# FlowCabal 架构设计文档 v3

**日期：2026.02.14**

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



---

# 设计背景

## 为什么需要 Agent

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。*

## 为什么需要 OpenViking

长篇小说涉及极重的设定管理：角色状态、情节线索、世界观规则、文风约束。8 万字以上的小说，每个节点生成时需要的上下文是 **动态的**、**层次化的**、**可追溯的**。

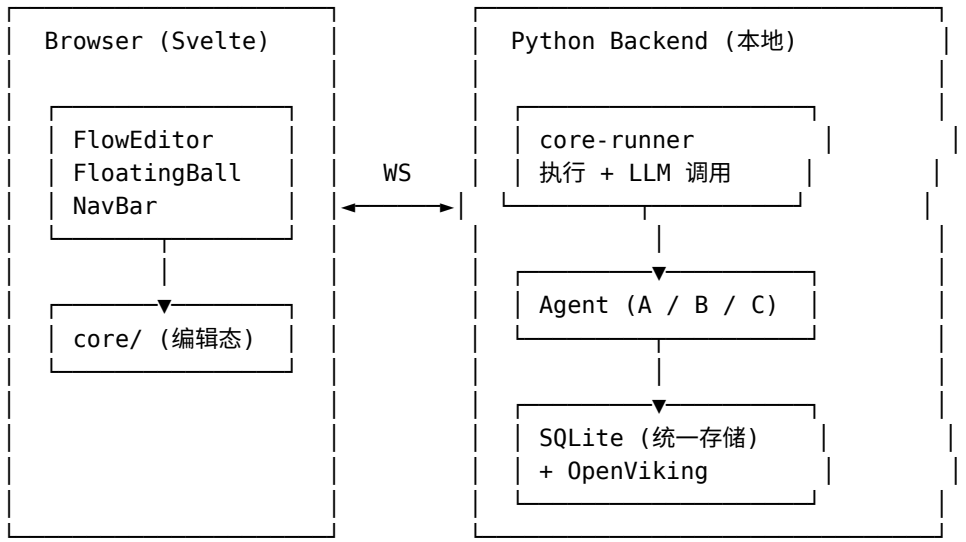
OpenViking 提供：虚拟文件系统语义、多级摘要、递归式上下文检索。

---

# 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。




## 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 | | SQLite |
| Agent 推理 | | Yes |
| 上下文管理（OpenViking） | | SQLite 直接读写 |
| 实体追踪 / 摘要 | | Yes |

### 两套 LLM

| 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、摘要、检索、建议 | Claude Haiku / GPT-4o-mini |

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 + 质量标准 | 质量达标? | 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。

### 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
```

```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
```

### 执行流程

```
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 SQLite/OpenViking → 返回上下文
    │   组装 prompt → 调用用户 LLM → 流式推送到浏览器
    │   [Role C] 评估质量 → 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 → SQLite → OpenViking 索引 + 摘要
```

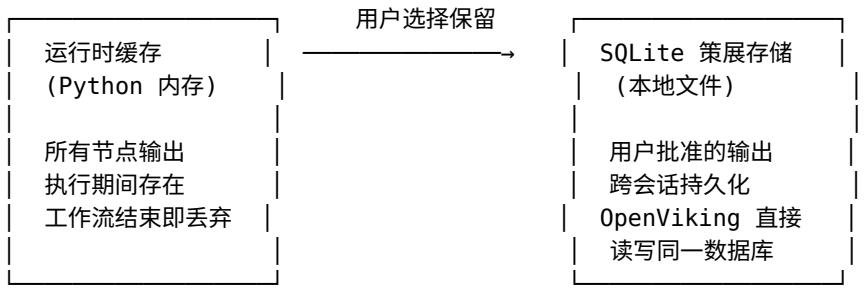
---

## 策展输出存储

### 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 SQLite，也只有这些内容才触发 OpenViking 摘要和索引。

### 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。
- **SQLite 策展存储**：单一数据库文件。工作流、策展输出、OpenViking 知识数据共存。OpenViking 直接读写同一实例。

---

# OpenViking

## 虚拟文件系统结构

```
/project
├── /meta
│   ├── outline.md
│   ├── style-guide.md
│   └── world-rules.md
├── /entities
│   ├── characters/
│   ├── locations/
│   └── plot-threads/
├── /manuscript
│   ├── chapter-01/
│   │   ├── content.md
│   │   ├── summary-paragraph.md
│   │   ├── summary-sentence.md
│   │   └── entity-changes.json
│   └── ...
└── /summaries
    ├── arc-1.md
    ├── arc-2.md
    └── full-work.md
```

以 SQLite 为存储后端。虚拟路径是逻辑概念，物理存储在 SQLite 表中。

## 检索示例

```
retrieved = viking.retrieve(
    query="第12章对话场景所需上下文",
    node_context={"chapter": 12, "scene_type": "dialogue",
                  "characters": ["protagonist", "antagonist"]}
)
# /summaries/full-work.md, /summaries/arc-3.md,
# /manuscript/chapter-11/summary-paragraph.md,
# /entities/characters/protagonist.md, ...
```

检索来源路径通过 `node:completed.contextSources` 推送到浏览器，确保可追溯性。

---

# 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

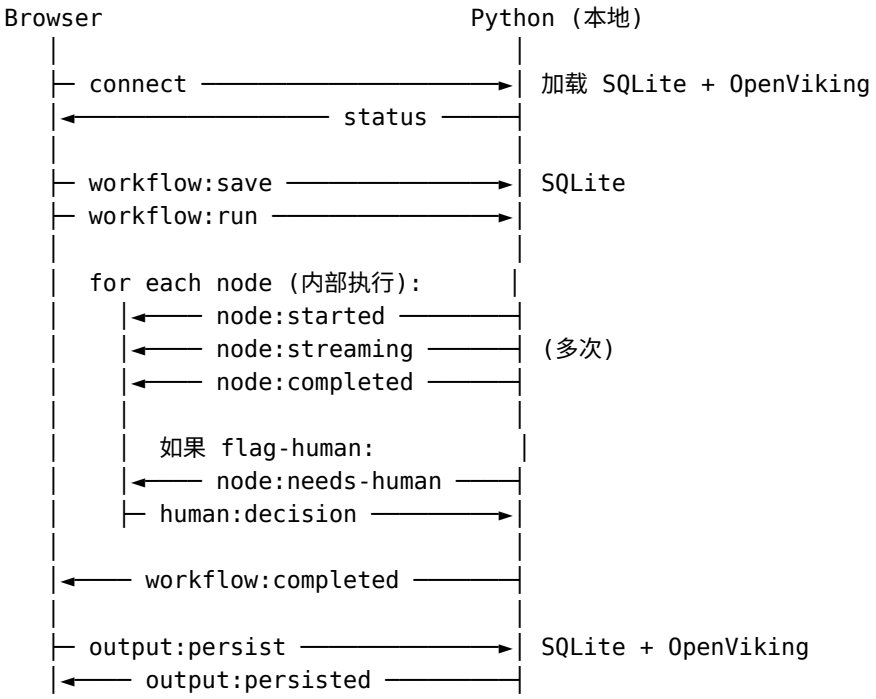
```
interface BrowserToServer {
  'connect': { projectId: string };
  'disconnect': void;

  '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 };
    contextSources: string[];
  };
  'node:needs-human': {
    nodeId: NodeId; reason: string; outputPreview: string;
    options: ('approve'|'retry'|'edit')[];
  };

  '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; updatedPaths: 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 SQLite 而非 IndexedDB

所有需要持久化的数据（工作流、策展输出、OpenViking 知识）都在 Python 进程中产生和消费。

| 维度 | IndexedDB | SQLite |
|---|---|---|
| 与 core-runner | 跨进程 | 同进程 |
| 与 OpenViking | 跨进程 | 同进程 |
| 查询能力 | 键值/索引 | SQL + FTS5 |
| 存储层数量 | 2 层 | 1 层 |
| 浏览器复杂度 | Dexie + persisted rune | 无 |
| 备份 | 需专门实现 | 复制文件 |

### 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 |
| 输出持久化 | Browser→WS→Python→SQLite | 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
    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
    ws.push(node_id, output, evaluation)          # 推送 UI
```

---

# 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 配置
├── protocol.py             # 消息类型（镜像 TS）
├── db.py                   # SQLite
├── runner/
│   ├── engine.py           # core-runner
│   ├── prompt.py           # Prompt 组装
│   └── cache.py            # 运行时输出缓存
├── agent/
│   ├── core.py             # Agent 循环
│   ├── context.py          # Role A
│   ├── builder.py          # Role B
│   ├── monitor.py          # Role C
│   └── skills/
│       ├── summarize.py
│       ├── retrieve.py
│       └── evaluate.py
```

```
|       └── entity.py
├── viking/
|   ├── adapter.py          # OpenViking（读写 SQLite）
|   └── project.py          # 项目结构管理
└── requirements.txt
```

---

## 与 v2 设计的关系

| 模块 | v2（design_doc.typ） | v3（本文档） |
|---|---|---|
| core/ | 浏览器端 metadata | 保留，通过 WS 同步到后端 |
| core-runner/ | 浏览器端 | 迁移至 Python |
| EventBus | 浏览器内 pub/sub | 移除 |
| db/ | IndexedDB | 移除，改为 SQLite |
| Agent | 无 | 三角色（A/B/C），同进程 |
| 上下文管理 | 无 | OpenViking + SQLite |
| LLM 调用 | 浏览器端 | Python 端（双配置） |
| 部署 | 纯浏览器 | 浏览器 + 本地 Python |

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. core-runner（Python，不含 Agent，先跑通基础执行）
3. 浏览器 WebSocket 客户端
4. 工作流同步：浏览器 → WS → Python → SQLite
5. 移除浏览器端 db/ 层

### 阶段二：Agent 核心

1. Agent 循环（observe → reason → act）
2. Role A: OpenViking + SQLite + 上下文注入
3. Role C: 质量评估 + 重试
4. 策展存储：用户保留 → SQLite → OpenViking 索引

### 阶段三：高级能力

1. Role B: 工作流构建建议
2. 高级查询函数（递归/迭代 LLM）
3. 节点输出历史 + 冻结
4. 子工作流

## 阶段四：打磨

1. Agent 对话界面（FloatingBall）
2. OpenViking 项目管理 UI
3. 上下文来源可视化
4. 性能优化与错误恢复