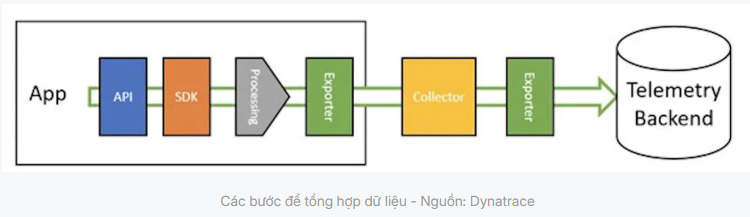
Triển Khai OpenTelemetry trong môi trường VETC

1. Giới Thiệu
   1. Thực trạng
   2. OpenTelemetry

Opentelemetry là một dự án mã nguồn mở được phát triển bởi Cloud Native Computing Foundation (CNCF). Là sự kết hợp nhưng ưu điểm của OpenCensus và OpenTracing, mục tiêu cung cấp tập hợp các SDK, API và một tiêu chuẩn chuẩn hóa khôn phụ thuộc vào bên thứ ba cho việc thu thập, chuyển đổi và gửi dữ liệu telemetry đến một hệ thống giám sát khác.

* 1. Phương thức hoạt động



* + 1. OpenTelemetry cung cấp các loại API để ứng dụng có thể dùng đó để ghi nhận lại các thông tin của request. Trong quá trình Tracing, OpenTelemetry sẽ theo dõi các request và gắn chúng vào các context của API cho phép chúng ta theo dõi quá trình thực thi từ đầu đến cuối.
    2. Ngoài ra, OpenTelemetry cung cấp SDK với nhiều ngôn ngữ lập trình khác nhau (Java, Python, Go, C#, và nhiều ngôn ngữ khác). Các SDK này giúp cho ứng dụng và hệ thống thu thập dữ liệu một cách nhanh chóng và dễ dàng hơn.
    3. OpenTelemetry ghi nhận dữ liệu của ứng dụng bao gồm thông tin về yêu cầu HTTP, thời gian xử lý, lỗi, thông tin về hiệu suất và nhiều thông tin khác.
    4. Những dữ liệu này được thu thập sau đó được xuất ra các hệ thống khác tùy theo nhu cầu giám sát như các nền tảng: Prometheus, Grafana, Jaeger hoặc các hệ thống lưu trữ dữ liệu như Elasticsearch, InfluxDB, và nhiều nền tảng khác. Các hệ thống bên ngoài này sẽ chịu trách nhiệm lưu trữ, truy vấn và hiển thị thông tin cho mục đích giám sát và phân tích.
  1. Các thành phần trong Opentelemetry và chức năng
     1. Log trong OpenTelemetry

Được hiểu là các thông điệp ghi nhận lại các sự kiện, trạng thái hoặc thông tin từ các ứng dụng và hệ thống. Log là một phần quan trọng trong việc giám sát và theo dõi ứng dụng để phát hiện ra sự cố, phân tích và khắc phục lỗi.

* + 1. Span trong Opentelemetry

Ngược lại với **Log**, **Span** đại diện cho một đơn vị công việc. Nó theo dõi các hoạt động cụ thể của một request được thực hiện. Từ đó tạo ra một bức tranh tổng thể về những gì đã xảy ra trong thời gian mà request đó được thực hiện.

* + 1. Span Context trong OpenTelemetry

Đây là một phần của **Span** được chuỗi hóa và truyền kèm cùng với **Distributed Context.**

* + 1. Distributed Trace trong OpenTelemetry

Nhìn vào bức ảnh trên, ta có thể hiểu **Distributed Trace** cung cấp cho chúng ta cái nhìn tổng quan về những gì xảy ra khi một yêu cầu được thực hiện đến một ứng dụng. Ngoài ra, nó cho phép các nhà phát triển theo dõi yêu cầu qua nhiều dịch vụ và thành phần. Điều này tăng khả năng quan sát từ đầu đến cuối trong các luồng phức tạp của hệ thống. Từ đó dễ dàng tìm được những điểm **bottelneck**[[1]](https://200lab.io/blog/opentelemetry-la-gi#fn1) của hệ thống

Một **Trace** được tạo thành từ một hoặc nhiều **Spans**. Điều này có nghĩa là nhiều **Spans** sẽ có cùng **traceId**[[2]](https://200lab.io/blog/opentelemetry-la-gi#fn2). Mỗi **root span**  
[[3]](https://200lab.io/blog/opentelemetry-la-gi#fn3) rootSpan đại diện cho một yêu cầu từ đầu đến cuối. Các **Span** phía dưới **Parent Span** sẽ đưa ra ngữ cảnh chi tiết hơn về những gì xảy ra trong quá trình yêu cầu

* + 1. Attribute trong OpenTelemetry

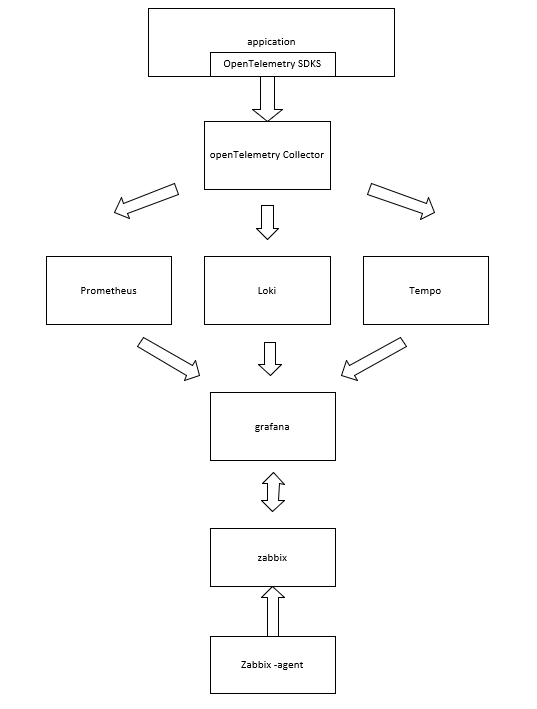
Đây là cặp key-value chứa dữ liệu mô tả mà ta có thể sử dụng để chú thích thêm một số thông tin phục vụ cho việc tracking.

* + 1. Context Propagation trong OpenTelemetry
* Context là một đối tượng chứa thông tin để các dịch vụ giữa bên nhận và bên gửi liên kết thành một Span để tạo thành một Trace tổng thể. Ví dụ, nếu service A gọi service B, thì một Span từ service A có ID trong ngữ cảnh sẽ được sử dụng làm Parent Span cho span tiếp theo được tạo ra trong service B.
* Propagation là cơ chế inject thông tin Context giữa các dịch vụ với nhau. Bằng cách làm như vậy, nó tạo thành một Distributed Trace. Nó sẽ serialize/deserialize Span Context và cung cấp thông tin Trace liên quan để truyền từ một dịch vụ này sang dịch vụ khác. Chúng ta có thể gọi đây là Trace Context.
  1. Collectors trong OpenTelemetry

các thành phần trong **Collector** của **OpenTelemetry.**

* **Agent**: Một phiên bản **Collector** chạy cùng với ứng dụng hoặc trên cùng máy chủ với ứng dụng (ví dụ: binary, sidecar).
* **Gateway**: Một hoặc nhiều phiên bản **Collector** chạy như một dịch vụ độc lập (ví dụ: container hoặc deployment) thường là mỗi cụm, mỗi trung tâm dữ liệu.
* Receivers là thành phần đầu tiên trong Collector, nó đóng vai trò như một cổng lắng nghe dữ liệu đến từ các nguồn khác nhau. Chúng ta có thể cấu hình nó lắng nghe dữ liệu từ các agent được triển khai trên các máy chủ ứng dụng, cổng giao thức (HTTP, gRPC, Thrift, etc.), hoặc từ các nguồn khác.
* Processors: Đóng vai trò để xử lý dữ liệu sau khi nó được nhận bởi Receiver. Các Processor thường thực hiện các tác vụ như chuyển đổi định dạng dữ liệu, lọc và lấy mẫu dữ liệu, thêm thông tin bổ sung (ví dụ: context) vào dữ liệu hoặc thực hiện các tác vụ tùy chỉnh khác tùy theo nhu cầu chúng ta mong muốn.
* Exporters chịu trách nhiệm xuất dữ liệu đã qua xử lý ở bước Processor ra ngoài hệ thống khác. Dữ liệu có thể được xuất ra các nền tảng giám sát như Prometheus, Jaeger, Grafana, hoặc các hệ thống lưu trữ dữ liệu như Elasticsearch, InfluxDB, và nhiều hệ thống khác. Exporter đảm bảo rằng dữ liệu được truyền sang một cách đáng tin cậy và hiệu quả.
* Extensions (tùy chọn) Collector có hỗ trợ các tiện ích mở rộng để cung cấp tính năng bổ sung hoặc tích hợp với các hệ thống và dịch vụ khác. Các tiện ích mở rộng này có thể liên
  1. Ưu điểm và nhược điểm

1. Cách thức triển khai



**1. Chức năng của từng thành phần**

**Applications**

* **Chức năng**: Đây là các ứng dụng hoặc dịch vụ (microservices, monolithic apps, v.v.) cần được giám sát. Chúng có thể được viết bằng nhiều ngôn ngữ (Java, Python, Go, v.v.) và chạy trên các nền tảng khác nhau (Kubernetes, VMs, bare-metal).
* **Vai trò**: Tạo ra dữ liệu telemetry (metrics, traces, logs) thông qua các công cụ như OpenTelemetry SDKs hoặc auto-instrumentation.

**OpenTelemetry SDKs**

* **Chức năng**: Cung cấp API và SDK để instrument ứng dụng, thu thập dữ liệu telemetry (metrics, traces, logs). Auto-instrumentation hoặc manual instrumentation có thể được sử dụng để giảm thiểu thay đổi mã nguồn.
* **Vai trò**: Gắn mã vào ứng dụng để ghi lại thông tin về hiệu suất (e.g., thời gian xử lý request, lỗi) và gửi dữ liệu này đến OpenTelemetry Collector.
* **Ví dụ**: Một ứng dụng Java sử dụng OpenTelemetry Java SDK để ghi lại thời gian phản hồi của API và gửi dưới dạng OTLP (OpenTelemetry Protocol).

**OpenTelemetry Collector**

* **Chức năng**: Một dịch vụ trung gian thu thập, xử lý, và xuất dữ liệu telemetry từ nhiều nguồn (ứng dụng, cơ sở hạ tầng) đến các backend như Prometheus, Loki, hoặc Tempo.
* **Vai trò**:
  + **Thu thập**: Nhận dữ liệu từ OpenTelemetry SDKs qua giao thức OTLP (gRPC/HTTP).
  + **Xử lý**: Lọc, nhóm, hoặc làm giàu dữ liệu (e.g., thêm metadata như môi trường hoặc vùng).
  + **Xuất**: Gửi dữ liệu đến các hệ thống lưu trữ phù hợp (Prometheus cho metrics, Loki cho logs, Tempo cho traces).
* **Ví dụ**: Collector nhận metrics từ ứng dụng, nhóm chúng theo lô (batch) để tối ưu, và gửi đến Prometheus.

**Zabbix Agents**

* **Chức năng**: Các tác nhân (agents) được cài đặt trên máy chủ hoặc thiết bị để thu thập metrics cơ sở hạ tầng (CPU, RAM, disk, network) và gửi về Zabbix Server.
* **Vai trò**: Giám sát các thiết bị vật lý, máy ảo, hoặc container mà không cần instrumentation ứng dụng.
* **Ví dụ**: Zabbix Agent trên một server Linux thu thập CPU usage và gửi đến Zabbix Server qua giao thức Zabbix.

**Zabbix**

* **Chức năng**: Hệ thống giám sát tập trung, lưu trữ và phân tích dữ liệu từ Zabbix Agents hoặc các nguồn khác (SNMP, IPMI, HTTP).
* **Vai trò**:
  + Lưu trữ dữ liệu vào cơ sở dữ liệu (e.g., PostgreSQL, MySQL).
  + Cung cấp API để truy vấn dữ liệu và tạo cảnh báo dựa trên ngưỡng (thresholds).
  + Hỗ trợ giám sát cơ sở hạ tầng truyền thống (servers, network devices).
* **Ví dụ**: Zabbix phát hiện một server có CPU usage > 90% và gửi cảnh báo qua email.

**Prometheus/Loki/Tempo**

* **Chức năng**:
  + **Prometheus**: Lưu trữ và truy vấn metrics thời gian thực (time-series data) từ OpenTelemetry hoặc các nguồn khác.
  + **Loki**: Lưu trữ và truy vấn logs hiệu quả, được tối ưu cho việc tìm kiếm và lọc.
  + **Tempo**: Lưu trữ và truy vấn traces để theo dõi luồng xử lý trong hệ thống phân tán.
* **Vai trò**: Là các backend lưu trữ dữ liệu telemetry từ OpenTelemetry Collector, được tối ưu cho từng loại dữ liệu (metrics, logs, traces).
* **Ví dụ**:
  + Prometheus lưu trữ metrics như số lượng request mỗi giây.
  + Loki lưu trữ logs ứng dụng như lỗi HTTP 500.
  + Tempo lưu trữ traces cho thấy thời gian xử lý của một request qua nhiều microservices.

**Grafana**

* **Chức năng**: Nền tảng trực quan hóa dữ liệu, tích hợp với nhiều nguồn dữ liệu (Zabbix, Prometheus, Loki, Tempo) để tạo dashboard và cảnh báo.
* **Vai trò**:
  + Kéo dữ liệu từ Zabbix (qua API hoặc database) và từ Prometheus/Loki/Tempo.
  + Tạo dashboard hiển thị metrics, logs, và traces trong một giao diện thống nhất.
  + Hỗ trợ phân tích và liên kết dữ liệu (e.g., từ metrics đến traces).
* **Ví dụ**: Một dashboard Grafana hiển thị CPU usage từ Zabbix, số lượng request từ Prometheus, và traces từ Tempo.

**Zabbix API/Database**

* **Chức năng**:
  + **API**: Cung cấp giao diện để Grafana truy vấn dữ liệu từ Zabbix.
  + **Database**: Lưu trữ dữ liệu lịch sử và trends từ Zabbix.
* **Vai trò**: Cho phép Grafana truy xuất dữ liệu Zabbix để hiển thị trên dashboard.
* **Ví dụ**: Grafana sử dụng Zabbix API để lấy dữ liệu về disk usage trong 24 giờ.

**Dashboards (Metrics, Traces, Logs)**

* **Chức năng**: Giao diện trực quan hiển thị dữ liệu từ nhiều nguồn, hỗ trợ phân tích và ra quyết định.
* **Vai trò**: Tổng hợp thông tin từ Zabbix và OpenTelemetry để cung cấp cái nhìn toàn diện về hệ thống.
* **Ví dụ**: Một dashboard hiển thị CPU usage (Zabbix), latency của API (Prometheus), và lỗi trong logs (Loki).

**2. Luồng dữ liệu**

Dữ liệu di chuyển qua các thành phần theo các bước sau:

1. **Từ Applications đến OpenTelemetry SDKs**:
   * Ứng dụng được instrument bằng OpenTelemetry SDKs hoặc auto-instrumentation (e.g., Grafana Beyla).
   * SDKs thu thập:
     + **Metrics**: Số liệu như request rate, error rate, latency.
     + **Traces**: Thông tin về luồng xử lý request qua các dịch vụ (e.g., API call từ service A đến service B).
     + **Logs**: Nhật ký ứng dụng như thông báo lỗi hoặc sự kiện.
   * Dữ liệu được gửi đến OpenTelemetry Collector qua giao thức OTLP (gRPC hoặc HTTP).
2. **Từ OpenTelemetry SDKs đến OpenTelemetry Collector**:
   * Collector nhận dữ liệu OTLP từ nhiều ứng dụng.
   * Collector xử lý dữ liệu:
     + Lọc bỏ dữ liệu không cần thiết (e.g., traces không quan trọng).
     + Nhóm dữ liệu thành lô (batch) để tối ưu truyền tải.
     + Thêm metadata như tên dịch vụ, môi trường (dev, prod).
   * Collector xuất dữ liệu đến các backend:
     + Metrics → Prometheus.
     + Logs → Loki.
     + Traces → Tempo.
3. **Từ Zabbix Agents đến Zabbix**:
   * Zabbix Agents chạy trên các server hoặc thiết bị, thu thập metrics cơ sở hạ tầng (CPU, memory, disk, network).
   * Dữ liệu được gửi đến Zabbix Server qua giao thức Zabbix (thường là TCP port 10051).
   * Zabbix Server lưu trữ dữ liệu vào cơ sở dữ liệu (e.g., PostgreSQL) và cung cấp qua API.
4. **Từ OpenTelemetry Collector đến Prometheus/Loki/Tempo**:
   * Collector gửi dữ liệu đã xử lý đến các backend tương ứng:
     + **Prometheus**: Nhận metrics qua Prometheus Remote Write.
     + **Loki**: Nhận logs qua HTTP API.
     + **Tempo**: Nhận traces qua OTLP.
   * Mỗi backend lưu trữ dữ liệu theo cách tối ưu cho loại dữ liệu của nó (time-series, log indexing, trace indexing).
5. **Từ Zabbix/Prometheus/Loki/Tempo đến Grafana**:
   * Grafana kết nối với các nguồn dữ liệu:
     + **Zabbix**: Qua Zabbix API (sử dụng plugin Grafana-Zabbix) hoặc truy vấn trực tiếp database (PostgreSQL/MySQL).
     + **Prometheus**: Qua HTTP API để lấy metrics.
     + **Loki**: Qua HTTP API để lấy logs.
     + **Tempo**: Qua gRPC/HTTP để lấy traces.
   * Grafana tổng hợp dữ liệu từ các nguồn này và hiển thị trên dashboard.
6. **Từ Grafana đến Dashboards**:
   * Grafana tạo các dashboard hiển thị:
     + Metrics: Biểu đồ CPU usage (Zabbix), request latency (Prometheus).
     + Logs: Danh sách lỗi từ Loki.
     + Traces: Luồng xử lý request qua các dịch vụ từ Tempo.
   * Người dùng tương tác với dashboard để phân tích, liên kết dữ liệu (e.g., click vào metric để xem trace liên quan), và nhận cảnh báo.