

模块十五：eBPF 零侵入可观测性开发实战

王炜/前腾讯云 CODING 高级架构师

目录

- 1 eBPF BCC 实战
- 2 通过 eBPF、Beyla 实现零侵入 Metrics 和 Tracing
- 3 通过 eBPF、Cilium、Hubble 实现零侵入可观测
- 4 借助 Flaco 实时监测 K8s 集群安全威胁

1. eBPF BCC 实战

什么是 BCC

BCC (BPF Compiler Collection) 是一个用于开发和运行 eBPF 程序的开源库，旨在让开发者更轻松地编写、加载和运行 eBPF 程序。它由 Facebook 开发并开源。BCC 提供了丰富的开发工具和 Python、C++ 等语言的 API，使得开发者可以更方便地在 Linux 系统中执行高性能的内核观察和调试任务。

使用条件：>= Linux 4.1

BCC 的核心功能

1. 简化 eBPF 开发：

BCC 提供了用户空间库和工具，屏蔽了许多 eBPF 程序开发的复杂细节，如字节码加载和验证等。

2. 多语言支持：

BCC 提供了 Python 和 C++ 的接口，允许开发者使用高级语言快速开发和测试 eBPF 程序。

3. 内置工具集：

BCC 包括一系列现成的 eBPF 工具（如 `execsnoop`, `biosnoop`, `tcplife` 等），用于内核调试、性能分析和安全监控。

4. 丰富的内核能力：

BCC 提供了访问 Linux 内核数据结构的能力，结合内核的跟踪点（如 `kprobe` 和 `tracepoints`）和网络数据路径功能，可以实现对内核行为的高效监控。

安装 BCC (Ubuntu) 并体验

1. 开通腾讯云虚拟机
2. `sudo apt-get install bpfcc-tools linux-headers-$(uname -r)`
3. `git clone https://github.com/iovisor/bcc.git`

BCC 入门

```
1  #!/usr/bin/python
2  # Copyright (c) PLUMgrid, Inc.
3  # Licensed under the Apache License, Version 2.0 (the "License")
4
5  # run in project examples directory with:
6  # sudo ./hello_world.py"
7  # see trace_fields.py for a longer example
8
9  from bcc import BPF
10
11 # This may not work for 4.17 on x64, you need replace kprobe__sys_clone with kprobe___x64_sys_clone
12 BPF(text='int kprobe__sys_clone(void *ctx) { bpf_trace_printk("Hello, World!\\n"); return 0;
    }').trace_print()
```

1. 导入 BCC 库

2. 执行一段 C 语言编写的 BPF 内核探测代码 (kprobe)

BPF 程序源代码

```
1  int kprobe__sys_clone(void *ctx) {  
2      bpf_trace_printk("Hello, World!\n");  
3      return 0;  
4  }
```

- `int kprobe__sys_clone(void *ctx):`
 - `kprobe`: 表示一个内核探针并挂载到内核函数上，用于在特定内核事件发生时执行用户定义的逻辑
 - `sys_clone`: 内核中用于进程克隆（类似 `fork()`）的系统调用函数。这个 eBPF 程序会在每次调用 `sys_clone` 时被触发
 - `ctx`: 上下文参数，用于提供事件的相关信息
- `bpf_trace_printk():`
 - 这是 eBPF 提供的一个函数，用于向用户空间输出调试信息
 - 在这个例子中，当内核发生进程克隆调用时，将会输出 "Hello, World!"

BPF 程序源代码

```
bcc.py
1 .trace_print()
```

- `trace_print()` 方法：
 - BCC 提供的一个方法，用于实时读取内核中 eBPF 程序输出的信息（由 `bpf_trace_printk()` 打印的内容）
 - 这个方法将会阻塞程序，持续监听并打印来自内核的调试输出

运行 eBPF 程序

1. `sudo examples/hello_world.py`

2. 不断输出打印内容

```
b'          <...>-16656  [000] ....1  887.941568: bpf_trace_printk: Hello, World!'
b''
b'    barad_agent-16656  [000] ....1  887.945494: bpf_trace_printk: Hello, World!'
b''
b'          <...>-9748   [001] ....1  889.312306: bpf_trace_printk: Hello, World!'
b''
b'    k3s-server-9671    [000] ....1  889.312638: bpf_trace_printk: Hello, World!'
b''
b'          <...>-16667  [001] ....1  892.945126: bpf_trace_printk: Hello, World!'
b''
b'          sh-16670     [000] ....1  892.949557: bpf_trace_printk: Hello, World!'
b''
b'          sh-16670     [000] ....1  892.949872: bpf_trace_printk: Hello, World!'
b''
b'          sh-16670     [000] ....1  892.950103: bpf_trace_printk: Hello, World!'
b''
b'          <...>-16675  [000] ....1  894.941562: bpf_trace_printk: Hello, World!'
b''
b'    barad_agent-16676  [001] ....1  894.945326: bpf_trace_printk: Hello, World!'
b''
b'    barad_agent-16675  [000] ....1  894.946765: bpf_trace_printk: Hello, World!'
b''
```

发生了什么（用户态）？

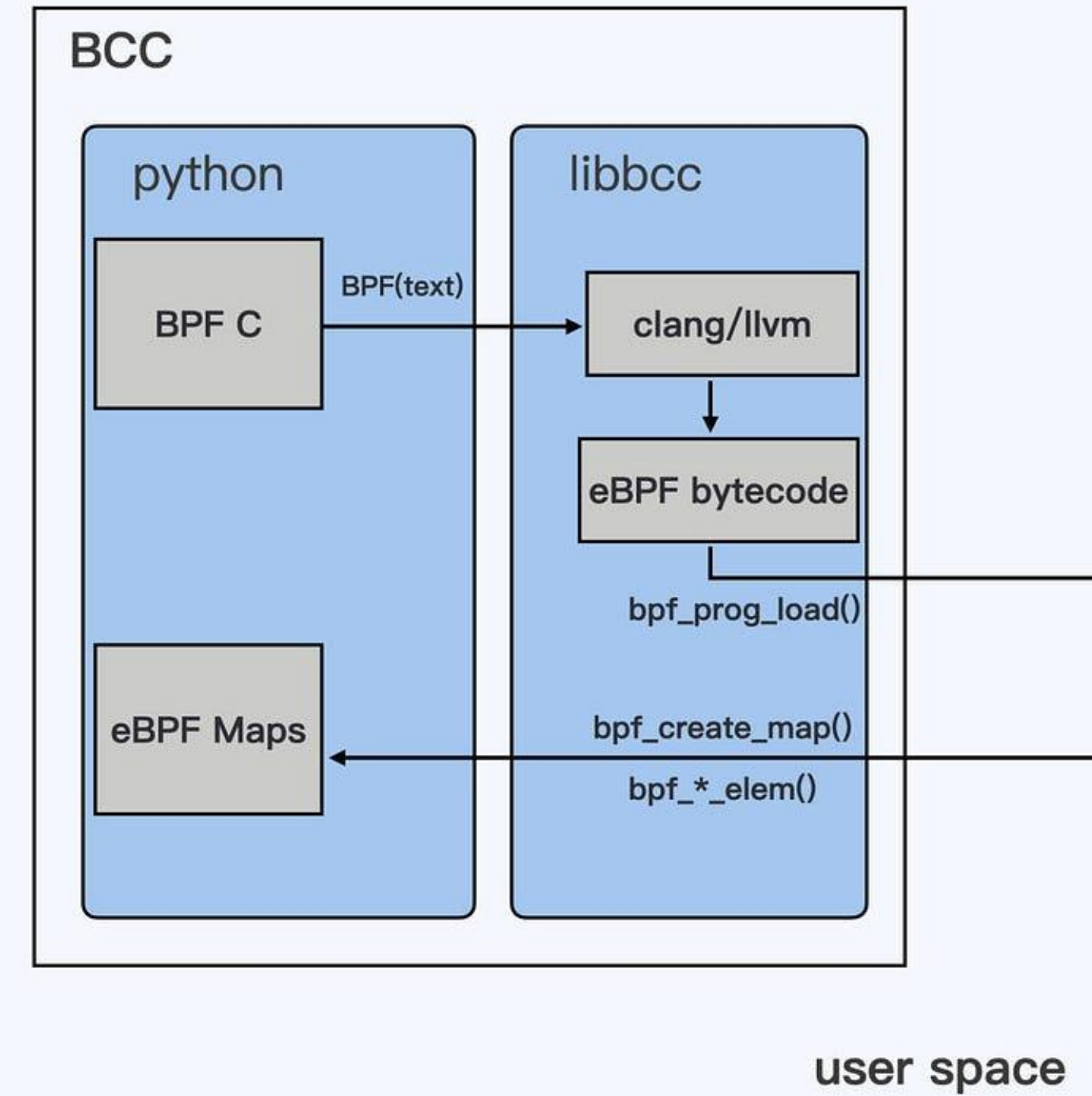
1. BCC 调用 Clang/LLVM 将 eBPF C 程序编译成 eBPF 字节码

1. 语法分析与语义检查

2. 生成 eBPF 中间表示 (IR)

3. 输出符合 eBPF 虚拟机运行要求的字节码

2. libbcc 通过调用 `bpf_prog_load()` 将字节码加载到内核



发生了什么（内核态）？

1. Verifier（验证器）验证 eBPF 程序

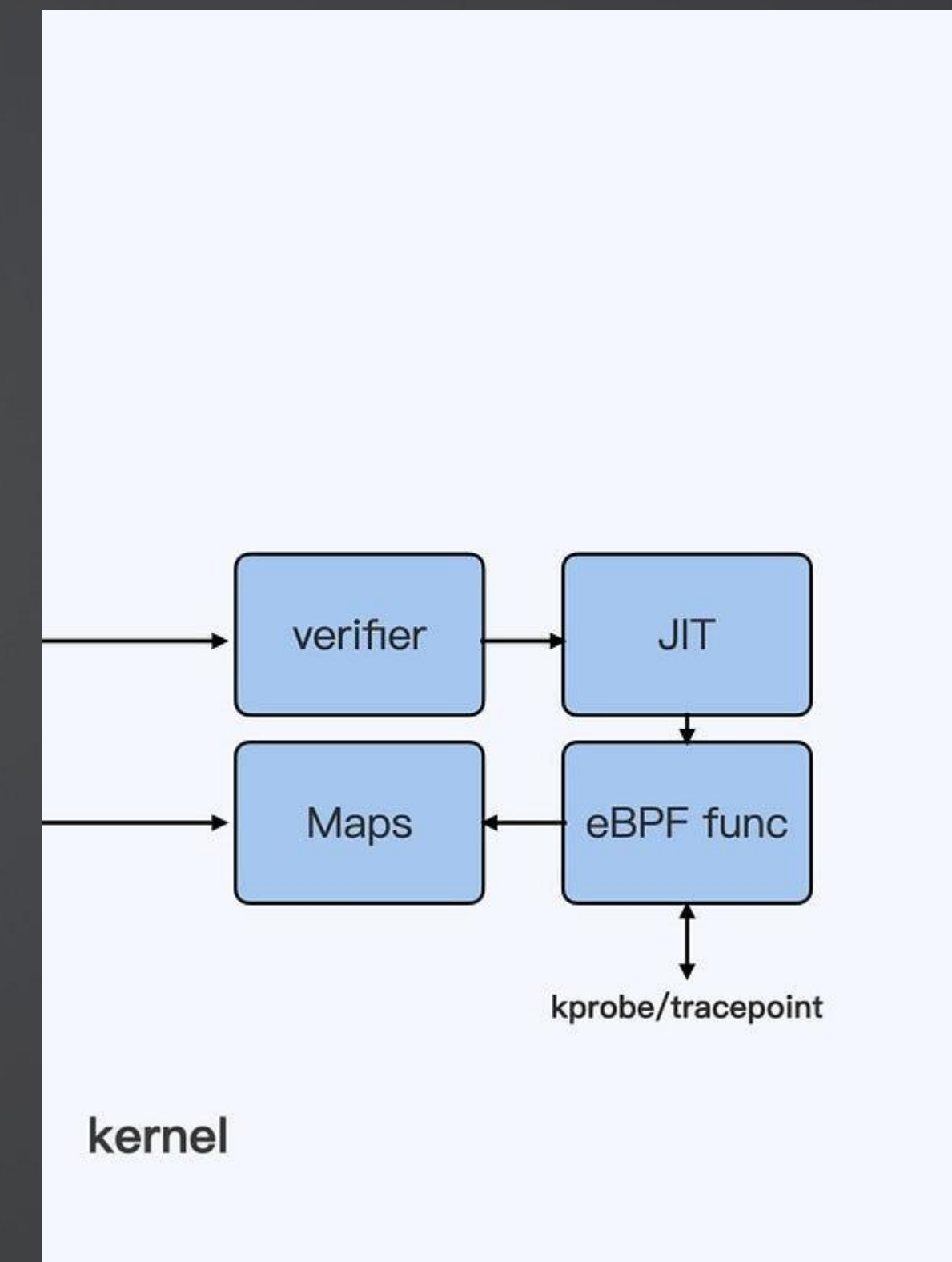
1. 没有非法的内存访问
2. 执行流程是有限的，不会陷入死循环
3. 符合 eBPF 的指令集限制

2. Just-In-Time (JIT) 编译器

1. 验证通过后，借助 JIT 将字节码编译成机器码，以便提升执行性能
2. 如果没有启动 JIT，eBPF 程序将以字节码形式在 eBPF 虚拟机中运行

3. eBPF func

1. eBPF 函数是最终运行的程序，它被挂载到内核的某个 Hook（如 kprobe 或 tracepoint）上，并在事件触发时运行
2. 例如挂载到 sys_clone 函数入口，拦截系统调用时运行

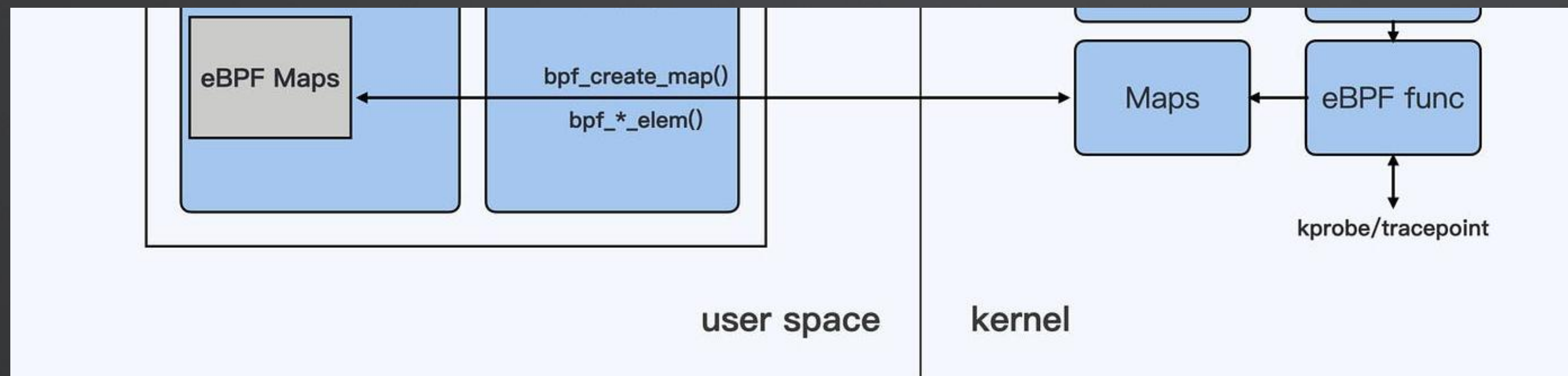


发生了什么（数据交互）？

1. eBPF Maps

1. 借助 Maps 在用户态中读取内核数据 (`bpf_create_map()`)
2. 也可以将用户数据传递给内核中的 eBPF 程序

2. Python 层调用相关接口（如 `bpf_map_update_elem()`）来操作这些 Map。



更复杂的例子

1. prog= : 定义了一段 eBPF 程序，在稍后引用

1. 函数名为 hello，未声明绑定到某个 Hook

2. 加载 eBPF 程序

3. 挂载 eBPF 程序到 kprobe

1. get_syscall_fnname BCC 辅助方法，用于获取指定系统调用的实际名称 (Clone)
2. fn_name="hello": 指定 eBPF 程序中要运行的函数名称
3. 当某个进程调用 clone (即创建新进程或线程) 时，内核会触发 kprobe，运行挂载的 eBPF 函数 hello

4. b.trace_fields()

1. 从内核的 trace_pipe 管道获取日志，并解析为多个字段

```
bcc.py
1  from bcc import BPF
2  from bcc.utils import printb
3
4  # define BPF program
5  prog = """
6  int hello(void *ctx) {
7      bpf_trace_printk("Hello, World!\\n");
8      return 0;
9  }
10 """
11
12 # load BPF program
13 b = BPF(text=prog)
14 b.attach_kprobe(event=b.get_syscall_fnname("clone"), fn_name="hello")
15
16 # header
17 print("%-18s %-16s %-6s %s" % ("TIME(s)", "COMM", "PID", "MESSAGE"))
18
19 # format output
20 while 1:
21     try:
22         (task, pid, cpu, flags, ts, msg) = b.trace_fields()
23     except ValueError:
24         continue
25     except KeyboardInterrupt:
26         exit()
27     printb(b"%-18.9f %-16s %-6d %s" % (ts, task, pid, msg))
```


实战：借助 eBPF 获取 HTTP 请求延迟

- 创建腾讯云虚拟机实验环境
- 写一个 Golang HTTP Server
- 编写 BCC 代码

```
○ ubuntu@loki-0:~$ ./golang_server
2024/11/28 22:05:03 Starting server on :8080
2024/11/28 22:06:04 Received request: method=GET, path=/
2024/11/28 22:06:59 Received request: method=GET, path=/test
```

```
● ubuntu@loki-0:~$ curl localhost:8080/
Hello, you've requested: / with method: GET
```

```
[09:40:20] PID: 576602, COMM: golang_server, Latency: 0.03 ms
[09:40:21] PID: 576602, COMM: golang_server, Latency: 512.14 ms
[09:40:21] PID: 576602, COMM: golang_server, Latency: 0.04 ms
[09:45:38] PID: 576602, COMM: golang_server, Latency: 0.03 ms
[10:32:48] PID: 576602, COMM: golang_server, Latency: 2825930.98 ms
[10:32:49] PID: 576602, COMM: golang_server, Latency: 521.84 ms
[10:32:49] PID: 576602, COMM: golang_server, Latency: 0.04 ms
[10:32:50] PID: 576602, COMM: golang_server, Latency: 0.04 ms
[10:32:50] PID: 576602, COMM: golang_server, Latency: 0.02 ms
```

2. eBPF + Beyla 零侵入 Metrics 和 Tracing

Beyla 简介

Grafana Beyla 是一款基于 eBPF 的应用程序自动检测工具，它使用 eBPF 自动检查应用程序和操作系统网络层，并能够以零侵入的方式捕获与 Web 事务和 Linux HTTP/S 和 gRPC 服务的速率错误持续时间 (RED) 指标相关的跟踪跨度。

Beyla 功能

Beyla 提供以下功能：

- 支持多种语言的零侵入集成，例如：Go、C/C++、Rust、Python、Ruby、Java（包括 GraalVM Native）、NodeJS、.NET 等
- 以 OpenTelemetry 格式和原生 Prometheus 指标导出数据
- Go 服务的分布式跟踪
- 可在任何 Linux 环境中运行
- 监听 Kubernetes API，使用 Pod 和服务元数据来修饰指标和跟踪

Beyla 要求

- Linux \geq 5.8 或更高版本
- 启用了 eBPF
- 要检测 Go 程序，至少使用 Go 1.17 进行编译

Beyla Metrics

RED Method:

- RED 指标由目前 Grafana 的 CTO Tom Wilkie 提出
- 重点监控速度（每秒请求数）、错误（失败请求数）和持续时间（请求所花的时间）
- Rate (the number of requests per second)
- Errors (the number of those requests that are failing)
- Duration (the amount of time those requests take)

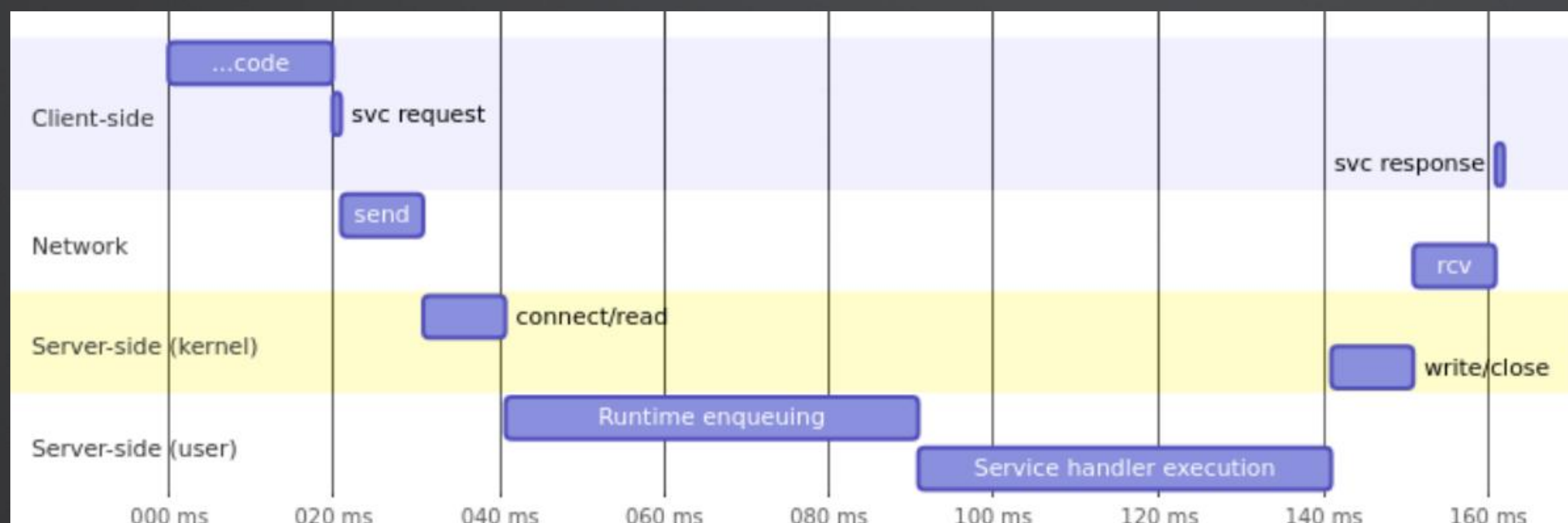
对比 USE Method

USE Method:

- USE 指标旨在监控资源利用率、饱和度和错误
- Utilization (% time that the resource was busy)
- Saturation (amount of work resource has to do, often queue length)
- Errors (count of error events)

Beyla 总请求时间指标

- 相比较传统服务端的自己记录请求耗时更加精确
- eBPF 记录请求耗时是从 Kernel TCP 建立连接到 write/close 的耗时
- 包括 server handler execution 的过程（服务端自己记录请求耗时）
- 在高负载情况下，请求可能在内部队列等待，耗时不容忽视



Beyla 部署方式

以 Sidecar 方式部署:

1. 监控特定的服务, 而不是集群的所有服务
2. 需启用 shareProcessNamespace 在容器之间共享 Pod 进程
3. 需要启用特权模式 securityContext.privileged: true

```
sidecar.yml
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: app
5  spec:
6    .....
7    spec:
8      # Required so the sidecar instrument tool can access the service process
9      shareProcessNamespace: true
10     serviceAccountName: beyla # required if you want kubernetes metadata
11     decoration
12     containers:
13       # Container for the instrumented service
14       - name: app
15       # Sidecar container with Beyla - the eBPF auto-instrumentation tool
16       - name: beyla
17         image: grafana/beyla:latest
18         securityContext: # Privileges are required to install the eBPF
19         probes
20         privileged: true
21         env:
22           # The internal port of the goblog application container
23           - name: BEYLA_OPEN_PORT
24             value: "8443"
25           - name: OTEL_EXPORTER_OTLP_ENDPOINT
26             value: "http://grafana-alloy:4318"
27           # required if you want kubernetes metadata decoration
28           - name: BEYLA_KUBE_METADATA_ENABLE
29             value: "true"
```


Beyla 部署方式

以 Daemonset 方式部署:

1. 监控集群所有服务
2. 需启用 hostPID: true 以便访问主机的所有进程
3. 可以用来检测 Daemonset

```
daemonset.yml
1  ---
2  apiVersion: apps/v1
3  kind: DaemonSet
4  metadata:
5    name: beyla
6  spec:
7    .....
8    spec:
9      hostPID: true # Required to access the processes on the host
10     serviceAccountName: beyla # required if you want kubernetes metadata
11     decoration
12     containers:
13       - name: autoinstrument
14         image: grafana/beyla:latest
15         securityContext:
16           privileged: true
17         env:
18           # Select the executable by its name instead of BEYLA_OPEN_PORT
19           - name: BEYLA_EXECUTABLE_NAME
20             value: "goblog"
21           - name: OTEL_EXPORTER_OTLP_ENDPOINT
22             value: "http://grafana-alloy:4318"
23           # required if you want kubernetes metadata decoration
24           - name: BEYLA_KUBE_METADATA_ENABLE
25             value: "true"
```

Beyla 实战

1. 开通 CVM 虚拟机，并安装 K3s 集群

2. 安装 kube-prometheus-stack (采集 Metrics)

1. `helm upgrade --install prometheus-stack prometheus-community/kube-prometheus-stack --values=kube-prom-values.yaml -n prometheus --create-namespace`

3. 安装 Tempo (采集 Tracing)

1. `helm upgrade --install tempo grafana/tempo-distributed -n tracing -f tempo-values.yaml --create-namespace`

4. 部署示例应用

1. `kubectl apply -f app.yaml`

Beyla 实战

1. 编写 Beyla configmap.yaml 配置文件

1. 将 tracing 导出至 tempo
2. 将 metrics 导出至 prometheus

2. 以 Daemonset 的方式部署 Beyla

1. `kubectl apply -f configmap.yaml,daemonset.yaml`

3. 部署 PodMonitor, 让 Prometheus 能够从 Beyla 获取监控指标

1. `kubectl apply -f pod-monitor.yaml`

4. 访问 Grafana, 并建立 Dashboard

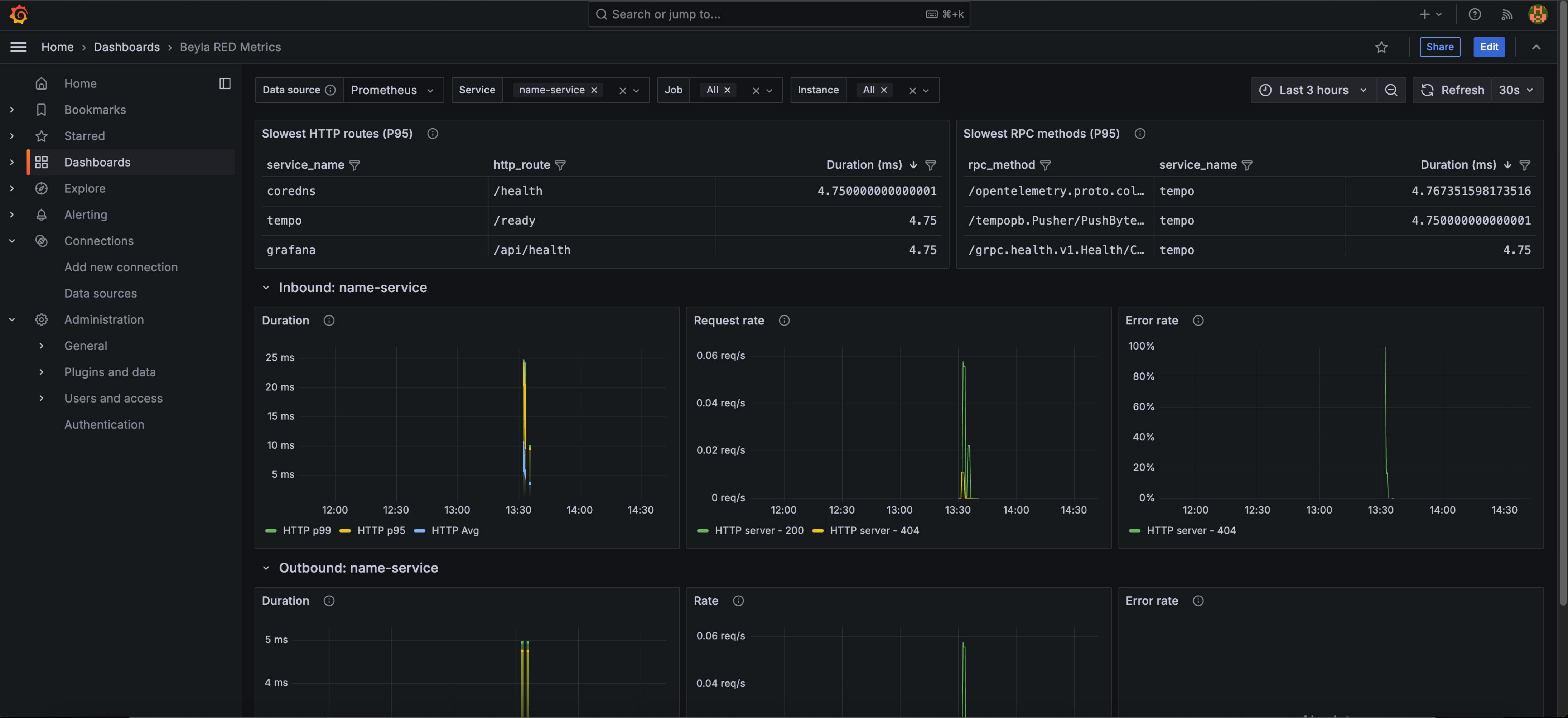
1. `kubectl port-forward svc/prometheus-stack-grafana -n prometheus 3000:80`
2. 导入 dashboard ID: 19923


5. 访问示例应用

1. `kubectl port-forward svc/frontend 8000:7777`, `http://127.0.0.1:8000/greeting`, 以便生成 Metrics 和 Tracing


极客时间训练营

Beyla 效果-Metrics



 极客时间 训练营




Beyla 效果-Traces



Q Search or jump to...

+

▼



Home > Explore > Tempo

Home

Bookmarks

Starred

Dashboards

Explore

Alerting

Connections

Add new connection

Data sources

Administration

General

Plugins and data

Users and access

Authentication

Outline

Tempo

Queries

Table

Split

Add

Last 6 hours

Run query

Options

Limit: 20

Spans Limit: 3

Table Format: Traces

Step: auto

Streaming: Disabled

Add query

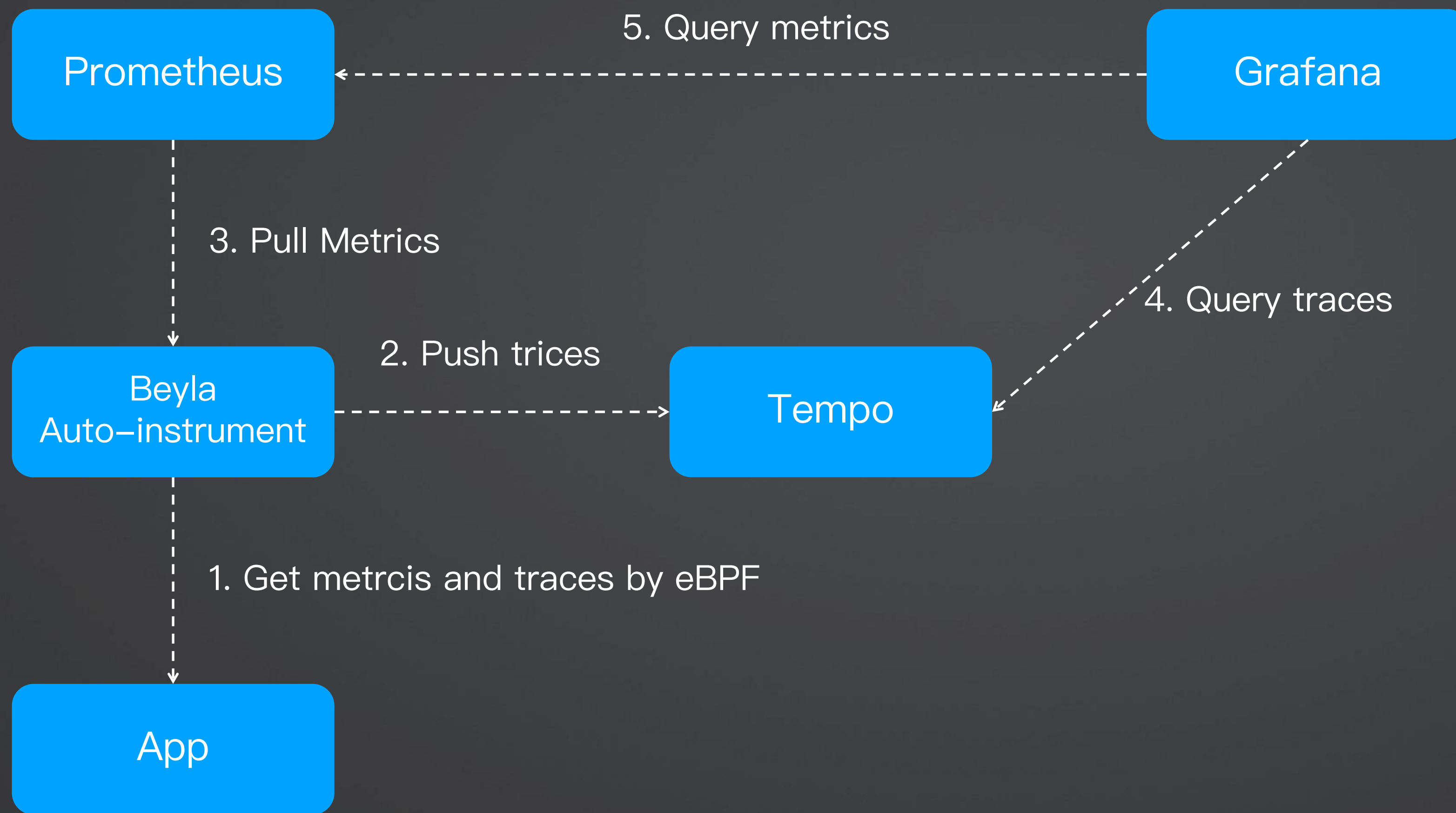
Query history

Query inspector

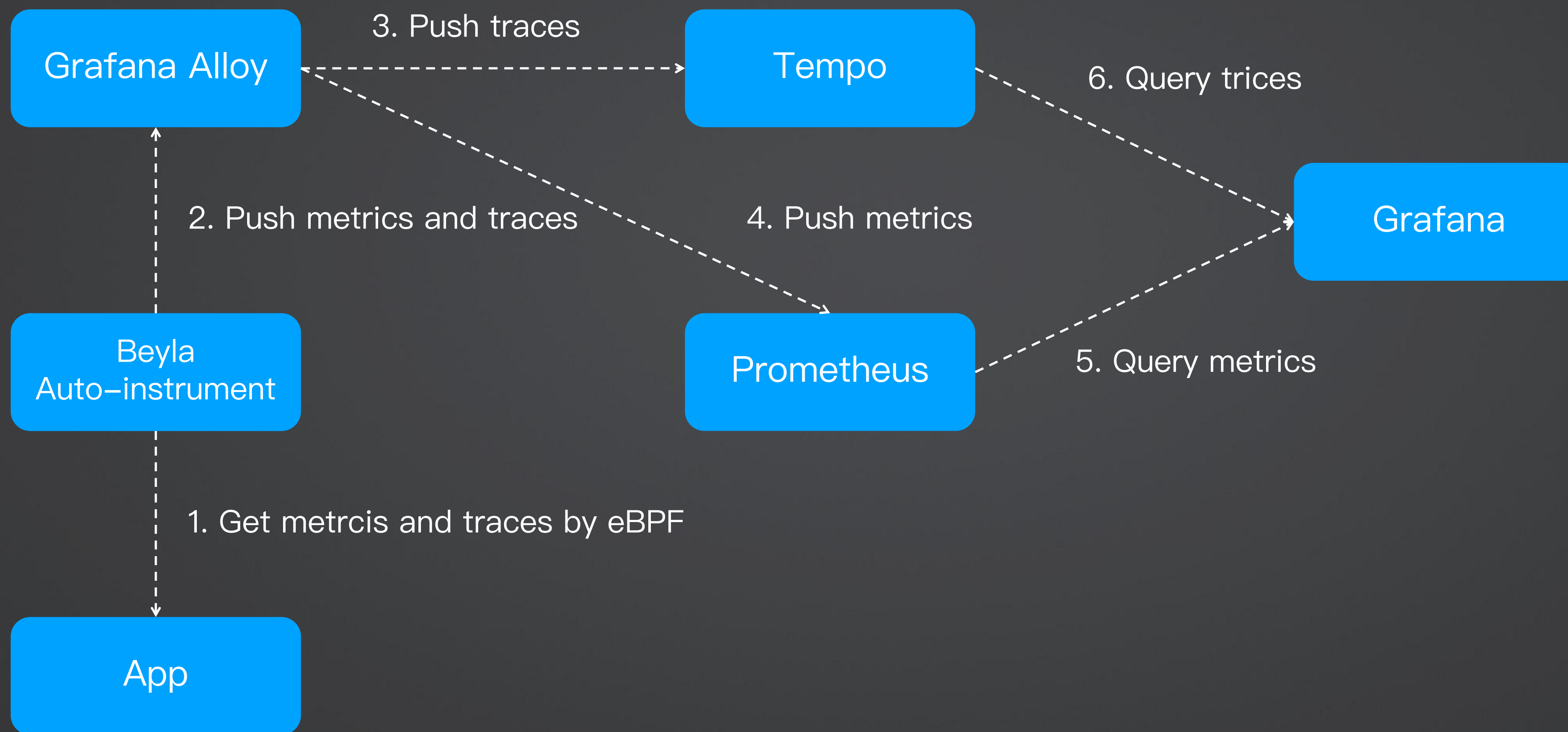
Table - Traces

Trace ID	Start time	Service	Name	Duration
> 10dd9cb87ec771b0b7e...	2024-11-30 14:27:02.364	tempo	/tempopb.MetricsGenerator/PushSpans	1 ms
> 22c7ccff2791cc948a1b...	2024-11-30 14:27:02.338	tempo	/tempopb.MetricsGenerator/PushSpans	1 ms
> 51780d00f3be49d193b...	2024-11-30 14:27:01.785	unknown_service:beyla	exporter/metrics/beyla/traces	2 ms
> fc38f5c4b174dd7f2e96...	2024-11-30 14:27:01.785	tempo	/tempopb.MetricsGenerator/PushSpans	1 ms
> 2696623f534f87b020d...	2024-11-30 14:27:01.778	unknown_service:beyla	exporter/metrics/beyla/traces	3 ms
> b2de44ba6bf841caea9...	2024-11-30 14:27:00.893	unknown_service:beyla	exporter/metrics/beyla/traces	2 ms
> 913070971f0e8a741a0c...	2024-11-30 14:26:59.889	tempo	/tempopb.MetricsGenerator/PushSpans	<1ms
> d7f8ed938916e3f9db4...	2024-11-30 14:26:57.905	tempo	/opentelemetry.proto.collector.trace.v1.TraceService/Export	1 ms
> 10b58a09f02c784897a...	2024-11-30 14:26:57.901	unknown_service:beyla	exporter/metrics/beyla/traces	2 ms
> e327740fb52dd5c8cbd...	2024-11-30 14:26:57.891	tempo	/opentelemetry.proto.collector.trace.v1.TraceService/Export	1 ms
> b8fe2d51f8e2630498c...	2024-11-30 14:26:57.221	tempo	/opentelemetry.proto.collector.trace.v1.TraceService/Export	1 ms
> d8077dc9a94362ee194...	2024-11-30 14:26:56.686	tempo	/opentelemetry.proto.collector.trace.v1.TraceService/Export	1 ms
> 112baf465d5374082c8...	2024-11-30 14:26:56.683	tempo	/tempopb.MetricsGenerator/PushSpans	<1ms
> 44d818bd5837ec8eae...	2024-11-30 14:26:55.680	unknown_service:beyla	exporter/metrics/beyla/traces	2 ms

Beyla 数据流



Beyla 数据流-Alloy 模式



Beyla Alloy 模式实战

1. 开通 CVM 虚拟机，并安装 K3s 集群

2. 安装 kube-prometheus-stack (采集 Metrics)

1. `helm upgrade --install prometheus-stack prometheus-community/kube-prometheus-stack --values=kube-prom-values.yaml -n prometheus --create-namespace`

3. 安装 Tempo (采集 Tracing)

1. `helm upgrade --install tempo grafana/tempo-distributed -n tracing -f tempo-values.yaml --create-namespace`

4. 部署示例应用

1. `kubectl apply -f alloy/app.yaml`

Beyla Alloy 模式实战

1. 提供 alloy config.alloy 配置文件

1. 将 tracing 导出至 tempo
2. 将 metrics 导出至 prometheus

2. 创建 configmap

1. `kubectl create configmap --namespace alloy alloy-config "--from-file=config.alloy=./config.alloy"`

3. 准备 Alloy values.yaml

1. alloy/values.yaml (hostPID: true, privileged: true)

4. 部署 Alloy: `helm upgrade --namespace alloy alloy grafana/alloy -f values.yaml --install`

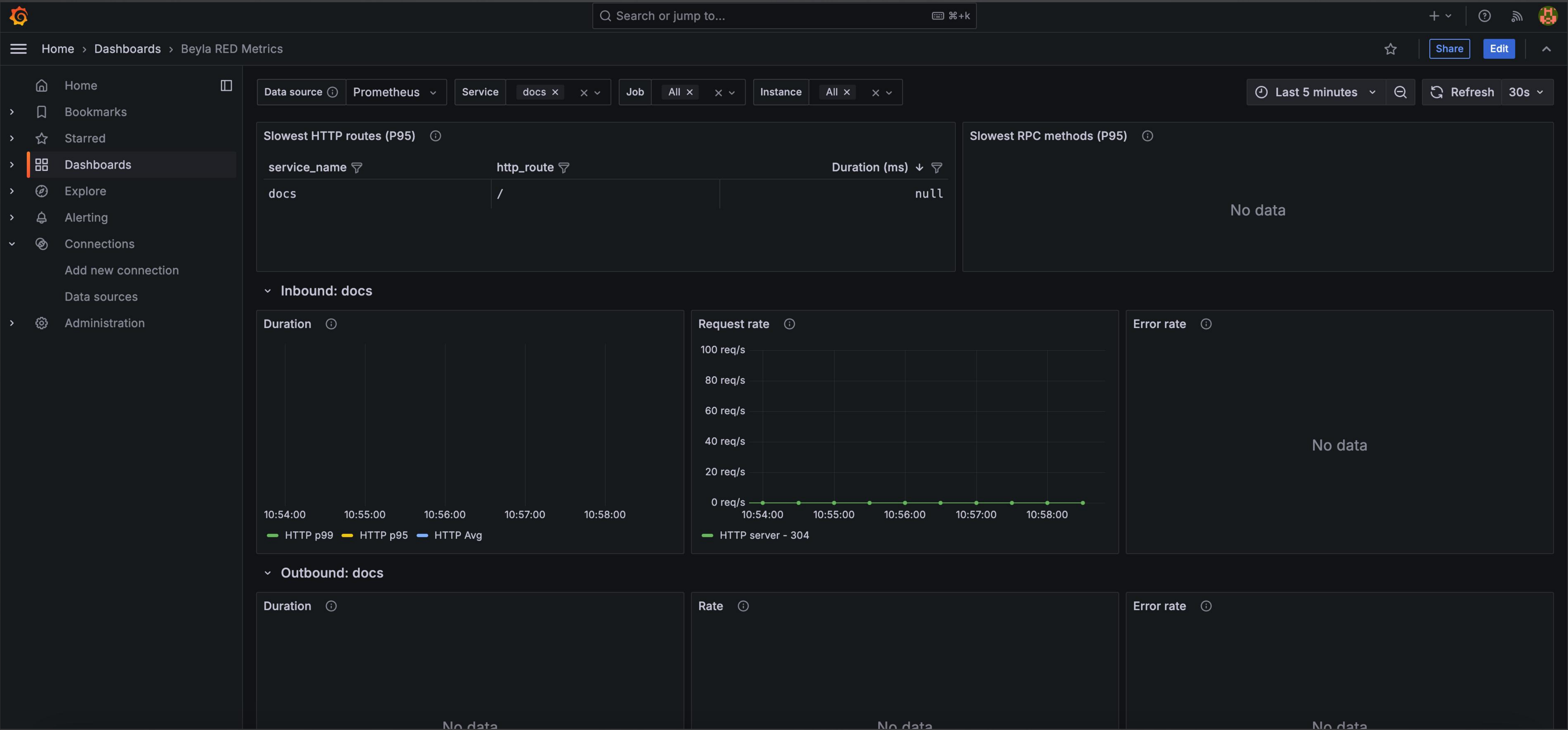
5. 访问 Grafana, 并建立 Dashboard

1. `kubectl port-forward svc/prometheus-stack-grafana -n prometheus 3000:80`
2. 导入 dashboard ID: 19923


6. 访问示例应用

1. `kubectl port-forward svc/docs 8081:80`, `http://127.0.0.1:8001`, 以便生成 Metrics 和 Tracing

Beyla Alloy 模式效果-Metrics



Beyla Alloy 模式效果-Traces



Q Search or jump to...

+ v | ⓘ | 📶 | 🏠

Home > Explore > Tempo | Tempo 🔗 v

Home

Bookmarks

Starred

Dashboards

Explore

Alerting

Connections

Add new connection

Data sources

Administration

Queries

Table

A (Tempo)

Query type Search TraceQL Service Graph Imp

Service Name = Select value

Span Name = Select value

Status = Select value

Duration span > e.g. 100

Tags span Select tag

{ }

> Options Limit: 20 Spans Limit: 3 Table Format: Traces Step:

+ Add query Query history Query inspector

Table - Traces

Trace ID	Start time
> 1cb84ddf2887a0374d7...	2024-12-01 10:47:02.613
> 58ee1a21752d608ca87...	2024-12-01 10:47:01.809
> 706323a963436e0a37...	2024-12-01 10:47:01.566
> 659e5aaf1082a7326a6...	2024-12-01 10:47:01.417
> b3b8f85a7d970d5f509...	2024-12-01 10:47:01.162
> a093a6f805274ddd24...	2024-12-01 10:47:00.395

Tempo

Close Add to dashboard < ⓘ > 🔍 ↺

Queries

Traces

A (Tempo)

Query type Search TraceQL Service Graph Import trace

Build complex queries using TraceQL to select a list of traces. Documentation

1cb84ddf2887a0374d700d0921be8dcc

> Options Limit: 20 Spans Limit: 3 Table Format: Traces Step: auto Streaming: Disabled

+ Add query Query history Query inspector

Trace

docs: GET / 130.75µs

2024-12-01 10:47:02.613 GET 304 /

> Span Filters ⓘ 1 spans ⓘ Prev Next

0µs 32.69µs 65.38µs 98.06µs 130.75µs

Service & Operat... > > > || 0µs 32.69µs 65.38µs 98.06µs130.75µs

docs GET / (130.64µs)

3. 借助 eBPF、Cilium、Hubble 实现零侵入可观测性

Cilium 简介

Cilium 是一个用于在云原生环境中实现高性能、安全的网络和服务互联的开源项目，专注于容器与微服务之间的网络连接，它以 eBPF 技术为核心，提供了透明的网络安全和高效的流量管理能力。

Cilium 核心功能

- 网络连接 (Networking)

- Cilium 提供了高性能的容器网络连接，支持动态负载均衡、服务发现、网络多租户等功能
- 支持 Kubernetes 的 CNI（容器网络接口）规范，是一种更现代化、更高性能的网络解决方案

- 网络安全 (Network Security)

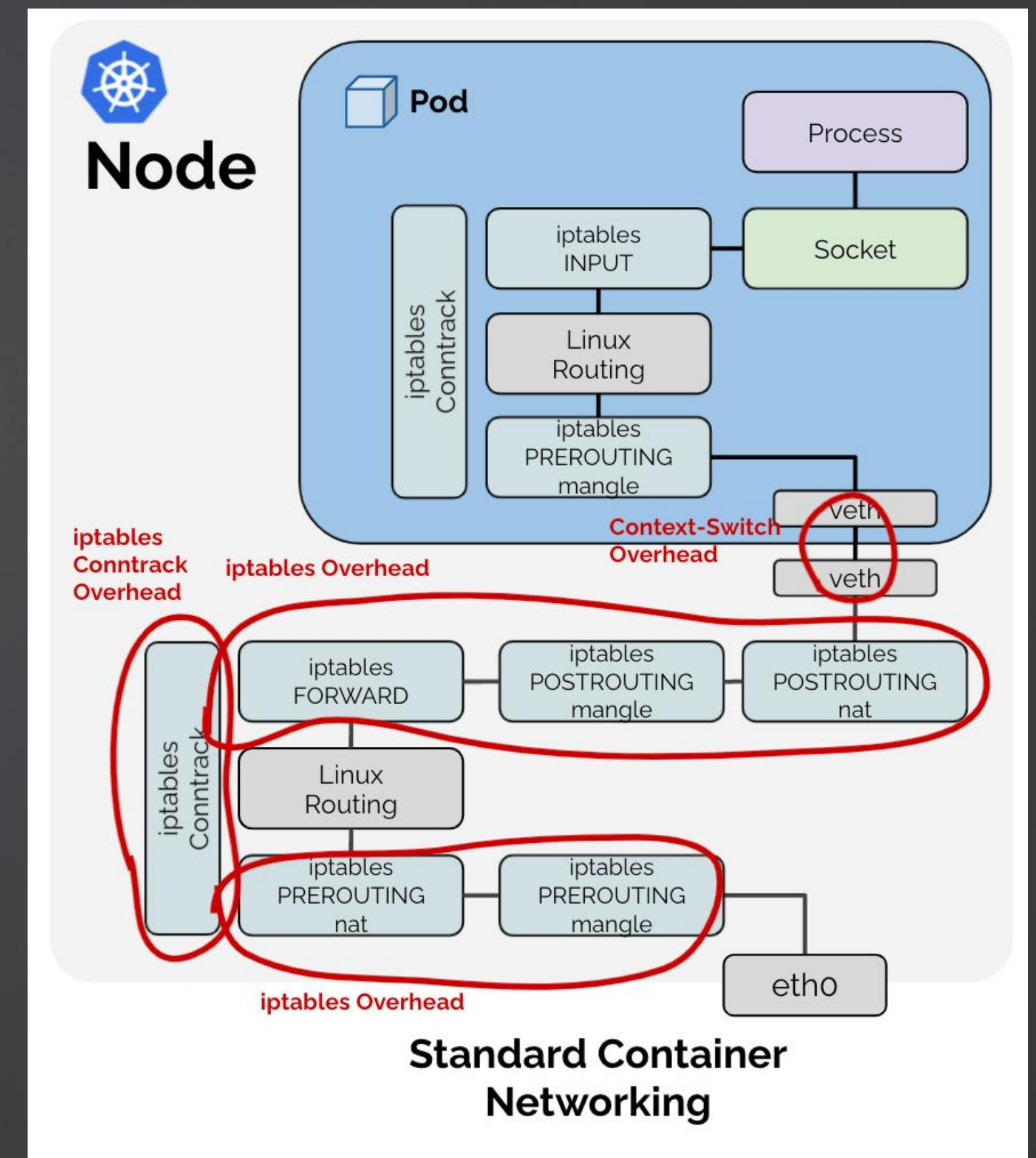
- Cilium 使用基于 eBPF 的流量过滤器，实现细粒度网络安全策略 (Network Policies)
- 提供 L3/L4 (IP层、防火墙) 和 L7 (应用层, 如 HTTP、gRPC) 的安全策略
- 可以微隔离 (Microsegmentation) 应用之间的通信，提升集群内服务之间的安全性

- 可观察性 (Observability)

- Cilium 允许用户在网络和服务层面进行流量的实时观测
- 提供强大的流量跟踪工具，例如 Hubble（集成了一套高效的分布式跟踪工具，专注于查看 Kubernetes 内部流量）
- 可以通过可视化工具查看集群中网络状态、服务交互等关键细节

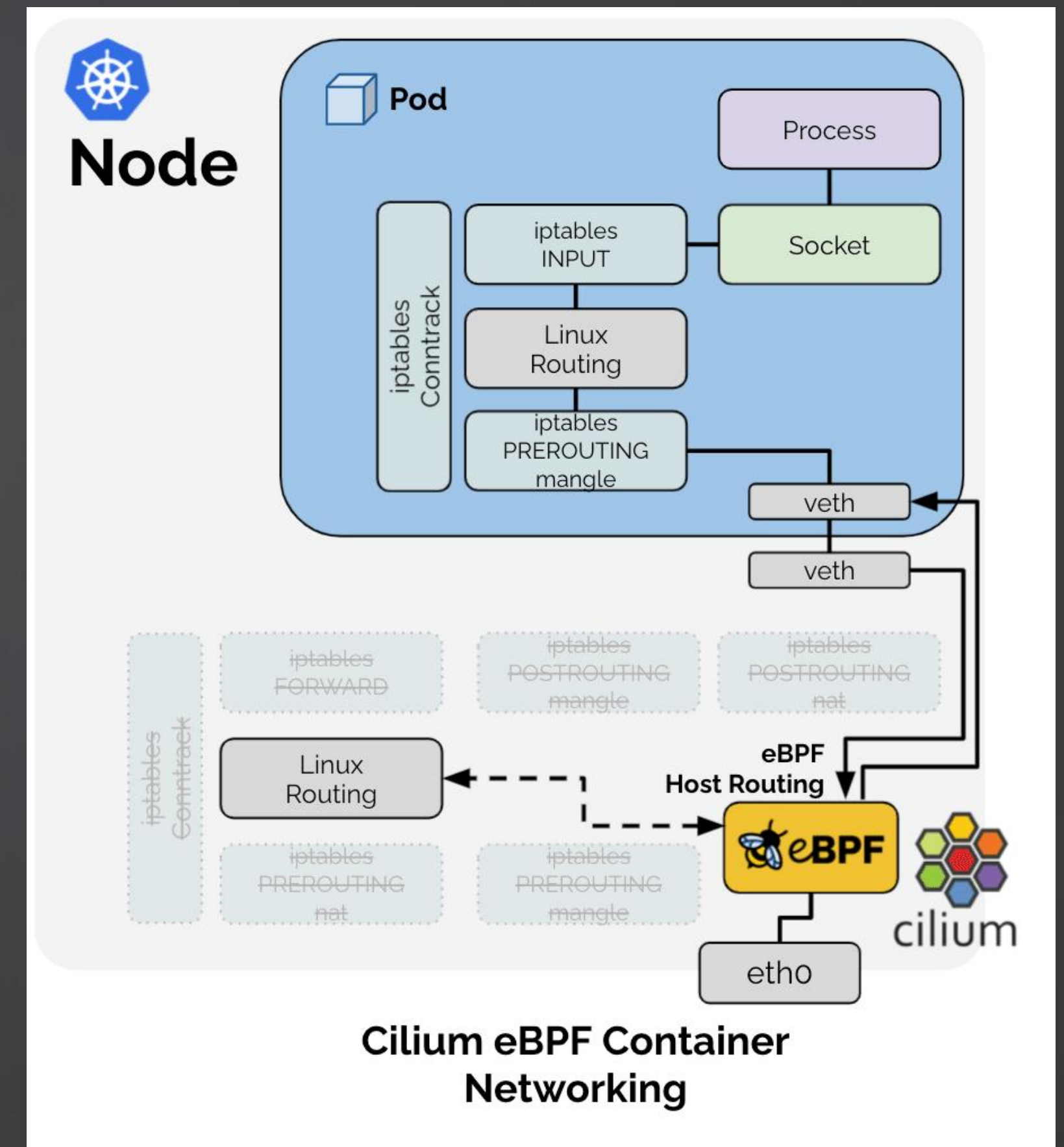
Cilium 取代 iptables

- 依赖 iptables:
 - 网络包的路由和处理使用了 Linux 的 iptables, 经过了多层处理, 包括 PREROUTING、POSTROUTING、FORWARD 等链
- 较高的开销:
 - 每一层 iptables (比如 NAT、链路转发等) 都会增加开销
 - 需要频繁的上下文切换 (从用户态到内核态)
 - 路径复杂, 网络包需要多次流经内核的数据处理模块
- 问题:
 - 性能瓶颈: 随着 iptables 规则增加或者流量增大, 性能会显著下降
 - 路由路径长, 增加了时延



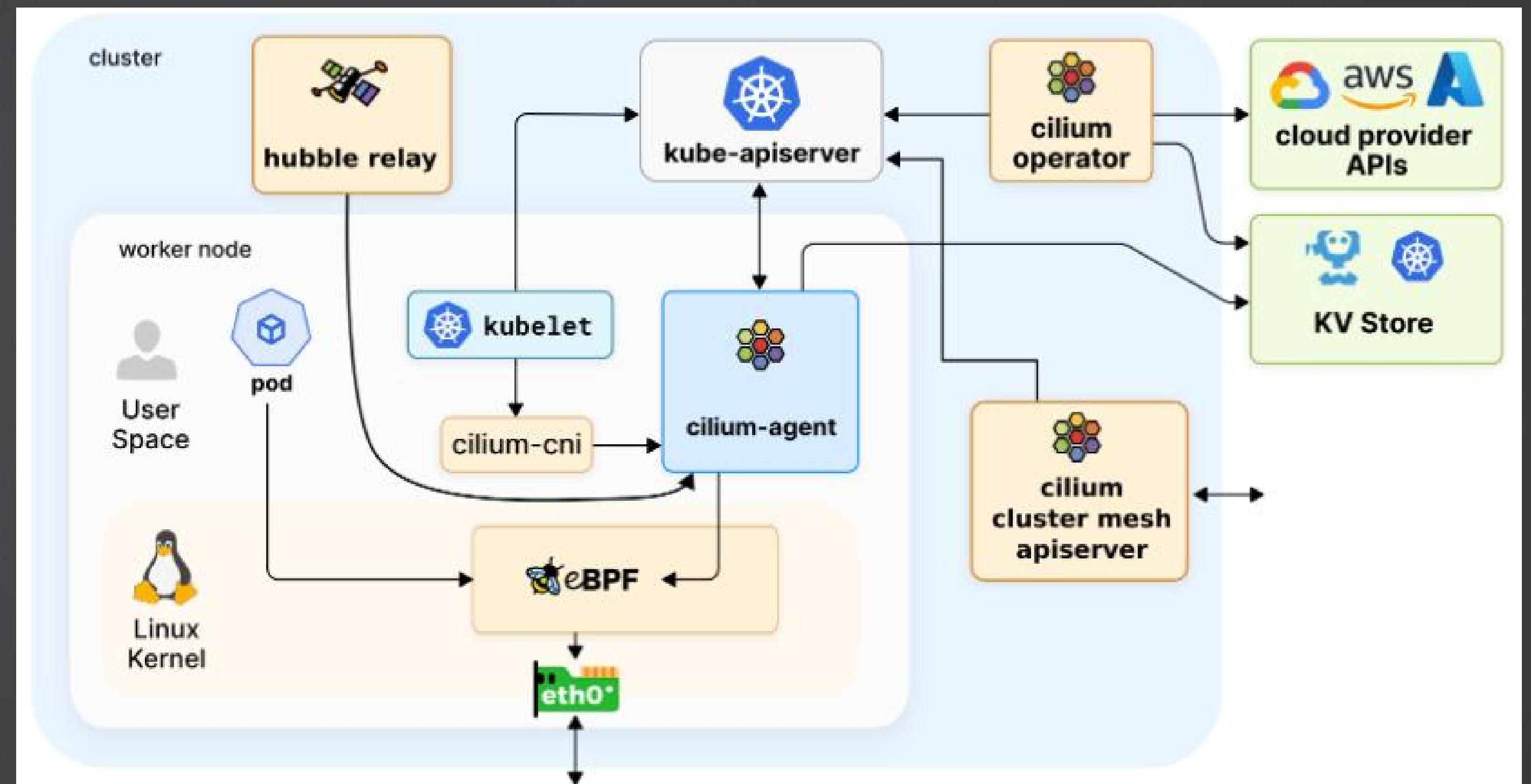
Cilium 取代 iptables

- eBPF:
 - 通过 eBPF 技术，在内核中直接运行用户定义的程序来进行高效的数据包处理
- 简化了流量路径：
 - 使用 eBPF 替代了复杂的 iptables 链
 - 数据包的处理可以直接在内核中完成，而无需通过多个 iptables 表的跳转
- 优势：
 - 性能提升：减少了 iptables 和 Conntrack 的开销
 - 灵活性：eBPF 可以动态加载程序，适应复杂的网络场景
 - 更低的延迟，因为数据包不需要经过冗长的转发路径



Cilium 架构

- Cilium Operator: Cilium 管理平面
- Cilium Agent: 以 Daemonset 方式运行, 主要负责:
 - 与 K8s API 交互, 同步集群状态
 - 与 Kernel 交互, 动态加载 eBPF 程序
- Cilium CNI Plugin
 - 配置节点 CNI 以使用 Cilium 网络插件
- Hubble
 - 通过 gRPC 的方式和 Agent 进行通信, 提供集群级可观测性



Hubble 简介

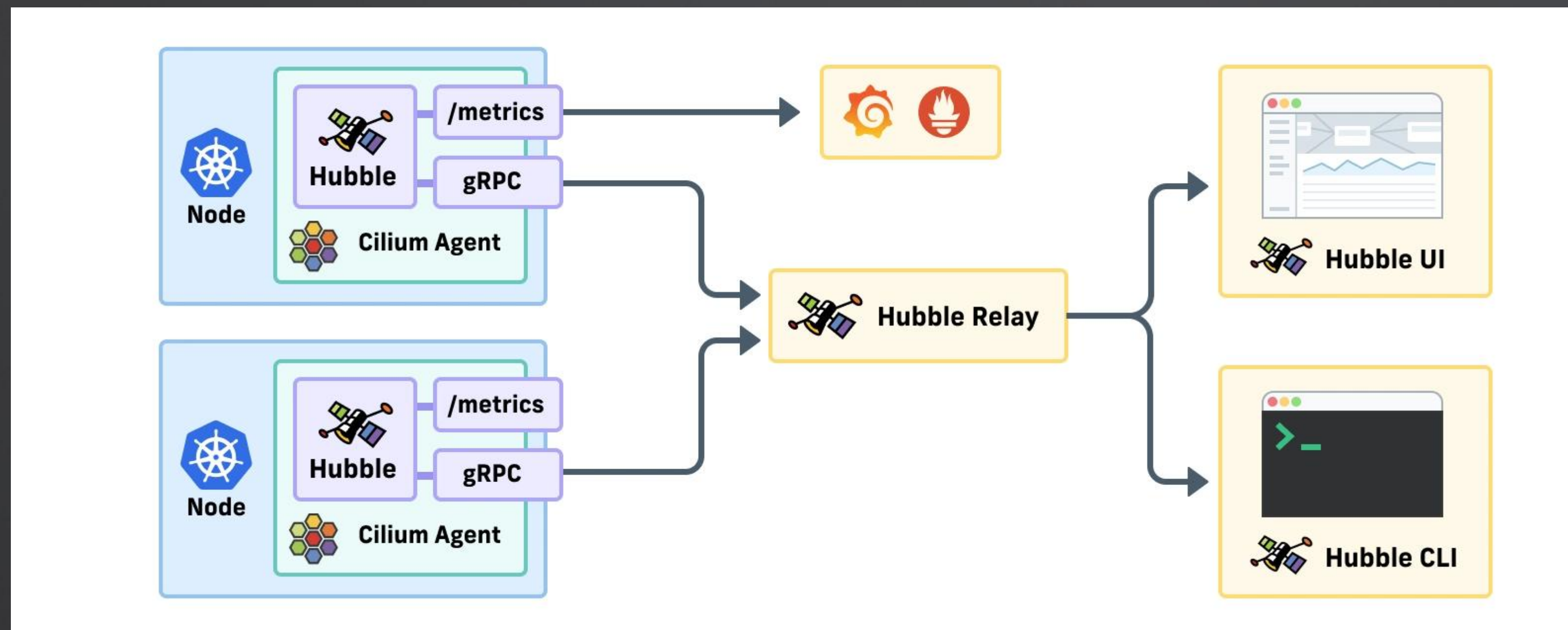
Hubble 是一个分布式的网络和安全可观察性平台。它建立在 Cilium 和 eBPF 之上，能够以透明的方式分析服务流量以及网络基础设施的通信。

Hubble 核心功能

- 服务依赖关系和通信图
 - 哪些服务正在相互通信以及服务依赖关系图
 - 正在进行哪些请求调用，例如 HTTP 和 GRPC
- 网络监控和警报
 - 是否有任何网络通信失败？是第 4 层 (TCP) 还是第 7 层 (HTTP) 上中断？
 - 过去 5 分钟内哪些服务遇到过 DNS 解析问题？
- 应用程序监控
 - 特定服务或所有集群的 5xx 或 4xx HTTP 响应代码的发生率是多少？
 - P99、P95？服务之间的通信延迟是多少？

Hubble 架构

- Hubble 运行在每一个 Node 节点，跟 Cilium Agent 协同工作，并从 eBPF Maps 中提取网络监控数据
- Hubble 对外暴露 gRPC 服务接口，允许其他组件订阅这些网络元数据，为其他工具例如 Hubble UI 提供数据支持
- Hubble 提供暴露的 /metrics 接口，这些数据可以直接被 Prometheus 拉取，用于监控和性能分析



Cilium + Hubble 实战

- 使用特殊参数安装 K3s
 - `--flannel-backend=none`
 - `--disable-network-policy`
- 安装 cilium cli:
 - `curl -L --fail --remote-name-all https://github.com/cilium/cilium-cli/releases/download/v0.16.20/cilium-darwin-arm64.tar.gz`
 - `sudo tar xzvfC cilium-darwin-arm64.tar.gz /usr/local/bin`
- 安装 cilium: `cilium install --version 1.16.4`
 - 检查安装状态: `cilium status --wait`

安装示例应用

- `kubectl create -f https://raw.githubusercontent.com/cilium/cilium/1.16.4/examples/minikube/http-sw-app.yaml`

```
> kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
deathstar-7599b65f55-vl9ht         1/1     Running   0           16s
deathstar-7599b65f55-xn4mb         1/1     Running   0           16s
tiefighter                          1/1     Running   0           16s
xwing                              1/1     Running   0           16s
```

- 安装 Hubble
 - `cilium hubble enable`
 - 检查安装状态: `cilium status`
- 安装 Hubble Cli
 - `curl -L --fail --remote-name-all https://github.com/cilium/hubble/releases/download/v1.16.4/hubble-darwin-arm64.tar.gz`
 - `sudo tar xzvfC hubble-darwin-arm64.tar.gz /usr/local/bin`

输出 Hubble eBPF 可观测日志

- cilium hubble port-forward&
- hubble status
- hubble observe

```
> hubble observe
Nov 30 12:27:00.053: 10.0.0.61:57162 (host) -> kube-system/metrics-server-854c559bd-d28qq:10250 (ID:12719)
o-endpoint FORWARDED (TCP Flags: SYN)
Nov 30 12:27:00.053: 10.0.0.61:57162 (host) <- kube-system/metrics-server-854c559bd-d28qq:10250 (ID:12719)
o-stack FORWARDED (TCP Flags: SYN, ACK)
Nov 30 12:27:00.053: 10.0.0.61:57162 (host) -> kube-system/metrics-server-854c559bd-d28qq:10250 (ID:12719)
o-endpoint FORWARDED (TCP Flags: ACK)
Nov 30 12:27:00.053: 10.0.0.61:57162 (host) -> kube-system/metrics-server-854c559bd-d28qq:10250 (ID:12719)
o-endpoint FORWARDED (TCP Flags: ACK, PSH)
Nov 30 12:27:00.056: 10.0.0.61:57162 (host) <- kube-system/metrics-server-854c559bd-d28qq:10250 (ID:12719)
o-stack FORWARDED (TCP Flags: ACK, PSH)
```

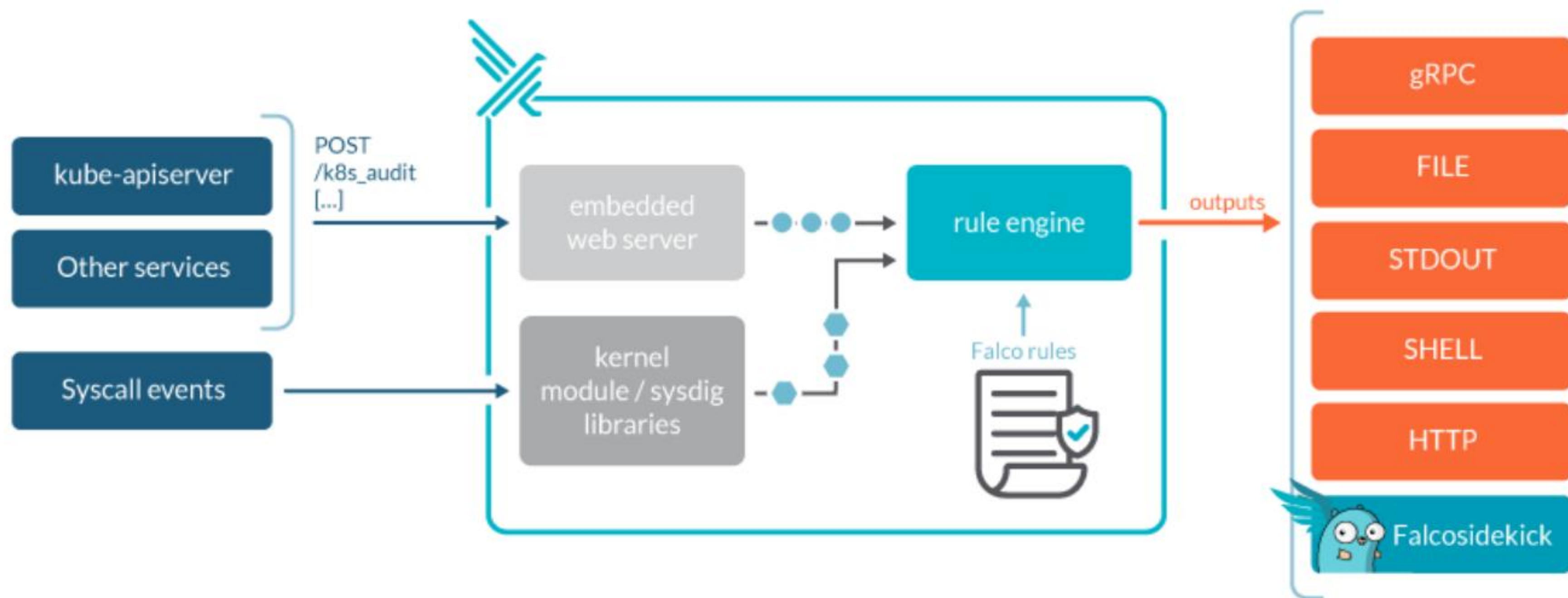
启用 Hubble UI

- 安装 UI: `cilium hubble enable --ui`
- 端口转发并打开 UI: `cilium hubble ui`
- 生成访问数据:
 - `kubectl exec xwing -- curl -s -XPOST deathstar.default.svc.cluster.local/v1/request-landing`
 - `kubectl exec tiefighter -- curl -s -XPOST deathstar.default.svc.cluster.local/v1/request-landing`



4. 借助 eBPF+Flaco 实时监测 K8s 安全威胁

Flaco



Falco 事件采集原理

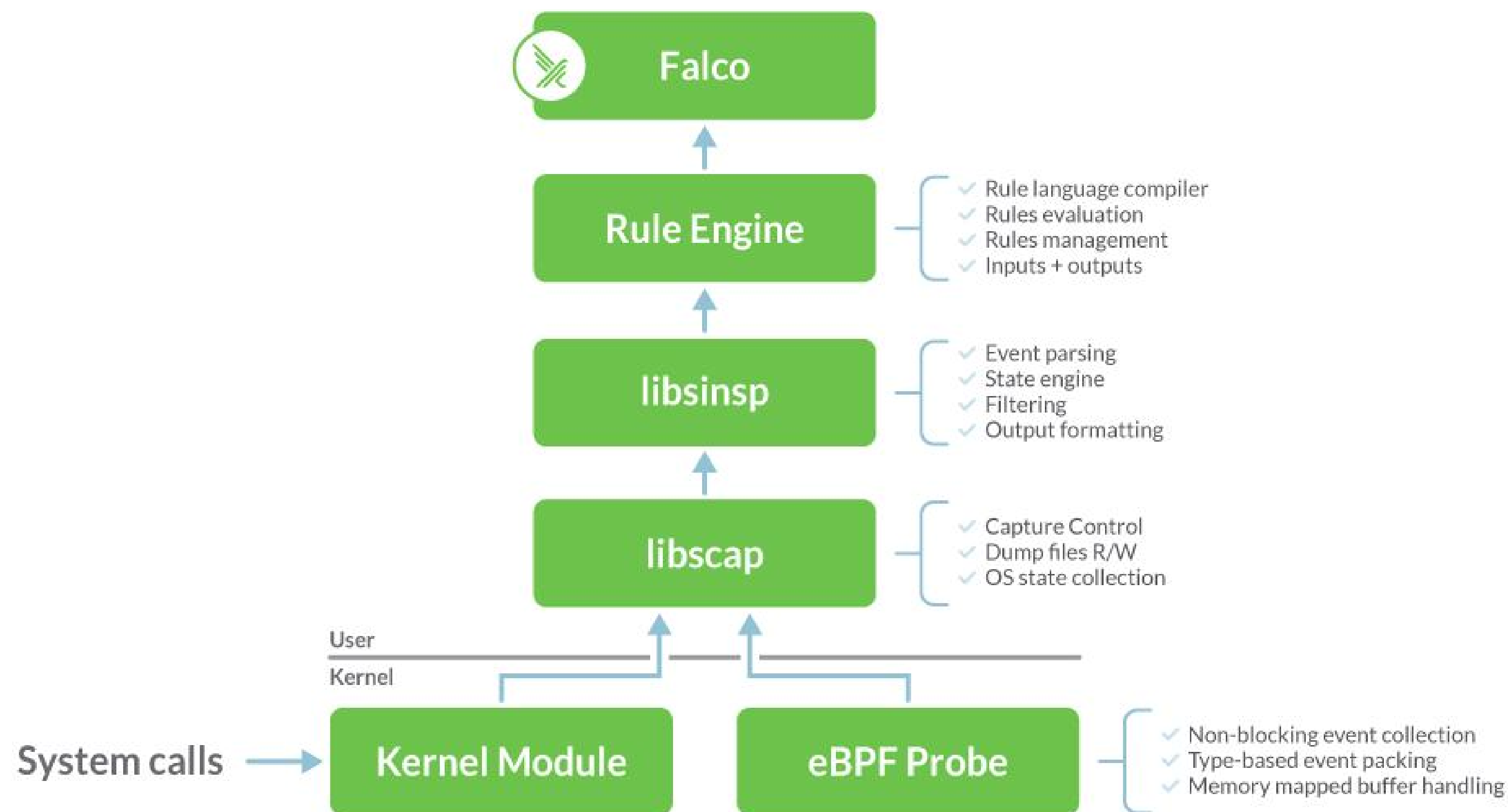
1. (默认) 基于 libscap 和 libsinsp C++ 库构建的内核模块, 从

Linux Ring Buffer 获取系统事件

2. 通过 eBPF 获取事件

使用 eBPF 获取事件要求 Linux 内核

版本 > 4.16



实战：安装 Falco 并识别安全威胁事件

1. 添加 Helm 仓库

1. `helm repo add falcosecurity https://falcosecurity.github.io/charts && helm repo update`
2. `helm install falco -n falco --set driver.kind=ebpf --set tty=true falcosecurity/falco --set falcosidekick.enabled=true --set falcosidekick.webui.enabled=true --create-namespace`
3. 验证安装：`kubectl get pods -n falco --watch`

2. 运行 alpine 镜像

1. `kubectl run alpine --image alpine -- sh -c "sleep infinity"`

3. 模拟安全事件（执行一段 shell 命令）

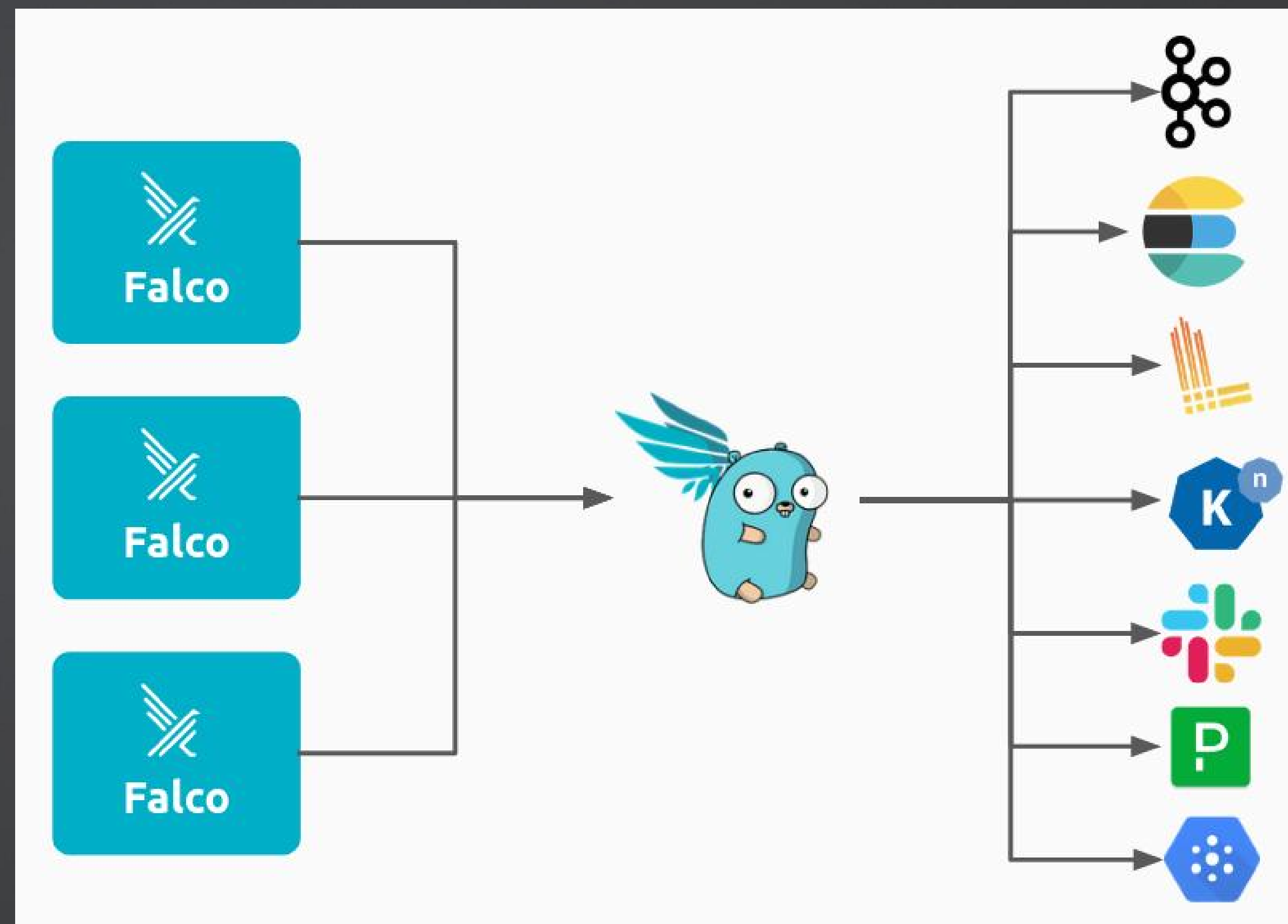
1. `kubectl exec -it alpine -- sh -c "uptime"`

4. 查看 Falco 输出的日志安全威胁日志

1. `kubectl logs -l app.kubernetes.io/name=falco -n falco -c falco | grep Notice`

Falcosidekick: 将安全事件发送到外部

1. 之前的安装已经启用 Falcosidekick 和 Falcosidekick-ui 组件
2. 访问 Dashboard
3. `kubectl port-forward svc/falco-falcosidekick-ui 2802:2802 -n falco`
 1. admin/admin
4. 集成外部服务:
<https://github.com/falcosecurity/falcosidekick>



建议监控的高危安全事件

1. shell、bash、zsh、sh 命令操作
2. ssh 登录
3. 用户和密码操作：useradd、usermod、passwd、adduser、addgroup.....
4. 提权操作：sudo、su、suexec
5. 导出数据操作：pg_dumpall、mysqldump
6. VPN 操作：openvpn
7. 定时命令：crontab、cron
8. 查看和删除日志
9. Linux Kernel Module 注入检测

Falco Rules 编写

– rule: shell_in_container

desc: notice shell activity within a container

condition: >

evt.type = execve and # <https://falco.org/ml/docs/reference/rules/supported-fields/>

evt.dir = < and

container.id != host and

(proc.name = bash or

proc.name = ksh)

output: >

shell in a container

(user=%user.name container_id=%container.id container_name=%container.name

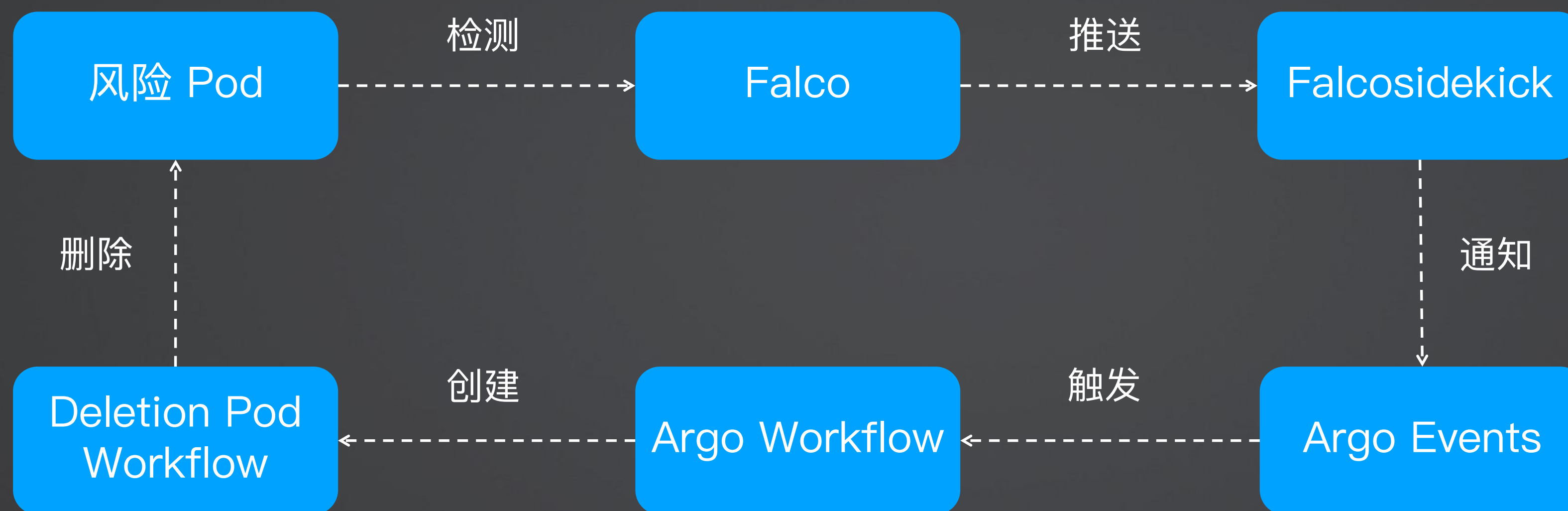
shell=%proc.name parent=%proc.pname cmdline=%proc.cmdline)

priority: WARNING

Falco 默认 Rules

- 默认: https://github.com/falcosecurity/rules/blob/main/rules/falco_rules.yaml
- sandbox rules: https://github.com/falcosecurity/rules/blob/main/rules/falco-sandbox_rules.yaml
- incubating rules: https://github.com/falcosecurity/rules/blob/main/rules/falco-incubating_rules.yaml

Falco + Argo Workflow 自动处理安全事件



THANKS