FlowCabal 架构设计文档 v3 – Agent 架构

日期：2026.02.13

本文档基于 v2 设计文档（EventBus + core-runner）的进一步演进。核心变化：**引入 Python 后端，Agent 系统，OpenViking 上下文管理**。FlowCabal 从浏览器单体应用升级为客户端-服务端架构。

目录

架构决策背景

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**：design_doc.typ 已覆盖，机械性工作
2. **高级查询函数 + 节点历史/冻结**：扩展 DAG 的控制流原语，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 辅助创作的定义

> 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**，人类与 AI 都不能独立胜任。

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**。

为什么必须使用 OpenViking

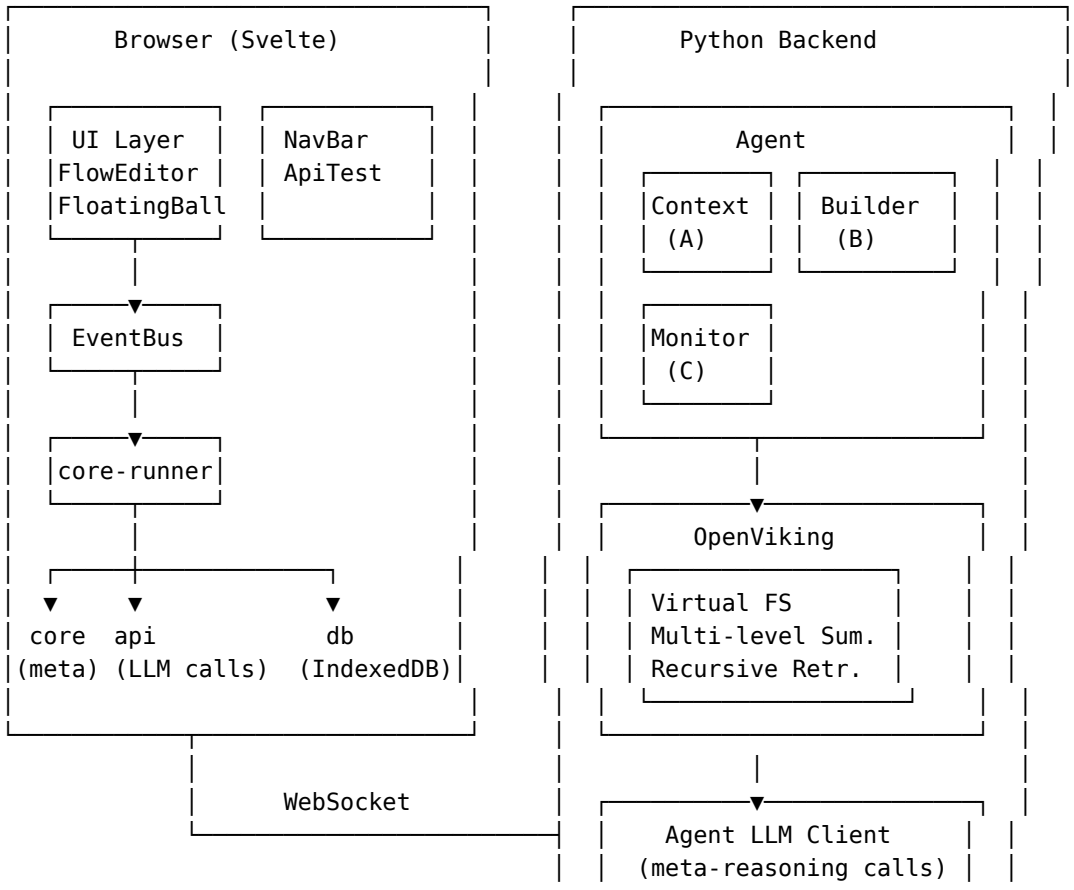
长篇小说涉及极重的设定管理：角色状态、情节线索、世界观规则、文风约束。一部 8 万字以上的小说，每个节点生成时需要的上下文是 **动态的、层次化的、可追溯的**。

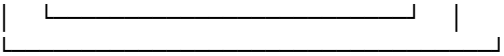
浏览器端的简易上下文方案（IndexedDB + Embeddings API）无法满足：
- 缺乏虚拟文件系统语义，无法追踪上下文来源
- 缺乏多级摘要能力
- 缺乏递归式上下文检索

OpenViking 是 Python 项目，因此 FlowCabal 必须引入 Python 后端。

整体架构

客户端-服务端分离





职责划分

职责	Browser	Python
UI 渲染与交互	Yes	
工作流 metadata 编辑	Yes	
工作流持久化（IndexedDB）	Yes	
LLM API 调用（用户的 key）	Yes	
工作流执行引擎（core-runner）	Yes	
Agent 推理（meta-reasoning）		Yes
上下文管理（OpenViking）		Yes
输出质量评估		Yes
工作流构建建议		Yes
实体追踪/摘要		Yes

两套 LLM 配置

配置	用途	位置	典型模型
用户 LLM 配置	工作流节点的创作生成	Browser（API key 不离开客户端）	Claude Opus / GPT-4
Agent LLM 配置	元推理：评估、摘要、检索、建议	Python Backend	Claude Haiku / GPT-4o-mini

用户的 API key 始终留在浏览器端，不传输给后端。Agent 使用独立的（通常更廉价的）模型进行元推理。

Agent 系统设计

统一 Agent 循环

三个角色（A/B/C）共享同一个 observe → reason → act 循环，由不同事件触发：

角色	触发时机	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	执行中（每节点前）	当前节点 + 项目知识	此节点需要什么上下文?	注入上下文到 prompt

C: Monitor	执行中（每节点后）	节点输出 + 质量标准	质量是否达标?	批准 / 重试 / 交由人类

Prompt 组装管线

Agent 不修改用户的 TextBlockList（那是持久化的 metadata）。上下文注入是 **临时的**，仅存在于执行期间：

```
Layer 1: 静态 prompt（用户的 TextBlockList — 持久化 metadata）
    ↓
Layer 2: Agent 上下文注入（临时，仅本次执行）
    ↓
Layer 3: VirtualTextBlock 解析（替换为上游节点输出）
    ↓
最终 prompt 字符串 → LLM API
```

```
// core-runner 中的 prompt 组装
function buildPrompt(
  node: NodeDefinition,
  runtimeState: WorkflowRuntimeState,
  agentContext?: AgentContext  // 来自 Python Agent 的注入
): { system: string; user: string } {
  let system = resolveTextBlockList(node.apiConfig.systemPrompt, runtimeState);
  let user = resolveTextBlockList(node.apiConfig.userPrompt, runtimeState);

  if (agentContext) {
    system = agentContext.systemPrefix + '\n' + system;
    user = user + '\n' + agentContext.userSuffix;
  }

  return { system, user };
}
```

执行流程（三角色协作）

```
用户: "写第三章"
    │
    ▼
[Role B] Agent 分析项目上下文
    → 从 OpenViking 检索: 第1-2章摘要、角色状态、大纲
    → 推理: "第三章需要4个节点: 场景铺设、对话、冲突、收束"
    → 生成 NodeDefinition + 连接建议
    → 通过 WebSocket 发送到浏览器展示
    → 用户审批（可能修改）
    │
    ▼
用户批准 → workflow:run
    │
    ▼
[core-runner] 按 Kahn 序执行
    │
    ├─ 节点 K 执行前:
    │   core-runner → ws → Python Agent
    │   [Role A] Agent 查询 OpenViking
    │       → "节点 K 正在写第三章的对话场景"
    │       → 检索: 角色语音样本、进行中的情节线索、第2章结尾
    │       → 返回 AgentContext
```

```
│   core-runner ← ws ← Python Agent
│   core-runner 注入上下文，调用用户的 LLM
│
├─ 节点 K 执行后:
│   core-runner → ws → Python Agent
│   [Role C] Agent 评估输出质量
│       → 检查: 连续性、风格一致性、prompt 遵循度
│       → 决策: approve / retry(附修改建议) / flag(交由人类)
│   [Role A] Agent 更新 OpenViking
│       → 索引节点 K 输出、更新实体状态、生成摘要
│   core-runner ← ws ← Python Agent
│
└─ 下一个节点...
```

WebSocket 协议

EventBus 通过 WebSocket 桥接浏览器和 Python 后端。以下是完整的消息类型定义。

Browser → Python

```typescript
interface AgentProtocol_BrowserToPython {
  // 连接管理
  'agent:connect': {
    workflowId: string;
    projectId: string;  // OpenViking 项目标识
  };
  'agent:disconnect': void;

  // Role B: 工作流构建请求
  'agent:build-request': {
    intent: string;           // 用户的自然语言意图
    currentWorkflow?: string; // 当前工作流 JSON（可选）
  };
  'agent:build-accept': {
    suggestionId: string;     // 接受某个建议
    modifications?: string;   // 用户的修改说明
  };
  'agent:build-reject': {
    suggestionId: string;
    reason?: string;
  };

  // Role A+C: 执行期间
  'agent:node-before': {
    nodeId: NodeId;
    nodeName: string;
    promptPreview: {          // 静态 prompt（未注入上下文）
      system: string;
      user: string;
    };
    dependencies: NodeId[];   // 此节点的上游依赖
  };
  'agent:node-output': {
    nodeId: NodeId;
    nodeName: string;
    output: string;           // 节点 LLM 输出
```

```
    usage?: UsageInfo;
  };

  // 人类介入响应
  'agent:human-decision': {
    nodeId: NodeId;
    decision: 'approve' | 'retry' | 'edit';
    editedOutput?: string;    // 如果 decision = 'edit'
  };
}
```

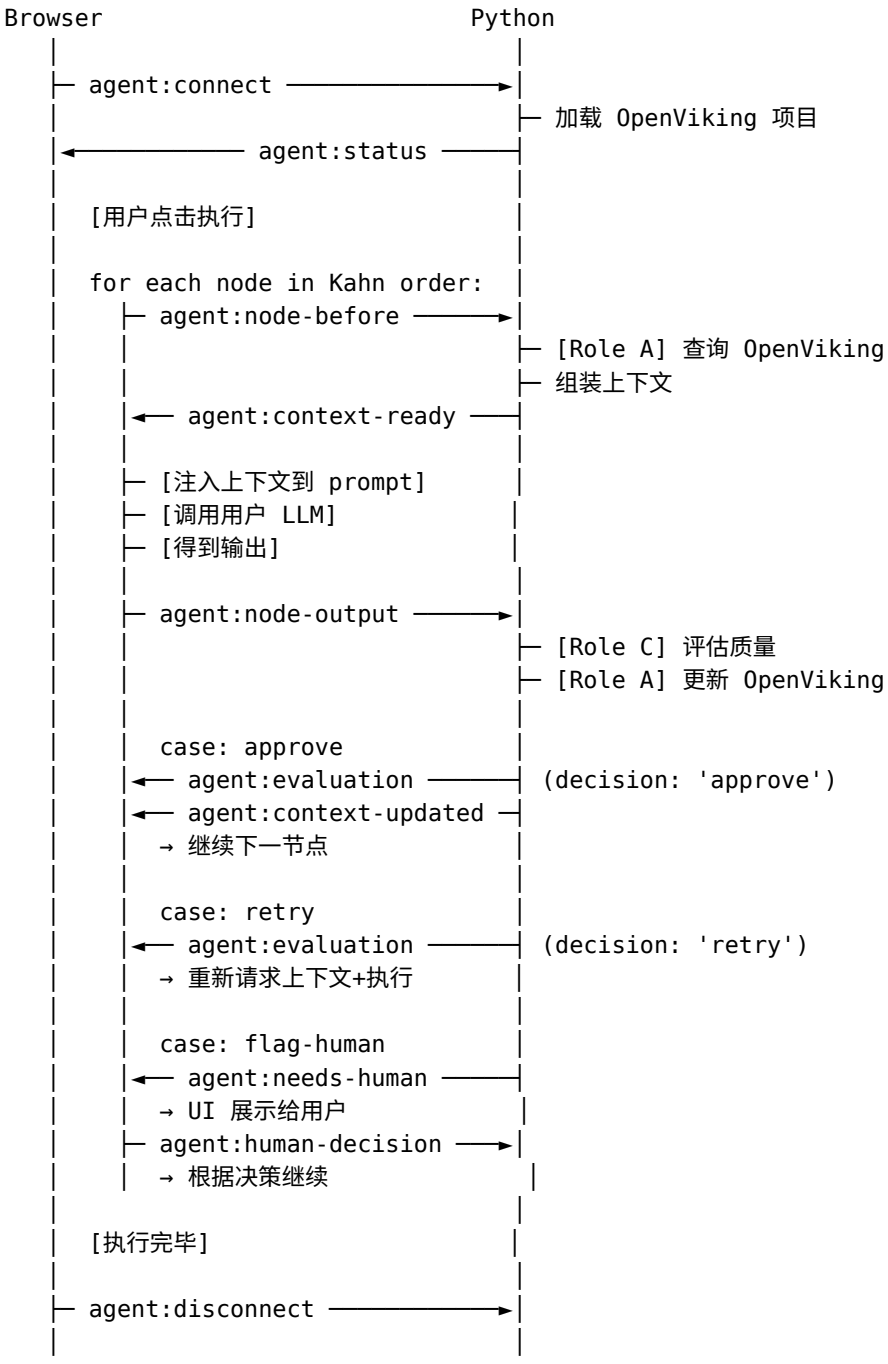
Python → Browser

```
interface AgentProtocol_PythonToBrowser {
  // Role B: 工作流建议
  'agent:build-suggestion': {
    suggestionId: string;
    nodes: NodeDefinition[];
    edges: { source: NodeId; target: NodeId }[];
    explanation: string;      // Agent 的解释
  };

  // Role A: 上下文注入
  'agent:context-ready': {
    nodeId: NodeId;
    context: {
      systemPrefix: string;   // 注入到 system prompt 前
      userSuffix: string;     // 注入到 user prompt 后
    };
    sources: string[];        // OpenViking 中的来源路径（可追溯性）
  };

  // Role C: 质量评估
  'agent:evaluation': {
    nodeId: NodeId;
    decision: 'approve' | 'retry' | 'flag-human';
    confidence: number;       // 0-1
    reason: string;
    retrySuggestion?: string; // 如果 retry，建议修改什么
  };

  // 上下文更新确认
  'agent:context-updated': {
    nodeId: NodeId;
    updatedPaths: string[];   // OpenViking 中更新的文件路径
    newSummaries: string[];   // 新生成的摘要路径
  };

  // 人类介入请求
  'agent:needs-human': {
    nodeId: NodeId;
    reason: string;
    outputPreview: string;
    options: ('approve' | 'retry' | 'edit')[];
  };

  // 状态与错误
```

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

消息流时序

完整执行流程



OpenViking 项目模型

长篇小说的虚拟文件系统结构

```
/project
├── /meta
│   ├── outline.md              # 全篇大纲
│   ├── style-guide.md          # 文风、语气、视角规则
│   └── world-rules.md          # 世界观设定与约束
├── /entities
│   ├── characters/
│   │   ├── protagonist.md      # 状态、关系、弧光
│   │   ├── antagonist.md
│   │   └── ...
│   ├── locations/
│   │   └── ...
│   └── plot-threads/
│       ├── main-thread.md
│       └── subplot-1.md
├── /manuscript
│   ├── chapter-01/
│   │   ├── content.md          # 全文
│   │   ├── summary-paragraph.md
│   │   ├── summary-sentence.md
│   │   └── entity-changes.json # 本章实体状态变更
│   ├── chapter-02/
│   └── ...
└── /summaries
    ├── arc-1.md                # 多章摘要
    ├── arc-2.md
    └── full-work.md            # 全书一页摘要
```

Agent 如何使用 OpenViking

当 Agent 为“第 12 章对话场景”节点准备上下文时，典型检索结果：

```
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"]
    }
)
# 可能返回:
# - /summaries/full-work.md          (全局概览)
# - /summaries/arc-3.md              (当前叙事弧)
# - /manuscript/chapter-11/summary-paragraph.md  (前章)
# - /entities/characters/protagonist.md
# - /entities/characters/antagonist.md
# - /meta/style-guide.md
```

每次检索的来源路径通过 `agent:context-ready.sources` 传回浏览器，确保可追溯性。

目录结构

浏览器端

```
flow-cabal/src/lib/
├── core/              # metadata 层（不变）
│   ├── textblock.ts
│   ├── node.ts
│   ├── workflow.ts
│   ├── apiconfig.ts
│   └── index.ts
├── core-runner/       # 执行引擎（新增）
│   ├── runner.ts      # WorkflowRunner
│   ├── prompt.ts      # Prompt 组装管线（含 Agent 注入）
│   ├── types.ts       # RuntimeState 类型
│   └── index.ts
├── bus/               # EventBus（新增）
│   ├── eventbus.ts
│   ├── events.ts      # 本地 + Agent 事件类型
│   └── ws-bridge.ts   # EventBus ↔ WebSocket 桥接
├── agent-client/      # Agent 客户端（新增）
│   ├── client.ts      # WebSocket 连接管理、重连、心跳
│   ├── types.ts       # AgentProtocol 类型
│   └── index.ts
├── api/               # LLM 客户端（不变）
├── db/                # 持久化（不变）
├── components/        # UI 组件
├── nodes/             # @xyflow 节点组件
└── utils/             # 布局、验证
```

Python 后端

```
backend/
├── server.py              # WebSocket 服务器
├── config.py              # Agent LLM 配置
├── protocol.py            # 消息类型定义（与 TS 类型镜像）
├── agent/
│   ├── core.py            # Agent 主循环（observe → reason → act）
│   ├── context.py         # Role A: OpenViking 查询 + 上下文组装
│   ├── builder.py         # Role B: 工作流构建建议
│   ├── monitor.py         # Role C: 输出质量评估
│   └── skills/
│       ├── summarize.py   # 多级摘要生成
│       ├── retrieve.py    # 上下文检索
│       ├── evaluate.py    # 质量评估
│       └── entity.py      # 实体追踪与状态管理
├── viking/
│   ├── adapter.py         # OpenViking 集成适配器
│   └── project.py         # 小说项目结构管理
└── requirements.txt
```

与之前设计的关系

模块	design_doc.typ (v2)	本文档 (v3)

core/	不变	不变
core-runner/	独立执行引擎	增加 Agent 上下文注入管线
EventBus	浏览器内 pub/sub	扩展 WebSocket 桥接到 Python
Agent	未涉及	三角色统一架构（A/B/C）
上下文管理	未涉及	OpenViking 后端
LLM 调用	浏览器端单一配置	双配置：用户 LLM + Agent LLM
部署模型	纯浏览器	客户端-服务端（WebSocket）

design_doc.typ 中的 EventBus 类型定义、core-runner 的 RuntimeState 类型、WorkflowRunner 类结构均保持有效，本文档在此基础上扩展。

实施路线

阶段一：基础设施

1. EventBus + WebSocket 桥接
2. core-runner（不含 Agent 注入，先跑通基础执行）
3. UI 层与 core 类型桥接（消除 @xyflow 直用问题）
4. Python WebSocket 服务器骨架

阶段二：Agent 核心

1. Agent 主循环（observe → reason → act）
2. Role A: OpenViking 集成 + 上下文检索 + 注入管线
3. Role C: 输出质量评估 + 重试逻辑
4. Prompt 组装管线增加 Agent 层

阶段三：Agent 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 模式）
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 扩展）
2. OpenViking 项目管理 UI
3. 上下文来源可视化（哪些 OpenViking 路径被注入）
4. 性能优化与错误恢复