贵州轻工职业技术学院

2026 届毕业设计

FlowCabal
Agent 半自动化写作辅助工具

学科专业：大数据技术

指导老师：曾小爽

学生学号：20230105235

学生姓名：田照涛

中国．贵州．贵阳

2026 年 3 月

摘　要

FlowCabal 是一款专注于 AI 辅助写作的软件，专门面向高质量长篇写作的场景。软件提供蓝图式工作流编排、内置记忆文件和 agent 交互方式。本文将从产品功能、技术选型、软件架构和具体用例等方面介绍这款软件。软件源代码在 https://github.com/isirin1131/FlowCabal 开源

关键词：工作流；智能体；记忆

Abstract

FlowCabal is a software focused on AI-assisted writing, specifically designed for high-quality long-form writing scenarios. The software provides blueprint-style workflow orchestration, built-in memory files, and agent interaction methods. This article introduces the software from aspects such as product features, technology selection, software architecture, and specific use cases. The source code of the software is open-sourced at https://github.com/isirin1131/FlowCabal.

Keywords: workflow; agent; memory

目　录

第 1 章 绪论

1.1 研究背景

2017 年，Vaswani 等人提出的 Transformer 架构[1] 为大语言模型（LLM）的发展奠定了基础。此后短短数年间，LLM 经历了爆发式增长：2020 年 Brown 等人发布的 GPT-3[2] 以 1750 亿参数和少样本学习能力震动了整个领域，2022 年底 ChatGPT 的上线将基于 LLM 的对话式 AI 带入大众视野，2023 年的 GPT-4[3] 更是将上下文窗口从千级 tokens 推向数万级。到 2026 年初，主流模型的上下文窗口已普遍达到百万 token 级别。在这一进程中，AI 辅助写作始终是最受关注的应用场景之一。

当基于 LLM 的 chat-ai-app 刚刚进入人们视野时，第一个被广泛关注的概念是**提示词工程**。GPT-3[2] 所展示的少样本提示能力表明，精心设计的提示词可以在无需微调的前提下引导模型完成各类任务；Wei 等人提出的思维链提示[4] 则进一步证明，在提示中加入中间推理步骤可以显著提升模型在复杂推理任务上的表现；White 等人在[5] 中进一步将提示词设计归纳为可复用的模式目录，标志着提示词工程从经验总结走向了方法论。然而提示词工程的关注点集中于单次调用的输入设计，当任务从简单问答扩展到需要多步协作、持续记忆和外部工具调用的复杂场景时，仅靠优化提示词已然不够。

2025 年年中，Andrej Karpathy 在一篇广泛传播的帖子[6] 中提出了**上下文工程**这一概念，指出在工业级 LLM 应用中，真正的挑战不是写一条好的提示词，而是“用恰当的信息填充整个上下文窗口的精细艺术与科学”。上下文工程自然地涵盖了提示词工程，但视野更广：不仅关注“对模型说什么”，还关注在什么时机、以什么结构、用多少信息来构建上下文。这一概念在 2025 年到 2026 年初的 AI 编程大爆发和 agent 大爆发中得到了充分的实践验证——Anthropic 和 OpenAI 的 agent 团队都在上下文管理与记忆方面积累了大量公开经验（详见研究意义一节）。

在 AI 辅助写作这个具体领域，提示词工程和上下文工程同样是最核心的挑战。Yang 等人的 DOC[7] 通过详细大纲控制提升了长篇故事的连贯性，证明了结构化控制手段对长文本生成的重要性——但这仍然停留在单次模型调用的层面。当写作规模扩大到数万字乃至更长的篇幅时，单次调用的方案无论如何优化都会撞上上下文窗口的硬性天花板。如何在多次调用间维持一致性、如何管理跨章节的记忆、如何编排复杂的写作流程——这些问题已经超出了提示词工程的范畴，呼唤更系统性的上下文工程方案。

1.2 研究意义

尽管 LLM 的上下文窗口已经从 2022 年的 4096 tokens 扩展到了百万级别，但 Wu 等人在 LongGenBench[8] 中的评测表明，十个主流模型在长文本生成任务中的表现均随文本长度增加而显著下降——模型能“读”长文本，不代表能“写”长文本。与此同时，Hong 等人提出的“上下文腐化”现象[9] 指出，输入 token 数量的增加会系统性地削

弱模型的推理能力，这对需要在数万字篇幅内维持连贯性的写作场景构成了根本性的挑战。

事实上，工业界最前沿的 agent 团队已经在上下文管理和记忆方面积累了大量经验。Anthropic 在 2024 年 12 月发表的《Building Effective Agents》[10] 中将增强型 LLM（配备检索、工具和记忆的 LLM）定义为 agent 系统的基本构建单元，并明确区分了“工作流”和“agent”两种编排范式。随后 Young 在《Effective Harnesses for Long-Running Agents》[11] 中进一步指出，长时间运行的 agent 的核心挑战在于每个新的会话都从零开始、没有前序记忆，他们的解决方案是通过初始化 agent 配合编码 agent 的双 agent 架构，辅以进度文件来在会话间传递状态。2026 年初，Anthropic 又在 Claude 开发者平台上正式推出了上下文编辑和记忆工具[12]，前者能在逼近 token 上限时自动清除陈旧的工具调用结果（在 100 轮 web 搜索评测中减少了 84% 的 token 消耗），后者则通过基于文件的系统让 agent 在上下文窗口之外存取信息。

OpenAI 的 Codex 团队也在同期给出了类似的答案。Bolin 在《Unrolling the Codex Agent Loop》[13] 中详述了 Codex CLI 的 agent 循环机制，其中上下文压缩（compaction）是关键策略——当 token 数超过阈值时，系统会用摘要替换原始对话，使 agent 能在有限的窗口内持续工作。Choi 在《Run Long Horizon Tasks with Codex》[14] 中则展示了一个实际案例：GPT-5.3-Codex 在不间断运行约 25 小时、消耗约 1300 万 token 的条件下生成了约 3 万行代码，其中最关键的技术是持久化项目记忆——将规格、计划、约束和状态写入 markdown 文件供 agent 反复回访，以防止长时间运行中的漂移。

另一方面，工作流编排已经在 agent 领域展现出了强大的组织能力。Fan 等人的 WorkflowLLM[15] 在 ICLR 2025 上证明了 LLM 可以通过工作流来编排复杂的多步骤任务，LangGraph 等框架也已将节点式工作流变成了 agent 应用的经典实践。然而，这些工作几乎都聚焦于代码生成、数据处理等结构化任务，将工作流编排应用于创意写作的尝试仍然稀缺。

FlowCabal 的意义就在于填补这个空白：上述团队在编程 agent 上验证过的上下文压缩、持久化记忆、进度文件等机制，完全有潜力迁移到长篇写作的场景中。FlowCabal 将工作流编排、记忆管理和 agent 监控三者结合，把一个原本高度依赖人类逐句干预的长篇写作过程变成半自动化的流水线，同时也为上下文工程在创作领域的落地提供了一个可供参考的实践。

第 2 章 产品功能

本节以渐进式的视角审视了 FlowCabal 的功能设计，省略了很多软件实现上的细节，力求以更通用的视角解释清楚 FlowCabal 的优点。

2.1 蓝图式工作流编排

研究背景一节已经指出，长篇写作必然涉及对 LLM 的多次调用。由此而来的核心问题是：如何编排这些调用之间的依赖关系，以及如何将前序调用的输出转化为后续调用的输入？蓝图式工作流编排是 FlowCabal 对这一问题的回答。

将 LLM 抽象为 `Model(query) = Answer`，常见的 chat-ai-app 以线性对话的形式组织 `query`：

```
"query": [
    { "role": "user", "content": "你好" },
    { "role": "assistant", "content": "你好！有什么我可以帮助你的吗？" },
    { "role": "user", "content": "你是谁" },
    ......
]
```

这种线性结构意味着每次调用都携带完整的对话历史，既浪费 token，也无法灵活定义调用间的依赖关系。节点工作流提供了更通用的替代方案：每个 LLM 调用被封装为独立节点，节点间通过显式连线声明数据依赖。Unreal Engine 的蓝图系统、ComfyUI 以及 agent 领域的 LangGraph 都采用了这一范式。

FlowCabal 的节点工作流可以形式化描述如下。设工作流包含 n 个节点 $M_1, M_2, ..., M_n$，第 k 个节点的查询由自定义的组装函数 f_k 从其依赖节点的输出中生成：

$$M_k.\text{query} = f_k(\{m.\text{Answer} \mid m \in D_k\})$$

其中 $D_k \subset \{M_1, ..., M_n\}$ 是 M_k 的依赖集。例如，一个简单的拼接函数 $f(a, b, c) = a + b + c$ 即可将三个依赖节点的输出合并为一条查询。这些节点间的依赖关系构成有向无环图（DAG），FlowCabal 使用 Kahn 算法进行拓扑排序后逐层并行执行。

值得注意的是，这种节点式调用天然具备上下文压缩的特性：每个节点 M_k 的查询仅包含其依赖节点的输出，而非整个工作流的完整历史——M_k 的查询不会出现在任何非依赖节点 M_p 的查询中。这恰好呼应了研究背景中的上下文工程问题：节点工作流通过结构化的依赖声明，从机制层面实现了对上下文的精确控制。

2.2 内置记忆文件

长篇写作中，Agent 需要在整部小说——包括已定稿的章节、角色设定、世界观和文体约定——中探索上下文并检查一致性。常见的做法是引入 RAG（检索增强生成），用向量数据库进行语义召回。然而 FlowCabal 明确放弃了这一路径：小说中语义相似

的段落可能多达数十上百种，embedding 召回无法区分哪个是当前真正需要的；更关键的是，伏笔与回收在语义空间里往往距离很远，余弦相似度捕捉不到因果链。

FlowCabal 采用纯文件系统的记忆架构，核心思路是将记忆文件视为定稿章节（manuscripts）的**有损缓存**。类比编程 agent：代码库是记忆，grep 是检索；对应到 FlowCabal：`manuscripts/` 目录存放完整的定稿章节作为信息源，而 `characters/`、`world/`、`voice.md` 等记忆文件则是为了避免每次都加载全部原文而提取的结构化摘要。

记忆的加载遵循三级渐进策略：L0 层是 `index.md` 导航索引，Agent 每次启动时自动加载；L1 层是各类记忆文件，Agent 读取 L0 后通过工具按需加载；L2 层是 `manuscripts/` 中的完整原文，通过文件间的稀疏跳转链接按需可达。这种设计使得上下文检索是**因果关系驱动**的——Agent 理解叙事上下文后自主决定加载什么，而非依赖统计意义上的语义相似性。所有记忆文件对人和 Agent 完全开放：用户可以直接用文本编辑器修改，Agent 也可以在写作过程中创建新文件或更新已有内容。

2.3 Agent 交互方式

FlowCabal 的 Agent 不仅负责文本生成，还承担上下文注入和约束检查的职责。其核心机制是 **agent-inject**：节点的 prompt 中可以嵌入注入点（hint），执行时 Agent 会以节点 prompt 和上游输出为锚点，从记忆系统中查询相关约束，并将查到的上下文自动注入。这实质上将约束查询和上下文注入合二为一——Agent 为了检查一致性而查到的信息，恰好就是生成时所需的上下文。

工作流的执行采用增量构建模式。每个节点的 prompt 经解析后计算 SHA-256 哈希值，与缓存中的哈希比对：匹配则跳过，不匹配则重新执行并级联更新所有下游节点。这意味着用户修改一个节点的 prompt 后再次运行，只有受影响的节点会重新生成，大幅节省 token 消耗和等待时间。

每个节点的输出以多版本形式保留，版本切换仅修改当前指针而不删除旧版本。用户既可以在不同版本间对比择优，也可以直接手动编辑某个节点的输出作为新版本。此外，工作流支持 step 模式——逐层执行，每层完成后暂停等待用户确认。用户可在暂停期间审视结果、调整参数或编辑输出，然后再推进到下一层。这种逐层暂停的机制将一个高度自动化的流水线转变为人机协作的迭代过程。

2.4 设计哲学的演进

FlowCabal 的开发历时数月，期间经历了多次方向性的转变。这些转变并非简单的技术替换，而是反映了开发者对“AI 辅助写作工具应该是什么”这一根本问题的认识不断深化。本节梳理三条贯穿始终的哲学线索，以说明当前设计的来由。

2.4.1 从完备设计到最小可用

项目早期的设计文档追求全面性：三角色 Agent（上下文检索、工作流修改、质量评估各司其职）、五种记忆侧写类型（角色、情节线、世界状态、主题、文风）、向量数据库语义检索、浏览器可视化编辑器——每一项都有清晰的理论动机。然而这些

设计停留在纸面的时间远长于代码实现的时间。三角色之间的协调协议尚未定义，执行引擎本身还不能跑通一个最简单的两节点工作流。

转折发生在放弃 Python 后端、回归 TypeScript 单进程的那一轮重构。这次重构的核心不是技术栈的更换，而是开发策略的转变：不再试图在动手之前设计出完备的系统，而是先实现一个能端到端跑通的最小内核，再在使用中发现真正需要的扩展点。三角色合并为单 Agent，五种侧写类型简化为自由增长的文件目录，向量检索被三级渐进加载取代——这些"简化"并非偷懒，而是承认了一个事实：在没有真实写作任务验证之前，无法判断哪些复杂度是必要的。

这一哲学贯穿了此后的每一轮迭代。例如，执行引擎最初设计了 `auto` 和 `step` 两种运行模式、事件总线、可 await 的 `RunHandle` 对象——这些抽象在 TUI 实时界面下是合理的，但当开发重心转向 CLI 时，它们全部被简化为一个普通的 `for` 循环逐层调用 `executeLevel()` 函数。每次简化都遵循同一个判断标准：当前阶段的唯一用户（开发者自己通过 CLI 驱动工作流）是否真的需要这个抽象？

2.4.2 从隐式推导到显式标记

工作流引擎的一个核心问题是缓存失效：用户修改了某个节点的 prompt 后，哪些节点需要重新执行？早期设计采用结构性哈希方案——将节点的所有 TextBlock 递归解析后计算 SHA-256 哈希值，与缓存中的哈希比对，不匹配则判定为过期并级联传播到所有下游节点。

这一方案在理论上是优雅的：缓存失效完全由数据驱动，无需维护额外的状态。但在实践中暴露了两个问题。其一，`agent-inject` 类型的 TextBlock 无法有效哈希——其内容由 Agent 在运行时动态生成，相同的 hint 在不同的记忆状态下可能产生完全不同的注入内容，哈希相同不代表语义相同。其二，哈希比对需要在每次查询节点状态时递归解析整棵依赖子树，当工作流规模增大时，即使是一次简单的 `getNodeStatus()` 调用也隐含了大量的计算。

最终方案转向显式标记：引入 Target Set（待执行节点集合）和 Stale Roots（过期根节点集合）两个持久化数据结构。当节点被修改时，系统显式地将其加入 Target Set 并标记下游为 Stale Root；当节点执行完成时，显式地将其从 Target Set 移出。这种方式牺牲了哈希方案的"零额外状态"特性，换来了两个收益：状态查询变为 $O(1)$ 的集合查找而非递归计算，以及 `agent-inject` 的失效判断不再依赖无法可靠计算的哈希值。

这一转变体现了更一般的设计倾向：当隐式推导的正确性难以保证或性能难以接受时，宁可引入少量显式状态来换取系统的可预测性。

2.4.3 从有状态对象到无状态函数

FlowCabal 的 API 风格经历了从面向对象到函数式的演化。早期的使用模式是"打开一个 Workspace 会话对象，在其上调用方法"：

```
const ws = await openWorkspace(rootDir, wsId, llmConfigs);
const nodes = ws.getNodes();          // 同步查询
const status = ws.getNodeStatus(id);  // 同步查询
const handle = ws.startRun(targets);  // 返回 RunHandle
```

```
handle.subscribe((event) => { ... }); // 订阅事件
await handle.done;                    // 等待完成
```

这种模式为 TUI 的实时刷新提供了便利——Workspace 对象在内存中维护完整状态，查询零延迟，变更通过事件推送。但它也带来了隐性的耦合：调用方必须正确管理 Workspace 的生命周期（打开、使用、关闭），多个调用方共享同一个 Workspace 对象时需要协调状态一致性，且整个 API 表面积（RunHandle、RunMode、EventBus、StateEvent 等）对于 CLI 这种"执行一次就退出"的场景而言过于庞大。

当开发重心从 TUI 转向 CLI，且外部 Agent（如 Claude Code）成为预期的主要调用方时，API 风格随之转变为无状态的原子函数：

```
const loc = { rootDir, workspaceId };
const nodes = await loadNodes(loc);           // 独立函数，加载→返回
await addNode(loc, newNode);                  // 独立函数，加载→修改→持久化
const results = await executeLevel({ loc, nodeIds, ... }); // 独立函数
```

每个函数自行完成"加载 → 操作 → 持久化 → 返回"的完整周期，调用方无需维护任何中间状态。这种设计对外部 Agent 尤为友好：Agent 可以在任意时刻调用任意函数，无需理解 Workspace 的生命周期语义，也无需担心并发状态冲突——每次调用都是一个独立的事务。

这一演化的深层逻辑是：工具的 API 应当匹配其主要调用方的心智模型。TUI 的调用方是持续运行的 React 渲染循环，适合有状态对象；CLI 和外部 Agent 的调用方是一次性的命令执行，适合无状态函数。当调用方发生变化时，API 风格也应当随之调整，而非用适配层来弥合差异。

2.4.4 其他设计取舍

除上述三条主线外，开发过程中还涉及若干较小但值得记录的设计取舍。

冻结机制的废弃。早期设计中，虚拟文本块（VirtualTextBlock）支持"冻结"操作：用户可以将某个节点的输出锁定在特定版本，使得上游重新执行时不会破坏下游已确认的结果。这一机制在浏览器可视化编辑器中有明确的使用场景——用户拖拽节点、实时编辑时需要保护部分结果不被意外覆盖。转向 CLI 后，工作流的执行变为显式的命令触发，不存在"意外覆盖"的场景，冻结机制被多版本存储加版本切换取代：每个节点保留所有历史版本，版本切换仅修改当前指针，用户可以随时回到任意历史版本。

环形节点的排除。设计文档中曾探讨过在 DAG 中引入环形节点以支持迭代式改写（节点将自身输出作为输入反复精炼），但最终明确排除了这一方向。原因有二：Kahn 拓扑排序算法天然不支持有环图，引入环形节点需要额外的循环检测和终止条件逻辑；更重要的是，迭代式改写本质上是 Agent 推理过程的一部分，应当发生在 Agent 的 tool-calling 循环内部而非 DAG 的拓扑结构中。

记忆文件的写入原则。围绕记忆文件应当写入什么内容，设计过程中经历了从"尽可能全面"到"只写生成性事实"的转变。早期方案试图在记忆文件中穷举角色的行为约束（"角色 A 绝不会做 X、Y、Z"），但这种派生性断言（derivative assertions）的列表天然不可能完备，且随着情节推进需要不断补充。最终确立的原则是只记录生成

性事实（generative facts）——即驱动角色行为的核心动机和规则（“角色 A 的行动逻辑源于 P”），由 Agent 在写作时从规则推导出具体行为，而非逐条查找约束清单。

种子文件的选择。项目初始化时应当预创建哪些记忆文件？设计文档中讨论了多种候选：大纲（outline.md）、时间线（chronicle.md）、情节线（threads.md）等。最终标准是“约束力”：只预创建那些 Agent 无法自动推导、且对写作质量有直接约束力的文件——`index.md`（导航索引）、`voice.md`（文体约定）、`characters/`（角色设定）和 `world/`（世界观）。大纲和时间线被排除在外，因为它们要么变化过快（大纲随写作推进不断调整），要么可以从已完成的章节中自动派生（时间线是定稿章节的事件摘要）。

第 3 章 技术选型

FlowCabal 是纯客户端、无服务器的本地应用程序，当前以命令行界面（CLI）发布，但在架构上预留了 Web UI 的接入能力。本节从编程语言与运行时、核心依赖库和存储方案三个维度阐述技术选型的理由。

3.1 编程语言与运行时

FlowCabal 选择 TypeScript 作为开发语言，Bun 作为运行时。选择 TypeScript 而非 Python 或 Go 的主要考量有三：其一，TypeScript 的静态类型系统为工作流引擎中大量的类型定义（节点、TextBlock、版本缓存等）提供了编译期保障；其二，TypeScript 原生的 `async/await` 和 `Promise` 机制天然适配 LLM 流式调用和 DAG 逐层并行执行的场景；其三，未来接入 Web UI 时可以在前后端之间共享类型定义和数据结构，避免序列化层的重复开发。

Bun 相较于 Node.js 的优势在于：启动速度更快（冷启动约为 Node.js 的四分之一），原生支持 TypeScript 无需额外的编译步骤，且提供了 `bun build --compile` 命令将整个项目打包为单个可执行二进制文件。这种单二进制分发方式借鉴了 Claude Code、OpenCode 等同类 CLI 工具的实践，用户无需预装 Node.js 或执行 `npm install` 即可直接使用。

项目采用 Monorepo 结构，划分为 `engine`（工作流引擎）和 `cli`（命令行界面）两个包。`engine` 作为纯库不包含任何终端 I/O 操作，所有副作用通过事件上报，使得未来的 Web 前端可以调用同一套引擎接口而无需适配。

3.2 核心依赖库

LLM 集成方面，FlowCabal 选择 Vercel AI SDK（`ai` 及 `@ai-sdk/*` 系列包）作为模型调用层。该 SDK 提供了统一的 Provider 接口，一套代码即可对接 OpenAI、Anthropic、Google、Mistral、xAI、Cohere 等主流模型服务商，同时通过 `openai-compatible` Provider 覆盖 baseURL 即可支持 DeepSeek 等兼容 OpenAI 接口的第三方服务。此外，Vercel AI SDK 内置了 Zod Schema 到 JSON Schema 的自动转换，配合其 tool calling 循环机制，大幅简化了 Agent 工具调用的实现。流式输出方面，`streamText` 接口提供开箱即用的 `AsyncIterable`，使得工作流运行时可以逐块读取生成内容并实时触发事件。

CLI 交互方面，命令解析使用 yargs，交互式提示（如确认、选择）使用 @clack/prompts。运行时数据校验使用 Zod：所有来自外部的输入（JSON 配置文件、用户填写的工作流定义等）在进入引擎前均通过 Zod Schema 校验，确保类型安全从编译期延伸到运行时。

3.3 存储方案

FlowCabal 的所有持久化数据——工作流定义、节点输出缓存、记忆文件——均以 JSON 或 Markdown 文件的形式存储在文件系统中，不依赖任何数据库。这一选择

基于以下判断：典型的 FlowCabal 项目包含 10 至 50 个节点，每个节点不超过 10 个版本，总数据量通常在 1 MB 以内，纯文件读写完全能满足性能需求；纯文本格式天然支持 Git 版本控制，用户可以对整个项目目录执行 `git diff` 和 `git merge`；此外，零外部依赖意味着用户无需安装 SQLite 或其他存储引擎，降低了部署门槛。

第 4 章 软件架构

本章从模块分层、核心数据结构、执行数据流和交互模型四个维度描述 FlowCabal 的软件架构。项目采用 monorepo 结构，分为 `engine`（纯库，零 I/O 假设）和 `cli`（命令行界面）两个包，使得未来的 Web 前端可以调用同一套引擎接口。

4.1 总体架构

4.2 数据流

4.3 过往决策案例

FlowCabal 的当前架构并非一蹴而就，而是经历了多轮设计迭代后逐步收敛的结果。本节选取四个具有代表性的架构决策，记录其动机、演化过程和最终定位，以展示软件设计中“从理想方案到实用方案”的典型路径。

4.3.1 从 Python 后端到纯 TypeScript 单进程

FlowCabal 最初的架构设想是浏览器前端（SvelteFlow 可视化编辑器）加本地 Python 后端的双进程方案。选择 Python 的理由在于其 AI/ML 生态成熟，且 Agent 的工具调用（function calling）在 Python 侧实现可以避免跨进程的 IPC 开销。数据存储使用 SQLite 统一管理工作流定义、执行状态和策展输出，前后端通过 WebSocket 通信。

然而，这一方案在原型开发阶段暴露了显著的复杂度问题：浏览器、Python 后端和 WebSocket 三个组件的生命周期管理和错误传播难以协调；SQLite 与浏览器前端之间的数据同步需要大量的胶水代码；最关键的是，用户需要预装 Python 运行时和相关依赖，与“绿色免安装”的分发理念相悖。

最终选择纯 TypeScript + Bun 单进程方案：Bun 原生支持 TypeScript 编译和执行，`bun build --compile` 可以将整个项目打包为单个可执行二进制文件；Vercel AI SDK 在 TypeScript 侧提供了与 Python 同等水平的 LLM 集成能力；文件系统存储替代 SQLite 后，所有数据天然支持 Git 版本控制。这一决策将三进程的 IPC 复杂度降为零，同时保留了未来接入 Web UI 时前后端共享类型定义的优势。

4.3.2 从三角色 Agent 到单 Agent + tool-use

在 Python 后端时期，Agent 子系统采用了精心设计的三角色架构：Role A（Context Agent）在每个节点执行前检索上下文并注入，Role B（Builder Agent）在执行前提议工作流结构修改，Role C（Monitor Agent）在每个节点执行后进行质量评估并决定是否批准、重试或标记。此外，记忆系统设计了五种侧写类型（角色、情节线、世界状态、主题、文风），配合向量数据库（OpenViking）进行语义检索。

这一设计的问题在于过度工程化：在没有可工作的执行引擎之前，三角色协调逻辑无从验证；五种侧写类型的边界模糊，难以判断一个事实应归入哪种类型；向量检

索在小说场景下的效果令人怀疑——伏笔与回收在语义空间中可能距离很远，余弦相似度无法捕捉因果链。

最终架构将三角色合并为单一 Agent，通过 Vercel AI SDK 的 tool calling 机制暴露记忆读写和运行时查询等工具，由 Agent 在推理过程中自主决定何时查询何种信息。记忆系统从五种固定侧写类型改为自由增长的文件结构：`characters/`、`world/`、`voice.md` 作为种子文件，Agent 可以按需创建新文件或更新已有内容。向量检索被完全废弃，取而代之的是 L0（索引）→ L1（记忆文件）→ L2（原稿）的三级渐进加载策略，Agent 通过理解叙事上下文来决定加载路径，而非依赖统计意义上的语义相似性。

4.3.3 从 EventBus 到 State 中心化

早期设计中，模块间的通信依赖 EventBus：UI 通过事件触发执行引擎的操作，执行引擎通过事件向 UI 报告状态变更。这种松耦合的设计在 SvelteFlow 前端时期是合理的，但当项目转向 TUI 界面后暴露了性能问题。

具体而言，TUI 需要实时刷新节点状态（done/stale/pending），而在文件系统存储方案下，每次查询都需要从磁盘读取版本文件。一个包含 10 个节点的工作流每次刷新就需要 20 次以上的文件读取操作，在 Bun 的文件 I/O 下虽然绝对耗时不大，但频繁的异步操作使得 TUI 的渲染逻辑变得复杂——所有查询都必须是异步的，React 组件的状态同步需要额外的 `useEffect` 链。另一个问题是 Agent 的运行时查询不可见：执行循环（`run.ts`）维护了一个私有的 `outputs: Map` 来追踪当前执行结果，Agent 通过 RuntimeContext 接口查询运行时状态时无法读到这个私有 Map。

State 中心化的方案引入了 `state.ts` 作为唯一的内存模型：`loadState()` 在工作区打开时一次性加载所有版本文件到内存，此后所有读操作均为同步、零 I/O 的内存访问。写操作采用“内存先更新 → await 磁盘持久化 → emit StateEvent”的模式，保证内存与磁盘的最终一致性。事件系统也从单一的 EventBus 分化为两个通道：StateEvent 报告持久化状态变更（`version:added`、`nodes:changed` 等），RunEvent 报告临时的执行生命周期事件（`node:start`、`level:done` 等），两者均采用可序列化的 discriminated union 类型，为未来的 Web 前端和事件回放留出空间。

4.3.4 从有状态会话到原子操作函数

State 中心化的架构解决了查询性能和 Agent 可见性问题，但引入了新的复杂度：`state.ts` 承担了存储、计算和事件三重职责，成为一个“上帝对象”；`run.ts` 维护了 `RunHandle`、`RunMode`（auto/step）、事件总线等抽象，使得外部自动化工具（如 Claude Code）难以零成本驱动工作流。对于 Beta 版而言，CLI 是唯一的交互界面，TUI 的实时刷新需求不再是约束条件。

最终的重构将“打开会话对象 → 调用方法”的模式变为“每次调用一个独立函数，加载 → 执行 → 持久化 → 返回”的原子操作模式。`state.ts` 被拆分为 8 个职责单一的模块：`io.ts`（薄 I/O 层）、`nodes.ts`（节点管理）、`targets.ts`（目标集合）、`stale.ts`（过期传播）、`versions.ts`（版本管理）、`todo.ts`（待执行列表计算）、`execute.ts`（执行）和 `preferences.ts`（偏好设置）。

缓存失效机制也随之简化。原方案使用 SHA-256 哈希比对节点 prompt 的结构性变化来判断是否需要重新执行——这需要在每次查询时递归解析所有 TextBlock 并计算哈希，逻辑复杂且对 agent-inject 类型的块无法有效哈希。新方案引入了两个持久化集合：Target Set（待执行节点）和 Stale Roots（过期根节点）。当节点的 prompt 被修改时，该节点自动加入 Target Set，其直接下游被标记为 Stale Root；执行完成后，节点从 Target Set 移出，并通过 BFS 从 Stale Roots 前向传播计算所有可能过期的节点。这种显式标记的方式比哈希比对更直观，也使得 `todo.ts` 可以作为纯函数实现——输入 Target Set、Stale Roots 和已有输出，输出拓扑排序后的待执行层次列表，无任何副作用。

4.3.5 其他架构变更

除上述四个主要决策外，开发过程中还有若干较小的架构变更值得记录。

显式边到隐式依赖。早期设计文档中，节点间的依赖关系由独立的边（edge）数据结构维护，包含 source 和 target 两个字段。随着 TextBlock 系统的成熟，依赖关系被改为从 `kind: "ref"` 类型的文本块中隐式推导——扫描节点的所有 TextBlock，收集其中 ref 块的 `nodeId` 即可得到完整的依赖图。这消除了边与 ref 块之间的冗余表示，也简化了节点增删时需要同步更新边列表的逻辑。

存储方案的三次迁移。数据存储经历了 IndexedDB → SQLite → 文件系统三个阶段。IndexedDB 是浏览器前端时期的自然选择，但其 API 的异步性和跨进程访问的困难使得后端 Agent 无法直接读写；迁移到 Python 后端后改用 SQLite 统一存储，但单进程 TypeScript 方案使得关系数据库的事务和查询能力成为不必要的复杂度；最终采用纯 JSON 文件存储，并将路径分为全局配置（`~/.config/flowcabal/`）和项目本地缓存（`<project>/.flowcabal/`）两层，前者存放跨项目共享的 LLM 配置和工作流模板，后者存放项目专属的记忆文件和工作区缓存。

界面形态的演化。用户界面经历了 SvelteFlow 可视化编辑器 → OpenTUI 终端界面 → yargs 命令行三个阶段。SvelteFlow 提供了直观的节点拖拽和连线体验，但浏览器 UI 与本地 Python 后端的 WebSocket 通信增加了大量胶水代码；转向纯 TypeScript 后尝试使用 OpenTUI 构建终端界面，实现了五个视图和六个 React hooks 的完整 TUI 应用；最终在 Beta 阶段将开发重心转向 CLI，原因是 CLI 命令可以被外部 Agent（如 Claude Code）直接调用，而 TUI 的交互式界面对自动化工具不友好。

工作流模板与工作区实例的分离。早期版本中，工作流定义和执行状态混杂在同一个数据结构中。后续设计将二者明确分离为三个层次：Workflow（纯模板，只描述节点结构和 prompt 组合方式，不含 LLM 配置，用于分享）、Workspace（模板的一次实例化，绑定具体项目，存储执行缓存和版本历史）、Project（小说项目，拥有独立的记忆目录）。这种分离使得同一个工作流模板可以在不同项目中复用，同一个项目也可以同时运行多个工作区以比较不同的编排策略。模板中的节点 ID 使用人类可读的顺序编号，导入工作区时自动替换为 nanoid 以保证全局唯一性。

per-node LLM 配置覆盖。LLM 配置最初只有全局默认一级。实践中发现不同节点对模型参数的需求差异显著——生成对话的节点可能需要较高的 temperature 以增加多样性，而生成大纲的节点需要较低的 temperature 以保持结构严谨。为此引入了

`preferences.json`, 存储在工作区级别, 允许用户按节点覆盖 temperature、maxTokens、topP 等参数，甚至指定完全不同的 LLM 配置。这一设计将工作流模板保持为纯结构定义（不含 LLM 配置），个性化偏好则跟随工作区而非模板。

双通道事件模型。State 中心化时期, 事件系统从单一的 EventBus 分化为 RunEvent 和 StateEvent 两个通道。RunEvent 报告临时的执行生命周期事件（`run:planned`、`node:start`、`level:done` 等），仅在执行期间存在；StateEvent 报告持久化的状态变更（`version:added`、`nodes:changed`、`targets:changed` 等），任何修改操作都会触发。两者均采用可序列化的 discriminated union 类型，使得事件流可以直接通过 JSON.stringify 发送至 WebSocket，也可以录制后回放用于测试和调试。在最新的原子操作重构中, RunEvent 和 RunHandle 已被移除, StateEvent 的职责由各原子函数的返回值隐式承担。

第 5 章 具体用例

5.1 用例一

5.2 用例二

第 6 章 总结与展望

参考文献

[1] VASWANI A, SHAZEER N, PARMAR N, 等. Attention Is All You Need[C/OL]//Advances in Neural Information Processing Systems: 卷 30. 2017. https://arxiv.org/abs/1706.03762.

[2] BROWN T B, MANN B, RYDER N, 等. Language Models are Few-Shot Learners[C/OL]//Advances in Neural Information Processing Systems: 卷 33. 2020: 1877-1901. https://arxiv.org/abs/2005.14165.

[3] OPENAI. GPT-4 Technical Report[EB/OL]. (2023). https://arxiv.org/abs/2303.08774.

[4] WEI J, WANG X, SCHUURMANS D, 等. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models[C/OL]//Advances in Neural Information Processing Systems: 卷 35. 2022. https://arxiv.org/abs/2201.11903.

[5] WHITE J, FU Q, HAYS S, 等. A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT[EB/OL]. (2023). https://arxiv.org/abs/2302.11382.

[6] KARPATHY A. Context Engineering[EB/OL]. (2025). https://x.com/karpathy/status/1937902205765607626.

[7] YANG K, KLEIN D, PENG N, 等. DOC: Improving Long Story Coherence With Detailed Outline Control[C/OL]//Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (ACL). 2023. https://arxiv.org/abs/2212.10077.

[8] WU Y, HEE M S, HU Z, 等. LongGenBench: Benchmarking Long-Form Generation in Long Context LLMs[C/OL]//Proceedings of the International Conference on Learning Representations (ICLR). 2025. https://arxiv.org/abs/2409.02076.

[9] HONG K, TROYNIKOV A, HUBER J. Context Rot: How Increasing Input Tokens Impacts LLM Performance[R/OL]. (2025-07). https://research.trychroma.com/context-rot.

[10] SCHLUNTZ E, ZHANG B. Building Effective Agents[EB/OL]. (2024). https://www.anthropic.com/research/building-effective-agents.

[11] YOUNG J. Effective Harnesses for Long-Running Agents[EB/OL]. (2025). https://www.anthropic.com/engineering/effective-harnesses-for-long-running-agents.

[12] ANTHROPIC. Managing Context on the Claude Developer Platform[EB/OL]. (2026). https://www.anthropic.com/news/context-management.

[13] BOLIN M. Unrolling the Codex Agent Loop[EB/OL]. (2026). https://openai.com/index/unrolling-the-codex-agent-loop/.

[14] CHOI D. Run Long Horizon Tasks with Codex[EB/OL]. (2026). https://developers.openai.com/blog/run-long-horizon-tasks-with-codex.

[15] FAN S, CONG X, FU Y, 等. WorkflowLLM: Enhancing Workflow Orchestration Capability of Large Language Models[C/OL]//Proceedings of the International Conference on Learning Representations (ICLR). 2025. https://arxiv.org/abs/2411.05451.

致　谢

感谢我的家人、老师和朋友。

原 创 性 声 明

本人郑重声明：所呈交的毕业设计（论文），是本人在指导教师的指导下，独立进行研究所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的科研成果。对本文的研究在做出重要贡献的个人和集体，均已在文中以明确方式标明。本人在导师指导下所完成的毕业设计（论文）及相关作品，知识产权归属贵州轻工职业技术学院。本人完全意识到本声明的法律责任由本人承担。

作者签名：_____　　日期：_____年____月____日

关于毕业设计（论文）使用授权的声明

本人完全了解贵州轻工职业技术学院有关保留、使用毕业设计（论文）的规定，同意学校保留或向国家有关部门或机构送交毕业设计（论文）的复印件和电子版，允许毕业设计（论文）被查阅和借阅；本人授权贵州轻工职业技术学院可以将本论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或其他复制手段保存毕业设计（论文）和汇编本论文。

本毕业设计（论文）属于：

保　密（　　），在_____年解密后适用授权。

不保密（　　）

(请在以上相应方框内打“√”)

作者签名：_____　　导师签名：_____

日　期：_____年____月____日