OpenViking

AI 智能体的上下文数据库

综合技术报告

基于源代码分析生成
代码仓库：volcengine/OpenViking
许可证：Apache 2.0
2026 年 2 月 17 日

发起并维护
字节跳动火山引擎 Viking 团队

目录

1 引言

1.1 背景与动机

在大语言模型（LLM）和自主 AI 智能体的时代，输入模型的上下文质量往往比模型本身更为重要。AI 智能体——能够规划、执行多步任务并与外部工具交互的软件系统——面临一个根本性挑战：**管理其高效运作所需的、不断增长的异构上下文**。

传统的检索增强生成（RAG）管线将上下文视为存储在向量数据库中的扁平文本块。虽然这种方法足以应对简单的问答任务，但它无法满足复杂智能体工作流的需求，原因如下：

- **碎片化的上下文**：记忆存储在一处，资源存储在另一处，技能分散在各种工具定义中。没有统一的管理方式。
- **激增的上下文需求**：长时间运行的智能体任务在每一步执行中都会产生上下文。简单的截断或压缩会导致不可逆的信息丢失。
- **检索效果不佳**：扁平的向量搜索缺乏全局结构视图，难以理解任何给定信息的完整上下文。
- **不可观测的检索**：传统 RAG 隐含的检索链是一个黑盒，当错误发生时难以调试。
- **有限的记忆迭代**：当前的记忆系统仅仅是用户交互的记录，缺乏从智能体自身任务执行经验中学习的能力。

1.2 什么是 OpenViking？

OpenViking 是一个专为 AI 智能体设计的开源上下文数据库。由字节跳动火山引擎 Viking 团队开发和维护，于 2026 年 1 月在 Apache 2.0 许可证下开源。项目的标语简洁地概括了其使命：*"Data in, Context out."*

OpenViking 放弃了传统 RAG 的碎片化向量存储模型，创新性地采用**文件系统范式**来统一组织智能体所需的记忆、资源和技能的结构化管理。借助 OpenViking，开发者可以像管理本地文件一样构建智能体的大脑。

1.3 关键创新

OpenViking 引入了五项关键创新，以解决上述挑战：

1. **文件系统管理范式**：基于虚拟文件系统和 `viking://` URI 方案，统一管理记忆、资源和技能。
2. **分层上下文加载（L0/L1/L2）**：三层信息结构按需加载，显著降低 Token 消耗。
3. **目录递归检索**：结合目录定位与语义搜索，实现递归精准的上下文获取。
4. **可视化检索轨迹**：完整保留目录浏览和文件定位轨迹，实现可观测性。
5. **自动会话管理**：自动从对话中提取长期记忆，让智能体越用越聪明。

1.4 项目历史

字节跳动的 Viking 团队在非结构化信息处理方面有悠久的历史：

年份	里程碑
2019	VikingDB 向量数据库支撑字节跳动全业务大规模使用

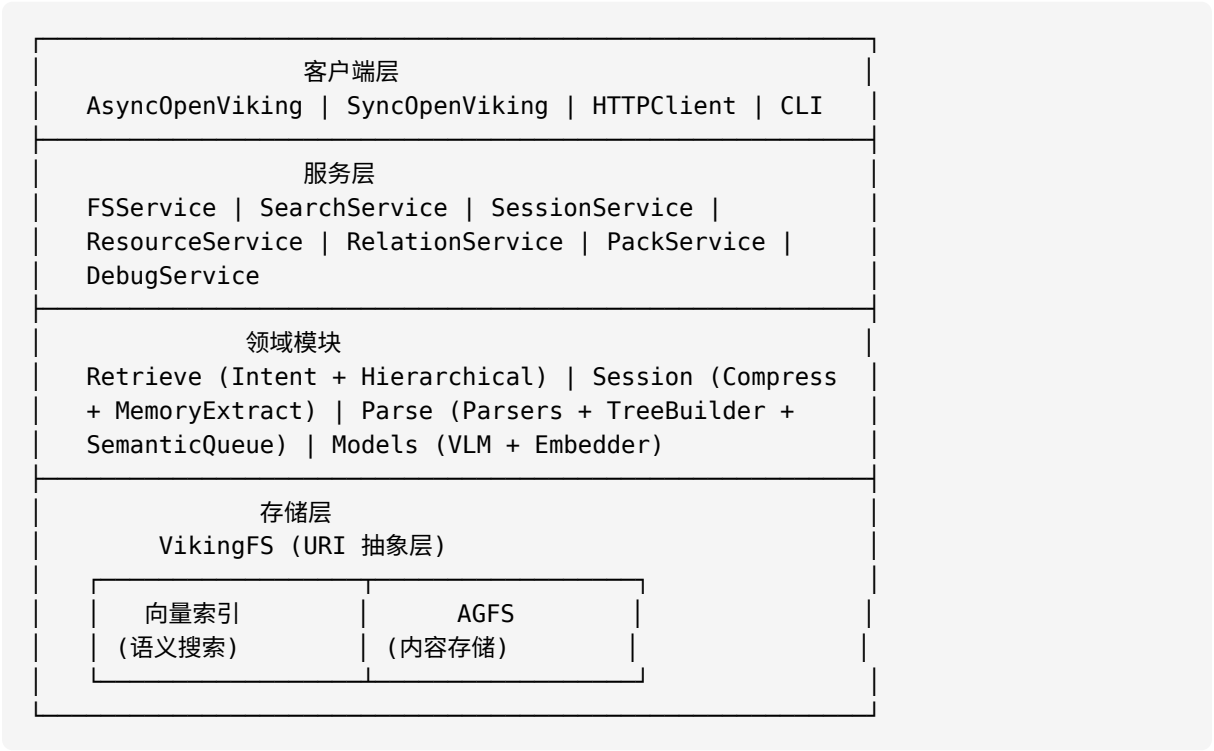
2023	VikingDB 在火山引擎公有云上销售
2024	推出开发者产品矩阵：VikingDB、Viking KnowledgeBase、Viking MemoryBase
2025	打造上层应用产品如 AI Search 和 Vaka 知识助手
2025 年 10 月	开源 MineContext，探索主动式 AI 应用
2026 年 1 月	开源 OpenViking，为 AI 智能体提供底层上下文数据库支持

2 系统架构

OpenViking 遵循分层、模块化的架构，将客户端接口、业务逻辑服务、领域模块和存储后端清晰分离。

2.1 架构概览

高层架构可以概括如下：



2.2 核心模块

模块	职责	关键能力
Client	统一入口点	提供所有操作接口；委托给服务层
Service	业务逻辑	FSService、SearchService、SessionService、ResourceService、RelationService、PackService、DebugService
Retrieve	上下文检索	意图分析（IntentAnalyzer）、层级化检索（HierarchicalRetriever）、Rerank
Session	会话管理	消息记录、使用追踪、压缩、记忆提交
Parse	上下文提取	文档解析（PDF/MD/HTML/DOCX/PPTX/XLSX/EPUB）、树构建、异步语义生成
Compressor	记忆压缩	6 类记忆提取、基于 LLM 的去重决策

Storage	存储层	VikingFS 虚拟文件系统、向量索引、AGFS 集成
Models	AI 模型集成	VLM 提供商（10+）、嵌入提供商、提供商注册表

2.3 服务层设计

服务层将业务逻辑与传输层解耦，使其可在 HTTP 服务器、嵌入式 SDK 和 CLI 中复用：

服务	职责	关键方法
FSService	文件系统操作	`ls`、`mkdir`、`rm`、`mv`、`tree`、`stat`、`read`、`abstract`、`overview`、`grep`、`glob`
SearchService	语义搜索	`search`、`find`
SessionService	会话管理	`session`、`sessions`、`commit`、`delete`
ResourceService	资源导入	`add_resource`、`add_skill`、`wait_processed`
RelationService	关系管理	`relations`、`link`、`unlink`
PackService	导入/导出	`export_ovpack`、`import_ovpack`
DebugService	调试与监控	Observer（队列、VikingDB、VLM、系统状态）

2.4 部署模式

OpenViking 支持两种部署模式：

2.4.1 嵌入模式

适用于本地开发和单进程应用。客户端自动启动 AGFS 子进程，使用本地向量索引，遵循单例模式：

```
import openviking as ov
client = ov.OpenViking(path="./data")
client.initialize()
```

2.4.2 HTTP 服务器模式

适用于团队共享、生产部署和跨语言集成。服务器作为独立的 FastAPI 进程运行：

```
# Python SDK 连接到 OpenViking 服务器
client = ov.SyncHTTPClient(url="http://localhost:1933", api_key="xxx")
client.initialize()
```

```
# 或使用 curl / 任意 HTTP 客户端
curl http://localhost:1933/api/v1/search/find \
```

```
  -H "X-API-Key: xxx" \
  -d '{"query": "how to use openviking"}'
```

HTTP 服务器支持 API Key 认证、CORS 中间件、请求计时和结构化 JSON 错误响应。可通过 `openviking-server` 或 `python -m openviking serve` 启动。

3 核心概念

3.1 上下文类型

OpenViking 将所有上下文抽象为三种基本类型，基于对人类认知模式的简化映射：

3.1.1 资源

资源是智能体可以引用的外部知识。它们是用户驱动的、静态的（添加后内容很少变化）、结构化的（按项目或主题在目录层级中组织）。示例包括 API 文档、产品手册、代码仓库、研究论文和技术规范。

```
client.add_resource(
    "https://docs.example.com/api.pdf",
    reason="API documentation"
)
```

3.1.2 记忆

记忆分为用户记忆和智能体记忆，代表关于用户和世界的已学习知识。它们是智能体驱动的（由智能体主动提取和记录）、动态更新的（从交互中持续更新）、个性化的（为特定用户或智能体学习）。

OpenViking 将记忆分为六个类别：

类别	位置	描述	可合并
profile	`user/memories/profile.md`	用户基本信息和身份	是
preferences	`user/memories/preferences/`	按主题组织的用户偏好	是
entities	`user/memories/entities/`	实体记忆（人物、项目、概念）	是
events	`user/memories/events/`	事件记录（决策、里程碑）	否
cases	`agent/memories/cases/`	智能体学习的具体问题+解决方案	否
patterns	`agent/memories/patterns/`	智能体发现的可复用流程/方法	是

3.1.3 技能

技能是智能体可以调用的能力，如工具定义、MCP 工具等。它们是已定义的能力（完成特定任务的工具定义）、相对静态的（定义在运行时不变）、可调用的（智能体决定何时使用哪个技能）。

```
await client.add_skill({
    "name": "search-web",
    "description": "Search the web for information",
    "content": "# search-web\n..."
})
```

3.2 上下文层级（L0/L1/L2）

OpenViking 的三层信息模型是其效率的基础。内容不是一次性将大量上下文塞入提示中，而是自动处理为三个渐进式详细级别：

层级	名称	文件	Token 预算	用途
L0	摘要	`.abstract.md`	~100 tokens	向量搜索、快速过滤、一句话摘要
L1	概览	`.overview.md`	~2k tokens	Rerank 评分、内容导航、智能体决策
L2	详情	原始文件	无限制	完整内容、按需加载用于深度阅读

3.2.1 生成机制

L0 和 L1 在资源添加后异步生成。`SemanticProcessor` 自底向上遍历目录，为每个目录生成 L0/L1。子目录的 L0 摘要被聚合到父目录的 L1 概览中，形成层级导航结构。

```
叶子节点 → 父目录 → 根目录（自底向上）
```

3.2.2 多模态支持

- **L0/L1**：始终为文本（Markdown），即使对于非文本内容
- **L2**：可以是任何格式（文本、图片、视频、音频）

对于图片或视频等二进制内容，L0/L1 提供由视觉语言模型（VLM）生成的文本描述。

3.3 Viking URI

Viking URI 是 OpenViking 中所有内容的统一资源标识符。每一条上下文——无论是资源文件、记忆条目还是技能定义——都有唯一的 URI。

3.3.1 格式

```
viking://{scope}/{path}
```

3.3.2 作用域

作用域	描述	生命周期	可见性
`resources`	独立资源（文档、代码仓库、网页）	长期	全局
`user`	用户级数据（个人资料、偏好、记忆）	长期	全局
`agent`	智能体级数据（技能、指令、记忆）	长期	全局
`session`	会话级数据（消息、工具、历史）	会话生命周期	当前会话
`queue`	处理队列	临时	内部

temp	解析期间的临时文件	解析期间	内部

3.3.3 初始目录结构

```
viking://
├── resources/{project}/       # 资源工作空间
│   ├── .abstract.md
│   ├── .overview.md
│   └── {files...}
├── user/
│   ├── .overview.md           # 用户档案
│   └── memories/
│       ├── preferences/       # 用户偏好
│       ├── entities/          # 实体记忆
│       └── events/            # 事件记录
├── agent/
│   ├── skills/                # 技能定义
│   ├── memories/
│   │   ├── cases/
│   │   └── patterns/
│   └── instructions/
└── session/{session_id}/
    ├── messages.jsonl
    ├── .abstract.md
    ├── .overview.md
    └── history/
```

4 存储架构

OpenViking 使用双层存储架构，将内容存储与索引存储分离。

4.1 VikingFS：虚拟文件系统

VikingFS 是统一的 URI 抽象层，隐藏底层存储细节。它将 Viking URI 映射到物理存储路径，并提供 POSIX 风格的 API：

方法	描述
`read(uri)`	读取文件内容
`write(uri, data)`	写入文件
`mkdir(uri)`	创建目录
`rm(uri)`	删除文件/目录（同步向量删除）
`mv(old, new)`	移动/重命名（同步向量 URI 更新）
`abstract(uri)`	读取 L0 摘要
`overview(uri)`	读取 L1 概览
`relations(uri)`	获取关系列表
`find(query, uri)`	语义搜索

VikingFS 自动维护向量索引和 AGFS 之间的一致性。当资源被删除时，所有对应的向量记录被移除。当资源被移动时，向量索引中的所有 URI 引用被更新。

4.2 AGFS：内容存储后端

AGFS（Agent Graph File System）提供 POSIX 风格的文件操作，支持多种后端。它实现为基于 Go 的服务器，在嵌入模式下作为子进程运行或作为共享服务。

后端	描述	配置
`localfs`	本地文件系统存储	`path`
`s3fs`	S3 兼容对象存储	`bucket`、`endpoint`
`memory`	内存存储（用于测试）	—

4.3 向量索引：语义层

向量索引存储语义索引用于快速检索。它支持稠密向量、稀疏向量和混合搜索。

4.3.1 Collection Schema

字段	类型	描述
id	string	主键
uri	string	资源 URI
parent_uri	string	父目录 URI
context_type	string	resource / memory / skill
is_leaf	bool	是否为叶子节点（文件 vs 目录）
vector	vector	稠密嵌入向量
sparse_vector	sparse_vector	稀疏嵌入向量
abstract	string	L0 摘要文本
name	string	资源名称
description	string	描述
created_at	string	创建时间戳
active_count	int64	使用计数

4.3.2 索引配置

```
index_meta = {
    "IndexType": "flat_hybrid",  # 混合稠密+稀疏索引
    "Distance": "cosine",        # 余弦相似度
    "Quant": "int8",             # INT8 量化
}
```

4.3.3 C++ 高性能引擎

向量索引和键值存储使用 C++17 实现以获得最大性能，通过 pybind11 暴露给 Python。C++ 引擎包括：

- **IndexEngine**：支持稠密和稀疏向量的混合向量索引，提供 `add_data`、`delete_data`、`search` 和 `dump`（持久化）操作。
- **PersistStore**：基于 LevelDB 的持久化键值存储，支持批量操作、范围扫描和序列化行存储。
- **VolatileStore**：用于测试和临时工作负载的内存键值存储。
- **BytesRow**：支持多种字段类型（int64、float32、string、binary、boolean、lists）的模式感知二进制序列化/反序列化层。

C++ 构建系统使用 CMake，依赖以下第三方库：

- **LevelDB 1.23**：持久化键值存储
- **spdlog 1.14.1**：高性能日志
- **RapidJSON**：JSON 解析
- **CRoaring**：Roaring 位图操作

构建支持 x86_64 架构的 SSE3/原生 SIMD 指令，目标为 C++17 标准，支持跨平台（Linux、macOS、Windows）。

4.3.4 后端支持

后端	描述
`local`	使用 C++ 引擎的本地持久化（默认）
`http`	HTTP 远程服务
`volcengine`	火山引擎 VikingDB 云服务

5 检索系统

OpenViking 的检索系统是其最具创新性的组件之一。它采用多阶段流水线：**意图分析** → **层级化检索** → **重排序**。

5.1 `find()` 与 `search()`

OpenViking 提供两个具有不同能力的检索 API：

特性	`find()`	`search()`
会话上下文	不需要	必需
意图分析	不使用	基于 LLM 的分析
查询数量	单一查询	0–5 个 TypedQuery
延迟	低	较高
使用场景	简单、直接的查询	复杂、多意图的任务

5.2 意图分析

`IntentAnalyzer` 使用 LLM 分析查询意图，生成 0–5 个 `TypedQuery` 对象。每个 TypedQuery 包含：

- **query**：重写优化后的搜索查询
- **context_type**：目标上下文类型（MEMORY、RESOURCE 或 SKILL）
- **intent**：查询目的
- **priority**：重要性排名（1–5）

查询风格根据上下文类型量身定制：
- **技能查询**：动词开头（“创建 RFC 文档”、“提取 PDF 表格”）
- **资源查询**：名词短语（“RFC 文档模板”、“API 使用指南”）
- **记忆查询**：所属格式（“用户的代码风格偏好”）

特殊情况：
- **0 个查询**：闲聊或问候，不需要检索
- **多个查询**：可能需要技能 + 资源 + 记忆的复杂任务

5.3 层级化检索算法

`HierarchicalRetriever` 使用优先队列递归搜索目录结构。这是 OpenViking 区别于扁平向量搜索的核心创新。

5.3.1 算法步骤

1. **确定根目录**：将 `context_type` 映射到根目录（例如 RESOURCE → `viking://resources/`）
2. **全局向量搜索**：定位 top-K 起始目录
3. **合并起始点**：使用重排序合并并评分候选目录

4. **递归搜索**：使用最小堆优先队列逐层深入目录树
5. **结果转换**：转换为 `MatchedContext` 对象

5.3.2 递归搜索伪代码

```
while dir_queue:
    current_uri, parent_score = heapq.heappop(dir_queue)

    # 搜索当前目录的子项
    results = await search(parent_uri=current_uri)

    for r in results:
        # 分数传播：结合嵌入分数与父级上下文
        final_score = 0.5 * embedding_score + 0.5 * parent_score

        if final_score > threshold:
            collected.append(r)

            if not r.is_leaf:  # 目录 → 继续递归
                heapq.heappush(dir_queue, (r.uri, final_score))

    # 收敛检测
    if topk_unchanged_for_3_rounds:
        break
```

5.3.3 关键参数

参数	值	描述
SCORE_PROPAGATION_ALPHA	0.5	50% 嵌入分数 + 50% 父级上下文分数
MAX_CONVERGENCE_ROUNDS	3	top-K 不变后 3 轮停止
GLOBAL_SEARCH_TOPK	3	全局搜索候选数量
MAX_RELATIONS	5	每个资源的最大关系数
DIRECTORY_DOMINANCE_RATIO	1.2	目录分数必须超过最高子项分数 20%

5.4 重排序

重排序使用专用的重排序模型（例如火山引擎的 `doubao-seed-rerank`）来优化候选结果。它在两个环节应用：

1. **起始点评估**：评估全局搜索候选目录
2. **递归搜索**：在递归的每一层评估子项

5.5 检索结果

```python
@dataclass
class FindResult:
    memories: List[MatchedContext]    # 匹配的记忆上下文
    resources: List[MatchedContext]   # 匹配的资源上下文
    skills: List[MatchedContext]      # 匹配的技能上下文
    query_plan: Optional[QueryPlan]  # 仅 search() 返回
    query_results: Optional[List[QueryResult]]
    total: int                        # 总匹配数
```

6 上下文提取与解析

OpenViking 使用三阶段异步架构进行文档解析和上下文提取。一个关键设计原则是**解析与语义分离**：解析器从不调用 LLM；语义生成完全异步。

6.1 解析流水线

```
输入文件 → 解析器 → TreeBuilder → SemanticQueue → 向量索引
                ↓            ↓               ↓
          解析与转换       移动文件        L0/L1 生成
          （不调用 LLM）   排队语义处理     （LLM 异步）
```

6.2 支持的格式

OpenViking 包含一个全面的解析器注册表，支持多种文档格式：

格式	解析器类	扩展名
Markdown	`MarkdownParser`	`.md`、`.markdown`
纯文本	`TextParser`	`.txt`
PDF	`PDFParser`	`.pdf`
HTML	`HTMLParser`	`.html`、`.htm`
Word	`WordParser`	`.docx`
PowerPoint	`PowerPointParser`	`.pptx`
Excel	`ExcelParser`	`.xlsx`
EPUB	`EpubParser`	`.epub`
代码	`CodeRepositoryParser`	`.py`、`.js`、`.go` 等
图片	`ImageParser`	`.png`、`.jpg` 等
视频	`VideoParser`	`.mp4`、`.avi` 等
音频	`AudioParser`	`.mp3`、`.wav` 等
ZIP	`ZipParser`	`.zip`
目录	`DirectoryParser`	文件系统目录

6.3 智能文档切分

文档根据 Token 数量进行智能切分：

```
如果 document_tokens <= 1024:
    → 保存为单个文件
否则:
    → 按标题切分
    → 段落 < 512 tokens → 与相邻段落合并
    → 段落 > 1024 tokens → 创建子目录
```

6.4 TreeBuilder

`TreeBuilder` 将解析后的内容从临时存储移动到 AGFS，并排队进行语义处理。它分五个阶段运行：

1. **查找文档根目录**：确保临时目录中恰好有一个子目录
2. **确定目标 URI**：按作用域映射基础 URI（`resources`、`user`、`agent`）
3. **递归移动目录树**：将所有文件复制到 AGFS
4. **清理临时目录**：删除临时文件
5. **排队语义生成**：提交 `SemanticMsg` 到队列

6.5 SemanticQueue：异步 L0/L1 生成

`SemanticProcessor` 处理异步 L0/L1 生成和向量化。处理流程自底向上：

1. **并发文件摘要生成**：最多 10 个并发 LLM 调用
2. **收集子目录摘要**：读取已生成的 `.abstract.md` 文件
3. **生成 `.overview.md`**：LLM 从子摘要生成 L1 概览
4. **提取 `.abstract.md`**：从概览中提取 L0
5. **写入文件**：保存到 AGFS
6. **向量化**：创建 `Context` 对象并排队到 `EmbeddingQueue`

7 会话管理

会话管理是 OpenViking 让智能体“从经验中学习”的机制。它管理对话消息、追踪上下文使用情况，并自动提取长期记忆。

7.1 会话生命周期

```
创建 → 交互（添加消息、追踪使用情况）→ 提交
```

```python
session = client.session(session_id="chat_001")
session.add_message("user", [TextPart("...")])
session.used(contexts=["viking://user/memories/profile.md"])
session.commit()  # 归档 + 记忆提取
```

7.2 消息结构

消息由类型化的部分组成：

部分类型	描述
TextPart	纯文本内容
ContextPart	带有 URI 和摘要的上下文引用
ToolPart	带有输入和输出的工具调用

7.3 压缩策略

当会话被提交时，自动进行归档：

1. 递增压缩索引
2. 将当前消息复制到归档目录（history/archive_NNN/）
3. 使用 LLM 生成结构化摘要
4. 清空当前消息列表

摘要遵循结构化格式，包括一行概述、关键分析、主要请求与意图、关键概念和待处理任务。

7.4 记忆提取

MemoryExtractor 是使智能体能够从交互中学习的核心组件。它通过多阶段流水线运行：

7.4.1 提取流水线

```
消息 → LLM 提取 → 候选记忆
                      ↓
向量预过滤 → 查找相似的现有记忆
```

```
                    ↓
LLM 去重决策 → CREATE / UPDATE / MERGE / SKIP
                    ↓
写入 AGFS → 向量化
```

7.4.2 去重决策

决策	描述
CREATE	新记忆；直接在相应类别目录中创建
UPDATE	用新信息更新现有记忆
MERGE	将多个相关记忆合并为一个综合条目
SKIP	与现有记忆重复；跳过

7.4.3 语言检测

记忆提取器包含自动语言检测功能，仅针对用户消息进行检测，以避免助手/系统文本影响存储记忆的目标输出语言。

8 模型集成

OpenViking 通过统一的提供商注册系统与多种 AI 模型提供商集成。

8.1 VLM（视觉语言模型）提供商

OpenViking 使用 VLM 模型进行内容理解、L0/L1 生成、意图分析和记忆提取。支持以下提供商：

提供商	模型系列	API Key 来源
`volcengine`	Doubao（豆包）	火山引擎控制台
`openai`	GPT	OpenAI 平台
`anthropic`	Claude	Anthropic 控制台
`deepseek`	DeepSeek	DeepSeek 平台
`gemini`	Gemini	Google AI Studio
`moonshot`	Kimi	Moonshot 平台
`zhipu`	GLM	智谱开放平台
`dashscope`	Qwen（通义千问）	DashScope 控制台
`minimax`	MiniMax	MiniMax 平台
`openrouter`	任意模型	OpenRouter
`vllm`	本地模型	自托管

系统使用**提供商注册表**进行统一的模型访问。提供商根据模型名称关键字自动检测，实现提供商之间的无缝切换。

8.1.1 VLM 后端实现

OpenViking 包含三种 VLM 后端实现：

- **OpenAI VLM**：使用 OpenAI 兼容的 Chat Completions API
- **Volcengine VLM**：使用火山引擎 ARK API，支持模型名称和端点 ID
- **LiteLLM VLM**：使用 LiteLLM 库作为任意 LLM 提供商的通用代理

8.1.2 Token 使用追踪

系统包含全面的 Token 使用追踪，记录每次 VLM 调用的输入 Token、输出 Token 和总 Token 数。

8.2 嵌入提供商

用于向量化和语义检索，OpenViking 支持：

提供商	描述
`volcengine`	火山引擎嵌入模型（例如 `doubao-embedding-vision-250615`）
`openai`	OpenAI 嵌入模型（例如 `text-embedding-3-large`）
`vikingdb`	VikingDB 原生嵌入

嵌入结果包括稠密向量，以及可选的用于混合检索的稀疏向量。

8.3 提示词管理

OpenViking 使用基于模板的提示词管理系统，采用存储为 YAML 文件的 Jinja2 模板。提示词按任务类别组织（例如记忆提取、意图分析、摘要生成），并支持参数化渲染。

9 HTTP 服务器与 API

OpenViking 提供基于 FastAPI 构建的全面 HTTP API，用于生产部署。

9.1 服务器架构

服务器实现在 `openviking/server/` 中，包含：

- **`bootstrap.py`**：服务器启动和配置
- **`app.py`**：带生命周期管理的 FastAPI 应用工厂
- **`auth.py`**：API Key 认证（X-API-Key 头部或 Bearer Token）
- **`config.py`**：服务器配置（端口、CORS、API Key）
- **`models.py`**：Pydantic 请求/响应模型
- **`dependencies.py`**：服务层的依赖注入

9.2 API 端点

9.2.1 系统端点

方法	路径	描述
GET	`/health`	健康检查（无需认证）
GET	`/api/v1/system/status`	系统组件状态
POST	`/api/v1/system/wait`	等待异步处理完成

9.2.2 资源端点

方法	路径	描述
POST	`/api/v1/resources`	添加资源（URL、文件或目录）
POST	`/api/v1/skills`	添加技能定义
POST	`/api/v1/pack/export`	导出上下文为 `.ovpack`
POST	`/api/v1/pack/import`	将 `.ovpack` 导入到目标 URI

9.2.3 文件系统端点

方法	路径	描述
GET	`/api/v1/fs/ls`	列出目录内容
GET	`/api/v1/fs/tree`	获取目录树
GET	`/api/v1/fs/stat`	获取资源元数据

POST	/api/v1/fs/mkdir	创建目录
DELETE	/api/v1/fs	删除资源
POST	/api/v1/fs/mv	移动/重命名资源

9.2.4 内容端点

方法	路径	描述
GET	/api/v1/content/read	读取完整内容（L2）
GET	/api/v1/content/abstract	读取摘要（L0）
GET	/api/v1/content/overview	读取概览（L1）

9.2.5 搜索端点

方法	路径	描述
POST	/api/v1/search/find	简单语义搜索
POST	/api/v1/search/search	带意图分析的上下文感知搜索
POST	/api/v1/search/grep	内容模式搜索
POST	/api/v1/search/glob	文件模式匹配

9.2.6 会话端点

方法	路径	描述
POST	/api/v1/sessions	创建新会话
GET	/api/v1/sessions	列出所有会话
GET	/api/v1/sessions/{id}	获取会话详情
DELETE	/api/v1/sessions/{id}	删除会话
POST	/api/v1/sessions/{id}/commit	提交会话（归档 + 提取）
POST	/api/v1/sessions/{id}/messages	向会话添加消息

9.2.7 关系与观察者端点

方法	路径	描述
GET	/api/v1/relations	获取资源关系

POST	/api/v1/relations/link	创建关系链接
DELETE	/api/v1/relations/link	移除关系链接
GET	/api/v1/observer/queue	队列处理状态
GET	/api/v1/observer/vikingdb	VikingDB 状态
GET	/api/v1/observer/vlm	VLM 状态
GET	/api/v1/observer/system	整体系统状态

9.3 响应格式

所有响应遵循统一的 JSON 格式：

```
// 成功
{
  "status": "ok",
  "result": { ... },
  "time": 0.123
}

// 错误
{
  "status": "error",
  "error": {
    "code": "NOT_FOUND",
    "message": "Resource not found: viking://resources/nonexistent/"
  },
  "time": 0.01
}
```

9.4 错误码

OpenViking 定义了一套完整的错误码，映射到 HTTP 状态码：

错误码	HTTP	描述
OK	200	成功
INVALID_ARGUMENT	400	无效参数
INVALID_URI	400	无效的 Viking URI 格式
NOT_FOUND	404	资源未找到
ALREADY_EXISTS	409	资源已存在
UNAUTHENTICATED	401	缺少或无效的 API Key
PERMISSION_DENIED	403	权限不足

RESOURCE_EXHAUSTED	429	速率限制超出
DEADLINE_EXCEEDED	504	操作超时
UNAVAILABLE	503	服务不可用
INTERNAL	500	内部服务器错误
EMBEDDING_FAILED	500	嵌入生成失败
VLM_FAILED	500	VLM 调用失败
SESSION_EXPIRED	410	会话已过期

10 Rust CLI

OpenViking 包含一个功能完整的命令行界面，使用 Rust 编写，提供快速的编译二进制文件用于与 OpenViking 服务器交互。

10.1 架构

Rust CLI 作为 Cargo workspace 成员组织在 `crates/ov_cli/` 中。它使用：

- **clap**：基于派生宏的命令行参数解析
- **reqwest**：用于服务器通信的 HTTP 客户端
- **tokio**：异步运行时
- **serde/serde_json**：JSON 序列化
- **tabled**：终端表格格式化

CLI 遵循清晰的架构：`main.rs` → `commands/` 模块 → `client.rs`（HTTP 客户端）→ 服务器。

10.2 可用命令

CLI 镜像了完整的 HTTP API 接口：

10.2.1 资源管理

- `add-resource <path>` — 导入资源（URL、文件或目录）
- `add-skill <data>` — 导入技能定义

10.2.2 文件系统操作

- `ls <uri>` — 列出目录内容（别名：`list`）
- `tree <uri>` — 显示目录树
- `mkdir <uri>` — 创建目录
- `rm <uri>` — 删除资源（别名：`del`、`delete`）
- `mv <from> <to>` — 移动/重命名（别名：`rename`）
- `stat <uri>` — 获取资源元数据

10.2.3 内容访问

- `read <uri>` — 读取完整内容（L2）
- `abstract <uri>` — 读取摘要（L0）
- `overview <uri>` — 读取概览（L1）

10.2.4 搜索操作

- `find <query>` — 简单语义搜索
- `search <query>` — 带意图分析的上下文感知搜索
- `grep <uri> <pattern>` — 内容模式搜索
- `glob <pattern>` — 文件模式匹配

10.2.5 会话管理

- `session new` — 创建新会话
- `session list` — 列出所有会话
- `session get <id>` — 获取会话详情
- `session delete <id>` — 删除会话
- `session add-message <id>` — 向会话添加消息
- `session commit <id>` — 提交会话
- `add-memory <content>` — 一次性记忆添加

10.2.6 关系管理

- `relations <uri>` — 列出关系
- `link <from> <to...>` — 创建关系链接
- `unlink <from> <to>` — 移除关系链接

10.2.7 导入/导出

- `export <uri> <to>` — 导出上下文为 `.ovpack`
- `import <file> <uri>` — 导入 `.ovpack`

10.2.8 系统与监控

- `status` — 显示组件状态
- `health` — 快速健康检查
- `wait` — 等待异步处理完成
- `observer queue|vikingdb|vlm|system` — 观察者状态
- `config show|validate` — 配置管理

10.3 输出格式

CLI 支持三种输出格式：

- **表格**（默认）：人类可读的表格格式
- **JSON**（`-o json`）：与 API 响应格式匹配的结构化 JSON
- **紧凑**（`-c true`，默认）：为智能体消费简化的输出

10.4 构建与安装

```
# 通过脚本安装
curl -fsSL https://raw.githubusercontent.com/volcengine/OpenViking/\
    main/crates/ov_cli/install.sh | bash

# 或从源码构建
cargo install --git https://github.com/volcengine/OpenViking ov_cli
```

发布构建使用 `opt-level = 3`、LTO（链接时优化）和符号剥离，以获得最大性能和最小二进制体积。

11 Python SDK 与 CLI

11.1 Python SDK

Python SDK 提供同步和异步客户端：

11.1.1 客户端类

类	描述
`SyncOpenViking`	嵌入模式的同步客户端（别名为 `OpenViking`）
`AsyncOpenViking`	嵌入模式的异步客户端（单例模式）
`SyncHTTPClient`	HTTP 模式的同步客户端
`AsyncHTTPClient`	HTTP 模式的异步客户端

所有客户端实现相同的 `BaseClient` 接口，确保跨部署模式的一致行为。

11.1.2 快速入门示例

```python
import openviking as ov

client = ov.OpenViking(path="./data")
client.initialize()

# 添加资源
res = client.add_resource(path="https://example.com/doc.md")
root_uri = res["root_uri"]

# 等待语义处理
client.wait_processed()

# 获取分层内容
abstract = client.abstract(root_uri)    # L0
overview = client.overview(root_uri)    # L1
content = client.read(root_uri)         # L2

# 语义搜索
results = client.find("authentication", target_uri=root_uri)
for r in results.resources:
    print(f"  {r.uri} (score: {r.score:.4f})")

client.close()
```

11.2 Python CLI

Python CLI 使用 **Typer** 实现，提供 `openviking` 命令。它在 `pyproject.toml` 中定义为入口点：

```
[project.scripts]
openviking = "openviking_cli.cli.main:app"
openviking-server = "openviking.server.bootstrap:main"
```

Python CLI 包（`openviking_cli/`）包含：

- `cli/` — CLI 命令定义
- `client/` — HTTP 和本地客户端实现
- `session/` — 会话管理和用户标识
- `utils/` — 配置、日志、URI 解析

12 项目结构与代码库

12.1 目录布局

```
OpenViking/
├── openviking/              # 核心 Python SDK
│   ├── __init__.py          # 公共 API 导出
│   ├── async_client.py      # AsyncOpenViking 客户端
│   ├── sync_client.py       # SyncOpenViking 客户端
│   ├── agfs_manager.py      # AGFS 子进程管理
│   ├── core/                # 核心数据模型
│   │   ├── context.py       # Context 基类
│   │   ├── directories.py   # 目录定义
│   │   ├── building_tree.py # 树构建逻辑
│   │   ├── skill_loader.py  # 技能加载
│   │   └── mcp_converter.py # MCP 工具转换
│   ├── client/              # 客户端实现
│   │   ├── local.py         # LocalClient（嵌入式）
│   │   └── session.py       # 会话集成
│   ├── models/              # AI 模型集成
│   │   ├── vlm/             # VLM 提供商
│   │   │   ├── registry.py  # 提供商注册表
│   │   │   ├── backends/    # OpenAI、Volcengine、LiteLLM
│   │   │   └── token_usage.py
│   │   └── embedder/        # 嵌入提供商
│   │       ├── base.py      # 基础嵌入器
│   │       ├── openai_embedders.py
│   │       └── volcengine_embedders.py
│   ├── parse/               # 文档解析器
│   │   ├── registry.py      # 解析器注册表
│   │   ├── tree_builder.py  # TreeBuilder
│   │   ├── parsers/         # 特定格式解析器
│   │   │   ├── markdown.py, pdf.py, html.py
│   │   │   ├── word.py, powerpoint.py, excel.py
│   │   │   ├── epub.py, text.py, media.py
│   │   │   ├── code/, directory.py, zip_parser.py
│   │   │   └── constants.py, upload_utils.py
│   │   ├── resource_detector/ # 资源检测
│   │   ├── converter.py     # 格式转换
│   │   └── vlm.py           # 基于 VLM 的解析
│   ├── retrieve/            # 检索系统
│   │   ├── hierarchical_retriever.py
│   │   └── intent_analyzer.py
│   ├── session/             # 会话管理
│   │   ├── session.py       # 会话核心
│   │   ├── compressor.py    # 会话压缩
│   │   ├── memory_extractor.py  # 记忆提取
│   │   └── memory_deduplicator.py
│   ├── service/             # 服务层
│   │   ├── core.py          # OpenVikingService
│   │   ├── fs_service.py, search_service.py
```

```
│   │   ├── session_service.py, resource_service.py
│   │   ├── relation_service.py, pack_service.py
│   │   └── debug_service.py
│   ├── server/               # HTTP 服务器
│   │   ├── app.py            # FastAPI 应用
│   │   ├── bootstrap.py      # 服务器启动
│   │   ├── auth.py, config.py, models.py
│   │   └── routers/          # API 路由处理器
│   ├── storage/              # 存储层
│   │   ├── viking_fs.py      # VikingFS 抽象层
│   │   ├── vectordb/         # 向量索引
│   │   └── queuefs/          # 语义处理队列
│   ├── message/              # 消息处理
│   ├── prompts/              # 提示词模板
│   └── utils/                # 配置、日志
├── openviking_cli/           # CLI 包
│   ├── cli/                  # CLI 命令
│   ├── client/               # HTTP/同步客户端
│   ├── session/              # 会话/用户 ID
│   └── utils/                # 配置、URI、日志
├── crates/                   # Rust crate
│   └── ov_cli/               # Rust CLI 二进制
│       └── src/              # Rust 源文件
├── src/                      # C++ 扩展
│   ├── CMakeLists.txt        # 构建配置
│   ├── pybind11_interface.cpp
│   ├── index/                # 向量索引引擎
│   ├── store/                # 持久化/易失性存储
│   └── common/               # 共享工具
├── third_party/              # 第三方依赖
│   ├── agfs/                 # AGFS 文件系统（Go）
│   ├── leveldb-1.23/         # LevelDB
│   ├── spdlog-1.14.1/        # 日志库
│   ├── croaring/             # Roaring 位图
│   └── rapidjson/            # JSON 解析
├── tests/                    # 测试套件
│   ├── client/               # 客户端测试
│   ├── engine/               # 引擎测试
│   ├── integration/          # 集成测试
│   ├── session/              # 会话测试
│   └── vectordb/             # 向量数据库测试
├── examples/                 # 使用示例
│   ├── quick_start.py        # 快速入门示例
│   ├── query/                # 查询示例
│   ├── chatmem/              # 聊天记忆示例
│   ├── memex/                # 记忆提取示例
│   ├── mcp-query/            # MCP 集成示例
│   ├── openclaw-skill/       # 技能集成示例
│   ├── server_client/        # 服务器/客户端示例
│   └── common/               # 共享示例代码
├── docs/                     # 文档
│   ├── en/                   # 英文文档
│   └── zh/                   # 中文文档
└── .github/workflows/        # CI/CD 工作流
```

12.2 语言组成

代码库是一个跨越四种编程语言的多语言项目：

语言	角色	关键组件
Python	核心 SDK、服务器、解析器、服务	openviking/、openviking_cli/、tests/
Rust	高性能 CLI	crates/ov_cli/
C++17	向量索引引擎、存储引擎	src/index/、src/store/
Go	AGFS 文件系统服务器	third_party/agfs/

13 构建系统与依赖

13.1 Python 构建

Python 包使用 `setuptools` 和 `setuptools-scm` 通过 Git 标签进行版本管理。构建过程包括：

1. **AGFS 编译**：从源码构建基于 Go 的 AGFS 服务器
2. **C++ 扩展编译**：使用 CMake 将向量索引引擎构建为 pybind11 模块
3. **包组装**：在 wheel 中包含 AGFS 二进制文件和编译后的 C++ 扩展

主要依赖包括：

- **pydantic**（>= 2.0）：数据验证和序列化
- **httpx**（>= 0.25）：HTTP 客户端
- **fastapi**（>= 0.128）+ **uvicorn**：HTTP 服务器
- **openai**（>= 1.0）：OpenAI API 客户端
- **litellm**（>= 1.0）：通用 LLM 代理
- **pdfplumber**、**python-docx**、**python-pptx**、**openpyxl**、**ebooklib**：文档解析
- **markdownify**、**readabilipy**：HTML 转 Markdown 转换
- **pyagfs**：AGFS 客户端库
- **jinja2**：提示词模板渲染
- **typer**：Python CLI 框架
- **xxhash**：快速哈希

13.2 Rust 构建

Rust CLI 使用 Cargo workspace 并配置发布优化：

```
[profile.release]
opt-level = 3
lto = true
strip = true
```

主要 Rust 依赖：

- **clap**：CLI 参数解析
- **reqwest**：HTTP 客户端
- **tokio**：异步运行时
- **serde/serde_json**：序列化
- **tabled**：表格格式化

13.3 C++ 构建

C++ 扩展使用 CMake 构建：

- C++17 标准
- pybind11 用于 Python 绑定
- LevelDB 用于持久化存储
- SIMD 优化（SSE3 / 原生指令集）

- **跨平台支持**（Linux、macOS、Windows）

14 CI/CD 与质量保障

14.1 GitHub Actions 工作流

OpenViking 使用全面、模块化的 GitHub Actions CI/CD 流水线：

14.1.1 自动工作流

触发条件	工作流	描述
Pull Request	`pr.yml`	Lint（Ruff、Mypy）+ 精简测试（Linux、Python 3.10）
推送到 Main	`ci.yml`	完整测试 (Linux/Win/Mac、Python 3.10–3.13) + CodeQL
发布版本	`release.yml`	构建 sdist + wheels，发布到 PyPI
每周定时	`schedule.yml`	每周日 CodeQL 安全扫描

14.1.2 手动触发工作流

名称	ID	描述
Lint 检查	`_lint.yml`	Ruff + Mypy 检查
测试套件（精简）	`_test_lite.yml`	快速集成测试，可配置矩阵
测试套件（完整）	`_test_full.yml`	所有平台和 Python 版本的完整测试
安全扫描	`_codeql.yml`	CodeQL 安全分析
构建分发包	`_build.yml`	仅构建 wheels（不发布）
发布分发包	`_publish.yml`	将构建的包发布到 PyPI
Rust CLI	`rust-cli.yml`	构建和测试 Rust CLI

14.2 代码质量工具

工具	用途	配置
Ruff	代码检查 + 格式化 + 导入排序	`pyproject.toml` — 行宽 100，双引号
Mypy	静态类型检查	`pyproject.toml` — Python 3.10，检查未类型化定义
pre-commit	自动化检查的 Git 钩子	`.pre-commit-config.yaml`
pytest	测试框架	`pyproject.toml` — 异步模式，覆盖率报告

CodeQL	安全漏洞扫描	GitHub Actions 工作流

14.3 测试策略

测试按模块组织：

- `tests/client/` — 客户端集成测试
- `tests/engine/` — C++ 引擎测试
- `tests/integration/` — 端到端集成测试
- `tests/session/` — 会话管理测试
- `tests/vectordb/` — 向量数据库测试

测试配置：asyncio 自动模式、详细输出，以及覆盖率报告 `--cov=openviking --cov-report=term-missing`。

14.4 版本管理

版本通过 `setuptools-scm` 从 Git 标签派生：

- **标签格式**：`vX.Y.Z`（语义化版本）
- **发布构建**：标签 → 版本（例如 `v0.1.0` → `0.1.0`）
- **开发构建**：包含提交计数（例如 `0.1.1.dev3`）

15 配置系统

OpenViking 使用双文件配置系统：

15.1 ov.conf — 核心配置

位于 ~/.openviking/ov.conf（或通过 OPENVIKING_CONFIG_FILE 指定），此 JSON 文件配置嵌入模型、VLM 模型、存储后端和其他核心设置。

```
{
  "embedding": {
    "dense": {
      "api_base": "https://ark.cn-beijing.volces.com/api/v3",
      "api_key": "your-api-key",
      "provider": "volcengine",
      "dimension": 1024,
      "model": "doubao-embedding-vision-250615"
    }
  },
  "vlm": {
    "api_base": "https://ark.cn-beijing.volces.com/api/v3",
    "api_key": "your-api-key",
    "provider": "volcengine",
    "model": "doubao-seed-1-8-251228"
  }
}
```

15.2 ovcli.conf — CLI/HTTP 客户端配置

位于 ~/.openviking/ovcli.conf（或通过 OPENVIKING_CLI_CONFIG_FILE 指定），此文件配置到 OpenViking 服务器的连接：

```
{
  "url": "http://localhost:1933",
  "api_key": "your-key",
  "output": "table"
}
```

16 导入/导出：OVPack 格式

OpenViking 支持将上下文树导出和导入为 `.ovpack` 文件。这使得：

- **上下文可移植性**：在 OpenViking 实例之间共享上下文树
- **备份与恢复**：创建特定上下文子树的备份
- **协作**：与团队成员共享预构建的上下文包

```
# 导出
client.export_ovpack(uri="viking://resources/my-project/", to="project.ovpack")

# 导入
client.import_ovpack(file_path="project.ovpack", target_uri="viking://
resources/")
```

17 关系系统

OpenViking 支持资源之间的显式关系链接，使智能体能够发现相关上下文：

```
# 创建关系
client.link(
    from_uri="viking://resources/docs/auth",
    uris=["viking://resources/docs/security"],
    reason="Related security documentation"
)

# 获取关系
relations = client.relations("viking://resources/docs/auth")
```

关系存储为每个目录内的 `.relations.json` 文件，在检索过程中作为附加到搜索结果的 `RelatedContext` 对象呈现。

18 MCP 集成

OpenViking 包含一个 **Model Context Protocol（MCP）**转换器（`mcp_converter.py`），可以自动将 MCP 工具定义转换为 OpenViking 技能格式。这使得使用 MCP 兼容框架构建的智能体能够无缝地将其工具注册为 OpenViking 中可搜索的技能。

19 示例与用例

代码仓库包含多个全面的示例：

示例	描述
quick_start.py	最小示例：添加资源、搜索、检索
query/	查询和检索示例
chatmem/	聊天记忆管理示例
memex/	从对话中提取记忆
mcp-query/	MCP 工具与 OpenViking 集成
openclaw-skill/	技能定义和注册
server_client/	服务器部署和客户端连接
common/	示例共享工具

20 未来路线图

项目已记录了清晰的未来发展路线图：

20.1 已完成功能

- 三层信息模型（L0/L1/L2）
- Viking URI 寻址系统
- 双层存储（AGFS + 向量索引）
- 异步/同步客户端支持
- 文本资源管理（Markdown、HTML、PDF、DOCX、PPTX、XLSX、EPUB）
- 自动 L0/L1 生成
- 基于向量索引的语义搜索
- 资源关系和链接
- 带意图分析的上下文感知搜索
- 带记忆提取的会话管理
- 基于 LLM 的记忆去重
- 技能定义、存储和 MCP 自动转换
- HTTP 服务器（FastAPI），支持 API Key 认证
- Python 和 Rust CLI
- 可插拔的嵌入和 LLM 提供商

20.2 计划功能

- **CLI 增强**：为所有操作提供完整的命令行界面；分布式存储后端
- **多模态支持**：图片、视频和音频资源的智能解析和访问
- **上下文管理**：上下文修改时的传播更新；类 git 的版本管理和回滚
- **访问控制**：多智能体和多用户支持；基于角色的隔离；资源目录节点的权限设计
- **生态集成**：流行的智能体框架适配器；自定义组件的插件系统

21 设计原则

OpenViking 遵循以下关键设计原则：

原则	描述
纯存储层	存储层仅处理 AGFS 操作和基础向量搜索；Rerank 逻辑在检索层
三层信息	L0/L1/L2 实现渐进式详情加载，节省 Token 消耗和成本
两阶段检索	向量搜索召回候选，Rerank 提升精度
单一数据源	所有内容从 AGFS 读取；向量索引仅存储引用，不存储内容
解析与语义分离	解析器从不调用 LLM；语义生成完全异步
文件系统范式	所有上下文组织为具有确定路径的虚拟文件系统

22 结论

OpenViking 代表了 AI 智能体基础设施领域的重大进步。通过从文件系统范式的视角重新构想上下文管理，它解决了传统 RAG 方法的根本局限性。

项目的关键技术贡献包括：

1. **统一的上下文模型**，将记忆、资源和技能作为虚拟文件系统中的一等公民，消除了困扰传统方法的碎片化问题。
2. **三层信息模型（L0/L1/L2）**，实现上下文的渐进式加载，在保持信息完整性的同时大幅降低 Token 消耗。
3. **层级化检索算法**，将全局向量搜索与递归目录探索相结合，在上下文发现中兼顾广度和深度。
4. **自动记忆提取系统**，使智能体能够从交互中学习，形成知识积累的自我改进循环。
5. **多语言架构**，横跨 Python、Rust、C++ 和 Go，每种语言都因其优势而被选用：Python 提供灵活性和生态系统，Rust 提供 CLI 性能，C++ 提供向量引擎速度，Go 提供文件系统操作。

项目由字节跳动火山引擎 Viking 团队积极维护，有清晰的未来发展路线图，包括多模态支持、版本管理、访问控制和生态集成。随着 AI 智能体从简单的聊天机器人持续演进为复杂的自主系统，像 OpenViking 这样的工具将成为管理驱动其智能的上下文的关键基础设施。

报告结束

生成于 2026 年 2 月 17 日

来源：`github.com/volcengine/OpenViking`