目录结构

概述（Overview）

01-overview.md

- **1.1 引言**
- **1.2 内容简介**
- **1.3 目标读者**

数据类型与指标

2. 可观测性数据类型总览（Telemetry Overview）

02-telemetry-overview.md

- **2.1 指标（Metrics）**
- **2.2 日志（Logs）**
- **2.3 追踪（Traces）**
- **2.4 事件（Events）**
- **2.5 性能剖析（Profiles）**
- **2.6 网络元组（Network Tuples）**
- **2.7 SLA / SLO / SLI**

---

eBPF 技术与应用

03-bpf-overview.md

- **3.1 概述**
- **3.2 对比**
- **3.3 场景**

应用系统故障排查

4.1 故障排查整体流程

4.1.1 从客户端到服务端的请求路径

- DNS 解析

- TCP 连接建立
- TLS 握手（如适用）
- 请求发送与响应接收

4.1.2 分层排查思路

- 应用层
- 服务层
- 网络层
- 物理层

4.2 应用请求关联指标的排查

4.2.1 响应时间

- 慢请求分析
- 依赖服务性能

4.2.2 错误率

- 服务端异常 vs 客户端异常
- HTTP 状态码分析

4.2.3 系统资源使用率

- CPU 和内存监控
- 资源泄漏检测

4.2.4 用户留存率

- 用户行为分析
- 功能可用性检查

4.2.5 吞吐量

- 负载测试
- 扩展能力评估

4.3 网络关键指标的排查

4.3.1 带宽利用率

- 流量监控
- 异常流量检测

4.3.2 延迟

- 网络路径优化
- 地理位置影响

4.3.3 丢包率

- 链路质量
- 设备故障排查

4.3.4 网络抖动

- 实时应用影响
- 网络稳定性提升

4.3.5 网络可用性

- 冗余设计
- 故障切换机制

4.3.6 重传率

- 拥塞控制
- 传输优化

4.3.7 TCP 连接时间

- 握手延迟分析
- 服务器性能

4.3.8 DNS 查询时间

- DNS 服务优化
- 缓存策略

4.3.9 HTTP 请求成功率

- 服务器健康检查
- 网络异常处理

4.4 日志分析与故障定位

4.4.1 网络日志分析

- 流日志
- 数据包日志

4.4.2 服务日志分析

- 调用链追踪
- 接口性能监控

4.4.3 应用日志分析

- 错误堆栈
- 业务逻辑验证

4.5 典型故障场景与案例分析

4.5.1 服务端异常

- 资源耗尽
- 依赖服务故障

4.5.2 客户端异常

- 请求格式错误
- 网络连接失败

4.5.3 综合案例

- 从问题描述到根因定位

附录

5.1 术语表

5.2 参考资料

5.3 工具与资源

5.4 生产级监控配置方案（Linux 裸机）

05-production-monitoring-linux-bare-metal.md

- 系统与进程指标采集、历史保底以及安全加固实践 ### **5.5 Vector 统一采集架构与 DeepFlow 对比 #### 05-vector-unified-collector-architecture.md**
- Vector 汇聚多类 exporter，比较 DeepFlow 与 Vector 的功能与部署策略，并给出后端存储选型建议

5.6 PostgreSQL + ClickHouse 分层写入架构

05-postgres-clickhouse-layered-architecture.md

- 使用 TimescaleDB + ClickHouse 构建热写冷存的可观测性存储，实现高并发写入与 OLAP 聚合

架构设计与容灾

6.1 技术选型对比（Technology Selection Comparison）

06-technical-selection-comparison.md

- 边缘采集对比
- 网关对比
- 存储对比
- 分析层对比

6.2 总体架构设计

07-overall-architecture-design.md

- 边缘采集与缓冲
- 区域网关与扇出
- 存储与分析层
- 双链路（近线检索 + Kafka 回放）

6.3 多区域架构与区域内拓扑

08-multi-region-architecture-topology.md

- 区域内拓扑（单区示例）
- 多区域部署模式（主区 + 从区）
- 联邦查询与统一入口

6.4 端到端可靠传输设计（At-least-once 语义）

09-end-to-end-reliable-transmission-design.md

- 多级持久化链（Vector → OTel → Kafka → OpenObserve）
- 扇出与去重策略（event_id + UPSERT）

- SRE 可观测与演练方法

6.5 PostgreSQL 二级分析域

10-postgresql-secondary-analysis-domain.md

- 时序分析（TimescaleDB 连续聚合）
- 向量检索（pgvector）
- 图查询（Apache AGE）
- 高基数与 TopK 分析（HLL/Toolkit）

6.6 多区域与容灾（Disaster Recovery, DR）

11-multi-region-disaster-recovery.md

- 接入与路由（Anycast/GeoDNS）
- 跨区复制与镜像（Kafka MirrorMaker2）
- 阈值控制与降级策略
- 历史回放与灰度演练

监控简史（Monitoring History）

7.1 漫谈监控简史到全栈可观测数据选型

12-1-monitoring-history-to-full-stack-observable-data-selection.md

7.2 典型运维工作场景

12-2-typical-operations-scenarios.md

可观测系统设计篇（Observability System Design）

8.1 边缘采集与网关的抉择

13-1-edge-collection-vs-gateway.md

8.2 OTel Gateway 的设计与思考

13-2-otel-gateway-design-considerations.md

8.3 数据采集多级持久化与可重放

13-3-data-ingestion-multi-level-persistence-and-replay.md

8.4 全栈可观测数据库设计

13-4-full-stack-observability-database-design.md

8.5 全栈可观测数据库 ETL 设计

13-5-full-stack-observability-database-etl-design.md

8.6 监控系统的多区域与容灾

13-6-monitoring-system-multi-region-and-dr.md

LLM OPS Agent 设计篇（LLM OPS Agent Design）

9.1 从 SSH/SCP 到 AI 驱动的 OPS Agent：落地前的思考

14-1-from-ssh-scp-to-ai-ops-agent-pre-deployment-considerations.md

9.2 从 SSH/SCP 到 AI 驱动的 OPS Agent：能力清单

14-2-from-ssh-scp-to-ai-ops-agent-capability-checklist.md

9.3 PostgreSQL 扩展驱动的复杂分析（向量 / 图 / 趋势）

14-3-postgresql-extension-driven-complex-analysis-vector-graph-trend.md

9.4 AI-OPS Agent MVP 架构方案

14-4-ai-ops-agent-mvp-architecture.md

Observability: An Overview

“**Observability**”（可观测性）是系统工程中的一个核心概念，尤其在分布式系统、云原生架构和微服务环境中至关重要。Observability 指的是通过系统外部可见的数据（如指标、日志、追踪信息等），推断系统内部状态并理解其运行行为的能力。它能够帮助工程师快速定位问题，优化性能，确保系统的可靠性和稳定性。

核心组成：从三大支柱到五大关键数据类型

传统上，Observability 的三大支柱是 **Metrics**（指标）、**Logs**（日志）和 **Traces**（分布式追踪）。在现代 Observability 中，这一体系已扩展为五大数据类型，涵盖 **Events**（事件）和 **Profiles**（性能剖析）。

1. Metrics（指标）

- **定义**：定量的时间序列数据，用于描述系统的性能和资源利用率。
- **用途**：趋势分析、容量规划、实时健康监控。
- **典型内容**：
  - CPU 使用率、内存占用、磁盘 I/O、网络流量。
  - 应用层指标（如请求数、响应时间、错误率）。
- **工具**：
  - Prometheus、Grafana、Datadog、CloudWatch。

2. Logs（日志）

- **定义**：系统运行时的记录信息，通常以文本形式存储。
- **用途**：问题排查和故障根因分析。
- **典型内容**：
  - 错误日志、访问日志、调试信息。
- **工具**：
  - Elasticsearch、Fluentd、Splunk。

3. Traces（分布式追踪）

- **定义**：记录请求在分布式系统中跨多个服务或组件的传播路径。
- **用途**：分析请求性能瓶颈、延迟和依赖关系。
- **典型内容**：
  - 请求 ID、服务调用耗时、失败服务节点。
- **工具**：
  - OpenTelemetry、Jaeger、Zipkin。

4. Events（事件）

- **定义**：系统中发生的重要状态变化或操作记录。
- **用途**：补充 Metrics 和 Logs，为关键操作提供上下文。
- **典型事件**：
  - Kubernetes Pod 生命周期变化（启动、销毁）。
  - 配置变更、用户行为（如点击、登录）。
- **工具**：
  - Kubernetes Events、EventBridge。

5. Profiles（性能剖析）

- **定义**：动态收集程序运行时的性能数据，揭示代码和资源的使用情况。
- **用途**：优化性能、识别瓶颈。
- **典型内容**：
  - CPU/内存使用热点图。
  - 函数调用栈及耗时分布。

- **工具**：
  - Pyroscope、Parca、Flame Graph。

CBPF/eBPF 与零代码开源 Agent 技术

现代 Observability 的实现得益于 **CBPF/eBPF** 技术，它们支持高性能、零代码的数据采集，并已催生出多种开源 Agent 工具。

CBPF 与 eBPF 技术概述

- **CBPF**：
  - **经典 BPF**，主要用于简单的网络数据包过滤操作，如 `tcpdump`。
  - 功能有限，仅支持基础网络流量分析。
- **eBPF**：
  - **扩展版 BPF**，支持复杂逻辑，可动态挂载到内核钩子点，执行安全沙箱程序。
  - 用于网络监控、性能分析和安全观测。
  - 优势：高性能、灵活性强、无需修改应用代码。

零代码开源 Agent

利用 eBPF 的强大能力，社区开发了多种开源工具，能够实现零代码的深度数据采集。

1. DeepFlow-Agent

- **特点**：
  - 利用 eBPF 和 XDP 技术，从内核采集网络流量、应用性能数据和容器信息。
  - 自动化关联 Kubernetes 元数据，生成服务依赖关系和网络拓扑。
  - 高性能、无侵入式，适合云原生环境。
- **功能**：
  - 数据流日志（Flow Logs）。
  - 应用性能指标（APM 数据）。
  - 容器、Pod、虚拟机的元数据采集。
- **应用场景**：
  - 网络观测、分布式应用监控、安全分析。
- **官网**：https://deepflow.yunshan.net/

2. Pixie

- **特点**：
  - 基于 eBPF 的全自动化 Kubernetes Observability 工具。
  - 无需代码修改，实时捕获请求延迟、应用状态和错误信息。
- **功能**：
  - 分布式追踪和服务依赖图。

  - 实时应用性能剖析。
- **应用场景**：
  - Kubernetes 环境的快速问题排查和性能优化。
- **官网**：https://pixielabs.ai/

3. Cilium

- **特点**：
  - 使用 eBPF 提供高效的网络和安全 Observability 功能。
  - 与 Kubernetes 深度集成，支持服务网格。
- **功能**：
  - 网络流量监控。
  - API 请求分析和服务依赖追踪。
- **应用场景**：
  - 云原生网络安全和 Observability。
- **官网**：https://cilium.io/

4. Parca

- **特点**：
  - 专注于性能剖析，基于 eBPF 的开源工具。
  - 无侵入式采集运行时的 CPU 和内存使用数据。
- **功能**：
  - 生成火焰图分析性能瓶颈。
- **应用场景**：
  - 微服务和云原生环境的性能优化。
- **官网**：https://parca.dev/

5. Sysdig

- **特点**：
  - eBPF 驱动的安全和监控工具。
  - 采集系统调用和网络流量，用于安全分析和性能监控。
- **功能**：
  - 深入的系统调用审计。
  - 容器和 Kubernetes 的运行时安全监控。
- **应用场景**：
  - 安全观测与性能监控结合。
- **官网**：https://sysdig.com/

关键术语

术语	定义
Monitoring	实时监控系统的健康状况和性能，用于快速检测异常。
Tracing	追踪请求在分布式系统中的传播路径，用于依赖关系和性能分析。
Logging	捕获系统运行时的信息，记录详细的事件历史。
Instrumentation	在代码中嵌入可观测性相关逻辑以生成数据，便于调试和监控。
CBPF	经典 BPF，主要用于简单的网络数据包过滤操作。
eBPF	扩展 BPF，可挂载到内核多个钩子点，支持复杂的监控和分析任务。
XDP	高性能网络数据包处理框架，运行在网络驱动层。

总结

Observability 是现代分布式系统中不可或缺的能力。通过 Metrics、Logs、Traces、Events 和 Profiles 的结合，以及 CBPF/eBPF、XDP 等强大的内核技术支持，我们能够全面了解系统的行为和状态，并快速诊断问题。结合 DeepFlow-Agent、Pixie、Cilium 等开源工具，可以轻松实现零代码、低开销的高效数据采集，保障系统的稳定性和性能。

概述

- **APM（Application Performance Monitoring）**：聚焦应用层性能与稳定性（调用链、错误率、数据库/外部依赖耗时、CPU/内存剖析等）。
- **NPM（Network Performance Monitoring）**：聚焦网络与流量（L3–L7 时延、丢包、吞吐、拓扑依赖、抓包/取证）。
- **eBPF 的地位**：作为**内核侧“零侵入”采集基座**，贯通 NPM 与 APM 数据面；K8s 场景中与 **OpenTelemetry（OTel）**形成“**eBPF + OTel**”双轨标准。

APM / NPM 一览

象限定位（X=APM 强度，Y=NPM 强度）

- 右上：**Datadog、Dynatrace**；**DeepFlow** 更靠上（网络更强）；“**Grafana Stack + Cilium/Hubble**”组合接近右上
- 右下：**New Relic、SkyWalking、SigNoz**
- 左上：**Colasoft、Cilium+Hubble、Pixie**（**DeepFlow** 亦可视为偏此）
- 中间偏右：**DataBuff**
- 标准件：**OTel**（采集层，属于标准，不列入）

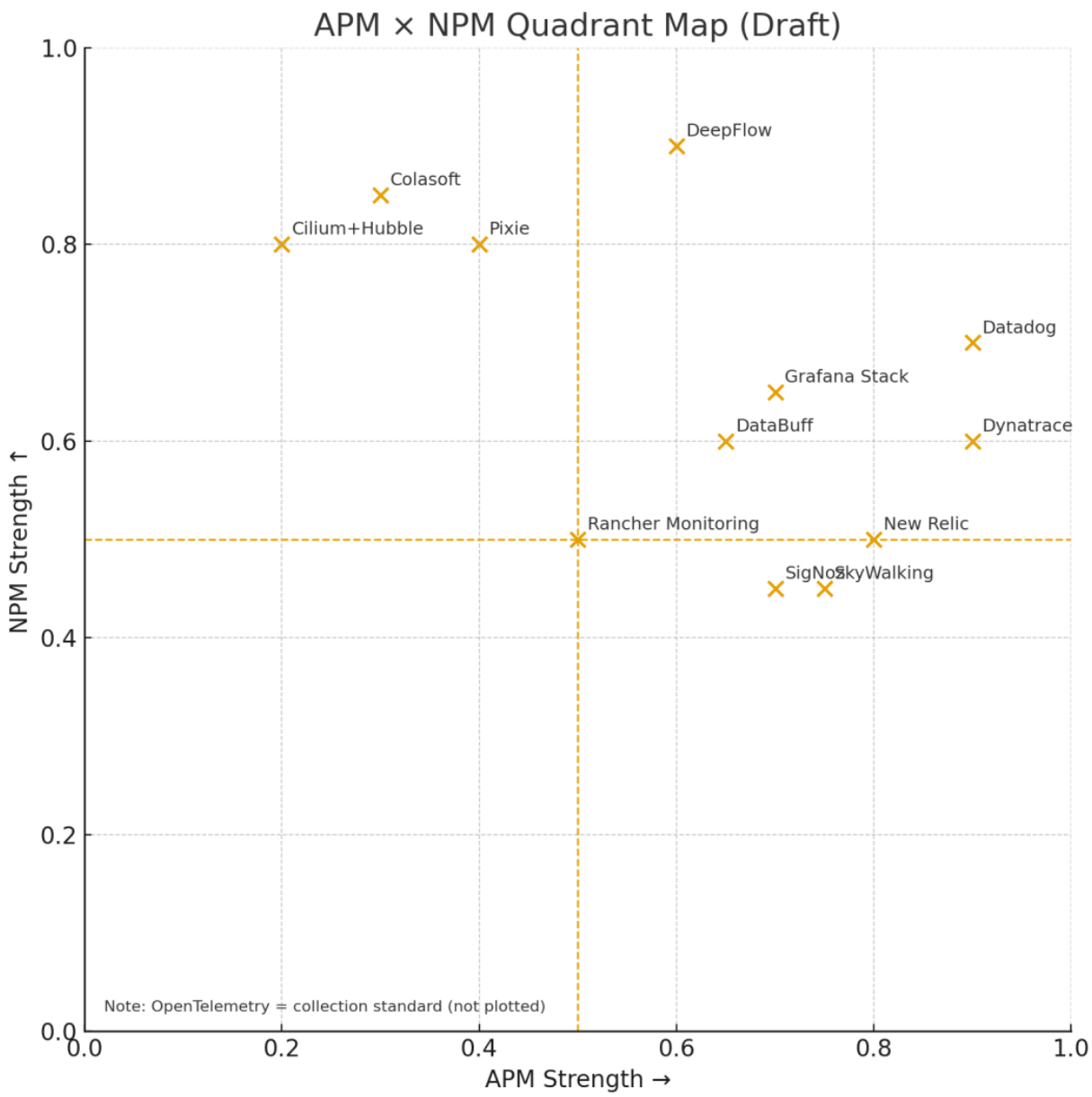

Figure 1：请在此添加图片描述

产品/厂商	类型定位	APM 能力（应用）	NPM 能力（网络）	亮点 / 差异
乘云数字 DataBuff	综合型	有（链路追踪、AI 辅助）	有（依赖/拓扑）	国产化与行业落地（成本/合规友好）
DeepFlow	NPM 起家	有一定（eBPF 侧采集）	强（eBPF+ 流量/拓扑/pcap）	零侵入、流量级全栈关联
科莱 Colasoft	NPM 为主	弱	强（抓包、协议解析、取证）	传统网络与取证场景
Dynatrace	APM 为主	强（OneAgent、因果分析、数据湖仓）	中（网络依赖/集成）	自动化一体化强，企业级
Datadog	综合型	强（APM/Trace/RUM/Profiler）	中–强（NPM/eBPF）	模块丰富、生态广
New Relic	APM 为主	强（全栈 APM）	中	免费层友好、易上手
Apache SkyWalking	APM 为主	强（Trace/Metrics/Log）	中（Rover eBPF Profiling）	开源可控，国产替代佳
OpenTelemetry（OTel）	采集标准	N/A（SDK/Collector）	N/A	统一三信号采集；需搭配后端
Grafana Labs（LGTM+Pyroscope）	平台拼装	Tempo=Trace（Grafana UI）	Loki/Mimir 为主（可联动 Cilium/Hubble）	自建成本友好、组件化灵活
Rancher（Monitoring 组合）	多集群集成	接 OTel/Tempo/Prometheus	接 Prom/Loki（非专业 NPM）	多集群可观测与运维整合
SigNoz（开源）	APM 为主	强（OTel 原生）	中（应用侧为主）	“开源 Datadog 替代”路线
Cilium + Hubble（开源）	NPM 为主	弱	强（L3–L7 拓扑/依赖）	K8s eBPF 网络观测标配
Pixie（开源）	NPM/运行时	中（运行时剖析）	强（eBPF 自动采集）	零代码采集，研发/调试友好

国内（代表性做法）

- **eBPF 无侵入 + 全栈观测** 中国移动 PaaS、腾讯等在生产使用 **DeepFlow** 做“零侵入”链路与网络观测，覆盖多语言与复杂平台；典型做法是用 eBPF 统一拿到网络与调用数据，再与既有监控/Trace 系统打通。DeepFlow
- **云厂商自研平台 + OTel 生态** 阿里云 **ARMS** 提供端到端 APM/前端/后端诊断，并

给出采样、Thread Profiling、日志与 Trace 关联等最佳实践；企业侧常把 ARMS 与 OTel/Prometheus 结合，用于混合场景。阿里云

- **大规模场景的融合观测** 金融/运营商把 ** 网络流量可观测（NPM）+ 调用链（APM）** 合并做根因定位：如银行/运营商案例用 vTap/流量侧数据+指标/Trace 组合，强化对“网络或应用谁更慢”的判定。（公开案例汇总页面可见行业分布）DeepFlow
- **数据库/中间件专项观测深耕** 国内互联网公司（如美团）对数据库在 K8s 的**专项监控与告警**做了深度集成，强调与原生监控体系融合（指标—> 早预警—> 自动化处置链）。kubeblocks.io

公司	APM（应用观测）	NPM/网络侧	主要栈/产品	公开亮点（摘）
字节跳动	自研/对外产品 APMPlus（火山引擎），内外统一用 OTel 采集；内部有 BytedTrace（统一 Tracing/Logging/Metrics 的平台化方案）	（公开资料以应用侧为主，网络侧细节披露较少）	APMPlus + OTel Collector（自动注入/接入）、CloudWeGo 生态	APMPlus 是字节内部大规模实践沉淀，对外提供全链路 APM；APMPlus×CloudWeGo 打造“一站式开发与观测”；OTel Collector 组件化部署。volcengine.com+2 火山引擎开发者社区 +2 另有 BytedTrace 作为内部一体化可观测方案报道。
京东 / 京东物流	自研 APM（京东云 APM 产品线、金融/物流落地），分布式链路追踪在金融/物流场景大规模推行	（公开材料更多聚焦业务/链路，不特指 eBPF 网络侧）	京东云 APM、金融场景分布式追踪实践（SGM 等）、大规模实时监控 +AIOps	金融场景“分批接入、快速见效”，APM 与告警/认证/CMDB 体系打通；物流线 AIOps + APM 保障大促稳定。
美团	CAT（自研开源 APM，日处理百 TB 级数据）+ 终端日志平台 Logan；还有“可视化全链路日志追踪”的体系文章	（公开文更多在应用/日志/终端侧）	CAT、Logan、可视化链路日志方案（与 ELK 辅助）	CAT 深耕多年、规模化 APM 的代表；终端实时日志 Logan 提升客户端问题定位；对“日志 → 链路”可视化方法有系统沉淀。

公司	APM（应用观测）	NPM/网络侧	主要栈/产品	公开亮点（摘）
滴滴	早期自研 DD-Falcon/夜莺 (Nightingale) 体系覆盖监控与告警，含分布式调用链、异常检测、压测平台等整体可观测构件	有提到“网络/数据通道/日志平台”配合，但细节披露以平台化监控为主	DD-Falcon/夜莺 (Nightingale) + ES/实时数据通道 + 可观测架构多阶段演进	公开演讲/文稿显示：从监控系统到异常检测、压测平台的全链路能力；夜莺作为分布式高可用监控系统在混合云/K8s 场景落地。pic.huodongjia.com+2 知乎专栏 +2

说明：NPM（网络观测）层的公开披露在国内通常**更少**，不少公司把网络侧能力融合在平台里（如流量证据、依赖拓扑、网格可视化等），但未必单独称“NPM”。如果要专看“网络/eBPF/流量侧”的公开国产案例，**DeepFlow 在运营商/银行/云厂商**的实践文章较多

国外（代表性做法）

- **自研底座 + 开源生态并用** Uber：自研 **M3**（指标）+ 开源 **Jaeger**（追踪）+ 内部采样服务组合治理可观测成本与效果，是“大规模微服务”常见路径。Uber Netflix：自研 **Atlas** 做维度时序数据的近实时遥测，满足高并发场景的运维分析。techblog.netflix.com+2netflix.github.io+2 Lyft：以 **Envoy** 为服务网格/网关基石，天然强化可观测指标/日志/追踪（后端常接 OTel/Jaeger/Tempo）。Lyft Engineering
- **开放标准 OTel 为核心“采集层”** 海外主流团队把 **OpenTelemetry** 视为“统一采集 + 语义标准”，后端选择灵活（Tempo/Jaeger/Elastic/New Relic/Dynatrace 等）。Uptrace

国外互联网大厂可观测实践：APM / NPM & 技术栈对比

公司	APM（应用观测）	NPM / 网络侧	主体技术栈 / 组件	亮点与取舍
Uber	Jaeger（自研并捐赠 CNCF；大规模分布式追踪）jaegertracing.io+1	Mesh/边车与网络指标结合（公开资料以应用侧为主）	M3（超大规模指标平台）+ Jaeger + 自研采样与告警链路（uMonitor/Neris）Uber+3Uber+3M3+3	“自研核心 + 开源输出”路线：百万级指标与大规模追踪并行，重采样与告警可扩展性。
Netflix	以 Atlas 为主的运行时遥测，APM 由多组件协同（Tracing 常接 OTel/Jaeger/Tempo）netflix.github.io+2	Envoy/Istio 等网格遥测配合（公开资料多在指标/平台侧）netflix.github.io+2	Atlas（维度时序、近实时运维洞察）+ Mesh 遥测接 OTel 后端 netflix.github.io+1	“指标为王”的实时运营视角：极强的在线查询与维度切片能力，Tracing 后端可插拔。
Google	以 OTel 标准为采集统一面（Cloud Operations/Stackdriver 体系）	Istio/Envoy 遥测 + 云探针（ThousandEyes 等为行业常见补充）	Monarch 星球级 TSDB（论文）+ OTel 采集 + Mesh 遥测 VLDB+1	“标准化采集 + 超大规模时序库”：统一三信号语义，后端服务化运营。
Meta（Facebook）	内部一体化观测（公开论文偏指标）	网络侧细节对外较少	Gorilla 内存 TSDB（论文）作为核心指标基座 VLDB+1	“指标压缩与近线价值”理念：高压缩、低延迟，强调近期数据的重要性。
Lyft	基于 OTel/Jaeger 等开源链路	Envoy（自研，后捐开源）提供 L7 遥测/依赖拓扑	Envoy + OTel/Jaeger（或 Tempo）链路后端 envoyproxy.io+1	“以网格为基础设施”的观测：网络/应用边界自然打通。
Airbnb	大量采用 Datadog（APM/告警/仪表）Datadog+1	Datadog NPM/合规探针配合（公开分享以 APM/告警为主）	Datadog SaaS 平台（监控即代码、统一告警）Datadog	“SaaS 化省心”路线：工程团队聚焦业务，代价是规模化成本需精算。
SaaS 用户如 Slack/Shopify	常见于 Datadog / New Relic / Dynatrace 组合（厂商公开案例）	NPM / Synthetics 结合 APM 使用	商用一体化平台 + OTel 接入	快速落地与运营省心，对数据量/留存的成本治理要求高。

公司	APM（应用观测）	NPM / 网络侧	主体技术栈 / 组件	亮点与取舍
Grafana 社区路线（Spotify 等）	OTel + Tempo（Trace）	Loki/Mimir + Cilium-Hubble（网络可视）	“开源拼装”：OTel → Tempo/Jaeger；Logs→Loki；Metrics→Mimir/Cortex；NPM→Hubble	“开源优先 + 成本可控”：需要更强的自运维与采样/留存策略。

快速对照（APM / NPM 归类视角）

区域	APM 主线（应用可观测）	NPM 主线（网络/流量可观测）	一体化趋势
国内	阿里云 ARMS、自研 + OTel/SkyWalking；部分采用商用平台	DeepFlow、Cilium/Hubble 在云原生里普及；运营商/金融偏爱流量侧证据链	更强调无侵入 eBPF + 统一数据面，兼顾合规与私有化落地。
国外	自研（Netflix Atlas、Uber M3）+ 开源（Jaeger/OTel）+ 商用（Datadog/Dynatrace/New Relic）	Envoy / Service Mesh + 流量遥测；NPM 与 APM 共同驱动 SLO	OTel 标准化采集成共识，后端/可视化自由拼装。

共同趋势

1. eBPF 上位、数据面“内核化”：以零侵入内核态采集作为地基，贯通网络 ↔ 应用的断点，兼顾性能与覆盖。（国内：DeepFlow；海外：Pixie / 各厂 eBPF 模块）采集标准化（OpenTelemetry），后端多样化
2. 统一（Metrics/Logs/Traces）与语义，解耦采集与存储/分析，降低厂商锁定。后端按成本与能力自由组合（Tempo/Jaeger/Mimir/Loki、或商用；也有 Uptrace 这类一体化选择）。
3. 成本治理成为刚需

- 采样：tail-based / 规则化采样，优先保留有价值的 Trace。
- 分层：冷热数据分层、短保留（原始）+ 指标化长保留（降粒度）。

- 存算与留存策略与告警价值挂钩，面向大规模可用性。

4. Trace 作为“上下文总线”：

- 以 Trace 串联指标与日志，携带服务/调用上下文，帮助做跨组件与跨团队的 RCA（根因分析）。
- “网络证据 + 应用 Trace”合并判定，提升定位效率与置信度。

5. 平台一体化 + AIOps:

- 从“自己拼零件”转向“用平台”，内置降噪、因果/异常检测与自动化修复。在海量场景下，以策略引擎和知识/因果图谱降低噪声，闭环故障处理。

概述

可观测性领域的主要数据类型包括指标、日志、追踪、事件、性能剖析以及网络元组，这些数据为 SLA、SLO 和 SLI 提供量化基础。它们共同构成对系统健康状况的多维度观察。

- **指标 (Metrics)**：以时间序列方式记录资源与性能数据，适合趋势分析与容量规划。
- **日志 (Logs)**：详细记录系统事件和调试信息，提供问题定位所需的上下文。
- **追踪 (Traces)**：关联分布式请求，展示跨服务的调用链路。
- **事件 (Events)**：表示系统状态或配置变化，常作为告警触发源。
- **性能剖析 (Profiles)**：采集函数调用与资源消耗，帮助定位性能瓶颈。
- **网络元组分析**：利用源/目的地址、端口与协议识别网络会话与流量特征。
- **SLA/SLO/SLI**：SLA 定义服务承诺，SLO 是可执行目标，SLI 为衡量目标的具体指标。

对比

类型	目的	优势	常用工具
指标	持续监控系统资源与性能	结构化、易聚合、查询高效	Prometheus, CloudWatch
日志	记录详细事件与错误	上下文丰富、可追溯	ELK Stack, Loki
追踪	展现请求的端到端路径	还原调用链路、分析延迟	Jaeger, Zipkin
事件	捕获状态变更	触发实时告警	Kubernetes Events, PagerDuty
性能剖析	分析代码级资源消耗	定位热点与瓶颈	pprof, eBPF
网络元组	识别网络会话与协议	精确流量分类	NetFlow, DeepFlow
SLA/SLO/SLI	度量服务质量	统一服务级别标准	各类 SLO 平台

使用场景

- **指标**：用于性能监控、容量规划和异常检测。
- **日志**：用于故障排查、行为审计与安全分析。
- **追踪**：用于分析分布式系统调用链，定位延迟或错误服务。
- **事件**：用于记录配置变化或异常情况，驱动自动化响应。
- **性能剖析**：用于深入分析程序性能，优化 CPU、内存等资源使用。
- **网络元组**：用于流量分析、入侵检测与协议行为研究。
- **SLA/SLO/SLI**：用于制定服务等级目标、监控达成度并驱动改进。

eBPF 技术综述

概述

- eBPF 是 Linux 内核的通用编程框架，允许在运行时安全地将自定义程序挂载到内核事件上。
- 通过 XDP、TC、kprobes、tracepoints、LSM 等钩子点，eBPF 能对网络栈、系统调用、内存和安全事件进行细粒度观测。
- eBPF 支持从 L2 到 L7 的网络协议解析，并可捕获文件、网络、进程等系统调用行为。
- 结合 Agent 模式，eBPF 可同时采集流量、日志、性能指标与元数据，实现云原生环境的全栈可观测。

对比

CBPF 与 eBPF

特性	CBPF	eBPF
用途	数据包过滤	网络过滤、性能分析、安全监控等
指令集	简单，功能有限	功能强大，支持复杂逻辑
运行环境	网络数据包	多种挂载点（网络、系统调用等）
验证器	无	严格的安全验证器
扩展性	差	强，支持用户态交互和多种映射类型
性能	高	接近 CBPF，有优化机制

传统模式与基于 Agent 的采集模式

维度	传统模式	基于 Agent 的采集模式
流量采集方式	流量镜像或旁路设备	eBPF/XDP 内核级采集
部署复杂度	需要交换机配置或旁路设备	只需在主机安装 Agent

维度	传统模式	基于 Agent 的采集模式
协议解析能力	通常仅支持 L3/L4	支持全面的 L7 协议解析
数据粒度	粗粒度流量统计	细粒度流日志与性能指标
元数据关联	需手动关联容器、进程信息	自动关联主机、容器与 Kubernetes 元数据
性能开销	镜像可能带来带宽占用	内核级采集开销极低
动态环境支持	对 Kubernetes 等动态环境支持较弱	原生适配动态环境，扩展性强
安全性	镜像存在数据泄露风险	数据在内核空间处理，安全性更高

场景

- **网络过滤与优化**：利用 XDP、TC 对 L2-L7 流量执行负载均衡、DDoS 防护和流量整形。
- **性能分析与故障排查**：通过 kprobes、tracepoints 捕获系统调用与内核事件，定位 CPU、I/O 和内存瓶颈。
- **安全监控**：结合 LSM、Seccomp 等钩子与系统调用追踪，检测异常行为并实施自定义安全策略。
- **全栈数据采集**：Agent 模式将流量、元数据和应用指标关联，构建服务拓扑并支持 APM 与安全分析。

4.1 故障排查整体流程

4.1.1 从客户端到服务端的请求路径

1. **DNS 解析**
2. **TCP 连接建立**
3. **TLS 握手（如适用）**
4. **请求发送与响应接收**

4.1.2 分层排查思路

- **应用层**
- **服务层**
- **网络层**
- **物理层**

排查应用系统故障的详细步骤

应用系统故障排查需要从应用层到网络层系统性地进行分析。以下是结合应用指标、网络指标和日志分类的详细排查流程：

1. 明确故障现象和范围

1.1 收集用户反馈

- **用户报告的现象**：页面无法加载、服务超时、功能异常等。
- **报错信息**：前端或客户端显示的具体错误提示（如 HTTP 404、500 等）。

1.2 划定故障范围

- **全局问题**：所有用户都受到影响，例如服务完全不可用。
- **局部问题**：仅部分用户或功能受到影响，例如特定接口报错。

2. 从应用层开始排查

2.1 查看应用日志　**目标**：确定是否有内部错误（服务端异常）或客户端发送无效请求。

步骤：

- **检查服务端异常**：
  - 查看应用日志是否有错误堆栈（如 NullPointerException、SQL 错误）。
  - 排查 HTTP 500 或其他 5xx 错误相关的记录。
  - **示例**：
    ```
    ERROR: Database connection failed for query: SELECT * FROM users WHERE id=123
    ```
- **检查客户端异常**：
  - 查看是否有 HTTP 400 错误（如请求格式错误）。
  - **示例**：
    ```
    WARN: Invalid input received: missing field 'username'
    ```

2.2 分析应用指标

- **响应时间**：是否显著增加，指向可能的性能问题。
- **错误率**：是否有明显的增长趋势。
- **系统资源使用率**：CPU、内存是否耗尽。

排查要点：

- 如果响应时间高：进一步分析是服务内部处理耗时，还是调用外部依赖（如数据库、第三方服务）导致。
- 如果错误率高：根据日志，确认是否是特定接口或全局问题。

2.3 验证依赖服务

- **数据库、缓存系统、消息队列等外部依赖是否正常：**
  - **数据库问题：**
    - 超时、连接池耗尽。
    - SQL 查询慢导致请求超时。
  - **缓存问题：**
    - 缓存未命中，导致频繁访问数据库。
    - Redis/Memcached 服务不可用。
  - **第三方服务问题：**
    - 调用第三方 API 超时或返回错误。

3. 从服务层排查

3.1 调用日志分析　**目标**：检查服务间调用是否正常，定位异常的服务节点。

步骤：

- **服务间调用异常：**
  - 查看调用日志是否有 HTTP 503、502 等状态码。
  - **示例：**
    ```
    2024-11-22 10:00:01 GET /api/orders/12345 Status: 503
    ```
- **调用性能问题：**
  - 检查调用链的延迟指标（Tracing 数据）。
  - 是否某个服务的响应时间显著增加。

3.2 服务依赖健康检查　**目标**：确认服务的健康状况。

步骤：

- 检查服务是否在注册中心（如 Consul、Etcd）中处于健康状态。
- 使用 `curl` 或 `ping` 测试服务接口的可用性。

4. 从网络层排查

4.1 查看流日志　**目标**：排查网络连接问题，确认流量是否正常到达。

步骤：

- 查看流日志是否有异常的流量模式：
  - 连接失败（如大量 RST 包）。
  - 丢包率是否过高。
  - **示例：**
    ```
    SrcIP: 192.168.1.10 DstIP: 10.0.0.5 Protocol: TCP Flags: RST
    ```

4.2 分析网络指标

- **延迟**：检查 RTT 是否显著增加。
- **丢包率**：高丢包可能导致服务不可用或超时。
- **重传率**：高重传可能是网络拥塞或链路质量差的信号。

4.3 网络层问题验证　**目标**：判断是否为网络问题导致的故障。

步骤：

- 使用 `ping` 或 `traceroute` 确认网络延迟和链路状态。
- 使用工具 `tcpdump` 或 Wireshark 捕获数据包，分析是否有丢包、RST 等异常。

5. 综合分析日志和指标

5.1 应用日志

- 定位代码错误或业务逻辑问题。
- **示例**：

ERROR: User authentication failed due to missing token.

markdown 复制代码

5.2 服务日志

- 追踪服务调用链，确认异常服务或节点。
- **示例**：

Service A -> Service B [Timeout]

5.3 网络日志

- 分析流量和连接状态，确认网络稳定性。
- **示例**：

TCP Retransmissions detected: 10% packets lost.

6. 常见故障场景及排查思路

故障现象	可能原因	排查方法
页面加载缓慢	服务响应时间长、数据库查询慢、网络延迟高	检查响应时间指标，分析慢查询日志，验证网络延迟和丢包。
HTTP 500 错误	服务端代码错误、资源耗尽、依赖服务不可用	查看应用日志中的异常堆栈，检查系统资源使用率，验证外部依赖服务状态。
HTTP 503 错误	服务过载或正在维护	检查系统资源使用情况，查看调用日志是否有过载信息，确认流量是否超出服务能力。

故障现象	可能原因	排查方法
HTTP 404 错误	请求资源不存在	查看客户端请求路径是否正确，分析应用日志是否有未处理的资源。
无法连接服务	网络中断、DNS 解析失败、服务未启动	检查网络流日志，测试目标服务连接状态，验证 DNS 解析。
请求被中断或取消	客户端刷新、网络不稳定	查看客户端错误日志，分析网络连接稳定性。

7. 整体流程总结

1. **确认现象**：收集用户反馈，划定问题范围。
2. **应用层分析**：查看应用日志，确认是否存在服务端或客户端错误。
3. **服务层分析**：检查服务调用链，验证依赖服务健康状态。
4. **网络层分析**：通过流日志和网络指标，排查连接和链路问题。
5. **综合定位**：结合日志、指标和用户反馈，确认根因并制定修复方案。

通过系统性的排查方法，可以快速定位故障的来源并采取对应的解决措施。

4.2 应用请求关联指标的排查

应用请求关联指标可以帮助我们深入了解系统性能、稳定性和用户体验。通过对这些指标的分析，可以快速定位问题并采取相应的改进措施。以下是针对关键指标的详细排查方法。

4.2.1 响应时间

慢请求分析

目标：识别导致应用响应时间过长的原因，优化用户体验。

步骤：

1. **数据收集**：
   - 使用性能监控工具（如 APM）收集请求的响应时间数据。
   - 记录平均响应时间、P95、P99 等关键指标。
2. **确定慢请求阈值**：
   - 根据业务需求设定响应时间的阈值（例如，超过 2 秒即为慢请求）。
3. **筛选慢请求**：
   - 从日志和监控数据中提取超过阈值的请求列表。
4. **分析慢请求原因**：
   - **代码效率**：检查是否存在低效的算法或冗余的逻辑。
   - **数据库性能**：查看是否有慢查询，检查索引是否缺失或未被利用。
   - **网络延迟**：评估网络传输时间，检查是否有带宽或延迟问题。
   - **资源竞争**：确认是否存在线程阻塞、锁竞争等问题。

解决方案：

- 优化代码逻辑，减少不必要的计算和 I/O 操作。
- 优化数据库查询，添加必要的索引，避免全表扫描。
- 使用 CDN、缓存等技术减少网络延迟。
- 引入异步处理和队列，避免阻塞主线程。

依赖服务性能

目标：确保应用所依赖的外部服务（如数据库、缓存、第三方 API）运行正常。

步骤：

1. **监控依赖服务的性能指标**：
   - 数据库连接数、查询响应时间、锁等待时间等。
   - 缓存的命中率、读写延迟。
   - 第三方 API 的响应时间和错误率。
2. **分析性能瓶颈**：
   - 确认是否因依赖服务的性能问题导致应用响应时间变长。
   - 检查网络连接是否稳定，带宽是否充足。
3. **排查服务异常**：
   - 查看依赖服务的日志，确认是否有错误或警告。
   - 检查服务是否达到容量上限，需要扩容或优化。

解决方案：

- 优化依赖服务的配置，如增加数据库连接池大小。
- 使用本地缓存或异步调用，减少对外部服务的依赖。
- 与第三方服务提供商协商，提升服务质量或寻找替代方案。

4.2.2 错误率

服务端异常 vs 客户端异常

目标：区分错误来源，以便采取针对性的解决措施。

服务端异常（5xx）：

- **特征**：服务器内部错误，无法完成请求。
- **常见原因**：
  - 代码异常（如空指针、类型转换错误）。
  - 资源耗尽（如内存泄漏、线程池耗尽）。
  - 依赖服务不可用（如数据库宕机）。

客户端异常（4xx）：

- **特征**：请求有误，服务器拒绝处理。
- **常见原因**：
  - 请求参数不合法（缺少必填字段、格式错误）。

  - 认证失败（未授权访问、Token 过期）。
  - 资源不存在（请求的 URL 错误或资源被删除）。

排查方法：

1. **收集和分类错误日志**：
   - 按照状态码和错误类型分类统计。
2. **分析错误模式**：
   - 如果 5xx 错误率高，重点检查服务器端问题。
   - 如果 4xx 错误率高，可能需要改进客户端请求或 API 文档。
3. **复现问题场景**：
   - 根据日志信息，尝试在测试环境中复现错误。

解决方案：

- 修复代码中的异常处理逻辑，增加健壮性。
- 提供详细的 API 文档，指导客户端正确使用。
- 实现参数校验，及时返回友好的错误信息。

HTTP 状态码分析

目标：通过分析 HTTP 状态码，了解系统运行状况和潜在问题。

常见状态码：

- **2xx（成功）**：
  - **200 OK**：请求成功，返回预期结果。
  - **201 Created**：资源成功创建。
- **3xx（重定向）**：
  - **301 Moved Permanently**：资源永久移动。
  - **302 Found**：资源临时移动。
- **4xx（客户端错误）**：
  - **400 Bad Request**：请求格式错误。
  - **401 Unauthorized**：未授权，需要身份验证。
  - **403 Forbidden**：服务器理解请求但拒绝执行。
  - **404 Not Found**：请求的资源不存在。
- **5xx（服务器错误）**：
  - **500 Internal Server Error**：服务器内部错误。
  - **502 Bad Gateway**：网关或代理收到无效响应。
  - **503 Service Unavailable**：服务器暂时过载或维护。

分析步骤：

1. **统计各状态码的出现频率和比例**。
2. **识别异常高的状态码**：
   - 如果 **4xx** 错误率高，检查客户端请求是否正确。
   - 如果 **5xx** 错误率高，检查服务器端是否有异常。
3. **深入分析具体错误**：
   - 查看对应的错误日志和请求参数。

- 确定错误发生的时间和频率。

解决方案：

- 针对高频错误，优化代码和配置，减少错误发生。
- 改进错误处理机制，提供更详细的错误信息。
- 定期审查和更新 API 文档，帮助客户端正确调用。

4.2.3 系统资源使用率

CPU 和内存监控

目标：确保系统有足够的计算和存储资源，防止性能下降或服务中断。

监控内容：

- **CPU 使用率**：
  - 总体 CPU 占用率，单个进程的 CPU 占用。
  - 线程数，线程的运行状态。
- **内存使用率**：
  - 总体内存占用，堆内存和非堆内存的使用情况。
  - 垃圾回收频率和停顿时间（对于 JVM）。

排查步骤：

1. **设置资源使用的警戒线**：
   - 如 CPU 使用率超过 80%，内存使用率超过 70%。
2. **实时监控和报警**：
   - 使用监控工具（如 Prometheus、Zabbix）设置报警规则。
3. **分析资源消耗来源**：
   - 查找高 CPU 或内存消耗的线程或进程。
   - 使用性能分析工具（如 jstack、jmap）获取线程堆栈和内存快照。

解决方案：

- 优化算法和代码逻辑，减少不必要的计算。
- 增加服务器资源，或进行负载均衡和集群部署。
- 调整 JVM 参数，优化垃圾回收机制。

资源泄漏检测

目标：发现并修复程序中未正确释放的资源，防止长期运行导致的资源耗尽。

常见的资源泄漏类型：

- 内存泄漏：对象未被回收，导致内存占用不断增加。
- 连接泄漏：数据库连接、网络连接未关闭。
- 句柄泄漏：文件句柄、线程未正确释放。

检测方法：

1. **监控资源使用的增长趋势**：
    - 如果资源使用率持续增长，且无法下降，可能存在泄漏。
2. **使用分析工具**：
    - **内存泄漏**：使用内存分析工具（如 MAT、VisualVM）生成堆转储，分析对象引用。
    - **连接泄漏**：检查连接池的活跃连接数，是否达到上限。
3. **代码审查**：
    - 检查代码中是否有未关闭的资源，特别是在异常处理和多线程场景下。

解决方案：

- 在代码中确保资源在使用完毕后被正确关闭(如使用 `try-finally` 或 `try-with-resources`)。
- 使用连接池和资源池，统一管理资源的创建和销毁。
- 定期重启服务作为临时措施，但应尽快修复根本问题。

4.2.4 用户留存率

用户行为分析

目标：通过分析用户的使用习惯和行为，提升用户满意度和产品竞争力。

关键指标：

- **DAU（每日活跃用户数）**、**MAU（月活跃用户数）**。
- **用户留存率**：新用户在一段时间后继续使用产品的比例。
- **用户流失率**：在一定时间内停止使用产品的用户比例。

分析步骤：

1. **收集用户行为数据**：
    - 用户登录、访问页面、使用功能等行为日志。
2. **构建用户行为模型**：
    - 细分用户群体，分析不同群体的行为特征。
3. **识别影响留存率的因素**：
    - 复杂的操作流程。
    - 功能不符合用户需求。
    - 竞争产品的影响。

解决方案：

- 优化产品功能和用户体验，降低使用门槛。
- 推出个性化推荐和定制服务，满足用户需求。
- 实施用户激励机制，如积分、优惠券等。

功能可用性检查

目标：确保应用的核心功能在任何时候都可用，提升用户满意度。

检查内容：

- **功能完整性**：所有功能是否按预期工作。
- **功能稳定性**：功能在不同环境和负载下的表现。
- **功能易用性**：用户是否容易理解和使用功能。

排查步骤：

1. **建立功能测试用例**：
    - 覆盖核心业务流程和边界条件。
2. **定期执行自动化测试**：
    - 使用测试框架（如 Selenium、Appium）进行回归测试。
3. **收集用户反馈**：
    - 通过客服、用户评论等渠道了解用户对功能的评价。

解决方案：

- 及时修复功能缺陷，优化用户操作流程。
- 提供帮助文档和指导，引导用户使用新功能。
- 持续监控功能使用情况，进行迭代改进。

4.2.5 吞吐量

负载测试

目标：评估系统在高并发访问下的性能，找出潜在的瓶颈。

测试流程：

1. **制定测试计划**：
    - 明确测试目标、范围和指标（如 QPS、响应时间、错误率）。
2. **设计测试场景**：
    - 模拟真实用户行为，包括不同的请求类型和比例。
3. **选择测试工具**：
    - 使用 JMeter、LoadRunner、Locust 等。
4. **执行测试**：
    - 从低到高逐步增加负载，观察系统的性能变化。
5. **分析测试结果**：
    - 确定系统的最大吞吐量和瓶颈所在。

解决方案：

- 优化代码和数据库查询，提高处理效率。
- 使用缓存、异步处理等技术减轻服务器压力。
- 进行水平扩展，增加服务器节点，实现负载均衡。

扩展能力评估

目标：确保系统能够随着业务增长而平滑扩展，满足未来需求。

评估内容：

- **架构可扩展性**：系统是否易于添加新功能和模块。
- **性能可扩展性**：能否通过增加资源来提升性能。
- **数据可扩展性**：能否处理不断增长的数据量。

评估步骤：

1. **分析当前架构**：
   - 识别单点瓶颈和耦合点。
2. **设计扩展方案**：
   - 考虑使用微服务、分布式架构等。
3. **验证扩展能力**：
   - 进行扩展性测试，模拟业务增长场景。

解决方案：

- 重构系统架构，拆分单体应用为微服务。
- 使用云服务和容器化技术，实现弹性扩展。
- 引入分布式缓存、消息队列等中间件，提升系统吞吐量。

通过对上述指标的分析和排查，可以全面了解应用的运行状况，及时发现和解决问题，保障系统的稳定性和用户满意度。定期监控和优化这些关键指标，对于持续提升应用性能和竞争力至关重要。

4.3 网络关键指标的排查

在网络性能和可靠性方面，关键指标的监控和分析至关重要。通过对这些指标的深入了解和排查，可以有效提升网络服务质量，确保应用系统的稳定运行。以下是优先级最高的网络指标，这些指标直接影响应用的可用性、性能和用户体验，适合在网络性能优化中重点关注：

指标	定义	重要性	优先级
延迟	数据包从源到目的地传输所需的时间。	高延迟会影响实时应用（如视频会议、在线游戏）以及 API 的响应速度。	高
丢包率	传输中丢失数据包的比例。	高丢包率会导致重传或数据丢失，影响应用性能和用户体验。	高
带宽利用率	网络已使用带宽占总带宽的百分比。	带宽不足可能导致网络拥塞，带宽利用率低可能是资源浪费的信号。	中等
网络抖动	连续数据包之间延迟的变化幅度。	高抖动会对实时服务（如 VoIP 和视频流）造成明显影响，可能导致卡顿或失真。	中等
网络可用性	网络在特定时间内可正常运行的比例。	网络中断会导致服务不可用，对高可靠性应用影响严重。	高
重传率	因丢包或错误而需要重传的数据包比例。	重传增加了网络负载，降低了传输效率，可能进一步加剧延迟和拥塞。	中等

指标	定义	重要性	优先级
TCP 连接时间	建立 TCP 连接所需的时间。	长连接时间会影响首次请求的响应速度，特别是需要频繁建立连接的应用。	中等
DNS 查询时间	域名解析到 IP 地址所需的时间。	慢速的 DNS 查询会延长应用的请求启动时间，影响整体性能。	中等
HTTP 请求成功率	成功的 HTTP 请求占总请求的比例。	HTTP 请求失败会直接导致功能不可用或用户体验下降。	高

以下是对这些关键网络指标的详细分析和排查方法。

4.3.1 带宽利用率

流量监控

目标：实时监控网络带宽的使用情况，确保网络资源被合理利用，避免带宽瓶颈。

步骤：

1. **收集流量数据**：
   - 使用网络监控工具（如 NetFlow、sFlow）收集网络接口的流量信息。
   - 记录每个接口的入站和出站流量，统计带宽利用率。
2. **分析带宽使用趋势**：
   - 观察带宽利用率的变化趋势，识别高峰时段。
   - 对比历史数据，判断是否存在异常增长。
3. **识别主要流量来源**：
   - 确定哪些应用、服务或用户消耗了大量带宽。
   - 分析流量协议和目的地址，了解流量类型。

解决方案：

- **优化流量分配**：根据业务优先级，调整带宽分配策略，确保关键业务的带宽需求。
- **升级带宽**：如果带宽长期接近饱和，考虑升级网络带宽。
- **流量压缩和优化**：采用压缩技术或协议优化，减少带宽消耗。

异常流量检测

目标：及时发现和处理异常流量，防止网络拥塞和安全问题。

步骤：

1. **设定基线**：
   - 建立正常情况下的流量模型，包括流量大小、协议分布等。
2. **实时监控**：
   - 使用入侵检测系统（IDS）或防火墙监控异常流量。
3. **识别异常行为**：

- 突发的大流量：可能是 DDoS 攻击或病毒传播。
- 非法协议或端口：可能是未授权的访问或攻击尝试。

解决方案：

- **阻断异常流量**：使用防火墙规则或流量清洗设备阻断攻击流量。
- **安全审计**：检查系统和应用的安全漏洞，更新安全补丁。
- **告警机制**：建立实时告警，及时通知管理员处理。

4.3.2 延迟

网络路径优化

目标：降低网络延迟，提高数据传输效率，提升用户体验。

步骤：

1. **测量延迟**：
   - 使用 `ping`、`traceroute` 等工具测量从源到目的地的延迟。
   - 记录往返时间（RTT），分析延迟变化。
2. **分析路径**：
   - 查看数据包经过的路由节点，识别延迟较高的节点或段。
3. **优化路径**：
   - 调整路由策略，选择延迟更低的路径。
   - 使用 MPLS、SD-WAN 等技术实现智能选路。

解决方案：

- **内容分发网络（CDN）**：将内容缓存到离用户更近的节点，减少传输距离。
- **就近接入**：在多个地区部署服务节点，实现本地化访问。
- **网络服务提供商合作**：与 ISP 协商，优化跨网传输路径。

地理位置影响

目标：理解地理位置对网络延迟的影响，针对性地优化跨地域通信。

分析：

- **物理距离**：数据传输速度受限于光速，物理距离越远，延迟越高。
- **跨国连接**：国际出口带宽有限，可能存在拥塞或政策限制。

解决方案：

- **全球加速服务**：使用云服务提供商的全球加速产品。
- **区域性部署**：在用户集中的地区部署数据中心或边缘节点。
- **协议优化**：使用加速协议（如 TCP Fast Open）减少握手时间。

4.3.3 丢包率

链路质量

目标：确保网络链路的可靠性，降低数据包丢失率。

步骤：

1. **监测丢包率**：
   - 使用 `ping` 命令发送多个 ICMP 包，统计丢包率。
   - 使用网络监控工具记录数据包传输情况。
2. **分析链路状态**：
   - 确定丢包发生的节点或链路段。
   - 检查网络设备的错误统计信息（如接口错误、碰撞）。

解决方案：

- **更换或修复故障链路**：对于物理损坏的线路，及时维修。
- **提高链路冗余**：采用链路聚合或备用路径，防止单点故障。
- **调整网络参数**：优化 MTU 值，避免分片导致的丢包。

设备故障排查

目标：检查网络设备的健康状况，排除因设备故障导致的丢包。

步骤：

1. **设备日志检查**：
   - 查看交换机、路由器、防火墙等设备的系统日志。
   - 关注错误信息、异常重启、端口状态等。
2. **接口状态检查**：
   - 检查网络接口的状态，是否有频繁的 up/down 变化。
   - 查看接口的错误计数器，如 CRC 错误、帧错误。
3. **固件和配置**：
   - 确认设备固件是否需要更新，配置是否正确。

解决方案：

- **更换故障设备**：对于无法修复的设备，及时更换。
- **更新固件**：升级设备固件，修复已知问题。
- **优化配置**：调整设备配置，避免资源耗尽或冲突。

4.3.4 网络抖动

实时应用影响

目标：降低网络抖动对实时应用（如 VoIP、视频会议）的影响，保证服务质量。

分析：

- **网络抖动**：指数据包到达时间的变动性，过高的抖动会导致音视频卡顿。
- **实时应用敏感性**：实时应用对延迟和抖动要求较高。

检测方法：

- 使用专用工具（如 iperf）测量网络抖动。
- 监控实时应用的质量指标（如 MOS 分数）。

解决方案：

- **QoS 策略**：配置服务质量策略，为实时流量设置优先级。
- **带宽保障**：预留带宽，防止其他流量挤占资源。
- **缓冲机制**：在应用层增加缓冲，平滑抖动影响。

网络稳定性提升

目标：通过网络优化，提高整体网络的稳定性，减少抖动。

措施：

- **消除网络瓶颈**：升级过载的网络设备或链路。
- **减少网络层次**：简化网络拓扑，降低转发延迟。
- **使用稳定的传输介质**：优先采用光纤等高质量链路。

4.3.5 网络可用性

冗余设计

目标：通过冗余设计，提升网络的可靠性，防止单点故障。

策略：

- **链路冗余**：使用双上行链路、环形拓扑等设计，保证链路故障时有备用路径。
- **设备冗余**：部署冗余的核心交换机、路由器，实现设备级备份。
- **多数据中心**：在不同区域部署数据中心，实现业务容灾。

实现方法：

- **协议支持**：使用 STP、VRRP、HSRP 等协议，实现冗余切换。
- **负载均衡**：采用负载均衡设备，分摊流量，提升可用性。

故障切换机制

目标：确保在故障发生时，系统能够自动切换到备用路径或设备，最小化服务中断。

步骤：

1. **配置故障检测**：
   - 设置心跳检测，实时监控设备和链路状态。
2. **设定切换条件**：
   - 明确何时触发切换，如连续心跳失败次数。

3. **测试切换流程**：
    - 定期演练故障切换，确保机制有效。

解决方案：

- **自动化切换**：使用网络协议自动完成故障切换，无需人工干预。
- **快速收敛**：优化路由协议参数，减少切换时间。

4.3.6 重传率

拥塞控制

目标：通过拥塞控制，降低数据包重传率，提升网络效率。

分析：

- **高重传率原因**：网络拥塞、丢包导致发送端重新发送数据。
- **影响**：增加网络负载，降低传输效率。

措施：

- **调整窗口大小**：优化 TCP 的拥塞窗口，避免发送过多数据导致拥塞。
- **主动队列管理（AQM）**：在路由器中使用 RED、ECN 等机制，预防拥塞。

传输优化

目标：优化传输协议和参数，减少重传，提升传输性能。

策略：

- **使用高效协议**：如在高延迟网络中使用 QUIC 协议。
- **协议参数优化**：调整 TCP 的超时时间、重传次数等参数。

解决方案：

- **传输压缩**：压缩数据，提高传输效率。
- **流控机制**：根据接收端的处理能力，调整发送速率。

4.3.7 TCP 连接时间

握手延迟分析

目标：降低 TCP 连接建立的时间，提升用户访问速度。

分析：

- **三次握手**：TCP 连接需要进行三次握手，延迟会累积。
- **延迟因素**：网络延迟、服务器响应时间。

检测方法：

- 使用抓包工具，测量 SYN、SYN-ACK、ACK 包的时间间隔。

- 分析握手过程中是否有重传或超时。

解决方案：

- **TCP Fast Open**：减少握手次数，缩短连接建立时间。
- **保持连接**：使用长连接（如 HTTP/2），减少重复握手。

服务器性能

目标：提升服务器处理连接的能力，避免成为瓶颈。

措施：

- **优化系统参数**：调整服务器的最大连接数、半连接队列长度等参数。
- **升级硬件**：提高 CPU、内存等资源，满足高并发需求。
- **负载均衡**：分摊连接请求，防止单台服务器过载。

4.3.8 DNS 查询时间

DNS 服务优化

目标：降低 DNS 查询时间，加快域名解析速度。

措施：

1. **本地部署 DNS 服务器**：
   - 搭建本地 DNS 解析服务器，减少查询跳数。
2. **优化 DNS 配置**：
   - 使用权威 DNS 服务器，减少递归查询。
3. **提高 DNS 服务器性能**：
   - 增加服务器带宽和处理能力，缩短响应时间。

缓存策略

目标：通过缓存机制，加快 DNS 解析，提高查询效率。

策略：

- **客户端缓存**：浏览器和操作系统层面的 DNS 缓存。
- **网络缓存**：在网络设备（如路由器）中启用 DNS 缓存。
- **TTL 设置**：适当延长 DNS 记录的 TTL，减少重复查询。

注意事项：

- **缓存一致性**：在 DNS 记录变更时，需考虑缓存的影响，避免解析错误。
- **安全性**：防止缓存中毒，使用 DNSSEC 等安全措施。

4.3.9 HTTP 请求成功率

服务器健康检查

目标：确保服务器运行正常，提升 HTTP 请求的成功率。

措施：

1. **定期检测**：
   - 使用监控工具，定期检查服务器的状态和响应。
2. **健康检查接口**：
   - 开发专用的健康检查接口，返回服务器运行状态。
3. **日志分析**：
   - 分析服务器日志，发现异常请求和错误码。

解决方案：

- **自动重启服务**：在检测到异常时，自动重启服务或切换到备用服务器。
- **容量规划**：根据负载，及时扩容，防止服务器过载。

网络异常处理

目标：处理网络异常情况，确保 HTTP 请求的可靠传输。

措施：

- **重试机制**：在请求失败时，客户端或中间件自动重试。
- **超时设置**：合理设置请求和连接的超时时间，避免长时间等待。
- **错误处理**：在网络异常时，提供友好的错误提示和应急措施。

解决方案：

- **使用可靠的传输协议**：如启用 HTTP/2 的多路复用，减少连接数量。
- **负载均衡和容错**：通过负载均衡器，自动将请求路由到健康的服务器。

TCP 传输超时机制整理

300ms 内没有收到 ACK 的场景

在 TCP 协议中，**300ms 内没有收到 ACK** 这种情况通常发生在以下两个阶段之一：**连接建立阶段** 或 **数据传输阶段**。

---

1. 连接建立阶段 (Connection Establishment Phase)

描述　这是 TCP 的三次握手过程，用于建立可靠的连接。

ACK 的作用

- 在三次握手过程中，每个通信方会使用 SYN 和 ACK 标志来确认连接的进展。
- 发送端发送 SYN 后，等待接收端返回 SYN+ACK 确认。
- 如果在 **300ms 内未收到 ACK**，发送端会超时重试，直到达到最大重试次数或完全放弃连接。

示例流程

1. **SYN**：发送方发送 SYN 请求建立连接。
2. **SYN+ACK**：接收方返回 SYN+ACK。
3. **ACK**：发送方确认 ACK，连接建立完成。

300ms 超时的情况

- 如果在第一步或第二步中，SYN 或 SYN+ACK 没有被成功接收或返回，则会进入超时逻辑。
- 可能原因：
  - 网络延迟。
  - 数据包丢失。
  - 目标主机不可达。

---

2. 数据传输阶段 (Data Transmission Phase)

描述　在连接建立后，双方通过 TCP 数据包进行数据传输，每个数据段都需要 ACK 确认。

ACK 的作用

- 接收端在成功接收数据包后，发送 ACK 确认包，告知发送端可以发送下一个数据段。
- 如果发送端在 **300ms 内未收到 ACK**，会触发重传机制。

示例流程

1. **发送数据**：发送端发送一个 TCP 数据包。
2. **返回 ACK**：接收端成功接收数据后，返回 ACK 确认。
3. **下一数据段**：发送端收到 ACK 后，发送下一个数据段。

300ms 超时的情况

- 如果接收端未能按时发送 ACK，可能导致数据包重传。
- 可能原因：
  - 数据包丢失。
  - 接收端处理延迟。
  - 网络延迟较高或网络抖动。

---

3. 阶段区别总结

阶段	ACK 的作用	300ms 超时的可能性
连接建立阶段	确保三次握手成功建立连接	网络延迟、丢包、目标主机不可达。
数据传输阶段	确认接收数据包，并告知发送端继续发送下一数据段	数据包或 ACK 丢失、接收端延迟、网络抖动。

如何应对 300ms 未收到 ACK 的情况?

1. 动态调整 RTO（重传超时时间）

- RTO 是根据网络条件动态计算的，初始值可能为 300ms。
- 如果网络较慢，可通过调整参数增大超时时间。

2. TCP 快速重传 (Fast Retransmit)

- 如果发送方收到多个重复的 ACK（表明某些包未达），可能会提前触发重传，而不是等待 RTO 超时。

3. 网络优化

- 检查是否存在网络拥塞或丢包，优化网络路径或增加带宽。

通过对网络关键指标的深入分析和有效排查，可以显著提升网络的性能和可靠性。定期监控这些指标，及时发现并解决问题，对于保障应用系统的正常运行和用户满意度至关重要。

TCP 时延相关指标表

指标	定义	阶段	影响因素	区别
TCP 连接时延	三次握手所需时间	连接建立阶段	网络延迟、服务器响应速度	测量连接建立效率，不涉及后续数据传输。
RTT	数据包从源到目的地再返回的总时间	全过程	网络物理距离、拥塞	网络底层延迟的综合反映，是其他时延的基础。
初始时延	从请求到收到首字节的时间（TTFB）	连接建立 + 数据开始	TCP 握手时延、服务器响应	包括了 TCP 握手和服务器处理时间，反映用户可感知的响应延迟。

指标	定义	阶段	影响因素	区别
数据传输时延	从开始发送到完全接收数据的时间	数据传输阶段	带宽、窗口大小、丢包重传	强调传输阶段性能，与握手阶段无关。
抖动	数据包到达时间的波动性（Jitter）	全过程	网络稳定性	关注时间一致性，而非单次时延，主要影响实时性要求较高的应用。
重传时延	因丢包重传导致的额外延迟	数据传输阶段	丢包率、网络质量	表示丢包的代价，直接影响传输性能。
慢启动时延	TCP 连接初期，数据传输受限所需的时间	数据传输阶段	窗口大小、网络拥塞	仅影响传输初期，与稳定阶段性能无关，适用于大文件传输分析。

TCP 建连异常相关指标表

指标	定义	异常阶段	常见原因	区别
TCP 连接失败率	未完成三次握手的连接比例	全过程	网络丢包、服务器未监听、超时	反映总体连接失败的宏观指标，是性能排查的起点。
TCP 超时率	建连超时的比例	全过程	高延迟、丢包重传、服务器过载	专注超时问题，通常与网络延迟或服务器负载有关。
TCP 重试次数	建连中因丢包需要重传的次数	全过程	丢包率高、拥塞、路由问题	表示网络传输可靠性，过高可能导致建连失败或性能下降。
SYN 丢包率	客户端发送的 SYN 包未收到响应的比例	第一步	链路丢包、防火墙拦截	专注建连的第一步，反映初始连接的成功率。
SYN-ACK 失败率	服务端返回的 SYN-ACK 包未到达客户端的比例	第二步	服务端丢包、防火墙或 NAT 问题	分析服务端到客户端方向的网络问题。
RST 包率	建连过程中收到 RST 包的比例	全过程	服务端拒绝连接、防火墙或策略问题	表明服务端主动中断连接，可能涉及配置错误或策略限制。
建连延迟	完成三次握手的时间	全过程	高 RTT、丢包重传	偏向时延分析，高延迟可能导致连接超时或多次重传。
拒绝连接率	被服务器拒绝的连接比例	全过程	服务器端口未监听、超出连接限制	表明服务器端主动拒绝连接的情况。
防火墙拦截率	数据包被防火墙拦截的比例	全过程	防火墙规则错误或策略限制	特指中间设备干预导致的连接失败。
NAT 超时率	因 NAT 转换失败导致建连异常的比例	全过程	NAT 表溢出、超时配置不当	特指 NAT 相关问题，区别于常规的丢包或超时原因。

TCP 超时类型

超时类型 (中文)	状态 (英文)	默认值 (Linux)	默认值 (Windows)	默认值 (macOS)
初始 RTO	Initial RTO (Retransmission Timeout)	200ms - 1 秒	300ms	1 秒
最大重传超时	Maximum Retransmission Timeout	13-30 分钟	240 秒	9 分钟
Keepalive 超时	Keepalive Timeout	7200 秒（2 小时）	7200 秒（2 小时）	7200 秒（2 小时）
连接建立超时	Connection Establishment Timeout	63 秒	21 秒	75 秒
连接重置超时	Connection Reset Timeout	0 秒	0 秒	0 秒

4.4 日志分析与故障定位

4.4.1 网络日志分析

流日志

流日志记录网络层的元数据，如源 IP、目标 IP、端口、协议和流量大小，用于分析网络流量和连接问题。

- **用途**：
  - 分析网络流量。
  - 排查网络连接问题（如丢包、延迟）。
  - 安全事件监控（如异常访问检测）。
- **典型工具**：AWS VPC Flow Logs、Azure NSG Flow Logs、NetFlow、sFlow。

数据包日志

数据包日志捕获网络中的详细数据包信息，包括 MAC 地址、IP 地址、端口和负载内容，用于深入分析网络问题。

- **用途**：
  - 分析 MTU 不匹配或路由环路问题。
  - 网络入侵检测。
- **典型工具**：Wireshark、Tcpdump。

4.4.2 服务日志分析

调用链追踪

记录服务间调用或客户端与服务间的调用详情，包括请求时间、路径、参数和响应时间。

- **用途**：
  - 排查接口错误或调用失败。
  - 分析接口性能。
  - 监控服务健康状况。
- **典型工具**：API Gateway Logs、Nginx Access Logs、Jaeger、Zipkin。

接口性能监控

通过调用日志中的响应时间和错误率，监控接口性能，定位瓶颈。

- **用途**：
  - 优化服务性能。
  - 发现高延迟或高错误率的接口。
- **典型工具**：Application APM Tools、SkyWalking。

4.4.3 应用日志分析

错误堆栈

记录应用运行中的详细错误信息，包括异常堆栈和出错时的上下文。

- **用途**：
  - 调试和修复代码问题。
  - 记录未捕获的异常。
- **典型工具**：Log4j、SLF4J、ELK Stack。

业务逻辑验证

记录应用内部的关键业务事件（如订单状态、用户登录）和逻辑流程。

- **用途**：
  - 验证业务逻辑的正确性。
  - 追踪用户行为。
- **典型工具**：Fluentd、Splunk。

4.4.4 排查思路：从客户端到服务端

日志分析的排查思路可以按照以下顺序进行：**应用日志 -> 接口调用日志 -> 流日志**，逐步深入定位问题。

1. 应用日志

- **检查内容**：
  - 用户行为记录：是否触发了对应的请求。
  - 错误堆栈：未捕获的异常是否导致任务中断。
  - 业务事件：如订单生成是否正常。
- **典型问题**：
  - 用户提交的请求数据格式错误。
  - 应用逻辑中处理边界条件失败。
  - 系统调用超时。
- **工具**：
  - 日志工具：Log4j、ELK Stack。
  - 调试工具：IDE 调试功能。

---

2. 接口调用日志

- **检查内容**：
  - 请求路径、参数和状态码是否正确。
  - 响应时间是否异常。
  - 调用链是否完整。
- **典型问题**：
  - HTTP 4xx（如 400 格式错误，401 未授权）。
  - HTTP 5xx（如 500 服务端错误，503 服务不可用）。
  - 服务间调用超时。
- **工具**：
  - 日志工具：Nginx Access Logs、API Gateway Logs。
  - 调用链追踪工具：Jaeger、Zipkin。
  - 性能监控工具：New Relic、SkyWalking。

---

3. 流日志

- **检查内容**：
  - 数据包是否成功传输。
  - 网络连接状态（如 SYN/ACK 是否成功）。
  - 是否有丢包、重传或延迟问题。
- **典型问题**：
  - 客户端网络连接失败。

  - 防火墙阻止流量。
  - 高并发导致服务端资源耗尽。
- **工具**：
  - 网络流日志工具：AWS VPC Flow Logs、NetFlow。
  - 数据包分析工具：Wireshark、Tcpdump。
  - 防火墙监控：Cloud Firewall Logs。

排查流程示例

示例 1：客户端报错“服务不可用”

1. **应用日志**：
   - 检查日志是否记录到用户触发了 API 请求。
   - 发现日志中正常触发了 `/api/v1/orders` 请求。
2. **接口调用日志**：
   - 查看接口返回 HTTP 503（服务不可用）。
   - 调用链显示 `order-service` 超时。
3. **流日志**：
   - 检查流量记录，发现服务端流量激增。
   - 分析服务端资源，确认因高并发导致 CPU 和内存耗尽。

示例 2：客户端显示“网络请求失败”

1. **应用日志**：
   - 日志中无 API 调用记录，怀疑网络问题。
2. **接口调用日志**：
   - 无相关日志记录。
3. **流日志**：
   - 检查流量日志，发现客户端 IP 未连接成功。
   - 使用抓包工具分析，发现 DNS 查询失败。
   - 排查 DNS 配置并修复问题。

4.4.5 日志类型分类表

大分类	日志类型	对应层面	主要内容	用途	典型工具或技术
应用日志	应用日志	应用层（业务逻辑）	- 应用程序内部运行时的详细信息（如业务处理流程、变量状态、异常堆栈）。	- 排查应用内部逻辑错误。- 记录业务事件（如订单处理、用户登录）。- 调试和优化代码。	Log4j、SLF4J、ELK Stack、Fluentd
应用日志	HTTP访问日志	应用层（HTTP协议）	- HTTP 请求和响应：方法（GET/POST）、路径、状态码、响应时间、用户代理。	- 分析网站流量。- 监控接口性能和错误率。	Apache Logs、Nginx Access Logs
应用日志	DNS查询日志	应用层（DNS协议）	- DNS 请求记录：查询域名、查询类型、响应时间。- 是否缓存命中、查询结果（成功或失败）。	- 分析域名解析性能。- 检测恶意域名访问或 DNS 放大攻击。	BIND DNS Logs、DNSCrypt Proxy Logs
应用日志	TLS握手日志	会话层（TLS协议）	- TLS 握手的详细信息：握手阶段、加密算法、证书验证结果。	- 检测加密通信的失败原因（如证书过期、算法不匹配）。- 分析加密通道的安全性。	OpenSSL Logs、Wireshark、SSL Labs Reports
服务日志	调用日志	服务层（接口层）	- 服务间或客户端与服务间的调用详情（如请求时间、参数、状态码、响应时间）。	- 排查接口错误或调用失败。- 分析接口性能（如响应时间和错误率）。- 监控服务健康状况。	API Gateway Logs、Nginx Access Logs、Application APM Tools

大分类	日志类型	对应层面	主要内容	用途	典型工具或技术
网络日志	流日志	网络层/传输层	- 网络流量的元数据（如 IP 地址、端口、协议、流量大小）。- 不包含具体的传输内容（仅元信息）。	- 分析网络流量。- 排查网络连接问题（如丢包、延迟）。- 安全事件监控（如检测异常访问）。	AWS VPC Flow Logs、Azure NSG Flow Logs、NetFlow、sFlow
网络日志	数据包日志	链路层/网络层	- 数据包的详细信息：MAC 地址、IP 地址、端口、协议标识。- 数据包的头部和负载内容。	- 深入分析网络问题（如 MTU 不匹配、路由环路）。- 网络入侵检测。	Wireshark、Tcpdump
网络日志	TCP 连接日志	传输层	- TCP 连接的状态：SYN、ACK、FIN、RST。- 重传次数、握手失败。	- 分析连接状态（如建立失败）。- 排查丢包或重传问题。	Wireshark、Nmap
应用日志	SMTP 日志	应用层（SMTP 协议）	- 邮件发送和接收的状态：邮件来源、目标、传输状态、延迟、错误代码。	- 分析邮件服务器性能。- 检测垃圾邮件或邮件服务异常。	Postfix Logs、Exim Logs
应用日志	VPN 日志	会话层/网络层	- VPN 连接的状态：握手时间、加密方式、连接 IP、断开原因。		

4.5 典型故障场景与案例分析

4.5.1 服务端异常

- **资源耗尽** 场景：服务器的 CPU、内存或磁盘空间达到上限，导致服务性能下降或无法正常提供服务。解决：

1. 监控资源使用情况，识别瓶颈。
2. 通过水平扩展或增加资源容量缓解压力。
3. 优化代码和服务逻辑，减少资源占用。

- **依赖服务故障** 场景：上游服务或第三方依赖出现问题，例如认证服务超时或存储服务不可用。解决：
  1. 在日志中查找与依赖服务的交互记录，确认异常点。
  2. 配置重试策略和降级方案，减少对依赖的影响。
  3. 与服务提供方协作，确认问题原因。

4.5.2 客户端异常

- **请求格式错误** 场景：客户端发送的请求不符合服务端要求，导致请求被拒绝或返回错误响应。解决：
  1. 检查客户端日志和抓包，确认请求的具体格式和参数。
  2. 比对服务端 API 文档，确保请求的字段和格式符合要求。
  3. 增加客户端的请求校验逻辑，提前发现问题。
- **网络连接失败** 场景：客户端与服务端之间的网络中断，导致无法访问服务。解决：
  1. 检查客户端的网络环境，如 DNS 配置、代理设置和防火墙规则。
  2. 使用 `ping` 或 `traceroute` 检测网络链路状态。
  3. 针对不稳定的网络，启用断线重连或备用节点机制。

4.5.3 综合案例

案例 1：应用-DNS 故障快速排查

场景：用户反馈应用无法访问，测试发现域名解析失败。解决步骤：1. 使用 `nslookup` 或 `dig` 检查 DNS 配置和解析结果。2. 排查 DNS 服务器状态，确认是否可用。3. 更新 DNS 缓存，或临时切换到其他 DNS 服务（如 8.8.8.8）。4. 检查网络路径中是否存在拦截（如防火墙或 NAT 问题）。

案例 2：应用-IO 性能问题定位分析

场景：批量处理任务时，应用性能急剧下降，分析发现磁盘 IO 过高。解决步骤：1. 使用 `iostat` 或 `iotop` 工具查看磁盘 IO 状态，确认读写是否过于频繁。2. 检查文件系统是否存在碎片，或日志文件是否持续增长。3. 对 IO 密集型任务进行限速或分批处理，减轻瞬时压力。4. 升级磁盘性能（如 SSD）或优化数据访问逻辑（如缓存）。

案例 3：应用-DB 性能问题定位分析

场景：数据库查询速度变慢，影响整体服务响应。解决步骤：1. 使用数据库自带的查询分析工具（如 EXPLAIN）定位慢查询。2. 检查索引是否存在或合理，避免全表扫描。3. 查看数据库服务器的资源使用情况，是否达到瓶颈。4. 使用分库分表或读写分离的方式优化查询效率。

案例 4：应用-中间件性能问题定位分析

场景：消息队列积压严重，消费者处理速度明显下降。解决步骤：1. 检查消息队列的队列长度和吞吐量，确认瓶颈点。2. 查看消费者的消费速率和日志，是否存在逻辑错误或阻塞。3. 增加消费者实例或优化消费逻辑，提升处理能力。4. 对非关键消息启用丢弃或延迟消费策略，避免影响核心业务。

案例 5：定位网络时延类故障

场景：用户反馈访问服务延迟明显增加，分析发现网络延迟异常。解决步骤：1. 使用 `ping` 或 `traceroute` 工具确认延迟节点或链路。2. 检查网络路由是否发生变更，或是否存在拥塞点。3. 使用 SD-WAN 或 MPLS 等技术优化网络路径。4. 针对跨区域访问，启用 CDN 加速或服务节点本地化。

案例 6：洞察、定位云网络丢包类故障

场景：云环境中的服务稳定性下降，监控显示存在高丢包率。解决步骤：1. 使用云平台提供的网络监控工具查看丢包位置（如 VPC 内或跨区域）。2. 排查相关网络设备（如网关、负载均衡器）的日志和健康状态。3. 检查服务端和客户端的 MTU 设置，避免分片问题。4. 增加链路冗余，或切换到不同区域的节点降低丢包影响。

通过对这些典型案例的分析和总结，可以为实际问题排查提供清晰的流程和有效的解决方案，同时为提升系统可靠性积累宝贵经验。

概述

结合原有的故障排查章节，本节总结了从客户端请求到服务端响应的完整排查思路，并梳理了应用指标、网络指标和日志分析的要点：

- **排查流程**：按照请求路径自上而下，从 DNS、TCP/TLS 到应用处理逐层分析，先确认故障范围，再定位应用层、服务层和网络层的异常。
- **应用指标**：关注响应时间、错误率、系统资源、用户留存和吞吐量，通过 APM、监控面板等工具定位慢请求和依赖服务瓶颈。

- **网络指标**：优先监控延迟、丢包率、带宽利用率和 TCP 连接时间，结合流量趋势和路径分析优化网络质量。
- **日志分析**：从应用日志、接口调用日志到网络流日志逐级深入，利用调用链追踪和抓包等方式快速定位问题根因。

对比

维度	关注重点	常用工具	典型问题示例
应用指标	响应时间、错误率、资源使用、吞吐量	APM、监控面板、性能测试工具	慢查询、依赖服务超时
网络指标	延迟、丢包率、带宽利用率、重传率	NetFlow、ping、traceroute	链路拥塞、跨区域访问延迟
日志分析	应用日志、调用日志、网络流/数据包日志	ELK、Jaeger、Tcpdump	错误堆栈、HTTP 4xx/5xx 等

场景

- **服务端资源耗尽**：CPU/内存达到上限导致 5xx 错误；通过扩容、优化代码或限流解决。
- **依赖服务故障**：认证或存储服务超时；配置重试与降级并与提供方协作修复。
- **请求格式错误**：客户端参数不合法返回 4xx；检查日志与 API 文档并增加校验。
- **网络连接失败**：DNS 配置错误或链路中断；使用 ping、traceroute 等定位并修复。
- **典型案例**：如 DNS 故障、IO 瓶颈、数据库慢查询或网络时延异常，均需结合上述指标和日志逐步排查。

PostgreSQL + ClickHouse 高性能分层写入架构

1. 架构目标

- 模仿 GreptimeDB 的分层写入能力：热写入 PostgreSQL（TimescaleDB 扩展）并按需冷聚合到 ClickHouse。
- 支持多协议接入（REST / gRPC / OpenTelemetry / Arrow Flight）。
- 动态表结构注册，借助 JSONB 与元数据表管理 schema。
- 易于在单机 PoC、集群与存算分离之间平滑扩展。

2. 架构总览

```
graph TD
    subgraph Client
        A[Vector Agent]
    end
```

```
    subgraph Transport
        A --> B[Unified API (REST/gRPC/OTel)]
    end

    subgraph Storage
        B --> C1[PostgreSQL (TimescaleDB + JSONB)]
        B --> C2[ClickHouse (OLAP 聚合)]
    end

    subgraph Query
        C1 --> D[Grafana / SQL API]
        C2 --> D
    end
```

3. 核心组件

3.1 API 层

使用 Go / Rust 编写 Ingest 网关：- `/write/metrics` → TimescaleDB - `/write/logs` → ClickHouse - `/schema/register` → 写入 `table_registry` 元数据表

3.2 PostgreSQL 写入端（实时指标）

```
CREATE TABLE metrics_events (
  time        TIMESTAMPTZ NOT NULL,
  app         TEXT,
  host        TEXT,
  labels      JSONB,
  value       DOUBLE PRECISION,
  trace_id    TEXT,
  level       TEXT
);
SELECT create_hypertable('metrics_events', 'time');
```

3.3 ClickHouse 写入端（归档聚合）

```
CREATE TABLE logs_events (
  timestamp DateTime,
  app       String,
  host      String,
  trace_id  String,
  message   String,
  labels    Nested(k String, v String)
) ENGINE = MergeTree()
```

```
PARTITION BY toYYYYMMDD(timestamp)
ORDER BY (timestamp, app);
```

3.4 Schema 管理

```
CREATE TABLE table_registry (
  table_name TEXT PRIMARY KEY,
  schema     JSONB,
  created_at TIMESTAMPTZ DEFAULT now()
);
```

注册示例：

```
{
  "columns": [
    {"name": "time",   "type": "timestamp"},
    {"name": "value",  "type": "double"},
    {"name": "labels", "type": "jsonb"}
  ]
}
```

4. 参考配置

4.1 Vector 采集配置

```
[sources.prom]
type = "prometheus_scrape"
endpoints = ["http://localhost:9100/metrics"]

[sources.logs]
type = "journald"

[transforms.jsonify_logs]
type = "remap"
inputs = ["logs"]
source = '''
.structured = parse_json!(string!(.message))
'''

[sinks.pg]
type = "postgresql"
inputs = ["prom", "jsonify_logs"]
database = "metrics"
endpoint = "postgresql://user:pass@pg:5432/metrics"
table = "observability_events"
mode = "insert"
```

```
compression = "zstd"

[sinks.ch]
type = "clickhouse"
inputs = ["jsonify_logs"]
endpoint = "http://clickhouse:8123"
database = "default"
table = "logs"
compression = "gzip"
```

4.2 Grafana 查询示例

- **PostgreSQL 数据源**

```
SELECT time_bucket('1 minute', event_time) AS ts, COUNT(*)
FROM observability_events
WHERE node = $__variable_node
  AND raw->>'source' = $__variable_source_type
  AND $__timeFilter(event_time)
GROUP BY ts
ORDER BY ts;
```

- **ClickHouse 数据源**

```
SELECT node, count() AS total
FROM logs
WHERE timestamp >= $__from AND timestamp <= $__to
GROUP BY node
ORDER BY total DESC
LIMIT 5;
```

5. 部署演进路径

阶段	特性	说明
单节点一体化	所有组件运行在一台主机	适合 PoC / 开发环境
多副本集群	API、PG、CH 分别部署为独立节点，可接入负载均衡	支持横向扩展与高可用
存算分离	引入 Kafka 等缓冲层，PG/CH 采用分布式与 Replication	写入与查询解耦，适合大规模场景

6. 测试验证方案

6.1 测试目标

目标类别	说明
□ 吞吐能力	是否能支撑 50K+ IOPS 持续写入并在高峰期达到 100K/s
□ 查询能力	事件查询/统计分析响应稳定在 100~500ms
□ 分层写入正确性	标签/类型是否正确路由至 PostgreSQL 与 ClickHouse
□ 数据一致性	PostgreSQL 中 Trace、事件无漏写
□ 可观测性	Vector/PG/CH 与系统资源均可监控

6.2 基础环境准备

数据源模拟

工具	用途
vector generator / log-generator	模拟高负载日志流
OpenTelemetry demo	生成 Trace 数据
prometheus-testgen	生成 Metrics 样本

写入目标部署

系统	推荐配置
PostgreSQL	16+ + TimescaleDB + pg_partman
ClickHouse	24.4+ + ReplicatedMergeTree
Vector	支持 transforms 路由、sink 至 PG/CH
监控组件	node_exporter、pg_exporter、vector internal、CH monitor

6.3 测试维度与方法

6.3.1 吞吐验证

项目	方法	期望
最大持续写入	持续向 Vector 发送 JSON logs @ 50K/s，观察是否丢包	ACK latency < 500ms
高峰瞬时写入	峰值 100K/s 持续 1 min，观察 PG/CH 报错情况	无错误、吞吐稳定
网络抖动模拟	通过 `tc` 注入延迟/丢包，观察重试	数据无丢失

6.3.2 数据分层准确性

验证点	检查项
Trace 事件 → PostgreSQL	是否按时间/traceID 正确分区落表
普通日志 → ClickHouse	是否按标签写入 MergeTree 表
异常标签	是否触发 fallback/dead-letter

6.3.3 查询验证

类型	查询目标	预期性能
PostgreSQL	精确事件 → TraceID 查询	<200ms
ClickHouse	日志聚合 → app error rate	<1s
联合	Trace + 日志聚合视图	<1.5s

6.3.4 资源使用验证

组件	监控项	期望值
Vector	CPU < 300m、Mem < 300MB	
PostgreSQL	IOPS < 40K / WAL flush latency	
ClickHouse	Merge 负载、后台线程瓶颈	

6.3.5 自动化脚本示例

```
for i := 0; i < 100000; i++ {
  body := map[string]interface{}{
    "app": "auth", "trace_id": fmt.Sprintf("t-%d", i), "message": "log",
    "ts": time.Now().UnixNano() / 1e6,
  }
  jsonBody, _ := json.Marshal(body)
  http.Post("http://localhost:9000/my-pg-sink", "application/json", bytes.NewBuffer(json
}
```

6.4 异常与恢复测试

验证目标	方案	期望指标
Vector 重启/崩溃	高负载写入时重启 Vector，观察数据补写	无重复或漏写
PG/CH 停机	短暂下线数据库，观察 Vector 缓冲	自动重试，恢复后数据一致
网络断连恢复	使用 netem 模拟断网，检查重连	不丢失、不时序异常

6.5 可观测性与告警

验证目标	方案	期望指标
指标完备性	检查 Vector、PG、CH、OS 指标	覆盖率 > 95%
告警规则有效性	模拟 CPU/延迟异常触发告警	告警及时
日志追踪	通过集中日志跟踪关键事件	故障可快速定位

6.6 验证成功标准

维度	成功判定
写入链路	≥ 95% 写入无丢失，ACK latency 稳定
数据正确性	PG 与 CH 数据能联查对齐，时间戳正确
系统资源	单节点 PG/CH/Vector CPU < 50%，无 OOM
查询性能	常用 Trace 查询 <500ms
异常处理	标签异常可自动 fallback 或告警记录

6.7 可选增强场景

- 断点恢复测试：模拟 Vector 中断、PG WAL 积压后的补写能力
- 导入真实日志样本：使用历史 prod 日志替代模拟器
- 滚动升级兼容性：升级 Vector/CH 时验证数据路径稳定性
- 跨地域部署：验证跨区复制对写入与查询的影响

7. 结语

通过 PostgreSQL + ClickHouse 的分层写入方案，可以在保证实时查询能力的同时实现高吞吐归档和聚合分析，为可观测性平台提供灵活、可扩展的存储基础。

生产级监控配置方案（Linux 裸机）

1. 目标

- 系统级指标（CPU/MEM/Load/磁盘/网卡）→ **node_exporter**
- 进程级指标（nginx/redis/postgres/xcontrol-server 等）→ **process-exporter**
- 本地历史保底（即使 Prometheus 挂掉也能事后回放）→ **atop + sysstat**
- 安全/加固：统一安装目录、专用用户、systemd 管理、端口限制、日志留存策略

2. 目录 & 用户规划

```
# 安装目录
/opt/metrics-agent/
    node_exporter
    process_exporter
    process_exporter.yml

# 日志目录
/var/log/atop/
/var/log/sysstat/

# 专用用户
useradd --system --no-create-home --shell /usr/sbin/nologin metrics
```

3. Node Exporter（系统级）

安装

```
cd /opt/metrics-agent
curl -L https://github.com/prometheus/node_exporter/releases/download/v1.8.2/node_export
  | tar xz --strip-components=1 -C /opt/metrics-agent --wildcards '*/node_exporter'
```

systemd 单元 `/etc/systemd/system/node-exporter.service`

```
[Unit]
Description=Prometheus Node Exporter
After=network.target

[Service]
User=metrics
Group=metrics
ExecStart=/opt/metrics-agent/node_exporter \
  --web.listen-address=127.0.0.1:9100 \
  --collector.processes \
  --collector.filesystem.ignored-mount-points="^/(sys|proc|dev|run|var/lib/docker/.+)($|
Restart=always

[Install]
WantedBy=multi-user.target
```

> 绑定 127.0.0.1，避免直接暴露公网，建议 Prometheus 通过内网采集或加反代。

4. Process Exporter（进程级）

安装

```
cd /opt/metrics-agent
curl -L https://github.com/ncabatoff/process-exporter/releases/download/v0.7.10/process-
  | tar xz --strip-components=1 -C /opt/metrics-agent --wildcards '*/process-exporter'
```

配置文件 /opt/metrics-agent/process_exporter.yml

```
process_names:
  - name: "nginx"
    exe: ["^/usr/local/openresty/nginx/sbin/nginx$"]
    comm: ["^nginx$"]
    cmdline: ["nginx"]

  - name: "redis"
    exe: [".*/redis-server$"]
    comm: ["^redis-server$"]
    cmdline: ["redis-server"]

  - name: "postgres"
    exe: [".*/postgres$", ".*/postmaster$"]
    comm: ["^postgres$", "^postmaster$"]
    cmdline: ["postgres", "postmaster"]

  - name: "xcontrol-server"
    exe: ["^/usr/bin/xcontrol-server$"]
    comm: ["^xcontrol-server$"]
    cmdline: ["xcontrol-server"]

  - name: "other"
    cmdline: [".+"]
```

systemd 单元 /etc/systemd/system/process-exporter.service

```
[Unit]
Description=Prometheus Process Exporter
After=network.target

[Service]
User=metrics
Group=metrics
ExecStart=/opt/metrics-agent/process-exporter \
  --config.path=/opt/metrics-agent/process_exporter.yml \
```

```
  --web.listen-address=127.0.0.1:9256
Restart=always

[Install]
WantedBy=multi-user.target
```

5. 本地保底（atop + sysstat）

```
apt-get update
apt-get install -y atop sysstat

# sysstat: 每分钟采样，保留 30 天
sed -ri 's/^ENABLED=.*/ENABLED="true"/' /etc/default/sysstat
sed -ri 's/^HISTORY=.*/HISTORY=30/' /etc/default/sysstat
echo '* * * * * root sa1 60 1' > /etc/cron.d/sysstat
systemctl enable --now sysstat

# atop: 每分钟采样，保留 30 天
sed -ri 's/^LOGINTERVAL=.*/LOGINTERVAL=60/' /etc/default/atop || true
sed -ri 's/^LOGGENERATIONS=.*/LOGGENERATIONS=30/' /etc/default/atop || true
systemctl enable --now atop
```

回放命令

```
atop -r /var/log/atop/atop_$(date +%Y%m%d)
sar -u -f /var/log/sysstat/sa16   # 16 日的 CPU 历史
```

6. Prometheus 抓取配置

```
scrape_configs:
  - job_name: 'node'
    static_configs:
      - targets: ['<vm-ip>:9100']

  - job_name: 'process'
    static_configs:
      - targets: ['<vm-ip>:9256']
```

7. Grafana 面板

- 系统总览：CPU、MEM、Load、磁盘、网络 → node_exporter
- 进程分组：nginx/redis/postgres/xcontrol-server CPU/MEM/IO → process-exporter
- 告警规则：
  - 系统 CPU > 80% 且持续 5m

- 内存可用 < 15%
- 单进程组 CPU > 50% 且 Load 高
- 磁盘 IO > 80% 饿和度

8. 加固要点

- 安全：exporter 绑定 127.0.0.1，Prometheus 从内网/隧道采集；外网暴露必须加 Nginx 反代 + BasicAuth/TLS
- 资源开销：node_exporter / process-exporter 常驻 <50MB 内存，CPU 几乎 0
- atop/sysstat 采样瞬时 <2% CPU，月日志 <200MB
- 高可用：Prometheus 可加远程存储（Thanos/VM）做持久化，atop+sysstat 本地兜底
- 分组精简：process-exporter 只分关键服务，避免标签爆炸

 最终效果：- 在线：Prometheus + Grafana → 时序趋势 & 告警 - 离线/断联：atop/sysstat → 回放事后诊断 - 可扩展：如需应用级指标，可加 redis_exporter、postgres_exporter、nginx-lua-prometheus

Vector 统一采集架构与 DeepFlow Agent 对比

1. 架构概览

Vector 可以作为统一的采集出口，将系统、进程、网络和日志数据汇聚后再转发到不同的可观测性后端。

示例链路：

```
node_exporter
process_exporter
DeepFlow Agent    Vector Agent    Loki
journald/syslog                      Prometheus Remote Write
                                      Tempo
```

核心设计

- **Sources**：`prometheus_scrape` 用于拉取 node_exporter、process-exporter 指标；`journald/file` 采集系统与 DeepFlow 日志。
- **Transforms**：`remap` 统一标签，如把 `instance` 改为 `host`，补充业务标签。
- **Sinks**：同时写入 Prometheus 兼容数据库、Loki、Tempo 等多种后端，支持 TLS 与鉴权。

2. 配置拆分与样例

Vector 支持将配置拆分为多个文件，主配置通过 `includes` 统一加载：

```
/etc/vector/
   vector.yaml
   sources/
       node_exporter.yaml
       process_exporter.yaml
       journald.yaml
   sinks/
       prometheus.yaml
       loki.yaml
   transforms/
        tags.yaml
```

主配置 vector.yaml

```yaml
data_dir: /var/lib/vector
includes:
  - /etc/vector/sources/*.yaml
  - /etc/vector/sinks/*.yaml
  - /etc/vector/transforms/*.yaml
```

示例 source/sink 片段

```yaml
# sources/node_exporter.yaml
sources:
  node_exporter:
    type: prometheus_scrape
    endpoints: ["http://localhost:9100/metrics"]
    scrape_interval_secs: 15

# sinks/prometheus.yaml
sinks:
  prometheus_out:
    type: prometheus_remote_write
    inputs: ["add_tags"]
    endpoint: "http://vm.example.com/api/v1/write"
```

拆分后可独立修改某个采集器配置，便于模块化维护和热更新。

3. 高负载下的可靠性机制

Vector 在高负载场景中通过多层保护减少数据丢失：

机制	说明
背压控制	当下游拥塞时暂停上游读取，防止爆仓
缓冲策略	支持内存或磁盘缓冲，`when_full = "block"` 默认阻塞而非丢弃

机制	说明
重试与限流	对 Loki、Prometheus 等 sink 内置重试与 backoff，支持 `rate_limit`
降级策略	`drop_on_full = false`，尽量保留数据
自监控指标	`/metrics` 暴露 `events_dropped_total`、`buffer_overflows_total` 等指标供告警

4. 采集器能力对比

特性/组件	node_exporter	process-exporter	DeepFlow Agent	Vector Agent
安装体积	<20MB	<20MB	≈70–100MB	≈70MB
资源占用	极低	低	中等（依赖 eBPF）	低 ~ 中，可限内存
采集维度	主机资源	进程资源	网络四/七层	指标、日志、链路
输出能力	Prometheus	Prometheus	DeepFlow Collector	Prometheus、Loki、Tempo 等
可靠性	无背压	无背压	重试有限	背压 + 缓冲 + 重试

Vector 在可靠性、可扩展性与自观测能力上相较 DeepFlow Agent 更为成熟，后者适合作为网络事件探针。

5. 渐进式部署路线

1. **平台自身稳定性**：部署 node_exporter、process-exporter 与 Vector，先掌握主机与关键进程状态。
2. **网络可观测**：引入 DeepFlow Agent，分析 L4/L7 流量，后端可用 ClickHouse + Grafana 展示。
3. **日志聚合**：Vector 同步写入 Loki 或其他日志系统，辅助故障排查与审计。
4. **链路追踪**：启用 Vector → Tempo 或 OTLP，将 DeepFlow 产出的 trace 信息整合展示。
5. **示例 process-exporter 配置**：`yaml process_names: - name: "deepflow-agent" cmdline: ["deepflow-agent"] - name: "clickhouse" cmdline: ["clickhouse-server"]`

6. Server 端存储选型

为了在后端构建统一、弹性、低成本的可观测平台，可按数据类型选用以下组件：

Metrics：VictoriaMetrics

- **优势**：单机即可实现百万点/秒写入，兼容 PromQL，支持自动压缩并将历史数据归档到 S3/GCS。
- **采集建议**：使用 `vmagent` 或 Vector 远程写入，也可通过 otel-collector 接入 OpenTelemetry Metrics。

Logs：Loki

- **优势**：基于标签的索引方式，运行成本低；原生支持 Vector、Fluent Bit、Promtail 等采集器，兼容结构化与非结构化日志。
- **采集建议**：通过 Vector 统一收集 journald 与文件日志，按主机和应用维度设置标签，可配置 S3 归档策略。

Traces：Tempo

- **优势**：兼容 OTLP、Zipkin、Jaeger 协议；依赖对象存储实现冷热分层，资源占用极低，并能与 logs/metrics 自动关联。
- **采集建议**：应用直接使用 OpenTelemetry SDK，或经由 otel-collector、Vector 输出 OTLP 数据。

结构化数据：PostgreSQL（可选）

- **用途**：存储事件、审计、成本等业务数据，配合 Grafana 表格/统计面板做结构化分析。
- **扩展**：
  - TimescaleDB：增强时间序列查询能力。
  - pgvector：提供向量检索，可结合 AI 做相似事件分析。
  - postgres_fdw：整合多个 PostgreSQL 数据源。

推荐部署组合

数据类型	后端存储	采集/转发组件	说明
Metrics	VictoriaMetrics	vmagent / Vector	远程写入并归档到对象存储
Logs	Loki	Vector	按小时本地留存，按日归档到 S3
Traces	Tempo	otel-collector / Vector	可按需采样与压缩
结构化数据	PostgreSQL (+Timescale)	应用或 ETL	用于业务事件与分析

组合优势

- **统一采集**：Vector 汇聚日志、指标、链路三类数据，再转发至不同后端。

- **安全加密**：各组件均支持 TLS 与 Token 鉴权。
- **高性能、低成本**：组件均为原生二进制部署，可使用对象存储做冷数据归档。
- **Grafana 统一展示**：VictoriaMetrics、Loki、Tempo、PostgreSQL 均为官方数据源，可在 Grafana 中集中呈现。

7. 总结

通过 Vector 构建统一采集出口，可在保证稳定性的同时整合指标、日志、链路与结构化数据。DeepFlow Agent 专注网络可观测，结合推荐的后端存储组件，可形成覆盖系统到业务的完整可观测体系。

技术选型对比

边缘采集对比

网关对比

存储对比

分析层对比

总体架构设计

边缘采集与缓冲

区域网关与扇出

存储与分析层

双链路（近线检索 + Kafka 回放）

多区域架构与区域内拓扑

区域内拓扑（单区示例）

多区域部署模式（主区 + 从区）

联邦查询与统一入口

端到端可靠传输设计（At-least-once 语义）

多级持久化链（Vector → OTel → Kafka → OpenObserve）

扇出与去重策略（event_id + UPSERT）

SRE 可观测与演练方法

PostgreSQL 二级分析域

- 时序分析（TimescaleDB 连续聚合）
- 向量检索（pgvector）
- 图查询（Apache AGE）
- 高基数与 TopK 分析（HLL/Toolkit）

IT 咖啡馆｜全栈可观测数据库设计

> 线 I/O 面交给 OpenObserve（OO），治理/知识面沉到 PostgreSQL 栈（Timescale + AGE + pgvector）。保留明细在 OO，只把聚合、摘要、索引与证据链写回 PG：既省钱、又好查、也好演进。

摘要

本文落地一套「全栈可观测」数据库设计：明细进 OO，PG 仅存 12 张核心表（维度、定位符、指标 1m、服务调用 5m、日志指纹/计数、拓扑时态、知识库、事件/证据），并用 AGE 维护“当前服务级调用图”。给出可直接执行的 DDL、保留与压缩策略、典型查询与接入流程。

设计目标

- **最小化**：仅保 1m 指标聚合、5m 服务调用/日志计数；图仅保“当前 10 分钟活跃边”。
- **可扩展**：OO 扛明细与重计算；PG 通过加列/并行表横向扩展。
- **可解释**：事件只挂索引与证据链；需要原文用 oo_locator 反查 OO。

分层原则

- **线 I/O 面（近线）**：OO（对象存 + 列式 + 低成本长期留存）。
- **治理/知识面**：PG（Timescale 连续聚合/压缩、AGE 图、pgvector 检索）。

数据模型总览（12 表 + AGE）

维度（2）：`dim_tenant`、`dim_resource`

定位符（1）：`oo_locator`（对象存路径 + 时间窗 + 查询 hint）

聚合（3）：`metric_1m`、`service_call_5m`、`log_pattern_5m`；

指纹（1）：`log_pattern`

拓扑（1）：`topo_edge_time`（边 + 有效期）

知识（2）：`kb_doc`、`kb_chunk`（HNSW 向量索引）

事件（2）：`event_envelope`、`evidence_link`

图（AGE）：仅维护“服务级调用图”的活跃子图（10 分钟窗口）。

> 注：日志明细、trace span、指标原始点位 **不入 PG**，统一留在 OO。

表设计与策略（逐表解读）

1）维度

- `dim_tenant`：租户/域；用 `code` 做业务唯一键。
- `dim_resource`：统一资源 URN（如 `urn:k8s:svc:ns/name`），含环境/区域/标签。

策略：

- 为 `type`、`labels` 建索引（B-Tree + GIN），支撑过滤与聚合。
- 跨区/多租户时，尽量靠 `tenant_id` 分片路由。

2）OO 定位符

- `oo_locator`：只存 **对象存路径、时间窗、查询 hint**，作为“回查线索”。

策略：

- 与聚合表通过 `sample_ref` 关联，形成“摘要 → 原文”的证据链。

3）指标聚合（1 分钟）

- `metric_1m`：按资源、指标名保 `avg/max/p95` 常用统计，Timescale Hypertable。

策略：

- **压缩**：7 天后压缩；
- **留存**：180 天；
- **索引**：`(resource_id, metric, bucket desc)` + `labels` GIN。

4）服务级调用聚合（5 分钟）

- `service_call_5m`：A→B 的 `rps/err_rate/p50/p95`，主键含窗口/租户/边端点。

用途：

- 报表/拓扑；
- 刷新 AGE “活跃调用图”。

策略：留存 365 天，适配容量评估与 SLO 分析。

5）日志指纹/计数（5 分钟）

- `log_pattern`：指纹模板库（去重 + 样例 + 严重级 + 抽取 schema），`pg_trgm` 支持模糊。
- `log_pattern_5m`：每 5 分钟的计数与错误计数；**不存明细**。

策略：

- 借 `sample_ref` 反查 OO 原文；
- 留存 180 天。

6）拓扑时态

- `topo_edge_time`：边 + 有效期 `tstzrange`，用于追溯“某时刻的拓扑”。

策略：

- `btree_gist` + GIST 范围索引，支持按时间窗口查询。

7）知识库 / 向量

- `kb_doc`：文档元信息（来源、标题、URL、元数据）。
- `kb_chunk`：切片 + 向量（`vector(1536)`），`USING hnsw` 建立近似检索。

策略：

- 以 `tenant_id`/资源/时间 作为结构化过滤条件，向量召回后再二次排序。

8）事件与证据链

- `event_envelope`：事件封套（时间/资源/严重级/类型/摘要/标签/指纹）。
- `evidence_link`：**多态引用**到 PG 记录或 `oo_locator`，串起证据链。

关键修正：

- PostgreSQL **主键不允许表达式**，因此 `evidence_link` 使用 **自增主键 + 唯一索引（基于 ref_pg 的 hash + ref_oo）**实现“幂等去重”。详见附录 DDL。

图：只维护“当前 10 分钟活跃调用图”

- 从 `service_call_5m` 挑选近 10 分钟活跃边（按租户/边聚合），用 AGE 的 `MERGE` 同步：
  - 点：`(:Resource {tenant_id, resource_id})`

  - 边：`[:CALLS {last_seen, rps, err_rate, p95}]`
- 图用于 **k-hop、最短路、影响面**，而不是长期时序（时序仍在 `service_call_5m` & `topo_edge_time`）。

最小化接入流程

写入：

1. 明细 → OO（logs/traces/metrics）；
2. 近线作业在 OO 内计算聚合：写入 `metric_1m`、`service_call_5m`、`log_pattern_5m`；
3. 同步 `oo_locator`（对象路径 + 时间窗 + hint）；
4. 用聚合结果刷新 AGE 活跃调用图。

读取：

- 近线排障：先看 `metric_1m` / `log_pattern_5m` / `service_call_5m`；
- 影响面/最短路：走 AGE；
- 需要原文：根据 `sample_ref` / `evidence_link.ref_oo` 反查 OO。

默认留存：

- `metric_1m`：180 天；`service_call_5m`：365 天；`log_pattern_5m`：180 天；
- OO 明细：30–180 天或更久（对象存便宜）。

典型查询

1）过去 30 分钟某服务下游 Top-5 慢/错依赖

```
SELECT d.urn AS dst, avg(p95_ms) AS p95, avg(err_rate) AS err
FROM service_call_5m c
JOIN dim_resource s ON s.resource_id = c.src_resource_id
JOIN dim_resource d ON d.resource_id = c.dst_resource_id
WHERE s.urn = $service_urn
  AND c.bucket >= now() - interval '30 minutes'
GROUP BY d.urn
ORDER BY p95 DESC, err DESC
LIMIT 5;
```

2）按事件拉取证据（含 OO 回查线索）

```
SELECT e.event_id, e.title, e.detected_at, r.urn,
       ev.dim, ev.ref_pg, ol.bucket, ol.object_key, ol.query_hint
FROM event_envelope e
JOIN dim_resource r ON r.resource_id = e.resource_id
LEFT JOIN evidence_link ev ON ev.event_id = e.event_id
```

```
LEFT JOIN oo_locator   ol ON ol.id = ev.ref_oo
WHERE e.event_id = $1;
```

3）恢复“某一时刻”的服务级拓扑

```
SELECT s.urn AS src, d.urn AS dst, t.relation
FROM topo_edge_time t
JOIN dim_resource s ON s.resource_id = t.src_resource_id
JOIN dim_resource d ON d.resource_id = t.dst_resource_id
WHERE t.tenant_id = $tenant
  AND t.valid @> $timestamp::timestamptz;
```

与「全塞 PG」的对比

- **成本**：明细不上 PG，写入抖动与存储爆炸显著降低；OO 对象存更省。
- **性能**：常用报表/排障命中 1m/5m 聚合；图遍历走 AGE；需要原文精准回查。
- **演进**：加维度/指标只需 **加列/新表**；图/向量索引也可独立扩展。

运维与演进建议

- **Timescale**：启用压缩 + 连续聚合（如需），按 7D/30D 切分 Chunk；
- **索引**：B-Tree（时间倒序复合）+ GIN（labels）+ HNSW（向量）；
- **多租户**：路由以 `tenant_id` 优先；必要时做 schema-per-tenant；
- **幂等**：聚合写入用 `ON CONFLICT`；`evidence_link` 采用哈希唯一索引去重；
- **数据治理**：严格区分“摘要/索引/证据”与“明细”；
- **备份**：PG 常规备份；OO 走对象存生命周期策略；
- **安全**：表级 RLS（Row Level Security）可选；`labels` 与 `metadata` 控敏字段打码。

附录 A：可直接执行的 DDL（含扩展/索引/策略）

```
-- 0）扩展（一次性）
CREATE EXTENSION IF NOT EXISTS timescaledb;
CREATE EXTENSION IF NOT EXISTS pg_trgm;
CREATE EXTENSION IF NOT EXISTS pgcrypto;   -- gen_random_uuid
CREATE EXTENSION IF NOT EXISTS vector;
CREATE EXTENSION IF NOT EXISTS age;        -- 图扩展（服务级调用图）
LOAD 'age';
CREATE EXTENSION IF NOT EXISTS btree_gist; -- 用于时态拓扑范围索引

-- 1）维度（2）
```

```sql
CREATE TABLE dim_tenant (
  tenant_id   BIGSERIAL PRIMARY KEY,
  code        TEXT UNIQUE NOT NULL,
  name        TEXT NOT NULL,
  labels      JSONB DEFAULT '{}'::jsonb,
  created_at  TIMESTAMPTZ DEFAULT now()
);

CREATE TABLE dim_resource (
  resource_id BIGSERIAL PRIMARY KEY,
  tenant_id   BIGINT REFERENCES dim_tenant(tenant_id),
  urn         TEXT UNIQUE NOT NULL,
  type        TEXT NOT NULL,
  name        TEXT NOT NULL,
  env         TEXT,
  region      TEXT,
  zone        TEXT,
  labels      JSONB DEFAULT '{}'::jsonb,
  created_at  TIMESTAMPTZ DEFAULT now()
);
CREATE INDEX idx_res_type ON dim_resource(type);
CREATE INDEX idx_res_labels_gin ON dim_resource USING GIN(labels);

-- 2) OO 定位符（1）
CREATE TABLE oo_locator (
  id          BIGSERIAL PRIMARY KEY,
  tenant_id   BIGINT REFERENCES dim_tenant(tenant_id),
  dataset     TEXT NOT NULL,              -- logs / traces / metrics
  bucket      TEXT NOT NULL,
  object_key  TEXT NOT NULL,
  t_from      TIMESTAMPTZ NOT NULL,
  t_to        TIMESTAMPTZ NOT NULL,
  query_hint  TEXT,
  attributes  JSONB DEFAULT '{}'::jsonb
);
CREATE INDEX idx_oo_time ON oo_locator(dataset, t_from, t_to);

-- 3) 指标聚合（1m，Hypertable）
CREATE TABLE metric_1m (
  bucket       TIMESTAMPTZ NOT NULL,
  tenant_id    BIGINT REFERENCES dim_tenant(tenant_id),
  resource_id  BIGINT REFERENCES dim_resource(resource_id),
  metric       TEXT NOT NULL,
  avg_val      DOUBLE PRECISION,
  max_val      DOUBLE PRECISION,
```

```
  p95_val      DOUBLE PRECISION,
  labels       JSONB DEFAULT '{}'::jsonb
);
SELECT create_hypertable('metric_1m','bucket',chunk_time_interval => interval '7 days');
CREATE INDEX idx_metric_key ON metric_1m(resource_id, metric, bucket DESC);
CREATE INDEX idx_metric_labels ON metric_1m USING GIN(labels);
ALTER TABLE metric_1m SET (
  timescaledb.compress,
  timescaledb.compress_segmentby = 'resource_id, metric',
  timescaledb.compress_orderby   = 'bucket'
);
SELECT add_compression_policy('metric_1m', INTERVAL '7 days');
SELECT add_retention_policy   ('metric_1m', INTERVAL '180 days');

-- 4) 服务级调用聚合（5m，Hypertable）
CREATE TABLE service_call_5m (
  bucket          TIMESTAMPTZ NOT NULL,
  tenant_id       BIGINT REFERENCES dim_tenant(tenant_id),
  src_resource_id BIGINT REFERENCES dim_resource(resource_id),
  dst_resource_id BIGINT REFERENCES dim_resource(resource_id),
  rps             DOUBLE PRECISION,
  err_rate        DOUBLE PRECISION,
  p50_ms          DOUBLE PRECISION,
  p95_ms          DOUBLE PRECISION,
  sample_ref      BIGINT REFERENCES oo_locator(id),
  PRIMARY KEY(bucket, tenant_id, src_resource_id, dst_resource_id)
);
SELECT create_hypertable('service_call_5m','bucket',chunk_time_interval => interval '30
CREATE INDEX idx_call_src_dst ON service_call_5m(src_resource_id, dst_resource_id, bucke
SELECT add_retention_policy('service_call_5m', INTERVAL '365 days');

-- 5) 日志指纹（去重）+ 5m 计数（2）
CREATE TABLE log_pattern (
  fingerprint_id  BIGSERIAL PRIMARY KEY,
  tenant_id       BIGINT REFERENCES dim_tenant(tenant_id),
  pattern         TEXT NOT NULL,
  sample_message  TEXT,
  severity        TEXT,
  attrs_schema    JSONB DEFAULT '{}'::jsonb,
  first_seen      TIMESTAMPTZ,
  last_seen       TIMESTAMPTZ
);
CREATE INDEX idx_logpat_tenant ON log_pattern(tenant_id);
CREATE INDEX idx_logpat_pattern_trgm ON log_pattern USING GIN (pattern gin_trgm_ops);
```

```
CREATE TABLE log_pattern_5m (
  bucket         TIMESTAMPTZ NOT NULL,
  tenant_id      BIGINT REFERENCES dim_tenant(tenant_id),
  resource_id    BIGINT REFERENCES dim_resource(resource_id),
  fingerprint_id BIGINT REFERENCES log_pattern(fingerprint_id),
  count_total    BIGINT NOT NULL,
  count_error    BIGINT NOT NULL DEFAULT 0,
  sample_ref     BIGINT REFERENCES oo_locator(id),
  PRIMARY KEY(bucket, tenant_id, resource_id, fingerprint_id)
);
SELECT create_hypertable('log_pattern_5m','bucket',chunk_time_interval => interval '30 d
CREATE INDEX idx_logpat5m_res ON log_pattern_5m(resource_id, bucket DESC);
SELECT add_retention_policy('log_pattern_5m', INTERVAL '180 days');

-- 6) 拓扑时态（1）
CREATE TABLE topo_edge_time (
  tenant_id       BIGINT REFERENCES dim_tenant(tenant_id),
  src_resource_id BIGINT REFERENCES dim_resource(resource_id),
  dst_resource_id BIGINT REFERENCES dim_resource(resource_id),
  relation        TEXT NOT NULL,
  valid           tstzrange NOT NULL,  -- [from, to)
  props           JSONB DEFAULT '{}'::jsonb,
  PRIMARY KEY(tenant_id, src_resource_id, dst_resource_id, relation, valid)
);
CREATE INDEX idx_topo_valid ON topo_edge_time USING GIST (tenant_id, src_resource_id, ds

-- 7) 知识库 / 向量（2）
CREATE TABLE kb_doc (
  doc_id     BIGSERIAL PRIMARY KEY,
  tenant_id  BIGINT REFERENCES dim_tenant(tenant_id),
  source     TEXT,
  title      TEXT,
  url        TEXT,
  metadata   JSONB DEFAULT '{}'::jsonb,
  created_at TIMESTAMPTZ DEFAULT now()
);

CREATE TABLE kb_chunk (
  chunk_id   BIGSERIAL PRIMARY KEY,
  doc_id     BIGINT REFERENCES kb_doc(doc_id) ON DELETE CASCADE,
  chunk_idx  INT NOT NULL,
  content    TEXT NOT NULL,
  embedding  vector(1536) NOT NULL,
  metadata   JSONB DEFAULT '{}'::jsonb
);
```

```
CREATE INDEX idx_kb_chunk_doc ON kb_chunk(doc_id, chunk_idx);
CREATE INDEX idx_kb_chunk_meta ON kb_chunk USING GIN(metadata);
CREATE INDEX idx_kb_vec_hnsw ON kb_chunk USING hnsw (embedding vector_l2_ops);

-- 8) 事件 & 证据链 (2)
DO $$ BEGIN
  IF NOT EXISTS (SELECT 1 FROM pg_type WHERE typname = 'severity') THEN
    CREATE TYPE severity AS ENUM ('TRACE','DEBUG','INFO','WARN','ERROR','FATAL');
  END IF;
END $$;

CREATE TABLE event_envelope (
  event_id     UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  detected_at  TIMESTAMPTZ NOT NULL DEFAULT now(),
  tenant_id    BIGINT REFERENCES dim_tenant(tenant_id),
  resource_id  BIGINT REFERENCES dim_resource(resource_id),
  severity     severity NOT NULL,
  kind         TEXT NOT NULL,      -- anomaly/slo_violation/deploy/incident/...
  title        TEXT,
  summary      TEXT,
  labels       JSONB DEFAULT '{}'::jsonb,
  fingerprints JSONB DEFAULT '{}'::jsonb
);
CREATE INDEX idx_event_time ON event_envelope(tenant_id, detected_at DESC);

-- 注意：主键不能用表达式，改为自增 + 唯一索引 (ref_pg 哈希 + ref_oo)
CREATE TABLE evidence_link (
  evidence_id BIGSERIAL PRIMARY KEY,
  event_id    UUID NOT NULL REFERENCES event_envelope(event_id) ON DELETE CASCADE,
  dim         TEXT NOT NULL,  -- metric/log/trace/topo/kb
  ref_pg      JSONB,          -- {"table":"...","keys":{...}}
  ref_oo      BIGINT REFERENCES oo_locator(id),
  note        TEXT,
  ref_pg_hash TEXT GENERATED ALWAYS AS (md5(coalesce(ref_pg::text, ''))) STORED
);
CREATE UNIQUE INDEX ux_evidence_unique
  ON evidence_link(event_id, dim, ref_pg_hash, coalesce(ref_oo, 0));
CREATE INDEX idx_evidence_event ON evidence_link(event_id);

-- AGE 图：初始化 (一次)
SELECT * FROM create_graph('ops');
SELECT * FROM create_vlabel('ops', 'Resource');
SELECT * FROM create_elabel('ops', 'CALLS');
```

附录 B：AGE 活跃调用图刷新（示例）

```
WITH active AS (
  SELECT tenant_id, src_resource_id, dst_resource_id,
         max(bucket) AS last_seen,
         avg(rps) AS rps, avg(err_rate) AS err_rate, avg(p95_ms) AS p95_ms
  FROM service_call_5m
  WHERE bucket >= now() - interval '10 minutes'
  GROUP BY 1,2,3
)
SELECT * FROM cypher('ops', $$
  UNWIND $rows AS row
  MERGE (s:Resource {tenant_id: row.tenant_id, resource_id: row.src})
  MERGE (d:Resource {tenant_id: row.tenant_id, resource_id: row.dst})
  MERGE (s)-[e:CALLS]->(d)
  ON CREATE SET e.first_seen = row.last_seen
  SET e.last_seen = row.last_seen, e.rps = row.rps, e.err_rate = row.err_rate, e.p95 = r
  RETURN 1
$$) AS (ok int)
PARAMS (rows := (
  SELECT json_agg(json_build_object(
    'tenant_id', tenant_id, 'src', src_resource_id, 'dst', dst_resource_id,
    'last_seen', last_seen, 'rps', rps, 'err_rate', err_rate, 'p95', p95_ms))
  FROM active));
```

收尾

这套「OO 扛明细 + PG 扛治理」的极简方案能把近线排障、影响面分析与知识复用串成闭环：

- **快**：常用分析直接命中 1m/5m 聚合与 AGE 图；
- **省**：明细留 OO，对象存生命周期策略友好；
- **稳**：PG 只承载“可解释的摘要与证据”，长期演进不怕重构。

多区域与容灾（Disaster Recovery, DR）

接入与路由（Anycast/GeoDNS）

跨区复制与镜像（Kafka MirrorMaker2）

阈值控制与降级策略

历史回放与灰度演练

From Monitoring History to Full-Stack Observable Data Selection

An overview of how monitoring evolved into modern full-stack observability and the key considerations when choosing diverse data sources.

1. 概述（ETL-OO2PG & AGE 活跃调用图 & 拓扑/IaC/Ansible）

```
[exporter]                    [Vector]         [OTel GW]        [OpenObserve]
NE
PE    metrics/logs   >   Vector     >   OTel GW     >        OO
DF
LG

                        （近线窗口 ETL：对齐=1m · 延迟=2m）


 IaC/Cloud         >
                                  ObserveBridge（ETL 任务）
 Ansible           >    • ETL 窗口聚合 / oo_locator
                                  • 拓扑（IaC/Ansible）
 OO 明细(OO→OB)      >    • AGE 10 分钟活跃调用图刷新


                  Postgres 套件
   PG_JSONB              PG Aggregates (Timescale)      PG Vector     AGE
 (oo_locator/events)   (metric_1m / call_5m / log_5m)  (pgvector)   Graph
```

[llm-ops-agent / 应用消费（查询/检索/推理）]

目标：在**单二进制 Go 编排器**内，完成三条主干 ETL 闭环：

- **OO2PG（近线聚合）**：从 OpenObserve（S3/API）读取 logs/metrics/traces 明细 → 聚合为 `metric_1m`、`service_call_5m`、`log_pattern_5m`，维护 `log_pattern` 指纹库与 `oo_locator` 回查线索。
- **AGE 活跃调用图**：以 `service_call_5m` 为源，刷新 AGE 图中 10 分钟**活跃服务级调用边** `(:Resource)-[:CALLS]->(:Resource)`。
- **拓扑 ETL（IaC/Ansible）**：从 Terraform/Pulumi/Cloud API 与 Ansible Playbook 抽取**结构/应用依赖**，以时态方式落入 `topo_edge_time`（开/关区间），可选同步到 AGE 作为 `:STRUCT` 边。

调度特性：**窗口对齐 + 延迟容忍 + DAG 依赖 + 幂等 Upsert + 分片多租户**。

可靠性：PG 唯一索引保证**一次性**；指数退避重试；失败熔断；事件补数回放。

---

2. 项目目录（合并版）

```
etl/
  cmd/etl/                    # 二进制入口/CLI
    main.go
  pkg/
    scheduler/                # 调度器（窗口计算/派发）
    runner/                   # Worker 池 + 执行/重试/回调
    registry/                 # Job 接口 + 注册中心 + DAG
    store/                    # 状态/一次性保证/简易队列（PG）
    window/                   # 时间对齐/窗口工具
    events/                   # 事件入口（HTTP/CloudEvents 风格）
    oo/                       # OO 读取（S3 客户端 + 查询 API 适配）
    agg/                      # 聚合器（指标 1m / 调用 5m / 指纹 5m）
    patterns/                 # 日志指纹挖掘（Drain / RE2）
    iac/                      # IaC/Cloud 归一（TF/Pulumi/aws/aliyun…）
    ansible/                  # Playbook/Inventory 解析与依赖抽取
    pgw/                      # PG 写入器（COPY 批量 + 幂等 upsert）
  jobs/
    ooagg.go                  # OO → metric_1m / call_5m / log_5m / locator
    age_refresh.go            # 近 10 分钟活跃调用图刷新（依赖 ooagg）
    topo_iac.go               # IaC/Cloud → topo_edge_time（时态差分）
    topo_ansible.go           # Ansible → topo_edge_time（时态差分）
  sql/
    age_refresh.sql           # AGE 刷新 SQL
  configs/
```

```
etl.yaml                    # 调度/并发/延迟 配置
```

---

3. 表/模块总览（合并表）

下面是**模块—接口—数据源/目标—窗口/键—幂等**的一览表（开发/联调用）。

模块	API/入口	SRC（输入）	DEST（输出）	窗口/键	幂等/唯一约束	备注
pkg/oo	Stream(ctx, tenant, w, fn)	OO（S3 分区或查询 API）logs/metrics/traces	回调 oo.Record	w=[From,To)	—	统一时间/URN；并发读取
pkg/agg	Feed(rec) / Drain()	oo.Record	Metrics1m / Calls5m / LogPatterns5m / PatternsUpsert / Locators	1m/5m	—	内存桶聚合、指纹抽取
pkg/pgw	Flush(ctx, tenant, w, out)	聚合结果 out	metric_1m、service_call_5m、log_pattern_5m、log_pattern、oo_locator、dim_resource	1m/5m 主键	ON CONFLICT DO UPDATE；oo_locator 唯一	PG 批量 COPY + Upsert
jobs/ooagg	Run(ctx, tenant, w)	pkg/oo → pkg/agg	pkg/pgw.Flush	Align=1m；Delay=2m	由目标表主键保证	成功后触发 age-refresh
sql/age_refresh.sql	psql -v scope=‘ops’ …	service_call_5m 近 10 分钟	AGE 图 CALLS 边	10 分钟	MERGE 唯一匹配	e.last_seen/rps/err/p9
jobs/age_refresh	Run(ctx, tenant, w)	service_call_5m	执行 SQL	Align=5m	—	After()=“oo-agg”
pkg/iac	Discover(ctx, tenant)	TF/Pulumi/Ansible API	边集合 []Edge	—	—	构造 URN、relation
pkg/ansible	ExtractDeps(ctx, tenant)	inventory/group_vars/roles	边集合 []Edge	—	—	解析 upstream/连接串

模块	API/入口	SRC（输入）	DEST（输出）	窗口/键	幂等/唯一约束	备注
pkg/pgw	UpsertTopoEdges(ctx, tenant, edges)	边	topo_edge_time（时态）	valid tstzrange	PK(tenant,src,dst,rel,valid)	差分时态区间
jobs/topo_iac	Run(ctx, tenant, w)	pkg/iac.Discover	topo_edge_time	Align=15m	—	结构拓扑
jobs/topo_ansible	Run(ctx, tenant, w)	pkg/ansible.Extract	topo_edge_time	Align=1h	—	应用依赖拓扑
pkg/events	/events/enqueue	CloudEvents/HTTP	etl_job_run 状态置 queued	任意窗口	ux_job_once	手动补数/回放
pkg/store	EnqueueOnce/Mark*	—	etl_job_run	job/tenant/win	ux_job_once	一次性保证/队列
pkg/scheduler	Tick()	dim_tenant & etl_job_run	入队窗口	Align/Delay/Lookback	—	动态加载配置

相关 PG 表（12 + ETL 元数据）

- 维度：`dim_tenant`、`dim_resource`
- 定位符：`oo_locator`
- 聚合：`metric_1m`（1m）、`service_call_5m`（5m）、`log_pattern_5m`（5m）
- 指纹：`log_pattern`
- 拓扑时态：`topo_edge_time`
- 知识：`kb_doc`、`kb_chunk`（向量）
- 事件：`event_envelope`、`evidence_link`
- ETL 元数据：`etl_job_run`、`etl_job_circuit`
- AGE：`ops` 图（`Resource`、`CALLS`）

---

4. 设计要点与边界

- **窗口推进**：`upper = floor(now - Delay, Align)`，从上次成功窗口末尾或 `initial_lookback` 起步。
- **幂等写入**：所有聚合表/计数表以主键覆盖；`log_pattern` 以 pattern hash 或唯一键 upsert；`oo_locator` 唯一组合避免重复。
- **资源归一（URN）**：标准化 `urn:k8s:svc:<ns>/<name>` / `urn:host:<host>` / `urn:db:postgres:<cluster>/<db>`，并缓存 `resource_id`。
- **AGE 图只保活跃 10 分钟**：长期时序仍在 `service_call_5m` 与 `topo_edge_time`。

- **拓扑差分**：用 tstzrange 开/关区间维护时态；当边消失时关闭上次开区间。

---

5. Codex Tasks（落地清单）

每个任务包含：**name / 描述 / 测试-验证**。可直接拆成 Codex 指令执行（create-or-update-files / patch），或作为 PR checklist。

T1 —完成 pkg/pgw.Flush（COPY + Upsert 批量写）

- **描述**：实现资源 upsert、oo_locator 写入并回填 sample_ref；metric_1m/service_call_5m/log_pat 批量写入，全部 ON CONFLICT DO UPDATE。
- **测试/验证**：
  - 用本地 mock（见 T3）生成 3 个窗口数据；重复跑同一窗口，行数不增加、统计覆盖一致。
  - 观察 etl_job_run 状态转为 success；冲突率 < 5%。

T2 —实现 sql/age_refresh.sql 的程序化执行

- **描述**：在 jobs/age_refresh.go 读取并执行 sql/age_refresh.sql，或用参数化 SQL 内嵌实现。
- **测试/验证**：
  - 先插入若干 service_call_5m 行，运行 age-refresh，检查 AGE 中 CALLS 边的 last_seen/rps/err_rate/p95。
  - 连续两次执行，边数量不增长；属性按平均值更新。

T3 —pkg/oo.Stream 的 MOCK 与真实 S3/API 适配

- **描述**：

- MOCK：生成固定分布的 logs/metrics/traces；支持 `--mock` 开关。
- S3：按 `dataset/yyyy=.../hh=.../mm=...` 列表对象；API：按时间窗查询。

- **测试/验证**：

- `--mock` 模式 1m 生成 5k 日志、500 指标点、1k span，端到端落库 < 3s/窗口
- 切换到 S3，验证窗口边界与对象命名正确。

T4 —pkg/agg 聚合正确性

- **描述**：实现指标 1m avg/max/p95、调用 5m rps/err/p50/p95、日志指纹 5m 计数；接入 patterns.MineTemplate。
- **测试/验证**：
  - 单测：给定样本序列，校验统计量；给定 span/log 对，校验 A→B 聚合。
  - 性能：10 万条日志 + 2 万 span，内存峰值 < 200MB。

T5 —pkg/patterns 指纹抽取（Drain/RE2）

- **描述**：实现日志模板归一；输出 fingerprint、severity；可配置忽略字段。
- **测试/验证**：
  - 对同模式不同变量的日志返回同一指纹；错误日志 count_error 正确累积。

T6 —pkg/iac.Discover + pkg/ansible.ExtractDeps

- **描述**：解析 TF/Pulumi/Cloud 与 inventory/group_vars/roles；映射 URN 与 DEPENDS_ON 边。
- **测试/验证**：
  - 构造样例仓库（或 JSON 快照），跑任务后 topo_edge_time 新增开区间；删除节点后再次运行，旧边区间关闭。

T7 — pkg/pgw.UpsertTopoEdges 时态差分

- **描述**：实现 E_now 与当前开区间 E_prev 的集合差；新增插入开区间，消失关闭区间。
- **测试/验证**：连续两轮相同输入不新增边；去掉一个边后再次运行，原边 valid 上界从无穷变为 now()。

T8 —pkg/events 事件补数接口

- **描述**：POST /events/enqueue 支持 {job, tenant_id, from, to} 回放窗口；写入 etl_job_run 为 queued 并入队。
- **测试/验证**：
  - 对已成功窗口再次 enqueue，应能覆盖写不冲突；对未来窗口入队，不会被执行（受 Delay 上界限制）。

T9 —配置加载与初始回看

- **描述**:pkg/config 已有;在 scheduler 注入 Cfg 并使用 tenants.initial_lookback.oo-agg 等。
- **测试/验证**：
  - 删除 etl_job_run 记录后启动，观察从 initial_lookback 开始补跑。

T10 —观测指标与窗口滞后

- **描述**：导出 Prometheus 指标（自定义 /metrics 或写回 OO）；记录窗口滞后。
- **测试/验证**：
  - 在稳定负载下，window_lag_seconds < Delay + Align；失败重试曲线可见。

附：典型验证 SQL

- **聚合正确性（1m 指标）**

```
SELECT metric, count(*), min(bucket), max(bucket)
FROM metric_1m
GROUP BY metric ORDER BY count(*) DESC LIMIT 10;
```

- **活跃调用图 vs 源一致性**

```
WITH active AS (
  SELECT src_resource_id, dst_resource_id
  FROM service_call_5m
  WHERE bucket >= now() - interval '10 minutes'
  GROUP BY 1,2
)
SELECT count(*) AS edges_in_source FROM active;
-- 再在 AGE 里查同样规模（CALLS 边数量），两者应近似一致（考虑租户过滤）
```

- **拓扑时态闭环**

```
-- 当前有效边
SELECT count(*) FROM topo_edge_time WHERE upper_inf(valid);
-- 历史边（已关闭）
SELECT count(*) FROM topo_edge_time WHERE NOT upper_inf(valid);
```

Typical Operations Scenarios

Illustrates common operational situations that highlight the need for effective monitoring across systems and services.

1) 发布与环境

- **灰度发布（Canary/Blue-Green）**触发：新版本可用 → 动作：按权重放量/自动回滚 → 校验：`service_call_5m` 的 p95/err_rate 门槛。
- **特性开关（Feature Flag）渐进开启** 触发：验收通过 → 动作：按租户/地域打开 → 校验：关键路径转化率、错误率对比。
- **依赖版本对齐（SDK/Sidecar/Agent）**触发：安全公告/CVE → 动作：批量滚更 → 校验：服务健康探针、崩溃率。
- **配置漂移检测 & 回收** 触发：Git 与运行态差异 → 动作：生成 PR/自动修复 → 校验：IaC 合规策略通过。

2) 可靠性与故障处置

- **自动限流/熔断/隔离** 触发：err_rate↑ / p95↑ → 动作：在网关/服务级应用限流与熔断 → 校验：下游恢复、全链 p95 回落。

- **异常回滚 & 一键 Mitigation** 触发：SLO 未命中 → 动作：Plan DSL 的 `compensate` 执行 → 校验：回滚后 SLO 达标。
- **降级策略编排（缓存兜底/只读模式）**触发：依赖服务不可用 → 动作：启用降级开关 → 校验：功能可用性 ≥ 阈值。
- **日志噪声抑制 & 指纹提纯** 触发：`log_pattern_5m` 激增 → 动作：更新过滤/聚合规则 → 校验：噪声占比下降。

3) 性能与容量优化

- **自动扩缩容（HPA/VPA/KEDA）**触发：RPS/CPU/GPU 利用率门槛 → 动作：扩容/缩容 → 校验：p95/队列长度回稳。
- **热点定位（Top-K 端点/表/索引）**触发：看板阈值 → 动作：生成优化建议（索引/缓存/批量）→ 校验：目标指标改善幅度。
- **慢查询守护（PostgreSQL/OLAP）**触发：计划回退/代价飙升 → 动作：计划冻结、索引重建、ANALYZE → 校验：执行时间降低。
- **CDN/缓存命中率调优** 触发：MISS 高/回源高 → 动作：规则更新、预热 → 校验：命中率 ↑、回源 ↓。

4) 数据与存储运维

- **分区/压缩/保留策略执行（Timescale/OO）**触发：窗口到期 → 动作：Timescale 压缩、OO 生命周期转冷 → 校验：存储曲线与查询命中率。
- **备份与恢复演练（DB/对象存）**触发：季度演练 → 动作：还原到沙箱/校验一致性 → 校验：RTO/RPO 达标。
- **Schema 变更护栏（DDL 门控）**触发：DDL PR → 动作：影子评估/回放 → 校验：读写延迟无回退。
- **数据质量监控（ETL/流式）**触发：空值/重复/延迟异常 → 动作：回填/重跑 → 校验：质量指标恢复。

5) 安全与合规

- **证书与密钥轮换（TLS/KMS/ACME）**触发：到期前 N 天 → 动作：自动签发/分发/热加载 → 校验：握手成功率、无中断。
- **CVE 修补与内核/容器镜像升级** 触发：高危 CVE → 动作：分批滚更 → 校验：漏洞基线清零。
- **访问评审与最小权限（RBAC/IAM）**触发：月度审计 → 动作：收敛权限/吊销闲置 → 校验：策略命中/拒绝率无异常。

- **合规窗口/变更冻结** 触发：活动/大促/监管窗口 → 动作：Gatekeeper 拒绝非白名单变更 → 校验：变更违规为 0。

6) 网络与边缘

- **入口网关/Ingress 规则变更与回滚** 触发：路径/权重调整 → 动作：灰度发布 Nginx/Envoy → 校验：5xx/延迟曲线。
- **跨区流量调度/健康探测** 触发：区域异常 → 动作：权重切换/只读路由 → 校验：丢包/RTT 恢复。
- **证书/ECH/TLS 指纹更新（Xray/Sing-box）**触发：策略更新 → 动作：节点批量更新 → 校验：连通率、误封率。
- **Egress/WAF 策略** 触发：异常出站/攻击面提示 → 动作：规则下发 → 校验：拦截率与误报率。

7) 集群与平台生命周期（K8s/容器/GPU）

- **K8s 小版本升级 & 节点滚更** 触发：版本落后 → 动作：逐组 Cordon/Drain/升级 → 校验：工作负载无中断。
- **GPU 驱动/Operator 更新（A10/L40/A100）**触发：新特性/修复 → 动作：分池灰度 → 校验：`nvidia-smi`/MIG 配置正确、作业吞吐。
- **容器运行时/镜像仓库镜像切换** 触发：镜像拉取慢/失效 → 动作：镜像加速/离线包 → 校验：拉取成功率、构建时长。
- **Ingress 离线安装包验证（你已有目标）**触发：新离线包 → 动作：沙箱校验/多架构镜像加载 → 校验：可用性与兼容性。

8) AI/ML 与数据应用运维

- **模型上线策略（Shadow→Canary）**触发：新模型评估通过 → 动作：影子对比/小流量灰度 → 校验：质量指标、成本/QPS。
- **数据/概念漂移监测** 触发：分布/性能漂移 → 动作：告警/Retrain 计划 → 校验：OOS 指标回升。
- **RAG 质量守护（向量库更新/召回评测）**触发：知识变更 → 动作：重嵌入/基准集评测 → 校验：Top-k 命中率/答案自信度。
- **推理路由与预算控制（多端点）**触发：延迟/价格波动 → 动作：路由切换/配额限流 → 校验：SLO 与成本曲线。

9) 可观测性与告警工程

- **SLO 定义与回归检测** 触发：SLO 变更 → 动作：历史重放评估 → 校验：误报/漏报下降。
- **告警噪声治理** 触发：高噪声规则 → 动作：聚合/抑制窗口优化 → 校验：On-call 负担降低。
- **追踪抽样自适应** 触发：流量/成本压力 → 动作：按错误/慢请求倾斜采样 → 校验：诊断覆盖率保持。

10) 灾备与演练

- **多区域演练（GameDay/Chaos）** 触发：季度/半年度 → 动作：注入故障/演练切流 → 校验：RTO/RPO/SLO 达标。
- **对象存储/消息系统灾备切换** 触发：主域不可用 → 动作：桶/Topic/镜像切换 → 校验：数据一致性与延迟。

11) FinOps（成本与资源）

- **实例规格与副本数 Rightsizing** 触发：长期低/高利用 → 动作：规格/副本调整 → 校验：性能不降、成本下降。
- **Spot/预留/竞价策略编排** 触发：价格/回收风险 → 动作：池化/回填 → 校验：业务无损、成本可控。
- **存储分层（热/温/冷）与清理** 触发：访问频次变化 → 动作：生命周期迁移/清理 → 校验：成本下降、命中率合理。

12) 自助与治理

- **服务目录 + 变更申请自助** 触发：团队提变更 → 动作：模板化 Plan、Gatekeeper 审批 → 校验：周期缩短、违规为 0。
- **合规审计与证据归档** 触发：审计周期 → 动作：导出 `event_envelope + evidence_link` → 校验：稽核一次通过。

Edge Collection vs Gateway

Discusses trade-offs between collecting observability data at the edge and routing through centralized gateways.

OTel Gateway Design and Considerations

Explores architectural patterns and design thoughts for building an OpenTelemetry-based gateway.

Data Ingestion with Multi-Level Persistence and Replay

Details strategies for persisting collected data at multiple stages and replaying it for analysis or recovery.

Full-Stack Observability Database Design

Outlines principles for structuring databases that support metric, log, trace, event, and profile storage in one system.

Full-Stack Observability Database ETL Design

Describes extract-transform-load workflows for moving observability data between hot and cold storage layers.

Monitoring System Multi-Region and Disaster Recovery

Covers approaches to deploying observability stacks across regions with failover and disaster recovery in mind.

From SSH/SCP to AI-Driven OPS Agent: Pre-Deployment Considerations

Reviews the challenges of traditional remote management and outlines factors to evaluate before adopting AI-powered operations agents.

From SSH/SCP to AI-Driven OPS Agent: Capability Checklist

Enumerates the functional requirements and competencies expected from an AI-assisted operations agent.

PostgreSQL Extension-Driven Complex Analysis (Vector, Graph, Trend)

Highlights how PostgreSQL extensions enable advanced analysis such as vector similarity search, graph traversal, and trend detection for operations data.

AI-OPS Agent MVP Architecture

Proposes a minimal viable architecture for an AI-powered operations agent integrating data collection, analysis, and automated actions.

如何设计 AI 驱动的 OPS Agent：漫谈状态机

> 关键词：Walking Skeleton ／状态机（FSM）／ Outbox ／ Idempotency ／事件驱动／ LLM 契约化

本文落地一条“最短闭环”（人工触发 → 计划 → 审批 → 执行 → 验证 → 归档），强调：**确定性的状态机做“合法性与原子落库”，不确定性的 LLM 做“生成、检索、归纳”**，两者以“可验证的契约”交汇。

---

0. 结论先行（TL;DR）

- **状态唯一事实源（SSOT）在 Orchestrator**：唯一改状态入口 `PATCH /case/{id}/transition`；事务内同时写 `ops_case`、`case_timeline`、`outbox`；所有副作用“先出盒，后投递”。
- **AI 的边界**：LLM 负责 Planner/Analyst/Librarian 的“生成与检索”；Gatekeeper 的策略判定与 Orchestrator 的状态迁移一律**不可由 LLM 直接驱动**，而是通过**结构化产物 + 守卫校验**进入确定世界。
- **Walking Skeleton 顺序**：Orchestrator → Planner → Gatekeeper → Executor → Verifier（打通闭环）；其后再补 Analyst、Sensor、Librarian（从命令式走向信号驱动与知识化）。

---

1. 为什么状态机是 AI OPS 的地基

LLM 擅长“生成与归纳”，但**不提供事务性与幂等保证**。生产级 OPS 需要：1) **可证明的合法迁移**（纯函数 FSM）2) **原子副作用**（状态 + 时间线 + 出盒）3) **至少一次投递 + 消费幂等**（Outbox/Inbox Pattern）

> 原则：**确定性包围不确定性**。所有非确定性的 AI 产物都转为**结构化契约**（JSON/YAML + Schema 校验）后，才允许进入状态机的下一跳。

---

2. Walking Skeleton（从零到可跑）

目标：先打通一条最短闭环，可演示、可观测、可回滚。

实现顺序：1. **Orchestrator（必须先做）**：纯函数 FSM + 原子事务（状态/时间线/出盒）+ 幂等/OCC。2. **Planner（最小可用）**：模板计划生成，产出 `plan_proposed` 与 `evt.plan.proposed.v1`。3. **Gatekeeper（自动审批）**：本地策略阈值，产出 `plan_approved` 与 `evt.plan.approved.v1`。4. **Executor（Step Runner）**：先做 `echo/script` 与 `k8s rollout` 两个适配器。5. **Verifier（SLO 校验）**：基于 `metric_1m` 的阈值判定。6~8. **Analyst / Sensor / Librarian**：把入口从“人工”扩展到“信号驱动”，把 Planner 从模板升级为基于证据的拟合。

---

3. 体系结构与事件流

```
flowchart LR
  subgraph UI/API
    H[Human/UI]
  end

  subgraph Core[Core Services]
    O[Orchestrator\nFSM+Outbox+Timeline]
    P[Planner\n(LLM/模板 → plan_proposed)]
    G[Gatekeeper\n(策略/OA 判定)]
    E[Executor\n(Adapters: script/k8s/...)]
    V[Verifier\n(SLO 阈值)]
    A[Analyst\n(规则/统计 → findings)]
    S[Sensor\n(oo_locator / ingest)]
    L[Librian\n(pgvector RAG)]
  end

  EV[(Event Bus\nNATS/Redpanda)]
  DB[(PostgreSQL/Timescale)]
  OO[(OpenObserve/Logs)]
```

```
  H -- REST --> O
  O -- outbox/cmd.* --> EV
  EV -- evt.* --> O
  P & G & E & V & A & S & L -- evt./cmd. <--> EV
  O -- R/W --> DB
  E -- logs/refs --> OO
```

接口与事件最小集 - REST:- `POST /case/create`,`GET /case/{id}`,`GET /case/{id}/timeline` - `PATCH /case/{id}/transition`（**唯一改状态入口**；需 `Idempotency-Key` + 建议 `If-Match`） - `POST /plan/generate`，`POST /gate/eval`，`POST /adapter/exec`，`POST /verify/run` - 事件：- `evt.case.transition.v1`（任何迁移事实） - `evt.plan.proposed.v1`，`evt.plan.approved.v1` - `evt.exec.step_result.v1`，`evt.exec.done|failed.v1` - `evt.verify.pass|failed.v1`，`evt.analysis.findings.v1` - `cmd.plan.generate` / `cmd.gate.eval` / `cmd.exec.run` / `cmd.verify.run`

---

4. 状态机（FSM）与守卫

```
stateDiagram-v2
  [*] --> NEW
  NEW --> ANALYZING: start_analysis
  ANALYZING --> PLANNING: analysis_done
  PLANNING --> WAIT_GATE: plan_ready (guard: complete plan)
  WAIT_GATE --> EXECUTING: gate_approved
  WAIT_GATE --> PARKED: gate_rejected
  EXECUTING --> VERIFYING: exec_done
  EXECUTING --> PARKED: exec_failed
  VERIFYING --> CLOSED: verify_pass
  VERIFYING --> PARKED: verify_failed
  PARKED --> PLANNING: resume / fix
```

守卫示例（plan 完整性） - `steps` 至少 1 个；每个 step 定义 `action`、`timeout`、`retry`；- `rollback` 与 `verify` 字段必须存在；- 计划大小、危险操作需标识并由 Gatekeeper 评分/审批。

> 任何守卫失败都不改变状态，仅记录时间线并返回 409/422。

---

5. Orchestrator 的“原子三件套”

单事务保证：1) 更新 `ops_case.version`，`status` 2) 追加 `case_timeline` 3) 写入 `outbox`（cmd.* 或 evt.* 载荷）

幂等与并发：- Idempotency-Key：命中即返回首次结果，不重复写时间线/出盒。- If-Match/expected_version：不匹配 → 409/412。

伪代码（Go/Pseudocode）：

```
func Transition(ctx, caseID, event, meta) (Case, error) {
  return WithTx(ctx, func(tx Tx) (Case, error) {
    c := repo.GetForUpdate(tx, caseID)
    allowed, next := workflow.Decide(c.Status, event)
    if !allowed { return error(409) }

    // guards (schema/plan completeness/risk window etc.)
    if err := Guards.Pass(tx, c, event, meta); err != nil { return error(422) }

    c.Status, c.Version = next, c.Version+1
    repo.Update(tx, c)

    timeline.Append(tx, c.ID, event, meta)

    outbox.Append(tx, BuildMessages(c, event, meta))

    return c, nil
  })
}
```

---

6. LLM “做擅长的事”：契约与位置

> **口号**：LLM 只输出“结构化、可验证、可回放”的产物；状态改变永远经 Orchestrator。

6.1 Planner × LLM（计划生成）

- **输入**：问题摘要、上下文证据（evidence_link）、历史相似案 kb_chunk、环境约束。
- **输出契约（JSON Schema）**：

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "ChangePlan",
  "type": "object",
  "required": ["title","steps","rollback","verify"],
  "properties": {
    "title": {"type":"string"},
    "risk_score": {"type":"number","minimum":0,"maximum":1},
    "steps": {
```

```
      "type":"array","minItems":1,
      "items": {"type":"object","required":["action","timeout_s","retry"],
        "properties": {
          "action": {"enum":["k8s.rollout","script.exec","gateway.flip","noop"]},
          "params": {"type":"object"},
          "timeout_s": {"type":"integer","minimum":10},
          "retry": {"type":"integer","minimum":0,"maximum":3}
        }
      }
    },
    "rollback": {"type":"object"},
    "verify": {"type":"object"}
  }
}
```

- **流程**：`/plan/generate` → 校验 Schema → 入库 `plan_proposed` → 发 `evt.plan.proposed.v1`。

Prompt 片段（示意）：

```
System: 你是资深 SRE。产出必须是符合 JSON Schema 的 ChangePlan。不要输出自然语言解释。
User: {问题摘要/证据/限制}
Assistant: {ChangePlan JSON}
```

6.2 Analyst × LLM（证据摘要）

- LLM 用于**长日志/多指标的摘要**，把粗粒度“异常信号”转换成**可检索的要点**（写入 `event_envelope`）。
- 开 Case 的动作依然通过 Orchestrator（NEW→ANALYZING）。

6.3 Librarian × LLM（RAG）

- `kb_doc/kb_chunk`（pgvector）做相似案例/Runbook 检索；LLM 用于**拟合与注释**，生成可执行/可审阅的 Plan。

6.4 Gatekeeper/Verifier 的 AI 用法（限定）

- **Gatekeeper**：策略由**确定性规则**实现，可让 LLM 提供“风险解释文本”，但**不直接决定通过与否**。
- **Verifier**：阈值判断（p95/err_rate）是确定性的；LLM 可做**事后报告**摘要。

---

7. 数据模型（关键表，最小集）

```
-- 1) 事实源
CREATE TABLE ops_case (
```

```
  id UUID PRIMARY KEY,
  tenant TEXT, title TEXT,
  status TEXT NOT NULL,
  version INT NOT NULL,
  created_at TIMESTAMPTZ DEFAULT now(),
  updated_at TIMESTAMPTZ DEFAULT now()
);

CREATE TABLE case_timeline (
  id BIGSERIAL PRIMARY KEY,
  case_id UUID REFERENCES ops_case(id),
  event TEXT, actor TEXT, reason TEXT,
  correlation_id TEXT, meta JSONB,
  created_at TIMESTAMPTZ DEFAULT now()
);

CREATE TABLE outbox (
  id BIGSERIAL PRIMARY KEY,
  topic TEXT, key TEXT, payload JSONB,
  created_at TIMESTAMPTZ DEFAULT now(),
  published_at TIMESTAMPTZ
);

CREATE TABLE idempotency (
  key TEXT PRIMARY KEY,
  response JSONB,
  created_at TIMESTAMPTZ DEFAULT now()
);

-- 2) Planner/Exec/Verify (示意)
CREATE TABLE plan_proposed (
  id UUID PRIMARY KEY,
  case_id UUID REFERENCES ops_case(id),
  plan JSONB NOT NULL,
  created_at TIMESTAMPTZ DEFAULT now()
);
CREATE TABLE plan_approved (
  id UUID PRIMARY KEY,
  case_id UUID REFERENCES ops_case(id),
  decision JSONB,
  created_at TIMESTAMPTZ DEFAULT now()
);
```

8. 可观测性（先打指标，再写业务）

- **Prom 指标**：
  - `case_transition_total{from,to,event}`
  - `conflict_total{reason}`（409/412/422）
  - `idempotent_replay_total`
  - `outbox_publish_total{topic,status}` / `inbox_dedup_total`
  - `adapter_step_latency_seconds`、`verify_latency_seconds`
- **Tracing**：每次迁移、每个 step、每次查询都带 `trace_id` 并写入 `case_timeline.meta`。
- **日志引用**：`exec_step.*_ref` 指向 OO/Loki 的 `oo_locator`。

---

9. 第一条端到端演示（cURL）

```
# 1) 创建 Case
CASE=$(curl -sX POST /case/create -d '{"title":"p95 spike","tenant":"t-001"}'|jq -r .dat

# 2) 开始分析
curl -X PATCH /case/$CASE/transition \
  -H 'Idempotency-Key:a1' \
  -d '{"event":"start_analysis","actor":"analyst@svc"}'

# 3) 生成计划（模板/LLM）
curl -X POST /plan/generate -d '{"case_id":"'$CASE'","template":"k8s.rollout.canary"}'
# Planner 发 evt.plan.proposed.v1 → Orchestrator 判定 plan_ready → WAIT_GATE

# 4) 自动审批
curl -X POST /gate/eval -d '{"case_id":"'$CASE'"}'
# Gatekeeper 发 evt.plan.approved.v1 → EXECUTING

# 5) 执行适配器（示例：script）
curl -X POST /adapter/exec -d '{"case_id":"'$CASE'","adapter":"script","script":"echo ok
# Executor 发 evt.exec.done.v1 → VERIFYING

# 6) 验证通过 → CLOSED
curl -X POST /verify/run -d '{"case_id":"'$CASE'"}'
```

---

10. 目录结构建议

```
.
  api/                # Gin/Fiber + OpenAPI + 中间件
  workflow/           # 纯函数 FSM
```

```
plans/              # 纯函数计划生成（模板 + LLM 契约）
gatekeeper/         # 策略引擎（本地规则/OPA/Cedar）
executor/
  adapters/        # k8s/script/gateway...
  runner/          # step 执行/超时/重试
verifier/
analyst/
sensor/
librarian/
internal/pubsub/    # NATS/Redpanda（至少一次 + 去重）
ports/              # Repo/Outbox/Timeline/Idempotency 接口
services/           # Orchestrator 等聚合服务
migrations/         # DDL
```

---

11. DoD 清单（可直接转 Issue）

- **Orchestrator**：三类集成测试全绿（非法迁移 409 / 幂等重放 / 失败落 PARKED）；指标 `case_transition_total / conflict_total / idempotent_replay_total`；Outbox 消费无重复副作用。
- **Planner**：`plan_proposed` 入库；DSL 过 schema；守卫挡下不完整计划。
- **Gatekeeper**：策略可配置（文件/内存）；自动审批延时 <1s；拒绝落 PARKED 并记原因。
- **Executor**：≥2 适配器；step 超时/重试；失败回写 `exec_failed` 并落 PARKED。
- **Verifier**：SLO 查询正确；阈值/窗口可配；误报率可控。
- **Analyst**：能从 `metric_1m` 发现简单异常并自动开 Case。
- **Sensor**：`oo_locator` 可被 `evidence_link` 回链；写入 p99 < 2s。
- **Librarian**：topK 语义检索可用于计划注释/回滚提示。

---

12. 风险与避坑

1) 事务一致性：禁止在 FSM 回调内做外部 IO；**先落地再发事件**。
2) 幂等：所有写接口都要 `Idempotency-Key`；消费者对 `message_id` 去重。
3) 并发：`SELECT ... FOR UPDATE` + 版本控制；冲突返回 409/412。
4) 守卫松紧：先宽后紧，避免卡死；策略统一在 Gatekeeper。
5) 可观测：**先打指标**，再写业务；没有指标就无法排障。

---

13. 路线图（建议）

- **Phase 0**：FSM + Outbox + 2 个适配器 + 阈值 Verifier（闭环演示）。

- **Phase 1**：Analyst（规则）+ Librarian（RAG）→ Planner/Runbook 拟合。
- **Phase 2**：Gatekeeper 外挂 OPA/Cedar；Executor 扩充生产适配器；Verifier 引入 SLO 画像。

一句话

> **Orchestrator 只做两件事**：判定合法迁移（纯函数）＋一次事务内记录（状态/时间线/出盒）。其余模块要么被它的命令驱动，要么把“事实事件”交还它，由它来改变世界。LLM 负责提出“更好的方案”，而不是亲自“按下回车”。