mp.weixin.qq.com /s/WYwjH08OkislsVq-XP_ASQ

声明式多智能体系统的架构剖析(上篇)

熊节：16-20 minutes

Part I：引言

AI正在改变架构的基本假设

当我们讨论软件架构时，传统上我们在讨论什么？我们在讨论如何组织类、如何划分模块、如何设计API、如何管理数据流。这些讨论的基础假设是：系统的行为通过代码精确定义，运行时环境机械地执行这些定义。从Erich Gamma等人的《设计模式》到Martin Fowler的《企业应用架构模式》，半个多世纪的软件工程智慧都建立在这个基本假设之上。

然而，当AI模型成为系统的核心执行单元时，这个假设开始动摇。系统的行为不再由代码精确规定，而是由自然语言描述的意图和标准引导。运行时环境不再是被动的执行器，而是主动的理解者和决策者。传统架构模式中的许多概念——如接口、抽象类、依赖注入——开始显得过于机械和低层，而新的架构概念——如Agent Blueprint、智能运行时、数据驱动行为——开始浮现。

这不是技术栈的简单更替，而是架构思维方式的根本性转变。我们面临一个严峻的问题：如何描述和讨论这种新型系统的架构？传统的架构词汇是否足够？我们需要什么样的新模式语言？

为什么需要模式语言

Martin Fowler在《企业应用架构模式》的序言中写道：“模式的价值不在于新颖性，而在于将经过时间检验的解决方案编纂成可传播的形式。”模式语言是一个行业成熟的标志——它表明我们已经从个案探索进入到系统化知识积累的阶段。

当前，AI多智能体系统的开发仍处于探索期。每个团队都在用自己的方式构建系统，使用自己的术语描述架构。“智能体”这个词在不同的系统中可能指代完全不同的概念。缺乏共同的模式语言导致知识难以传播，经验难以复用，问题难以清晰讨论。

我们需要的是一套针对AI多智能体系统的模式语言——不是生硬地套用传统模式，也不是凭空创造概念，而是从实际系统中提炼出可复用的设计知识，为这些知识命名，阐明它们的适用场景和权衡，并揭示它们之间的关系。这正是本文的目标。

案例系统概览

本文分析的系统表面上看很简单：每月自动生成一份追踪某些组织活动的报告。但当你深入观察它的运作方式时，会发现它代表了一种新的软件构建范式——系统中没有一行传统意义上的“代码”，取而代之的是：

- • **1200行Markdown格式的自然语言提示词**，描述智能体应该做什么

- • **1400行JSON和Markdown格式的参考数据**，定义领域知识和质量标准
- • 一个**智能运行时环境**(Claude Code)，理解这些描述并智能执行

系统的核心工作流程是：搜寻六个组织在特定月份的所有公开活动，将非结构化的网页内容转化为结构化数据，再转化为符合学术规范的中文报告，最后进行质量检查和整合。整个过程涉及大量的网络搜索、信息提取、语言生成和质量判断。传统上，这需要编写数千行爬虫代码、数据清洗逻辑、模板引擎和验证规则。

更重要的是，这种架构不是技术炫技，而是对特定问题的深思熟虑的解决方案。它在批处理、信息密集、质量优先的场景下表现出色，但也清楚地知道自己的边界——它不适合实时系统、不适合确定性要求极高的场景、不适合离线环境。理解这个架构的价值，在于理解它所代表的设计哲学：**在合适的约束条件下，声明式+智能运行时的组合，可以带来开发效率、适应性和可维护性的质的飞跃**。

这个系统为我们提供了一个具体的、可分析的案例，从中我们可以提炼出模式，理解权衡，并探索AI时代的架构设计原则。

分析方法：借鉴PoEAA的经验

在描述这个系统的架构时，我们面临一个方法论问题：如何系统地分析和呈现架构知识？幸运的是，我们有Martin Fowler的《企业应用架构模式》作为方法论参考。

Fowler的书之所以成功，不仅因为它收录了有价值的模式，更因为它采用了一种有效的叙述结构：**先叙事，后模式**。书的第一部分(“The Narratives”)通过问题驱动的叙述，建立读者对领域问题的理解，引入核心概念和权衡。第二部分(“The Patterns”)则是结构化的模式目录，每个模式遵循固定格式：问题、解决方案、权衡、示例。

这种结构的智慧在于：**模式不是凭空出现的，而是从具体问题中自然涌现的**。如果直接呈现模式列表，读者会觉得抽象和空洞;但如果先通过叙述建立问题意识，模式的价值就变得显而易见。

本文借鉴了这种方法，但做了适应性调整：

第一部分：架构解剖(本篇)

- • 建立词汇表：明确定义系统中的核心概念
- • 静态结构分析：剖析文件组织和设计意图
- • 动态执行追踪：通过一次完整执行理解运行机制
- • 运行时环境分析：深入理解智能运行时的角色

第二部分：模式与反思(下篇，待续)

- • 架构模式提炼：从具体系统中抽象出可复用模式
- • 模式语言构建：揭示模式间的关系网络
- • 对比研究：与传统架构和其他AI系统对比
- • 综合反思：这个架构代表了什么，预示着什么

这种结构确保了分析的深度和系统性：我们不是肤浅地列举特征，而是层层剖析，从具体到抽象，从观察到洞察，从描述到评判。

更重要的是，我们遵循了Fowler强调的一个原则：**问题驱动，而非技术驱动**。我们不是为了炫耀AI技术而构建系统，而是为了解决实际问题——如何高效地生成高质量的月度追踪报告。技术选择(声明式、智能运行时、文件系统数据总线)都是从这个问题出发，经过深思熟虑的权衡。

接下来，让我们开始架构解剖的旅程。

Part II：架构解剖

第一步：建立词汇表

在分析这个多智能体系统的架构之前，我们面临一个根本性的困难：缺乏一套明确的词汇来描述系统中的各种概念。“智能体”(Agent)这个词在不同语境下被用来指代不同的事物——有时指一个Markdown文件，有时指一个正在执行任务的实例，有时又指整个Claude Code环境。这种词汇的模糊性阻碍了我们对架构的清晰理解。

让我们从静态结构(设计时)和动态结构(运行时)两个维度来识别和定义核心概念。

核心概念一：**Agent Blueprint(智能体蓝图)**

定义：存储在`prompts/`目录下的Markdown文件，使用自然语言详细描述一个智能体应该完成什么任务、遵循什么规范、如何与其他组件协作。

典型示例：

- • `01-research-agent.md`：定义Research Agent的职责、搜寻策略、数据记录要求
- • `02-writing-agent.md`：定义Writing Agent的写作任务、文体规范、质量标准

关键特征：

- • 使用声明式的自然语言，而非指令式的代码
- • 详细描述“做什么”和“质量标准”，但不规定“怎么做”
- • 包含角色定义、任务参数、参考资料、工作流程、输出要求等结构化信息
- • 可以包含参数占位符(如`{ORGANIZATION_ID}`, `{YEAR}`)，在实例化时被替换

命名理由：我们选择“Blueprint”(蓝图)这个词，是因为它强调了这些文件的设计性质——它们是智能体的设计图纸，而非智能体本身。建筑师的蓝图不是建筑，而是对建筑的描述;同样，Agent Blueprint不是Agent，而是对Agent行为的描述。

核心概念二：**Agent Instance(智能体实例)**

定义：在运行时，基于Agent Blueprint创建的、正在执行特定任务的智能体实体。一个Agent Instance是由运行时环境(Claude Code)读取Agent Blueprint并结合具体参数而启动的执行单元。

典型示例：当Orchestrator启动阶段一时，基于`01-research-agent.md`创建了6个Research Agent Instance，分别负责搜寻6个不同组织的活动。每个Instance接收不同的参数，这6个Instance并行执行，相互独立。

关键特征：

- • 具有独立的执行上下文和状态
- • 拥有具体的任务参数(如目标组织、目标月份)
- • 可以并行执行(多个Instance同时运行)
- • 完成任务后产生输出(文件)，然后终止

与传统概念的对比：类似于面向对象编程中的“对象实例”,Blueprint是“类”,Instance是“对象”。不同于传统的“进程”的是，Agent Instance由AI模型驱动，具有理解、推理和决策能力。

核心概念三：**Runtime Environment(运行时环境)**

定义：提供Agent Instance执行所需的所有基础设施和能力的环境。在本系统中，这个角色由Claude Code承担。

提供的能力：

- • **文件系统操作**：创建目录、读取文件、写入文件、搜索文件
- • **网络能力**：搜索引擎、网页抓取、内容提取
- • **Agent管理**：启动Agent Instance、传递参数、并行调度、等待完成
- • **工具集成**：提供Read、Write、WebSearch、WebFetch等工具
- • **智能决策**：理解自然语言指令并转化为具体行动

关键特征：不仅是被动的“容器”，而是主动参与决策的智能环境。能够理解自然语言指令(Agent Blueprint)并将其转化为具体行动。

命名问题：“Runtime Environment”这个词可能不够准确，因为Claude Code的作用远超传统的运行时环境。它更像是一个“智能执行平台”(Intelligent Execution Platform)。但为了保持与传统术语的连续性，我们暂且使用Runtime Environment，同时明确其“智能”特性。

核心概念四：**Data Bus(数据总线)**

定义：Agent Instance之间传递数据的机制。在本系统中，文件系统充当了数据总线的角色。

工作方式：

- • Research Agent Instance将搜寻结果写入`data/{YYYY-MM}/raw/{org_id}/`
- • Writing Agent Instance从raw目录读取，将报告写入`data/{YYYY-MM}/drafts/`
- • Integration QA Agent从drafts目录读取，生成最终报告到`data/{YYYY-MM}/final/`

关键特征：

- • 基于约定的目录结构和文件命名
- • 使用标准化的数据格式(JSON、Markdown)
- • 松耦合：Agent之间不需要知道彼此，只需要知道数据的位置和格式
- • 可追溯：所有中间产物都被保留

设计权衡：使用文件系统作为数据总线牺牲了性能和实时性，换取了简单性、可检查性和与AI工具的天然契合。

核心概念五：**Reference Data(参考数据)**

定义：存储在`references/`目录下的结构化数据和文档，用于规范和指导Agent的行为。

具体内容：

- `organizations.json`：定义需要追踪的组织的基本信息
- `activity-types.json`：定义84种活动类型的分类体系
- `activity-record-template.json`：定义活动记录的标准化字段结构
- `output-format-guide.md`：详细的报告写作规范(1028行)

架构意义：Reference Data的存在体现了“数据驱动”的架构思想——系统的行为不是硬编码在Agent Blueprint中，而是由外部数据定义。改变Reference Data可以改变系统行为，而无需修改Agent Blueprint。

静态与动态的对应关系

理解这个系统的一个关键是区分静态结构(设计时)和动态结构(运行时)：

静态结构(在文件系统中可见)：

```
project/
├── prompts/              # Agent Blueprints
├── references/           # Reference Data
└── data/                 # Data Bus(结构定义)
```

动态结构(运行时)：

```
[Orchestrator Instance]
    ↓ 读取01-research-agent.md,参数化,批量启动
[Research Agent × 6] 并行执行
    ↓ 写入raw/目录
[Writing Agent × 6] 并行执行
    ↓ 写入drafts/目录
[Integration QA Agent × 1] 串行执行
    ↓ 写入final/目录
```

静态元素	动态元素	关系
Agent Blueprint	Agent Instance	Blueprint是模板，Instance是运行实体
Reference Data	Agent Instance的输入	Data指导Instance的行为
Data目录结构	Data Bus	目录定义数据流动路径
Runtime Environment	不存在静态对应物	运行时基础设施

有了这套精确的词汇表，我们现在可以清晰地讨论系统的架构，而不会陷入术语混淆的泥潭。

第二步：静态结构分析

打开项目目录，你会看到清晰的三层分离：

```
project/
├── prompts/                # 行为定义层(1200行Markdown)
│   ├── 00-orchestrator.md        (220行)
│   ├── 01-research-agent.md      (262行)
│   ├── 02-writing-agent.md       (331行)
│   └── 03-integration-qa-agent.md(398行)
│
├── references/             # 领域知识层(1400行数据)
│   ├── organizations.json        (6个组织元数据)
│   ├── activity-types.json       (84种活动类型)
│   ├── activity-record-template.json(数据契约)
│   └── output-format-guide.md    (1028行质量规范)
│
└── data/                   # 运行时数据层
    └── {YYYY-MM}/          # 按月分区
        ├── raw/            # 阶段一输出:结构化数据
        ├── drafts/         # 阶段二输出:报告草稿
        └── final/          # 阶段三输出:最终报告
```

这个结构不是随意的文件堆砌，而是深思熟虑的架构设计的体现。让我们逐层剖析。

行为定义层：prompts/目录

文件名以数字前缀编号(00, 01, 02, 03)，这不仅仅是方便排序，更重要的是反映了执行顺序：

- • **00-orchestrator**：最先执行，负责启动其他所有智能体
- • **01-research-agent**：第一阶段执行，进行信息搜寻
- • **02-writing-agent**：第二阶段执行，进行报告撰写
- • **03-integration-qa-agent**：第三阶段执行，进行整合质检

这种编号约定使得系统的执行流程在目录结构中一目了然，是一种自文档化的设计。

虽然四个Agent Blueprint定义了不同的智能体，但它们遵循一个共同的结构模式：

```
# Agent Blueprint结构模式

## 你的角色
[定义智能体的身份和总体职责]
```

```
## 任务参数
[列出智能体接收的参数,包含占位符]

## 参考资料
[指明需要读取的Reference Data文件]

## [核心工作内容]
[详细的任务描述、策略、流程]

## 输出要求
[明确的产出标准和文件路径]

## 完成标准
[质量检查清单]
```

这个共同的结构模式体现了：一致性(降低理解成本)、完整性(确保必要信息)、可组合性(清晰的输入输出)、质量导向(每个Blueprint都包含完成标准)。

特别值得注意的是`00-orchestrator.md`的特殊地位。它不仅是一个Agent Blueprint，更是整个系统的“元程序”——它描述了如何协调其他智能体。从软件工程的角度看，Orchestrator相当于一个“编译器”或“解释器”——它读取其他Blueprint并将它们转化为可执行的Agent Instance。但与传统编译器不同的是，这个“编译”过程是由AI理解和执行的。

知识层：**references/目录**

这一层包含四类Reference Data，每类都外化了一种知识：

1. organizations.json：搜索空间定义
定义了系统的**搜索目标**。如果要追踪新组织，只需在这个文件中添加新条目，无需修改Agent Blueprint。这体现了**数据驱动的可扩展性**。

2. activity-types.json：领域本体
10大类84种活动类型的层次化分类体系。这不是简单的“配置”，而是对“组织活动”这个领域的系统性认识的编码。

3. activity-record-template.json：数据契约
定义了活动记录的标准化字段结构。这个模板确保Research Agent产生的数据具有一致结构，Writing Agent知道如何解读，Integration QA Agent知道如何验证。字段被分为critical、important、supplementary三个优先级，体现了**务实的数据质量策略**。

4. output-format-guide.md：质量规约
1028行的详细写作规范，将“什么是高质量报告”这个模糊概念转化为可操作标准。这个文件的规模表明，**质量标准的定义是系统中最复杂的部分之一**，反映了一个设计哲学：质量不是检查出来的，而是设计和引导出来的。

这四个文件构成了**分层的领域知识外化**：

```
领域知识的层次：
├── 元数据层：组织信息（我们追踪谁）
├── 本体层：概念分类体系（活动有哪些类型）
├── Schema层：数据结构定义（数据应该如何组织）
└── 规范层：质量标准（好的报告是什么样的）
```

每一层都可以独立演化，职责边界清晰，同一份知识可以被多个Agent使用。

数据层：data/目录

顶层按时间周期分区（{YYYY-MM}/），反映了系统的**批处理性质**：每个月度报告是一个独立的、完整的执行单元。

在每个月份目录下，数据按照处理阶段分为三层：

```
raw/      （Research Agent输出）
  ↓ 从非结构化到结构化
drafts/   （Writing Agent输出）
  ↓ 从结构化到叙述性
final/    （Integration QA Agent输出）
  ↓ 从分散到整合
```

这个三层结构直接映射了系统的三阶段处理流程，体现了几个关键特点：

单向数据流：数据只向前流动，不回溯。这种设计简化推理、支持并行、支持重试。

分组织存储：在raw/层，每个组织有独立子目录，使得6个Research Agent Instance可以并行写入不会冲突。

双文件模式：每个组织目录包含`activities.json`（机器可读）和`sources.md`（人类可读），平衡了机器处理和人工审查的需求。

最终层的特殊性：final/层包含`report-{YYYY-MM}.md`（结果）和`qa-checklist.md`（过程），体现了**过程与结果的并重**。qa-checklist不仅记录了“检查通过”，还记录了“检查了什么”、“发现了什么”，使质量过程透明化。

静态结构的整体架构模式

综合三个目录，我们可以识别出几个关键的架构模式：

模式1：代码-数据-运行时三层分离

```
prompts/    ← 定义"做什么"
    ↑ read
references/ ← 提供"标准和规范"
    ↑ read
data/       ← 存储"执行结果"
```

三层之间是单向依赖，没有反向依赖。

模式2：声明式配置模式
整个系统的行为由行为配置(Agent Blueprint)和数据配置(Reference Data)定义。改变配置可以改变系统行为，而运行时环境保持不变。这类似于“基础设施即代码”，但应用于智能体系统：**行为即声明**。

模式3：文件系统即数据库
系统将文件系统当作数据库：目录结构相当于表结构，JSON文件相当于表记录，文件存在性相当于事务完成标志。这种设计牺牲了性能，换取了简单、可检查、版本控制友好、AI友好。

模式4：约定优于配置
系统大量使用约定来减少显式配置：文件命名约定(`00-{YYYY-MM}`)、目录结构约定(`data/{YYYY-MM}/raw/{org_id}/`)、文件名约定(`activities.json`总是存储活动数据)。这些约定使得Agent Blueprint不需要显式配置文件路径，减少了配置，增强了一致性。

缺失的元素

值得注意的是一些在传统软件中常见但这里缺失的元素：

没有传统代码文件：系统中没有`.py`、`.js`等代码文件。这个缺失是**有意为之的架构决策**，反映了设计者选择将所有可以交给AI决策的部分都声明化。

没有测试代码：这引发一个有趣的问题：如何测试一个由AI驱动的声明式系统？系统采用的质量保证策略是：在Agent Blueprint中嵌入“完成标准”，用Integration QA Agent进行全面质检，保留所有中间产物供人工审查。这可能代表了**AI系统质量保证的新范式**：从“测试正确性”转向“验证符合规范性”。

没有依赖管理文件：系统的“依赖”是Runtime Environment提供的工具和能力，是隐式的，内嵌在Claude Code中。这简化了部署，但也增加了对特定Runtime Environment的绑定。

没有配置文件：系统的“配置”已经融入Reference Data和Agent Blueprint中，没有独立的运行时配置层。

第三步：动态执行追踪

静态结构告诉我们系统“是什么”，而执行追踪则揭示系统“如何运作”。让我们通过一次完整的202509月度报告生成过程，观察系统的动态行为。

初始化阶段：启动与准备

执行的起点是Orchestrator Agent接收到指令：生成202509的月度报告。

Orchestrator的第一个动作是读取自己的Agent Blueprint——`00-orchestrator.md`。这个动作至关重要，因为它体现了系统的核心架构特征：**行为与执行的分离**。Orchestrator并非通过硬编码的程序逻辑来执行任务，而是通过读取自然语言描述的蓝图来理解自己应该做什么。

Orchestrator在读取提示词后立即规划了整个工作流程，并将其转化为任务清单：

```
- [ ] 确认参考文件存在
- [ ] 创建2025-09工作目录结构
- [ ] 阶段一:6个组织并行信息搜寻
- [ ] 阶段二:6个组织并行报告撰写
- [ ] 阶段三:整合质检生成最终报告
- [ ] 向用户交付最终报告
```

这个任务清单不仅是进度追踪工具，更是**执行计划的具象化**，揭示了Orchestrator对整个流程的理解：六个步骤，三个主要阶段，从数据准备到最终交付，体现了一种**管道式架构**。

Orchestrator随后验证所有必需的参考文件是否存在(organizations.json、activity-types.json、activity-record-template.json、output-format-guide.md)。这个看似简单的检查体现了**防御性设计原则**：在启动昂贵的网络搜寻任务之前，确保所有必需的基础设施和配置都已就位。

然后Orchestrator创建本次执行所需的工作目录结构：

```
mkdir -p data/2025-09/raw/{org_1,org_2,org_3,org_4,org_5,org_6}
mkdir -p data/2025-09/drafts
mkdir -p data/2025-09/final
```

这是基于**文件系统的Data Bus模式**的体现。目录结构本身蕴含架构信息：`raw/`下按组织划分暗示了数据的分区策略，三层目录结构(**raw → drafts → final**)直接映射了数据精炼的三个阶段。

阶段一：并行信息搜寻

在工作目录准备完毕后，Orchestrator进入第一阶段：信息搜寻。核心是启动6个并行的Research Agent Instance，分别负责搜寻6个不同组织的活动。

Orchestrator首先读取了`01-research-agent.md`的内容。这个动作值得注意：Orchestrator不是直接知道如何启动Research Agent的，而是通过读取Agent Blueprint来学习。这再次体现了系统的**声明式特性**。

随后，Orchestrator在同一个消息中发起了6个并行的Task调用：

```
Task:搜寻某组织A 2025年09月活动
Task:搜寻某组织B 2025年09月活动
Task:搜寻某组织C 2025年09月活动
...
```

每个Task的输入都包含：

- • **subagent_type**:"general-purpose"

- • **description**：人类可读的任务描述
- • **prompt**：完整的Agent Blueprint内容，其中参数占位符已被替换

这种**参数化的Agent实例化**机制是系统实现可扩展性的关键。同一个Agent Blueprint通过不同参数实例化，产生了6个独立的Agent Instance。

从执行时间戳看，6个Research Agent在大约10分钟内全部完成。如果串行执行，总耗时将是现在的6倍。**并行化**带来了显著的性能提升。

但并行化不仅是性能优化手段，更是**架构约束的反映**。为什么这6个Agent可以并行？因为它们之间**没有数据依赖**。每个Agent负责一个独立组织，搜寻的信息互不重叠，产生的数据写入不同目录。这种设计体现了：**无状态与数据分区**原则。

每个Research Agent的工作过程是：

1. 1. 网络搜索：使用WebSearch工具搜索特定组织在202509的活动
2. 2. 内容提取：使用WebFetch工具访问相关网页，提取活动信息
3. 3. 数据结构化：根据`activity-record-template.json`，将非结构化网页内容转化为结构化JSON
4. 4. 文件写入：将数据写入`data/2025-09/raw/{org}/activities.json`，同时创建`sources.md`记录信息来源
5. 5. 摘要报告：生成简短摘要，汇报搜寻结果

这个过程展示了AI智能体的独特优势：**非结构化数据的理解与结构化**。传统爬虫可以抓取网页，但很难理解语义，无法准确提取“活动”这个抽象概念。而Research Agent通过语言模型，可以识别哪些信息构成一个“活动”，并映射到预定义数据结构。

在所有6个Research Agent完成后，Orchestrator进行了第一阶段总结，生成了详细的搜寻结果概览表，还提取了跨组织模式。这种**元级别的监控**是Orchestrator角色的重要职责：不仅协调流程，还评估质量，识别异常。

阶段二：并行报告撰写

第一阶段完成后，系统进入第二阶段：报告撰写。模式与第一阶段高度相似。

Orchestrator读取了`02-writing-agent.md`，然后再次并行启动了6个Agent Instance，每个负责撰写一个组织的报告章节。

这种模式的重复是一种**架构模式的复用**。Orchestrator似乎遵循一个通用协调策略：读取Agent Blueprint → 为每个数据分区实例化一个Agent → 等待所有Agent完成 → 汇总结果。这种策略的通用性使得系统可以轻松扩展。

Writing Agent的输入是第一阶段的输出：`data/2025-09/raw/{org}/`目录下的`activities.json`和`sources.md`。任务是将这些结构化JSON数据转化为符合格式规范的中文Markdown报告。

这个转换过程涉及几个关键智能任务：

- • 数据解析：读取并理解JSON结构
- • 内容组织：决定活动呈现顺序

- • 语言生成：将结构化字段转化为流畅中文叙述
- • 格式控制：遵循`output-format-guide.md`的文体规范
- • 引用管理：为每个活动标注信息来源
- • 篇幅控制：确保每个活动描述在合理长度

这些任务如果用传统编程实现，需要复杂的模板引擎、自然语言生成系统和格式验证逻辑。而Writing Agent通过语言模型能力，可以在统一框架内完成。这展示了AI智能体的另一个优势：**多种智能任务的统一处理**。

在所有Writing Agent完成后，Orchestrator再次汇总，并明确列出了写作质量保证检查点，体现了**分层质量控制策略**。

阶段三：整合与质量检查

前两阶段都采用并行化策略，但第三阶段采用了完全不同的模式：**启动唯一的**Integration QA Agent。

这个转变不是偶然，而是由任务性质决定。整合任务需要读取所有6个组织的草稿报告，将它们合并成统一文档，并进行跨章节一致性检查。这些操作本质上是**全局性**的，无法分区并行化。这体现了：**并行化不是万能的，某些任务天然是串行的**。

Integration QA Agent的任务是多方面的：

- • 文档整合：读取所有`drafts/{org}.md`，按统一结构合并
- • 格式统一：确保整个文档的Markdown格式一致
- • 语言润色：检查和修正语言表达
- • 事实核查：验证信息准确性，检查引用完整性
- • 质量评估：根据预定义质检清单，评估报告各维度
- • 问题修正：如果发现问题，进行必要修正

Integration QA Agent产生两个关键输出：

- • `final/report-2025-09.md`：最终整合的完整报告
- • `final/qa-checklist.md`：质检清单，记录所有检查项的结果

第二个文件的存在特别值得注意。它不仅是内部工作产物，更是**质量保证过程的可追溯性证据**。任何人都可以查看这个清单，了解报告经过了哪些检查。这种透明性在传统自动化系统中很少见。

控制流与数据流的整体图景

通过完整执行追踪，我们可以看到系统的控制流模式：

阶段驱动的状态机：

```
初始化 → 阶段一(Research) → 阶段二(Writing) → 阶段三(Integration QA) → 交付
```

每个状态转换的触发条件是：**前一阶段的所有Agent完成**。这是一种**基于完成的同步机制**，类似于**屏障同步**(Barrier Synchronization)。

数据在系统中经历了三个精炼层次：

```
网页内容(非结构化)
  → activities.json(结构化)
    → drafts/{org}.md(叙述性)
      → final/report.md(整合性)
```

每个箭头代表一个Agent的处理：

- • Research Agent：非结构化 → 结构化
- • Writing Agent：结构化 → 叙述性
- • Integration QA Agent：叙述性 → 整合性

这种**渐进式精炼**(Progressive Refinement)的模式在数据处理系统中很常见，但在AI智能体系统中的实现是独特的：不是通过预定义转换函数，而是通过AI的理解和生成能力。

Agent之间没有直接接口定义，契约是**隐式的文件路径约定**。观察执行历史，我们注意到一个重要特性：每个阶段产生的数据不会被后续阶段修改。这种**数据不可变性**(Data Immutability)带来了可追溯性、并发安全、容错性。

系统还展示了一个经典的**分区-聚合**(Partition-Aggregate)模式：

```
阶段一:1个任务 → 6个分区(按组织)
阶段二:6个分区 → 6个分区(保持分区)
阶段三:6个分区 → 1个聚合报告
```

第四步：智能运行时环境剖析

在传统软件架构中，运行时环境是相对被动的角色。JVM机械执行字节码，Python解释器忠实执行脚本，操作系统按指令调度进程。它们的职责是**准确执行**开发者编写的指令，而非**理解和决策**。

然而在这个多智能体系统中，Claude Code扮演的运行时环境角色与传统概念有本质不同。用户的一句话道出了这个差异的核心：“我自己写的bash或Python代码，不会比Claude Code执行到那儿的时候选择的指令方法更好。”

这句话揭示了一个深刻的架构转变：**从“指令式执行”到“意图理解与智能执行”**。

Claude Code的能力清单

第一层：基础工具能力

- • 文件系统操作：创建目录、读写文件、搜索文件
- • 网络能力：WebSearch工具、WebFetch工具、内容理解

- • Shell命令执行：Bash工具支持任意shell命令

这些看似简单，但Claude Code提供了一个**完整的文件系统抽象层**。Agent Blueprint中只需声明“读取某文件”,Claude Code会自动选择合适工具、处理错误、确保权限正确。

第二层：**Agent管理能力**

- • Agent实例启动：通过Task工具启动新的Agent Instance
- • 并行调度：在同一消息中发起多个Task，自动并行执行
- • 上下文隔离：每个Agent Instance有独立上下文
- • 上下文传递：通过文件系统共享数据

这是**元编程机制**。Orchestrator不是通过API调用来启动Agent，而是通过传递一个“提示词”来定义一个新的智能体。这个新智能体读取提示词，理解任务，然后自主执行。

第三层：**智能决策能力**

这是Claude Code最独特、最强大的能力层：

自然语言理解：能够理解Agent Blueprint中的自然语言描述，将其转化为具体执行计划。

例如，Agent Blueprint写道：“访问组织的官方网站，查找活动日历”。Claude Code需要理解：

- • “访问官方网站”意味着先从organizations.json读取URL，然后用WebFetch获取内容
- • “查找活动日历”意味着在网页内容中识别特定结构或关键词
- • “优先级最高”意味着遇到问题时应重点确保这部分完成

工具选择与组合：在执行时拥有**完整的上下文信息**(当前任务目标、可用工具能力、已完成操作、环境当前状态)，基于这些信息做出**最优的工具选择决策**。

如果开发者提前写死bash或Python代码，就失去了这种**运行时适应性**。代码只能基于开发时假设，而Claude Code可以基于执行时实际情况。

错误处理与恢复：Agent Blueprint可以声明性地描述期望和标准，Claude Code会智能处理异常。例如：“如果某个字段信息不可获得，留空或标注'信息未公开'”。Claude Code理解这些规则，在遇到问题时会尝试从备选来源获取，如果仍失败，按规则标注，继续执行而非崩溃。

质量判断与优化：Claude Code不仅执行任务，还**评估输出质量**。读取output-format-guide.md理解质量标准，在生成报告时不断对照标准自我检查，发现不符合时自我修正。

第四层：**元认知能力**

最高层次是Claude Code的**元认知**——对自己执行过程的认知和管理：

进度追踪：Todo工具允许Orchestrator维护任务清单，实时更新状态，提供自我监控和用户可见性。

执行规划：Orchestrator在读取Agent Blueprint后，先生成执行计划(Todo列表)，然后按计划执行，体现了**计划-执行分离**模式。

反思与调整：虽然在这次执行中未明显体现，但Claude Code理论上具有反思能力：如果某个策略失败，可以分析原因并调整;如果输出质量不符合预期，可以重新生成。

与传统运行时的对比

维度	传统运行时(JVM/Python)	智能运行时(Claude Code)
输入	字节码/源代码	自然语言Agent Blueprint
执行方式	机械执行指令序列	理解意图，智能选择执行方式
决策能力	无(完全由代码决定)	强(可在执行时做出适应性决策)
错误处理	抛出异常，停止执行	理解错误情境，尝试恢复或降级
优化	JIT编译等低层优化	策略选择、工具组合等高层优化
确定性	高(相同输入→相同输出)	中(相同Blueprint可能有不同执行路径)

根本差异：传统运行时是**被动执行器**，忠实执行每一行代码，不做判断。Claude Code是**主动决策者**，理解目标，选择方法，评估结果。

关键引语的深度分析

“我自己写的bash或Python代码，不会比Claude Code执行到那儿的时候选择的指令方法更好。”

为什么这是真的?

原因1：信息不对称

- • 开发时：只知道任务一般性描述，不知道具体执行时环境状态，必须做各种假设
- • 执行时：知道具体要处理的组织，知道当前网络状态，知道前序步骤结果，可以根据实际情况调整

Claude Code拥有**执行时的完整上下文**，可以做出更优决策。

原因2：智能的层次不同

- • 预写代码：只能包含开发者预见的逻辑分支
- • Claude Code智能：可以处理未预见情况(网站结构改变了？分析新结构提取信息)

Claude Code的**智能是开放式的**，不限于预定义逻辑分支。

原因3：工具选择的灵活性

- • 预写代码：必须提前决定用什么工具
- • Claude Code：根据实际情况选择最优工具组合(页面简单？直接WebFetch;内容复杂？先WebFetch再用Grep精确提取)

Claude Code有**工具选择的自由度**，可以优化执行效率。

原因4：质量感知

- • 预写代码：只能检查格式正确性
- • Claude Code：可以评估内容质量(这个活动描述是否客观？这个日期是否合理？)

Claude Code有语义层面的质量感知。

这揭示了一个深刻架构洞察：**将决策推迟到执行时**是一种强大的架构模式。传统软件工程强调提前规划、详细设计、代码固化逻辑。AI驱动的架构开启了新可能：高层规划、规范约束、运行时决策。

声明式与执行的分离

Claude Code作为智能运行时，体现了一个重要架构原则：**声明式定义与智能执行的分离**。

Agent Blueprint：声明层

- • 职责：定义角色职责、描述任务目标、规定质量标准、提供策略指导
- • 特征：使用自然语言、描述性而非指令性、留有执行时决策空间

Claude Code：执行层

- • 职责：理解Blueprint意图、选择合适工具和方法、执行具体操作、评估结果质量、处理异常和边界情况
- • 特征：具有理解和推理能力、可以做出运行时决策、适应具体情况

分离的价值：

- • **灵活性**：声明可以保持不变，执行可以优化
- • **可维护性**：修改质量标准只需修改Blueprint中描述
- • **可理解性**：Blueprint是人类可读的，即使非程序员也能理解
- • **可移植性**：理论上，同样Blueprint可以在不同智能运行时上执行

这种分离不是绝对的。Blueprint不是纯粹“声明”，也包含一些“过程性”指导(如“第一步”、“第二步”)。为什么不完全声明式？因为**完全声明式可能导致执行不可控**。Blueprint中的过程性指导是一种**引导**(guidance)，而非**指令**(instruction)。它告诉Claude Code“建议的策略是什么”，但Claude Code仍有灵活性调整。

这体现了“agents on rails”概念：**在轨道上的智能体**。轨道提供方向和约束，但智能体仍有决策自由度。

局限性与依赖

Claude Code虽然强大，但也有局限：

- • **不可离线运行**：依赖在线服务，需要网络连接
- • **运行成本**：每次执行Agent Instance消耗AI模型调用，有成本
- • **执行的不确定性**：同样Blueprint和参数，不同次执行可能产生略有不同结果
- • **对特定运行时的绑定**：强依赖Claude Code特定能力，难以迁移到其他AI平台

这些局限是否意味着架构有问题？不一定。

权衡的逻辑：

- • 用“执行成本”换“开发成本”：虽然运行成本高，但开发成本极大降低
- • 用“不确定性”换“适应性”：虽然失去确定性，但获得了适应复杂情况能力

- • 用“平台绑定”换“能力增强”：虽然绑定到Claude Code，但获得了强大理解和决策能力

适用场景：

- • 任务复杂多变，难以用固定代码实现
- • 执行频率不高(如月度报告)，运行成本可接受
- • 质量重于速度
- • 开发团队规模小，开发效率重要

不适合的场景：

- • 高频执行(如实时系统)
- • 成本敏感
- • 需要绝对确定性(如金融交易)
- • 需要离线运行

小结与展望

通过这四步架构解剖——建立词汇表、分析静态结构、追踪动态执行、剖析运行时环境——我们对这个声明式多智能体系统有了深入理解。

我们看到了一个精心设计的架构，它通过明确的分工(三层分离)、清晰的数据流(三阶段精炼)、合理的并行化(数据分区)和严格的质量控制(分层保证)，完成复杂的信息处理任务。

我们理解了系统的核心架构特征：

- • **声明式定义**：用自然语言描述意图和标准，而非代码规定步骤
- • **智能运行时**：Claude Code理解意图，智能选择方法，评估质量
- • **文件系统数据总线**：简单、透明、可追溯
- • **数据驱动行为**：通过修改Reference Data改变系统行为
- • **分层质量保证**：在每个环节构建质量，而非最后检查

我们也认识到了系统的适用边界和权衡选择：它在批处理、信息密集、质量优先场景下表现卓越，但不适合实时、高频、确定性要求极高的场景。

然而，架构解剖只是第一步。我们还没有系统地提炼出可复用的架构模式，还没有揭示这些模式之间的关系网络，还没有将这个系统与传统架构和其他AI系统进行对比，还没有深入反思这种架构对软件工程的长远意义。

这些都将在下篇中展开。

（待续：下篇将包含架构模式提炼、模式语言构建、对比研究和综合反思）