
- OpenRouter 适配器（云端聚合器）：调用 OpenRouter.ai [OpenRouter 2024]API，通过统一端点访问 100+ 模型（Claude、GPT、Llama、Mistral、Gemini 等）。这在基础设施层面实现了 SPL 的供应商无关性愿景。
- 直连云适配器（原生云端）：为 Anthropic Claude、OpenAI GPT 和谷歌 Gemini API 提供专用适配器，直接调用各提供商的原生端点，从而能够使用提供商特有的功能（扩展思考、结构化输出、信息溯源），而无需通过聚合器路由。
- Claude CLI 适配器（开发）：通过子进程封装 Claude Code CLI，利用订阅制计费，实现开发时的零边际成本（Claude Pro 订阅用户无需向开发者支付额外费用）。此适配器表明，SPL 可以接入任何大语言模型访问方式，而不仅限于 HTTP API。

6.3 存储层

SPL 集成了两种存储系统，二者均基于文件，便于移植：

- SQLite 记忆存储（`.spl/memory.db`）：为`memory.get()` 和 `STORE RESULT IN memory.<key>` 提供持久化键值存储。它还支持存储带 TTL 的提示缓存条目。选择 SQLite 是因为它无需配置即可部署、可靠性久经考验，并且包含在 Python 标准库中。
- 向量存储（`.spl/vectors.*`）：为 `rag.query()`提供语义相似度搜索。该引擎支持两种后端：**FAISS** [Johnson 等人， 2019]适用于基于文件、无需配置的部署；以及 **ChromaDB** [Chroma Core， 2023] 适用于更丰富的元数据筛选和托管服务器模式。两者提供相同的 Python 接口，因此 SPL 查询无需关心具体选择哪一种。文档元数据存储在配套的 SQLite 数据库中（`.spl/vectors_meta.db`）。

两个存储都位于 `.spl/` 目录中，其位置取决于部署模式：从克隆的源代码仓库运行时，目录会创建在 `$REPO_ROOT/.spl/` （与代码位于同一位置，类似于 `.git/`）；通过 `pip install spl-llm` 安装时，默认使用 `$HOME/.spl/` （一个供多个项目共享的用户级存储）。该目录可以加入 gitignore，且内容自成一体，从而支持可复现的部署。

7 相关工作

我们将 SPL 与现有的大语言模型交互管理方法进行比较。表5从功能层面进行了比较；下文将逐一讨论这些方法。

表5：大语言模型提示管理方法的功能对比。

ID	功能	SPL	LMQL	Prompty	DSPy	LangChain
F1	SQL知识可复用	✓	部分[d]	–	–	–
F2	类 SQL 公用表表达式与函数	✓	–	–	–	–
F3	形式文法（扩展巴科斯-瑙尔范式）	✓	✓	–	–	–
F4	零依赖解析器	✓	–	N/A	N/A	N/A
F5	声明式查询语言	✓	✓	–	–	–
F6	显式令牌预算	✓	部分[a]	–	–	–
F7	EXPLAIN 执行计划	✓	–	–	–	–
F8	自动压缩	✓	–	–	–	–
F9	内置检索增强生成	✓	–	–	–	通过插件
F10	持久化记忆	✓	–	–	–	通过插件
F11	与提供商无关	✓	✓	部分[b]	✓	✓
F12	提示优化	–[c]	–	–	✓	–
	扩展与应用					
F13	多语言自然语言→查询界面	✓	–	–	–	–
F14	混合模型路由	✓	–	–	–	通过代码
F15	声明式 Map-Reduce 分块	✓	–	–	–	通过代码
F16	弹性智能体流水线	✓	–	–	部分	通过 LangGraph[f]
F17	并行模型基准测试	✓	–	–	部分[e]	–

[a] LMQL 支持对每个变量设置长度约束，但不支持全局预算分配。[b] Prompty 主要面向 Azure/OpenAI；也有社区适配器支持其他平台。[c] SPL 优化令牌分配 （资源管理）；DSPy 优化提示内容（少样本示例）。两者关注点互为补充，而非相互竞争。[d] LMQL 借用 SQL 关键字（`FROM`、 `WHERE`）来约束词元级生成，但其思维模型与关系查询思维有很大不同。SPL 则有意在端到端层面采用 SQL 模型，使具备 SQL 经验的从业者无需学习新的编程范式，即可采用生成式人工智能工作流。[e] DSPy 支持针对提示内容的评估与优化循环；SPL `BENCHMARK`则并行比较模型输出、令牌成本和延迟，并自动保存获胜的模型标识符，以供后续执行使用。[f] LangGraph 要求在应用程序代码中显式构建图，并定义节点、边和状态管理。SPL-flow 完全采用声明式方式：提供方回退（Ollama →OpenRouter → 自愈式重试）会自动透明地应用于 `.spl` 脚本。DSPy 支持模块链式调用，但不提供提供方级别的容错能力。

7.1 LMQL

LMQL [Beurer-Kellner 等人， 2023]（语言模型查询语言）是一种受 SQL 启发、用于约束大语言模型生成的查询语言。它支持变量级约束（例如，`where len(ANSWER) < 100`）、logit 操控和多模型调度。

与 SPL 的主要区别：

- 侧重约束与侧重资源：LMQL 专注于约束大语言模型生成的输出（格式、长度、有效令牌）。SPL 专注于管理输入——如何收

集、压缩文本，并在令牌预算内分配。这两者相辅相成：LMQL 约束输出内容，SPL 管理输入内容。

- 没有全局预算：LMQL 支持为每个变量设置词元限制，但没有全局词元预算和跨组件分配的概念。
- 没有上下文管理：LMQL 不提供内置检索增强生成、记忆或上下文压缩。
- 侧重研究：LMQL 起源于一个研究项目，在生产环境中的应用有限。

7.2 Prompty

Prompty [微软， 2024] 是微软通过基于 YAML 的文件格式（`.prompty`）实现提示工程标准化的方法。Prompty 文件声明模型参数、输入/输出模式，以及带有变量插值的模板文本。它与微软 Semantic Kernel 和 Azure AI Foundry 生态系统集成。

与 SPL 的主要区别：

- 模板与查询：Prompty 是一种模板格式——它定义带有变量插槽的静态提示结构。SPL 是一种查询语言——它将动态的上下文收集、优化与生成表达为单一声明式查询。
- 无令牌预算：Prompty 依赖底层模型的 `max_tokens` 参数，且未提供在各提示组件之间分配词元的机制。
- 无执行计划：Prompty 没有与 `EXPLAIN` 等效的机制来检查词元分配。
- 无集成的检索增强生成/记忆：上下文来源（检索增强生成、记忆）必须在外部管理，并以变量形式注入。
- 生态系统耦合：Prompty 主要面向 Azure/Semantic Kernel 生态系统设计，而 SPL 在设计上与提供商无关。

Prompty 与 SPL 在不同的抽象层级上运作：Prompty 标准化提示文件的格式，而 SPL 将上下文管理的 语义 标准化为一种查询语言。

7.3 DSPy

DSPy [Khattab 等人， 2024] 是一个用于“编程而非提示基础模型”的框架。它引入了一种嵌入式 Python 范式：将大语言模型交互定义为带有输入/输出字段的类型化 `Signature` 类，使用模块（`ChainOfThought`、`ReAct` 等）进行组合，并通过自动提示调优进行优化。

与 SPL 的主要区别：

- 嵌入式 *Python* 与独立语言：DSPy 是一个 Python 库；SPL 是一种拥有自身语法和解析器的独立查询语言。这使得 Python 开发者更容易采用 DSPy，但也使其更难进行静态分析、独立版本管理以及移植到其他生态系统。
- 提示优化与资源优化：DSPy 的关键创新是自动提示优化——通过搜索找出最佳的少样本示例、指令和模块配置。SPL 的关键创新是资源优化——管理异构上下文来源之间的令牌预 算。这些关注点相辅相成：经过 DSPy 优化的提示词仍需符合令牌预算，而 SPL 查询也可以在其 `SELECT` 子句中受益于 DSPy 式的内容优化。
- 没有令牌可见性：DSPy 不会向开发者公开令牌分配。没有与 `EXPLAIN` 或 `WITH BUDGET` 等效的机制。
- 无集成存储：DSPy 不提供内置记忆或检索增强生成；这些必须从外部集成。

我们将 DSPy 与 SPL 视为针对同一问题中相互正交的方面。未来的集成可以使用 DSPy 来优化每个 SELECT 项的内容，同时使用 SPL 来管理各项之间的预算。

7.4 命令式框架

LangChain [Chase、 2022], LlamaIndex [Liu 2022], 和 Semantic Kernel [微软 2023] 为大语言模型应用开发提供了命令式 API。它们功能强大且灵活，但要求开发人员手动管理令牌计数、上下文组装和截断。它们提供了构建模块（向量存储、记忆模块、提示模板），却将编排工作留给命令式代码。近来，LangChain 表达式语言（LCEL）和 LangGraph 引入了基于图的声明式组合原语，缩小了这一差距，但并未彻底消除；二者都未提供显式的令牌预算管理或执行前 EXPLAIN。AutoGen [Wu 等人、 2023]（微软）和 CrewAI [CrewAI, Inc. 2024] 等多智能体编排框架，则通过基于角色的智能体组合和多智能体对话模式扩展了这一领域。SPL-flow 的声明式图（第 9.4 节）旨在实现类似目标——将大语言模型智能体组合成弹性流水线——但它将完整的 PROMPT查询作为一等节点进行路由，并为其指定显式令牌预算，从而在编排层保留资源可见性保证，而自由形式的消息传递框架无法做到这一点。相较于上述所有框架，SPL 的价值主张类似于 SQL 相较于过程式数据库访问的价值：声明式层能够实现优化并提升透明性。

7.5 预算感知的大语言模型优化

近期研究开始将大语言模型令牌消耗视为首要优化考量。Zhao 等人 [Zhao 等人， 2025] 提出了适用于大语言模型驱动的排序操作的访问路径，并为基于大语言模型生成排名的 ORDER BY 语义设计了预算感知优化器。他们的工作与 SPL 的核心理念一致——令牌预算应由查询优化器管理，而非由应用程序代码管理——但其目标是特定操作员（排序），而非提供通用查询语言。SelfBudgeter [Li 等人， 2025]从输出侧着手提升令牌效率，通过训练模型在推理时动态分配推理令牌预算。SPL 与之互补：它在任何大语言模型调用之前，以声明式方式管理输入上下文预算，而 SelfBudgeter 则优化模型在生成答案时使用的词元数量。

7.6 输出结构化与受约束解码

有几种框架专注于强制规范输出结构，而非管理输入上下文。Guidance [微软研究院， 2023]（微软）和 Outlines [Willard 和 Louf， 2023]通过在词元生成时使用上下文无关文法或正则表达式约束解码器的 logit 分布，来强制实现结构化生成。BAML 和 TypeChat 采用模式优先的方法，通过提示工程和验证循环来保证 JSON 类型输出。SPL 在 GENERATE 中的 FORMAT json 子句目前依赖模型提示来规范输出结构；集成受约束解码后端（例如 Outlines）将提供形式化输出保证，也是自然的后续扩展方向。这些框架作用于输出边界，而 SPL 作用于输入边界——两者关注点彼此独立且互为补充。

7.7 提供商原生缓存

主要大语言模型提供商现已为静态前缀提供服务端提示缓存——Anthropic 的提示缓存[Anthropic、 2024], OpenAI 的类似功能，以及 Google Gemini 的上下文

缓存。SPL 的 `CACHE FOR` 子句是一个客户端语义层：它以渲染后的提示词为键，在用户指定的时长内缓存完整的大语言模型响应。这两种机制互为补充——提供商侧缓存可降低会话内重复静态前缀的成本，而 SPL 的客户端缓存则能完全消除跨会话相同查询的冗余 API 调用。未来的 SPL-flow 版本可以检测可缓存的静态前缀并自动启用提供商侧缓存，从而同时在两个层面实现节省。

8 评估

我们通过五项实验评估 SPL：(1) 与命令式 Python 的开发者体验对比，(2) 词元预算优化行为，(3) 跨模型成本估算，(4) EXPLAIN 输出展示，以及 (5) 系统性功能验证。所有实验均不调用大语言模型 API——它们用于测试解析器、分析器、优化器和 EXPLAIN 流水线，展示 SPL 如何在尚未 花费任何词元之前就能发挥价值。

8.1 实验 1：开发者体验

我们分别使用 SPL 和命令式 Python 实现五项基准任务，并衡量代码复杂度和开发者负担。Python 基线使用手动的 `tiktoken` [OpenAI、 2023]计数、显式截断和手写缓存——这些做法体现了现实中常见的提示工程实践。

表 6：五项基准任务中的开发者体验：SPL 与命令式 Python 对比。

Task	SPL LoC	Python LoC	缩减	令牌操作 (SPL/Py)	预算可见	静态验证
简单问答	9	20	55.0%	0 / 4	是 / 否	是 / 否
RAG增强问答	17	51	66.7%	0 / 9	是 / 否	是 / 否
多步公用表表达式	24	63	61.9%	0 / 7	是 / 否	是 / 否
函数复用	16	43	62.8%	0 / 11	是 / 否	是 / 否
缓存重复	9	42	78.6%	0 / 4	是 / 否	是 / 否
平均值	**15**	**44**	**65.0%**	**0 / 7**		

SPL 在五项任务中平均减少了**65% 的代码行数**，同时消除了全部 35 项手动词元计数操作。（词元操作是指任何明确调用词元计数 API、执行词元运算或应用手动截断逻辑的代码行，例如调用 `tiktoken.encode()`、 `len(tokens) > limit`，或根据词元计数进行子串截断。）每个 SPL 查询都支持执行前预算可见性（`EXPLAIN`）和静态验证（`spl validate`）——而命令式方法完全不具备这些能力。

8.2 实验 2：词元预算优化

分析优　化器在不同预算水平和上下文压力不断增大时的行为　　e.

8.2.1 不同预算下的分配

使用相同的查询结构，预算从 2K 到 32K 词元：

优化器按比例分配词元：检索增强生成和记忆随预算增加而扩展，而系统角色和问题保持固定。在 32K 时，检索增强生成上限（8K）和记忆上限（2K）均已达到，因而留下了大量缓冲区——开发者可通过 `EXPLAIN` 查看。

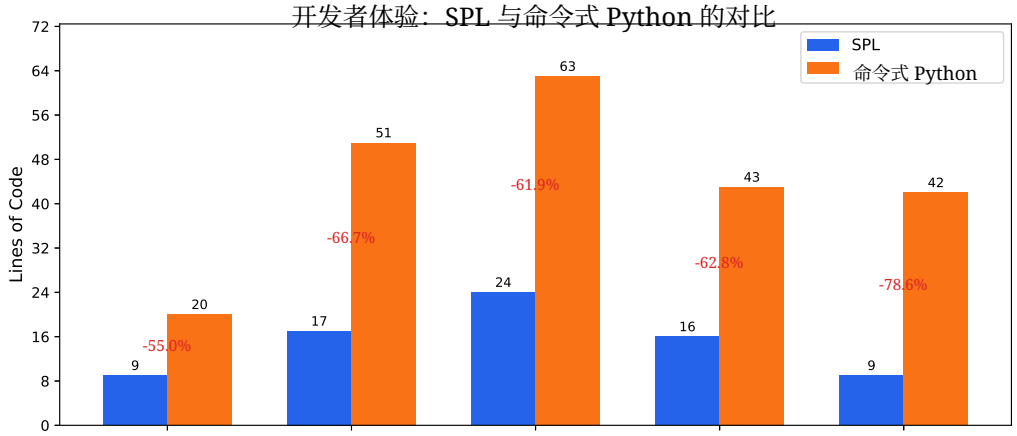

图 2：代码行数对比：SPL（蓝色）与命令式 Python（橙色）。百分比标签显示了代码缩减幅度。缓存查询的缩减幅度最大（78.6%）：SPL 提供自动缓存，而 Python 则需要手动实现缓存。

表 7：令牌分配策略会根据预算级别按比例调整。

预算系统	问题	RAG	记忆	输出缓冲区	利用率%	
2,000	20	200	666	250	500	364	81.8%
4,000	20	200	1,333	500	1,000	947	76.3%
8,000	20	200	2,666	1,000	2,000	2,114	73.6%
16,000	20	200	5,333	2,000	4,000	4,447	72.2%
32,000	20	200	8,000	2,000	8,000	13,780	56.9%

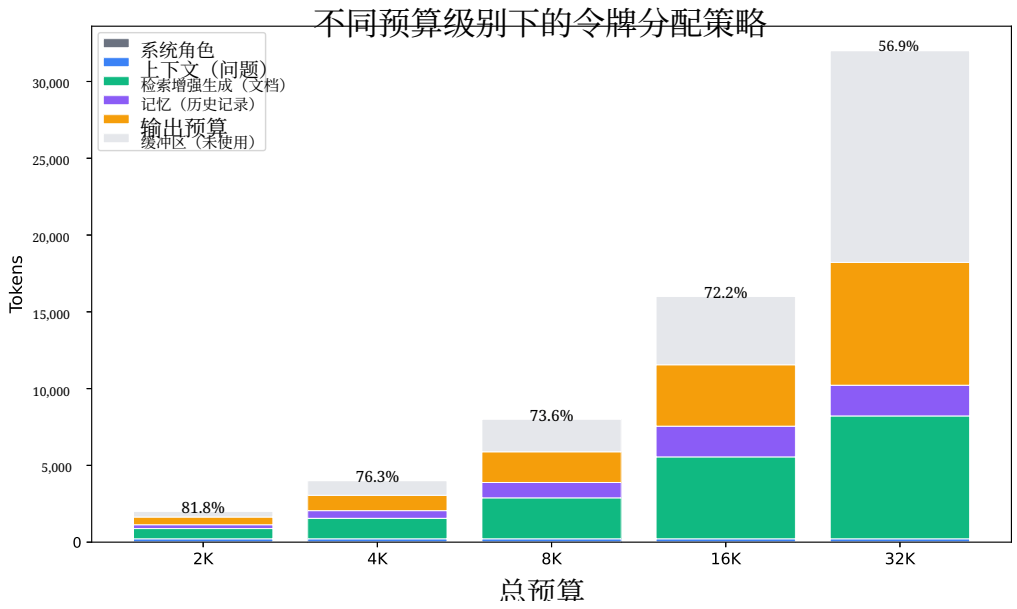

图3：不同预算水平下的堆叠式词元分配。检索增强生成（绿色）和记忆（紫色）按比例扩展；系统角色和问题保持不变。当来源限制达到上限时，缓冲区（灰色）会在高预算下增长。

8.2.2 预算压力下的压缩

当上下文需求超出预算时，优化器会应用按比例压缩（优先处理最大的项目）：

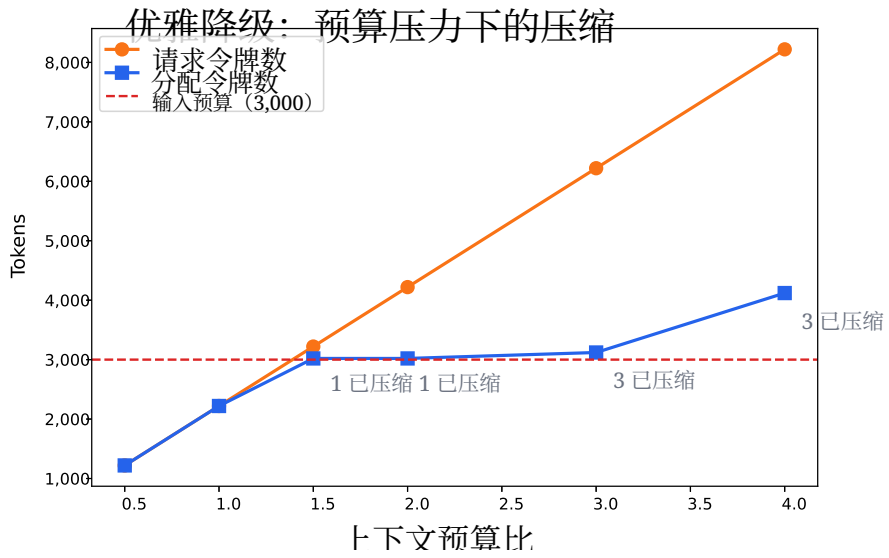

图4：优雅降级：随着上下文预算比从 0.5 倍增至 4 倍，优化器会逐步加强对来源的压缩，使其符合预算（红色虚线）。在超出预算 3–4 倍时，所有来源均被压缩 50%。

在超出预算 1.5 倍时，仅最大的来源（RAG 文档）会被压缩。在 3–4 倍时，所有来源均按比例压缩。此行为通过 `EXPLAIN` 完全透明，它会为被压缩的来源标注其压缩比。

8.3 实验 3：跨模型成本估算

SPL 的 `EXPLAIN` 机制可提供执行前成本估算。在六个模型上运行相同的 8K 预算查询：

表8：跨大语言模型的执行前成本估算（相同查询，3,720 个输入词元，1,000 个输出词元）。

模型	Tier	估计成本	相对
GPT-4（旧版）	传统高端	$0.2316	67.5倍
Claude Opus 4.6	高端	$0.2058	60.0倍
Claude Sonnet 4.5	均衡	$0.0412	12.0倍
GPT-4o	旗舰	$0.0293	8.5x
GPT-3.5 Turbo	预算	$0.0049	1.4x
Claude Haiku 4.5	经济型	$0.0034	1.0x

最昂贵与最便宜的模型之间的 **68× 成本差异** 在执行之前即可见。[2] 其他任何提示词管理框架均不提供此能力。这与 SQL `EXPLAIN` 揭示全表扫描成本高于索引查找的方式直接类似——从而支持有依据的优化。

[2]价格截至 2026 年 2 月，数据来自 OpenRouter。模型定价变动频繁；相对排序仅供示意。

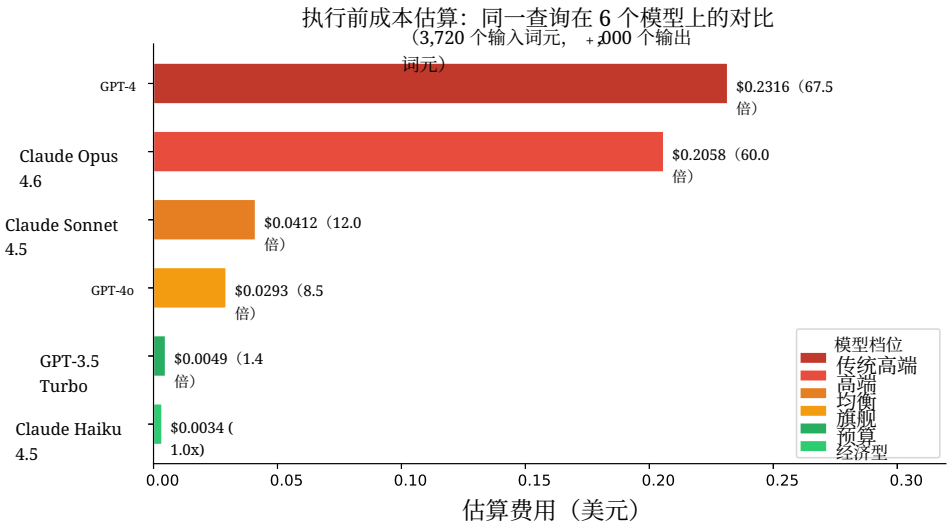

图5：执行前跨模型的成本比较。同一SPL查询在 Claude Haiku 上花费 $0.0034，而在 GPT-4 上花费 $0.2316——68倍的差异在任何词元被消耗之前即可看到。

8.4 实验4：EXPLAIN 输出

以下是 RAG增强型问答查询的实际 `EXPLAIN` 输出——即 SQL 执行计划的 SPL 等价物：

```
Execution Plan for: answer_question
============================================================
Budget: 8,000 tokens | Model: claude-sonnet-4-5

Token Allocation:
+-- __system_role__              20 tokens  (  0.2%)
+-- history                     500 tokens  (  6.2%)  [from memory]
+-- docs                      3,000 tokens  ( 37.5%)  [cache MISS]
+-- question                    200 tokens  (  2.5%)
+-- Output Budget             2,000 tokens  ( 25.0%)
\-- Buffer                    2,280 tokens  ( 28.5%)
                              ----------
Total                         5,720 / 8,000 tokens (71.5%)

Estimated Cost: $0.0412
```

该计划表明：(1) 首先获取记忆（优先级排序）；(2) RAG 文档占比最大（37.5%）；(3) 预算中有 28.5% 作为缓冲区保留；(4) 预估成本为 0.04 美元。开发者可以检查此计划、调整 `LIMIT` 值或切换模型，然后重新运行`EXPLAIN`——整个过程不会消耗任何词元。

8.5 实验5：特性验证

我们通过针对 SPL 引擎运行 20 项自动化检查，系统地验证了表5中的所有竞争性声明。20 项检查全部通过：

- 4/4 个示例文件均可成功解析（声明式语法）
- 3/3 项令牌预算功能正常运行（`WITH BUDGET`, `LIMIT`, `OUTPUT BUDGET`）
- 3/3 项 EXPLAIN 功能已验证（树形输出、百分比、成本）
- 2/2 检索增强生成特性正常运行（`rag.query()` 解析，可导入 VectorStore）

- 2/2 记忆功能正常运行（`memory.get()` 解析，可导入 MemoryStore）
- 1/1 公用表表达式（CTE）支持已验证（`WITH ... AS`）
- 1/1 函数支持已验证（`CREATE FUNCTION`）
- 2/2 提供方适配器可导入（Claude CLI、OpenRouter）
- 1/1 超出预算的查询会触发自动压缩
- 1/1 已确认是零依赖解析器（不导入 ANTLR/Lark）

8.6 效度威胁

我们的评估存在三项效度威胁，值得说明。第一，实验 1 中的 Python 基线是手动构建的，旨在代表常见的提示工程实践；经验丰富的开发者可能会编写出更精简的实现，因此，代码行数减少 65% 只是估算值，而非下限。第二，所有实验均为无 API 实验：我们没有测量大语言模型输出质量，因此，关于提示工程效率的结论只是间接得出的。第三，实验 3 中的成本估算反映的是截至 2026 年 2 月的模型定价；定价经常变动，因此这些绝对数值很快就会过时。

8.7 解析器正确性

参考实现通过了一套包含 40 项测试的全面测试套件：

- 词法分析器：14 项测试，涵盖关键字、字符串、数字、操作符、注释、点表示法的词法切分以及错误情况。
- 解析器：20 项测试，涵盖基本提示词、生成子句、WHERE 子句、CTE、STORE 子句，以及对所有示例文件进行端到端解析。
- 优化器：6 项测试，涵盖执行计划生成、令牌分配、执行顺序、EXPLAIN 输出呈现以及超预算压缩。

9 SPL 扩展与应用

SPL 生态系统不仅涵盖核心查询语言，还扩展到一系列已实现的应用，这些应用展示了 SPL 作为应用平台的能力。它们共同打通了从自然语言意图到可靠的多模型执行与交付的完整流程，也展现了高效大语言模型工程更广阔的架构方向。

9.1 Text2SPL：多语言自然语言接口

Text2SPL 模块通过大语言模型，将自由形式的自然语言查询转换为有效的 SPL 脚本，使没有编程背景的从业者也能使用 SPL。其流程为：自然语言 $\rightarrow$ 大语言模型（少样本） $\rightarrow$ SPL $\rightarrow$ `spl validate` $\rightarrow$ [重试 $\leq 3\times$] $\rightarrow$ 执行。少样本提示词提供涵盖常见查询模式的代表性 SPL 示例；验证与重试循环会自动修正语法无效的输出，从而支持对类型差异显著的查询和多种自然语言进行可靠的 SPL 生成。

在一项具有代表性的实验中，研究人员将英语、中文和阿拉伯语查询提交给 Text2SPL，并通过两种独立的适配器配置执行：本地 Ollama 和 OpenRouter 云端。全部六次生成+执行均首次尝试即完成，无需重试。表 9 总结了实验结果；完整生成的 SPL 脚本见附录 A。

关键的是，“自然语言”指的是任何自然语言，而不仅仅是英语。大语言模型接受以用户母语（中文、阿拉伯语、法语等）提出的查询，并将其直接翻译为 SPL，或通过英语中转，从而使 Text2SPL 成为真正的多语言接口。由此形成的流程（自然语言 $\rightarrow$ SPL $\rightarrow$ 执行）与关系数据库的自然语言接口相呼应 [Androutsopoulos 等， 1995]并拓展了自然语言到 SQL 的研究脉络 [Yu 等，

表 9：Text2SPL 实验：两个适配器（Ollama、OpenRouter）的自然语言→SPL 生成（阶段 1）和执行指标（阶段2）。六次运行均在首次尝试时生成成功。

语言	适配器	SPL 行数	Gen (s)	词元	Exec (s)	模型（自动路由）
英语	Ollama	32	54.4	1,653	35.4	gemma3
中文	Ollama	8	17.2	2,925	82.0	qwen3
阿拉伯语	Ollama	8	19.2	3,933	110.3	qwen3
英语	OpenRouter	32	61.9	1,974	30.1	claude-opus-4.6
中文	OpenRouter	8	16.0	4,434	65.5	claude-opus-4.6
阿拉伯语	OpenRouter	8	18.4	4,423	68.3	claude-opus-4.6

2018, 2019] 从静态关系表延伸到生成式知识库。附录A展示了多种语言中具有代表性的自然语言→SPL 脚本。

9.2 混合模型路由（USING MODEL auto）

USING MODEL auto 扩展会在执行时而非编译时解析目标模型。路由表将推断出的任务类别映射到专用模型:

表10：MoM 路由　　表：任务类别及代表性　　专用模型　　s.

任务类别	专业模型
中日韩语言（中文、日语、韩语）	通义千问2.5
代码生成 / 分析	DeepSeek-Coder
数学 / 形式化推理	deepseek-r1
多步推理 / 综合	Claude Opus
欧洲语言	mistral
通用	Llama 3.1

路由器根据 GENERATE 子句和 system_role 内容推断任务类型，然后为每个 PROMPT 子句分配一个专业模型，并行分发任务。推断采用基于规则的关键词匹配：路由器扫描 GENERATE 函数名称和 system_role 字符串，查找领域信号——中日韩字符集名称（kanji、hanzi、 chinese）、代码相关术语（function、 algorithm、 debug）、数学词汇（proof、equation、 theorem）——并将匹配结果映射到路由表（表10）。如果未检测到领域信号，则回退到通用模型。这种轻量级启发式方法避免了为分类额外调用大语言模型所带来的延迟和成本；学习型分类器是自然的扩展方向（第10节）。这种模型混合 （MoM）范式可提升异构多 CTE 脚本的输出质量，而且无需开发者预先了解每个模型的领域优势。

MoM 路由可以减少对显式智能体编排的需求，降低应用层的负担。开发者无需在应用程序代码中嵌入由智能体构成的图，并处理消息传递、重试和状态管理，而只需编写一份声明式 SPL 脚本；运行时，路由器会将每个 PROMPT 子句静默分配给相应的专业模型。（SPL-flow 本身在内部使用 PocketFlow[Huang,2025] 图，但这种复杂性被封装在声明式接口之下。）最终呈现的是一场*“AI 交响乐”*：一个多 CTE 查询，其中每个声部——由语言专家模型生成的中日韩文本、由代码模型生成的代码，以及由推理模型完成的最终综合——都恰逢其时地奏响，

整个过程由查询本身的声明式结构协调，而非由应用层中的命令式编排代码控制。MoM 路由实验的路由日志和模型分配轨迹见附录 B。

9.3 逻辑分块：长上下文的声明式 Map-Reduce s

优化器目前对超出预算的上下文的处理方式很简单，即直接截断——这是第 10 节承认的已知局限。SPL 的 CTE 架构提供了一种在结构上更优的替代方案：*逻辑分块*，将较长的源文档拆分为可独立处理的片段，并将每个片段分配给各自的 CTE，最后由 `GENERATE` 子句综合每个分块的输出。这就是声明式 *Map-Reduce*：

- Map 阶段：每个 CTE 独立处理一个分块，也可以通过 `USING MODEL auto`将其分派给领域专家模型。
- Reduce 阶段：外层 `GENERATE` 将各分块的输出综合为一个连贯的整体。

```
PROMPT analyze_research_paper
WITH BUDGET 32000 tokens
USING MODEL auto     -- each CTE routed to its specialist

-- Map phase: independent CTE per document section
WITH chunk_intro AS (
    SELECT context.section_intro AS text LIMIT 3000 tokens
    GENERATE section_summary(text)
    WITH OUTPUT BUDGET 600 tokens
),
chunk_method AS (
    SELECT context.section_method AS text LIMIT 3000 tokens
    GENERATE section_summary(text)
    WITH OUTPUT BUDGET 600 tokens
),
chunk_results AS (
    SELECT context.section_results AS text LIMIT 3000 tokens
    GENERATE section_summary(text)
    WITH OUTPUT BUDGET 600 tokens
),
chunk_discussion AS (
    SELECT context.section_discussion AS text LIMIT 3000 tokens
    GENERATE section_summary(text)
    WITH OUTPUT BUDGET 600 tokens
)

-- Reduce phase: synthesize all chunk summaries
SELECT
    system_role("You are a thorough research analyst"),
    chunk_intro      AS intro_summary,
    chunk_method     AS method_summary,
    chunk_results    AS results_summary,
    chunk_discussion AS discussion_summary

GENERATE
    comprehensive_review(intro_summary, method_summary,
                         results_summary, discussion_summary)
WITH OUTPUT BUDGET 2000 tokens, TEMPERATURE 0.2, FORMAT markdown

STORE RESULT IN memory.paper_review;
```

清单 7：　通过 CTE 实现逻辑分块：针对长篇研究论文的声明式 Map-Reduce r.

文档拆分使用开源分块工具（例如 LangChain、LlamaIndex、chonkie）；SPL 实际上提供了声明式编排层。

计算效率。Transformer模型中的注意力机制随序列长度呈 $O(n^2)$增长。将包含 N 个词元的文档作为单个提示词处理，成本为 $O(N^2)$。将其拆分为 k个分块，每块包含 N/k个词元，并行处理的成本为 $O(k \cdot (N/k)^2) = O(N^2/k)$——注意力计算量降低至原来的 k 分之一。对于 $k = 8$，这意味着降低 $8\times$，且与模型规模或服务提供方无关。这个 $O(N^2/k)$ 数值指注意力浮点运算总次数（计算成本）。要提升墙钟时间，还需要并行执行；将分块分派给互相独立的工作进程（例如通过 OpenRouter）时，所有 k 个分块都能在大约 $1/k$ 的顺序执行墙钟时间内完成。顺序执行（例如使用单个本地 Ollama 实例）仍能实现相同的计算量削减，但墙钟时间会按比例延长。

在一项具有代表性的实验中，《*Attention Is All You Need*》 [Vaswani等人， 2017]被处理为 k=4 个并行分块（摘要+引言、架构、训练、结果+结论），并由 `claude-haiku-4.5` 通过 OpenRouter 分别进行总结。四个 Map 公用表表达式同时分派，并在 5.2 秒墙钟时间内完成（若顺序执行则需 ≈20.8 秒），与上述 $O(N^2/k)$ 分析一致，注意力计算量减少至原来的四分之一；Map 阶段共消耗 1,914 个词元。Reduce步骤（`claude-sonnet-4-5`，1,200 词元预算）用 32 秒综合生成了一份结构化的批判性评述，总计 4,749 个词元，估算成本为 0.025 美元。完整脚本和综合输出见附录 C。

逻辑执行与物理执行。声明式分块将逻辑 查询结构与物理 执行策略分离。同一 `.spl` 脚本无需修改，即可在两种有效模式下运行，例如：(1) 并行（云端）：通过 `asyncio.gather` 将CTE并发发送至OpenRouter或直接云端API；几秒内即可得到结果；(2) 顺序执行（本地）：由本地运行的Ollama实例在消费级GPU上逐个执行CTE；一夜之间即可得到相同结果，且无需支付边际成本。

这体现了SQL的基础抽象：同一 `SELECT` 查询既可以在笔记本电脑上的SQLite数据库上运行，也可以在分布式集群上运行。逻辑意图保持不变，变化的只有物理执行引擎。在消费级硬件上顺序执行一整夜，是完全可行的一流模式——让个人、研究人员以及对隐私敏感的部署能够以零成本、零云端依赖运行复杂的多模型工作流。

与开源分块器集成。文档切分工作交由成熟的开源库完成——LangChain的 `RecursiveCharacterTextSplitter` [Chase、 2022], LlamaIndex节点解析器 [Liu 2022], 以及语义句子分块器——先生成逻辑连贯的片段，再将它们绑定到SPL上下文变量。SPL不会重新实现分块功能，而是提供声明式编排层，协调分块的处理与综合方式。

架构启示。逻辑分块与 MoM 路由相结合表明，跨专用模型进行拆解与编排的能力，可以替代——而不仅仅是补充——不断扩大的上下文窗口。将大型文档拆分为k 个分块并交由专用 7B 模型处理的系统，其表现可能媲美使用百万词元上下文窗口的单个 70B 通用模型，而推理成本仅为其一小部分。我们将此作为一项架构假设，供社区进一步研究；该假设以以上 $O(N^2/k)$ 效率论证为基础，并得到第 9.4 节所述 SPL-flow 实验的支持。附录 C 展示了完整的分块脚本，以及一项具有代表性的长文档实验所生成的综合输出。

9.4 SPL-flow：弹性智能体编排

SPL-flow 使用 PocketFlow [Huang、 2025] 图中的五个节点，将 SPL 查询编排成有向智能体流水线：`Text2SPLNode` → `ValidateSPLNode` （重试 $\leq 3\times$） → `ExecuteSPLNode` → `SyncDeliverNode` | `AsyncDeliverNode`。图 6 展示了完整的编排图。

该平台可通过 Streamlit 网页用户界面、Python API（`pip install spl-flow`）和 Click 命令行界面访问。

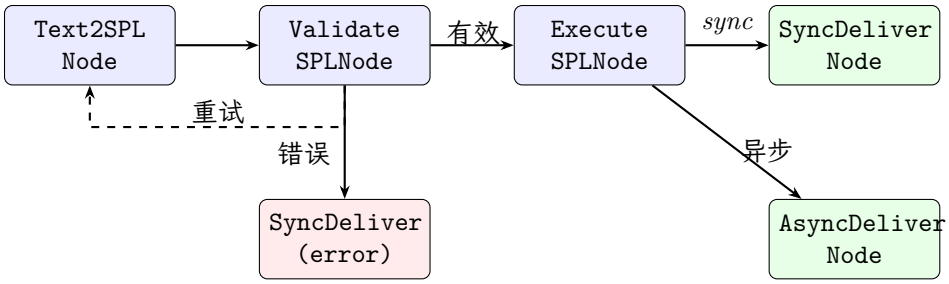

图 6：SPL-flow PocketFlow 编排图。 `Text2SPLNode` 使用用户选择的适配器将自然语言转换为 SPL； `ValidateSPLNode` 负责解析和验证，若输出无效则最多重试 3× 次； `ExecuteSPLNode` 通过 `asyncio.gather` 并行分派 CTE，并采用 MoM 路由；交付方式为同步（直通）或异步（基于文件）。虚线箭头表示重试回边。

9.4.1 韧性工程

SPL-flow 实现了一种提供方回退策略，以实现容错执行。每次适配器调用都由包含三个层级的重试策略封装：

- 主用 — Ollama（本地）：零成本，完全私密，适用于离线和对隐私敏感的工作负载。
- 回退 — OpenRouter / 直接云端 [OpenRouter, 2024]：当 Ollama 超时、内存不足或出现模型不可用错误时自动启用；可访问 100+云模型。
- 自愈式重试：采用带抖动的指数退避；每个适配器最多尝试三次，然后升级为回退方案。

这种自愈式执行模型意味着，面向本地优先部署的开发者无需在 Ollama 不可用时修改其 `.spl` 脚本——SPL-flow 会透明地重新路由至云端。这种模式类似于数据库读副本故障转移，如今应用于大语言模型应用。

9.4.2 已验证结果：多模型编排实验

在一个具有代表性的 SPL-flow 实验中，一段包含两个 CTE 的脚本并行协调了两个专用模型——通义千问 2.5（中日韩语言分析）和 Mistral（德语翻译）——并由 Claude 将输出综合为一张结构化表格，呈现中文部首「日」（太阳/白天）字族，并为每个字符标注构形公式、拼音、英语释义、德语对应词和自然语义洞见。云端成本总计：$0.002（OpenRouter）。同一段 `.spl` 脚本在本地 Ollama 实例上以顺序过夜模式运行，成本为零，且无需修改。完整的「日」字族 `.spl` 脚本及生成的字符表见附录 D。

9.5 BENCHMARK：并行模型比较与优胜者选择

`BENCHMARK` 引入了一种全局模型比较语法，可覆盖各查询中的 `USING MODEL`声明：

```
    -- Global declaration: run every PROMPT against all listed models


PROMPT answer_question
```

```
WITH BUDGET 8000 tokens
USING MODEL claude-sonnet-4-5    -- overridden by global USING MODELS

SELECT
    system_role("You are a knowledgeable assistant"),
    context.question AS question LIMIT 200 tokens,
    rag.query("relevant docs", top_k=3) AS docs LIMIT 3000 tokens

GENERATE detailed_answer(question, docs)
WITH OUTPUT BUDGET 2000 tokens;
```

清单 8: USING MODELS 全局语法覆盖本地模型声明，并并行运行所有候选项

所有列出的模型都会通过 asyncio.gather 并行执行；对于云适配器，所有请求并行发送；对于本地适配器（例如 Ollama），由于单个 GPU 无法处理并发推理工作负载，引擎会通过信号量串行处理请求。两种情况下，引擎都会将响应文本、词元消耗量、延迟和成本收集到结构化 JSON 报告中。**获胜者** 由用户可配置的目标函数选出（例如，在满足质量阈值的前提下最小化成本，或在延迟预算内最大化质量）。默认质量指标是 由大语言模型担任评审 评分：基准测试运行器会将收集到的响应提交给指定的评审模型，并附上评分标准，为每个候选项返回一个数值分数。用户可以改用基于参考答案的指标（BLEU、ROUGE）或领域特定评分函数；目标函数根据最终得到的 (质量、成本、延迟) 元组进行运算。获胜模型的标识符会自动写回源 USING MODEL 子句，并可通过 STORE RESULTIN memory.best_model持久化，因此后续执行会使用实测最优模型，无需重新运行基准测试。这让用户只需使用单一声明式配置，即可轻松从测试运行切换到部署——类似于 PostgreSQL 中的 EXPLAIN ANALYZE，后者会衡量查询成本的实际值与估算值。

表 14 展示了第 §9 节中中文“日”字旁多语言表格任务的七种模型对比，该任务通过 Ollama 在本地运行（API 费用为零）。 mistral 的延迟最低（44.4 秒，576 个输出词元），而 phi4 的总体输出最精简（521 个输出词元，88.3 秒）。内置思维链推理的模型（qwen3、deepseek-r1）生成的词元比精简型模型多出 3–5× 个；即使货币成本为零，这仍是一个不可忽视的选择依据，因为输出词元数量会直接影响下游上下文预算的消耗。各模型的完整指标及 JSON 报告见附录 E。

10 局限性与未来工作

10.1 当前局限性

- 轻微超出预算时的截断：逻辑分块（第 9.3 节）解决了结构化长上下文拆分问题；不过，对于无需完整分块处理的轻微 CTE 预算超限情况，简单截断仍是备用方案。语义压缩——通过成本较低的模型概括超出预算的内容，而非直接删减——是自然而然的下一步。

- 类型系统有限：SPL 尚未为上下文来源提供类型化模式。类型化 RAG 文档模式、记忆值类型和函数签名将有助于进行更深入的静态分析，并改进 IDE 工具。

- 评估范围：当前基准测试套件涵盖问答变体。更广泛的任务多样性——代码生成、多轮对话、复杂的多跳推理和智能体工具使用——是目前使用 SPL-flow 开展评估工作的重点。

- 多轮对话管理：虽然 memory.get("history") 提供对话历史记录访问功能，但用于精简长对话线程的显式语言层面策略（淘汰最早内容、

优先淘汰最早内容、摘要压缩、按相关性加权保留）尚未在 SPL 语法中正式定义。

- 基于关键词的 MoM 路由：当前实现通过基于规则的关键词匹配（例如，中日韩字符模式 → `qwen2.5`、代码词元 → `deepseek-coder`）将子任务路由到专用模型。这种启发式方法透明且可预测，但面对含糊不清或跨领域查询时较为脆弱。基于嵌入或学习式的路由——即路由器本身是一个轻量级模型——被视为主要的改进方向（另请参阅第 10 节中的“动态 MoM 路由”）。
- 语法与实现的对应关系：已发布的扩展巴科斯-瑙尔范式（附录 F）通过了包含 40 项测试的测试套件验证，并已与参考实现进行人工交叉核对。形式化的机械验证——例如，通过解析器生成工具直接根据扩展巴科斯-瑙尔范式生成解析器，并确认所有产生式均符合 LL(1) 规范性——仍有待未来开展。

10.2 学习式优化

SPL 的优化器目前采用启发式规则；未来的方向包括：

- 令牌分配学习：基于执行轨迹进行训练，根据查询结构和历史性能预测每个来源的最佳令牌分配。
- 分块策略选择：学习如何针对给定的任务类型和模型，以最佳方式划分文档（按句子、段落、语义边界或结构部分）。
- 动态 MoM 路由：改进任务类型推断，并根据执行历史中观察到的模型性能更新路由表。

10.3 与现有生态系统集成

SPL 旨在与现有工具互补，而非取而代之：

- SPL + DSPy：在 SPL 的令牌预算框架内使用 DSPy 优化提示词内容；两者解决的是相互独立的问题。
- SPL + LangChain/LlamaIndex：将 SPL 用作现有检索和分块基础设施之上的声明式编排层 [Chase，2022年，Liu，2022]。
- 集成开发环境中的 SPL：在 VS Code、JetBrains 等工具中，为 `.spl` 文件提供语法高亮、自动补全和内联 `EXPLAIN`。
- 用于合规的 SPL：将 `EXPLAIN` 输出为标准格式，以便进行提示审计、成本治理和 AI 合规报告。

10.4 安全与提示注入加固

安全性是 SPL 设计中的首要考量。语法本身提供了结构性防护措施，可缩小大语言模型驱动的应用的攻击面，但无法将其完全消除。

内置的结构性防护。

- 通过 Text2SPL 实现结构化沙箱隔离：自由格式的用户输入会先被转换为符合语法的 SPL 脚本，然后才会传递给任何下游模型。不符合 SPL 语法的任意文本都会被解析器拒绝，从而防止整类原始提示注入攻击，例如攻击者将对抗性指令直接嵌入请求字符串。
- 权限分离：`system_role(…)` 槽位在结构上与用户可控的 `context.*` 字段和 `GENERATE` 任务字符串相互独立。编写完善的 SPL 脚本不会因输入数据而在无声无息中覆盖系统角色，除非明确修改语法层级的规则。

- 令牌预算上限： `WITH BUDGET` 和 `OUTPUT BUDGET` 子句会限制生成长度，从而遏制依赖诱导无限输出的放大攻击。
- 先声明后执行模型：在发起任何大语言模型调用之前，完整的执行计划就已可见（`EXPLAIN`）。操作员或策略引擎可以在执行前检查并拒绝脚本——类似于 SQL 查询防火墙。

尚待解决的问题与未来工作的展望。SPL 验证器检查的是语法正确性，而非语义安全性。通过 `ValidateSPLNode` 验证的脚本在结构上是合理的，但仍可能在引用的字符串字面量中包含有害意图。我们认为，SPL 的声明式结构让以下尚待解决的问题更易于在未来加以处理：

- 对字面量进行语义内容过滤：策略层可以在将内容发送给大语言模型之前，扫描 `system_role(…)` 与 `GENERATE` 字符串字面量，过滤不允许的内容。这类似于在查询层（而非 HTTP 层）运行的 Web 应用防火墙。
- 模型允许列表：可以将 `USING MODEL` 子句限制为经操作员批准的一组模型，防止通过对抗性构造脚本将请求路由到未经内容审查或未经审核的模型。
- RAG 注入防护：通过 `rag.query(…)` 检索到的结果本身可能包含对抗性指令；在进行 CTE 上下文注入之前加入净化步骤，可以缓解经由检索引发的间接提示注入。
- 函数体沙箱化：`CREATE FUNCTION …AS $$…$$` 接受任意文本；为用户定义函数提供能力受限的执行环境，是一种自然的扩展。

我们认为，SPL 以语法为先的声明式架构，比临时的字符串操作更能为提示安全奠定严谨的基础，正是因为其攻击面由形式文法明确定义（附录 F）。我们欢迎安全研究社区基于 SPL 执行模型构建内容安全、访问控制和审计层。

11 结论

我们介绍了 SPL（结构化提示语言），这是一种用于将大语言模型上下文作为受限资源进行管理的声明式查询语言。SPL 运用了 SQL 成功背后的原则——声明式规范、自动优化、可组合性、可移植性和执行计划透明度——将提示工程从临时的命令式实践转变为一门结构化、可优化的学科。

SPL 的以下关键创新共同构成了一个全新的声明式框架，能够高效、可靠地支持广大从业者开展大语言模型应用开发。通过抽象提示工程和模型编排中的复杂性，SPL 使开发人员能够专注于核心应用逻辑，同时依托底层各层优化资源使用和执行策略。

1. 将令牌预算作为一等概念： `WITH BUDGET` 和 `LIMIT` 子句使资源约束一目了然且可优化。2. 生成式知识库模型： `SELECT/GENERATE` 二元性体现了上下文检索与内容合成之间的根本差异。3. 面向大语言模型查询的 EXPLAIN：透明的执行计划让开发者能够在执行前检查并优化令牌分配。4. 集成式上下文管理：内置检索增强生成（FAISS/ChromaDB）和记忆（SQLite）无需依赖外部上下文基础设施。5. Text2SPL：提供多语言自然语言→SPL 接口。6. 混合模型路由（`USING MODEL auto`）支持声明式多模型编排，可在运行时将每个 `PROMPT` 子句分派给领域专家模型。7. 作为声明式 Map-Reduce 的逻辑分块：基于 CTE 的文档拆分可将 k 个分块的 Transformer 注意力成本从 $O(N^2)$ 降至 $O(N^2/k)$，同时保持相同的

脚本可通过 Ollama 在云端并行运行，也可在消费级 GPU 上于夜间顺序运行，且无需额外边际成本。8. 弹性多提供商执行：SPL-flow 实现了三级回退策略（Ollama → OpenRouter → 自愈式重试），无需修改 `.spl` 脚本即可实现容错执行。9. 基准测试：`USING MODELS` 支持并行多模型比较、优胜者选择以及自动部署持久化。

我们的成果以两个开源 Python 软件包和一个 VS Code 扩展的形式发布：（1）SPL 语言，位于 `https://github.com/digital-duck/SPL` (`pip install spl-llm`)；（2）SPL-flow，位于 `https://github.com/digital-duck/SPL-flow` (`pip install spl-flow`)；（3）VS Code 扩展，位于 `https://github.com/digital-duck/SPL-LLM`，另附完整的文档和示例。我们欢迎社区探索、评议并基于 SPL 和 SPL-flow 进行开发。

11.1 更广泛的影响

SPL 在这一领域中占据独特地位，因为它结合了：

- 一种独立的声明式语言（类似 SQL，不同于 DSPy 嵌入 Python 的方式或 LangChain 的命令式 API）
- 将令牌预算管理作为首要考量（不同于所有其他方法）
- 与检索增强生成和记忆集成的上下文来源（不同于仅使用模板的方法）
- 执行计划透明度，通过 `EXPLAIN`（SPL 独有）

逻辑分块 中的效率论据还蕴含着更广泛的架构意义。随着大语言模型推理成本和能源消耗日益成为工程和社会关注的焦点 [Han 等人， 2024], 以声明式方式分解复杂查询的能力——将每个分块或子任务路由给特定领域的小型专家模型，而不是把越来越大的上下文发送给一个庞大的单体模型——提供了一条结构迥异的扩展路径。每次请求所需注意力计算的 $O(N^2/k)$ 减少，加上通过 MoM 路由至 70 亿至 130 亿参数的专家模型，以及在消费级 GPU 硬件上零成本顺序执行的选项，表明在许多实际工作负载中，声明式编排或许可以替代上下文窗口扩展。我们将此作为一项架构假设，供业界进一步探讨。

除软件工程领域外，SPL 还降低了广大数据工程师和分析师群体的入门门槛，他们已经具备扎实的 SQL 专业知识。正如作者本人从数据工程转向 AI 工程的经历所示——先是构建由检索增强生成驱动的分析应用 [Gong， 2024] 这得益于多年 Oracle 和 SQL 实践经验，随后又参与 SPL 本身的开发——从 `SELECT` 操作关系表到 `PROMPT/SELECT/GENERATE`操作大语言模型上下文源，其概念跨度远小于从零开始学习通用编程框架。对于数百万习惯以集合、模式和声明式查询进行思考的从业者而言，SPL 为进入生成式人工智能领域提供了一条熟悉的入门路径，既保留并扩展他们已有的思维模型，也不必将其抛弃。

自然语言到SQL社区有可能代表一个早期采用者群体。从事将自然语言翻译为SQL的研究人员与从业者[Yu等人， 2018, 2019] 已经精通这一核心范式——将自由形式意图映射到结构化查询语言——而从自然语言→SQL扩展到自然语言→SPL是最小的概念跨越。自然语言到SQL系统面向静态 知识库（关系表），而Text2SPL面向生成式知识库（大语言模型权重），从而使SPL成为该研究社区自然的下一个研究领域。

致谢

这项工作由人类+人工智能协作完成。作者提供了构想、领域专业知识（20+ 年的 SQL/Oracle经验）、架构决策，以及实验设计与执——

行；AI助手则协助进行算法设计、编程、单元测试、手稿起草，并对草稿提供批判性反馈。

作者感谢 FAISS、ChromaDB、tiktoken 背后的开源社区，以及 SPL 所依赖的广泛 Python 生态系统；还特别感谢以下 3 个大语言模型：Claude（Anthropic） [Anthropic、2026], Gemini（谷歌） [Google，以及 2026], GLM-5（智谱AI） [GLM-5 Team 2026]。

参考文献

Ion Androutsopoulos、Graeme D. Ritchie 和 Peter Thanisch。自然语言数据库接口——简介。自然语言工程，1(1):29–81，1995年。Anthropic。提示缓存。 https://docs.anthropic.com/en/docs/build-with-claude/prompt-caching，2024年。Anthropic 文档。访问日期：2026年2月22日。Anthropic。Claude 4.6 sonnet。大语言模型，2026。网址 https://www.claude.ai。访问日期：2026年2月21日。Luca Beurer-Kellner、Marc Fischer 和 Martin Vechev。提示即编程：一种面向大语言模型的查询语言。《*ACM* 编程语言汇刊》，7 (PLDI):1946–1969，2023年。Tom Brown、Benjamin Mann、Nick Ryder、Melanie Subbiah、Jared D. Kaplan、Prafulla Dhari- wal、Arvind Neelakantan、Pranav Shyam、Girish Sastry、Amanda Askell 等。语言模型是少样本学习器。《神经信息处理系统进展》，33:1877–1901，2020年。Donald D. Chamberlin 和 Raymond F. Boyce。SEQUEL：一种结构化英语查询语言。*1974*年 *ACM SIGFIDET* 数据描述、访问与控制研讨会论文集，第 249–264 页，1974年。Harrison Chase。LangChain：通过可组合性构建大语言模型应用。 https://github.com/langchain-ai/langchain，2022年。访问日期：2026年2月12日。Chroma Core。ChromaDB：开源嵌入数据库。 https://github.com/chroma-core/chroma，2023年。GitHub代码仓库。Edgar F. Codd。大型共享数据库的数据关系模型。《*ACM* 通讯》，13(6):377–387，1970年。CrewAI, Inc.。CrewAI：用于编排角色扮演型自主人工智能智能体的框架。 https://github.com/crewAIInc/crewAI，2024年。GitHub代码仓库。Eshraw。结构化提示语言的 VS Code 语言扩展。 https://github.com/eshraw/structured-prompt-language，2025 年。GitHub代码仓库。GLM-5 Team。GLM-5：从氛围编程到智能体工程。*arXiv* 预印本 *arXiv:2602.15763*，2026年。网址 https://arxiv.org/abs/2602.15763。Wen G. Gong。Data-Copilot：由 RAG 驱动的对话分析应用。 https://github.com/digital-duck/data-copilot，2024年。GitHub代码仓库。谷歌。Gemini（2月21日版本）。大语言模型，2026。网址 https://gemini.google.com/。访问日期：2026年2月21日。Tingxu Han、Zhenting Wang、Chunrong Fang、Shiyu Zhao、Shiqing Ma 和 Zhenyu Chen。令牌预算感知型大语言模型推理。*arXiv* 预印本 *arXiv:2412.18547*，2024年。网址 https://arxiv.org/abs/2412.18547。Zachary Huang。PocketFlow：面向智能体和工作流的极简大语言模型框架。 https://github.com/The-Pocket/PocketFlow，2025年。GitHub代码仓库。Jeff Johnson、Matthijs Douze 和 Hervé Jégou。利用图形处理器进行十亿规模的相似性搜索。《*IEEE* 大数据汇刊》，7(3):535–547，2019年。

Omar Khattab、Arnav Singhvi、Paridhi Maheshwari、Zhiyuan Zhang、Keshav Santhanam、Sri Vardhamanan、Saiful Haq、Ashutosh Sharma、Thomas T. Joshi、Hanna Moazam、Heather Miller、Matei Zaharia 和 Christopher Potts。《DSPy：将声明式语言模型调用编译为自我改进流水线》。载于第十二届国际学习表征会议（*ICLR*），2024年。URL https://arxiv.org/abs/2310.03714。Patrick Lewis、Ethan Perez、Aleksandra Piktus、Fabio Petroni、Vladimir Karpukhin、Naman Goyal、Heinrich Küttler、Mike Lewis、Wen-tau Yih、Tim Rocktäschel 等。面向知识密集型自然语言处理任务的检索增强生成。《神经信息处理系统进展》，33：9459–9474，2020年。Zheng Li、Qingxiu Dong、Jingyuan Ma、Di Zhang、Kai Jia 和 Zhifang Sui。SelfBudgeter：面向高效大语言模型推理的自适应词元分配。*arXiv*预印本 *arXiv:2505.11274*，2025年。URL https://arxiv.org/abs/2505.11274。Jerry Liu。LlamaIndex：面向大语言模型应用的数据框架。https://github.com/run-llama/llama_index，2022年。访问日期：2026年2月12日。微软。Semantic kernel：快速、轻松地将尖端大语言模型技术集成到应用中。 https://github.com/microsoft/semantic-kernel，2023年。访问日期：2026年2月12日。微软。Prompty：大语言模型提示词的资产类别与格式。 https://github.com/microsoft/prompty，2024年。GitHub代码仓库。访问日期：2026年2月12日。微软研究院。Guidance：用于控制大语言模型的引导语言。https://github.com/guidance-ai/guidance，2023年。GitHub代码仓库。OpenAI。tiktoken：一款适用于 OpenAI 模型的快速 BPE分词器。 https://github.com/openai/tiktoken，2023年。访问日期：2026年2月12日。OpenRouter。OpenRouter：面向大语言模型的统一API。 https://openrouter.ai，2024年。访问日期：2026年2月12日。Ashish Vaswani、Noam Shazeer、Niki Parmar、Jakob Uszkoreit、Llion Jones、Aidan N. Gomez、Łukasz Kaiser 和 Illia Polosukhin。Attention is all you need。《神经信息处理系统进展》，30，2017年。Jason Wei、Xuezhi Wang、Dale Schuurmans、Maarten Bosma、Brian Ichter、Fei Xia、Ed Chi、Quoc V. Le 和 Denny Zhou。思维链提示能够引出大语言模型的推理能力。《神经信息处理系统进展》，35：24824–24837，2022年。Brandon T. Willard 和 Rémi Louf。大语言模型的高效引导式生成。https://arxiv.org/abs/2307.09702，2023年。arXiv预印本 arXiv:2307.09702。Qingyun Wu、Gagan Bansal、Jieyu Zhang、Yiran Wu、Beibin Li、Erkang Zhu、Li Jiang、Xiaoyun Zhang、Shaokun Zhang、Jiale Liu、Ahmed Hassan Awadallah、Ryen W. White、Doug Burger 和 Chi Wang。AutoGen：通过多智能体对话赋能新一代大语言模型应用。https://arxiv.org/abs/2308.08155，2023年。arXiv预印本 arXiv:2308.08155。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。Spider：用于复杂跨领域语义解析和文本转SQL任务的大规模人工标注数据集。载于《*2018*年自然语言处理实证方法会议（*EMNLP*）论文集，2018年。URL https://arxiv.org/abs/1809.08887。

Tao Yu、Rui Zhang、Michihiro Yasunaga、Yi Chern Tan、Xi Victoria Lin、Suyi Li、Heyang Er、Irene Li、Bo Pang、Tao Chen、Emily Chi、Zeyu Hu、Naoki Kitaev、Richard Zhu、Lucas Yeh、Samuel Ting 和 Dragomir Radev。SParC：上下文中的跨领域语义解析。载于计算语言学协会第*57*届年会论文集（*ACL*），2019年。URL https://arxiv.org/abs/1906.02285。

Fuheng Zhao、Jiayue Chen、Yiming Pan、Tahseen Rabbani、Sohaib、Divyakant Agrawal、Amr El Abbadi、Paritosh Aggarwal、Anupam Datta 和 Dimitris Tsirogiannis。《大语言模型的高效排序访问路径》。*arXiv* 预印本 *arXiv:2509.00303*，2025年。URL https://arxiv.org/abs/2509.00303。

A Text2SPL：代表性自然语言SPL示例

这三个脚本均由 Text2SPL 生成，并且首次尝试即成功执行（完整指标见表 9）。英语脚本（查询 1）的结构最为丰富：它使用两个 CTE，并明确指定了模型（`qwen2.5` 用于中日韩分析，`mistral` 用于德语翻译），这表明当查询涉及多个语言领域时，Text2SPL 会自然地生成经 MoM 路由的多公共表表达式脚本。

A.1 英语中的查询 1

> 生成一个包含带有部首日 （rì）的汉字的多语言表格，包括汉字拆分公式、拼音、英文释义、中文解释、德语翻译和自然语义。

```
PROMPT chinese sun radicals_   _WITH BUDGET 8000 tokensUSING MODEL "auto"
```

列表 9：Text2SPL 输出——英语查询（32 行数，采用显式模型路由的双 CTE MoM 脚本）。

A.2 查询 2（中文）

用中文解释大语言模型的工作原理，从参数知识、上下文知识和推理能力三个维度进行分析，并比较 GPT、Claude和开源模型（如通义千问）的主要异同。(用中文解释大语言模型如何工作，分析参数化知识、上下文知识和推理能力；比较 *GPT*、*Claude* 与通义千问等开源模型。)

```
PROMPT llm_working_principles_analysis
WITH BUDGET 6000 tokens
USING MODEL "qwen3"

SELECT
    system_role("You are an AI and machine learning expert
      specializing in LLM architecture, training, and performance.")
GENERATE
    explain_llm_principles("Explain in Chinese how LLMs work across
      three dimensions: (1) parametric knowledge, (2) contextual
      knowledge, (3) reasoning capabilities. Compare GPT, Claude, and
      open-source models (Qwen, LLaMA, Mistral) on training methods,
      architecture, alignment techniques, and performance.")
WITH OUTPUT BUDGET 4000 tokens, TEMPERATURE 0.1, FORMAT markdown;
```

列表 10：Text2SPL 输出 — 中文查询（8 行数，单提示词， qwen3 自动路由）。

A.3 阿拉伯语查询 3

ما أبرز إسهامات العلماء العرب في تطوير علم الرياضيات والفلك خلال العصر الذهبي الإسلامي؟(在伊斯兰黄金时代，阿拉伯学者对数学和天文学作出了哪些最突出的贡献？)

```
PROMPT arabic_golden_age_contributions
WITH BUDGET 8000 tokens
USING MODEL "qwen3"

SELECT
    system_role("You are a historian of Islamic science specializing
      in the Islamic Golden Age (8th--13th centuries), with expertise
      in contributions to mathematics, astronomy, and their influence
      on modern science.")
GENERATE
    explain_contributions("Explain in Arabic the contributions of
      Arab and Islamic scholars to mathematics and astronomy during
      the Islamic Golden Age. Cover: algebra (al-Khwarizmi),
      trigonometry, astronomical instruments, key scholars
      (al-Battani, Ibn al-Haytham, Omar Khayyam, al-Biruni),
      the House of Wisdom, transmission to Europe, and lasting
      influence on modern science.")
WITH OUTPUT BUDGET 4000 tokens, TEMPERATURE 0.2, FORMAT markdown;
```

清单 11：Text2SPL 输出——阿拉伯语查询（8 行数，单提示词， qwen3 自动路由）。

B 模型混合：路由日志

Text2SPL 实验（§9）中的英语查询生成了一段包含两个 CTE 的脚本，其中 Text2SPL 自行将明确的专用模型分配给每个 CTE。表11展示了 SPL-flowexecution 日志中的路由轨迹。

表11：MoM 路由轨迹：英语日部首查询。两个 CTE 并行分发；外层 PROMPT 综合结果。来源：logs/streamlit/streamlit-20260220-062710.log。

Step	模型	词元	延迟 (s)	领域	
cjk_analysis	qwen2.5	674	39.1	中文	
german_translations	mistral	285	14.2	德语	
chinese_sun_radicals	claude-sonnet-4-5	1,086	55.8	推理 综合	/
总计		2,045	55.8*		

*墙钟时间：CTE 并行运行（14.2s + 39.1s → 39.1s 墙钟时间），综合按顺序执行。

关键日志摘录。

```
06:41:22 [chinese_sun_radicals] dispatching 2 CTE(s) in parallel: cjk_analysis,
 german_translations06:41:22 [CTE:cjk_analysis] starting model=qwen2.5
06:41:22 [CTE:german_translations] starting model=mistral
06:41:36 [CTE:german_translations] done tokens=285 latency=14180ms
06:42:01 [CTE:cjk_analysis] done tokens=674 latency=39110ms
06:42:18 [chinese_sun_radicals] LLM response model=claude-sonnet-4-5
tokens=756+330=1086 latency=55848ms
```

两个 CTE 在同一时间戳（06:41:22）被派发，并在相隔 25 秒后完成，证实通过 asyncio.gather 实现了真正的并行执行。路由分配（qwen2.5 用于中日韩， mistral 用于德语）是根据 GENERATE 子句和 system_role 字符串中的关键词信号推断得出的，没有额外的大语言模型调用。

跨适配器路由（Text2SPL 实验）。在 Text2SPL 实验（表 9）中，单提示词查询会按适配器自动路由：中文和阿拉伯语查询使用 qwen3 （Ollama，中日韩/多语言专用）；三种语言的查询均使用 claude-opus-4.6 （OpenRouter），这反映了在未配置特定领域覆盖设置时，OpenRouter 默认采用可用的最佳路由。

C 逻辑分块：脚本与综合输出

C.1 实验元数据。

- **文档：**《*Attention Is All You Need*》（Vaswani 等，2017年） [Vaswani 等， 2017]。
- **适配器：**OpenRouter。
- **映射：** anthropic/claude-haiku-4.5， $k = 4$ 个 CTE 并行分发；墙钟时间为 5.2 秒（顺序执行为 ≈20.8 秒）。
- **归约：** anthropic/claude-sonnet-4-5，1,200 词元输出预算。
- **词元：** 1,914（映射） + 2，835（归约） = 4，总计 749。
- **成本：** ≈$0.025。

C.2 SPL 脚本（映射-归约）。

```
-- chunking_transformer.spl
-- Map-Reduce review of "Attention Is All You Need" (Vaswani et al., 2017)
-- Map:    4 CTEs summarise each section in parallel (asyncio.gather)
-- Reduce: 1 PROMPT synthesises a structured critical review

PROMPT transformer_review
WITH BUDGET 16000 tokens
USING MODEL "anthropic/claude-sonnet-4-5"

WITH intro_summary AS (
    PROMPT summarise_intro
    WITH BUDGET 3000 tokens
    USING MODEL "anthropic/claude-haiku-4.5"
    SELECT system_role("You are an expert NLP researcher. Be concise.")
    GENERATE answer("Summarise the Abstract and Introduction of
      'Attention Is All You Need': core thesis, RNN/LSTM limitations
      addressed, key claims about quality and parallelism.")
    WITH OUTPUT BUDGET 350 tokens, FORMAT markdown
),
architecture_summary AS (
    PROMPT summarise_architecture
    WITH BUDGET 3000 tokens
    USING MODEL "anthropic/claude-haiku-4.5"
    SELECT system_role("You are an expert NLP researcher. Be concise.")
    GENERATE answer("Summarise the Model Architecture: encoder-decoder
      stack, multi-head attention, positional encoding, feed-forward
      sub-layers, residual connections.")
    WITH OUTPUT BUDGET 350 tokens, FORMAT markdown
),
training_summary AS (
    PROMPT summarise_training
    WITH BUDGET 3000 tokens
    USING MODEL "anthropic/claude-haiku-4.5"
    SELECT system_role("You are an expert NLP researcher. Be concise.")
    GENERATE answer("Summarise Training: WMT datasets, 8xP100 hardware,
      Adam with warmup schedule, dropout and label smoothing.")
    WITH OUTPUT BUDGET 350 tokens, FORMAT markdown
),
results_summary AS (
    PROMPT summarise_results
    WITH BUDGET 3000 tokens
    USING MODEL "anthropic/claude-haiku-4.5"
    SELECT system_role("You are an expert NLP researcher. Be concise.")
    GENERATE answer("Summarise Results: BLEU scores on WMT EN-DE/EN-FR,
      FLOPs comparison, ablation findings, broader applicability.")
    WITH OUTPUT BUDGET 350 tokens, FORMAT markdown
)

SELECT
    system_role("You are a senior AI researcher writing a structured
      critical review. Be analytical, not merely descriptive."),
    context.intro_summary        AS intro        LIMIT 500 tokens,
    context.architecture_summary AS architecture LIMIT 500 tokens,
    context.training_summary     AS training     LIMIT 500 tokens,
    context.results_summary      AS results      LIMIT 500 tokens
GENERATE
```

```
    answer("Write a structured critical review under five headings:
      1. Core Contribution, 2. Key Design Decisions, 3. Empirical
      Evidence, 4. Limitations and Open Questions, 5. Lasting Impact.")
WITH OUTPUT BUDGET 1200 tokens, FORMAT markdown;
```

列表 12：逻辑分块脚本：对《*Attention Is All You Need*》的映射-归约评审（4 个并行 Map 公用表表达式，1 个归约）。

C.3 综合输出（Reduce 阶段）。

以下　　评论由 Reduce步骤 在 1,200 词元的输出预算内生成　　.

1. 核心贡献。论文的根本创新在于证明，最先进的序列建模并不需要循环。Transformer 以纯自注意力取代顺序执行的 RNN/LSTM 处理，消除了串行瓶颈：自注意力在任意两个位置之间提供 $O(1)$深度的信息路径（而 RNN 为 $O(n)$），同时实现训练期间的完全并行化。作者将序列建模重新定义为同时学习所有位置对之间的关系——这一范式转变使表示能力与序列归纳偏置脱钩。

2. 关键设计决策。有三项选择尤为重要。缩放点积注意力：$1/\sqrt{d_k}$ 缩放因子可防止点积幅值过大，使 softmax 远离梯度消失区域，并支持大规模稳定训练。多头注意力：投影到 $h{=}8$ 个并行的 64 维子空间，使不同注意力头能够专门处理不同类型的关系；移除注意力头会导致 BLEU 降低 0.6。位置编码：正弦位置编码注入位置信息，同时避免重新引入顺序计算；不过，后续研究（可学习嵌入、相对位置表示）重新审视了这一架构中理论依据最薄弱的部分。

3. 实证证据。结果表现出色，但适用范围较窄。在 WMT 英德任务上，BLEU 提升幅度显著（$+2.4$），在英法任务上相较于此前的最先进水平也有所提升（$+1.8$）；训练效率同样令人瞩目（仅需 25% 的浮点运算次数即可达到相当的质量）。但仍存在关键不足：评估仅限于机器翻译；消融实验未能充分检验许多设计选择；而模型在长程依赖上的优势只是被声称，并未得到直接测量。

4. 局限性与开放问题。二次复杂度：自注意力机制随序列长度呈 $O(n^2)$ 增长，使其难以用于长文档——论文承认了这一局限，却未予以解决，由此催生了“高效Transformer”这一研究分支（Linformer、Performer 等）。数据需求：舍弃循环机制也意味着舍弃序列归纳偏置；模型必须纯粹从数据中学习局部性与组合性，这意味着在低资源场景中样本复杂度较高。可解释性：后续的 BERTology 研究发现，注意力图往往存在冗余，且难以将其解读为推理的直接证据。

5. 持久影响。Transformer 已成为现代深度学习的通用基底，其影响远远超出了机器翻译。它为 BERT、GPT 和 T5 奠定了架构基础；通过将任何领域的表征标准化为一组相互作用的词元，它还推动了计算机视觉（ViT）、蛋白质结构预测（AlphaFold）和机器人学领域的突破。回过头看，这篇论文不仅改进了翻译，还提供了一种用于神经处理的通用计算语言，八年后的今天仍占据主导地位。

D SPL-flow：多模型编排脚本与输出 ut

D.1 生成的SPL 脚本

以下脚本由 Text2SPLNode (适配器： ollama，模型： qwen2.5) 根据英语日部首多语言表查询生成。外层 PROMPT 使用 "auto" 模型选择；两个 CTE 明确请求 qwen2.5 （中日韩文字专家）和 mistral （欧洲语言专家），对于不支持这些名称的适配器，系统会在执行时自动对其进行清理。

```
PROMPT chinese_sun_radicals
WITH BUDGET 8000 tokens
USING MODEL "auto"

WITH cjk_analysis AS (
    PROMPT chinese_data
    WITH BUDGET 2500 tokens
    USING MODEL "qwen2.5"

    SELECT
        system_role("You are a Chinese linguistics expert specializing in
            character etymology, radicals, and decomposition."),
        context.topic AS topic LIMIT 50 tokens

    GENERATE
        chinese_characters(topic, "List exactly 10 Chinese characters
        containing the sun/day radical. For each character provide: the
        character itself, decomposition formula showing how radicals
           combine,
        pinyin pronunciation, English meaning, and Chinese explanation.
        Output as JSON array with keys: character, decomposition, pinyin,
        english_meaning, chinese_explanation.")
    WITH OUTPUT BUDGET 800 tokens, TEMPERATURE 0.1, FORMAT json
),

german_translations AS (
    PROMPT german_data
    WITH BUDGET 2000 tokens
    USING MODEL "mistral"

    SELECT
        system_role("You are a professional German translator specializing
            in Chinese language concepts."),
        context.topic AS topic LIMIT 50 tokens

    GENERATE
        translate_german(topic, "For 10 common Chinese characters with
        sun/day radical, provide accurate German translations.
        Output as JSON array with keys: character, german_meaning.")
    WITH OUTPUT BUDGET 600 tokens, TEMPERATURE 0.1, FORMAT json
)

SELECT
    system_role("You are an expert at creating structured multilingual
        reference tables from JSON data with cultural insights."),
    context.cjk_analysis AS chinese_data LIMIT 2000 tokens,
    context.german_translations AS german_data LIMIT 1500 tokens
```

```
GENERATE
   compose_table(chinese_data, german_data, "Merge the Chinese character
```

清单13：Text2SPL生成的脚本：中文 日部首多语言表格（ollama）

D.2 跨适配器执行结果

通过两个适配器执行了相同的三个查询（英语、中文、阿拉伯语）。不兼容适配器的模型名称（"qwen2.5"、 "mistral"）会在执行时替换为 "auto" ，并由 MoM 路由器针对各适配器解析。所有 Ollama 运行均未产生 API 费用；OpenRouter 总费用为 ≈$0.03。

表12：跨适配器 Text2SPL 结果。英语使用双 CTE 并行流水线；中文和阿拉伯语使用单提示词。Ollama：本地推理，$0.00。OpenRouter：云推理，≈$0.03。

适配器	查询	模型	词元	延迟 (s)
Ollama（本地）	英语†	gemma3	1,653	35.4
	中文	qwen3	2,925	82.0
	阿拉伯语	qwen3	3,933	110.4
	总计		**8,511**	110.4
OpenRouter (云端)	英语†	claude-opus-4.6	1,974	30.5
	中文	claude-opus-4.6	4,434	65.5
	阿拉伯语	claude-opus-4.6	4,423	68.4
	总计		**10,831**	68.4

†英语 CTE 并行运行： qwen2.5 （中日韩） + mistral （德语）在 Ollama 上运行；两者均自动路由至qwen/qwen-2.5-7b-instruct OpenRouter。总计不包含 CTE令牌。

D.3 EnglishQuery输出（Ollama）

表13展示了多CTE Ollama运行的综合结果。两个 CTE（qwen2.5用于中日韩数据， mistral 用于德语翻译）并行分派； gemma3 在两者都完成后汇总出最终表格。

表13：多 CTE 输出：中文日部首多语言表（Ollama）。CTE： qwen2.5+ mistral 并行。综合：gemma3。墙钟时间：35.4 秒。成本：$0.00。

Char	拆分	拼音	英语	中文解释	德语	自然语义
日	日	rì	太阳；白天	表示太阳或白天的意思。	Tag	对中文至关重要 计时； 太阳的 在中文文 化中的核心地位。
明	日 +月	míng	明亮；清晰	表示明亮的意思。	明亮，清晰	太阳 +月亮 =明亮 度：对立 力量在中文哲 学中的相互作用。
晚	日 +免	wǎn	傍晚；深夜	表示傍晚的意思。	Abend	太阳被移除（免），暗示 夜间。
旦	日 +一	dàn	黎明；破晓	表示早晨的意思。	黄昏	地平线上的太阳 线（一）： 生动的 图标- 日出图。
晶	日 + 日 + 日	jīng	水晶；明亮	表示透明或明亮的意思。	水晶	唤起纯洁与清澈之感—— ity； 用于 珍贵的 宝石和清澈的水。

E 基准测试：FullModel 对比报告

以下实验使用 SPL-flow 的 benchmark CLI 命令，针对第 9 节中的中文 日部首多语言表格脚本运行（与第 9.2 节 MoM 路由实验及附录 D 中使用的脚本相同）。通过 Ollama 适配器在本地评估了七个模型；所有运行均成功完成，API 费用为零。

清单 14：SPL-flow BENCHMARK CLI 调用（Ollama，七个模型）

python3 –m s r c . c l i 基准测试 \ s p l −9.2–A–c l a u d e _ c l i −20260220−063838. s p l \—模型”qwen2 . 5，qwen3，m i s t r a l，l l a m a 3 . 1，d e e p s e e k –r1，gemma3，p h i 4”\—适配器 o l l a m a \—输 出 基准测试–ollama −20260221−v2 . j s o n

表 14：七模型 BENCHMARK 运行：中文日部首表任务。适配器：Ollama（本地）。成本：所有模型均为 $0.00（无 API 费用）。 ⋆ 标记每列的优胜者。

模型	输入词元	输出词元	总计词元	延迟 (s)	备注
mistral	955	576	1,531	**44.4**⋆	最快；直接输出表格
phi4	832	521	**1,353**⋆	88.3	词元效率最高
qwen2.5	939	724	1,663	76.0	中日韩质量出色；均衡
llama3.1	1097	868	1,965	97.5	包含导语句
gemma3	1200	884	2,084	111.5	将输出置于代码围栏中
deepseek-r1	1121	1,146	2,267	105.4	思维链 <think> 轨迹使计数增加
qwen3	1219	3,000	4,219	177.7	扩展思维链；最详尽

解读。该表展示了声明式模型比较的核心主张：同一份 SPL 脚本原封不动地提交给七个模型，系统便会在一份结构化报告中呈现所有成本与质量信号。添加或移除候选模型都无需更改应用程序代码。

有三种模式尤为突出。首先，输出冗长度是设计选择，而非质量代理指标：qwen3 和 deepseek-r1 会输出内部思维链推理（<think> 个区块），占其输出令牌预算的

2–3×，但在呈现最终答案前会将其剥离；在 $4\times$ 的词元成本下，它们的答案质量与 qwen2.5 相当。其次，延迟与输出量相关（七个模型的皮尔逊 $r \approx 0.97$；我们注意到样本量仅为 $n{=}7$，因此将其视为说明性趋势，而非具有统计定论的结论），这证实了在相同硬件条件下，词元效率高的模型也是最快的。第三，领域专门化无需额外边际成本： qwen2.5更强的中日韩训练带来了比 mistral 更丰富的中文解释，尽管两者延迟相近—— BENCHMARK 报告让这一权衡一目了然，无需手动检查七个模型各自的输出。

优胜者选择。运行获胜者命令:

```
python3 -m src.cli winner benchmark-ollama-20260221-v2.json
```

结果为：**mistral**（最快，44.4 秒），**phi4**（词元效率最高，1,353 个词元），**qwen2.5**（最适合中日韩密集型任务的均衡选择）。获胜模型标识符可通过 STORE RESULT IN memory.best_model 写回 USINGMODEL 子句，因此下一次执行会自动使用经实测选出的模型。

F FullSPL 语法（扩展巴科斯-瑙尔范式）

本附录包含 SPL v1.0 的完整形式文法。该语法是上下文无关的，可由手写递归下降解析器按 LL(1) 方式解析。关键字不区分大小写。

清单 15：SPL 形式文法——完整的扩展巴科斯-瑙尔范式规范（v1.0）

```
(* SPL - Structured Prompt Language *)
(* Formal Grammar Specification v1.0 *)
(* Date: February 12, 2026 *)

program          = statement (";" statement)* ";"? EOF ;

statement        = prompt_stmt
                 | create_func_stmt
                 | explain_stmt
                 | execute_stmt ;

(* === PROMPT Statement === *)

prompt_stmt      = "PROMPT" IDENT
                   header_clauses
                   cte_block?
                   select_clause
                   where_clause?
                   order_clause?
                   generate_clause
                   store_clause? ;

header_clauses   = ("WITH" "BUDGET" INTEGER "TOKENS")?
                   ("USING" "MODEL" (IDENT | STRING))?
                   ("CACHE" "FOR" INTEGER IDENT)? ;

(* === Common Table Expressions === *)

cte_block        = "WITH" cte_def ("," cte_def)* ;
cte_def          = IDENT "AS" "(" cte_body ")" ;
cte_body         = prompt_stmt
                 | ( select_clause
                     from_clause?
```

```
                    where_clause?
                    ("LIMIT" INTEGER "TOKENS")? ) ;

(* === SELECT Clause === *)

select_clause    = "SELECT" select_item ("," select_item)* ;
select_item      = source_expr ("AS" IDENT)?
                   ("LIMIT" INTEGER "TOKENS")? ;

source_expr      = system_role_call
                 | context_ref
                 | rag_call
                 | memory_call
                 | function_call
                 | IDENT ;

system_role_call = "system_role" "(" STRING ")" ;
context_ref      = "context" "." IDENT ;
rag_call         = "rag" "." "query" "(" STRING
                   ("," "top_k" "=" INTEGER)? ")" ;
memory_call      = "memory" "." "get" "(" STRING ")" ;
function_call    = IDENT "(" arg_list? ")" ;
arg_list         = expression ("," expression)* ;

(* === FROM Clause === *)

from_clause      = "FROM" source_expr ("AS" IDENT)? ;

(* === WHERE Clause === *)

where_clause     = "WHERE" condition (("AND" | "OR") condition)* ;
condition        = expression comparator expression
                 | expression "IN" "(" expression_list ")" ;
comparator       = "=" | "!=" | ">" | "<" | ">=" | "<=" ;
expression_list  = expression ("," expression)* ;

(* === ORDER BY Clause === *)

order_clause     = "ORDER" "BY" order_item ("," order_item)* ;
order_item       = expression ("ASC" | "DESC")? ;

(* === GENERATE Clause === *)

generate_clause  = "GENERATE" IDENT "(" arg_list? ")"
                   generate_opts? ;
generate_opts    = "WITH" generate_opt ("," generate_opt)* ;
generate_opt     = "OUTPUT" "BUDGET" INTEGER "TOKENS"
                 | "TEMPERATURE" FLOAT
                 | "FORMAT" IDENT ;

(* === STORE Clause === *)

store_clause     = "STORE" "RESULT" "IN" "memory" "." IDENT ;

(* === CREATE FUNCTION === *)

create_func_stmt = "CREATE" "FUNCTION" IDENT
                   "(" param_list? ")"
```

```
                    "RETURNS" IDENT
                    "AS" "$$" func_body "$$" ;
param_list        = param_def ("," param_def)* ;
param_def         = IDENT IDENT? ;
func_body         = { any character except "$$" } ;

(* === EXPLAIN === *)

explain_stmt      = "EXPLAIN" "PROMPT" (IDENT | inline_prompt) ;
inline_prompt     = prompt_stmt ;

(* === EXECUTE === *)

execute_stmt      = "EXECUTE" "PROMPT" IDENT
                    ("WITH" "PARAMS" "(" param_assignments ")")? ;
param_assignments = param_assign ("," param_assign)* ;
param_assign      = qualified_name "=" expression ;
qualified_name    = IDENT ("." IDENT)* ;

(* === Expressions === *)

expression        = primary_expr (("+" | "-") primary_expr)* ;
primary_expr      = IDENT ("." IDENT)*
                  | STRING
                  | INTEGER
                  | FLOAT
                  | function_call
                  | "@" IDENT ;

(* === Terminals === *)

IDENT             = letter (letter | digit | "_")* ;
INTEGER           = digit+ ;
FLOAT             = digit+ "." digit+ ;
STRING            = '"' { any char except '"' } '"'
                  | "'" { any char except "'" } "'" ;
letter            = "a".."z" | "A".."Z" | "_" ;
digit             = "0".."9" ;

(* === Comments === *)

comment           = "--" { any char except newline } newline ;
```