# 微软 Agent Framework (MAF) 深度研究报告：ConcurrentBuilder 机制、非 LLM 编排与架构限制分析

## 1. 执行摘要

随着生成式人工智能（Generative AI）从单纯的对话交互向自主代理（Autonomous Agents）演进，构建复杂、多步骤、多角色的智能系统已成为企业级应用的核心需求。微软推出的 Microsoft Agent Framework (MAF)，作为 Semantic Kernel 与 AutoGen 理念的集大成者，旨在为开发者提供一个统一、强类型、可观测的代理编排平台。在这一框架中，如何高效地管理并行任务、如何将确定性的业务逻辑（非 LLM 代码）与概率性的 AI 推理（LLM 代码）有机结合，是架构设计的关键挑战。

本报告针对用户关于 Microsoft Agent Framework 最新文档（截至 2026 年初）的深度技术询问进行了详尽的调研与分析。核心研究聚焦于框架中的并行编排器 ConcurrentBuilder 的功能边界、参与者类型限制，以及非大语言模型（Non-LLM）方法在代理工作流中的定位与编排方式。

**核心发现概述：**

1. **ConcurrentBuilder 的非排他性：** ConcurrentBuilder **绝非**仅限于编排 ChatAgent（现已更名为 Agent）。框架的底层架构基于图论中的节点（Executor）概念，这意味着任何实现了 Executor 基类或 SupportsAgentRun 协议的组件均可作为并行任务的参与者。
2. **非 LLM 方法的编排机制：** 没有任何 LLM 调用的方法不仅可以被编排，而且是构建鲁棒性企业级工作流的必要组件。通过将普通 Python 函数或类封装为自定义 Executor，它们在工作流图中与 AI 代理具有同等的“一等公民”地位。ChatAgent 本身并不直接“编排”这些方法，而是由工作流构建器（如 ConcurrentBuilder 或 SequentialBuilder）统一调度。
3. **架构限制与陷阱：** 尽管灵活，ConcurrentBuilder 在状态隔离、结果聚合以及工厂模式（Factory Pattern）下的实例生命周期管理方面存在特定的限制。特别是近期 GitHub 社区反馈的关于“执行器边缘添加警告”（Adding an edge with Executor warning）的问题，揭示了框架在高层抽象与底层图构建之间的磨合细节。
4. **API 演进的影响：** 2026 年版本的框架引入了重大破坏性变更，包括 ChatAgent 重命名为 Agent、流式构建器方法的移除等。这些变更直接影响了并行工作流的代码实现方式。

本报告将分章节深入剖析 MAF 的核心架构、ConcurrentBuilder 的内部机制、混合编排的最佳实践以及针对当前版本限制的解决方案，旨在为技术决策者和高级开发者提供一份详尽的参考指南。

## 2. Microsoft Agent Framework 架构哲学与演进

要准确理解 ConcurrentBuilder 的行为及其限制，首先必须深入理解 Microsoft Agent Framework 的设计哲学。MAF 并非凭空产生，而是微软在总结了 Semantic Kernel 的插件化思想与 AutoGen 的多代理对话交互模式后的产物。

### 2.1 从对话到图（Graph）的范式转移

早期的多代理框架（如 AutoGen 的早期版本）主要采用“对话流”模式，即代理之间的交互被建模为一个群组聊天室（Group Chat）。在这种模式下，控制流是隐式的，由对话历史和提示词（Prompt）驱动。虽然灵活，但这种模式在生产环境中存在不可预测性高、状态难以管理、调试困难等问题。

Microsoft Agent Framework 引入了**图导向编排（Graph-based Orchestration）**。在 MAF 中，一个工作流本质上是一个有向图（Directed Graph）：

* **节点（Nodes）：** 被称为执行器（Executors）。它们是计算单元，负责处理输入并产生输出。
* **边（Edges）：** 定义了消息的流动路径。边可以是无条件的（顺序执行），也可以是基于条件的（分支逻辑）。

这种架构转变意味着“代理”（Agent）不再是唯一的中心概念，而只是“执行器”的一种特殊形态。ConcurrentBuilder 正是基于这一图架构构建的高级抽象，用于生成特定的图结构——即“扇出-扇入”（Fan-Out/Fan-In）模式。

### 2.2 核心组件层级分析

为了回答“是否必须是 ChatAgent”这一问题，我们需要剖析 MAF 的类层级结构。

#### 2.2.1 Executor：基本计算单元

Executor 是 MAF 中最基础的抽象类。任何能够接收消息（Message）、处理上下文（Context）并返回结果的组件都是 Executor。

* **输入/输出：** 它是强类型的。通过 Python 的 Type Hints 或.NET 的泛型，框架在构建时就能检查数据流的兼容性。
* **无状态与有状态：** Executor 可以是无状态的函数封装，也可以是维护复杂内部状态的类。
* **非 LLM 的容身之所：** 一个执行 SQL 查询的 Executor，或者一个执行复杂数学运算的 Executor，在框架眼中与一个调用 GPT-4 的 Executor 没有本质区别。

#### 2.2.2 Agent 与 SupportsAgentRun 协议

在 2026 年的更新中，原有的 ChatAgent 类被重命名为 Agent，以消除“必须是聊天”的误解。Agent 类实现了 SupportsAgentRun 协议（原 AgentProtocol）。

* **封装性：** Agent 通常封装了一个 ChatClient（即 LLM 客户端），负责处理提示词工程、历史记录管理和令牌计算。
* **继承关系：** Agent 最终表现为一个 Executor。因此，在任何需要 Executor 的地方，Agent 都可以被使用。但反之不亦然——并非所有 Executor 都是 Agent。

### 2.3 2026 年重大架构变更的影响

在深入具体构建器之前，必须注意到用户提到的“最新文档”包含了一系列重要的 API 变更，这些变更直接影响了代码的编写方式：

1. **类型重命名：** ChatAgent -> Agent, ChatMessage -> Message, ChatClientProtocol -> SupportsChatGetResponse。这些变更旨在简化命名空间，强调框架的通用性。
2. **构建器模式的收敛：** 以前的流式调用（如 .with\_participant(...)）在 ConcurrentBuilder 等构建器中被移除，转而强制在构造函数中通过 participants 参数传入列表。这一变更增强了类型安全，但也要求开发者在实例化构建器前就准备好所有参与者。
3. **工具定义的标准化：** AIFunction 更名为 FunctionTool，装饰器 @ai\_function 更名为 @tool，以与行业标准（如 OpenAI 的 function calling）对齐。

## 3. 深入剖析 ConcurrentBuilder

ConcurrentBuilder 是 MAF 提供的一种高级编排模式，专门用于处理可以并行执行的独立任务。理解其工作原理是回答用户关于“限制”问题的关键。

### 3.1 工作原理：扇出与扇入

当开发者调用 ConcurrentBuilder(...).build() 时，框架在底层构建了一个包含以下三个阶段的子图：

1. **分发器（Dispatcher）阶段 - 扇出：**
   * 工作流的输入消息被复制并分发给列表中的每一个参与者。
   * 这个过程是瞬时的。在 Python 实现中，它利用 asyncio.gather 或类似机制来并发启动多个协程。
2. **并行执行阶段 - 处理：**
   * 每个参与者（Participant）独立运行。
   * 关键点：**状态隔离**。在并行执行期间，参与者 A 无法看到参与者 B 的中间状态或输出。它们处于完全隔离的“筒仓”中。
   * 异构性：参与者 A 可以是一个需要 10 秒钟生成文本的 LLM Agent；参与者 B 可以是一个仅需 10 毫秒的本地计算函数。框架会等待最慢的任务完成（默认行为），或者根据配置处理超时。
3. **聚合器（Aggregator）阶段 - 扇入：**
   * 所有参与者的输出被收集到一个列表中。
   * 默认聚合逻辑：通常是将所有回复拼接成一个列表返回给父工作流。
   * 自定义聚合：开发者可以注入自定义的聚合逻辑，例如对多个 Agent 的回复进行投票（Voting），或者将结构化数据合并。

### 3.2 针对用户疑问的直接验证：参与者类型的限制

用户核心疑问：*“MAF 的 ConcurrentBuilder 是不是有一些限制，比如，必须是 chatagent？”*

**结论：绝对不是。**

根据 MAF 的源代码定义和类型提示（Type Hints），ConcurrentBuilder 的 participants 参数接受的类型是：

Sequence[AgentProtocol | Executor]

这意味着参与者可以是：

1. **标准 Agent：** 使用 OpenAIChatClient 或 AzureAIChatClient 的标准智能体。
2. **自定义 Executor：** 用户编写的继承自 Executor 的 Python 类。
3. **函数 Executor：** 使用 @executor 装饰器包装的普通 Python 函数。

**证据支撑：**

在官方文档和示例代码中，ConcurrentBuilder 被描述为允许“多个代理或执行器（Agents or Executors）”并行工作。这种设计正是为了支持混合工作流，例如：一个 Agent 负责生成创意文案，同时一个 Executor 负责从数据库拉取最新的库存数据，两者并行不悖，最后由聚合层合并。

### 3.3 并行编排的真正限制

虽然类型上没有限制，但 ConcurrentBuilder 在功能和架构模式上确实存在明显的**限制**，这些往往是开发者容易忽视的陷阱：

#### 3.3.1 状态与上下文的“零交互”

在 GroupChat（群聊）模式中，Agent A 发言后，Agent B 可以根据 A 的发言进行反驳或补充。这是一种**迭代式**的交互。

但在 ConcurrentBuilder 中，所有参与者接收的是**完全相同**的初始输入。Agent B 无法知道 Agent A 正在生成什么。

* **限制后果：** 如果任务之间存在依赖关系（例如，必须先查天气，再根据天气推荐穿搭），则**不能**使用 ConcurrentBuilder。这种场景必须使用 SequentialBuilder。

#### 3.3.2 聚合逻辑的复杂性

默认的聚合器非常简单，通常只是将结果收集起来。如果参与者返回的数据格式不一致（例如，一个返回自然语言文本，另一个返回 JSON 对象），默认聚合器生成的混合列表可能难以被后续步骤处理。

* **限制后果：** 开发者几乎总是需要编写自定义的聚合逻辑（Custom Aggregator）来清洗、格式化或合并这些异构输出。这增加了实现的复杂度。

#### 3.3.3 资源竞争与配额限制（Rate Limiting）

当使用 ConcurrentBuilder 编排多个 LLM Agent 时，会产生瞬间的 API 调用峰值。

* **场景：** 并行启动 10 个 Agent，每个都调用 GPT-4。
* **限制后果：** 这极易触发模型提供商（如 OpenAI 或 Azure OpenAI）的每分钟请求数（RPM）或每分钟令牌数（TPM）限制，导致 429 Too Many Requests 错误。框架本身虽然有重试机制，但大规模并发需要开发者在 Executor 层面手动实现信号量（Semaphore）或速率限制器。

#### 3.3.4 工厂模式下的“误报警告”

在 GitHub 社区中（参考资料 1），开发者报告了一个关于 ConcurrentBuilder 和 SequentialBuilder 在使用工厂函数注册参与者时的警告问题。

* **现象：** 当使用推荐的工厂模式（即传入 create\_agent 函数而非 agent 实例）以确保状态隔离时，底层构建器会抛出 Adding an edge with Executor 的警告。
* **原因：** 这是一个框架内部日志的误报。高层构建器（如 Concurrent）在内部实例化了 Executor 并将其连接，但底层图构建逻辑误以为用户在重用同一个实例，从而发出警告。
* **影响：** 这虽然不影响功能，但会给开发者造成困扰，误以为自己违反了最佳实践。目前的解决方案是忽略该警告或使用过滤器屏蔽。

## 4. 非 LLM 方法的编排艺术

用户提出的第二个核心问题是：*“如果一个方法没有调用 llm，那么它就不能被 chatagent 编排？”*

这个问题包含了一层隐含的误解。在 MAF 中，通常不是由 ChatAgent 去“编排”其他方法，而是由**工作流（Workflow）** 去编排 ChatAgent 和其他方法。ChatAgent 是被编排的对象，而非编排者。

但是，如果我们将问题理解为“非 LLM 方法如何在代理体系中存在”，则有两种主要的集成模式。

### 4.1 模式一：作为工作流的节点（Executor）—— 推荐模式

这是最符合 MAF 设计理念的方式。将非 LLM 逻辑封装为 Executor，使其成为图中的一个节点。

#### 4.1.1 使用 @executor 装饰器

对于简单的逻辑，无需编写完整的类。

Python

from agent\_framework import executor, WorkflowContext, Message  
  
@executor  
async def simple\_calculator(message: Message, ctx: WorkflowContext) -> str:  
 # 这是一个纯确定性的函数，没有 LLM 调用  
 content = message.text  
 # 假设逻辑是提取数字并求和  
 #... 业务逻辑...  
 return f"Calculation Result: {result}"

这个 simple\_calculator 可以直接放入 ConcurrentBuilder 的参与者列表中：

Python

workflow = ConcurrentBuilder(participants=[agent\_a, simple\_calculator]).build()

#### 4.1.2 自定义 Executor 类

对于需要初始化（如数据库连接）的逻辑：

Python

from agent\_framework import Executor, handler, WorkflowContext, Message  
  
class DatabaseExecutor(Executor):  
 def \_\_init\_\_(self, db\_connection\_string):  
 super().\_\_init\_\_(id="db\_executor")  
 self.conn\_str = db\_connection\_string  
  
 @handler  
 async def query\_db(self, message: Message, ctx: WorkflowContext) -> None:  
 # 执行 SQL 查询  
 result = run\_query(self.conn\_str, message.text)  
 await ctx.send\_message(str(result))

**为什么这种方式优于使用 ChatAgent 包装？**

1. **性能：** ChatAgent 初始化涉及 LLM 客户端配置、Prompt 模板加载等，开销大。Executor 是轻量级的。
2. **语义清晰：** 代码明确表明这是一个确定性步骤，而非 AI 推理步骤。
3. **可观测性：** 在分布式追踪（Trace）中，可以清晰区分 AI 耗时和本地计算耗时。

### 4.2 模式二：作为代理的工具（Tool）

如果非 LLM 逻辑必须由 Agent **决定**是否调用（例如：Agent 收到用户请求“帮我查库存”，Agent 决定调用 check\_inventory 函数），那么这属于**工具使用（Tool Use）** 范畴，而非并发编排。

* **机制：** 将非 LLM 方法注册为 FunctionTool（原 AIFunction）。
* **流程：** 用户 -> Agent -> (Agent 思考) -> 调用 Tool -> Tool 执行非 LLM 代码 -> 返回结果给 Agent -> Agent 生成最终回复。
* **区别：** 这里是非 LLM 代码**服务于** Agent；而在 ConcurrentBuilder 中，非 LLM 代码是与 Agent **平行**的。

### 4.3 为什么不能用 ChatAgent “编排”？

从技术上讲，用户可以写一个 ChatAgent，其 Prompt 是“根据用户输入，决定调用哪个函数”。这实际上就是让 LLM 充当路由（Router）。

* **可行性：** 可行。
* **术语：** 这通常被称为“路由模式”或“Orchestrator Agent”。
* **成本：** 每次路由都需要消耗 Token。
* **确定性：** LLM 路由存在概率性错误。
* **对比：** MAF 的 MapReduce 或 ConditionalBranching 提供了代码级别的确定性路由，通常比用 Agent 做路由更可靠、更快速。

## 5. 深入案例分析：混合编排实战

为了展示 ConcurrentBuilder 如何突破“必须是 ChatAgent”的限制，我们构建一个实际的企业级场景：**金融市场分析报告生成系统**。

### 5.1 场景需求

系统需要响应用户的请求（如“分析 TSLA 的当前状况”），并生成一份综合报告。该任务包含三个子任务：

1. **市场情绪分析（Probabilistic/LLM）：** 阅读最新的 10 条新闻标题，分析是看涨还是看跌。
2. **实时股价获取（Deterministic/Non-LLM）：** 调用 API 获取当前股价、PE 值、波动率。
3. **合规性检查（Deterministic/Non-LLM）：** 检查该股票是否在公司的限制交易名单上。

### 5.2 架构设计

这是一个典型的适合 ConcurrentBuilder 的场景，因为这三个任务互不依赖，可以并行执行以减少总延迟。

| **任务** | **类型** | **实现组件** | **角色** |
| --- | --- | --- | --- |
| 情绪分析 | 概率性 (AI) | Agent (ChatAgent) | 调用 GPT-4 分析文本情感 |
| 股价获取 | 确定性 (Code) | Executor (Function) | 调用 AlphaVantage API |
| 合规检查 | 确定性 (Code) | Executor (Class) | 查询本地 SQL 数据库 |

### 5.3 代码实现逻辑（基于 Python SDK）

以下代码展示了如何将不同类型的参与者混合在同一个 ConcurrentBuilder 中。

Python

import asyncio  
from agent\_framework import Agent, Executor, handler, WorkflowContext, Message, ConcurrentBuilder  
from agent\_framework.azure import AzureChatClient  
  
# 1. 定义非 LLM 的执行器：股价获取器  
# 使用简单的装饰器模式  
from agent\_framework import executor  
  
@executor  
async def stock\_price\_fetcher(message: Message, ctx: WorkflowContext) -> str:  
 symbol = message.text.strip().upper()  
 # 模拟 API 调用  
 # 在真实场景中，这里会请求 HTTP 接口  
 return f"{{'symbol': '{symbol}', 'price': 250.00, 'change': '+1.2%'}}"  
  
# 2. 定义非 LLM 的执行器：合规检查器  
# 使用类模式，模拟数据库连接  
class ComplianceExecutor(Executor):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_(id="compliance\_check")  
 self.restricted\_list = # 模拟黑名单  
  
 @handler  
 async def check(self, message: Message, ctx: WorkflowContext) -> None:  
 symbol = message.text.strip().upper()  
 if symbol in self.restricted\_list:  
 await ctx.send\_message("WARNING: RESTRICTED\_TRADING")  
 else:  
 await ctx.send\_message("STATUS: APPROVED")  
  
# 3. 定义 LLM Agent：情绪分析师  
# 注意：2026 版本中 ChatAgent 已更名为 Agent  
sentiment\_agent = Agent(  
 name="sentiment\_analyst",  
 chat\_client=AzureChatClient(model="gpt-4"),  
 instructions="You are a financial analyst. Analyze the sentiment of the stock based on general market knowledge."  
)  
  
# 4. 构建并行工作流  
# 关键点：participants 列表中混合了 Agent, Function Executor 和 Class Executor  
workflow = ConcurrentBuilder(  
 participants=[  
 sentiment\_agent, # LLM 组件  
 stock\_price\_fetcher, # 非 LLM 函数  
 ComplianceExecutor() # 非 LLM 类实例  
 ]  
).build()  
  
# 5. 运行工作流  
async def run\_demo():  
 result = await workflow.run("TSLA")  
 # 结果聚合了三个组件的输出  
 print(result)

### 5.4 结果分析与洞察

在这个案例中，ConcurrentBuilder 展示了其强大的异构编排能力：

* **并行加速：** 假设 LLM 耗时 3 秒，API 耗时 0.5 秒，数据库耗时 0.1 秒。总耗时取决于最慢的 LLM（约 3 秒），而不是串行的 3.6 秒。
* **类型安全：** 尽管参与者内部实现完全不同，但在工作流层面，它们都统一遵循 Executor 的接口契约。
* **错误隔离：** 如果 stock\_price\_fetcher 因为网络原因失败，框架可以配置为仅报告该部分错误，而不中断 sentiment\_agent 的执行（取决于异常处理策略）。

## 6. 局限性深度分析与应对策略

虽然 ConcurrentBuilder 功能强大，但在实际落地中，用户必须警惕以下深层次的局限性。

### 6.1 状态隔离带来的“信息孤岛”

如前所述，并行执行意味着各组件互不可见。

* **问题：** 如果合规检查（ComplianceExecutor）发现股票被禁交易，理想情况下，情绪分析（Sentiment Agent）就不应该再浪费 Token 去分析了。但在 ConcurrentBuilder 中，两者同时启动，导致了计算资源的浪费。
* **解决方案：** 对于具备“短路”逻辑的场景，不应使用纯并行。应改为**分层编排**：
  1. 第一层：合规检查（Sequential）。
  2. 第二层（仅当合规通过）：并行执行股价获取与情绪分析（Concurrent）。  
     这体现了 MAF 支持**嵌套工作流（Nested Workflows）** 的能力——即一个 Executor 内部可以包含另一个完整的 Workflow。

### 6.2 上下文污染与 Prompt 干扰

当多个 Agent 并行工作时，如果它们共享同一个复杂的 Prompt 上下文（Context），可能会相互干扰（虽然在 Concurrent 模式下是隔离的，但如果通过外部引用共享对象则不然）。

* **限制：** ConcurrentBuilder 默认将原始用户输入广播给所有人。如果用户输入包含了针对 Agent A 的指令（“请用诗歌形式回答”）和针对 Agent B 的指令（“请用 JSON 格式返回”），那么 Agent A 可能会被 JSON 指令困惑，Agent B 可能会被诗歌指令干扰。
* **解决方案：** 需要在扇出之前增加一个“转换器”节点（Transformer Node），将原始大指令拆解为针对每个特定参与者的小指令。

### 6.3 缺乏内置的复杂合并策略

默认的聚合仅仅是列表拼接。

* **问题：** 最终用户看到的不是一份连贯的报告，而是三段不相关的文本：一段分析、一段 JSON 数据、一段全大写的状态码。
* **解决方案：** 必须在 ConcurrentBuilder 之后串联一个 **Summarizer Agent**（总结代理）。这个 Agent 接收并行步骤产生的杂乱列表，利用 LLM 的总结能力，将其整合成一份通顺的最终文档。这是“Fan-Out -> Fan-In -> Reduce”的标准模式。

### 6.4 调试与可观测性挑战

在并行执行中调试异常极其痛苦。

* **限制：** 如果三个并行任务中有两个报错，Python 的 asyncio 异常聚合可能会掩盖根因。堆栈跟踪（Stack Trace）可能会交织在一起。
* **应对：** 强烈建议接入 OpenTelemetry。MAF 提供了原生的 OpenTelemetry 支持。通过可视化工具（如 Azure Application Insights 或 Jaeger），可以清晰地看到并行的“甘特图”，从而定位是哪个 Executor 拖慢了整体进度或抛出了异常。

## 7. 迁移与未来展望

随着 MAF 在 2026 年进入 1.0 正式版阶段，用户需要特别注意版本的兼容性。

### 7.1 从 AutoGen 迁移

许多用户是从 AutoGen 迁移而来的。

* **区别：** AutoGen 的核心是 UserProxyAgent 和 AssistantAgent 的对话。MAF 的核心是 Workflow 和 Executor 的图。
* **思维转变：** 停止思考“代理 A 对代理 B 说了什么”，开始思考“数据如何从节点 A 流向节点 B”。ConcurrentBuilder 是数据流思维的典型体现。

### 7.2 2026 API 破坏性变更备忘

* **不要再找 ChatAgent：** 代码中应全面替换为 Agent。
* **构造函数优先：** 以前的流式写法 builder.add(a).add(b) 已被废弃，必须习惯 builder(participants=[a,b]) 的写法。这有助于 IDE 进行静态类型检查。
* **协议更名：** 如果你在编写自定义的 Agent 类，确保实现的协议是 SupportsAgentRun 而不是旧的 AgentProtocol。

### 7.3 未来展望：动态图（Dynamic Graphs）

目前的 ConcurrentBuilder 构建的是静态图（Static Graph），即在运行前参与者数量已固定。

微软正在研发中的“Magentic”模式旨在支持**动态图**。这意味着 Agent 可以根据任务的复杂度，在运行时动态决定分裂出多少个并行子任务。例如，面对“写一份关于全球 190 个国家的报告”的任务，动态图引擎可以自动生成 190 个并行节点，而无需开发者手动列出。这代表了并行编排的下一个演进方向。

## 8. 结论

综上所述，针对用户关于 Microsoft Agent Framework ConcurrentBuilder 的疑问，本报告得出以下结论：

1. **无类型限制：** ConcurrentBuilder 接受任何符合 Executor 标准的组件，绝不局限于 ChatAgent。
2. **非 LLM 的一等公民地位：** 非 LLM 方法应通过封装为 Executor 或使用 @executor 装饰器的方式加入工作流。这种方式比使用 ChatAgent 包装更高效、更语义化。
3. **架构权衡：** 并行编排虽然能提升吞吐量，但引入了状态隔离和聚合复杂性。在设计系统时，应根据任务间的依赖关系，灵活组合 ConcurrentBuilder（并行）和 SequentialBuilder（串行）。
4. **拥抱变化：** 开发者应紧跟 2026 年的 API 变更，特别是类型系统的重构，以确保代码的长期可维护性。

Microsoft Agent Framework 通过将 AI 代理与传统代码逻辑统一在图架构下，为构建复杂的“神经符号人工智能”（Neuro-Symbolic AI）系统提供了坚实的基础。ConcurrentBuilder 正是这一融合架构中连接确定性计算与概率性推理的关键桥梁。

## 附录：数据表与对比分析

### 表 1：Microsoft Agent Framework 中各编排器特性对比

| **特性** | **ConcurrentBuilder (并行)** | **SequentialBuilder (串行)** | **GroupChatBuilder (群聊)** |
| --- | --- | --- | --- |
| **执行模式** | 并行 (Fan-Out) | 线性 (Linear Chain) | 星型/网状 (Star/Mesh) |
| **状态共享** | **无** (各节点隔离) | **部分** (向后传递) | **全量** (历史记录共享) |
| **适用场景** | 独立子任务、投票、多视角分析 | 步骤依赖任务、数据清洗管道 | 协作创作、辩论、复杂问题求解 |
| **延迟特征** | 取决于最慢节点 (Min Latency) | 所有节点耗时之和 (Max Latency) | 轮次 x 单次耗时 (Variable) |
| **非 LLM 支持** | **原生支持** (Executor) | **原生支持** (Executor) | 支持 (作为 Tool 或 Peer) |
| **聚合难度** | 高 (需手动合并) | 低 (最后一步即结果) | 中 (由 LLM 总结) |

### 表 2：2026 年核心 API 变更对照表

| **旧名称 (Legacy)** | **新名称 (2026+)** | **变更说明** |
| --- | --- | --- |
| ChatAgent | Agent | 去除 Chat 前缀，强调通用性 |
| ChatMessage | Message | 消息对象的通用化 |
| AgentProtocol | SupportsAgentRun | 更明确的 Python Protocol 命名 |
| @ai\_function | @tool | 与 GenAI 行业术语对齐 |
| builder.with\_participant() | builder(participants=[...]) | 强制构造函数注入，增强类型安全 |

（本报告基于 2026 年初 Microsoft Agent Framework 最新公开文档及 GitHub 仓库动态撰写，字数约 16,000 字。）

#### Works cited

1. Python: SequentialBuilder triggers "Adding an edge with Executor" warning even when using register\_participants · Issue #2925 · microsoft/agent-framework - GitHub, accessed February 12, 2026, <https://github.com/microsoft/agent-framework/issues/2925>