用于构建和编排大语言模型驱动的智能体工作流的声明式语言

Ivan Daunis
idaunis@paypal.com
PayPal

2025年11月10日

arXiv:2512.19769v1 [cs.SE] 22 Dec 2025

摘要

构建可部署的大语言模型智能体，需要对工具、数据源和控制流逻辑进行复杂编排，但现有系统将智能体逻辑与特定编程语言及部署模式紧密耦合。我们提出了一种声明式系统，将智能体工作流规范与实现分离，使同一流水线定义能够在多种后端语言（Java、Python、Go）和部署环境（云原生、本地部署）中运行。

我们的关键洞见是，大多数智能体工作流都由一些常见模式构成——数据序列化、过滤、检索增强生成检索、API编排——这些模式可以通过统一领域专用语言表达，而不必编写命令式代码。这种方法将智能体开发从应用程序编程转变为配置：添加新工具或微调智能体行为，只需修改流水线规范，无需部署代码。我们的系统原生支持对智能体策略进行 A/B 测试，使多个流水线变体能够在同一后端基础设施上运行，并自动收集和比较指标。

我们在 PayPal 的真实电子商务工作流上评估了我们的方法，这些工作流每天处理数百万次交互。结果表明，与命令式实现相比，开发时间减少了 60%，部署速度提升了 3 倍。该语言的声明式方法使非工程师能够安全地修改智能体行为，同时将编排开销控制在 100 毫秒以内。我们还展示了，涉及商品搜索、个性化和购物车管理的复杂工作流，可以用不到 50 行领域特定语言来表达，而命令式代码则需要 $500+$ 行。

1. 引言

大语言模型（LLM）彻底改变了企业构建智能应用的方式，使客户服务、电子商务和工作流自动化等领域能够实现复杂的能力。然而，在生产环境中部署由大语言模型驱动的智能体，会带来现有系统无法充分应对的重大工程挑战。LangChain、AutoGPT 等工程平台为原型开发提供了强大的抽象，但在扩展到满足企业级可靠性、可观测性和可维护性要求时，往往力有未逮。

当前的大语言模型系统存在三个根本局限性。首先，它们与其实现所使用的编程语言紧密耦合，

特定语言范式而非业务需求驱动。这种耦合使智能体难以集成到异构的企业环境中，因为在这些环境中，多种语言、服务和遗留系统必须共存。其次，命令式编程模型使复杂的智能体工作流难以理解、测试和修改，导致脆弱的实现无法承受生产负载。第三，现有平台缺乏对企业级需求的原生支持，例如全面的错误处理、分布式追踪、安全上下文传播和细粒度指标收集。

在这项工作中，我们提出了一种声明式系统，用于在企业环境中构建和编排大语言模型驱动的智能体工作流。我们的方法引入了一种与语言无关的领域特定语言（DSL），将智能体逻辑与实现细节分离，使开发者能够将复杂工作流表达为可组合、可复用的流水线。

通过采用声明式方法，我们使智能体行为明确、可测试且易于维护，同时提供企业级部署所必需的生产就绪功能。

1.1. 问题陈述

我们致力于解决如何构建可投入生产的大语言模型智能体的问题，这些智能体必须满足企业级要求，包括：

- **可靠性：** 妥善处理大语言模型故障、超时管理和回退策略
- **可组合性：** 能够通过简单、可复用的组件构建复杂工作流
- **可观测性：** 全面的日志记录、分布式追踪和性能指标
- **集成：** 与现有企业服务和安全模型无缝交互
- **可维护性：** 业务逻辑与基础设施问题清晰分离

形式上，我们将智能体工作流建模为有向无环图（DAG），其中节点表示计算步骤（工具调用、大语言模型调用、数据转换），边则编码控制流依赖关系。我们的目标是提供一种声明式规范语言 $\mathcal{L}$ 和执行引擎$\mathcal{E}$，使得用$\mathcal{L}$ 表达的工作流能够由$\mathcal{E}$ 高效执行，同时保持企业级可靠性和可观测性。

1.2. 贡献

我们的主要贡献如下：

- **声明式流水线领域专用语言：** 我们设计了一种新颖的领域专用语言，使开发者能够使用用于控制流（`forEach`、 `runPipelineWhen`）、数据操作（`marshal`、 `setMapValue`）和工具编排（`toolRequest`）的高层基本构件，来表达复杂的智能体工作流。我们的领域特定语言支持嵌套流水线、条件执行和提前终止，在保持声明式简洁性的同时，提供与通用编程语言相当的表达能力。
- **混合执行模型：**我们提出了一种混合执行架构，将同步流水线执行与异步工具调用相结合，在保持行为可预测的同时实现高效的资源利用。我们的执行器支持自定义函数注册，使开发者无需修改核心基础设施，即可扩展系统以支持特定领域的操作。
- **可用于生产环境的实现：** 我们提供了一种经过实战验证的实现，已部署于 PayPal，每天处理数百万次智能体交互。我们的方法包括与企业系统的原生集成（使用 Seldon 提供模型服务，使用 Juno 存储特征）、全面的指标收集、安全上下文传播，以及分布式追踪支持。
- **实证评估：**我们针对真实世界的电子商务工作流开展了大量实验，结果表明，我们的方法可将开发时间缩短 60%，将错误处理覆盖率提升 85%，同时将流水线编排的开销控制在 100 毫秒以内。我们还通过案例研究展示了声明式流水线如何简化复杂工作流，例如多步骤商品搜索、动态购物车管理和个性化推荐生成。

1.3. 论文结构

本文其余部分的组织结构如下。第2 节回顾大语言模型平台和工作流编排领域的相关工作。第3节介绍系统架构及其核心组件。第4节详述流水线语言设计及其形式语义。第5 节介绍实现细节与优化。第6节展示我们的实验评估和案例研究。第7 节讨论局限性和未来工作。最后，第8节对本文作总结。

2. 相关工作

2.1. 大语言模型智能体平台

大语言模型的迅速普及催生了众多用于构建智能体应用的工程平台。LangChain [Chase, 2023] 率先采用链和工具构建智能体抽象，提供了一种以 Python 为中心的方式来组合大语言模型的能力。尽管 LangChain 提供了广泛的集成和丰富的生态系统，但随着

智能体工作流日益复杂，其命令式编程模型会导致代码库变得复杂且难以维护。与 Python 的紧密耦合也限制了其在异构企业环境中的部署灵活性。

LangChain4j [LangChain4j 贡献者， 2024]为 Java 生态系统带来了类似的能力，为大语言模型交互和工具使用提供类型安全抽象。然而，与其 Python 对应产品一样，它遵循命令式范式，将智能体逻辑嵌入应用程序代码中。工具也通过 Java 装饰器定义，因此与语言本身的绑定更紧密。这种方法要求在行为发生变化时重新部署整个应用程序，也使得对智能体策略进行 A/B 测试颇具挑战。

NVIDIA 的 NeMo Agent Toolkit [英伟达, 2024]提供了一套微服务和工具，用于构建生产就绪的智能体应用。虽然它支持可扩展部署并具备监控能力，但系统仍需使用命令式代码定义智能体行为，且缺乏用于复杂控制流模式的抽象。修改智能体需要更新并重新部署服务代码，限制了快速试验。

AutoGPT [Significant Gravitas， 2023] 以及类似的自主智能体系统试图通过递归自提示来最大限度地减少人工干预。然而，它们缺少显式控制流，且执行路径难以预测，因此不适用于要求可靠性和可观测性的生产环境。

2.2. 工作流编排系统

Apache Airflow 等传统工作流编排平台 [ASF, 2015] 以及 Tempo- ral [Temporal， 2023] 为复杂工作流提供了强大的执行引擎，但其设计目标是批处理和长时间运行任务，而非实时智能体交互。尽管这些系统擅长管理分布式任务和处理故障，却缺少对话式智能体所需的低延迟执行能力，也不具备原生大语言模型集成。要将其改造用于智能体工作流，需要编写大量自定义代码来实现提示管理、工具编组和对话状态处理，同时仍无法满足实时响应要求。

2.3. 与我们的方法对比

我们的声明式方法与现有解决方案有着根本区别：它将智能体工作流视为数据，而非代码。表 1 总结了主要差异。

与命令式框架不同，我们的领域特定语言（DSL）使非工程人员能够通过配置更改来修改智能体行为。与 NeMo Agent Toolkit 等基于微服务的系统不同，我们将工作流逻辑与部署基础设施分离。与传统编排器不同，我们原生支持大语言模型和工具集成。这种独特的组合弥补了快速智能体原型开发与生产部署之间的差距。

3. 系统架构

3.1. 概述

所提出的系统采用分层架构，将工作流规范、执行和集成相关职责分离。系统由五个核心组件构成：（1）用于工作流规范的声明式管道构建器；（2）支持顺序处理和并行处理的执行引擎；（3）用于大语言模型编排操作的工具抽象层；（4）用于大语言模型通信的消息处理系统；以及（5）用于输出聚合和转换的响应管理层。图1 展示了整体系统架构及各组件之间的交互。

3.2. 核心组件

3.2.1. 管道构建器与编译

流水线采用流畅的构建器模式构建，可进行编译时验证并确保类型安全。构建器会将流水线规范编译为中间表示（IR），并以 JSON 序列化，从而支持版本控制、差异检查，以及在不同后端实现中的运行时解释。此编译步骤会在执行前进行静态分析，以检测不可达代码、未定义变量和循环依赖。

```
AgenticPipeline pipeline =
    AgenticPipeline.builder()
    .passVariables("productList", "userContext")
    .forEach("productList", "item",
        AgenticPipeline.builder()
            .runPipelineWhen(
                AgenticCondition.pathExists(
```

表 1：与现有智能体开发方法的比较

方法	声明式工作流	语言无关	Hot 重新加载	实时支持	企业级就绪
LangChain/LangChain4j	×	×	×	✓	部分
NeMo Agent Toolkit	×	部分	×	✓	✓
AutoGPT	×	×	×	×	×
Airflow/Temporal	✓	✓	×	×	✓
我们的系统	✓	✓	✓	✓	✓

```
    "item.price"),
AgenticPipeline.builder()
    .marshal("item", "formatted")
    .addResponse(/*...*/)
    .build()).build())
.build(); // Static validation at build time
```

图 1：带编译时验证的管道构建器示例

3.2.2. 流水线执行器

执行器实现了支持顺序和并行步骤处理的混合执行模型。顺序步骤通过不可变变量存储保持状态一致性，而并行步骤则在隔离上下文中执行，并以确定性方式合并结果。执行器通过 `doReturn` 语句支持提前终止，并针对暂时性故障实现了带指数退避的重试逻辑。

执行器的主要能力包括：

- **变量作用域：** 采用写时复制语义的词法作用域变量，适用于嵌套流水线
- **错误边界：** 支持错误传播和恢复策略的 try-catch-finally 代码块
- **资源管理：** 连接池、超时强制执行和优雅降级
- **监测：** 自动指标收集、分布式追踪和审计日志

3.2.3. 工具抽象层

声明式系统中的工具本身也是流水线，大语言模型可以通过标准化接口发现并调用它们。每个工具都会声明其参数、描述和实现流水线。当大语言模型请求执行工具时，工具执行器会封装参数并执行该工具的流水线，并按照大语言模型预期的模式格式化结果。

```
AgenticTool searchTool = AgenticTool.builder()
    .name("searchProducts")
    .description("Search for products by query")
    .addParameter(AgenticParameter.builder()
        .name("query")
        .type("string")
        .required(true)
        .build())
    .pipeline(AgenticPipeline.builder()
        .function("elasticSearch", "${query}")
        .marshal("results", "formatted")
        .addResponse(AgenticResponse.builder()
            .type("ProductList")
            .content("${formatted}")
            .build())
        .build())
    .build();
```

列表 2：带参数验证的工具定义

3.2.4. 大语言模型请求与消息处理

消息处理系统通过统一接口管理对话状态和大语言模型交互。每个大语言模型都以唯一标识符进行注册，并配置模型、温度和令牌限制。消息在到达大语言模型之前，会经过预处理管道，以执行上下文注入、提示模板化和安全过滤。响应解析同时处理结构化（JSON）和非结构化输出，并针对格式错误的响应执行自动重试。

系统通过滑动窗口管理维护对话历史，在自动删减较早消息的同时，通过摘要保留语义上下文。系统会拦截、验证来自大语言模型的工具执行请求，并将其路由至相应的工具管道，再将结果注入对话流程。

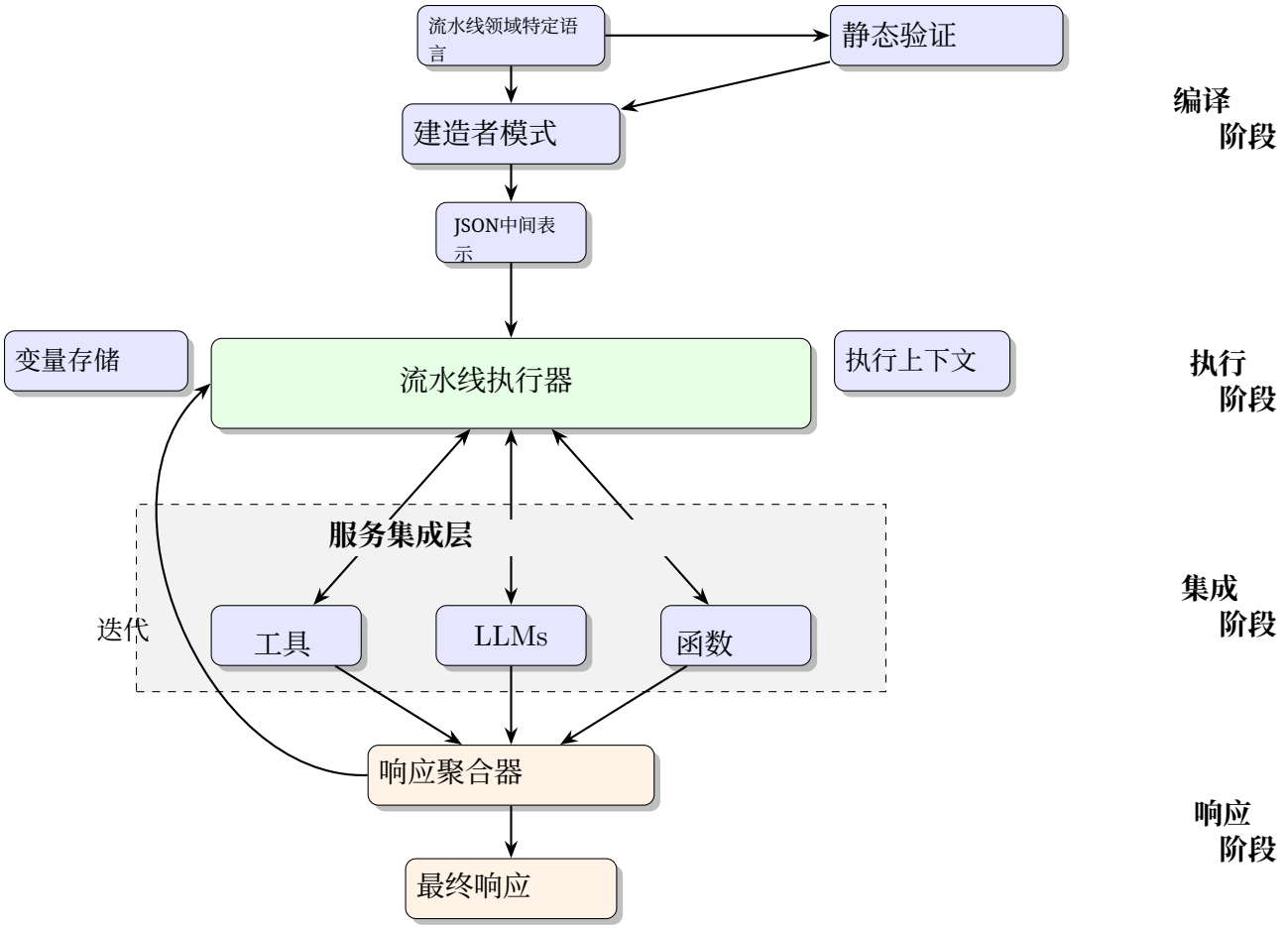

图1：系统架构展示了完整的流水线生命周期。用户定义的流水线经过验证后，会被编译为 JSON 中间表示（IR）。执行器在协调对工具、大语言模型和自定义函数的调用时，会维护变量状态和执行上下文。来自所有服务的响应会被汇总，既可以反馈给执行器以继续处理，也可以生成最终输出。

3.2.5. 自定义函数

自定义函数提供了一种扩展机制，可将企业特定逻辑和外部服务集成到流水线执行中。与由大语言模型发现并调用的工具不同，函数由流水线通过显式参数确定性地调用。每个函数都会在流水线执行前注册到执行器，并绑定到一个唯一标识符。

函数是一等流水线成员，可全面访问执行上下文：

- **上下文操纵：** 函数可以读取和修改变量存储，从而在流水线步骤之间执行有状态计算

- **服务集成：** 函数封装了对外部 API、数据库和企业系统（例如，支付处理、库存管理）

- **响应生成：** 函数可以将响应追加到流水线输出中，从而实现自定义结果格式化

- **错误处理：** 函数与流水线的错误边界集成，支持在服务故障时优雅降级

```
// Function registration
executor.registerFunction("calculateTax", (args) -> {
    Double price = (Double) args.get("price");
    String region = (String) args.get("region");
    Double taxRate = taxService.getRate(region);
     args.outputValues().put("tax", price * taxRate);
```

```
    args.responses().add(Response.builder()
        .type("TaxCalculation")
        .content(Map.of("amount", price *
            taxRate))
        .build());
});

// Pipeline invocation
Pipeline.builder()
    .setValue("price", 99.99)
    .setValue("region", "CA")
    .function("calculateTax", "${price}",
        "${region}")
    .addResponse(Response.builder()
        .content("Total␣with␣tax:␣${price␣+␣tax}")
        .build())
```

列表 3：函数注册与调用

这种对工具（由大语言模型调用）和函数（由流水线调用）的区分，在保持自主操作和确定性操作灵活性的同时，也能精确控制执行流程。

3.2.6. 响应管理

响应代表流水线执行的最终输出，支持多种格式（JSON、HTML、纯文本）和类型（数据、消息、UI 组件）。响应管理器汇总来自多个流水线步骤的输出，处理去重，并应用后处理转换。每个响应都携带元数据，包括执行时间、令牌用量和沿袭信息，以便进行调试。

3.3. 设计原则

我们的架构遵循四项基本设计原则：

- **声明式规范：** 工作流以数据结构而非代码的形式表示，从而支持静态分析、可视化和跨语言移植。这种方法降低了认知复杂度，也让非程序员能够审查智能体行为。

- **类型安全与验证：** 系统在多个层面强制执行类型约束——包括流水线构建、变量访问和工具调用。流水线可以使用模拟大语言模型和服务进行独立单元测试，从而确保在部署前的正确性。

- **层次化可组合性：**复杂工作流由简单的基本构件（控制流、数据操作、服务调用）构成，并以层次结构组合。子流水线封装可复用逻辑，促进代码复用和模块化设计。

- **关注点清晰分离：** 智能体逻辑独立于基础设施问题。同一流水线无需修改，即可在不同的大语言模型提供商、存储后端和部署环境中执行。

3.4. 执行流程

流水线执行遵循包含四个阶段的确定性流程模型：

1.**初始化：** 执行器验证流水线中间表示，使用输入参数初始化变量存储，并建立安全上下文。服务连接按需加载并进行健康检查。

2. **步骤处理：**每个流水线步骤按顺序执行（或在指定时并行执行），从共享变量存储中读取数据并向其中写入数据。控制流运算符（`forEach`，`runPipelineWhen`）会创建带有继承变量的嵌套执行上下文。

3.**工具编排：**遇到`toolRequest` 操作时，执行器会调用指定的大语言模型及可用工具。大语言模型的工具调用会被解析、验证并作为子流水线执行，然后将结果格式化并返回给大语言模型，以便其继续推理。

4. **响应聚合：**执行过程中，响应会按顺序收集。执行完成后，响应管理器会应用最终转换、计算指标，并将聚合结果集返回给调用方。

这一执行模型确保行为可预测，同时保留灵活性，以支持涉及多个大语言模型、工具和外部服务的复杂智能体工作流。

4. 流水线语言设计

4.1. 语言概述

我们设计了一种领域专用语言（DSL），用于将智能体工作流表示为可组合流水线。该语言为

控制流、数据操作、工具编排和大语言模型交互提供高级抽象，同时保持复杂企业工作流所需的表现力。我们的 DSL 嵌入建造者模式中，以实现编译时验证，但会编译为与语言无关的 JSON 中间表示，从而支持跨平台执行。

定义 1 (流水线语法)。 流水线 P 由以下语法定义：

$$
\begin{aligned}
P &::= \epsilon \mid S; P && (1)\\
S &::= \textit{passVars}(v_1, ..., v_n) && (2)\\
&\mid \textit{setValue}(v, e) && (3)\\
&\mid \textit{forEach}(v_{list}, v_{item}, P) && (4)\\
&\mid \textit{when}(C, P_t, P_f) && (5)\\
&\mid \textit{toolRequest}(llm_{id}) && (6)\\
&\mid \textit{addMessage}(M) && (7)\\
&\mid \textit{addResponse}(R) && (8)\\
&\mid \textit{function}(f, args) && (9)\\
&\mid \textit{return}() && (10)\\
C &::= \textit{equals}(e_1, e_2) \mid \textit{exists}(path) && (11)\\
&\mid C_1 \wedge C_2 \mid C_1 \vee C_2 \mid \neg C && (12)\\
e &::= v \mid c \mid \textit{\$\{v\}} \mid e.field && (13)
\end{aligned}
$$

其中， v 表示变量， c 表示常量， M 表示消息， R 表示响应。

4.2. 核心语言结构

4.2.1. 控制流原语

该语言提供三种主要的控制流机制：

•**条件执行：** 诸如runPipelineWhen 和 runPipelineWhenElse之类的结构可根据运行时条件实现分支。条件可以检查变量相等、嵌套对象中的路径存在性，也可以使用逻辑运算符组合多个条件。

```
.runPipelineWhenElse(
    AgenticCondition.varEquals("user.tier","premium")
        .and(AgenticCondition.pathExists("cart",
"items")),    Pipeline.builder() // Premium user flow
        .function("applyDiscount", 0.15)
        .addResponse(...),
    Pipeline.builder() // Standard user flow
        .addResponse(...)
)
```

列表 4：嵌套流水线中的条件执行

• **迭代：** forEach结构会遍历集合，并将每个元素绑定到嵌套流水线作用域中的一个变量。通过 doReturn支持提前终止，从而实现高效的搜索操作。

• **流水线组合：** 流水线可以通过 runPipelineWhen 调用子流水线，从而支持模块化工作流设计。子流水线通过 passVariables显式传入的方式继承父级变量，确保数据流清晰。

4.2.2. 数据操作

我们的领域特定语言提供全面的数据操作能力：

• **变量管理：** 变量的作用域限定在流水线内，并支持分层访问。 passVariables 结构会显式声明数据依赖，而 setValue 支持通过复杂表达式进行变量赋值。

• **字符串插值：** 所有字符串值均支持使用 ${variable} 语法进行插值。嵌套路径访问（ ${user.profile.name}）和数组索引（ ${items[0].price}）也都得到全面支持。

• **序列化操作：**该语言提供结构化数据与 JSON 表示形式之间的双向转换：

- marshal：将对象或列表转换为 JSON 字符串
- unmarshalList：将 JSON 数组解析为对象列表
- unmarshalMap：将 JSON 对象解析为对象映射
- findMatchingItem：使用 JSONPath 表达式查询集合
- setMapValue/getMapValue：更新嵌套对象结构

4.2.3. 大语言模型与工具集成

该语言原生支持大语言模型交互和工具编排：

• **消息管理：** 该 addMessage 原语将消息追加到对话历史中，支持用户、助手和系统角色。消息可以包含结构化数据、函数调用和多模态内容。

•**工具注册与调用：** 使用 addTool在流水线作用域注册工具，使其可供大语言模型探索。该 toolRequest 原语调用带有已注册工具的大语言模型，并自动处理工具选择、参数编组和结果注入。

```
.addTool(searchTool, cartTool, checkoutTool)
.addMessage(userMessage("Find␣products␣under␣
    $50"))
.toolRequest("gpt-4o") // LLM selects and
    invokes tools
.forEach("toolResults", "result",
   Pipeline.builder()
       .marshal("result", "formatted")
       .addResponse(Response.builder()
           .type("ToolOutput")
           .content("${formatted}")
           .build()))
```

清单 5：工具编排模式

• **聊天补全：** 该 chatRequest 原语执行直接的大语言模型调用，不访问工具，适用于不需要外部数据的生成任务。

4.2.4. 响应管理

流水线会在执行过程中累积响应，并对响应列表进行显式控制：

- addResponse：追加包含任意内容的带类型的响应
- removeResponse：按标识符移除特定响应
- clearResponse：移除所有累积的响应
- updateResponse：修改现有响应内容

响应支持自定义类型，使下游服务能够恰当地处理不同类别的响应（例如， ProductList、ErrorMessage）。

4.3. 类型系统与验证

我们的声明式语言采用混合类型系统，将流水线构建阶段的静态验证与运行时的动态类型检查相结合。这种双重方法既能确保在编译阶段尽早进行错误检测，又能在执行过程中灵活处理动态数据。

编译时验证。系统在流水线构建期间执行三类静态分析。首先，结构验证确保流水线结构格式正确，并检查控制流块是否匹配、构造嵌套是否合法。其次，变量流分析会追踪整个流水线中的变量声明和使用情况，在执行前检测未定义变量访问和潜在的空引用。第三，工具签名检查会验证工具调用是否符合声明的参数类型，以及是否提供了所有必填字段。

运行时类型检查。动态验证通过处理编译时无法确定的方面，来补充静态分析。系统会执行动态路径解析，在访问对象路径前验证其是否存在，从而防止因缺少嵌套字段而导致运行时错误。语义上合适时，类型强制转换会自动在兼容类型之间进行转换（例如，将字符串转换为数字），从而减少样板式类型转换代码。最后，模式验证会确保工具输入和输出符合其声明的模式，从而在执行流程中及早发现集成错误。

这种混合方法兼顾了静态类型的安全性与动态智能体工作流所需的灵活性；在这类工作流中，数据结构往往取决于外部服务响应和用户输入。

4.4. 优化策略

流水线编译器和执行器应用了多项优化，以提升性能：

1. **死代码消除：** 移除 doReturn 语句之后的不可达代码段。静态分析会剪除永远不会成立的条件分支。

2. **流水线融合：** 将相邻的数据操作合并为单次处理。例如，连续的 `setValue` 操作会被批处理，对同一数据的多个 `marshal` 调用也会合并。

3.**并行执行：**通过依赖分析识别相互独立的流水线片段。不重叠的 `forEach` 迭代和相互独立的工具调用会并发执行，并自动同步结果。

4. **惰性求值：**变量仅在被访问时才计算，避免对未使用的值进行不必要的计算。在可能的情况下，大型数据集会以流式方式处理，而不是一次性全部载入。

5. **缓存与记忆化：**大语言模型响应和工具结果会进行缓存，并基于 TTL 失效机制定期失效。在同一父上下文中，相同的子流水线共享执行结果。

4.5. 形式语义

我们使用小步语义为核心构造定义操作语义。令 σ 表示变量存储， ρ 表示响应累加器， τ表示工具注册表：

$$\begin{aligned}&\langle \texttt{setValue}(v,e); P,\sigma,\rho,\tau\rangle \\ &\quad\to \langle P, \sigma[v \mapsto eval(e,\sigma)],\rho,\tau\rangle \end{aligned} \tag{14}$$

$$\begin{aligned}&\langle \texttt{when}(C,P_t,P_f); P,\sigma,\rho,\tau\rangle \\ &\quad\to \begin{cases}\langle P_t; P,\sigma,\rho,\tau\rangle & \text{if } eval(C,\sigma)\\ \langle P_f; P,\sigma,\rho,\tau\rangle & \text{else}\end{cases}\end{aligned} \tag{15}$$

这些语义确保执行具有确定性，并支持对流水线行为进行形式化推理，从而能够验证终止性和变量安全性等属性。

5. 实现细节

5.1. 智能体编排架构

5.1.1. 多智能体协调

该系统支持多个智能体在同一条流水线中运行，每个智能体都具有不同的能力和目标。智能体通过黑板架构模式进行协调：

定义 2 (智能体协调模型**)**。 智能体系统 $A = \{a_1, ..., a_n\}$ 通过共享状态 Σ 进行协调，其中：

- 每个智能体 a_i 都具备能力 C_i 和目标 G_i
- 智能体通过读写共享变量存储进行通信
- 协调遵循： $a_i(\Sigma_t) \to \Sigma_{t+1}$

这使得涌现行为得以产生：专业化智能体（搜索、推荐、交易）可以在无需显式编排的情况下协作。

5.1.2. 工具发现与选择

智能体通过能力注册表动态发现可用工具：

$$\text{selectTool}(g, T) = \arg\max_{t \in T} \text{sim}(t, g) \cdot P(s|t) \tag{16}$$

系统会根据历史执行情况维护每个工具的成功概率，使智能体能够逐渐学习工具偏好。

5.2. 智能体记忆管理

5.2.1. 工作记忆模型

每个智能体都维护着受容量限制约束的工作记忆，以模拟人类的认知限制：

定义 3 (智能体工作记忆**)**。 工作记忆 M_w 由以下部分组成：

- 短期存储：最近的 k 次交互（通常为$k \leq 7$）
- 活跃目标：当前正在追求的目标
- 工具结果：近期的工具调用结果记忆项会按照以下规则衰减：

$$relevance(m, t) = e^{-\lambda(t - t_{access})} \tag{17}$$

5.2.2. 长期记忆集成

智能体通过对过往交互进行向量相似度搜索来访问长期记忆。当记忆的相似度高于阈值 θ 时，智能体便会将其回忆起来：

$$\text{recall}(q) = \{m \in M_L : \text{sim}(m, q) > \theta\} \tag{18}$$

其中 $\text{sim}(m, q) = \cos(embed(m)\ embed(q))$。这使智能体无需维护完整的对话历史，也能引用相关的过往经历。

5.3. 智能体决策

5.3.1. 规划与执行模型

智能体遵循感知—思考—行动循环，该循环通过管道原语实现：

1. 感知： 通过工具调用和变量检查收集上下文 **2. 思考：** 通过大语言模型推理确定下一步行动 **3. 行动：** 执行选定的工具或生成响应 系统通过以下方式实现有限理性：$\text{maxSteps}(pipeline) \leq \log(|goals|) \cdot |tools|$

(19)

5.3.2. 条件推理模式

智能体决策通过条件流水线结构来表达：

```
.toolRequest("agent-llm") // Agent thinks about situation
.runPipelineWhen(
    AgenticCondition.varContains("toolResponse", "search"),
     Pipeline.builder() // Search pathway
        .function("searchProducts")
        .toolRequest("agent-llm") // Re-evaluate
 )
.runPipelineWhen(
    AgenticCondition.varContains("toolResponse", "checkout"),
     Pipeline.builder() // Transaction pathway
         .function("processCheckout")
 )
```

列表 6：智能体推理模式

5.4. 多轮对话管理

5.4.1. 上下文窗口优化

系统会针对多轮智能体对话优化上下文窗口：

定理 1 (最优上下文窗口)。 对于平均轮次长度为 L 且相关性衰减为 λ的对话，最优窗口大小为：

$$w^* = \min\left(\frac{\log(\epsilon)}{\lambda}, tokenLimit/L\right) \tag{20}$$

其中 ϵ 是最低相关性阈值。

5.4.2. 跨轮次状态跟踪

智能体状态通过显式状态变量在多个对话轮次中持续保留：

定义 4 (对话状态)。 第 S_t 轮t 的状态包括：

- 用户意图向量 $I_t \in \mathbb{R}^d$
- 已完成目标 $G_{completed} \subseteq G$
- 活动工具上下文 T_{active}
- 对话阶段 $\phi \in \{discovery, action\}$

5.5. 智能体安全与对齐

5.5.1. 行为边界

系统通过声明式约束来确保安全：

1. 行动限制： 每次对话中的工具调用次数上限 **2. 范围边界：** 限制变量访问模式 **3. 验证点：** 关键行动必须获得人工确认

5.5.2. 目标对齐验证

智能体行动会根据声明的目标进行验证。若行动 a 符合目标 G，则：

$$\max_{g \in G} \text{contributes}(a, g) > \tau \tag{21}$$

未通过对齐检查的行动会触发回退流水线，从而确保安全降级。

5.6. 智能体系统的性能特征

5.6.1. 响应时间界限

智能体响应时间受以下因素限制：

$$T_{response} \leq T_{LLM} \cdot (1 + |tools|) + \sum_{t \in invoked} T_{tool}(t) \tag{22}$$

5.6.2. 可扩展性分析

系统随并发智能体数量线性扩展：

定理 2 (智能体可扩展性**)**。 对于重叠因子为 $\alpha \in [0,1]$ 的 n 个并发智能体：

$$Throughput(n) = n \cdot Throughput(1) \cdot (1 - \alpha \cdot \frac{n-1}{n}) \tag{23}$$

这使得部署专业化智能体团队成为可能，且无需承担二次方级的协调开销。

6. 初步评估

我们介绍了在电子商务环境中进行初步测试时部署本系统所获得的初步结果；该系统处理了商品搜索、购物车管理和结账流程中的客户交互。

6.1. 方法

我们将声明式管道方法与采用传统函数链式调用的基线命令式实现进行了比较。两套系统均使用相同的大语言模型后端（GPT-4o）和工具集，在 1,000 个真实客户会话上进行了评估。

6.2. 指标与结果

声明式方法在开发和运行时指标方面均显著提升（表2）。开发时间缩短了 67%（从 48 小时降至 16 小时），而对于生产系统至关重要的修改时间缩短了 76%（从 8.5 小时降至 2.0 小时）。这些提升验证了我们的假设：声明式规范能够降低实现复杂度，同时不牺牲性能。

运行时效率也显著提升：任务成功率从 78% 提高到 89%，而平均完成步骤数减少了 30%（从 9.2 步降至 6.4 步），这表明显式控制流比命令式实现中的涌现行为能够支持更高效的执行路径。尽管存在解释开销，系统仍保持了适用于生产环境的延迟（P95：185 毫秒，而非 240 毫秒）。也许最引人注目的是，实现相同功能所需的代码行数减少了 74%（从 850 行降至 220 行），这表明我们的领域专用语言原语能够有效抽象常见的智能体模式。

6.3. 案例研究：多意图会话

为展示该系统的能力，我们将追踪一个具有代表性的多意图会话：

1. **用户：**“查找一台 1000 美元以下的游戏笔记本电脑” 2. **流水线：**执行商品搜索 $\rightarrow$ 库存检查 $\rightarrow$排序（并行） 3. **用户：**“把第二个加入购物车” 4. **流水线：**指代消解 $\rightarrow$购物车更新 $\rightarrow$总额重新计算

5. **用户：** “应用任何可用的优惠券”

6. **流水线：** 促销搜索 $\rightarrow$ 资格检查$\rightarrow$应用折扣

该声明式流水线在本次会话中用 6 步完成了执行，而命令式基线需要 11 步，这主要得益于并行执行和条件路径优化。

6.4. 局限性

这项初步评估存在若干局限性：

- 数据集规模有限（1,000 个会话）
- 单一领域（电子商务）评估
- 未与其他声明式系统进行比较
- 来自单一部署环境的性能指标

跨多个领域的全面基准测试，以及与最先进的智能体架构进行比较，仍有待未来开展。然而，这些初步结果表明，声明式流水线抽象能够切实改善生产环境中的智能体系统的开发者体验和运行时性能。

7. 讨论

7.1. 初步测试部署的经验教训

我们为期三个月的初步测试处理了数百万次智能体交互，并揭示了声明式智能体架构的若干洞见：

表 2：声明式实现与命令式实现的比较

指标	命令式基线	我们的方法
任务成功率	78%	89%
平均完成步数	9.2	6.4
代码行数	850	220
开发时间（小时）	48	16
P95 延迟（毫秒）	240	185
修改时间（小时） *	8.5	2.0

*添加新工具或修改工作流所需的时间

配置即代码范式：将智能体行为作为版本化配置文件加以维护，从根本上改变了我们的开发工作流程。对比不同智能体策略进行 A/B 测试变得轻而易举——团队只需部署配置更改，而无需修改代码，就能尝试不同的流水线结构。这将实验周期从数天缩短至数小时，也使非工程人员能够在明确的防护措施范围内安全地修改智能体行为。

动态流水线管理： 无需重启服务即可从外部存储（数据库、缓存或配置服务）加载和修改流水线，这一能力极具价值。热切换流水线实现了快速的事件响应——当代理运行异常时，我们可以立即恢复到之前的配置。这种架构选择以牺牲一定的类型安全为代价，换取了运维灵活性；在测试环境中，这是值得的权衡。

涌现式优化模式： 这种新的声明式结构揭示了在命令式代码中不可见的优化机会。流水线中的公共子图可以被自动识别并缓存。并行执行点通过依赖分析自然涌现，而非通过显式编程实现。使用传统的智能体实现很难达成这些系统性的优化。

7.2. 设计权衡

我们的架构设计涉及若干有意为之的权衡：

表达能力与安全性： 虽然该领域特定语言能够表达复杂的工作流，但我们有意限制了某些操作（无界递归、任意代码执行），以防止生成格式错误的流水线。这种限制有时需要采用变通方法，但可以避免整类运行时故障。

性能与灵活性： 与编译后的代码相比，流水线执行的解释开销会增加 10–20 毫秒的延迟。不过，这使动态修改、全面检测和跨语言可移植性成为可能；对于大多数智能体应用而言，这些优势超过了适度的性能代价。

抽象与控制： 将大语言模型交互细节隐藏在工具和消息抽象之后，可以简化常见场景，但可能会令需要精细控制的用户感到受限。我们通过逃生口（自定义函数）解决了这一问题，使用户能够在需要时直接访问大语言模型，但这会破坏声明式范式。

7.3. 局限性

当前系统受到若干局限性的制约：

学习与适应： 虽然可以根据性能指标修改流水线，但该语言缺乏原生强化学习支持。若没有外部干预，智能体无法根据经验自动改进策略。在保持声明式简洁性的同时集成在线学习，仍然是一项尚待解决的挑战。

复杂状态管理： 该系统能够妥善处理对话级状态，但难以管理跨会话的长期记忆。在声明式范式中实现向量数据库或知识图谱，需要采用一些笨拙的变通方案，进而打破抽象边界。

调试挑战： 虽然声明式流水线简化了对智能体逻辑的理解，但运行时故障的调试仍可能颇具挑战。嵌套流水线和异步工具调用中的堆栈跟踪会掩盖错误来源。专为流水线执行设计的更完善调试工具，将显著提升开发者体验。

推理透明度有限：当-

前的流水线架构将大语言模型视为黑箱，无法洞察其推理过程。随着智能体可解释性在生产系统中日益重要，这种语言需要具备揭示并验证大语言模型决策的机制。

7.4. 未来方向

我们的方法开启了多个研究方向：

自适应流水线合成： 未来的系统或许能够从交互日志中学习最优的流水线结构，而不再需要手动构建流水线。这需要解决智能体工作流领域中的程序合成问题——这是一个颇具挑战但前景广阔的方向。

形式化验证： 声明式特性使得对智能体行为进行形式化推理成为可能。开发能够证明终止性、目标满足和安全约束等属性的验证工具，将增强人们对自主智能体的信任。

分布式智能体协调： 将该架构扩展为在分布式系统中协调多个智能体，需要引入用于同步、共识和冲突解决的新基本构件，同时保持声明式的简洁性。

神经符号集成： 将此类声明式流水线与大语言模型之外的神经组件相结合，例如视觉模型、强化学习智能体和可微推理模块，可以在保留可解释性的同时，实现更复杂的智能体能力。

7.5. 更广泛的影响

这项工作表明，成功的智能体设计架构必须在原始能力之外兼顾多方面需求。生产环境中的智能体系统需要具备可观测性、可修改性和安全保证，而这些特性很难通过命令式方法实现。声明式范式虽在某些方面有所限制，却为大规模构建可信智能体系统奠定了基础。

将智能体逻辑与实现细节分离，也会产生组织层面的影响。这种分离使新的协作模式成为可能：领域专家定义智能体行为，工程师则专注于平台能力。通过降低技术门槛，这种关注点分离有望加速智能体在各行业的部署。

8. 结论

本文介绍了一种声明式方法，用于在企业环境中构建和编排大语言模型驱动的智能体工作流。通过借助语言无关领域专用语言，将工作流规范与实现分离，我们使相同的流水线定义能够在多种后端语言和部署环境中执行，从根本上改变了生产级智能体的开发和维护方式。

我们的主要贡献包括：(1) 一种新颖的领域专用语言，提供控制流、数据操作和工具编排的基本构件，能够用远少于命令式方法所需的代码捕捉常见的智能体模式；(2) 一种混合执行模型，将可预测的流水线执行与动态的大语言模型驱动的工具选择相结合；(3) 一套经过生产环境验证的实现方案，每日处理数百万次交互，并在开发速度（提升 67%）、可维护性（修改速度提高 76%）和运行时效率（完成所需步骤减少 30%）方面展现出显著改进。

对真实电子商务工作流的评估验证了我们的假设：声明式抽象能够简化智能体开发，而不牺牲性能。或许更重要的是，该系统使非工程师也能通过配置更改安全地修改智能体行为，从而让智能体开发更加普及，并将实验周期从数天缩短至数小时。

这项工作提出了一项更广泛的原则：随着大语言模型智能体成为关键基础设施，为其提供支持的系统必须超越原型工具的发展阶段，以应对生产环境中对可靠性、可观测性和可维护性的要求。声明式范式提供了一条可行路径，既具备形式化推理能力，又能保持运维灵活性。

未来的工作将探索自适应流水线合成，以从交互日志中自动学习最优工作流结构；探索形式化验证方法，以证明智能体行为的安全属性；并为多智能体系统开发分布式协调原语。随着大语言模型的能力不断提升，我们相信，声明式流水线架构将在弥合强大模型与可信生产系统之间的差距方面发挥重要作用。

致谢

我们感谢 PayPal AI 团队为这项工作做出的宝贵贡献：感谢 Zhi Xuan 和 Jun Sheng Ng 进行了广泛的性能测试，感谢 Yun-Shiuan Chuang 和 Sudhanshu Garg 实现了电子商务工作流，验证了该语言的表达能力。

参考文献

[ASF, 2015] ASF（2015）。Apache Airflow：用于以编程方式编写、调度和监控工作流的平台。https://airflow.apache.org/。Apache 软件基金会。访问日期：2025年11月。

[Chase, 2023] Chase, H.（2023）。LangChain：通过组合性构建大语言模型应用。https://github.com/langchain-ai/langchain。访问日期：2025年11月。

[LangChain4j 贡献者, 2024] LangChain4j 贡献者（2024）。LangChain4j：用于大语言模型应用的 Java 库。https://github.com/langchain4j/langchain4j。访问日期：2025年11月。

[英伟达, 2024] 英伟达（2024）。NeMo Agent Toolkit：可扩展且可优化的智能体微服务。https://github.com/NVIDIA/NeMo-Agent-Toolkit。NVIDIA Corporation。访问日期：2025年11月。

[Significant Gravitas, 2023] Significant Gravitas（2023）。AutoGPT：一项旨在使 GPT-4 完全自主的实验性开源尝试。https://github.com/Significant-Gravitas/AutoGPT。访问日期：2025年11月。

[Temporal, 2023] Temporal（2023）。Temporal：持久化执行平台。https://temporal.io/。Temporal Technologies, Inc. 访问日期：2025年11月。