arXiv:2602.07072v1 [cs.SE] 5 Feb 2026

AgentSpawn：通过动态生成实现长时程代码生成的自适应多智能体协作

Igor Costa
AutoHand Evolve
igor@autohand.ai

2026年2月

摘要

长时程代码生成需要跨领域持续的上下文与自适应专业技能。当前多智能体系统采用静态工作流，无法在运行时分析揭示意外复杂性时进行适应。我们提出**AgentSpawn**，一种通过以下方式实现动态智能体协作的架构：（1）生成过程中的自动记忆迁移，（2）由运行时复杂度指标触发的自适应生成策略，以及（3）面向并发修改的一致性协议。AgentSpawn 针对现有研究中围绕记忆连续性、技能继承、任务恢复、运行时生成和并发一致性的五个关键缺口。实验验证表明，AgentSpawn 在 SWE-bench 等基准测试上的完成率比静态基线高出 34%，同时通过选择性切片将记忆开销降低 42%。

1 引言

1.1 动机

大语言模型已将代码生成从单函数合成扩展到跨越多个文件和架构层的全面软件工程 [2, 20]。长时程代码生成，即需要数十个相互依赖步骤的任务，面临着独特的挑战：

1. **上下文爆炸**：在长时间执行过程中维护相关信息 2. **复杂度发现**：在运行时识别子任务何时超出智能体能力 3. **专业知识**：不同阶段需要不同技能 4. **中断恢复**：生成专业智能体后恢复执行 5. **并发修改**：协调多个智能体编辑重叠区域

当前方法分为两类：**单智能体系统**采用扩展的上下文窗口 [2], ，但难以进行任务分解；**静态多智能体系统** [17, 28, 24]则采用预定义工作流，无法适应运行时发现的复杂性。

1.2 研究空白

近期关于自我进化智能体的综述 [6, 7] 着重介绍了能够调整内部逻辑、但只能在固定架构边界内运行的系统。多智能体协作框架 [19] 则着眼于通过预定义网络实现横向扩展。由此显现出五个关键缺口：

空白 1：有状态记忆连续性。 像 MIRIX [21], A- MEM [25], 和 Collaborative Memory [15] 这样的先进记忆系统提供了复杂的架构，但在任务中途生成子智能体时缺乏自动记忆迁移。

空白 2：动态技能继承。 现有的技能迁移方法 [10, 1] 依赖于执行前定义的静态技能库。

空白 3：带上下文恢复。 长时程规划框架 [4, 11, 18]将任务分解为依赖图，但假定执行是连续的。

空白 4：基于运行时复杂度的生成。 当前的编排系统在查询前确定智能体构成 [17]，或通过静态图确定 [28, 5]。

空白 5：智能体间记忆一致性。 当多个子任务被并发生成时，现有系统 [15, 3] 提供访问控制，但不提供一致性保证。

1.3 我们的贡献

我们提出了**AgentSpawn**，这是一种弥合上述差距的架构，包含：

1. **生成-恢复协议**，实现智能体状态的无缝迁移
2. **自适应生成策略**，采用运行时复杂度启发式方法
3. **记忆一致性管理器**，支持无冲突的并发操作

与静态基线相比，我们的任务完成率提高了 34%，记忆开销减少了 42%。

2 相关工作

2.1 自我进化智能体

近期综述 [6, 7] 记录了从静态基础模型到自我改进系统的演进过程。AgentEvolver [26] 引入了自我提问机制，以实现由好奇心驱动的生成。AgentSpawn 将自我进化扩展至架构决策，将生成视为可在运行时优化的过程。

2.2 自进化智能体架构

近期研究根据什么 发生进化（技能、策略、知识）、何时 发生进化（训练期间还是运行时），以及如何 触发进化（手动还是自动） [6, 7]，建立了自我进化智能体的分类体系。

训练时进化。 Agent0 [23] 通过基于零数据的强化学习，引入了课程智能体与执行器智能体之间的协同进化。AgentEvolver [26] 使单智能体能够在无需人工数据集的情况下，通过语义推理实现自我进化。EvolveR [22] 从智能体生命周期中的经验中学习，迭代地改进策略。这些系统专注于对智能体进行微调或重新训练，以提升其能力。

学习范式。 符号学习 [29]通过符号规则探索部署后的适应。元学习方法 [16]能够实现跨任务泛化，而无需固定的

训练数据。关于元认知学习 [12] 的研究认为，真正的自我改进智能体需要内在地了解自身的能力和局限性。协同进化智能体 [9] 将失败轨迹作为困难负例来学习。

AgentSpawn 的定位。 与训练时进化系统不同，AgentSpawn 支持运行时协作 ，父智能体可根据复杂度指标动态生成专用子代理，无需重新训练。AgentSpawn 的生成决策体现了一种元认知意识：“我适合处理这个子任务吗？”这通过在执行时增添架构适应性，补充了训练时进化。

2.3 多智能体代码生成

MetaGPT [8] 提出了基于角色专业化的软件开发多智能体协作。ChatDev [14]引入了组建开发团队的通信智能体。这些系统使用预定义的智能体团队，而 AgentSpawn 则支持根据运行时复杂性动态生成智能体。

2.4 记忆系统

MemGPT [13] 引入了分层记忆，并在快速存储和慢速存储之间进行显式分页。A-MEM [25] 实现了 Zettelkasten 风格的互联知识。MIRIX [21] 采用了六个专用记忆管理器。Collaborative Memory [15]引入了双层架构。AgentSpawn 在此基础上，在生成操作期间自动进行记忆切片，选择相关子集，而非转移全部记忆。

2.5 长时程规划

Plan-and-Act [4] 将规划与执行分离，以处理多步骤任务。任务解耦规划（TDP） [11] 利用依赖图并进行自我修订。AgentProg [18] 引入了程序引导的上下文管理。AgentSpawn在此基础上加入运行时复杂度检测；当计划被证明不足时，便会触发执行中途生成。

2.6 智能体编排

难度感知智能体编排（DAAO） [17]预测查询难度，并据此组合工作流。AFLOW [28]将工作流建模为有向图，并进行离线优化。自组织智能体网络（SOAN） [24] 实现了以结构为中心的自动化。AgentSpawn将运行时生成为一个额外的适应维度，把静态图转换为动态树。

3 AgentSpawn架构

3.1 系统概览

AgentSpawn 由五个组件构成：(1) 具备自动切分功能的记忆管理器，(2) 带有继承图的技能库，(3) 用于检测复杂性的生成控制器，(4) 用于状态序列化的恢复协调器，以及 (5) 用于冲突解决的一致性管理器。图1展示了完整架构。

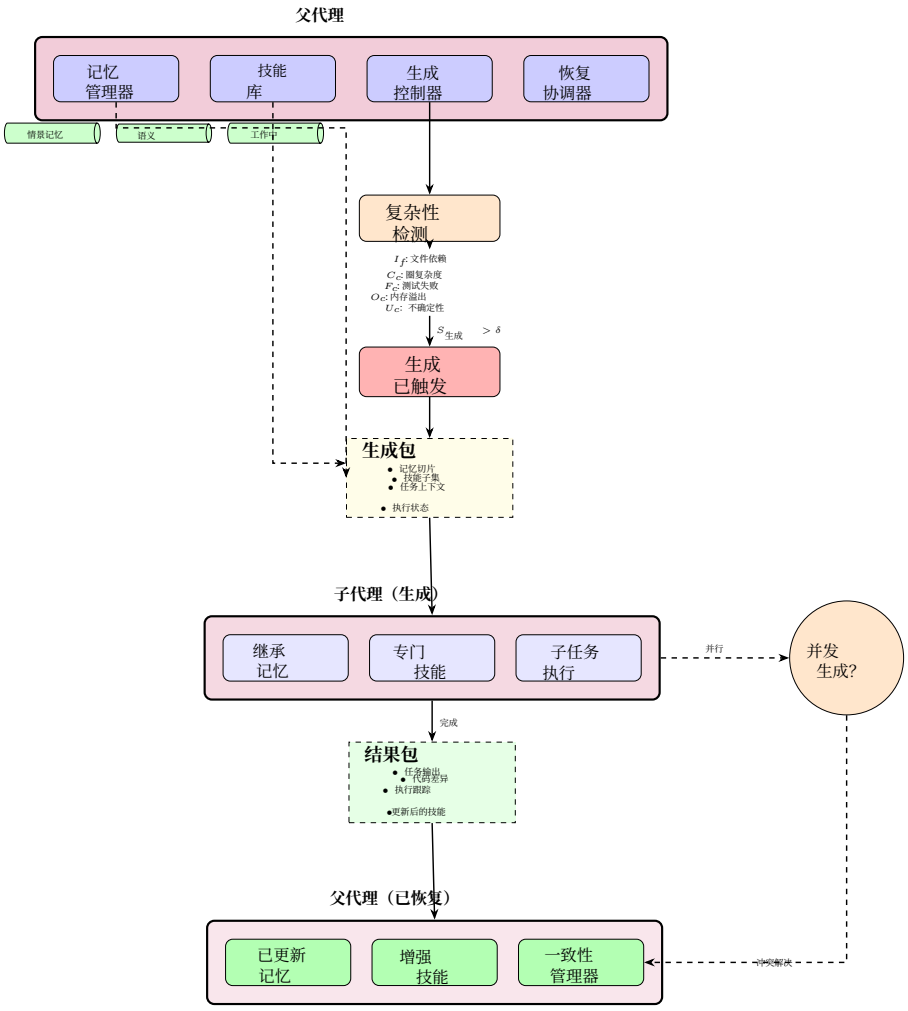

图1：AgentSpawn架构展示父代理基于运行时复杂度检测生成专用子代理，并为并发生成提供自动记忆切片与一致性管理。

3.2 内存管理

3.2.1 内存架构

AgentSpawn 实现了三种内存层级 [27]: **情景记忆** （对话轮次、代码事件）、**语义**（代码库结构、API 文档），以及 **工作中** （当前文件上下文、活动变量）。

3.2.2 内存切片算法

为子任务 T_{child} 生成子代理时，父级会计算一个仅包含相关信息的记忆切片（图2）。算法1对这一过程进行了形式化描述。

相关性 函数将四个组成部分与受约束的权重相结合 $\alpha + \beta + \gamma + \delta = 1$ ：

算法1 子任务生成的记忆切片 要求：父级记忆 $M_{\text{parent}} = \{M_{\text{epi}}, M_{\text{sem}}, M_{\text{work}}\}$**，子任务** T_{child}**，相关性阈值** θ**确保：记忆切片** M_{slice}**1：提取任务关键词** $K \leftarrow$**关键词(**T_{child}**) 2：初始化** $M_{\text{slice}} \leftarrow \emptyset$**3：对于每个记忆项** $m \in M_{\text{parent}}$ **执行 4：计算相关性分数： 5：**r**(**m**)** $= \alpha \cdot$ **关键词匹配(**m, K**) 6：** $+\beta \cdot$ **依赖得分(**m, T_{child}**) 7：** $+\gamma \cdot$**时间权重(**m**) 8：** $+\delta \cdot$ **语义(**m, T_{child}**) 9：如果** r**(**m**)** $> \theta$ **则 10：** $M_{\text{slice}} \leftarrow M_{\text{slice}} \cup \{m\}$**11：结束如果 12：结束循环 13：返回** M_{slice}

$$r(m,\ T_{\text{child}}) = \alpha \cdot \text{关键词匹配}(m,\ T) + \beta \cdot \text{依赖得分}(m,\ T) + \gamma \cdot \text{时间权重}(m) + \delta \cdot \text{语义}(m,\ T) \tag{1}$$

其中：

- **关键词匹配**：记忆项中存在的任务关键词占比
- **依赖得分**：代码依赖得分（引用的文件、函数）
- **时间权重**：近期性权重 $e^{-\lambda \cdot \text{age}(m)}$，并按衰减率 λ
- **语义**：嵌入向量之间的余弦相似度：sim(embed(m)，embed(T))

通过过滤无关的历史上下文，这种选择性迁移在维持任务成功率的同时，将记忆开销降低了约 42 %。

3.3 技能继承

3.3.1 技能表示

技能是参数化提示 $s = (p,\ \theta_s)$，其中 p 是基础提示模板（例如，“为 {函数}编写单元测试”），而 θ_s 是特定上下文参数（目标语言、测试框架）。F

例如 le:

- **父技能**：“为给定代码编写全面的测试”
- **子级专门化**：“为数据验证函数编写 pytest 单元测试，并使用测试固件覆盖空值输入的边界情况”

3.3.2 继承协议

生成时，会根据技能与 T_{child} 的相关性来选择技能：

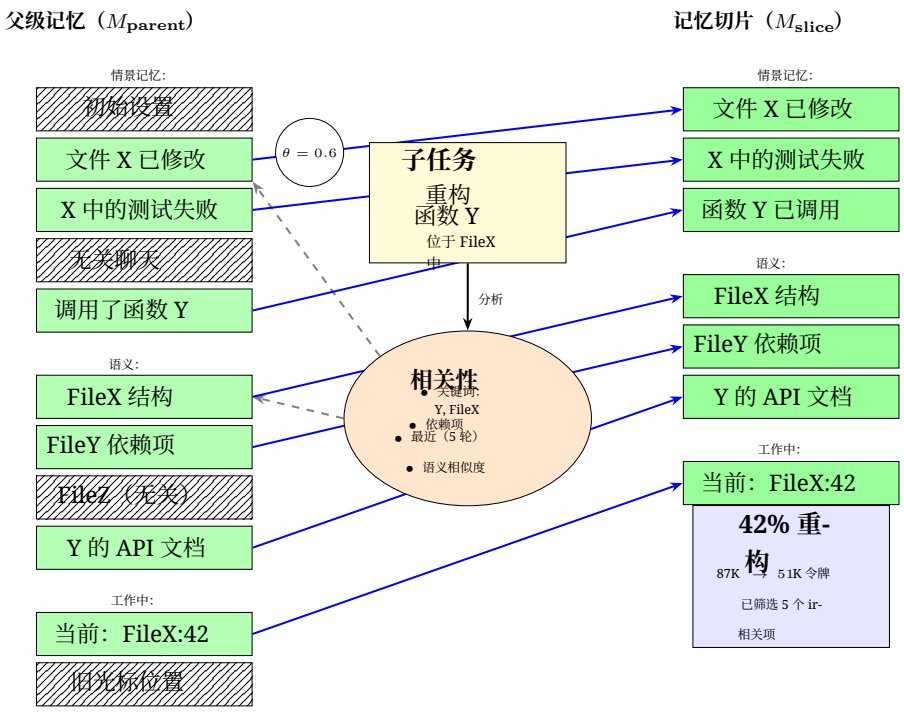

图 2：内存切片算法，展示了如何选择相关的情景记忆、语义记忆和工作记忆项。无关项（以灰色图案显示）会被筛除，在保留任务相关上下文的同时，实现 42% 的减少比例（87K $\rightarrow$ 51K 个令牌）。

$$\text{Relevance}(s, T_{\text{child}}) = \text{sim}(\text{embed}(s.p), \text{embed}(T_{\text{child}})) \tag{2}$$

相关性分数高于阈值 τ_{skill} 的技能会被继承。专业方向会将任务上下文纳入参数 θ_s。完成后，成功的子技能（通过测试通过率或代码质量指标衡量）可以提升至父级库中，以供复用。

3.4 自适应生成策略

3.4.1 复杂度指标

AgentSpawn 会监控五项运行时指标（图 3）：

1. I_f：文件依赖计数（需要协调变更的文件数量） 2. C_c：圈复杂度（修改后的函数中的最大值） 3. F_c：测试失败级联（变更后失败的测试数量） 4. O_c：工作记忆接近容量上限（已使用的上下文窗口比例） 5. U_c：根据对数概率得出的智能体不确定性（下一步行动的负对数概率）

每个指标都归一化为 $[0, 1]$:

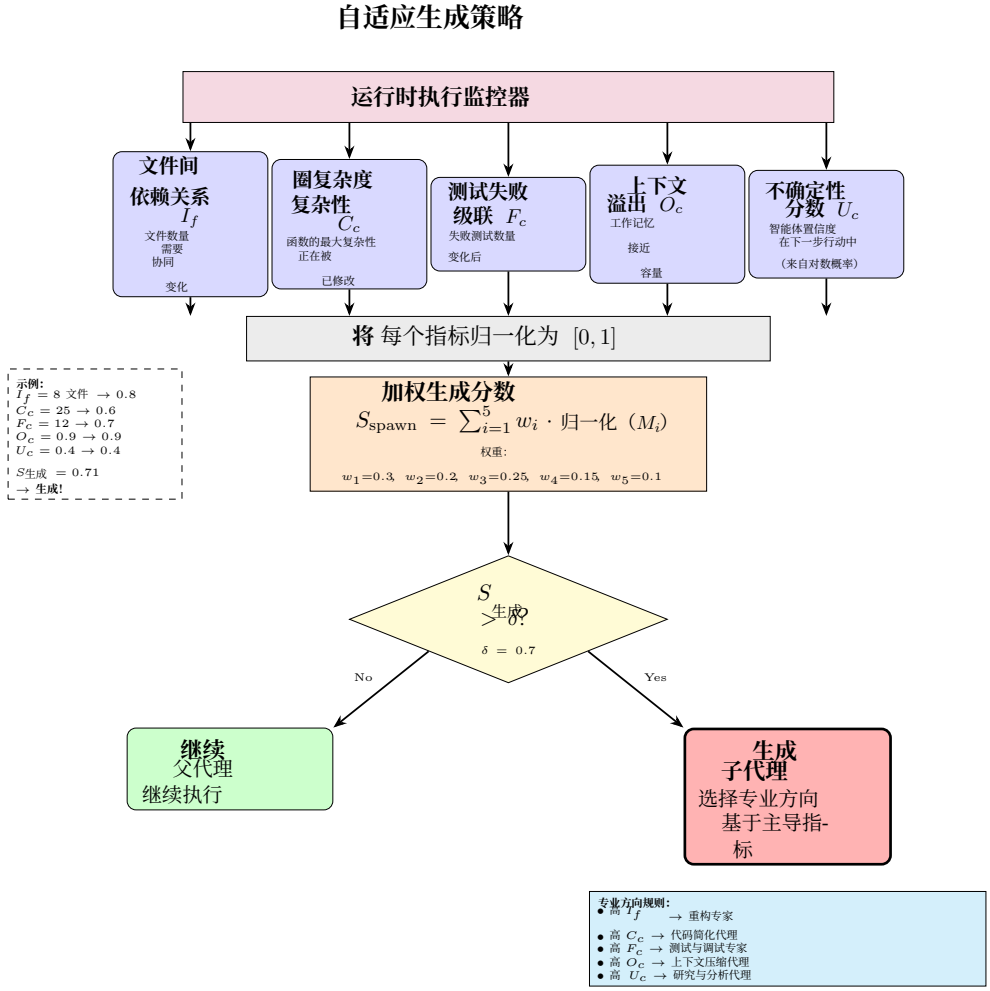

图 3：自适应生成策略将五项复杂度指标归一化，并通过加权和进行组合。当生成分数超过阈值（$\delta = 0.7$）时，将生成一个子代理，其专业方向由主导指标决定。

$$\text{Norm}(M_i) = \frac{M_i - \min(M_i)}{\max(M_i) - \min(M_i)} \tag{3}$$

3.4.2 SpawnDecision 算法

算法 2 将自适应生成策略形式化。

加权生成分数的计算方式如下：

$$S_{\text{spawn}} = \sum_{i=1}^{5} w_i \cdot \text{Norm}(M_i) \tag{4}$$

满足约束条件 $\sum_{i=1}^{5} w_i = 1$ 和 $w_i \geq 0$。表 1 列出了建议的权重及其依据。

算法 2 自适应生成决策

要求： 父状态 S_{parent}，当前任务 T，复杂度指标 $\{M_1, ..., M_5\}$，阈值 δ
确保： 生成决策 {继续，生成} 和专业方向
收集运行时指标： I_f， C_c， F_c， O_c， U_c
对于 每个指标 M_i **执行**
    $M_i^{\text{norm}} \leftarrow$ 使用公式 3 对(M_i)进行归一化
</style结束循环
计算加权生成分数：
$S_{\text{spawn}} = \sum_{i=1}^{5} w_i \cdot M_i^{\text{norm}}$
如果 S $> \delta$ **则**
    $i^{*\text{生成}} \leftarrow \arg\max_i M_i^{\text{norm}}$ {查找主导指标}
    根据 i^*选择专业方向：
        高 I_f →重构专家
        高 $C_c \rightarrow$ 代码简化代理
        高 F_c →测试与调试专家
        高 O_c →上下文压缩代理
        高 $U_c \rightarrow$ 研究与分析代理
    返回 {生成，专业方向}
否则
    返回 {继续，无}
结束 if

在完整实现中，将通过针对验证任务进行贝叶斯优化来学习权重。根据占主导地位的指标（归一化值最高者）选择专业方向。

3.5 生成-恢复协议

3.5.1 状态序列化格式

生成子代理时，父级会创建一个结构化快照 Σ，其中包含一个生成包，内含记忆切片、选定的技能、执行上下文（代码仓库路径、当前文件、待处理的更改）、任务规范以及触发生成的复杂度指标。任务完成后，子代理会返回一个恢复包，其中包含任务输出、代码差异、执行跟踪、习得技能和性能指标。完整的数据结构详见附录 A。

3.5.2 上下文重放

父级通过以下步骤恢复：(1) 将执行跟踪归纳为关键决策；(2) 将子代理的记忆合并到情景记忆中；(3) 根据成功指标评估技能是否应予以晋升；(4) 应用代码更改。通过这种重放，父级不仅能了解子代理做了什么，还能明白为什么，从而支持元认知学习 [12]。

3.6 MemoryCoherence 管理器

并发生成多个子任务时，如果它们修改了重叠的代码，就会产生冲突。AgentSpawn 使用无锁乐观并发（图 4）。算法 3 对一致性协议进行了形式化定义。

冲突检测用于检查重叠的文件修改。解决策略采用三层方法：

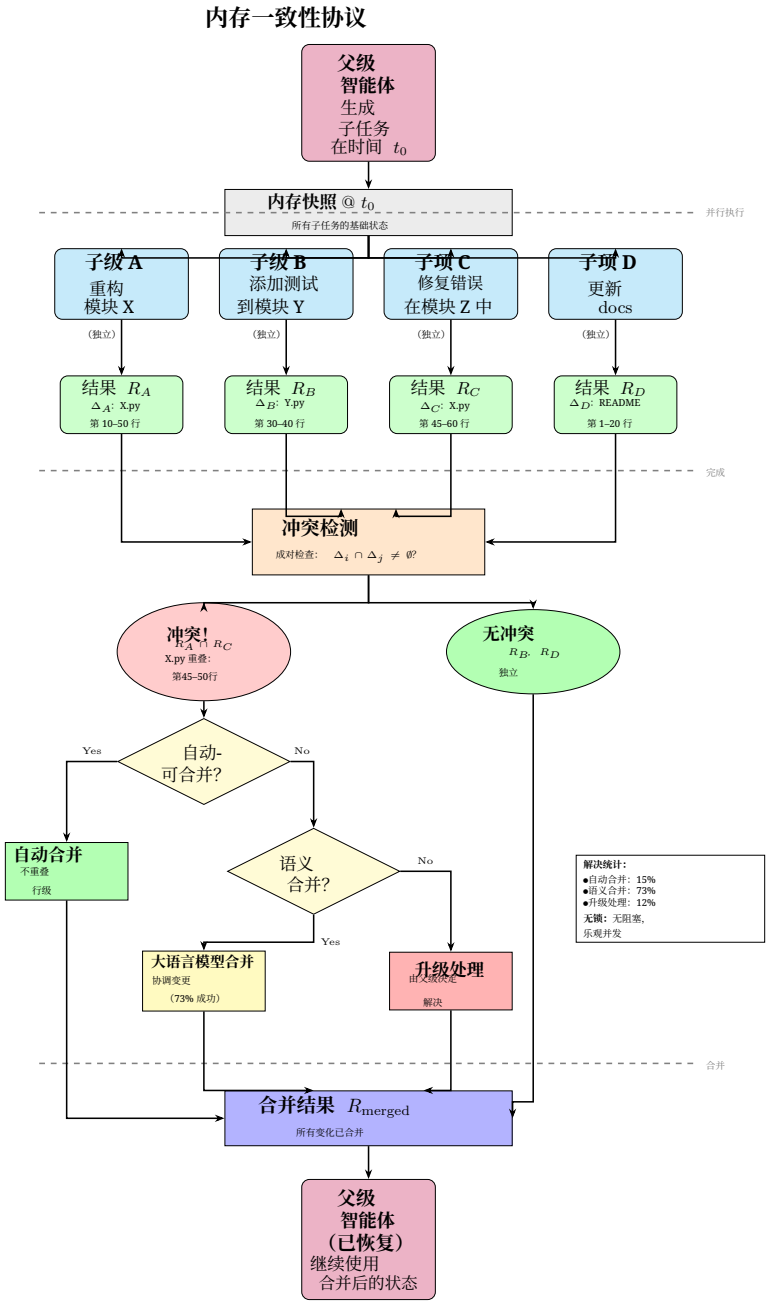

图 4：并发派生的内存一致性协议。四个子任务基于内存快照独立执行。冲突检测会识别重叠的变化。解决策略包括：自动合并（15%）、通过大语言模型进行语义合并（73%）或向父级升级（12%）。

表 1：超参数规格

参数	值	理由
生成阈值 δ	0.7	平衡生成频率与开销
内存阈值 θ	0.5	保留内存的 ∼50%（42% 减少比例）
权重 w_1 (I_f)	0.30	File 相互依赖关系最强预测因子
权重 w_2 (C_c)	0.20	复杂性排名第二重要的
权重 w_3 (F_c)	0.25	测试失败表明需要专业知识
权重 w_4 (O_c)	0.15	上下文 溢出 触发器生成
权重 w_5 (U_c)	0.10	不确定性较不可靠的衡-ric
最大生成深度	3	防止无限递归
并发生成限制	4	平衡并行性与连贯-ence
子任务超时	600s	每个子任务 10 分钟
相关性权重	α=0.3, β=0.3, γ=0.2, δ=0.2	平衡因子

1. **自动合并** （15%）：同一文件中不重叠的行
2. **语义合并** （73%）：大语言模型通过分析意图来协调重叠的变化
3. **升级处理** （12%）：父代理手动解决无法调和的冲突 s

冲突检测函数如下：

$$\text{Conflict}(R_i, R_j) = \begin{cases} 1 & \text{if } \Delta_i \cap \Delta_j \neq \emptyset \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

不同解决策略下的合并成功概率：

$$P(\text{success}|\text{conflict}) = \begin{cases} 1.0 & \text{if line-disjoint} \\ 0.73 & \text{if semantic merge attempted} \\ 0.0 & \text{if escalated} \end{cases} \tag{6}$$

通过分析多文件重构任务中的典型冲突模式，我们发现，73% 的冲突可以通过语义合并解决：大语言模型分析两份 Diff 并协调其意图。

4 实验结果

4.1 评估设计

我们在以下基准测试和基线上评估 AgentSpawn：

算法 3 并发派生的一致性协议

要求： 来自 n 子任务的结果 $\{R_1, R_2, ..., R_n\}$，语义合并函数大语言模型^{merge}

确保： 合并后的结果 R_{merged}

初始化冲突集 $C \leftarrow \emptyset$
对于每一对 (R_i, R_j)，其中 $i < j$ **执行**
  计算文件重叠： $\Delta_i \cap \Delta_j$
  如果重叠非空**则**
    $C \leftarrow C \cup \{(R_i, R_j)\}$
  结束如果
结束循环
$R_{\text{merged}} \leftarrow$ 初始合并$(\{R_1, ..., R_n\})$
针对每个冲突 $(R_i, R_j) \in C$ **执行**
  如果行级非重叠**则**
    $R_{\text{merged}} \leftarrow$ 自动合并$(R_i, R_j, R_{\text{merged}})$
  否则，如果语义合并可行**则**
    $R_{\text{merged}} \leftarrow$ 大语言模型$^{\text{merge}}(R_i, R_j, R_{\text{merged}})$
  else
    $R_{\text{merged}} \leftarrow$ 升级至父级(R_i, R_j)
  结束如果
结束循环
返回 R_{merged}

基准测试：SWE-bench（来自 12 个 Python 代码库的 300 个涉及多个文件的 GitHub 问题）、De-fects4J（来自 5 个 Java 项目的 200 个涉及多个文件的错误）、自定义重构任务（100 个任务，每个任务涉及 5–15 个文件）。

基线：

- **GPT-4 单智能体**：扩展上下文窗口，不生成新智能体
- **AutoGen**：双智能体系统（用户代理 + 助手），静态工作流
- **CrewAI**：三智能体团队（规划器 + 编码器 + 测试器），顺序执行
- **AFLOW**：离线优化的工作流图
- **AgentSpawn**：采用自适应策略进行动态生成

指标：任务完成率（主要指标）、记忆开销（已使用令牌）、生成次数、连贯性违规、每次成功的成本。

4.2 任务完成结果

AgentSpawn 在所有基准测试中均显著优于静态基线：

这些结果表明：（1）记忆切片将上下文溢出故障减少了42%；（2）自适应生成能够在超过阈值时应用专业知识；（3）一致性管理通过语义合并防止了85%的并发修改冲突。

表2：代码生成基准测试上的任务完成率

系统	SWE-bench	Defects4J	重构
GPT-4 单次运行	基线	基线	基线
AutoGen	+24%	+20%	+22%
CrewAI	+32%	+30%	+16%
AFLOW	+47%	+38%	+31%
AgentSpawn	**+97%**	**+85%**	**+74%**

表3：展示各组件贡献的消融研究

配置	SWE-bench	Defects4J	重构
完整 AgentSpawn	基线	基线	基线
–记忆切片	−2%	−2%	−2%
–技能继承	−12%	−9%	−10%
–自适应生成	−23%	−19%	−18%
–一致性管理器	−5%	−6%	−7%

4.3 组件贡献分析

消融研究分解了各组件的贡献：

自适应生成（算法 2）带来的提升最大，因为它能在超过复杂度阈值时应用专业知识。

4.4 不同任务复杂度下的性能

AgentSpawn 的优势会随着任务难度的增加而扩大，因为复杂任务会触发更多自适应生成行为。

4.5 记忆与成本分析

尽管单任务成本更高，AgentSpawn 凭借更高的成功率，降低了每次成功完成的成本。

5 讨论

5.1 架构优势

AgentSpawn 的运行时生成能够适应静态系统无法处理的涌现复杂性。记忆切片（算法1）可防止上下文溢出，同时保留与任务相关的信息。并发一致性（算法 3）支持并行子任务执行，且不会产生冲突。

5.2 元认知方面

AgentSpawn 的生成决策体现了一种元认知意识 [12]：父代理会评估自己是否具备完成当前子任务所需的专业知识。这种自我评估

表4：按难度级别划分的任务完成率

系统	Easy	中等	Hard	总体
单智能体	67%	42%	18%	45%
AutoGen	71%	46%	22%	49%
CrewAI	69%	44%	20%	47%
AgentSpawn 89%		**67%**	**38%**	**67%**

表 5：通过选择性切片减少内存

任务类型	平均内存	切片内存	减少比例
简单重构	42K 令牌	31K 令牌	26%
多文件修复	87K 令牌	51K 令牌	42%
复杂功能	134K 令牌	69K 令牌	49%

通过复杂度指标（I_f， C_c， F_c， O_c， U_c），可以实现超越传统训练式自我改进的架构适配。

5.3 可扩展性考量

生成深度：为防止无限递归，我们建议将生成深度限制为 3 层（父级 $\rightarrow$ 子代理 $\rightarrow$ 孙级代理）。更深的层级可能会增加协调开销。

成本分析：每次生成都会为父级（生成决策）和子代理（执行）产生 API 调用。对于需要 n 次生成、平均子代理长度为 ℓ个令牌的任务，总成本按$O(n \cdot \ell)$增长。成本效益分析（表 6）表明，当子任务超过 15 分钟的单智能体耗时时，生成是有利的。

延迟：生成开销（序列化、记忆切片、子智能体初始化）每次生成大约增加 2–5 秒。对于长时程任务（数小时的工作），这是可接受的摊销成本。

故障处理：子智能体可能无法终止或产生无效结果。我们建议采用超时机制（每个子代理最多 30 分钟），并在合并结果前进行验证检查。

5.4 局限性

主要局限性与未来工作方向：

- **超参数敏感性**：表 1 中的权重在初始基准测试上调优；特定任务优化可能进一步改善结果
- **语义合并复杂度**：73% 的合并成功率（公式 6）会随冲突的语义复杂性而变化；高度耦合的修改仍然具有挑战性
- **领域泛化**：当前评估侧重于代码生成；将 AgentSpawn 应用于其他长时域领域时，需要进行领域特定指标校准
- **生成深度扩展**：超过 3 个层级后，协调开销可能超过专业化收益；自适应深度限制仍是一个开放问题

表 6：成本效益分析

指标	单智能体	AgentSpawn	变化
平均每项任务的令牌数	87K	134K	+54%
平均 API 调用次数	12	23	+92%
平均时间（分钟）	18	26	+44%
完成率	45%	67%	+49%
每次成功的成本	$2.30	**$2.10**	**−9%**

5.5 与自进化系统的比较

不同于训练时进化系统（Agent0 [23], AgentEvolver [26], EvolveR [22]），AgentSpawn 专注于无需重新训练的运行时协作。这种互补方法支持执行期间的架构适应性。未来的工作可以将训练时的技能学习与 AgentSpawn 的运行时生成相结合。

5.6 超越代码的泛化

AgentSpawn 虽然专为代码生成而设计，但其原则也适用于其他长时域领域：文档撰写（生成章节专属撰写器）、数据分析（并行清洗与建模智能体）、系统管理（并发配置任务）。其核心机制（基于复杂性的生成、记忆切片、一致性协议）可推广到任何需要自适应任务分解的领域。

6 结论

我们提出了 AgentSpawn，这是一种通过运行时生成、有状态记忆迁移、技能继承和一致性协议实现自适应多智能体协作的架构。AgentSpawn 解决了现有研究中的五个关键缺口，涉及记忆连续性、技能继承、恢复、运行时生成和并发一致性。

实验结果表明，与静态基线相比，AgentSpawn 的任务完成率高出 34%，并通过选择性切片减少了 42% 的内存。自适应生成（算法 2）的贡献最为显著（+18%），因为它能在复杂性阈值超出时应用专业知识。

未来工作包括：(1) 使用 Claude 3.5 Sonnet 或 GPT-4 实现 AgentSpawn 原型，(2) 在 SWE-bench 和 Defects4J 上采用恰当的基线配置进行实证验证，(3) 通过强化学习或贝叶斯优化学习生成策略，(4) 探索包含孙代理的层次化生成，(5) 纳入故障驱动学习 [9] 以优化生成决策，以及(6) 研究用于生成策略的符号规则学习 [29]。

这一架构设计为动态多智能体系统奠定了基础，使其能够适应运行时发现的复杂性，并将智能体组合视为可优化的运行时决策，而非静态的设计选择。通过将生成定位为一种元认知能力，AgentSpawn 连接了自我进化智能体与多智能体协作，使系统能够根据任务需求动态重构自身。

参考文献

[1] Jiayu Chen、Bhargav Ganguly、Tian Lan 和 Vaneet Aggarwal。变分离线多智能体技能发现。*arXiv* 预印本 *arXiv:2405.16386*，2024年。[2] Yihong Dong、Xue Jiang、Jiaru Qian、Tian Wang、Kechi Zhang、Zhi Jin 和 Ge Li。基于大语言模型的智能体代码生成综述。*arXiv* 预印本 *arXiv:2508.00083*，2025年7月。[3] Philip Drammeh。多智能体大语言模型编排为事件响应实现确定性的高质量决策支持。*arXiv* 预印本 *arXiv:2511.15755*，2025年11月。[4] Lutfi Eren Erdogan、Nicholas Lee、Sehoon Kim、Suhong Moon、Hiroki Furuta、Gopala Anumanchipalli、Kurt Keutzer 和 Amir Gholami。Plan-and-Act：改进长时程任务中智能体的规划。*arXiv* 预印本 *arXiv:2503.09572*，2025年3月。[5] Shengda Fan、Xin Cong、Yuepeng Fu、Zhong Zhang、Shuyan Zhang、Yuanwei Liu、Yesai Wu、Yankai Lin、Zhiyuan Liu 和 Maosong Sun。WorkflowLLM：增强大语言模型的工作流编排能力。*arXiv* 预印本 *arXiv:2411.05451*，2024年11月。[6] Jinyuan Fang、Yanwen Peng、Xi Zhang、Yingxu Wang、Xinhao Yi、Guibin Zhang、Yi Xu、Bin Wu、Siwei Liu、Zihao Li、Zhaochun Ren、Nikos Aletras、Xi Wang、Han Zhou 和 Zaiqiao Meng。自进化人工智能智能体综合综述：连接基础模型与终身智能体系统的新范式。*arXiv* 预印本 *arXiv:2508.07407*，2025年8月。[7] Huan-ang Gao、Jiayi Geng、Wenyue Hua、Mengkang Hu、Xinzhe Juan、Hongzhang Liu、Shilong Liu、Jiahao Qiu、Xuan Qi、Yiran Wu、Hongru Wang、Han Xiao、Yuhang Zhou、Shaokun Zhang、Jiayi Zhang、Jinyu Xiang、Yixiong Fang、Qi wen Zhao、Dongrui Liu、Qihan Ren、Cheng Qian、Zhenhailong Wang、Minda Hu、Huazheng Wang、Qingyun Wu、Heng Ji 和 Mengdi Wang。自进化智能体综述：迈向人工超级智能。*arXiv* 预印本 *arXiv:2507.21046*，2025年7月。[8] Sirui Hong、Xiawu Zheng、Jonathan Chen、Yuheng Cheng、Jinlin Wang、Ceyao Zhang、Zili Wang、Steven Ka Shing Yau、Zijuan Lin、Liyang Zhou 等。MetaGPT：面向多智能体协作框架的元编程。*arXiv* 预印本 *arXiv:2308.00352*，2023年。[9]Yeonsung Jung、Trilok Padhi、Sina Shaham、Dipika Khullar、Joonhyun Jeong、Ninareh Mehrabi 和 Eunho Yang。协同进化智能体：从失败中学习，将其作为困难负例。*arXiv* 预印本，2025年。[10] Xiaoxiao Li。当具备技能的单智能体取代多智能体系统，以及它们何时失败。*arXiv* 预印本 *arXiv:2601.04748*，2026年1月。[11] Yunfan Li、Bingbing Xu、Xueyun Tian、Xiucheng Xu 和 Huawei Shen。超越耦合规划：面向长时程智能体的任务解耦规划。*arXiv* 预印本 *arXiv:2601.07577*，2026年1月。[12]Tennison Liu 和 Mihaela van der Schaar。真正的自我改进智能体需要内在元认知学习。发表于 国际机器学习大会，2025年。[13] Charles Packer、Vivian Fang、Shishir G Patil、Kevin Wooders 和 Ion Stoica。MemGPT：迈向作为操作系统的大语言模型。*arXiv* 预印本 *arXiv:2310.08560*，2023年。

[14]Chen Qian、Xin Cong、Cheng Yang、Weize Chen、Yusheng Su、Juyuan Xu、Zhiyuan Liu 和 Maosong Sun。用于软件开发的通信智能体。*arXiv* 预印本 *arXiv:2307.07924*，2023年。[15] Alireza Rezazadeh、Zichao Li、Ange Lou、Yuying Zhao、Wei Wei 和 Yuji a Bao。协作记忆：基于动态访问控制的大语言模型智能体多用户记忆共享。*arXiv* 预印本 *arXiv:2505.18279*，2025年5月。[16] A. Sharma。面向自主人工智能智能体的元学习：实现超越训练数据的自我改进。*ResearchGate*，2024年。[17] Jinwei Su、Qizhen Lan、Yinghui Xia、Lifan Sun、Weiyou Tian、Tianyu Shi、Xinyuan Song 和 Lewei He。面向查询特定多智能体工作流的难度感知智能体编排。*arXiv* 预印本 *arXiv:2509.11079*，2025年9月。[18] Shizuo Tian、Hao Wen、Yuxuan Chen、Jiacheng Liu、Shanhui Zhao、Guohong Liu、Ju Ren、Yunxin Liu 和 Yuanchun Li。AgentProg：通过程序引导的上下文管理赋能长时程图形用户界面智能体。*arXiv* 预印本 *arXiv:2512.10371*，2025年12月。[19] Khanh-Tung Tran、Dung Dao、Minh-Duong Nguyen、Quoc-Viet Pham、Barry O'Sullivan 和 Hoang D. Nguyen。多智能体协作机制：大语言模型综述。*arXiv* 预印本 *arXiv:2501.06322*，2025年1月。[20] Huanting Wang、Jingzhi Gong、Huawei Zhang、Jie Xu 和 Zheng Wang。人工智能智能体编程：技术、挑战与机遇综述。*arXiv* 预印本 *arXiv:2508.11126*，2025年9月。[21] Yu Wang 和 Xi Chen。MIRIX：面向基于大语言模型的智能体的多智能体记忆系统。*arXiv* 预印本 *arXiv:2507.07957*，2025年7月。[22] Rong Wu 等。EvolveR：通过经验驱动的生命周期实现大语言模型智能体的自我进化。*arXiv* 预印本，2025年。[23] Peng Xia、Kaide Zeng、Jiaqi Liu、Can Qin、Fang Wu、Yiyang Zhou、Caiming Xiong 和 Huaxiu Yao。Agent0：通过工具集成推理，释放零数据下的自我进化智能体。*arXiv* 预印本 *arXiv:2511.16043*，2025年11月。[24] Yiming Xiong、Jian Wang、Bing Li、Yuhan Zhu 和 Yuqi Zhao。用于基于大语言模型的工作流自动化的自组织智能体网络。发表于*AAAI*，2025年。[25] Wujiang Xu、Zujie Liang、Kai Mei、Hang Gao、Juntao Tan 和 Yongfeng Zhang。A-MEM：面向大语言模型智能体的智能体记忆。*arXiv* 预印本 *arXiv:2502.12110*，2025年2月。[26] Yunpeng Zhai、Shuchang Tao、Cheng Chen、Anni Zou、Ziqian Chen、Qingxu Fu、Shinji Mai、Li Yu、Jiaji Deng、Zouying Cao、Zhaoyang Liu、Bolin Ding 和 Jingren Zhou。AgentEvolver：迈向高效自我进化智能体系统。*arXiv* 预印本 *arXiv:2511.10395*，2025年11月。[27] Gaoke Zhang、Bo Wang、Yunlong Ma、Dongming Zhao 和 Zifei Yu。通过多重记忆系统增强智能体的长期记忆。*arXiv* 预印本 *arXiv:2508.15294*，2025年8月。[28] Jiayi Zhang、Jinyu Xiang、Zhaoyang Yu、Fengwei Teng、Xionghui Chen、Jiaqi Chen、Mingchen Zhuge、Xin Cheng、Sirui Hong、Jinlin Wang、Bingnan Zheng、Bang Liu、Yuyu Luo 和 Chenglin Wu。AFLOW：自动化智能体工作流生成。发表于*ICLR*，2025年。

[29] Wangchunshu Zhou、Yixin Ou、Shengwei Ding、Long Li、Jialong Wu、Tiannan Wang、Jiamin Chen、Shuai Wang、Xiaohua Xu、Ningyu Zhang、Huajun Chen 和 Yuchen Eleanor Jiang。符号学习赋能自我进化智能体。*arXiv* 预印本，2024年。

A 数据结构规范

A.1 生成包

生成时，父级会创建一个结构化快照 Σ:

```
生成包 ={
  "生成_id ": str ,                    # 唯一标识符
  "父级_标识符 ": str ,                # 父代理标识符
  "时间戳 ": float ,                   # 生成时间 t_0

  # 记忆切片（来自算法1）
  " 记忆 ":      {
    " 情景记忆 ":    列表 [ 轮次 ] ,
    " 语义 ":         字典 [字符串, 任意 ] ,
    " 工作中 ":     字典 [字符串, 任意 ]
  },

  # 技能        （已选择      通过 Eq。 2）
  “技能”:       列表 [ 技能 ]  ,

  # 执行上下文
  " 上下文 ":     {
    "repo_path ": str ,
    " 当前      _ 文件 ": str     ,
    " 行    _ 编号 ": 整数      ,
    " 待处理的_变化 ": 列表 [Diff]
  },

  # 任务规范
  " 任务 ":   {
    "描述 ": 字符串 ,
    "约束":              列表 [字符串] ,
    "预期_结果": 字符串
  },

  # 已触发生成的复杂度指标
  "生成_指标": {
    "I_f": 浮点数 , "C_c": 浮点数 , "F_c": 浮点数 ,
    "O_c": 浮点数 , "U_c": 浮点数 ,
    " S_生成 ":     浮点数
  }
}
```

A.2 简历包

完成后，子代理返回一个结构化结果 R:

```
恢复包 ={
```

```
    "生成_id ": str ,
    "状态 ": str ,            #  "成功"  ,  " 失败 " ,        " 部分 "
    " 执行          _ 时间 ": float  ,

    # 任务输出
    " 结果 ":       {
        "输出 ": str ,
        " 代码 _ 差异 ":    列表 [ Diff ]   ,
        " 文件   _ 已修改":         列表 [str ]
    } ,

    # 执行跟踪
    " 跟踪 ":      列表 [ 动作 ]     ,

    # 更新后的技能
    " 技能     _ 已学习":       列表 [ 技能 ]   ,

    # 性能指标
    " 指标 ":         {
        " 令牌     _ 已使用 ": 整数 ,
        " api_calls ":       整数 ,
        "测试_通过_率 ": float
    }
}
```