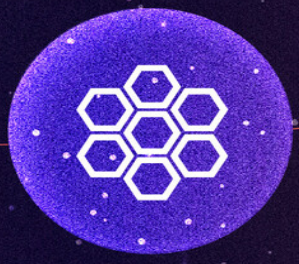
KUBERNETES OBSERVABILITY IN ACTION



By Google Dev Expert
Gerardo López

Kubernetes Observability in Action

Gerardo López

This book is not just a solo endeavor; it is a collaborative creation nurtured by the unwavering support of those closest to my heart. To my lovely wife, Yirian, and my son, Nathaniel, I dedicate these pages. Their belief in my pursuits and their unwavering support have been the wind beneath my wings. In the journey of authoring this book, their encouragement has been my guiding star. I extend my deepest gratitude to the Almighty for instilling in me the passion to create, to learn, and to share. This book is a testament to the divine gift of community building, a journey where every line of code, every lesson shared, is an offering to the greater collective wisdom. To all those who have joined me on this expedition—whether as colleagues, students, or fellow adventurers in the tech realm—thank you. Your presence in this journey is a source of inspiration and a reminder that knowledge grows richer when cultivated together. May this book serve as a beacon, illuminating the path for those who wish to embark on their own odyssey in the vast and ever-expanding universe of software development. Together, let us continue to learn, to create, and to empower.

Table of Contents

About this edition

About the Author

Preface

1: What observability really means in the Kubernetes environment
- Handling the observability's nuances
- The three foundational elements of observability: Metrics, Logging, and Tracing
- The spectrum of Kubernetes observability: difficulties and prospects
- A lot of theory, we need practice: Create a Kubernetes cluster
- Summary

2: Understanding the Cilium Fundamentals
- Cilium components
- eBPF: the key component enabling Cilium's success
- Cilium features: High level overview
- Enhancing Kubernetes Security and Observability with Cilium
- Installing Cilium
- Summary

3: Prometheus and Grafana - Empowering Kubertenes Observability
- Prometheus: Unveiling the Power of Metrics
- Grafana: Crafting Visual Narratives for Observability
- Merge your data
- The Symbiotic Relationship
- Installing Cilium and Hubble
- Installing Prometheus and Grafana
- Summary

4: Service Map & Hubble UI

Service Map
Hubble UI
Installation
Using hubble UI
Summary

5: Log Aggregation with Grafana Loki
Log Collection in Kubertenes
Log Formats in Kubertenes
Grafana Loki Overview
How does Grafana Loki work?
Instalation
Configure Data Source
Visualize and Analyze
Summary

6: Navigating the Trace Path in Kubertenes with Grafana Tempo
What are traces and their importance?
What is distributed tracing in Kubertenes?
How distributed tracing works
How Grafana Tempo works
Summary

7: Advancing your observability journey
Future Challenges and Advanced Topics
Happy observing

About this edition

This edition was published in June 2024.

“Kubernetes Observability in Action” is an essential guide for professionals looking to enhance their understanding and capabilities in monitoring and troubleshooting Kubernetes environments. This book delves into the fundamental techniques and tools necessary for achieving deep observability in Kubernetes clusters, leveraging the powerful features of eBPF, Cilium, and Grafana.

Take part in a thorough learning process that will help you understand not only the details of Kubernetes observability but also how to implement an initial observability platform. Use the combined power of K8s, Cilium, Grafana, Tempo, and Loki according to the guidance of a skilled professional to build a strong observability infrastructure.

I hope you can enjoy the book and can start to create amazing observabilities tools.

About the Author

Greetings, fellow explorers of the digital frontier. My name is Gerardo Lopez,I have over 12 years of experience in the realm of software development, I've ventured into the vast expanse of cloud technologies, uncovering the secrets that empower modern applications to soar to new heights.

But my journey extends beyond the lines of code. I am a devoted teacher and content creator, driven by the belief that knowledge is most valuable when shared. Whether standing before a live audience at global events or crafting virtual content for the online realm, my mission is clear: to empower others with the skills and insights that fuel innovation.

As a cloud enthusiast, I've had the honor of participating as a speaker in various events worldwide, discussing the latest trends in cloud technology and sharing the wisdom gathered from the ever-evolving landscape of software development. This book emerges as an extension of my commitment to knowledge sharing, encapsulating years of experience, insights, and a passion for community building.

I am a fan of soccer, movies and seeing new places. I regularly go out to play soccer with friends, travel with my wife and son to different destinations and of course try local food.

gelopfalcon

Preface

“Kubernetes Observability in Action” is not just a book; it’s an exploration into the realms of Kubernetes, eBPF, and the powerful synergy of Cilium and Grafana. In these pages, readers will embark on a journey that transcends traditional observability, diving into the intricate layers of containerized environments. We’ll unravel the potential of eBPF (Extended Berkeley Packet Filter) as a game-changing technology, unlocking unprecedented visibility into the inner workings of Kubernetes clusters. This book is not just a guide; it’s an invitation to master the art of observability and elevate your understanding of cloud-native landscapes.

Within the digital universe of Kubernetes, achieving deep observability is akin to wielding a cosmic telescope that pierces through the layers of abstraction, revealing the inner workings of microservices and containers. “Kubernetes Observability in Action” empowers readers to harness the full potential of eBPF, a revolutionary technology that extends the traditional packet filtering capabilities of the Linux kernel. We’ll explore the integration of Cilium, a cutting-edge CNI (Container Network Interface), and Grafana, a versatile observability platform, to craft a comprehensive solution for monitoring and troubleshooting Kubernetes environments. As we navigate through practical implementations, readers will emerge equipped with the knowledge to elevate their observability game, ensuring their Kubernetes deployments operate at peak efficiency and resilience. Get ready to delve deep into the cosmic wonders of Kubernetes observability and redefine your approach to managing cloud-native architectures.

1: What observability really means in the Kubernetes environment

In the dynamic universe of Kubernetes, where orchestrating complicated microservices and applications is king, observability is more than just logging and monitoring. It represents an all-encompassing strategy that encompasses a thorough comprehension of all aspects of the behavior, functionality, and well-being of the system. Observability is the keystone that allows engineers and operators to obtain a profound understanding of the complex relationships and interdependencies that shape the operational environment of their Kubernetes clusters within this complex ecosystem.

Handling the observability's nuances

Managing observability in the complex world of Kubernetes requires a thorough comprehension of the many different issues that arise from the dynamic nature of distributed systems orchestration and containerized environments. Workload orchestration, service mesh interactions, and the fleeting nature of container lifecycles are examples of complex issues that require careful consideration and strategic knowledge. To handle these subtleties with efficacy, one must put strong strategies in place for handling changing workloads, guaranteeing smooth service discovery, and coordinating effective microservice-to-microservice communication.

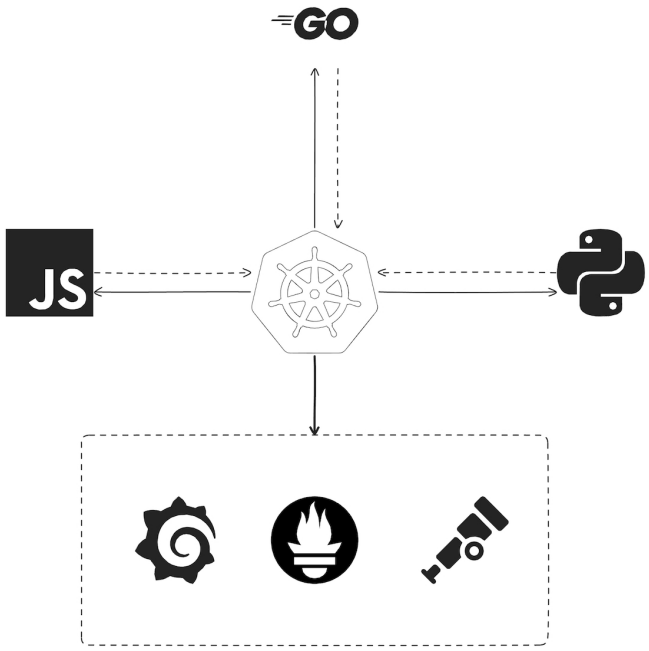
Moreover, the complex aspects of observability require the development of efficient procedures for addressing momentary failures, maximizing resource usage, and reducing possible performance bottlenecks in Kubernetes clusters. Engineers can reduce downtime and improve the overall reliability of Kubernetes applications by minimizing transient failures and guaranteeing service delivery through the implementation of resilient fault-tolerant mechanisms. Furthermore, operators can maximize resource utilization, reduce waste, and maintain optimal performance levels

by optimizing resource allocation through intelligent resource provisioning and efficient workload balancing. This guarantees the smooth operation of Kubernetes clusters under a range of workloads and operational conditions.

In addition, proactively identifying latency problems, streamlining network communication, and fine-tuning application dependencies within the Kubernetes environment are necessary to mitigate potential performance bottlenecks. Through the use of sophisticated profiling tools, engineers are able to identify areas of high latency and optimize important network paths, which facilitates data transfer with minimal interruption and faster response times within the Kubernetes cluster. Furthermore, by implementing effective load balancing mechanisms and intelligent service routing strategies, it is possible to fine-tune application dependencies and create a robust and resilient application ecosystem. This facilitates seamless communication between interconnected microservices and improves the overall performance and reliability of the Kubernetes infrastructure.

Engineers and operators can create a strong and resilient operational framework that not only guarantees the smooth operation of applications but also promotes a culture of continuous improvement and operational excellence in the dynamic and constantly changing Kubernetes landscape by carefully navigating and addressing the complex challenges of observability within Kubernetes.

The three foundational elements of observability: Metrics, Logging, and Tracing



• Kubernetes deploys apps.
• The apps creates logs, traces and metrics in the cluster.
• Other tools use the data created by apps to create dashboards and visualizations that allow knowing and improving the performance of apps running in the cluster.

Metrics

As the front-runner of the observability triad, monitoring is the first line of defense for preserving Kubernetes cluster performance and stability. Engineers are able to closely monitor the health and resource usage of individual pods, services, and nodes by utilizing an extensive suite of monitoring tools and metrics. By keeping a close eye on critical performance metrics like CPU usage, memory usage, and network throughput, operators can quickly spot irregularities, foresee possible bottlenecks, and proactively assign resources to maximize the cluster’s overall reliability and performance.

Managing metrics from containers is crucial for monitoring the health, performance, and resource utilization of your applications in a Kubernetes environment. Here are some popular tools used for managing metrics from containers:

1. **Prometheus:**
    - **Description:** An open-source monitoring and alerting toolkit designed for reliability and scalability. Prometheus is well-suited for Kubernetes environments and provides a flexible query language, powerful alerting, and easy integration with Grafana.
    - **Key Features:**
        - Pull-based model for collecting metrics.
        - Rich query language (PromQL) for analysis.
        - Integrates seamlessly with Grafana.

2. **Grafana:**
    - **Description:** Grafana is an open-source platform for monitoring and observability that works well with Prometheus and other data sources. It allows you to visualize and analyze metrics through customizable dashboards.
    - **Key Features:**
        - Support for various data sources, including Prometheus.
        - Rich visualization options with interactive dashboards.
        - Alerting and notification capabilities.

3. **kube-state-metrics:**
    - **Description:** kube-state-metrics is an add-on service that generates metrics about the state of Kubernetes objects. It exposes cluster-level metrics, such as the number of pods in various states, deployments, and more.
    - **Key Features:**
        - Provides metrics about the state of Kubernetes objects.
        - Facilitates tracking of resource usage and state changes.

4. **Metrics Server:**

- **Description:** A scalable, efficient source of container resource metrics in Kubernetes clusters. Metrics Server is part of the Kubernetes project and is designed to scale horizontally to handle the monitoring needs of large clusters.
- **Key Features:**
  - Exposes resource usage metrics for pods and nodes.
  - Lightweight and designed for scalability.

Logs

The log, which is a notebook that documents each and every incident that happens in the Kubernetes environment, provides a detailed account of the cluster's history of operations. Through the careful documentation of system-generated events, user interactions, and application-specific logs, logging makes it easier to comprehend the subtleties and characteristics that make up the Kubernetes ecosystem. The thorough examination of logs enables engineers to track application errors, identify performance bottlenecks, and look into security breaches. These insights lead to proactive troubleshooting and informed decision-making, which ultimately improves the resilience and integrity of the Kubernetes infrastructure.

Managing logs from containers in Kubernetes is crucial for understanding the behavior of applications, diagnosing issues, and maintaining system health. Here are some popular tools used for managing logs in Kubernetes:

1. **Fluentd:**

   - **Description:** Fluentd is an open-source data collector that allows you to unify data collection and consumption. It's commonly used for log forwarding, aggregation, and normalization.
   - **Key Features:**
     - Extensive plugin ecosystem for various input and output sources.
     - Efficient log processing and forwarding.

2. **Fluent Bit:**

- **Description:** Fluent Bit is a lightweight and fast log processor and forwarder. It's designed to be resource-efficient and is often used in environments where a minimal footprint is crucial.
- **Key Features:**
  - Low resource usage, suitable for edge and IoT environments.
  - Integration with various output backends.

3. **ELK Stack (Elasticsearch, Logstash, Kibana):**

   - **Description:** The ELK Stack is a popular combination of tools for log management and analysis. Elasticsearch stores and indexes logs, Logstash processes and ships logs, and Kibana provides a web interface for searching and visualizing logs.
   - **Key Features:**
     - Powerful search and visualization capabilities with Kibana.
     - Scalable log storage with Elasticsearch.

4. **Prometheus with Loki:**

   - **Description:** Prometheus, known for metrics monitoring, can be combined with Loki for log aggregation. Loki stores logs as a set of labels and is designed to be efficient and cost-effective.
   - **Key Features:**
     - Correlates metrics and logs for comprehensive observability.
     - Efficient storage and querying of log data.

Tracing

Engineers can unravel the complexity of distributed systems architecture with the support of tracing, the thread that runs through Kubernetes and weaves a detailed weaving of interconnectivity. Tracing facilitates a thorough understanding of the dependencies, latencies, and communication patterns that characterize the operational dynamics of the Kubernetes environment by following the flow of requests and interactions between microservices and application components. Tracing acts as a catalyst to improve the overall responsiveness and reliability of the Kubernetes ecosystem by helping to detect latency issues, and optimize service

interdependencies. This allows operators to proactively fine-tune the application stack and provide smooth user experiences.

Managing traces from containers is essential for gaining insights into the performance and interactions of microservices in a distributed environment. Here are some popular tools used for managing traces in containerized applications:

1. **Jaeger:**
    - **Description:** Jaeger is an open-source, end-to-end distributed tracing system. It provides monitoring and troubleshooting capabilities for microservices-based applications by capturing, storing, and visualizing traces.
    - **Key Features:**
        - Distributed context propagation.
        - High-resolution, low-overhead sampling.
        - Integration with various programming languages and frameworks.
2. **Zipkin:**
    - **Description:** Zipkin is an open-source distributed tracing system. It helps troubleshoot latency problems in service architectures and provides a user-friendly web interface for trace visualization.
    - **Key Features:**
        - Collects, stores, and visualizes trace data.
        - Integrates with various languages and frameworks.
        - Supports different storage backends.
3. **OpenTelemetry:**
    - **Description:** OpenTelemetry is an observability framework for cloud-native software. It includes APIs, libraries, agents, instrumentation, and instrumentation standards for distributed tracing, among other observability needs.
    - **Key Features:**
        - Unified instrumentation and data collection.

- Vendor-agnostic and cloud-native.
- Supports multiple programming languages.

The spectrum of Kubernetes observability: difficulties and prospects

Although the term “observability” has gained popularity recently, the ideas behind it are not particularly novel. Developers, IT engineers, and DevOps teams have been integrating logs, metrics, and traces to identify patterns and issues in their applications for years.

However, the fact that traditional observability paradigms cannot be simply lifted and shifted into Kubernetes is what makes life difficult for K8s administrators, for a number of reasons.

Handling the complexity in a changing Environment

The quest for observability in the constantly changing Kubernetes environment is not without its difficulties. The dynamic properties of microservices, the transient nature of containerized environments, and the complicated nature of distributed systems orchestration present significant challenges that necessitate creative thinking and flexible approaches. Engineers and operators must navigate a maze of challenges to maintain the resilience, security, and performance integrity of the Kubernetes ecosystem. These challenges include the complexities of managing ephemeral pod lifecycles to the details of tracking interrelated microservices interactions.

Kubernetes clusters are dynamic and extremely complex at the same time. In reaction to variations in demand, container instances spin up and down. Depending on node availability and scheduling priorities, pods may end on one node and migrate to another. Depending on storage needs, the mappings between storage volumes and specific containers may vary. And onwards.

Conventional infrastructure is less flexible. Periodically, a virtual machine may start up or shut down, but this usually doesn’t happen very often. Additionally, virtual machines hardly ever switch hosts. The location of a

virtual disk on the network usually doesn’t change, even though the data within it does. Stated differently, virtual machines are regarded as “pets”: every single one is a distinct entity that is handled carefully.

Kubernetes, on the other hand, views resources as “cattle.” The system continually switches between these immutable resources, stops, and restarts them. You can never assume that logs, metrics, or traces collected at one point in time accurately reflect the state of the cluster at a later time due to the dynamic nature of a Kubernetes cluster. Furthermore, you cannot presume that a log stream or metrics stream that you set up at first will always offer observability.

On the other hand, even if the historical data no longer accurately represents the state of your cluster or resources, you still need to be able to use it to guide ongoing observability operations. This means that you must manage observability continuously and in true real-time.

Unknown data sources

As we learned earlier, one last significant observability issue with Kubernetes is that observability data isn’t easily accessible. There is no master log file created by Kubernetes itself that you can easily tail to monitor the cluster. To get complete log visibility, users must consume log data from several sources. Although Kubernetes provides a metrics API, it is up to the user to gather metrics from it as well: Metric streaming is not supported by any built-in Kubernetes tooling.

Similarly, logs are not automatically consolidated in one place at the application level. Instead, log data is written by applications running inside of pods and containers to their respective internal environments.

Harnessing chances for Progress and Originality

However, there are opportunities to be found in the world of challenges. Staying ahead in a constantly changing technological landscape requires seizing opportunities for innovation and advancement in the dynamic field of Kubernetes observability. Engineers and operators can capitalize on the

revolutionary potential of cutting-edge technologies like edge computing, quantum-inspired analytics, and adaptive monitoring systems by establishing a culture of ongoing learning and experimentation. By harnessing the power of edge computing, real-time data can be seamlessly collected and analyzed at the network's edge, accelerating decision-making and improving Kubernetes applications' responsiveness in dispersed environments. Similarly, operators can now identify subtle anomalies and predict potential system failures with unprecedented precision thanks to the integration of quantum-inspired analytics, which enables them to decipher complex patterns within large datasets.

Adoption of adaptive monitoring solutions, which are based on the concepts of dynamic thresholding and self-learning, also makes it easier for monitoring parameters to be automatically adjusted in response to changing operational patterns and system behaviors. Operators can create a self-aware observability framework that prioritizes critical performance metrics, optimizes resource utilization, and adapts autonomously to changing workloads by utilizing machine learning and adaptive algorithms. This will help to guarantee that Kubernetes applications continue to function smoothly even in the face of dynamic operational environments and shifting demands. Engineers and operators can take the lead in Kubernetes observability by adopting these cutting-edge technologies and promoting an innovative culture. This will allow them to drive revolutionary advancements and redefine what operational excellence means in complex distributed systems.

A lot of theory, we need practice: Create a Kubernetes cluster

To build this platform, we have decided to use a local Kubernetes environment, in order to save time and money for those who do not have access to a cluster from a cloud provider. Likewise, if you have or want to use a cloud provider of your choice, you can do so, you just have to check if the official Cilium website recommends any extra configuration in this regard.

If you don’t already have one, you can build a local Kubernetes cluster by following the steps listed below.

Installing the cluster

Kind is a tool that uses Docker container “nodes” to run local Kubernetes clusters. kind can be used for local development or continuous integration, but its main purpose was testing Kubernetes itself.

Kind includes a default CNI kindnet by default. This basic CNI handles the fundamentals of networking but does not support more complex features such as NetworkPolicy.

For engineers and developers working on Kubernetes-based applications, there are a number of benefits to using a local KIND (Kubernetes in Docker) cluster. The following are some major advantages of utilizing a KIND cluster locally:

1. **Setup Ease:** It is not too difficult to set up a KIND cluster locally, and no complicated configurations are needed. With it, developers can easily build and maintain Kubernetes clusters without requiring a large amount of hardware or specialized infrastructure.

2. **Testing and Development:** KIND clusters provide an environment that closely mimics a production Kubernetes cluster, allowing developers to test and validate their applications. This configuration makes it possible to develop, debug, and troubleshoot applications more effectively without requiring a large-scale deployment.

3. **Separate Setting:** Local KIND clusters offer developers a secluded setting in which to test various configurations, deployments, and integrations. As a result, they can test new features and functionalities without worrying about how the production environment will remain stable.

4. **Economic Viability:** For development and testing purposes, running a local KIND cluster lowers the cost of provisioning and maintaining

cloud-based Kubernetes clusters. It saves developers money by enabling them to model real-world situations.

5. **Offline Development:** Local KIND clusters offer the convenience of working offline with flexibility.

There are other k8s options that you could use locally, like K3d, K3s, it will all depend on your tastes, feel free to explore.

Install kind >= v0.7.0 per kind documentation.

When this book was written, the version used was the `0.20.0`. Check the version running:

```
$ kind --version
```

One can supply a cluster spec along with cluster configuration when building a Kind cluster. Configuring the cluster default CNI and other cluster components is possible through the networking stanza.

You must create a YAML configuration file in order to configure kind cluster creation. The versioning and other conventions of this file adhere to Kubernetes.

A minimum acceptable configuration is:

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
```

Different versions of a given kind may be supported, each with its own set of options and behavior. For this reason, the version must always be specified.

The following command will spin up a cluster without installing kindnet:

```
$ kind create cluster --config - <<EOF
kind: Cluster
name: cilium-observability
apiVersion: kind.x-k8s.io/v1alpha4
networking:
  disableDefaultCNI: true
nodes:
- role: control-plane
```

```
- role: worker
- role: worker
- role: worker
EOF
```

Check that you are using the current Kubernetes context:

```
$ kubectl cluster-info --context kind-cilium-observability
```

Once created the cluster, run the following command:

```
$ kubectl get nodes
NAME                                  STATUS     ROLES
cilium-observability-control-plane   NoReady    control-plane
cilium-observability-worker          NoReady    <none>
cilium-observability-worker2         NoReady    <none>
cilium-observability-worker3         NoReady    <none>
```

Do not be scared when you see that your nodes are in the “Not ready” state, this is because we have enabled the “disableDefaultCNI” option when creating the cluster, for now, it is what is expected, in the next chapter, when learning about Cilium, we will install Cilium CNI in the cluster and your nodes will be ready to be used.

Summary

The source code for this chapter is available in the book’s GitHub.[1]

In this chapter, the focus is on explaining what observability means in the Kubernetes environment. We understood the three foundational elements of observability: Metrics, Logging, and Tracing and the spectrum of Kubernetes observability: difficulties and prospects

This knowledge sets the stage for the next chapter, **Chapter 2, Understanding the Cilium Fundamentals** where the focus shifts to exploring the core concepts and functionalities of Cilium, a powerful container networking and security solution that plays a crucial role in improving of observability within Kubernetes clusters.

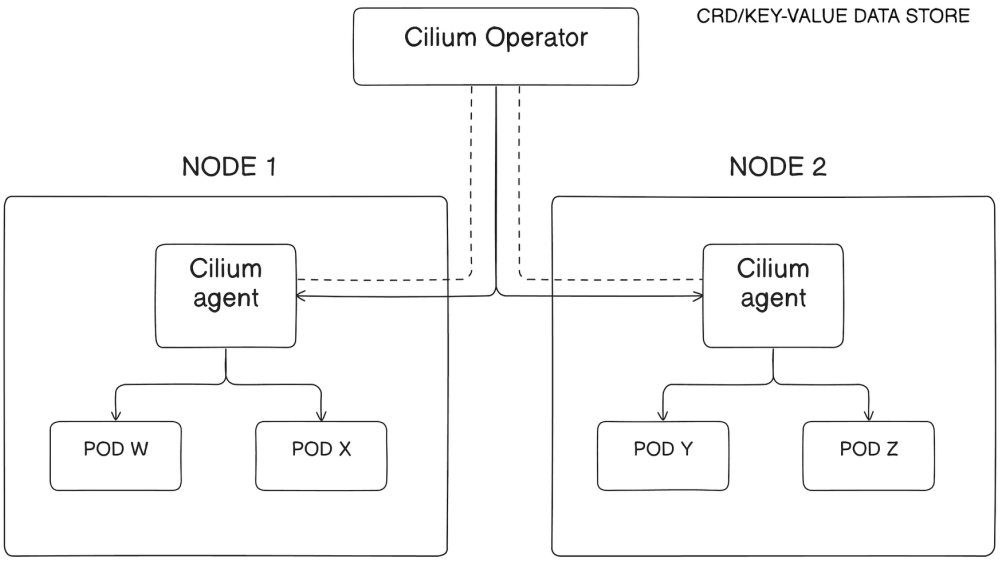
1. https://github.com/falconcr/cilium-observability/blob/main/chapter-1.md↩

2: Understanding the Cilium Fundamentals

Cilium is not just another networking solution; it’s a robust, modern, and downright ingenious networking and security project designed for the container age. At its core, Cilium provides transparent, high-performance networking, and security for microservices-based applications, helping you transcend the limitations of traditional networking approaches.

In this chapter we want to explain the fundamentals of Cilium to you at a high level, the idea is that you can know and understand them. Additionally, we will update the cluster created in the previous chapter, so that it uses Cilium as its CNI.

Cilium components



- Operator: The Cilium Operator simplifies deploying, updating, and maintaining Cilium on a Kubernetes cluster. Automate tasks such as installing Cilium, updating to new versions, and managing specific configurations.

The operator uses custom resource definitions (CRDs) to define how Cilium should be configured in the cluster. These CRDs contain information about the specific Cilium settings that the operator must apply.

- Agent: The cilium-agent is the main component that runs on each node of the Kubernetes cluster. It is responsible for interacting with the operating system kernel and managing Cilium's network and security settings.

The Cilium Agent is responsible for implementing network policies, applying eBPF (Extended Berkeley Packet Filter) rules in the kernel for security, and managing the advanced networking functions offered by Cilium.

It works together with the Cilium Operator to receive the configuration and policies defined at the cluster level and apply them appropriately on each node.

- eBPF (Extended Berkeley Packet Filter): Cilium leverages eBPF, a groundbreaking technology that enables programmability at the kernel level without the need to modify or recompile it. This allows developers to enforce security policies, perform deep packet inspection, and implement custom network functions with unparalleled efficiency.

- Layer 7 Visibility: Cilium operates at the application layer, providing deep visibility into API interactions. It understands the context of your microservices, allowing you to enforce security policies based on application layer attributes.

- HTTP and gRPC Aware: Cilium doesn't just stop at IP and port; it's aware of HTTP and gRPC protocols, enabling fine-grained security policies for modern, API-driven applications.

- Identity-Aware Connectivity: Cilium ensures that only authenticated and authorized services can communicate, adding an extra layer of security to your microservices architecture.

- Load Balancing and Service Discovery: Seamlessly integrate load balancing and service discovery into your Kubernetes clusters, simplifying the deployment and scaling of your applications.

- Native Kubernetes Support: Cilium is Kubernetes-native, seamlessly integrating with your clusters. This means effortless deployment, automatic scaling, and native compatibility with Kubernetes Network Policies.

- Global Network Policies: Cilium provides a unified policy framework for multi-cluster and multi-tenancy environments, allowing you to define policies that span across clusters and tenants.

eBPF: the key component enabling Cilium's success

Cilium's robust feature set is centered around a technology known as eBPF. A tool for observing the Linux operating system, the Berkeley Packet Filter, or BPF for short (no, we did not forget the 'e'), has been around for a few years. Through the swift and safe execution of a small piece of code within the operating system, BPF provides users with access to the kernel.

Although packet filtering was the original purpose of BPF, it has since been improved to enable dynamic Linux kernel tracing. Now, for instance, a user can write a program related to cgroups that permits or prohibits access to system resources such as CPU, memory, network bandwidth, and other process groups.

So, eBPF (extended BPF) is a new Linux kernel technology, which allows strong security visibility and control logic to be dynamically inserted within Linux. Cilium security policies can be applied and updated without requiring modifications to the application code or container configuration because eBPF operates inside the Linux kernel.

Given that the kernel has the ability to monitor and control the entire system, operating systems have historically been the ideal platform for implementing features related to security, observability, and networking. The operating system kernel's pivotal role and strict security and stability requirements make it resistant to evolution. Because of this, operating system-level innovation has lagged behind features that are applied externally.

This formula is drastically altered by eBPF, which permits sandboxed programs to function within the operating system. eBPF programs are a useful tool for application development teams to add functionality at runtime. Like native compilation, the operating system ensures safe and effective execution by using a just-in-time (JIT) compiler and validation engine.

Cilium features: High level overview

Cilium offers unparalleled security and observability.

For Kubernetes, there are numerous CNIs available, but there are significant differences in their features, scale, and performance. Many of them depend on antiquated technology (iptables), which is unable to manage the volume and turbulence of Kubernetes environments, increasing latency and decreasing throughput. Additionally, the majority of CNIs only provide minimal support for L3/L4 Kubernetes network policies. Customers using multi-cloud environments face operational complexity as a result of the many cloud providers' unique CNIs.

In large-scale, highly dynamic cloud native environments, where hundreds or even thousands of containers are created and destroyed in a matter of seconds, Cilium's control and data plane was designed from the ground up. With thousands of nodes and 100,000 pods operating in Kubernetes clusters, Cilium's control plane is extremely optimized. eBPF is used in Cilium's data plane to provide incremental updates and efficient load balancing while avoiding the drawbacks of extensive iptables rule sets.

Scale is a feature of Cilium. It is capable of handling clusters with thousands of nodes or a few running on it. The eBPF-powered networking from Cilium is designed with large-scale operations in mind. As a result, you can expand your business without being concerned that the network will become congested.

Layer 4 Load Balancer

With the help of XDP and eBPF, Cilium can leverage BGP to draw in traffic and quicken it. When combined, these technologies offer a very reliable and secure load balancing implementation. eBPF and Cilium both function at the kernel layer. This degree of context allows for the intelligent selection of the best way to connect various workloads, either within or between clusters, on the same node. Performance and latency can be significantly improved with eBPF and XDP using Cilium. The stand-alone load balancer from Cilium offers a high-performance LB with significant throughput increases and minimal CPU overhead.

The reliable, high-performance load balancing solution from Cilium is designed for the churn and scale of cloud native environments. Cilium can be used as a stand-alone load balancer in your network in place of pricey legacy boxes. This frees up DSR and Maglev to handle traffic north-south in on-premises environments without requiring network border management from Kubernetes.

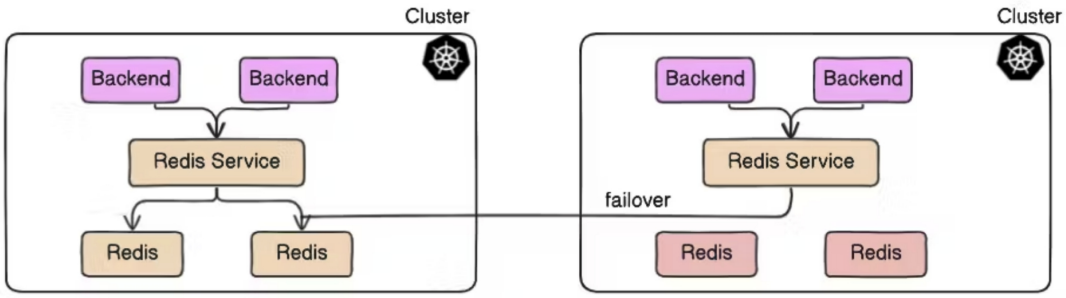
Using Cilium Cluster Mesh to Unlock the Power of Multi-Cluster Networking

Multi-cluster Kubernetes configurations are widely used for reasons such as geographic dispersion, scalability, and fault isolation. This strategy might make networking more difficult. In this situation, load balancing across clusters, policy enforcement, network segmentation, and service discovery are challenges for traditional networking models. Additionally, because services are distributed, it can be difficult to manage security policies and protocols across several environments.

If every cluster uses Cilium as its CNI, Cilium Cluster Mesh enables you to link the networks of several clusters so that pods in each cluster can find and access services in every other cluster in the mesh. This makes it possible to efficiently combine several clusters into a sizable, cohesive network, regardless of the Kubernetes distribution or the physical location of each cluster.

Your services' fault tolerance and high availability are improved by cluster mesh. It enables Kubernetes clusters to run across several availability zones or regions. It allows failover to other clusters in case resources are misconfigured in one cluster, go offline for upgrades, or become temporarily unavailable. This guarantees that your services are always accessible.

Your Kubernetes clusters' service discovery is automated by Cluster Mesh. It automatically combines services with the same names and namespaces from different clusters into a single global service by using standard Kubernetes services. This makes cross-cluster communication much simpler because your applications can find and communicate with services regardless of which cluster they are located in.



In the previous diagram we can see that we have 2 clusters with the same services and namespaces. Suppose that the Redis pods in the cluster on the right fail, this implies that the backend pods will have a problem accessing Redis, causing connectivity errors.

With Cilium, we could route the service in the cluster on the right to point to the pods in the cluster on the left, it enables failover to other clusters, ensuring your services remain accessible at all times.

Utilize Cilium to fully utilize Border Gateway Protocol's (BGP) limitless potential

Particularly in cloud-native environments where workloads are continuously added, moved, and removed, traditional IP routing can be static and rigid. Modern cloud native environments are dynamic and distributed, and BGP is ideally suited for managing complex network topologies and routing data.

The internet's backbone, BGP, is strengthened by cilium to provide you with fast, scalable, and secure routing for your cloud environments. Cilium's BGP seamlessly integrates with current infrastructure, making it ideal for a variety of deployments, including hybrid, multi-cloud, and edge. You can optimize the performance and security of your network by having fine-grained control over your network traffic thanks to advanced traffic engineering features.

The BGP support in Cilium is made to be straightforward and simple to integrate with your current networking setup. For all of your workloads, Cilium can guarantee effective BGP routing, regardless of where your application is running. BGP is already widely used in network infrastructures for routing. Through the use of BGP, Cilium can be easily integrated with current infrastructure, facilitating communication between Kubernetes pods and other network nodes.

Enhancing Kubernetes Security and Observability with Cilium

In the dynamic landscape of containerized applications orchestrated by Kubernetes, ensuring robust security and comprehensive observability are paramount. Cilium emerges as a powerful solution, offering a suite of features designed to fortify your Kubernetes clusters and elevate your operational insights.

1. Service-Centric Security: Cilium's service-aware network security policies redefine how we approach security in Kubernetes. By focusing on

the identity of services rather than mere IP addresses, it introduces granular controls that significantly reduce the attack surface and enhance overall resilience.

2. API-Aware Network Security: Elevating security to the application layer, Cilium enables policies based on actual API calls. This contextual understanding empowers administrators to craft nuanced security postures, adding an extra layer of defense against potential threats.

3. Efficient and Performant: Leveraging the efficient Berkeley Packet Filter (BPF) technology, Cilium operates within the kernel, delivering high performance and low latency. This efficiency is crucial for maintaining the speed and responsiveness expected in modern, containerized environments.

4. Comprehensive Observability: Cilium doesn't stop at security; it provides robust observability features. Metrics and tracing capabilities ensure that you have the tools necessary to monitor, troubleshoot, and optimize the performance of your applications running in Kubernetes.

5. Seamless Integration with Kubernetes: Serving as a Container Network Interface (CNI) plugin, Cilium seamlessly integrates into Kubernetes clusters. This ensures a smooth adoption process without disrupting existing infrastructure, making it an accessible and practical choice for enhancing your cluster's capabilities.

6. Distributed Load Balancing: With built-in load balancing capabilities, Cilium enhances the availability and reliability of your applications by distributing traffic efficiently across your services. This feature becomes increasingly vital as applications scale and demand greater resilience.

7. Community Support and Future-Ready: As part of the Cloud Native Computing Foundation (CNCF), Cilium benefits from a vibrant community and ongoing development. This ensures that your investment in Cilium aligns with the latest best practices and seamlessly integrates with emerging technologies in the Kubernetes ecosystem.

In conclusion, Cilium stands as a versatile and comprehensive solution for fortifying the security and observability of your Kubernetes clusters.

Whether you are navigating the complexities of microservices, enforcing fine-grained security policies, or optimizing performance, Cilium empowers you to navigate the challenges of modern container orchestration with confidence.

Installing Cilium

You already have your cluster created, it is the time to install the Cilium.

```
$ helm repo add cilium https://helm.cilium.io

$ helm upgrade --install \
  --namespace kube-system \
  cilium cilium/cilium \
  --values - <<EOF
  kubeProxyReplacement: strict
  hostServices:
    enabled: false
  externalIPs:
    enabled: true
  nodePort:
    enabled: true
  hostPort:
    enabled: true
  image:
    pullPolicy: IfNotPresent
  ipam:
    mode: Kubernetes
  EOF
```

You can check if Cilium was deployed in your cluster running the following command:

```
$ kubectl get pods -n kube-system -o wide --watch
```

If you are working with a single-node then you will see only one cilium pod, however, if you are working with multi-nodes, you will see one cilium pod per node.

```
$ kubectl get pods -n kube-system -o wide
NAME            READY   STATUS    NODE
cilium-749rw    1/1     Running   cilium-observability-worker
cilium-bjpm2    1/1     Running   cilium-observability-worker2
cilium-jntpg    1/1     Running   cilium-observability-worker3
cilium-lg2d8    1/1     Running   cilium-observability-control-plane
```

Of course, run `kubectl get nodes` in order to check that your nodes are `Ready`.

```
$ kubectl get nodes
NAME                                  STATUS  ROLES          VERSION
cilium-observability-control-plane  Ready   control-plane  v1.27.3
cilium-observability-worker         Ready   <none>         v1.27.3
cilium-observability-worker2        Ready   <none>         v1.27.3
cilium-observability-worker3        Ready   <none>         v1.27.3
```

Check the instalation

Install the most recent Cilium CLI version. Installing Cilium, checking the status of an installation, and enabling/disabling different features (like clustermesh and Hubble) can all be done with the Cilium CLI.

Depends on what operating system you are using, you should choose the appropriate commands, go to https://docs.cilium.io/en/latest/installation/kind/#validate-the-installation and choose your preference.

For example, using Linux, you can install the lastest version of Cilium CLI running:

```
$ CILIUM_CLI_VERSION=$( \
  curl -s \
  https://raw.githubusercontent.com/cilium/cilium-cli/main/stable.txt)

$ CLI_ARCH=amd64

$ if [ "$(uname -m)" = "aarch64" ]; then CLI_ARCH=arm64; fi

$ curl -L --fail --remote-name-all \
  "https://github.com/cilium/cilium-cli/releases/download/${CILIUM_CLI_VERSION}\
/cilium-linux-${CLI_ARCH}.tar.gz" \
  "https://github.com/cilium/cilium-cli/releases/download/${CILIUM_CLI_VERSION}\
/cilium-linux-${CLI_ARCH}.tar.gz.sha256sum"


$ sha256sum --check cilium-linux-${CLI_ARCH}.tar.gz.sha256sum

$ sudo tar xzvfC cilium-linux-${CLI_ARCH}.tar.gz /usr/local/bin

$ rm cilium-linux-${CLI_ARCH}.tar.gz{,.sha256sum}
```

You can run `cilium status --wait` to confirm that Cilium has been installed correctly. You should see an output like this:

```
    /¯¯\
 /¯¯\__/¯¯\    Cilium:             OK
 \__/¯¯\__/    Operator:           OK
 /¯¯\__/¯¯\    Envoy DaemonSet:    disabled (using embedded mode)
 \__/¯¯\__/    Hubble Relay:       disabled
    \__/       ClusterMesh:        disabled

Deployment         cilium-operator    Desired: 2, Ready: 2/2, Available: 2/2
DaemonSet          cilium             Desired: 4, Ready: 4/4, Available: 4/4
Containers:        cilium             Running: 4
                   cilium-operator    Running: 2
Helm chart version: 1.14.3
Image versions      cilium             quay.io/cilium/cilium:v1.14.3
                    cilium-operator    quay.io/cilium operator-generic:v1.14.3
```

Good Job, you already have cilium running in your k8s cluster.

Summary

The source code for this chapter is available in the book's GitHub.[1]

In this pivotal chapter, the focus is on delving into the fundamental principles of Cilium, a robust and versatile container networking and security solution designed to enhance the observability of Kubernetes environments. The chapter begins by providing readers with an insightful overview of Cilium's core concepts and functionalities, shedding light on how it contributes to the overall observability of containerized applications.

The closing section of the chapter serves as a bridge to the next phase of the journey—an exploration of "**Chapter 3, Prometheus and Grafana - Empowering Kubernetes Observability**." Readers are primed with foundational knowledge, prepared to harness the combined power of Cilium, Prometheus, and Grafana for a holistic observability solution in the dynamic landscape of Kubernetes.

This chapter serves as a pivotal juncture, equipping readers with the essential insights into Cilium's role in Kubernetes observability and paving the way for a deeper dive into the integration of Prometheus and Grafana in the chapters that follow.

1. https://github.com/falconcr/cilium-observability/blob/main/chapter-2.md↩

3: Prometheus and Grafana - Empowering Kubertenes Observability

In the ever-evolving landscape of Kubertenes, the need for robust observability has become indispensable. As organizations embrace container orchestration, understanding and monitoring the health, performance, and behavior of their applications is crucial. In this chapter, we delve into two stalwarts of the observability toolkit: Prometheus and Grafana.

Prometheus: Unveiling the Power of Metrics

Prometheus, an open-source monitoring and alerting toolkit, has emerged as a cornerstone for Kubertenes observability. Its design principles align seamlessly with the dynamic and scalable nature of containerized environments. Prometheus operates on a pull-based model, where it scrapes metrics from instrumented targets, providing real-time insights into the performance of your applications.

Data Model and Query Language

At the heart of Prometheus lies a dimensional data model, allowing metadata-rich time series data. This model enables flexible querying and slicing, providing a nuanced understanding of your Kubertenes ecosystem. The PromQL query language allows users to express complex queries, empowering them to extract valuable information from the vast sea of collected metrics.

Dynamic Service Discovery

Kubertenes, with its dynamic nature, requires a monitoring solution that can adapt on the fly. Prometheus excels at this by leveraging Kubertenes' service discovery mechanisms. As pods and services scale up or down, Prometheus dynamically discovers and monitors them, ensuring a comprehensive view of your applications.

Alerting and Recording Rules

Beyond metrics collection, Prometheus boasts a robust alerting system. Operators can define alerting rules based on custom conditions, empowering them to proactively respond to anomalies. Additionally, recording rules allow users to precompute frequently needed or computationally expensive queries, enhancing the efficiency of metric retrieval.

Grafana: Crafting Visual Narratives for Observability

While Prometheus excels at collecting and storing metrics, Grafana steps in to visualize and make sense of this wealth of data. Grafana is a versatile open-source platform that provides a rich set of visualization options, dashboards, and tools for creating interactive and insightful displays.

A strong and comprehensive monitoring and visualization tool is essential in the dynamic world of Kubertenes observability. A centralized, user-friendly, and potent platform is required to interpret complex performance metrics and behavior patterns in increasingly distributed and diverse complex infrastructures. This chapter takes us on an in-depth investigation of the many functions offered by Grafana, the leading open-source platform in the industry that is renowned for its adaptability and potent visualization tools.

Merge your data

Data ingest to a backend store or vendor database is not necessary when using Grafana. Rather, Grafana employs a novel strategy to offer a “single pane of glass” by consolidating all of your current data, regardless of where it is stored.

Any existing data you have, from a single dashboard, can be visualized in any way you choose using Grafana. This includes data from your Kubertenes cluster, Raspberry Pi, various cloud services, and even Google Sheets.

To sum up, Grafana is more powerful than just a visualization tool; it is a strategic tool that helps engineers and operators get deeper insights, make wise decisions, and maximize the efficiency of their Kubertenes systems. Through the utilization of Grafana’s extensive feature set, users can effectively decipher complex environments, promote operational excellence, and guarantee the smooth operation of distributed systems in the constantly changing Kubertenes observability landscape.

Flexible Dashboards

Grafana’s strength lies in its flexibility. Users can design custom dashboards tailored to their specific use cases. Whether visualizing resource utilization, latency patterns, or application-specific metrics, Grafana allows for a high degree of customization, turning raw data into meaningful insights.

Grafana provides a large selection of pre-built dashboards that are easy to use and can be accessed by people with different levels of technical proficiency. With the help of these dashboards, users can easily and efficiently monitor, analyze, and optimize the behavior and performance of their infrastructure. They offer users comprehensive visualizations and insights into key performance indicators, metrics, and data points within their Kubertenes environments. Among the noteworthy characteristics of these intuitive dashboards are:

Easy Navigation: The dashboards' easy-to-use interface makes it possible for users to quickly go between different data points and visualizations, creating a smooth and engaging user experience. Users may easily obtain the information they require and make deft decisions based on real-time data insights thanks to an intuitive interface and simplified navigation.

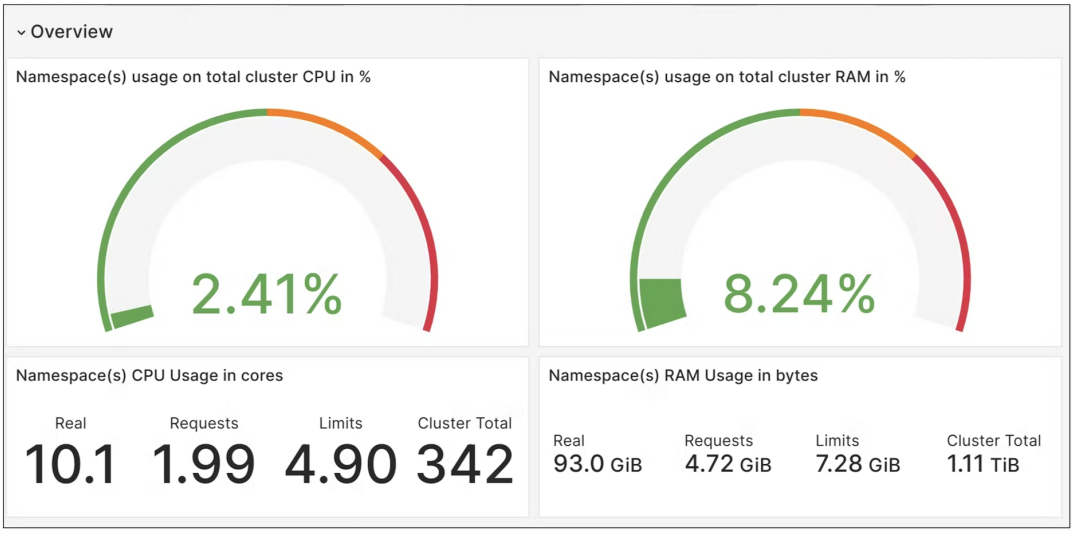
Customized Templates: Grafana provides a selection of dashboard templates that can be customized to suit a variety of use cases and business sectors. These templates provide users with an adaptable framework for customizing dashboards that meet their unique operational needs. This enables users to efficiently track and examine important performance metrics and patterns of behavior in their Kubertenes environments.

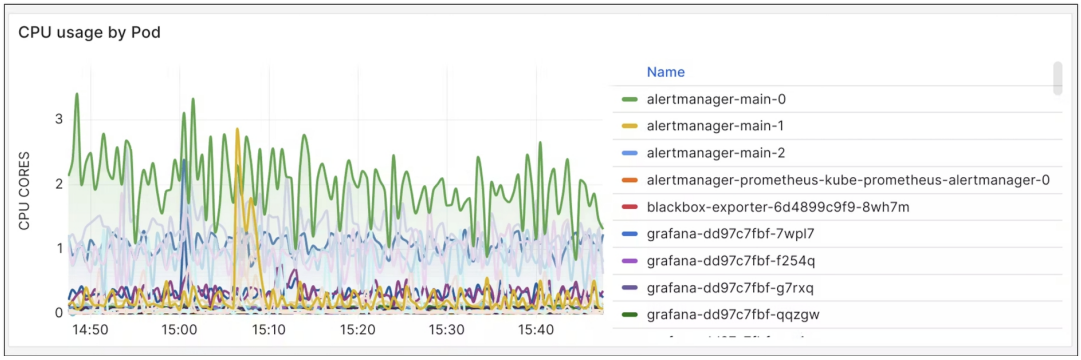
Real-time Monitoring: By enabling real-time monitoring of crucial metrics and KPIs, the dashboards give users the most recent information on the functionality and condition of their infrastructure. The dashboards help users stay updated about the latest developments and trends in their Kubertenes environments, allowing them to make well-informed decisions in a timely manner to enhance operational excellence. This is achieved by providing real-time data visualization and analytics.

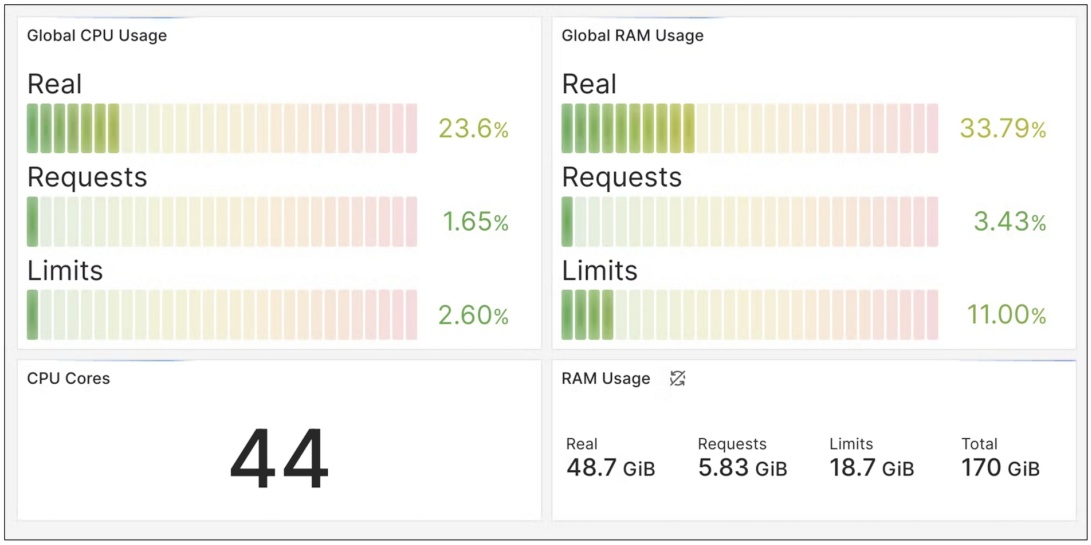
Interactive Visualizations: Grafana's dashboards allow users to review and evaluate data points with a great level of accuracy and granularity. The dashboards provide users with a thorough understanding of the performance and behavior of their infrastructure through the integration of dynamic and interactive elements like heatmaps, graphs, and charts. This enables users to make proactive decisions and optimize their infrastructure strategically by acting on actionable insights.

In the following dashboard you can review the status of the cluster nodes, for example:

- Network I/O pressure
- Total memory usage, CPU, Filesystem.
- CPU usage by pods.







Overall, Grafana’s user-friendly dashboards serve as a powerful tool for democratizing access to critical data insights and analytics within

Kubertenes environments, enabling users with varying levels of technical expertise to monitor, analyze, and optimize the performance and behavior of their infrastructure effectively and efficiently.

Rich Panel Options

Grafana offers a variety of panels that cater to different visualization needs. From time series graphs and single stat metrics to heatmaps and tables, each panel is a building block that contributes to a comprehensive and intuitive view of your Kubertenes clusters.

Integration with Prometheus

Grafana seamlessly integrates with Prometheus, making it a natural companion in the observability stack. Through simple configuration, users can connect Grafana to Prometheus as a data source, unlocking the ability to create dashboards that leverage the metrics stored in Prometheus.

The Symbiotic Relationship

Prometheus and Grafana form a symbiotic relationship that transforms raw metrics into actionable insights. Prometheus collects and stores the data, while Grafana brings it to life through visually appealing and informative dashboards. Together, they empower Kubertenes operators and developers to monitor, troubleshoot, and optimize their applications with unparalleled precision.

I’ll embark on a journey to set up Prometheus and Grafana in a Kubertenes environment, configure them to work harmoniously, and demonstrate how this dynamic duo elevates observability, making it an integral part of your Kubertenes strategy. Prepare to unlock a new realm of insights and efficiencies as we dive into the implementation and utilization of Prometheus and Grafana in your Kubertenes observability toolkit.

Installing Cilium and Hubble

Installing Cilium and Hubble involves a series of steps to set up these tools in a Kubertenes environment. Cilium is a powerful networking and security project, while Hubble is a real-time network visibility tool that integrates seamlessly with Cilium. Here is a step-by-step guide to help you install Cilium and Hubble in your Kubertenes cluster:

Step 1: Prerequisites

Before you begin, ensure that you have the following prerequisites:

- A running Kubertenes cluster (minikube, kind, or a cloud-managed cluster).
- kubectl installed and configured to connect to your Kubertenes cluster.
- Helm version 3.x installed.

Step 2: Verify Cilium Installation

Verify that Cilium pods are running:

```
$ kubectl get pods -n kube-system -l k8s-app=cilium
NAME           READY   STATUS    RESTARTS   AGE
cilium-8ncbq   1/1     Running   0          3d23h
cilium-mws75   1/1     Running   0          3d23h
cilium-tb6sm   1/1     Running   0          3d23h
cilium-xxg6s   1/1     Running   0          3d23h
```

Step 3: Install Hubble

Hubble is a component of Cilium that facilitates the collection and aggregation of metrics from Cilium’s datapath. These metrics provide deep insights into the network behavior and performance of your Kubertenes cluster. By deploying Hubble alongside Cilium, you gain the ability to visualize and analyze real-time and historical network traffic patterns.

Hubble can offer visibility at the node, cluster, or even cross-cluster levels in a multi-cluster (cluster mesh) situation.

Metrics Collected by Hubble

Flow Metrics

Source and Destination: Hubble captures information about the source and destination of network flows. This includes pod or container identifiers, making it easy to identify communication patterns between different services.

Packet and Byte Counts: Hubble tracks the number of packets and bytes exchanged in each flow. This information is crucial for understanding the volume of data being transferred and can aid in identifying potential bottlenecks or anomalies.

Latency Metrics: Hubble measures the latency of network flows, providing insights into the time taken for communication between different services. This is valuable for diagnosing performance issues and optimizing the overall responsiveness of your applications.

DNS Metrics

DNS Query and Response Times: Hubble provides metrics related to DNS queries and response times. Monitoring these metrics can help identify potential DNS-related issues impacting the performance of your applications.

HTTP Metrics

HTTP Request and Response Metrics: For services communicating over HTTP, Hubble captures metrics related to HTTP requests and responses. This includes details like status codes, response times, and more, aiding in the analysis of web application behavior.

Installation

Hubble is included in the Cilium Helm chart. To enable Hubble, run:

```
$ helm upgrade cilium cilium/cilium --version 1.14.3 \
   --namespace kube-system \
```

```
--reuse-values \
--set prometheus.enabled=true \
--set operator.prometheus.enabled=true \
--set hubble.enabled=true \
--set hubble.metrics.enableOpenMetrics=true \
--set hubble.metrics.enabled="{dns,drop,tcp,flow, \
port-distribution,icmp,httpV2:exemplars=true; \
labelsContext=source_ip\,source_namespace\, \
source_workload\,destination_ip\,destination_namespace\, \
destination_workload\,traffic_direction}"
```

With the previous command we are enabling both metrics for cilium itself and for the pods that are managed by cilium.

Let me explain what each of those flags mean:

- `--set prometheus.enabled=true` configures prometheus metrics on the configured port at `/metrics`. It enables metrics for the `cilium-agent`.
- `--set operator.prometheus.enabled=true` enables prometheus metrics for cilium-operator on the configured port at `/metrics`.
- `--set hubble.enabled=true` enables hubble (true by default).
- `--set hubble.metrics.enableOpenMetrics=true` enables exporting hubble metrics in OpenMetrics format.
- `--set hubble.metrics.enabled={...}` configures the list of metrics to collect. If empty or null, metrics are disabled.

Hubble metrics let you keep an eye on the connectivity and security of your Kubertenes pods that are managed by Cilium, while Cilium metrics let you keep an eye on the state of Cilium itself.

Installing Prometheus and Grafana

Prometheus metrics can be served by both Hubble and Cilium. Grafana is a metrics visualization frontend that can receive data from Prometheus, a pluggable metrics collection and storage system. Prometheus demands that

metrics be pulled from every source, in contrast to some metrics collectors such as statsd.

This deployment serves as a way of combining Grafana and Prometheus into one.

Included in the default installation are:

- Grafana: A dashboard for visualization that comes pre-loaded with Cilium Dashboard.
- Prometheus is a database and monitoring system for time series.

You can deploy a Prometheus and Grafana stack using the following if one isn't already up and running:

```
$ kubectl apply -f \
  https://raw.githubusercontent.com/falconcr/cilium-observability/main/chapter-\
3/monitoring.yaml
```

Check that grafana and prometheus pods are running:

```
$ kubectl get pods -n monitoring --watch
```

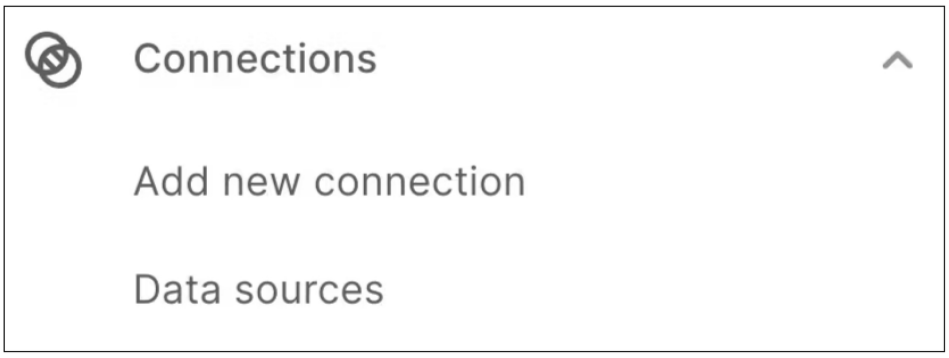
Prometheus and Grafana will be executed in the `monitoring` namespace. Prometheus will automatically scrape Cilium or Hubble metrics if you have enabled them. After that, you can use your browser to access Grafana by exposing it.

Expose the port on your local machine:

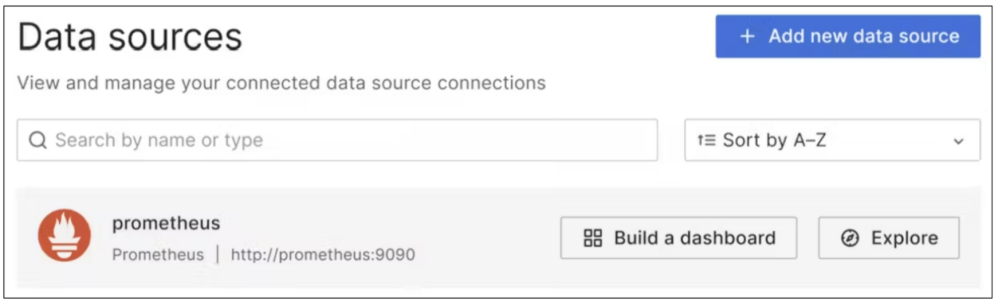
```
$ kubectl -n monitoring port-forward \
  service/grafana \
  --address 0.0.0.0 \
  --address :: 3000:3000
```

Go to the browser and navigate `http://localhost:3000/`.

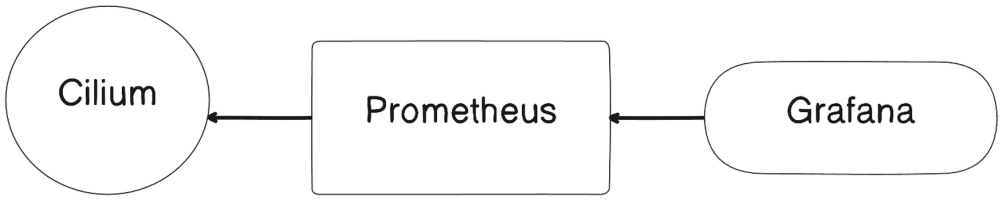
Go to the left side, click on `Connections` button and click on `Data sources` option:



The previous deployment configured Prometheus as a data source for Grafana, which means that Grafana pulls metrics from Prometheus and builds the dashboards based on them.



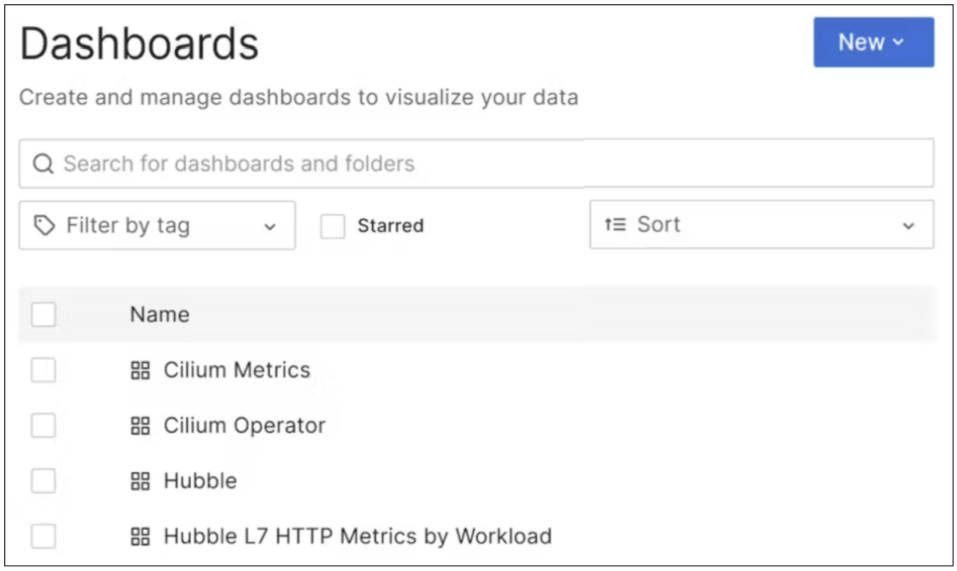
If you are not familiar with prometheus and grafana, in the following image we will review at a very high level how each component connects to each other.



As you can see, Prometheus itself does not create the metrics, but rather collects them and allows queries to be made using a language called PromQL (Prometheus Query Language). So who creates the metrics that

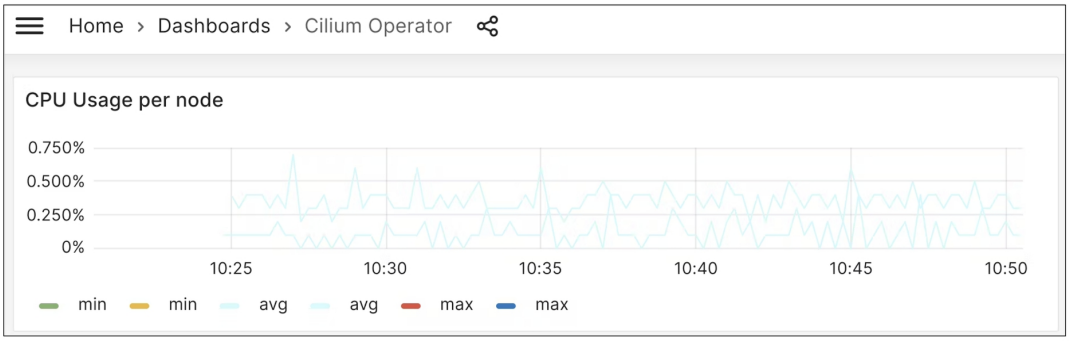
Prometheus uses? There is a concept called “exporter” which is responsible for creating the metrics and exposing them in a format that Prometheus can interpret through an endpoint, which is generally “/metrics” (if you want to read more about it, visit the website https://prometheus.io/docs/instrumenting/exporters/). For the context of the book, the exporter is Cilium, this is creating and exposing metrics in a format that Prometheus can process. On the other hand, we have Grafana, which needs a data source to be able to create or view metrics using dashboards. Among the many data sources that Grafana supports is Prometheus, which we have configured.

At this point, we have a system that collects cilium metrics, which are used to generate dashboards in Grafana. To view the dashboards, navigate in the left sidebar of Grafana, find the section called Dashboards, click on the Browse option. The following dashboard list will be shown:



- Cilium Metrics: Visualize the metrics the each cilium agent in the cluster. Some metrics shown are CPU usage per node, BPF map pressure, API return codes (average node), and connectivity health.

- Cilium Operator: Visualize the metrics of the operator. Some metrics shown are CPU usage per node, IPAM and resident memory status.



- Hubble: Visualize the network behavior of your Cilium-managed Kubertenes pods with respect to connectivity and security. Some metrics shown are flow types, trace flow distribution, L7 Flow distribution, Drop Reason, etc.

Summary

The source code for this chapter is available in the book's GitHub.[1]

Well, that's enough for now, we have learned how to configure cilium to expose metrics that prometheus and grafana can use to create dashboards. But we still have other interesting tools to integrate, that is the case of Hubble service map, which we will see in the **Chapter 4, Service Map & Hubble UI**.

1. https://github.com/falconcr/cilium-observability/blob/main/chapter-2.md↩

4: Service Map & Hubble UI

Service Map and Hubble UI are key components in the Cilium ecosystem for Kubertenes. Cilium is a container networking and security platform based on eBPF, enabling advanced visibility, security, and connectivity within Kubertenes environments.

Service Map

A Service Map is a graphical representation that illustrates the relationships and dependencies between different components or services within a system or application. In the context of microservices architecture in Kubertenes, a Service Map provides a visual overview of how various microservices communicate and interact with each other. It allows users to understand the flow of data and dependencies, providing a high-level view of the entire system.

Key features of a Service Map may include:

- **Node Representation**: Each microservice or component is represented as a node in the map.
- **Connection Lines**: Arrows or lines indicate the connections and communication pathways between nodes.
- **Color Coding**: Different colors may be used to represent various states or attributes of the services.
- **Real-time Updates**: Some Service Maps offer real-time updates to reflect dynamic changes in the system.

Hubble UI

Hubble UI is a component of the Cilium project, an open-source software that provides networking and security functions for containerized applications in Kubertenes. Hubble UI specifically focuses on enhancing observability within a Kubertenes cluster. It visualizes the network traffic and interactions between microservices, offering a real-time Service Map that aids in understanding the communication patterns.

Key features of Hubble UI include:

- **Live Network Visualization**: Hubble UI provides a live, interactive visualization of the network traffic between microservices. It allows users to observe the flow of traffic, understand communication patterns, and identify potential bottlenecks.

- **Topology Overview**: It offers a topology overview, allowing users to see how different services are interconnected.

- **Security Insights**: Hubble UI helps in visualizing and enforcing security policies, enhancing the overall security posture of the Kubertenes environment. From a security perspective, Hubble UI helps in monitoring and understanding the security policies enforced by Cilium. Users can visualize how network security policies are applied and gain insights into which services are communicating and the policies governing those communications.

- **Troubleshooting Support**: The real-time monitoring capabilities of Hubble UI assist in troubleshooting by quickly identifying issues such as bottlenecks or anomalies in network traffic. When issues arise in a Kubertenes environment, understanding the network behavior is crucial for effective troubleshooting. Hubble UI simplifies this process by providing a graphical representation of the network, making it easier for operators and developers to identify and address problems.

- **Integration with Cilium and eBPF**: Cilium leverages eBPF (Extended Berkeley Packet Filter) for efficient packet processing.

Hubble UI integrates with Cilium and eBPF to present meaningful and actionable insights derived from low-level network data, making it accessible and understandable for users.

- **User-Friendly Interface**: The graphical user interface of Hubble UI is designed to be user-friendly, enabling both developers and operators to interact with the data easily. Visualization of complex network structures becomes more manageable, even for those who may not have an in-depth understanding of networking internals. Historical Data Analysis:

Hubble UI often includes features for historical data analysis, allowing users to review past network activities. This is beneficial for understanding trends, identifying patterns over time, and conducting post-incident analyses. Enhanced Decision-Making:

By providing a clear and intuitive view of the network, Hubble UI empowers Kubertenes administrators, operators, and developers to make informed decisions related to network policies, security configurations, and overall system optimization.

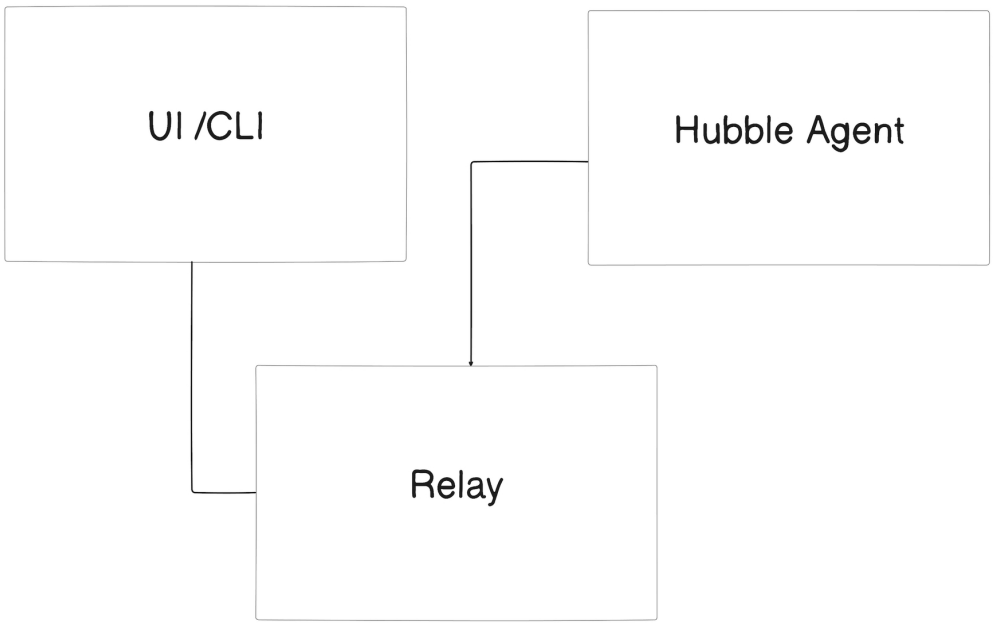
Installation

To install the hubble service map, we need to enable some extra components, such as hubble relay and hubble ui. To do this, you must execute the following command:

```
$ helm upgrade cilium cilium/cilium --version 1.14.3 \
   --namespace kube-system  \
   --reuse-values \
   --set hubble.relay.enabled=true \
   --set hubble.ui.enabled=true
```

Run `kubectl get pods -n kube-system` and you will see new two pods with status `Running`, the prefixs of pods are `hubble-ui` and `hubble-relay`.

At a very high level, these new components interact in the following way:



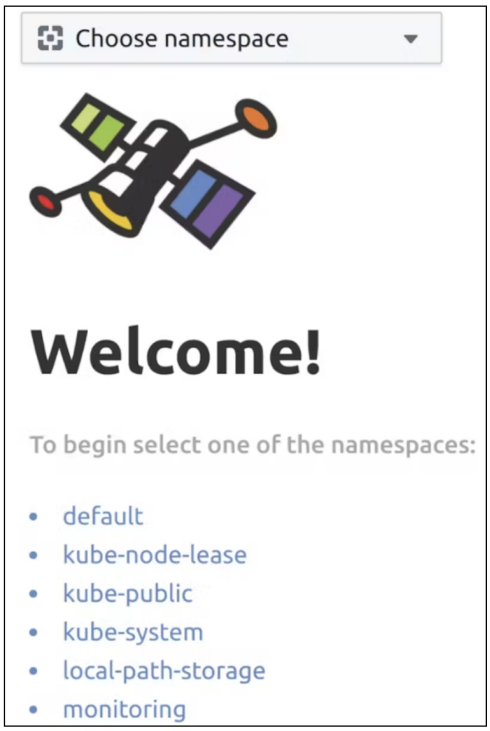
- **UI/CLI (User Interface/Command Line)**: Hubble provides a user interface and command line that allows users to interact with the network metrics and visualizations provided by Cilium.
- **Hubble Agent**: Hubble Agent is a component that is deployed on each node in the cluster. Its main function is to collect information about network traffic and send it to the Hubble Relay for processing and storage.
- **Hubble Relay**: The Hubble Relay is a centralized component that receives and stores metrics sent by Hubble agents. It facilitates the aggregation and processing of data from multiple nodes to provide a comprehensive view of the cluster network.

Using hubble UI

Run the command `cilium hubble ui`, you should see a message such as the following:

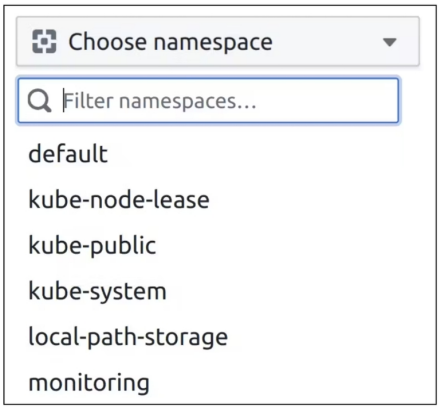
$ cilium hubble ui **i** Opening “http://localhost:12000” in your browser…

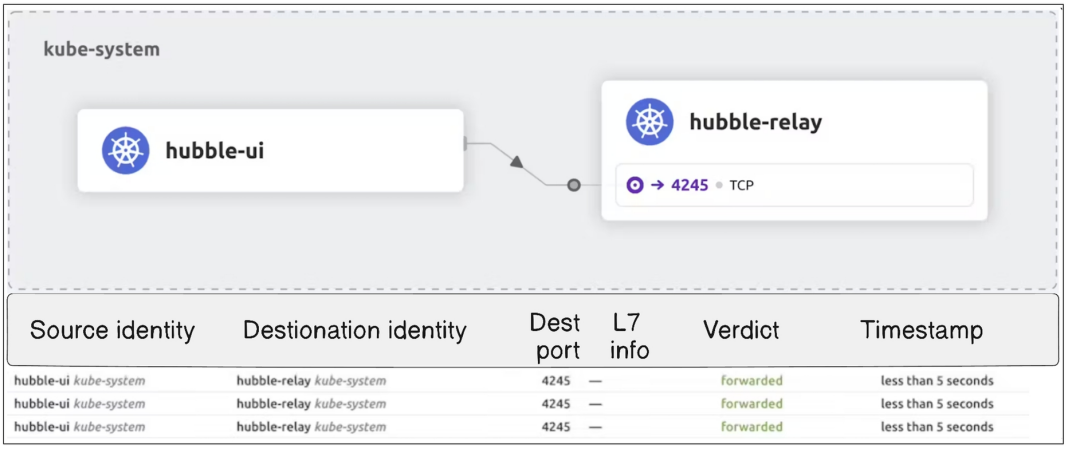
A new tab is opened in your window browser to navigate to http://localhost:12000. If none tab is opened automatically, you can do it by your own. Go to the browser and you should see something like:



On that screen, you can see a list and drop down of the namespaces that exist in the cluster. Because? This is because each namespace has its own pods with respective flows that it will be collecting and showing to the user, this way the user has a better understanding of what is happening in an orderly and separate way.

Click on the drop-down menu located in the upper left corner. Select the “kube-system” option:





Great, this is called a service-map, because you can see the flows and connectivity information of the different components (services) that are present within the namespace.

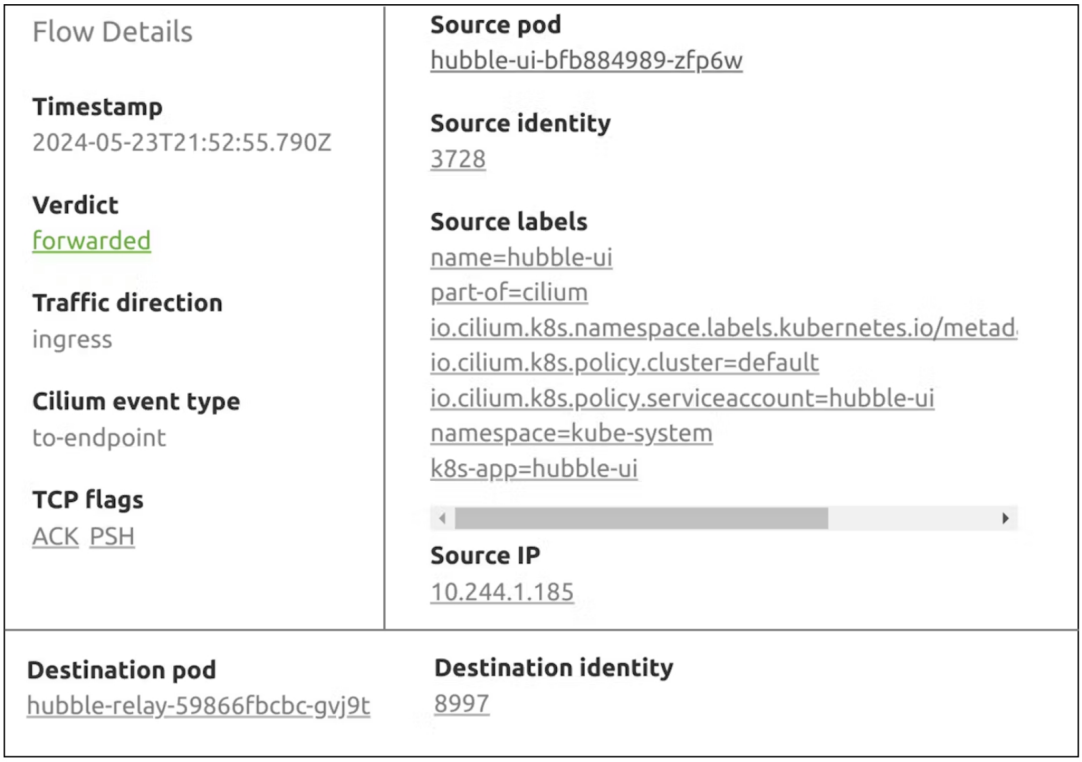
At the top, you can see the different components that are interacting in the namespace, their dependencies, forms of communication between them. For example, in the illustration, you can see how the hubble-ui service communicates with the hubble-relay service using the TCP protocol and port 4245.

At the bottom of the illustration, there is a table with the following columns that show the history of flows between services:

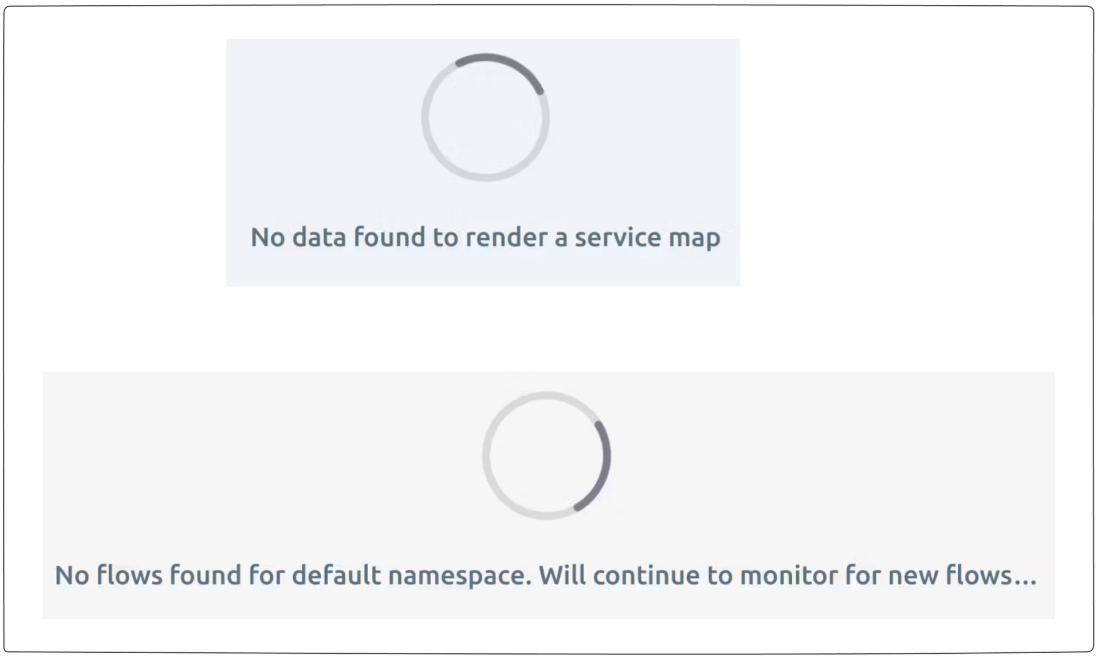
- Source identity: Origin of the flow event.
- Destination identity: Final destination of the flow event.

- Destination port: What is the destination port of the flow event.
- L7 Info: If the flow passes through layer 7 of the OSI model, it will record information regarding that.
- Verdict: Shows if the flow event was approved or rejected by cilium.
- Timestamp: They denote the time and date (or one of them) in which the flow event occurred.

In addition, if you click on whatever cell of the table, it appears a box with flow details:



On the other hand, if you choose a namespace that does not have services that talk to each other, you will see a screen like:



You probably want to see more example working right? OK, next we are going to deploy an application that is composed of 3 microservices, each with its pods and services that run on Kubertenes.

Then, run the following command in the console:

```
$ kubectl apply -k \
  github.com/falconcr/cilium-observability/chapter-4/kustomize-bases/podinfo
```

If you run `kubectl get ns`, you will see that a new namespace was created: `podinfo`. If you run `kubectl get services -n podinfo`, you can see that there are 3 services: podinfo-cient, podinfo-frontend, podinfo-backend.

The communication of the apps is as follows:

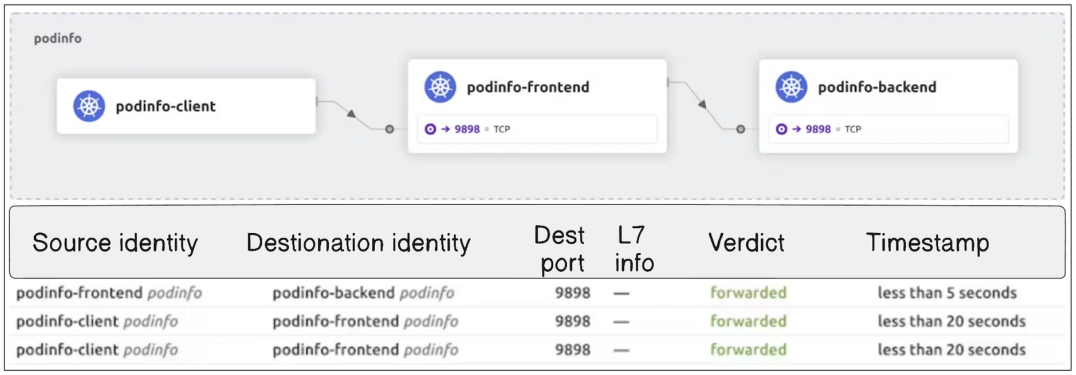
- The podinfo-client is like a CLI that allows you to make calls to the podinfo-frontend.
- When the podinfo-frontend receives a request, it calls the podinfo-backend.

How about we generate flows and data and then see them in the Hubble UI & Service Map?

To do this, let’s first make some calls to the podinfo-frontend via the podinfo-client:

```
$ kubectl -n podinfo \
  exec deployment/podinfo-client -c podinfod -- sh -c \
  'for i in $(seq 100) ; do \
  curl -S http://podinfo-frontend:9898/echo \
  done'
```

Now, we will go to the Hubble UI & Service Map webpage that enabled above,choose the podinfo namespace to see what flows are occurring:



Well, by seeing the different services communicating with each other, you can have a better idea of the interactions between them, and also be able to debug any network or communication problems in a more intuitive and graphic way.

Summary

The source code for this chapter is available in the book’s GitHub.[1]

As we conclude this exploration into the microservices cosmos, one truth becomes evident—Hubble Cilium is not just a tool; it’s a catalyst for mastery. Its role in providing deep visibility, enhancing security, and facilitating swift decision-making and troubleshooting propels microservices orchestration into a realm of unparalleled efficiency and innovation. Embark on a technical journey as we guide you through the intricacies of installing Loki in Kubertenes. Uncover the seamless integration with Grafana and learn hands-on techniques for effective log

analysis. From deployment to practical usage, this **Chapter 5, Log aggregation system: Gafana Loki** is your compass for mastering Loki's monitoring capabilities within the Kubertenes ecosystem. Navigate with precision and elevate your understanding of distributed system logging.

1. https://github.com/falconcr/cilium-observability/blob/main/chapter-2.md↩

5: Log Aggregation with Grafana Loki

Logs serve as a crucial component in understanding the inner workings of applications and the Kubertenes infrastructure. In this chapter, we delve into the significance of logs, their role in troubleshooting, monitoring, and security, and explore how log aggregation systems can enhance the management of log data. Specifically, we will focus on Grafana Loki, a popular log aggregation system in the Kubertenes ecosystem.

Log Collection in Kubertenes

In Kubertenes environments, log collection is vital for gaining insights into containerized applications and the underlying infrastructure. Key aspects of log collection in Kubertenes include:

Collecting and using logs in Kubertenes is crucial for several reasons, as it provides valuable insights into the health, performance, and behavior of applications and infrastructure. Here are some key reasons why log collection in Kubertenes is important:

1. **Troubleshooting and Debugging:** Logs help identify and troubleshoot issues within applications or the Kubertenes cluster itself. When something goes wrong, analyzing logs can provide detailed information about the sequence of events leading up to the problem, making it easier to identify and fix issues.

2. **Monitoring and Alerting:** Log data is essential for monitoring the performance of applications and the Kubertenes environment. By collecting and analyzing logs, you can set up alerts based on specific

patterns or anomalies, allowing you to proactively address potential problems before they impact the system.

3. **Security and Compliance:** Logs play a crucial role in detecting and investigating security incidents. By collecting and analyzing logs, you can monitor for suspicious activities, unauthorized access, or other security-related events. Logs are also valuable for meeting compliance requirements by providing an audit trail of system activities.

4. **Performance Optimization:** Analyzing logs helps in optimizing the performance of applications and the overall Kubertenes infrastructure. By identifying resource bottlenecks, errors, or inefficient processes, you can make informed decisions to improve system efficiency and enhance the user experience.

5. **Capacity Planning:** Log data is useful for capacity planning, helping you understand resource usage patterns over time. By analyzing logs, you can make informed decisions about scaling resources, whether it's adding more nodes to the cluster or adjusting resource allocations for specific workloads.

6. **Audit Trails and Historical Analysis:** Logs serve as a historical record of events within the system. This can be valuable for auditing purposes, providing a detailed trail of changes, access, and actions taken. It helps in understanding how the system has evolved and in diagnosing issues that occurred in the past.

7. **Operational Visibility:** Logs provide operational visibility into the state of applications and the Kubertenes cluster. This visibility is crucial for administrators and DevOps teams to understand what is happening within the system, making it easier to manage and maintain a healthy and efficient environment.

8. **Centralized Logging:** Kubertenes environments often involve multiple containers and nodes. Centralized logging solutions aggregate logs from various sources, making it easier to search, analyze, and visualize log data from a single location. This simplifies log management and enhances the overall observability of the system.

Log Formats in Kubertenes

Standardizing log formats is crucial for effective log analysis. Common log formats include:

- **JSON**: Providing structured data for easy parsing and analysis.
- **Plain Text**: Traditional log entries for human-readable information.
- **Syslog**: A standardized protocol for logging messages.
- **Custom Formats**: Tailoring log formats based on specific application or organizational requirements.

Choosing the JSON format for logging in containers offers several significant advantages compared to traditional log formats. Firstly, JSON provides a clear and readable data structure that facilitates the interpretation and analysis of logs. Each log entry is presented as a JSON object with key-value pairs, allowing for better contextualization of the recorded information. This is crucial for understanding complex events and quickly identifying specific information during troubleshooting or performance analysis.

Moreover, opting for the JSON format comes with advantages related to compatibility with widely used log-consuming tools today. Many log aggregation solutions, such as Elasticsearch, Loki, Fluentd, and Logstash (ELK Stack), and monitoring and visualization services like Grafana, are optimized to work with structured data in JSON format. The self-contained and semantic nature of JSON makes efficient indexing and querying of data possible, enabling operations and development teams to perform specific searches and analyses more accurately. This direct compatibility with leading log management tools makes choosing the JSON format a strategic option for ensuring seamless and effective integration in Kubertenes environments and modern log management systems.

Grafana Loki Overview

Grafana Loki is a log aggregation system designed for cloud-native environments, and it stands out for its lightweight, efficient, and scalable

approach to log collection and storage. Developed by Grafana Labs, Loki was specifically crafted to address the challenges of logging in dynamic containerized environments like Kubertenes. It is a part of the CNCF (Cloud Native Computing Foundation) and has gained popularity for its ability to provide observability without compromising on resource utilization.

Indexing in Grafana Loki:

Grafana Loki employs a unique indexing mechanism that sets it apart in the realm of log aggregation systems. Inspired by Prometheus, Loki utilizes label-based indexing to organize and retrieve log data efficiently. In this approach, logs are associated with key-value pairs, or labels, which provide contextual information about the log entry. This labeling system allows users to perform targeted searches and filter logs based on specific criteria, enabling a more granular and intuitive log exploration experience. By leveraging label-based indexing, Loki optimizes the retrieval of log data, enhancing the speed and precision of queries.

Log Storage and Compaction:

Loki is designed to be resource-efficient, particularly in terms of storage requirements. Traditional log management systems may face challenges with large volumes of logs over time, leading to increased storage needs. Loki addresses this by implementing log compaction and intelligent retention policies. Log compaction involves removing redundant log entries, optimizing storage utilization without sacrificing the ability to analyze historical data. Additionally, Loki’s retention policies allow users to define rules for how long log data should be retained, ensuring a balance between historical analysis and storage efficiency.

Challenges of Long-Term Log Storage:

Storing logs over an extended period can be a daunting task for log management systems. It often involves trade-offs between storage costs, query performance, and the need for historical data. Loki tackles these challenges by providing configurable retention policies, allowing organizations to strike the right balance based on their specific

requirements. By intelligently compacting logs and defining retention rules, Loki ensures that long-term storage remains both practical and cost-effective.

Scalability and Horizontal Architecture:

To handle the dynamic and distributed nature of modern cloud-native environments, Loki adopts a scalable and horizontally scalable architecture. Loki's distributed design allows it to scale horizontally by adding more instances or nodes to the system, accommodating growing log volumes and ensuring optimal performance. This scalability is essential for organizations dealing with large-scale containerized workloads in Kubertenes, where the number of logs generated can be substantial.

Integration with Grafana for Visualization:

A distinctive feature of Loki is its seamless integration with Grafana, a popular open-source dashboard and visualization platform. Logs stored in Loki can be effortlessly visualized alongside other monitoring metrics within the Grafana dashboard. This integration enhances the overall observability of the system by providing a unified platform for metrics and log analysis. Users can correlate log events with metrics, simplifying the process of identifying patterns, anomalies, and root causes during troubleshooting and analysis. The synergy between Grafana and Loki amplifies the power of log exploration and analysis in cloud-native environments.

Key Features of Grafana Loki:

1. **Distributed Architecture:** Loki employs a distributed and horizontally scalable architecture, allowing it to handle large volumes of logs across multiple Kubertenes clusters. This design ensures optimal performance and resilience, making it well-suited for dynamic and highly distributed environments.

2. **Label-Based Indexing:** Inspired by Prometheus, Loki uses label-based indexing to efficiently organize and query log data. This approach allows users to filter and retrieve logs based on specific

labels, providing a powerful and flexible mechanism for log exploration.

3. **Minimal Storage Requirements:** One of Loki's standout features is its ability to minimize storage requirements through log compaction and intelligent retention policies. This is particularly beneficial in environments with high log volumes, as it optimizes storage usage without compromising on the ability to analyze historical data.

4. **Prometheus Integration:** Loki seamlessly integrates with Prometheus, a popular monitoring and alerting toolkit. This integration allows users to correlate log data with metrics, providing a more comprehensive view of system behavior and facilitating root cause analysis during troubleshooting.

5. **LogQL Query Language:** Loki introduces LogQL, a query language tailored for log data. LogQL allows users to express complex queries and filters, making it easier to extract valuable insights from logs. The query language is designed to be intuitive and efficient for log analysis.

6. **Highly Configurable:** Grafana Loki is highly configurable and adaptable to various log formats and sources. Its flexibility makes it suitable for a wide range of applications and use cases, providing users with the ability to tailor the system to their specific logging requirements.

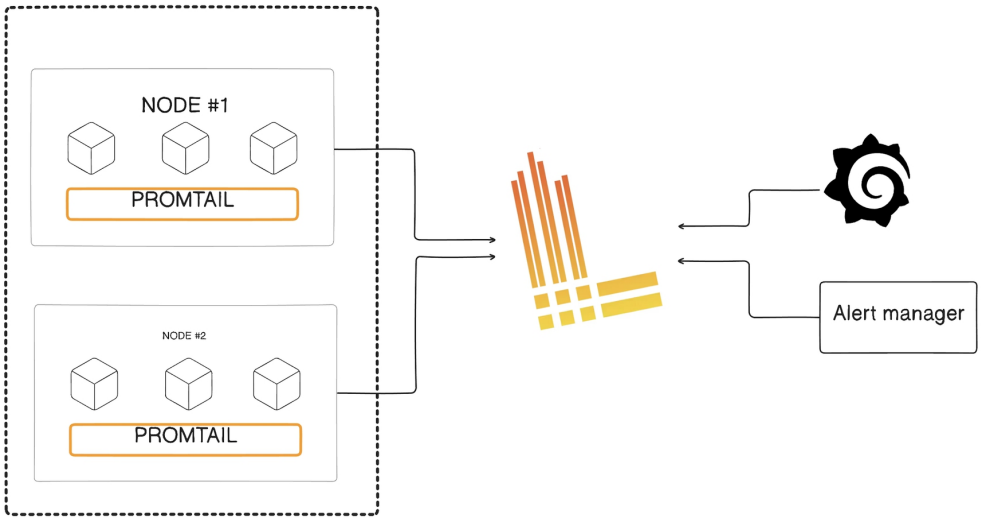
7. **Grafana Integration:** Grafana, a popular open-source dashboard and visualization platform, integrates seamlessly with Loki. This integration allows users to visualize log data alongside other monitoring metrics, providing a unified and cohesive observability platform.

8. **Kubertenes Native:** One of Loki's strengths is its native support for Kubertenes. Loki understands Kubertenes metadata, making it easier to filter and search logs based on container names, pod names, namespaces, and other Kubertenes-specific attributes. This native

integration simplifies the management of logs in a Kubertenes environment.

How does Grafana Loki work?

Grafana Loki operates as a log aggregation system designed for cloud-native environments, and it employs a unique architecture and set of components to efficiently collect, index, and store log data. At the core of Loki's functionality are Promtail, an agent responsible for log collection, and Grafana, a popular open-source dashboard and visualization platform.



Promtail for Log Collection:

Promtail is Grafana Loki's agent responsible for collecting logs from various sources, including application containers running in Kubertenes pods. It tailors logs from log files and standard output, adding labels and metadata to each log entry based on Kubertenes-specific information like pod name, container name, and namespace. Promtail uses the same label-based approach as Loki for log indexing, enhancing the granularity and efficiency of log queries. Its lightweight design allows it to be deployed as a sidecar container alongside application containers, ensuring minimal resource overhead.

Label-Based Indexing and Storage:

As logs are collected by Promtail, they are enriched with labels and metadata, making them easily searchable and filterable. Loki utilizes this label-based indexing to organize log entries, similar to how Prometheus indexes metrics data. This indexing mechanism enables users to perform precise and targeted queries, retrieving logs based on specific criteria. Loki's storage architecture is optimized for minimal resource consumption, employing log compaction and retention policies to efficiently manage storage while retaining the necessary historical data.

Integration with Grafana:

One of Loki's notable features is its seamless integration with Grafana. Grafana serves as the user interface for querying, visualizing, and analyzing log data stored in Loki. The integration allows users to create comprehensive dashboards that combine log data with other monitoring metrics. The synergy between Grafana and Loki enhances the overall observability of the system by providing a unified platform for metrics and log analysis. Users can correlate log events with metrics, streamlining the troubleshooting and analysis process. Grafana's powerful visualization capabilities complement Loki's efficient log indexing and storage.

In summary, Grafana Loki works by utilizing Promtail for log collection, employing label-based indexing for efficient log organization, adopting a distributed and scalable architecture for handling dynamic environments, and seamlessly integrating with Grafana for a unified observability experience. Together, these components make Loki a powerful solution for log aggregation and analysis in cloud-native architectures, especially in Kubertenes environments.

How Loki Differs from Other Log Collection Tools

Grafana Loki distinguishes itself from traditional log collection tools through its efficient use of resources, label-based indexing, and seamless integration with Grafana. Unlike some other log management systems that may be resource-intensive, Loki's design prioritizes scalability without compromising on performance. The label-based indexing, inspired by

Prometheus, allows for a more streamlined and effective log querying experience.

Moreover, Loki's emphasis on minimal storage requirements sets it apart from solutions that may face challenges in handling large volumes of logs over time. By intelligently compacting and retaining log data, Loki ensures that historical logs can be accessed without causing undue strain on storage infrastructure.

Integration with Kubertenes

Loki is tailored for Kubertenes environments, offering native support for the intricacies of container orchestration. It understands Kubertenes metadata, making it well-suited for filtering and querying logs in a Kubertenes-native manner. This native integration simplifies the implementation of Loki within Kubertenes clusters, aligning with the principles of observability and operational excellence in cloud-native architectures. With its ability to handle the dynamic nature of containerized workloads, Loki becomes an invaluable tool for gaining insights into the performance and behavior of applications in Kubertenes environments.

In summary, collecting and using logs in Kubertenes is essential for maintaining a healthy, secure, and efficient environment. Logs are a valuable source of information that can be leveraged for troubleshooting, monitoring, security, and overall system optimization.

LogQL

LogQL is the query language used in Loki, a log aggregation system designed for cloud-native environments. It serves as a powerful tool for searching and analyzing log data efficiently. LogQL was purpose-built to complement the distributed and dynamic nature of Kubertenes and microservices architectures.

At its core, LogQL simplifies the process of querying logs by offering a syntax that is both expressive and easy to understand. It allows users to filter and aggregate log entries based on labels, timestamps, and other

metadata, making it well-suited for navigating large volumes of log data. Its design is influenced by Prometheus's PromQL, providing a familiar experience for those already accustomed to working with Prometheus in monitoring setups.

One distinctive feature of LogQL is its ability to process logs in a horizontally scalable manner. Loki's architecture, combined with LogQL, enables efficient querying across vast log datasets distributed across multiple instances. This scalability aligns with the dynamic and elastic nature of modern containerized environments, ensuring that querying performance remains robust as the infrastructure scales.

In summary, LogQL is the query language that empowers users to interact with log data stored in Loki. Its user-friendly syntax, inspired by PromQL, and its scalability make it a valuable tool for navigating and extracting insights from the rich and diverse log information generated in cloud-native environments.

Here's a simple LogQL example that demonstrates how to use LogQL to query logs:

Suppose you have log entries with the following structure:

```
{"level":"info", "app":"web", "message":"Request received", "status":200}
{"level":"error", "app":"api", "message":"Internal server error", "status":500}
{"level":"info", "app":"web", "message":"Request processed", "status":200}
```

1. **Filtering by Log Level:**
    - To retrieve all logs with a level of "error," you can use the following LogQL query:

      ```
      {level="error"}
      ```

    This query filters logs where the "level" label is set to "error."

2. **Filtering by Application and Status Code:**
    - To retrieve logs from the "web" application with a status code of 200, you can use the following query:

```
{app="web", status=200}
```

This query filters logs where the “app” label is set to “web” and the “status” label is set to 200.

3. **Aggregating Log Entries:**

    - You can also aggregate log entries. For example, counting the number of logs for each application:

      ```
      count_over_time({}[$interval])
      ```

This query counts the number of logs over a specified time interval.

These are just basic examples, and LogQL supports more advanced features like regular expressions, grouping, and aggregations, allowing users to tailor their queries to extract specific information from their logs effectively. More information about that go to https://grafana.com/docs/loki/latest/query/.

Instalation

Installing Loki using Helm charts in Kubertenes is a streamlined and efficient way to set up this log aggregation system within your cluster. Helm is a package manager for Kubertenes that simplifies the deployment and management of applications. Below is a brief introduction to installing Loki using Helm charts in a Kubertenes environment:

1. Prerequisites:

Before installing Loki, ensure that you have Helm installed on your local machine and Tiller (Helm’s server-side component) deployed in your Kubertenes cluster.

2. Add Loki Helm Repository:

Add the Loki Helm repository to your Helm configuration:

```
$ helm repo add grafana https://grafana.github.io/helm-charts
```

You can customize the deployment by providing your own values.yaml file with configuration options. This step is optional, as Helm provides default values for Loki. Here's an example of how you might structure a values.yaml file:

```
loki:
  auth_enabled: false
  commonConfig:
    replication_factor: 1
  storage:
    type: 'filesystem'
singleBinary:
  replicas: 1
  persistence:
    storageClass: 'standard'
monitoring:
  selfMonitoring:
    enabled: false
test:
  enabled: false
```

Use Helm to install Loki into your Kubertenes cluster. You can customize the installation by providing the namespace and specifying your custom `values.yaml` file:

```
$ helm install \
  --values values.yaml loki \
  --namespace=monitoring grafana/loki
```

Create a `promtail.yaml` file and add the following:

```
initContainer:
  # # -- Specifies whether the init container for setting
  # # -- notify max user instances is to be enabled
   - name: init
  #   # -- Docker registry, image and tag for the init container image
     image: docker.io/busybox:1.33
  #   # -- Docker image pull policy for the init container image
     imagePullPolicy: IfNotPresent
  #   # -- The inotify max user instances to configure
     command:
       - sh
       - -c
       - sysctl -w fs.inotify.max_user_instances=512
     securityContext:
       privileged: true
```

Install the promtail helm chart using the above configuration file:

```
$ helm upgrade --install promtail \
  --values promtail.yaml \
  grafana/promtail -n monitoring
```

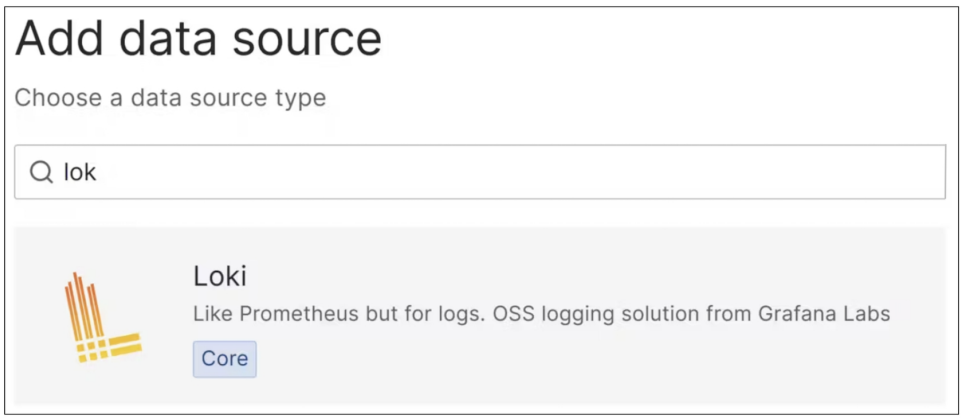
Configure Data Source

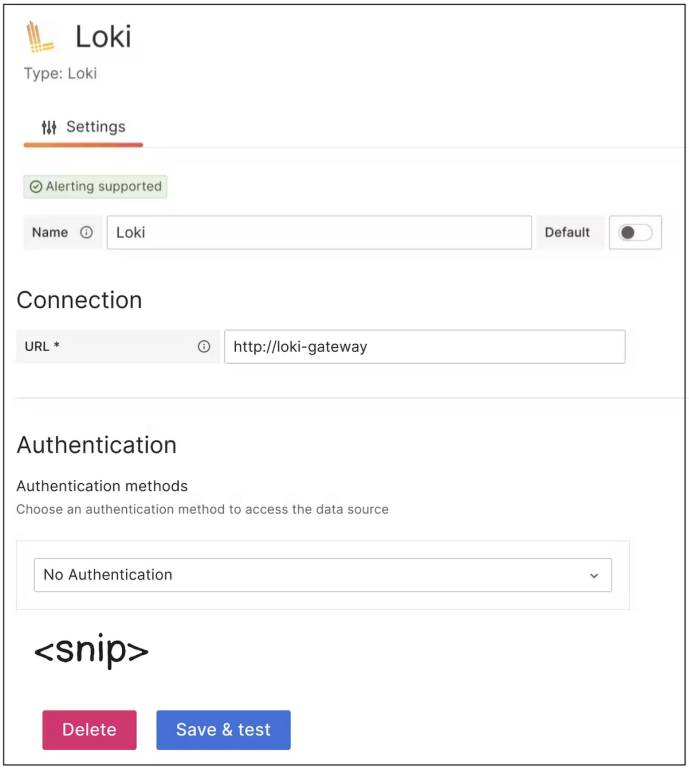
Port Forward to Grafana (If you already have it, ignore this step):

```
$ kubectl -n monitoring port-forward \
  service/grafana \
  --address 0.0.0.0 \
  --address :: 3000:3000
```

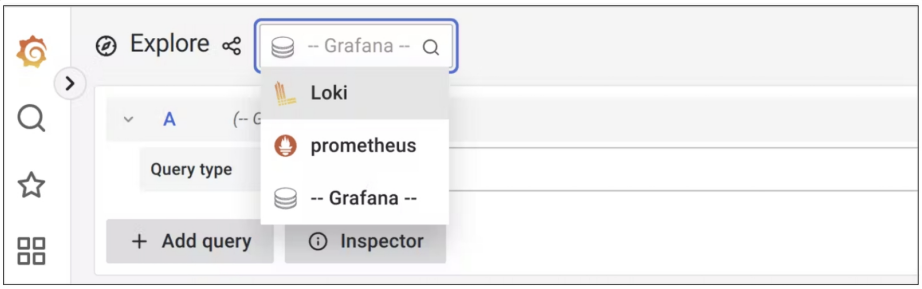
Open your web browser and navigate to http://localhost:3000. Once logged in, you will be prompted to add a data source:

- Click on “Add your first data source” or go to `Connections > Add new connection` in the side menu.
- Click on “Add your first data source.”
- Choose “Loki” from the list of available data sources.
- In the “HTTP” section, set the URL to http://loki-gateway (the default Loki service endpoint in the same namespace).
- Click “Save & Test” to verify the connection.



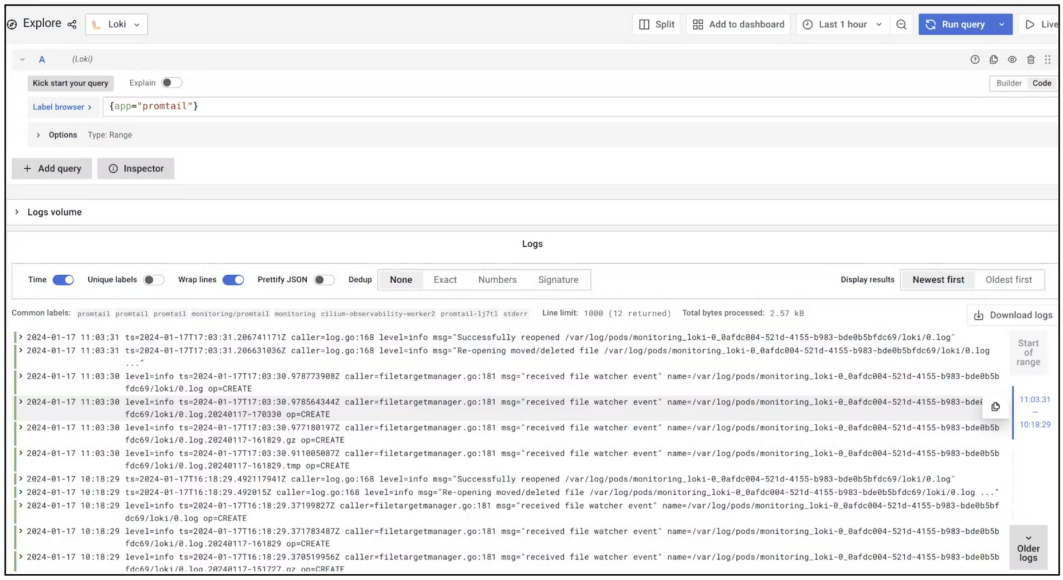


In the left sidebar, click on “Explore” (the compass icon). Ensure that “Loki” is selected as the data source in the top-left corner.



You should now be in the Explore view with the “Logs” tab selected. Enter a LogQL query in the query box to retrieve logs. For example, you can use a simple query like `{app="promtail"}`.

Press “Run Query” to see the logs matching your query.

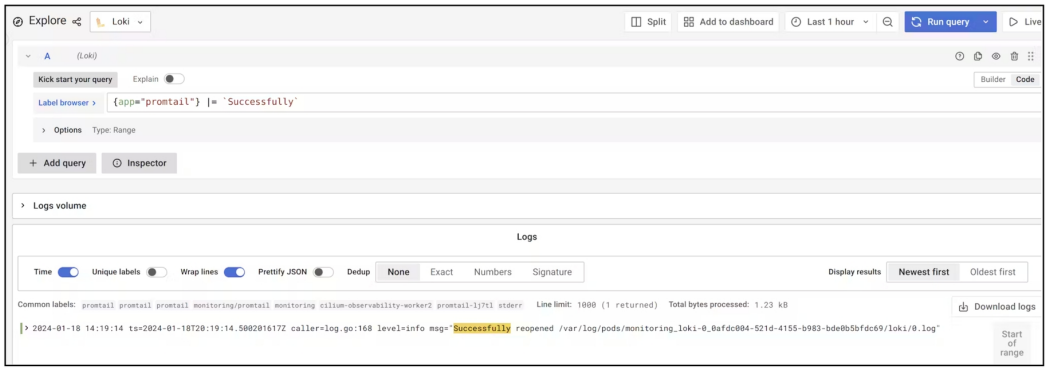


You can also search for text within the logs, in fact this is the option most used by developers, since it allows them to search for keywords within the log and thus make searches more efficient.

For example, if you want to search for all logs that have the label `app="promtail`and within that log contains the word “Successfully”, you can use something like this:

```
{app="promtail"} |= `Successfully`
```

Run the query and see the result:



Visualize and Analyze

Explore the logs visually, apply filters, and adjust the time range as needed. Grafana's Explore feature provides a powerful interface for interacting with your logs.

The logs are being collected by the promtail pods and sent to Loki. Thanks to Grafana, we can use its user interface to search and filter logs. The book is not intended to explain how to use Loki in depth, but at least to give you an introduction.

The Loki interface in Grafana provides a visual and analytical experience for efficiently exploring and querying logs. Here's an explanation of key sections:

1. Log Explorer: In the Log Explorer, you can write log queries using label expressions and text filters. Results are displayed in an interactive list view, making it easy to identify specific patterns and events.

2. Graphical Visualizations: Grafana allows you to create graphical visualizations based on retrieved logs. You can generate panels to represent line charts, bar charts, tables, and more, providing a better understanding of log data.

3. Labels and Filters: You can filter logs using specific labels or apply filters to narrow down results and focus on relevant information. This makes searching and analyzing specific events within large datasets easier.

4. Temporal Scaling: The Loki interface in Grafana provides controls for temporal scaling, allowing you to adjust the time range to view specific logs, whether in real-time or in past intervals.

5. Export and Sharing: Grafana enables the export of visualizations and query results for sharing with other team members. You can also save and load queries for easy reuse.

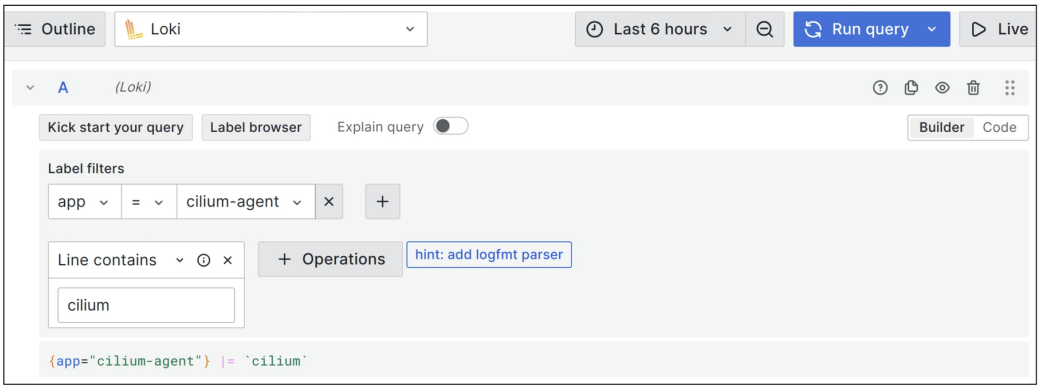
6. Intuitive Graphic Interface: The interface itself is designed to be intuitive, with navigation tools, search capabilities, and customization

options that make log exploration efficient and accessible.

7. Alerts and Notifications: You can configure alerts based on Loki queries and receive notifications when specific events or patterns are met, aiding in proactive issue identification.

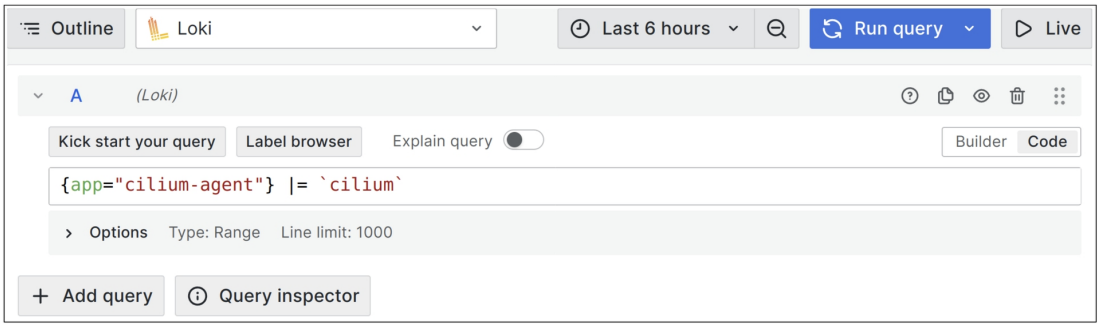
In the Loki interface in Grafana, the **Builder** and **Code** tabs are key components that provide specific functionalities for creating and refining log queries.

Builder Tab



- **Main Functionality:** The “Builder” tab is a graphical and intuitive interface that simplifies the construction of log queries without the need to manually write code.
- **Advantages:**
    - **Query Builders:** Offers visual assistants to select and filter labels, making it easy to create complex queries.
    - **Autocompletion:** Provides autocompletion and suggestions, reducing errors and speeding up the query-building process.
    - **Interactive Preview:** Shows a real-time preview of query results as you construct the query.

Code Tab



- **Main Functionality:** The “Code” tab provides a code view of the log query, allowing users to write and edit queries directly in code format.
- **Advantages:**
    - **Precise Control:** Allows experienced users to have more precise and granular control by directly writing queries in the Loki query language.
    - **Advanced Customization:** Facilitates the implementation of more complex and customized queries that may not be as straightforward to achieve using the “Builder” tab.
    - **Reuse:** Enables users to copy, share, and reuse log queries in their code format.

You can continue experimenting with the Loki tool, the key is to learn more about LogQL, syntax and good practices.

Summary

The source code for this chapter is available in the book’s GitHub.[1]

In this chapter, we delved into the implementation and benefits of the log aggregation system Grafana Loki. From its seamless integration with Kubertenes to its efficient handling of volumes of logs, Loki emerged as a

powerful tool for log aggregation and observability in complex environments.

We explored how to configure and leverage Loki with Grafana to gain valuable insights into the performance and health of our systems. As we move forward to **Chapter 6, Navigating the Trace Path in Kubertenes with Grafana Tempo** , we will dive into the fascinating realm of Grafana Tempo. We'll uncover what Tempo is, learn how to install it, and explore the collection and visualization of traces to gain a deeper understanding of our applications and microservices. Join us in this exciting continuation of our journey towards comprehensive observability!

1. https://github.com/falconcr/cilium-observability/blob/main/chapter-2.md↩

6: Navigating the Trace Path in Kubertenes with Grafana Tempo

Welcome to the exciting world of tracing in Kubertenes with Grafana Tempo! In this chapter, we embark on a journey to explore the intricacies of tracing and its significance in the realm of cloud-native applications. As experts in OpenTelemetry and Kubertenes, we understand the critical role that traces play in gaining insights into the behavior and performance of distributed systems.

Traces offer us a holistic view of the flow of requests as they traverse through various components of our applications, allowing us to pinpoint bottlenecks, troubleshoot issues, and optimize performance. With the emergence of Kubertenes as the de facto platform for deploying microservices at scale, the need for effective tracing mechanisms has never been more pronounced.

In this chapter, we delve deep into the fundamentals of tracing, uncovering why it’s essential in the context of cloud-native architectures. We explore how OpenTelemetry, an open-source observability framework, empowers us to instrument our applications seamlessly, capturing rich telemetry data across distributed environments.

Furthermore, we’ll discover the power of Grafana Tempo, a horizontally scalable, distributed tracing backend designed specifically for Kubertenes environments. By leveraging Tempo’s integration with Prometheus and Grafana, we’ll learn how to visualize, analyze, and derive insights from our traces with ease.

So, join us as we navigate the trace path in Kubertenes with Grafana Tempo, unraveling the mysteries of observability and empowering ourselves to build resilient, high-performing applications in the cloud-native era.

What are traces and their importance?

Imagine you're navigating through an unfamiliar city, relying on a detailed map showing every street and intersection. Traces in the world of distributed applications are like that detailed map, but for your computer systems.

A trace is essentially a detailed recording of the journey a request takes through the different parts of your application, from the moment it's received to when it's completed. Instead of looking at the application as a whole, traces allow you to dive into each step of the process, much like following the path of a package through a mail system.

Now, what are the benefits of these traces? Think of it this way: when something goes wrong in your application, like slow response times or unexpected errors, traces are like a magnifying glass that lets you pinpoint exactly where the problem lies. By following the path traced by a request, you can spot bottlenecks, understand the flow of data between components, and ultimately troubleshoot issues more quickly.

Moreover, traces aren't just useful when things go awry; they can also be an invaluable tool for optimizing your application's performance and efficiency. By understanding how your requests behave under normal conditions, you can identify areas for improvement and take steps to optimize your application before problems arise.

In summary, traces are like a detailed map that guides you through the intricacies of your distributed application, allowing you to identify issues, optimize performance, and keep your application running smoothly.

What is distributed tracing in Kubertenes?

Distributed tracing in Kubertenes is a method of monitoring and troubleshooting complex, distributed applications deployed on Kubertenes clusters. It involves capturing and correlating trace data from multiple services or components involved in processing a single user request or transaction across a distributed system.

In Kubertenes, where applications are often composed of numerous microservices running in containers across multiple pods and nodes, distributed tracing becomes essential for understanding how requests flow through the system and identifying performance bottlenecks or errors.

Distributed tracing typically involves instrumentation of applications with tracing libraries or agents that capture timing and context information for each request as it traverses through the various services. These traces are then collected by a centralized tracing backend, such as Jaeger or Grafana Tempo, where they are aggregated, stored, and visualized for analysis.

By leveraging distributed tracing in Kubertenes, developers and operators gain insights into the end-to-end latency, dependencies between services, and the overall health of their applications. This allows them to diagnose issues,optimize performance, and ensure a seamless user experience in a distributed, containerized environment.

Distributed tracing and logging are both essential components of observability in distributed systems, but they serve different purposes and provide distinct types of insights:

1. **Distributed Tracing:**
    - Distributed tracing focuses on capturing and correlating timing and context information for individual requests as they traverse through a distributed system.
    - It provides a detailed, end-to-end view of how requests flow through different services, enabling developers to understand the latency, dependencies, and performance of their applications.

- Traces are typically visualized as a sequence of spans, representing individual operations or segments of a request, and can be aggregated, analyzed, and traced back to identify bottlenecks or errors.

2. **Logging:**

   - Logging involves recording events, messages, or data generated by applications, services, or infrastructure components.
   - It provides a record of what happened within an application or system, including errors, warnings, informational messages, and debugging information.
   - Logs are valuable for troubleshooting issues, auditing, compliance, and monitoring the health and behavior of applications and infrastructure.

In summary, while both distributed tracing and logging contribute to observability, distributed tracing focuses on understanding the flow and performance of individual requests across a distributed system, while logging provides a record of events and activities within applications and infrastructure. Combining both approaches allows for comprehensive visibility and troubleshooting capabilities in complex, distributed environments like Kubertenes.

How distributed tracing works

Distributed tracing works by instrumenting applications deployed on Kubertenes clusters to capture timing and context information for individual requests as they traverse through various services. Here’s how it typically works:

1. **Instrumentation:**

   - Developers instrument their applications with tracing libraries or agents, such as OpenTelemetry or Jaeger, to generate trace data.
   - These libraries or agents create spans, which represent individual operations or segments of a request, and inject context

information, such as trace IDs and span IDs, into the request headers.

2. **Request Propagation:**

   - When a request enters the system, the tracing context is propagated through the different services involved in processing the request.
   - Kubertenes plays a crucial role in managing the deployment and scaling of these services, ensuring that requests are properly routed and load-balanced across pods and nodes.

3. **Span Creation:**

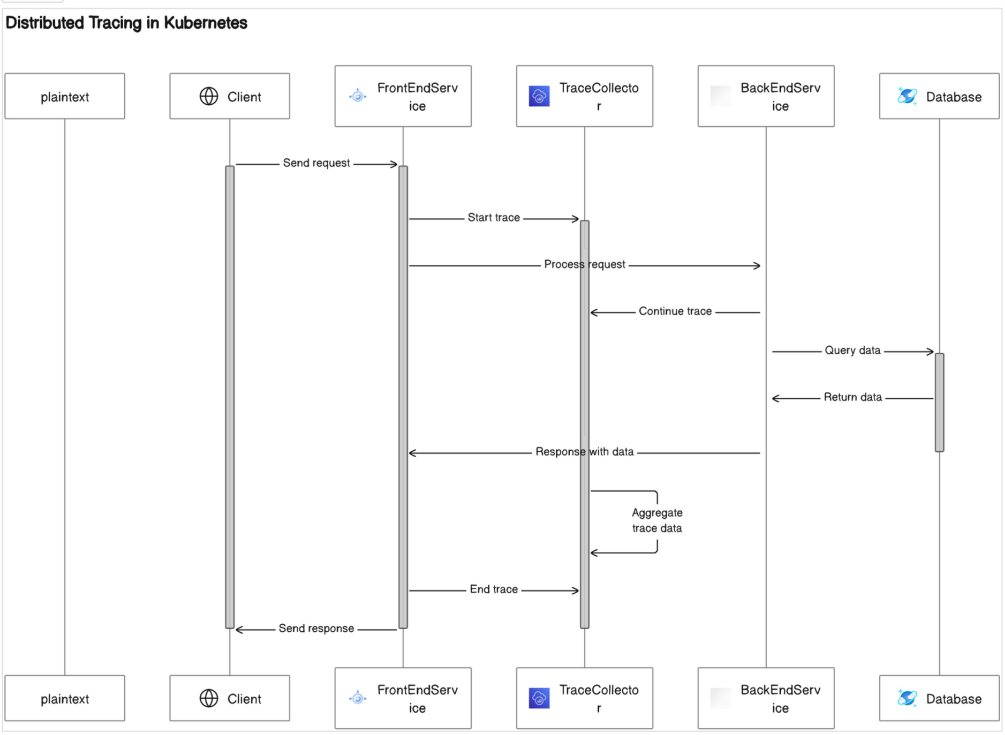
   - As the request flows through each service, spans are created to track the timing and execution of specific operations within that service.
   - Spans are linked together to form a trace, representing the entire journey of the request across the distributed system.

4. **Data Collection:**

   - The trace data, consisting of spans with timing information and contextual metadata, is collected by a centralized tracing backend, such as Jaeger or Grafana Tempo.
   - Kubertenes ensures that the tracing backend is deployed and accessible to all services within the cluster, facilitating the collection and aggregation of trace data.

5. **Visualization and Analysis:**

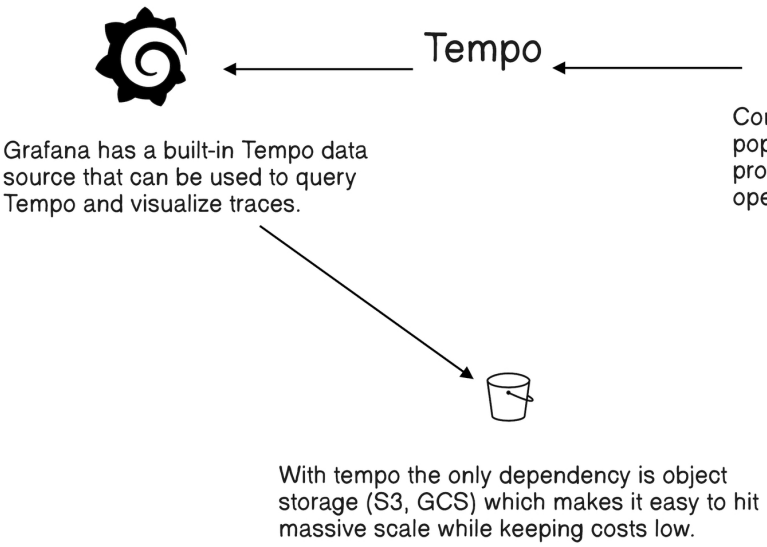
   - Once collected, the trace data can be visualized and analyzed using tracing UIs or integrated into monitoring dashboards.
   - Developers and operators can explore individual traces, identify performance bottlenecks or errors, and gain insights into the overall health and behavior of their applications.



In summary, distributed tracing in Kubertenes involves instrumenting applications to generate trace data, propagating tracing context through the system, creating spans to track the timing of operations, collecting trace data in a centralized backend, and visualizing and analyzing the data to gain insights into application performance and behavior.

How Grafana Tempo works

Grafana Tempo is a horizontally scalable, distributed tracing backend designed for high cardinality and long-term retention. It is built to handle the high volume of traces generated by modern, microservices-based applications and provides a scalable solution for storing and querying trace data.



Here’s how Grafana Tempo works:

1. **Distributed Architecture:**
    - Grafana Tempo is designed to be horizontally scalable, meaning it can handle large volumes of trace data by distributing the workload across multiple instances.
    - It follows a distributed architecture, allowing it to scale out horizontally by adding more instances as the volume of trace data increases.
2. **Simplified Storage:**
    - Tempo employs a simplified storage model based on object storage (e.g., Amazon S3 or Google Cloud Storage) to store trace data.
    - By leveraging object storage, Tempo can store vast amounts of trace data cost-effectively and with long-term retention.
3. **Index-Free Tracing:**
    - Unlike traditional tracing backends that rely on indexing trace data for querying, Tempo uses an index-free approach.

- Instead of indexing every span, Tempo stores spans directly in object storage, allowing for efficient storage and retrieval of trace data without the need for complex indexing infrastructure.

4. **Querying and Retrieval:**

   - Tempo supports querying and retrieval of trace data through a distributed query engine.
   - Queries are executed in parallel across multiple Tempo instances, allowing for fast and scalable retrieval of trace data.

5. **Integration with Grafana:**

   - Grafana Tempo is tightly integrated with Grafana, a popular open-source visualization and monitoring platform.
   - Users can visualize and analyze trace data directly within Grafana, leveraging Grafana's powerful visualization capabilities to gain insights into application performance and behavior.

In summary, Grafana Tempo is a horizontally scalable, distributed tracing backend that simplifies the storage and retrieval of trace data for modern, microservices-based applications. It provides a scalable solution for storing and querying large volumes of trace data cost-effectively, while also integrating seamlessly with Grafana for visualization and analysis.

Installing Tempo

To install and configure Tempo in Kubertenes using Helm, you can follow these steps:

Install Tempo using Helm, specifying the desired configuration options:

```
$ helm install tempo /grafana/tempo -n monitoring
```

Ensure that the Tempo pods are in the `Running` state.

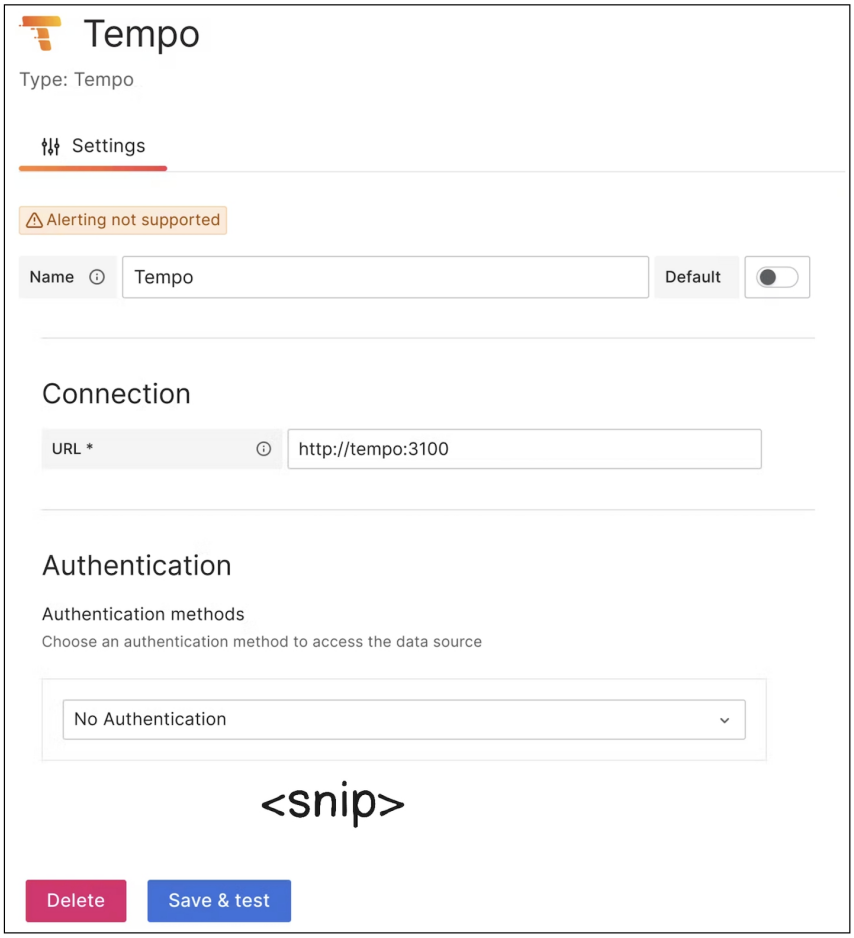
Access the Grafana UI using port-forwarding:

```
$ kubectl -n monitoring port-forward \
  service/grafana \
```

```
--address 0.0.0.0 \
--address :: 3000:3000
```

Open your browser and navigate to `http://localhost:3000`. Configure Tempo Data Source:

- In the Grafana UI, navigate to Connections > Add new connections.
- Select Tempo as the data source type.
- Configure the connection details for Tempo, including the URL of the Tempo server (`http://tempo:3001`).



Send traces to Tempo

The `xk6-client-tracing` is an extension module for k6, an open-source tool for load testing and stress testing. This module provides tracing capabilities for HTTP clients emulated by k6 during load test execution.

When using `xk6-client-tracing`, k6 can generate and send traces of HTTP requests made by virtual clients during test execution. These traces can contain detailed information about the start and finish times of requests, as well as any relevant events or metrics associated with each request.

The primary goal of using HTTP client tracing in k6 is to help developers and operations teams better understand the performance and behavior of their applications under simulated load. By examining the traces generated by k6, users can identify bottlenecks, failure points, and other areas for improvement in their systems, allowing them to optimize and enhance the performance of their applications.

When using `xk6-client-tracing` along with Grafana and Tempo, you can integrate k6-generated traces into Grafana dashboards for comprehensive observability and analysis of your application's performance. Here's a high-level overview of how to use it:

1. **Configure k6 with `xk6-client-tracing`:**
    - Build a custom k6 binary with the `xk6-client-tracing` extension module enabled. You can do this by compiling k6 with the `xk6-client-tracing` module included.
2. **Run Load Tests with k6:**
    - Use the custom k6 binary to run your load tests as usual. During test execution, k6 will generate traces of HTTP requests made by virtual clients.
3. **Export Traces to Tempo:**
    - Configure k6 to export the generated traces to Tempo, a distributed tracing backend for storing and analyzing trace data. You can do this by specifying the Tempo endpoint and any required authentication credentials in the k6 test script.
4. **Visualize Traces in Grafana:**

- Set up Grafana to visualize trace data from Tempo. Create dashboards in Grafana to display metrics, visualize latency distributions, and analyze the performance of your application under load.

By integrating k6-generated traces with Grafana and Tempo, you can gain deep insights into your application's performance characteristics, identify areas for optimization, and ensure that your systems meet performance and reliability requirements under real-world conditions.

Deploy the following resource in the cluster in order to `xk6-client-tracing` can send traces and Tempo:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: xk6-tracing
  namespace: monitoring
spec:
  minReadySeconds: 10
  replicas: 1
  revisionHistoryLimit: 10
  selector:
    matchLabels:
      app: xk6-tracing
      name: xk6-tracing
  template:
    metadata:
      labels:
        app: xk6-tracing
        name: xk6-tracing
    spec:
      containers:
      - env:
        - name: ENDPOINT
          value: tempo:4317
        image: ghcr.io/grafana/xk6-client-tracing:v0.0.2
        imagePullPolicy: IfNotPresent
        name: xk6-tracing
```

How to use Tempo in Grafana Explorer

Here's a step-by-step guide on how to use Tempo in Grafana Explorer:

1. **Access Grafana Explorer**:
    - Open your web browser and navigate to your Grafana instance.

- In the sidebar navigation menu, click on “Explorer” to open the metrics exploration tool.

2. **Select Tempo as Data Source**:

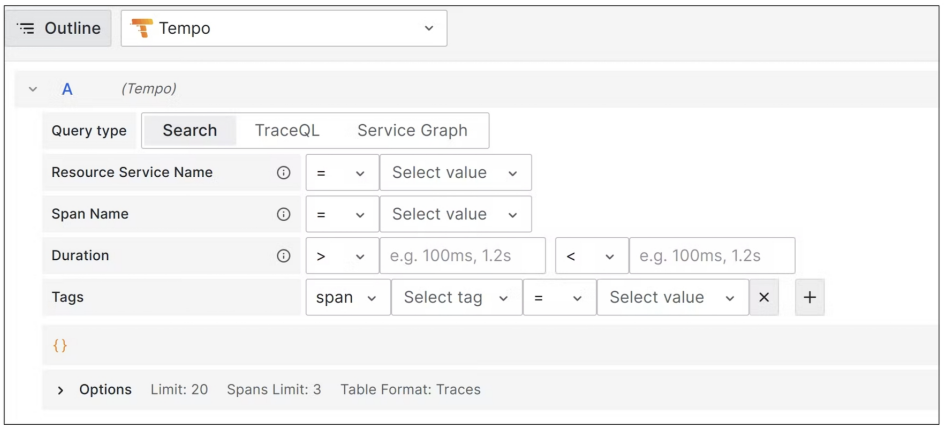
   - At the top left of the Explorer, ensure that the selected data source is Tempo. You can change the data source by clicking on the dropdown menu and selecting “Tempo”.

3. **Choose Query Type “Search”**:

   - Once you’ve selected Tempo as your data source, locate the section where you can enter your query.
   - Next to “Query Type”, select “Search” from the dropdown menu. This will allow you to search for Tempo traces based on different criteria.

4. **Configure the Query**:

   - In the query field, you can specify the search criteria for the Tempo traces you want to retrieve. For example, you can search for traces based on a specific service, operation, tag, etc.
   - Enter your query in the search field based on your needs. For example, you can type a specific tag that identifies a service or operation in your traces.



5. **Run the Query**:

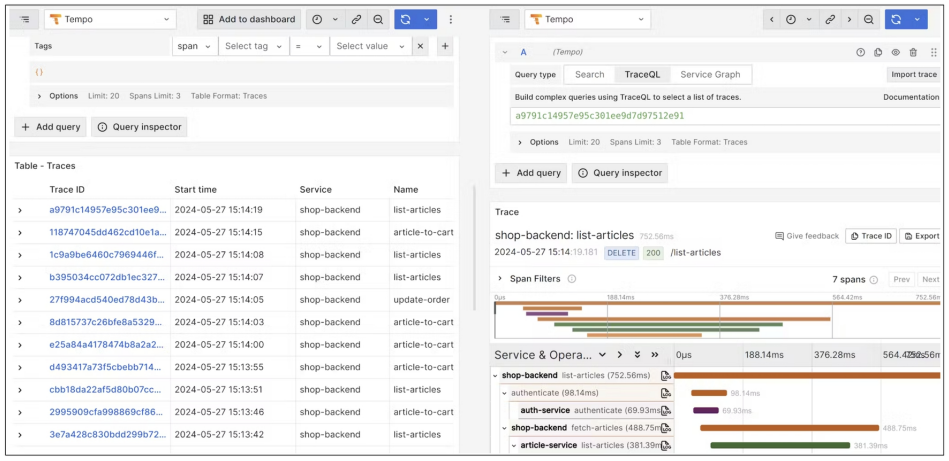
- Once you've configured your query, click the "Run Query" button to execute it.
- Grafana will send the query to Tempo and retrieve traces that match the specified criteria.

Table - Traces

	Trace ID	Start time	Service	Name	Duration
›	a9791c14957e95c301ee9...	2024-05-27 15:14:19	shop-backend	list-articles	752 ms
›	118747045dd462cd10e1a...	2024-05-27 15:14:15	shop-backend	article-to-cart	794 ms
›	1c9a9be6460c7969446f...	2024-05-27 15:14:08	shop-backend	list-articles	627 ms
›	b395034cc072db1ec327...	2024-05-27 15:14:07	shop-backend	list-articles	896 ms
›	27f994acd540ed78d43b...	2024-05-27 15:14:05	shop-backend	update-order	635 ms
›	8d815737c26bfe8a5329...	2024-05-27 15:14:03	shop-backend	article-to-cart	1.02 s
›	e25a84a4178474b8a2a2...	2024-05-27 15:14:00	shop-backend	article-to-cart	928 ms
›	d493417a73f5cbebb714...	2024-05-27 15:13:55	shop-backend	article-to-cart	1.14s
›	cbb18da22af5d80b07cc...	2024-05-27 15:13:51	shop-backend	list-articles	330 ms
›	2995909cfa998869cf86...	2024-05-27 15:13:46	shop-backend	article-to-cart	881 ms
›	3e7a428c830bdd299b72...	2024-05-27 15:13:42	shop-backend	list-articles	516 ms

6. **Explore the Results**:

- After running the query, Grafana will display the results at the bottom of the screen. Here you can see the Tempo traces that match your query.
- You can explore the details of each trace, such as duration, tags, associated logs, etc.



7. **Refine the Query (Optional)**:
    - If needed, you can refine your query and rerun it to get more specific results. You can adjust the search criteria as necessary to find the traces you're interested in.

Summary

The source code for this chapter is available in the book's GitHub.[1]

In this chapter, you've learned how to use Tempo in Grafana Explorer to search for Tempo traces using the "Search" query type. Now you can explore and analyze your application traces in a more efficient way.

1. https://github.com/falconcr/cilium-observability/blob/main/chapter-2.md↩

7: Advancing your observability journey

Congratulations on completing “Kubernetes Observability in Action”! Throughout this book, we have journeyed through the intricate world of Kubernetes observability, leveraging powerful tools such as Cilium, Grafana, Prometheus, Loki, and Tempo. Here are some key takeaways from our exploration:

1. **Comprehensive Observability**: We have seen how to establish a robust observability stack that provides deep insights into the performance and health of a Kubernetes cluster. By integrating Cilium with Grafana, Prometheus, Loki, and Tempo, we achieve full-stack observability from metrics and logs to traces.

2. **Cilium and eBPF**: We delved into the capabilities of Cilium powered by eBPF, enabling efficient networking, security, and observability within Kubernetes clusters. The flexibility and performance benefits of eBPF were highlighted as key advantages for modern cloud-native environments.

3. **Monitoring and Visualization**: Through Grafana and Prometheus, we learned how to collect, visualize on metrics, providing actionable insights to maintain the performance and reliability of our applications.

4. **Centralized Logging**: Loki allowed us to aggregate and query logs effectively, ensuring that we have a centralized logging solution that scales with our applications.

5. **Distributed Tracing**: With Tempo, we explored distributed tracing, which is crucial for understanding the flow of requests through

complex microservices architectures and diagnosing latency issues.

It’s time to reflect on the incredible strides we’ve made and look forward to the vast horizons ahead. From the foundational concepts of Kubernetes observability to the powerful integrations with Cilium, Grafana, Prometheus, Loki, and Tempo, you’ve equipped yourself with a formidable toolkit for maintaining and optimizing your cloud-native environments.

But this is just the beginning.

The world of observability is ever-evolving, presenting endless opportunities for those who are curious and driven. You’ve built a solid foundation, but there is always more to learn, explore, and create. Here are some parting thoughts to inspire your continued journey:

1. **Embrace Curiosity**: Never stop asking questions. The field of observability is full of nuanced challenges and intricate details that can always be explored further. Let your curiosity lead you to new discoveries and innovative solutions.

2. **Challenge Yourself**: Push the boundaries of what you know. Tackle advanced topics like custom eBPF programs, service mesh integration, and machine learning for anomaly detection. These challenges will not only deepen your knowledge but also sharpen your skills.

3. **Collaborate and Share**: The tech community thrives on collaboration. Share your insights, experiences, and innovations with others. Engage in communities, contribute to open-source projects, and learn from the experiences of your peers.

4. **Stay Resilient**: Observability is about resilience, both in systems and in learning. When faced with obstacles, remember that each challenge is an opportunity to grow stronger and more proficient.

5. **Innovate Fearlessly**: Don’t be afraid to experiment and innovate. The landscape of cloud-native technologies is ripe for creative solutions and bold ideas. Your contributions could shape the future of observability.

Future Challenges and Advanced Topics

As you move beyond the foundational concepts covered in this book, here are some advanced topics and challenges to consider for further exploration:

1. **Advanced eBPF Programs**: Dive deeper into writing custom eBPF programs to tailor observability and security to your specific needs. Explore advanced eBPF features and use cases beyond what Cilium offers out-of-the-box.

2. **Service Mesh Integration**: Integrate your observability stack with a service mesh like Istio or Linkerd. This will enhance traffic management, security, and observability for microservices communication.

3. **Chaos Engineering**: Implement chaos engineering principles using tools like Chaos Mesh or LitmusChaos to test the resilience and robustness of your Kubernetes cluster under unpredictable conditions.

4. **Automated Incident Response**: Develop automated workflows for incident response and remediation using tools like Prometheus Alertmanager, Grafana on-call, and Kubernetes operators.

5. **Machine Learning for Anomaly Detection**: Incorporate machine learning techniques to detect anomalies and predict failures in your observability data. Tools like Prometheus Anomaly Detection can help in this regard.

6. **Scaling Observability**: Address the challenges of scaling your observability stack to handle large clusters and high-throughput applications. Consider best practices for optimizing the performance and cost of your observability tools.

7. **Security Observability**: Enhance your observability setup to include security monitoring. Use Cilium’s security features, along with tools like Falco, to detect and respond to security threats in real-time.

8. **Custom Dashboards and Reports**: Create custom dashboards and reports tailored to different stakeholders within your organization. Use Grafana's advanced features to provide insightful visualizations that drive decision-making.

By continuing to build on the knowledge and skills acquired from this book, you will be well-equipped to tackle these advanced topics and challenges, further enhancing the observability and resilience of your Kubernetes environments. Keep experimenting, learning, and innovating—there's always more to discover in the ever-evolving landscape of cloud-native technologies.

Happy observing

As you embark on this next phase of your journey, carry with you the knowledge and confidence you've gained from this book. The path ahead is filled with possibilities, and you have the tools to navigate it with skill and vision. Keep learning, keep experimenting, and keep pushing the boundaries of what's possible.

The world of observability is yours to explore. Go forth and create something extraordinary.

Thank you for being a part of this journey. I look forward to seeing the remarkable things you'll achieve.

Happy learning and happy observing!