FlowCabal 架构设计文档 v4

日期：2026.02.17

浏览器 UI + 本地 Python 后端。浏览器负责工作流编辑和结果展示，本地 Python 负责执行、Agent、存储。

目录

FlowCabal 架构设计文档 v4 1
设计背景 2
  三层能力需求 2
  AI 辅助创作 2
  目标规模：超长篇小说 2
  参考设计 2
整体架构 3
  本地架构 3
  双存储架构 3
  职责划分 4
  模型配置 4
Agent 系统 4
  三角色 4
    Role C：低层级事实检查 5
  Prompt 组装 5
  执行流程（基础线性模式） 5
多角度侧写系统 6
  核心思想 6
  侧写类型 6
  侧写生成 6
  侧写在检索中的角色 7
策展管线 7
  原则 7
  两级模型 7
  策展时的一致性检查 7
  异步语义处理 8
  幂等性 8
高级工作流组件 8
  递归调用 8
  进化式迭代 8
  与基础 DAG 的关系 8
OpenViking 9
  虚拟文件系统结构 9
  层次化检索与意图分析 9
  实体关系追踪 10
  溯源追踪 10
WebSocket 协议 10
  Browser → Python 10
  Python → Browser 11

消息流 12
架构决策记录 12
为什么本地 Python 后端 12
为什么双存储（SQLite + OpenViking） 12
为什么 core-runner 在 Python 13
为什么多角度侧写而非实体状态机 13
为什么 Role C 只做低层级检查 13
目录结构 14
浏览器端 14
Python 后端 14
与 v3 设计的关系 14
实施路线 15
阶段一：基础设施 15
阶段二：Agent 核心 + 基础侧写 15
阶段三：高级能力 15
阶段四：打磨 16

设计背景

三层能力需求

paper.typ 提出了三层能力需求：

1. **DAG 工作流 + 执行引擎**：机械性工作
2. **高级查询函数 + 节点历史/冻结**：机械性工作
3. **Agent + 上下文管理**：自主代理监控和介入工作流运行

前两层是机械性实现。第三层是架构性决策。

AI 辅助创作

> 人类定义 **what**（意图、约束、美学标准）并 **评判质量**。AI 负责 **how**（执行、上下文组装、战术决策）。两者之间的结构/策略层—分解、编排、上下文选择—是 **共享领域**。

Agent 的角色：填充这个共享领域。不是纯执行器，不是自主创作者，而是 **结构层的协作者**。

未来考虑：Agent 系统可设计为可替换的（pluggable），允许用户选择不同的 Agent 实现或完全禁用 Agent。当前版本先硬编码三角色架构，降低实现复杂度。

目标规模：超长篇小说

FlowCabal 面向的是**高质量超长篇小说**—潜在达到百万字级别。在这个规模下：

- 任何可预见的上下文窗口都无法容纳完整手稿
- 即使上下文窗口足够大，注意力分配也会退化—模型无法在百万 token 中精准聚焦
- 检索基础设施不是权宜之计，是**必需品**

这决定了 OpenViking 的引入不是过度设计，而是核心基础设施。

参考设计

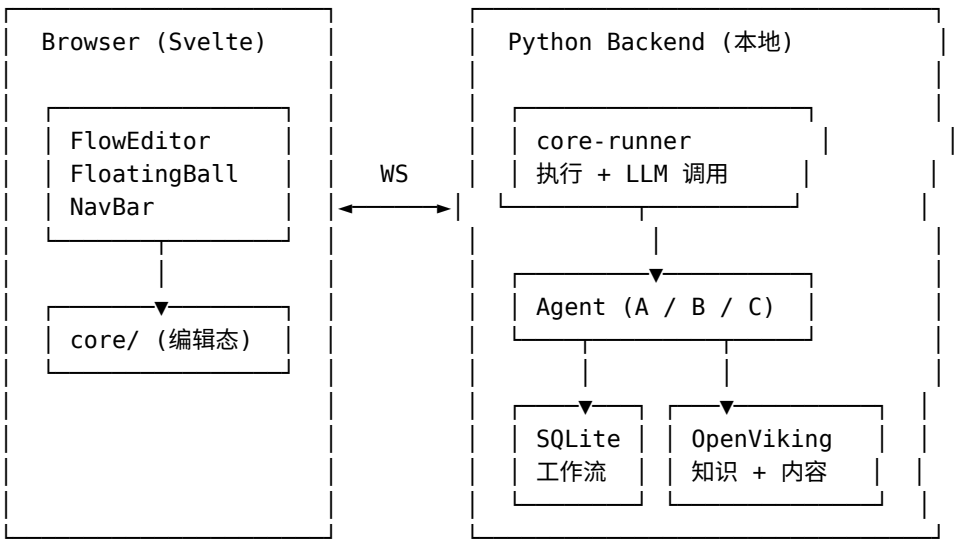
v4 设计整合了三个参考项目的关键设计（完整分析见 `docs/reference_design_analysis_zh.typ`）：

来源	采纳的设计	在 FlowCabal 中的体现
OpenViking	三级信息模型（L0/L1/L2）	多角度侧写系统的基础
OpenViking	层次化检索 + 意图分析	Role A 上下文组装
OpenViking	虚拟文件系统 + 实体关系追踪	直接使用 OpenViking 作为依赖
OpenClaw	策展持久化哲学 + 幂等性	策展管线的一致性检查
Trellis	规范注入优于记忆（临时性原则）	三层 prompt 组装
Trellis	质量门禁 + 重试循环	Role C 结构性评估

整体架构

本地架构

Python 后端运行在用户本地机器上。所有数据（API key、输出、项目知识）留在本机。



双存储架构

存储	技术	内容
SQLite	单一 `.sqlite` 文件	工作流定义、执行状态、策展元数据、配置
OpenViking	AGFS + LevelDB + 向量索引	手稿内容、多角度侧写、实体、摘要、检索索引

SQLite 管理**结构化工作流数据**。OpenViking 管理**知识和内容**。两者均在本地运行，数据留在用户机器上。

OpenViking 以嵌入模式运行（Python 进程内），AGFS 作为子进程自动启动。

职责划分

职责	Browser	Python
UI 渲染与交互	Yes	
工作流编辑（内存中）	Yes	
工作流持久化		SQLite
工作流执行（core-runner）		Yes
LLM API 调用		Yes（key 留在本机）
策展输出存储		OpenViking
Agent 推理		Yes
上下文管理		OpenViking（嵌入模式）
多角度侧写		Agent LLM + OpenViking
检索与索引		OpenViking

模型配置

配置	用途	典型模型
用户 LLM	工作流节点的创作生成	Claude Opus / GPT-4
Agent LLM	元推理：评估、侧写生成、检索意图分析	Claude Haiku / GPT-4o-mini
嵌入模型	OpenViking 向量化与语义检索	text-embedding-3-large

Agent LLM 同时用作 OpenViking 的 VLM 配置（用于 L0/L1 生成和意图分析），避免额外的模型配置。三套模型均在本地 Python 进程中管理。

Agent 系统

三角色

角色	触发	观察	推理	行动
B: Builder	执行前	用户意图 + 项目上下文	需要什么结构?	创建/修改节点和 prompt
A: Context	每节点前	当前节点 + 项目知识	需要什么上下文?	注入上下文到 prompt
C: Monitor	每节点后	节点输出 + 事实标准	有无事实错误?	批准 / 重试 / 交人类

三个角色共享 observe → reason → act 循环。因为 core-runner 和 Agent 在同一 Python 进程中，角色调用是函数调用，不经过网络。

Role C：低层级事实检查

Role C **只做低层级审核**：事实一致性、连续性错误、实体状态矛盾。不做创意判断—LLM 评估创意输出会偏好通用安全的写法，过滤掉有风险但有趣的选择。

多 Agent 交叉检查：多个 Role C 实例从不同角度（时间线、角色状态、世界观规则、前后文呼应）交叉验证，达成交叉信息意义上的"无过"—在可检查的维度上确保零错误。

人类终审：创意质量由人类评判。Role C 的职责边界清晰：捕捉机器可验证的错误，创意判断留给作者。

Prompt 组装

Agent 不修改用户的 TextBlockList（持久化 metadata）。上下文注入仅存在于执行期间。

```
Layer 1: 用户的 TextBlockList（持久化 metadata）
    ↓
Layer 2: Agent 上下文注入（同进程调用 Role A）
    ↓
Layer 3: VirtualTextBlock 解析（从内存缓存读取上游输出）
    ↓
最终 prompt → LLM API
```

临时性原则（来自 Trellis）：上下文注入是纯函数—`(node_config, project_state) -> additional_context`。它读取项目状态但永不写入。在相同项目状态下重新运行节点，产生相同的上下文注入。上下文组装对其输入是确定性的，即使 LLM 输出是随机的。

唯一的写入通过策展管线（用户批准的输出）或 Role B（工作流拓扑变更，同样需要用户批准）发生。

```python
def build_prompt(node, cache, agent_ctx=None):
    system = resolve_text_blocks(node.system_prompt, cache)
    user = resolve_text_blocks(node.user_prompt, cache)
    if agent_ctx:
        system = agent_ctx.system_prefix + '\n' + system
        user = user + '\n' + agent_ctx.user_suffix
    return system, user
```

Role A 的上下文注入是**性能最关键的操作**。每个节点执行都以 Role A 决定注入什么上下文开始。

执行流程（基础线性模式）

```
用户："写第三章"
    │
    ▼
[Role B] 分析项目上下文，生成节点 + 连接建议 → 推送浏览器 → 用户审批
    │
    ▼
Browser → workflow:run → Python
    │
    ▼
[core-runner] 按 Kahn 序执行（Python 进程内）
    │
    ├─ 节点 K:
    │   [Role A] 查询 OpenViking → 返回上下文
```

```
    │   组装 prompt → 调用用户 LLM → 流式推送到浏览器
    │   [Role C] 多角度交叉检查 → approve / retry / flag
    │   如果 flag → 推送 node:needs-human → 等待 human:decision
    │
    └─ 下一节点...
    │
    ▼
workflow:completed → 浏览器展示所有输出
    │
    ▼
用户选择保留 → output:persist → OpenViking 入库 → 异步侧写更新
```

这是基础线性执行。高级模式（递归调用、进化式迭代）见第 6 节。

多角度侧写系统

核心思想

L0/L1/L2 不局限于章节摘要。对整个作品进行**多角度侧写**—任何维度的内容都可以建立侧写：角色、情节线、世界状态、主题、文风、情感弧线。

每个侧写本身是 OpenViking 中的一个资源，自动获得 L0/L1/L2 三级信息层。侧写在策展输出变更时重新生成，是策展管线的一部分。

多角度侧写使 Role A 能从不同视角组装上下文：写对话场景时拉取角色侧写，写战斗场景时拉取世界状态侧写，写情感转折时拉取情感弧线侧写。

侧写类型

侧写类型	虚拟路径	内容
角色侧写	`/profiles/characters/`	性格特征、关系网络、发展弧线、当前状态
情节线侧写	`/profiles/plot-threads/`	起源、发展阶段、涉及角色、悬念状态
世界状态侧写	`/profiles/world-state/`	世界观规则的当前生效状态、已揭示的设定
主题侧写	`/profiles/themes/`	作品主题的展现方式、隐喻模式、象征系统
文风侧写	`/profiles/style/`	叙事视角、语言风格特征、节奏模式

每种侧写的 L0/L1/L2 内容不同。例如，角色侧写的 L0 是“主角—倔强的少年剑士，正经历从复仇到释怀的转变”；L1 是结构化的角色档案（外貌、性格、关系、当前状态）；L2 是完整的角色分析（跨所有章节的发展轨迹）。

侧写生成

侧写由策展管线触发，非手动操作：

1. 用户策展一章输出 → `output:persist`
2. 系统立即存储内容到 OpenViking → 返回 `output:persisted`
3. **异步**：Agent LLM 分析新策展内容，识别受影响的侧写
4. **异步**：重新生成受影响的侧写，写入 OpenViking
5. OpenViking 自动重新生成侧写的 L0/L1，更新向量索引

用户获得即时确认，侧写在后台更新。下次执行时，Role A 使用最新侧写。

侧写由 Agent LLM 生成。Agent LLM 理解小说创作的领域知识，生成的侧写比通用 VLM 更准确。

侧写在检索中的角色

Role A 使用侧写作为**主要导航机制**：

1. **有界上下文预算**：侧写将整部作品压缩为固定大小的表示。无论手稿多长，角色侧写的 L0 始终是 ~50 token。
2. **O(1) 访问**：随着作品增长，侧写提供对任何维度的常数时间访问。不需要扫描全部章节来了解某个角色的当前状态。
3. **多角度组装**：写一个场景时，Role A 可以同时拉取相关角色的 L1、当前情节线的 L1、世界状态的 L0——从多个投影角度组装上下文。

当手稿增长到数百章时，Role A 的检索策略：

```
1. 读取相关侧写的 L1（角色、情节线、世界状态）      ~5K token
2. 扫描全部章节 L0（每章 ~50 token）                  ~10-15K token
3. 读取最相关章节的 L1                                 ~2-4K token
4. 深读 1-2 章的 L2                                    ~6K token
5. 始终包含：/meta（大纲、风格指南、世界规则）        ~2K token
总预算：~25-30K token
```

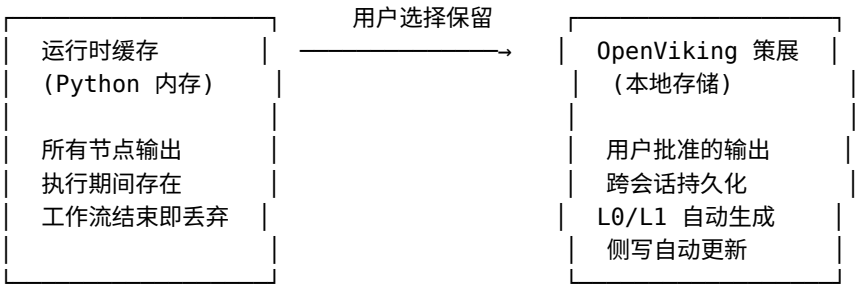
当章节数超过 ~300 时，L0 扫描本身变得昂贵。此时切换到**先扫描弧线级别摘要**，再在选定弧线内进行章节级别扫描。这是时间压缩原则在叙事结构上的应用。

策展管线

原则

并非所有节点输出都值得长期保存。**只有用户明确选择保留的输出**才进入 OpenViking，也只有这些内容才触发侧写更新和索引。

两级模型



- **运行时缓存**：Python 内存中，供下游 VirtualTextBlock 解析。执行结束后推送到浏览器供用户审阅。
- **OpenViking 策展存储**：内容存储在 OpenViking（AGFS）中，向量索引存储在 LevelDB 中。元数据（策展时间、关联节点、标签）存储在 SQLite 中。

策展时的一致性检查

当用户选择 `output:persist` 时，系统不盲目存储。Role C（或专门的策展步骤）执行：

- **矛盾检测**：检查与现有策展内容的矛盾（如角色外貌在章节间非意图地变化）

- **实体状态验证**：确认实体状态变更是有意的
- **自包含性**：输出是否足够自包含，可作为未来上下文使用

检查结果附加到 `output:persisted` 消息中，供用户参考。检查不阻止存储—最终决策权在用户。

异步语义处理

策展管线分离即时操作和异步处理：

1. **存储**（即时）：将策展输出写入 OpenViking，SQLite 记录元数据 → 返回 `output:persisted`
2. **语义处理**（异步）：
   - OpenViking 自动生成 L0/L1 摘要
   - Agent LLM 识别并重新生成受影响的侧写
   - 更新实体关系（通过 OpenViking 的 `link` API）
   - 向量索引自动更新

用户获得即时确认，无需等待语义处理完成。

幂等性

- 策展同一输出两次是空操作
- 侧写重新生成是幂等的：对相同输入产生相同结果或原地更新
- `output:persist` 处理程序在写入前检查现有内容

高级工作流组件

基础 DAG 执行是线性的：Kahn 排序 → 逐节点执行。但工作流 DAG 支持更高级的模式。本节仅描述概念和用例，实现细节推迟到实现阶段。

递归调用

是什么：一个节点或子工作流可以调用自身，参数经过修改。

用例：迭代精炼—起草一章 → 自我批评 → 修改 → 重复，直到满足质量标准或达到迭代上限。

关键约束：深度限制防止无限循环。每次递归调用携带递减的深度计数器。

进化式迭代

是什么：生成 N 个输出 → 评估 → 选择最优 → 在此基础上再迭代。类似进化算法的多输出-淘汰-迭代模式。

用例：探索创意可能性。生成 5 个版本的章节开头 → 多角度评估（Role C 交叉检查 + 人类选择）→ 保留最好的 → 以此为基础继续写作。

评估方式：可以是多 Agent 自动评估（Role C 从不同角度打分），也可以是人类选择。两者可组合：Agent 初筛淘汰明显不合格的，人类从候选中最终选择。

与基础 DAG 的关系

递归调用和进化式迭代是构建在基础 DAG 执行之上的**高级组件**。它们不破坏 DAG 模型—迭代和递归**在 DAG 内表达**，作为特殊的节点类型或子工作流模式。

详细的执行语义（节点状态机、迭代状态管理、UI 表示）推迟到实现阶段。

OpenViking

FlowCabal 直接使用 OpenViking（`pip install openviking`）作为知识和内容管理的基础设施。OpenViking 以嵌入模式（Python 进程内）运行，AGFS 子进程自动管理。

虚拟文件系统结构

```
viking://resources/project/
├── /meta
│   ├── outline.md
│   ├── style-guide.md
│   └── world-rules.md
├── /entities
│   ├── characters/
│   ├── locations/
│   └── plot-threads/
├── /manuscript
│   ├── chapter-01/
│   │   └── content.md
│   ├── chapter-02/
│   │   └── content.md
│   └── ...
├── /summaries
│   ├── arc-1.md
│   ├── arc-2.md
│   └── full-work.md
└── /profiles
    ├── characters/
    │   ├── protagonist.md
    │   └── antagonist.md
    ├── plot-threads/
    │   ├── revenge.md
    │   └── coming-of-age.md
    ├── world-state/
    │   └── current.md
    ├── themes/
    │   └── redemption.md
    └── style/
        └── narrative-voice.md
```

每个资源自动获得 L0（`.abstract.md`）和 L1（`.overview.md`），由 OpenViking 异步生成。

层次化检索与意图分析

Role A 的检索使用 OpenViking 的 `search()` API（带意图分析）：

```
# Role A 的检索管线
def get_context(node_id, node_config, project):
    # 步骤 1：意图分析（OpenViking IntentAnalyzer）
    #   "对话场景，第 12 章，角色：[主角，反派]"

    # 步骤 2：确定性包含（始终相关）
    ctx = [read(project.meta.style_guide), read(project.meta.outline)]

    # 步骤 3：实体查找（结构化路径构造）
    for char in intent.characters:
        ctx.append(read(f"/entities/characters/{char}"))

    # 步骤 4：侧写导航
```

```
    #    读取相关角色侧写 L1，情节线侧写 L1

    # 步骤 5：层次化手稿搜索
    #    扫描全部章节 L0 → 阅读有前景的 L1 → 深读最佳 L2

    # 步骤 6：弧线摘要（更广的叙事上下文）

    return ctx
```

OpenViking 的 `HierarchicalRetriever` 结合全局向量搜索和递归目录遍历，用 L0/L1 摘要决定深入哪些子树。这比扁平向量搜索更精确—它利用目录结构编码的领域知识。

有界上下文预算：Role A 在 ~25-30K token 预算内操作，无论手稿规模。

实体关系追踪

使用 OpenViking 的关系系统（`client.link()`）维护实体间关系：

```
/entities/characters/protagonist  --关联-->  /entities/locations/castle
/entities/characters/protagonist  --冲突-->  /entities/characters/antagonist
/manuscript/chapter-05            --引入-->  /entities/characters/mentor
/profiles/plot-threads/revenge    --涉及-->  /entities/characters/antagonist
```

关系在策展期间由 Agent 自动维护。当一章被策展时，Agent 分析实体变更并通过 `client.link()` 更新关系图。

Role A 沿关系链接追溯上下文：涉及主角 → 追溯主角关联的地点和冲突角色 → 包含相关实体文件。

溯源追踪

`node:completed` 消息中的 `contextSources` 扩展为结构化记录：

```
contextSources: {
  uri: string;           // 如 "viking://resources/project/entities/characters/
protagonist"
  reason: string;        // 如 "角色出现在场景中"
  level: 'L0' | 'L1' | 'L2';  // 读取了哪个层级
}[]
```

用户可以理解和调试上下文组装过程："为什么 AI 认为这一章需要关于 X 的上下文？"

WebSocket 协议

以**单向推送**为主。Python 内部完成执行 + Agent 调用，浏览器只接收结果和发送用户决策。

Browser → Python

```
interface BrowserToServer {
  'connect': { projectId: string };
  'disconnect': void;

  'workflow:save': { workflow: WorkflowDefinition };
  'workflow:load': { workflowId: string };
  'workflow:list': void;
  'workflow:run': {
    workflowId: string;
    apiConfig: { endpoint: string; apiKey: string;
                 model: string; parameters: ApiParameters };
```

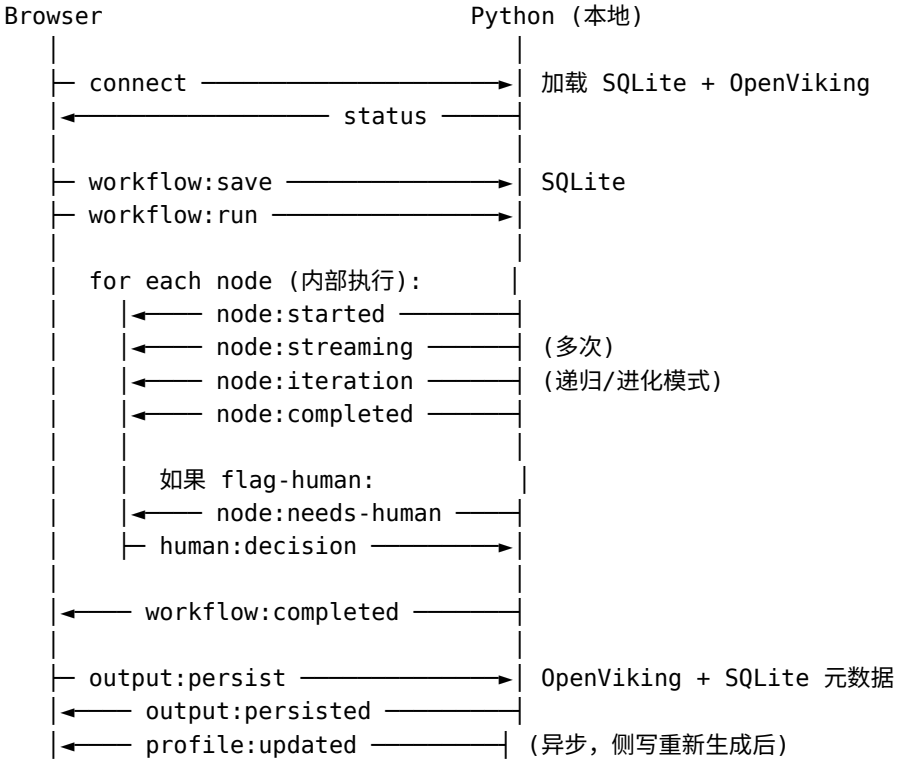
```
  };
  'workflow:cancel': void;

  'build:request': { intent: string; currentWorkflow?: string };
  'build:accept': { suggestionId: string; modifications?: string };
  'build:reject': { suggestionId: string; reason?: string };

  'output:persist': { nodeId: NodeId; tags?: string[] };
  'output:delete': { outputId: string };

  'human:decision': {
    nodeId: NodeId;
    decision: 'approve' | 'retry' | 'edit';
    editedOutput?: string;
  };
}
```

Python → Browser

```
interface ServerToBrowser {
  'workflow:data': { workflow: WorkflowDefinition };
  'workflow:list': { workflows: WorkflowSummary[] };

  'node:started': { nodeId: NodeId; nodeName: string };
  'node:streaming': { nodeId: NodeId; chunk: string };
  'node:completed': {
    nodeId: NodeId; output: string;
    evaluation: {
      decision: 'approve'|'retry'|'flag-human';
      confidence: number; reason: string;
      checks: { dimension: string; passed: boolean; detail: string }[];
    };
    contextSources: { uri: string; reason: string; level: 'L0'|'L1'|'L2' }[];
  };
  'node:needs-human': {
    nodeId: NodeId; reason: string; outputPreview: string;
    options: ('approve'|'retry'|'edit')[];
  };
  'node:iteration': {
    nodeId: NodeId;
    iteration: number; totalIterations: number;
    type: 'recursive' | 'evolutionary';
    candidateCount?: number;
  };

  'workflow:completed': { outputs: { nodeId: NodeId; output: string }[] };
  'workflow:error': { error: string; nodeId?: NodeId };

  'build:suggestion': {
    suggestionId: string; nodes: NodeDefinition[];
    edges: { source: NodeId; target: NodeId }[];
    explanation: string;
  };

  'output:persisted': {
    outputId: string; nodeId: NodeId;
    consistencyChecks?: { dimension: string; passed: boolean; detail: string }[];
```

```
  };
  'profile:updated': {
    profiles: { type: string; name: string; uri: string }[];
  };

  'status': { status: 'connected'|'busy'|'error'; message?: string };
}
```

消息流



架构决策记录

为什么本地 Python 后端

- API key 留在本机，无第三方信任问题
- 浏览器与 Python 同机通信，延迟可忽略
- 用户启动应用时 Python 进程随之启动

为什么双存储（SQLite + OpenViking）

维度	决策理由
工作流数据	结构化、关系型 → SQLite 更自然
知识/内容	需要 L0/L1/L2、向量检索、层次化组织 → OpenViking 已实现
避免重复实现	OpenViking 提供完整的检索、摘要、关系系统，无需在 SQLite 上重写
嵌入模式	OpenViking 可作为 Python 库直接使用，AGFS 作为子进程自动管理
备份	SQLite 复制文件 + OpenViking 数据目录复制（或 `ovpack` 导出）

曾考虑将所有数据统一存储在 SQLite 中（v3 设计），但这需要在 SQLite 上重新实现 OpenViking 的 L0/L1/L2 生成、层次化检索、向量索引和关系管理——工作量大且无必要。直接使用 OpenViking 作为依赖，换取成熟的知识管理基础设施。

为什么 core-runner 在 Python

维度	浏览器端	Python 端
Agent 集成	WebSocket 往返	函数调用
OpenViking 访问	跨进程	同进程（嵌入模式）
输出持久化	Browser→WS→Python→OpenViking	直接写入
WS 消息/节点	~4 次往返	~1 次推送
复杂度	分布式协调	单进程管线

```
for node_id in kahn_order:
    context = role_a.get_context(node_id)        # 函数调用 + OpenViking 检索
    prompt = build_prompt(node, context)           # 函数调用
    output = llm.generate(prompt)                  # 本地 HTTP
    evaluation = role_c.evaluate(output)           # 函数调用（多角度交叉检查）
    ws.push(node_id, output, evaluation)           # 推送 UI
```

为什么多角度侧写而非实体状态机

维度	实体状态机	多角度侧写
灵活性	刚性记录（角色在位置 Y）	任意角度的投影
扩展性	每种实体类型需要单独的 schema	新侧写类型只需新目录
与 L0/L1/L2 集成	独立系统	天然复用——侧写即资源
维护成本	代码维护状态机逻辑	Agent LLM 生成自然语言侧写
覆盖范围	仅实体	角色、情节、世界观、主题、文风…

多角度侧写是 L0/L1/L2 的泛化：不只是章节的摘要，而是整部作品的多个投影。

为什么 Role C 只做低层级检查

- LLM 评估创意输出会偏好通用安全的写法，抑制创意冒险
- 事实一致性是可验证的问题（角色眼睛颜色是否变了？时间线是否矛盾?）
- 创意质量是主观判断，应由人类作者最终把关
- 多 Agent 交叉检查在可检查维度上实现“无过”，而非试图评判“好不好”

目录结构

浏览器端

```
flow-cabal/src/lib/
├── core/               # 工作流编辑状态（通过 WS 同步到后端）
│   ├── textblock.ts
│   ├── node.ts
│   ├── workflow.ts
│   ├── apiconfig.ts
│   └── index.ts
├── ws/                 # WebSocket 客户端
│   ├── client.ts
│   ├── protocol.ts
│   └── index.ts
├── components/         # UI 组件
├── nodes/              # @xyflow 节点组件
└── utils/              # 布局、验证
```

不包含 core-runner、EventBus、db（IndexedDB）。

Python 后端

```
backend/
├── server.py               # WebSocket 服务器
├── config.py               # LLM + OpenViking 配置
├── protocol.py             # 消息类型（镜像 TS）
├── db.py                   # SQLite（工作流 + 元数据）
├── runner/
│   ├── engine.py           # core-runner（基础线性执行）
│   ├── prompt.py           # Prompt 组装
│   ├── cache.py            # 运行时输出缓存
│   ├── recursive.py        # 递归调用组件
│   └── evolution.py        # 进化式迭代组件
├── agent/
│   ├── core.py             # Agent 循环
│   ├── context.py          # Role A（使用 OpenViking 检索）
│   ├── builder.py          # Role B
│   ├── monitor.py          # Role C（多角度交叉检查）
│   └── skills/
│       ├── summarize.py
│       ├── retrieve.py
│       ├── evaluate.py
│       └── entity.py
├── viking/
│   ├── client.py           # OpenViking 客户端初始化与配置
│   ├── project.py          # 项目结构管理（目录初始化）
│   └── profiles.py         # 多角度侧写生成与管理
└── requirements.txt
```

与 v3 设计的关系

模块	v3	v4（本文档）
存储	SQLite 统一存储	SQLite（工作流）+ OpenViking（知识）

OpenViking	在 SQLite 上重新实现	直接使用 OpenViking 库（嵌入模式）
Role C	通用质量评估	仅低层级事实检查 + 多 Agent 交叉
L0/L1/L2	章节摘要	多角度侧写（泛化）
高级工作流	无	递归调用 + 进化式迭代
模型配置	两套 LLM	两套 LLM + 嵌入模型
检索	概念描述	使用 OpenViking HierarchicalRetriever
实体追踪	概念描述	使用 OpenViking 关系系统
溯源	`contextSources: string[]`	扩展为结构化记录（URI + reason + level）
core/	保留	保留，通过 WS 同步到后端
Agent 三角色	保留	保留，Role C 职责收窄
Prompt 组装	三层模型	保留，增加临时性原则
策展	两级模型	保留，增加一致性检查和异步处理

core/ 类型定义（NodeDefinition、WorkflowDefinition、TextBlockList）保持有效。

实施路线

阶段一：基础设施

1. Python WebSocket 服务器 + SQLite 初始化
2. OpenViking 嵌入模式集成 + 项目结构初始化
3. core-runner（Python，不含 Agent，先跑通基础执行）
4. 浏览器 WebSocket 客户端
5. 工作流同步：浏览器 → WS → Python → SQLite
6. 移除浏览器端 db/ 层

阶段二：Agent 核心 + 基础侧写

1. Agent 循环（observe → reason → act）
2. Role A: OpenViking 检索 + 上下文注入
3. Role C: 低层级事实检查（单 Agent 先行）
4. 策展管线：用户保留 → OpenViking 入库 → 异步 L0/L1 生成
5. 基础侧写：角色侧写、情节线侧写

阶段三：高级能力

1. Role B: 工作流构建建议
2. Role C: 多 Agent 交叉检查
3. 递归调用组件
4. 进化式迭代组件
5. 扩展侧写类型（世界状态、主题、文风）

阶段四：打磨

1. Agent 对话界面（FloatingBall）
2. 侧写管理 UI
3. 上下文来源可视化（溯源追踪）
4. 性能优化与错误恢复