📚 Observability 101 Guide

PREFACE

Welcome to the 7-Chapter Observability 101 Guide by Abhishek Veeramalla (www.youtube.com/@abhishekveeramalla).

This Guide contains a detailed explanation of Observability with Kubernetes using Prometheus, Grafana, Elasticsearch, Fluent Bit, Kibana, Jaeger and open telemetry.

This 7-Chapter Observability 101 Guide is also available as a video tutorial course at https://www.youtube.com/playlist?list=PLdpzxOOAlwvJUIfwmmVDoPYqXXUNbdBmb

ABOUT THE AUTHOR

Abhishek Veeramalla is known for his Free Zero to Hero DevOps and Cloud Video courses on YouTube. His youtube channel has helped more than 1000 aspirants get placed into the DevOps and Cloud roles. Some of his popular zero to hero courses include -

DevOps Zero to Hero
AWS Zero to Hero
Azure Zero to Hero
Kubernetes Zero to Hero
Terraform Zero to Hero
Ansible Zero to Hero
Python for DevOps

You can find more details at www.youtube.com/@abhishekveeramalla

📅 Overview of Seven Chapters

Chapter 1: Introduction to Observability

- **Concepts Covered**:
  - Introduction to Observability, Monitoring, Logging, and Tracing.
  - The difference between Monitoring and Observability.
  - Tools available for Monitoring and Observability.
  - Comparison between monitoring and observing in Bare-Metal Servers vs. Kubernetes.
- **Key Learning**:
  - Understand the fundamental concepts of observability.
  - Learn why monitoring and observability are crucial in modern IT environments.

Chapter 2: Prometheus - Setting Up Monitoring

- **Concepts Covered**:
  - Introduction to Prometheus and its architecture.
  - Setup and configuration of Prometheus in an EKS cluster.
  - Installation of kube-prometheus-stack with Helm and integrating it with Grafana.
  - Basic queries and setup for monitoring with Prometheus and Grafana.
- **Key Learning**:
  - Get hands-on experience with Prometheus and Grafana.
  - Learn to install and configure Prometheus on Kubernetes.

Chapter 3: Metrics and PromQL in Prometheus

- **Concepts Covered**:
  - Introduction to PromQL and basic querying techniques.
  - Aggregation and functions in PromQL to analyze metrics data.
- **Key Learning**:

    - Master the Prometheus Query Language (PromQL) for querying and analyzing metrics.

Chapter 4: Instrumentation and Custom Metrics

- **Concepts Covered**:
    - Instrumentation for adding monitoring capabilities to applications.
    - Understanding different types of metrics in Prometheus: Counter, Gauge, Histogram, and Summary.
    - Writing custom metrics in a Node.js application using the `prom-client` library.
    - Dockerization of the application and deploying it on Kubernetes.
    - Setting up Alertmanager for alerting based on custom metrics.
- **Key Learning**:
    - Learn how to instrument applications to expose custom metrics.
    - Configure alerts in Alertmanager to monitor application performance.
    - Understand how to work with different types of metrics in Prometheus.

Chapter 5: Logging with EFK Stack

- **Concepts Covered**:
    - Introduction to logging in distributed systems and Kubernetes.
    - Setting up the EFK stack (Elasticsearch, Fluentbit, Kibana) on Kubernetes.
    - Detailed setup and configuration for collecting and visualizing logs.
    - Cleaning up the Kubernetes cluster and resources.
- **Key Learning**:
    - Understand the importance of logging and how to set up

Chapter 6: Distributed Tracing with Jaeger

- **Concepts Covered**:
  - Introduction to Jaeger and its architecture for distributed tracing.
  - Setting up Jaeger in a Kubernetes cluster using Helm.
  - Instrumenting services using OpenTelemetry to enable tracing.
  - Viewing and analyzing traces in the Jaeger UI.
  - Cleaning up the environment after setting up Jaeger.
- **Key Learning**:
  - Gain insights into distributed tracing and how it helps in debugging and performance optimization.
  - Learn how to set up and configure Jaeger for tracing in a microservices architecture.

Chapter 7: OpenTelemetry – Setting Up Unified Observability

- **Concepts Covered**:
  - Introduction to OpenTelemetry, a unified framework for observability.
  - Understanding how OpenTelemetry integrates tracing, metrics, and logging.
  - Comparison of OpenTelemetry with prior observability tools like Jaeger, Prometheus
  - Supported programming languages and multi-language support in OpenTelemetry.
  - Step-by-step setup of OpenTelemetry in Kubernetes.
- **Key Learning**:
  - Learn how OpenTelemetry simplifies the process of collecting and exporting telemetry data.
  - Understand the benefits of a unified observability approach using OpenTelemetry.
  - Gain hands-on experience with setting up OpenTelemetry Collector, Prometheus, Jaeger, and Elasticsearch to monitor a Golang microservice application.

💡 Chapter 1: Introduction to Observability

- Observability is the ability to understand the internal state of a system by analyzing the data it produces, including logs, metrics, and traces.

- Monitoring(Metrics): involves tracking system metrics like CPU usage, memory usage, and network performance. Provides alerts based on predefined thresholds and conditions
    - `Monitoring tells us what is happening.`
- Logging(Logs): involves the collection of log data from various components of a system.
    - `Logging explains why it is happening.`
- Tracing(Traces): involves tracking the flow of a request or transaction as it moves through different services and components within a system.
    - `Tracing shows how it is happening.`

Observability

Metrics

Logs

Traces

Monitoring

Tells us what is happening

Logging

Explains why it is happening

Tracing

Shows how it is happening

🤔 Why Monitoring?

- Monitoring helps us keep an eye on our systems to ensure they are working properly.
- Purpose: maintaining the **health, performance, and security** of IT environments.
- It enables early detection of issues, ensuring that they can be addressed before causing significant downtime or data loss.
- We use monitoring to:
    - Detect Problems Early
    - Measure Performance:
    - Ensure Availability:

🤔 Why Observability?

- Observability helps us understand why our systems are behaving the way they are.
- It's like having a detailed map and tools to explore and diagnose issues.
- We use observability to:
    - Diagnose Issues:
    - Understand Behavior:
    - Improve Systems:

Why Monitoring?

Detect Problems Early

Measure Performance

Ensure Availability

Why Observability?

Diagnose Issues

Understand Behavior

Improve Systems

🆚 What is the Exact Difference Between Monitoring and Observability?

🔥 Monitoring is the *when and what* of a system error, and observability is the *why and how*

Category	Monitoring	Observability
Focus	Checking if everything is working as expected	Understanding why things are happening in the system
Data	Collects metrics like CPU usage, memory usage, and error rates	Collects logs, metrics, and traces to provide a full picture
Alerts	Sends notifications when something goes wrong	Correlates events and anomalies to identify root causes
Example	If a server's CPU usage goes above 90%, monitoring will alert us	If a website is slow, observability helps us trace the user's request through different services to find the bottleneck
Insight	Identifies potential issues before they become critical	Helps diagnose issues and understand system behavior

🔭 Does Observability Cover Monitoring?

- Yes!! Monitoring is subset of Observability
- Observability is a broader concept that includes monitoring as one of its components.
- monitoring focuses on tracking specific metrics and alerting on predefined conditions
- observability provides a comprehensive understanding of the system by collecting and analyzing a wider range of data, including **logs, metrics, and traces**.

🖥️ What Can Be Monitored?

- Infrastructure: CPU usage, memory usage, disk I/O, network traffic.
- Applications: Response times, error rates, throughput.
- Databases: Query performance, connection pool usage, transaction rates.
- Network: Latency, packet loss, bandwidth usage.
- Security: Unauthorized access attempts, vulnerability scans, firewall logs.

👀 What Can Be Observed?

- Logs: Detailed records of events and transactions within the system.
- Metrics: Quantitative data points like CPU load, memory consumption, and request counts.
- Traces: Data that shows the flow of requests through various services and components.

🆚 Monitoring on Bare-Metal Servers vs. Monitoring Kubernetes

- Bare-Metal Servers:
    - Direct Access: Easier access to hardware metrics and logs.
    - Fewer Layers: Simpler environment with fewer abstraction layers.
- Kubernetes:

    - Dynamic Environment: Challenges with monitoring ephemeral containers and dynamic scaling.
    - Distributed Nature: Requires tools that can handle distributed systems and correlate data from multiple sources.

🆚 Observing on Bare-Metal Servers vs. Observing Kubernetes

- Bare-Metal Servers:
    - Simpler Observability: Easier to collect and correlate logs, metrics, and traces due to fewer components and layers.
- Kubernetes:
    - Complex Observability: Requires sophisticated tools to handle the dynamic and distributed nature of containers and microservices.
    - Integration: Necessitates the integration of multiple observability tools to get a complete picture of the system.

⚒️ What are the Tools Available?

- **Monitoring Tools**: Prometheus, Grafana, Nagios, Zabbix, PRTG.
- **Observability Tools**: ELK Stack (Elasticsearch, Logstash, Kibana), EFK Stack (Elasticsearch, FluentBit, Kibana) Splunk, Jaeger, Zipkin, New Relic, Dynatrace, Datadog.

Chapter 2: Monitoring

Metrics vs Monitoring

Metrics are measurements or data points that tell you what is happening. For example, the number of steps you walk each day, your heart rate, or the temperature outside—these are all metrics.

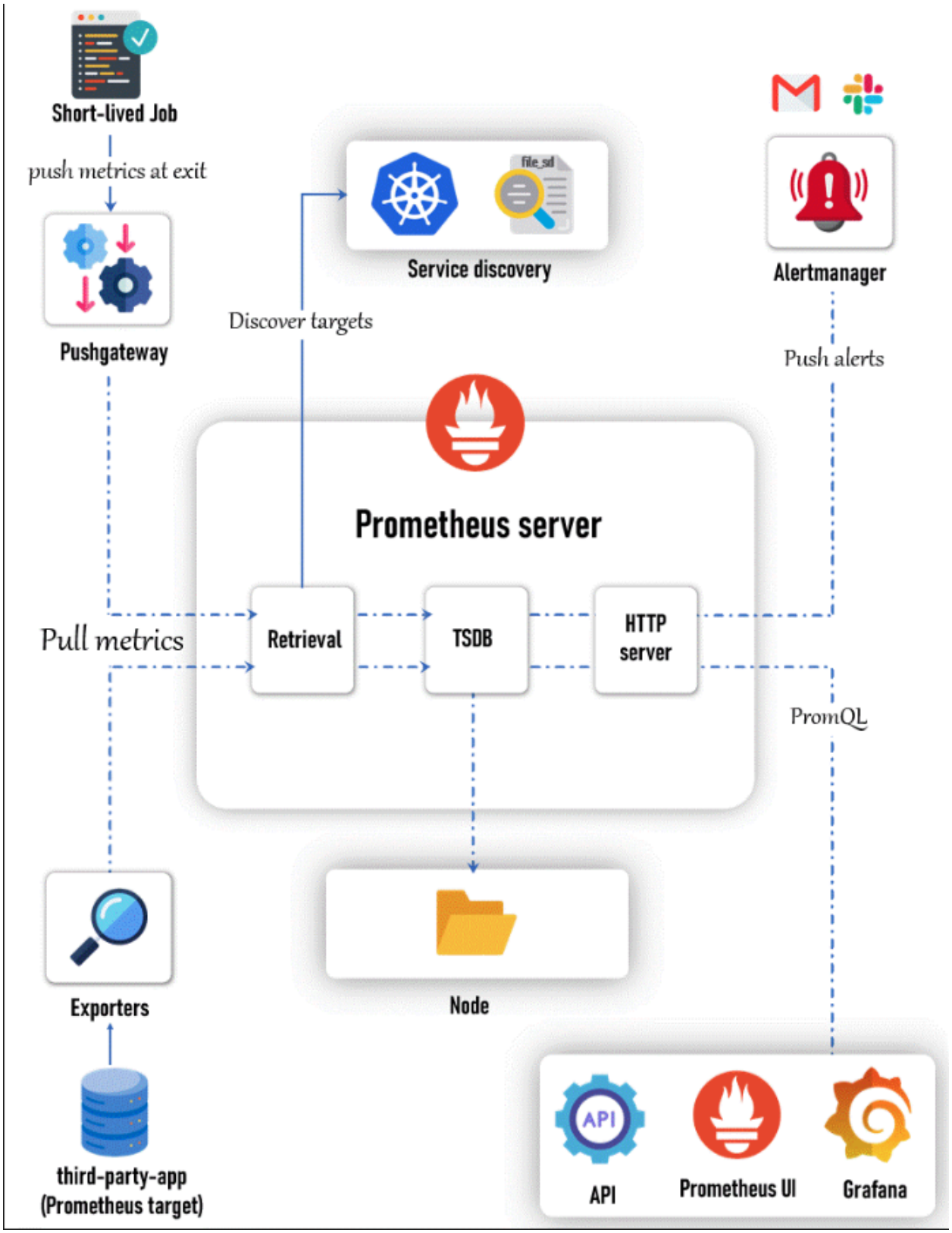
Monitoring is the process of keeping an eye on these metrics over time to understand what's normal, identify changes, and detect problems. It's like watching your step count daily to see if you're meeting your fitness goal or checking your heart rate to make sure it's in a healthy range.

🚀 Prometheus

- Prometheus is an open-source systems monitoring and alerting toolkit originally built at SoundCloud.
- It is known for its robust data model, powerful query language (PromQL), and the ability to generate alerts based on the collected time-series data.
- It can be configured and set up on both bare-metal servers and container environments like Kubernetes.

🏠 Prometheus Architecture

- The architecture of Prometheus is designed to be highly flexible, scalable, and modular.
- It consists of several core components, each responsible for a specific aspect of the monitoring process.



🔥 Prometheus Server

- The Prometheus server is the core of the monitoring system. It is responsible for scraping metrics from various configured targets, storing them in its time-series database (TSDB), and serving queries through its HTTP API.
- Components:
    - **Retrieval**: This module handles the scraping of metrics from endpoints, which are discovered either through static configurations or dynamic service discovery methods.
    - **TSDB (Time Series Database)**: The data scraped from targets is stored in the TSDB, which is designed to handle high volumes of time-series data efficiently.
    - **HTTP Server**: This provides an API for querying data using PromQL, retrieving metadata, and interacting with other components of the Prometheus ecosystem.
- **Storage**: The scraped data is stored on local disk (HDD/SSD) in a format optimized for time-series data.

🌐 Service Discovery

- Service discovery automatically identifies and manages the list of scrape targets (i.e., services or applications) that Prometheus monitors.
- This is crucial in dynamic environments like Kubernetes where services are constantly being created and destroyed.
- Components:
    - **Kubernetes**: In Kubernetes environments, Prometheus can automatically discover services, pods, and nodes using Kubernetes API, ensuring it monitors the most up-to-date list of targets.
    - **File SD (Service Discovery)**: Prometheus can also read static target configurations from files, allowing for flexibility in environments where dynamic service discovery is not used.

📤 Pushgateway

- The Pushgateway is used to expose metrics from short-lived jobs or applications that cannot be scraped directly by Prometheus.
- These jobs push their metrics to the Pushgateway, which then makes them available for Prometheus to scrape(pull).
- Use Case:
    - It's particularly useful for batch jobs or tasks that have a limited lifespan and would otherwise not have their metrics collected.

🚨 Alertmanager

- The Alertmanager is responsible for managing alerts generated by the Prometheus server.
- It takes care of deduplicating, grouping, and routing alerts to the appropriate notification channels such as PagerDuty, email, or Slack.

🧲 Exporters

- Exporters are small applications that collect metrics from various third-party systems and expose them in a format Prometheus can scrape. They are essential for monitoring systems that do not natively support Prometheus.
- Types of Exporters:
    - Common exporters include the Node Exporter (for hardware metrics), the MySQL Exporter (for database metrics), and various other application-specific exporters.

🖥️ Prometheus Web UI

- The Prometheus Web UI allows users to explore the collected metrics data, run ad-hoc PromQL queries, and visualize the results directly within Prometheus.

📊 Grafana

- Grafana is a powerful dashboard and visualization tool that integrates with Prometheus to provide rich, customizable visualizations of the metrics data.

🔌 API Clients

- API clients interact with Prometheus through its HTTP API to fetch data, query metrics, and integrate Prometheus with other systems or custom applications.

🛠️ Installation & Configurations

📦 Step 1: Create EKS Cluster

Prerequisites

- Download and Install AWS Cli
- Setup and configure AWS CLI using the `aws configure` command.
- Install and configure eksctl.
- Install and configure kubectl.

Clone the GitHub Repo
https://github.com/iam-veeramalla/observability-zero-to-hero

```
cd observability-zero-to-hero
```

```
eksctl create cluster --name=observability \
                      --region=us-east-1 \
                      --zones=us-east-1a,us-east-1b \
                      --without-nodegroup
```

```
eksctl utils associate-iam-oidc-provider \
    --region us-east-1 \
    --cluster observability \
    --approve


eksctl create nodegroup --cluster=observability \
                        --region=us-east-1 \

--name=observability-ng-private \
                        --node-type=t3.medium \
                        --nodes-min=2 \
                        --nodes-max=3 \
                        --node-volume-size=20 \
                        --managed \
                        --asg-access \
                        --external-dns-access \
                        --full-ecr-access \
                        --appmesh-access \
                        --alb-ingress-access \
                        --node-private-networking

# Update ./kube/config file
aws eks update-kubeconfig --name observability
```

🧰 Step 2: Install kube-prometheus-stack

```
helm repo add prometheus-community
https://prometheus-community.github.io/helm-charts
helm repo update
```

🚀 Step 3: Deploy the chart into a new namespace "monitoring"

```
kubectl create ns monitoring

cd day-2

helm install monitoring
prometheus-community/kube-prometheus-stack \
-n monitoring \
-f ./custom_kube_prometheus_stack.yml
```

✅ Step 4: Verify the Installation

```
kubectl get all -n monitoring
```

- **Prometheus UI**:

```
kubectl port-forward service/prometheus-operated -n
monitoring 9090:9090
```

- **Grafana UI**: password is `prom-operator`

```
kubectl port-forward service/monitoring-grafana -n
monitoring 8080:80
```

- **Alertmanager UI**:

```
kubectl port-forward service/alertmanager-operated -n
monitoring 9093:9093
```

🧼 Step 5: Clean UP

- **Uninstall helm chart**:

```
helm uninstall monitoring --namespace monitoring
```

- **Delete namespace**:

```
kubectl delete ns monitoring
```

- **Delete Cluster & everything else**:

```
eksctl delete cluster --name observability
```

📊 Chapter 3: Metrics in Prometheus

- Metrics in Prometheus are the core data objects that represent measurements collected from monitored systems.
- These metrics provide insights into various aspects of **system performance, health, and behavior**.

🏷️ Labels:

- Metrics are paired with Labels.
- Labels are key-value pairs that allow you to differentiate between dimensions of a metric, such as different services, instances, or endpoints.

🔍 Example:

```
container_cpu_usage_seconds_total{namespace="kube-system", endpoint="https-metrics"}
```

- `container_cpu_usage_seconds_total` is the metric.
- `{namespace="kube-system", endpoint="https-metrics"}` are the labels.

🛠️ What is PromQL?

- PromQL (Prometheus Query Language) is a powerful and flexible query language used to query data from Prometheus.
- It allows you to retrieve and manipulate time series data, perform mathematical operations, aggregate data, and much more.
- 🔑 Key Features of PromQL:
  - Selecting Time Series: You can select specific metrics with filters and retrieve their data.
  - Mathematical Operations: PromQL allows for mathematical operations on metrics.

    - Aggregation: You can aggregate data across multiple time series.
    - Functionality: PromQL includes a wide range of functions to analyze and manipulate data.

💡 Basic Examples of PromQL

- `container_cpu_usage_seconds_total`
    - Return all time series with the metric container_cpu_usage_seconds_total
- `container_cpu_usage_seconds_total{namespace="kube-system",pod=~"kube-proxy.*"}`
    - Return all time series with the metric `container_cpu_usage_seconds_total` and the given `namespace` and `pod` labels.
- `container_cpu_usage_seconds_total{namespace="kube-system",pod=~"kube-proxy.*"}[5m]`
    - Return a whole range of time (in this case 5 minutes up to the query time) for the same vector, making it a range vector.

⚙️ Aggregation & Functions in PromQL

- Aggregation in PromQL allows you to combine multiple time series into a single one, based on certain labels.

Sum Up All CPU Usage:

```
sum(rate(node_cpu_seconds_total[5m]))
```

-
    - This query aggregates the CPU usage across all nodes.

Average Memory Usage per Namespace:

```
avg(container_memory_usage_bytes) by (namespace)
```

-
    - This query provides the average memory usage grouped by namespace.

- **rate() Function:**
    - The rate() function calculates the per-second average rate of increase of the time series in a specified range.

```
rate(container_cpu_usage_seconds_total[5m])
```

-
    - This calculates the rate of CPU usage over 5 minutes.
- **increase() Function:**
    - The increase() function returns the increase in a counter over a specified time range.

```
increase(kube_pod_container_status_restarts_total[1h])
```

-
    - This gives the total increase in container restarts over the last hour.
- **histogram_quantile() Function:**
    - The histogram_quantile() function calculates quantiles (e.g., 95th percentile) from histogram data.

```
histogram_quantile(0.95,
sum(rate(apiserver_request_duration_seconds_bucket[5m]
)) by (le))
```

    - This calculates the 95th percentile of Kubernetes API request durations.

Chapter 4: Instrumentation

- Instrumentation refers to the process of adding monitoring capabilities to your applications, systems, or services.
- This involves embedding/Writing code or using tools to collect metrics, logs, or traces that provide insights into how the system is performing.

🎯 Purpose of Instrumentation:

- **Visibility**: It helps you gain visibility into the internal state of your applications and infrastructure.
- **Metrics Collection**: By collecting key metrics like CPU usage, memory consumption, request rates, error rates, etc., you can understand the health and performance of your system.
- **Troubleshooting**: When something goes wrong, instrumentation allows you to diagnose the issue quickly by providing detailed insights.

⚙️ How it Works:

- **Code-Level Instrumentation**: You can add instrumentation directly in your application code to expose metrics. For example, in a `Node.js` application, you might use a library like prom-client to expose custom metrics.

📈 Instrumentation in Prometheus:

- 📤 **Exporters**: Prometheus uses exporters to collect metrics from different systems. These exporters expose metrics in a format that Prometheus can scrape and store.
    - **Node Exporter**: Collects system-level metrics from Linux/Unix systems.

    - **MySQL Exporter (For MySQL Database)**: Collects metrics from a MySQL database.
    - **PostgreSQL Exporter (For PostgreSQL Database)**: Collects metrics from a PostgreSQL database.
    - 📊 **Custom Metrics**: You can instrument your application to expose custom metrics that are relevant to your specific use case. For example, you might track the number of user logins per minute.

📈 Types of Metrics in Prometheus

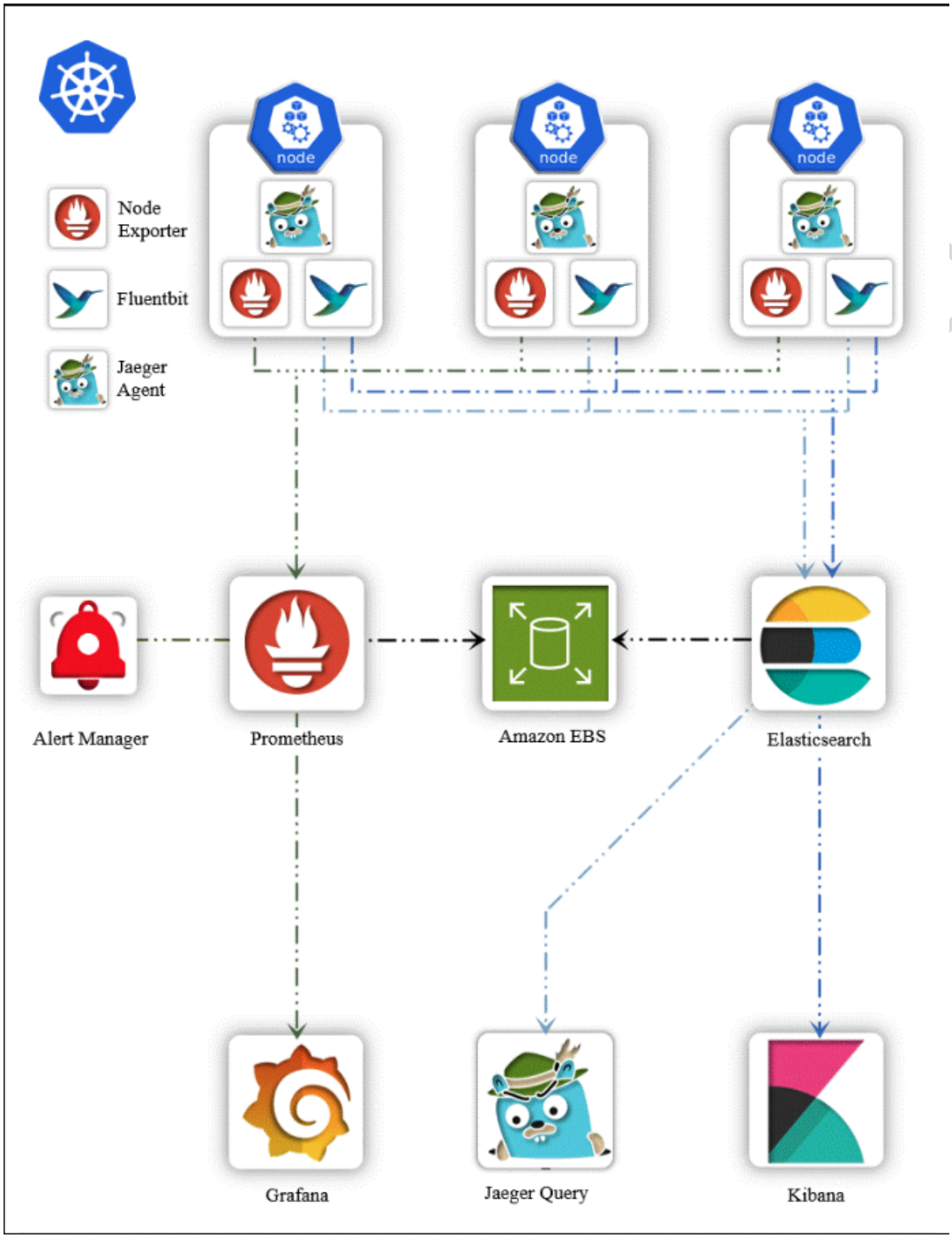
- 🔄 **Counter**:
    - A Counter is a cumulative metric that represents a single numerical value that only ever goes up. It is used for counting events like the number of HTTP requests, errors, or tasks completed.
    - **Example**: Counting the number of times a container restarts in your Kubernetes cluster
    - **Metric Example**: `kube_pod_container_status_restarts_total`
- 📏 **Gauge**:
    - A Gauge is a metric that represents a single numerical value that can go up and down. It is typically used for things like memory usage, CPU usage, or the current number of active users.
    - **Example**: Monitoring the memory usage of a container in your Kubernetes cluster.
    - **Metric Example**: `container_memory_usage_bytes`
- 📊 **Histogram**:
    - A Histogram samples observations (usually things like request durations or response sizes) and counts them in configurable buckets.

    - It also provides a sum of all observed values and a count of observations.
    - **Example**: Measuring the response time of Kubernetes API requests in various time buckets.
    - **Metric Example**: `apiserver_request_duration_seconds_bucket`
- 📝 Summary:
    - Similar to a Histogram, a Summary samples observations and provides a total count of observations, their sum, and configurable quantiles (percentiles).
    - **Example**: Monitoring the 95th percentile of request durations to understand high latency in your Kubernetes API.
    - **Metric Example**: `apiserver_request_duration_seconds_sum`

🎯 Project Objectives

- 🛠️ **Implement Custom Metrics in Node.js Application**: Use the prom-client library to write and expose custom metrics in the Node.js application.
- 🚨 **Set Up Alerts in Alertmanager**: Configure Alertmanager to send email notifications if a container crashes more than two times.
- 📝 **Set Up Logging**: Implement logging on both application and cluster (node) logs for better observability using EFK stack(Elasticsearch, FluentBit, Kibana).
- 📸 **Implement Distributed Tracing for Node.js Application**: Enhance observability by instrumenting the Node.js application for distributed tracing using Jaeger. enabling better performance monitoring and troubleshooting of complex, multi-service architectures.

🏠 Architecture



Prerequisite

- Clone the GitHub Repo - https://github.com/iam-veeramalla/observability-zero-to-hero
- cd observability-zero-to-hero

1) Write Custom Metrics

- Please take a look at `day-4/application/service-a/index.js` file to learn more about custom metrics. below is the brief overview
- **Express Setup**: Initializes an Express application and sets up logging with Morgan.
- **Logging with Pino**: Defines a custom logging function using Pino for structured logging.
- **Prometheus Metrics with prom-client**: Integrates Prometheus for monitoring HTTP requests using the prom-client library:
  - `http_requests_total`: counter
  - `http_request_duration_seconds`: histogram
  - `http_request_duration_summary_seconds`: summary
  - `node_gauge_example`: gauge for tracking async task duration

Basic Routes:

- `/` : Returns a "Running" status.
- `/healthy`: Returns the health status of the server.
- `/serverError`: Simulates a 500 Internal Server Error.
- `/notFound`: Simulates a 404 Not Found error.
- `/logs`: Generates logs using the custom logging function.
- `/crash`: Simulates a server crash by exiting the process.
- `/example`: Tracks async task duration with a gauge.
- `/metrics`: Exposes Prometheus metrics endpoint.
- `/call-service-b`: To call service b & receive data from service b

2) dockerize & push it to the registry

- To containerize the applications and push it to your Docker registry, run the following commands:

```
cd day-4
```

```
# Dockerize microservice - a

docker build -t <<NAME_OF_YOUR_REPO>>:<<TAG>>
application/service-a/
```

```
# Dockerize microservice - b

docker build -t <<NAME_OF_YOUR_REPO>>:<<TAG>>
application/service-b/
```

3) Kubernetes manifest

- Review the Kubernetes manifest files located in `day-4/kubernetes-manifest`.
- Apply the Kubernetes manifest files to your cluster by running:

```
kubectl create ns dev

kubectl apply -k kubernetes-manifest/
```

4) Test all the endpoints

- Open a browser and get the LoadBalancer DNS name & hit the DNS name with following routes to test the application:
  - `/`
  - `/healthy`

    - `/serverError`
    - `/notFound`
    - `/logs`
    - `/example`
    - `/metrics`
    - `/call-service-b`
- Alternatively, you can run the automated script `test.sh`, which will automatically send random requests to the LoadBalancer and generate metrics:

```
./test.sh <<LOAD_BALANCER_DNS_NAME>>
```

5) Configure Alertmanager

- Review the Alertmanager configuration files located in `day-4/alerts-alertmanager-servicemonitor-manifest` but below is the brief overview
    - Before configuring Alertmanager, we need credentials to send emails. For this project, we are using Gmail, but any SMTP provider like AWS SES can be used. so please grab the credentials for that.
    - Open your Google account settings and search App password & create a new password & put the password in `day-4/alerts-alertmanager-servicemonitor-manifest/email-secret.yml`
    - One last thing, please add your email id in the `day-4/alerts-alertmanager-servicemonitor-manifest/alertmanagerconfig.yml`
- **HighCpuUsage**: Triggers a warning alert if the average CPU usage across instances exceeds 50% for more than 5 minutes.
- **PodRestart**: Triggers a critical alert immediately if any pod restarts more than 2 times.
- Apply the manifest files to your cluster by running:

```
kubectl apply -k
alerts-alertmanager-servicemonitor-manifest/
```

- Wait for 4-5 minutes and then check the Prometheus UI to confirm that the custom metrics implemented in the Node.js application are available:
  - `http_requests_total`: counter
  - `http_request_duration_seconds`: histogram
  - `http_request_duration_summary_seconds`: summary
  - `node_gauge_example`: gauge for tracking async task duration

6) Testing Alerts

- To test the alerting system, manually crash the container more than 2 times to trigger an alert (email notification).
- To crash the application container, hit the following endpoint
- `<<LOAD_BALANCER_DNS_NAME>>/crash`
- You should receive an email once the application container has restarted at least 3 times.

🔍 Chapter 5: Logging

- Logging is crucial in any distributed system, especially in Kubernetes, to monitor application behavior, detect issues, and ensure the smooth functioning of microservices.

🚀 Importance:

- **Debugging**: Logs provide critical information when debugging issues in applications.
- **Auditing**: Logs serve as an audit trail, showing what actions were taken and by whom.
- **Performance** Monitoring: Analyzing logs can help identify performance bottlenecks.
- **Security**: Logs help in detecting unauthorized access or malicious activities.

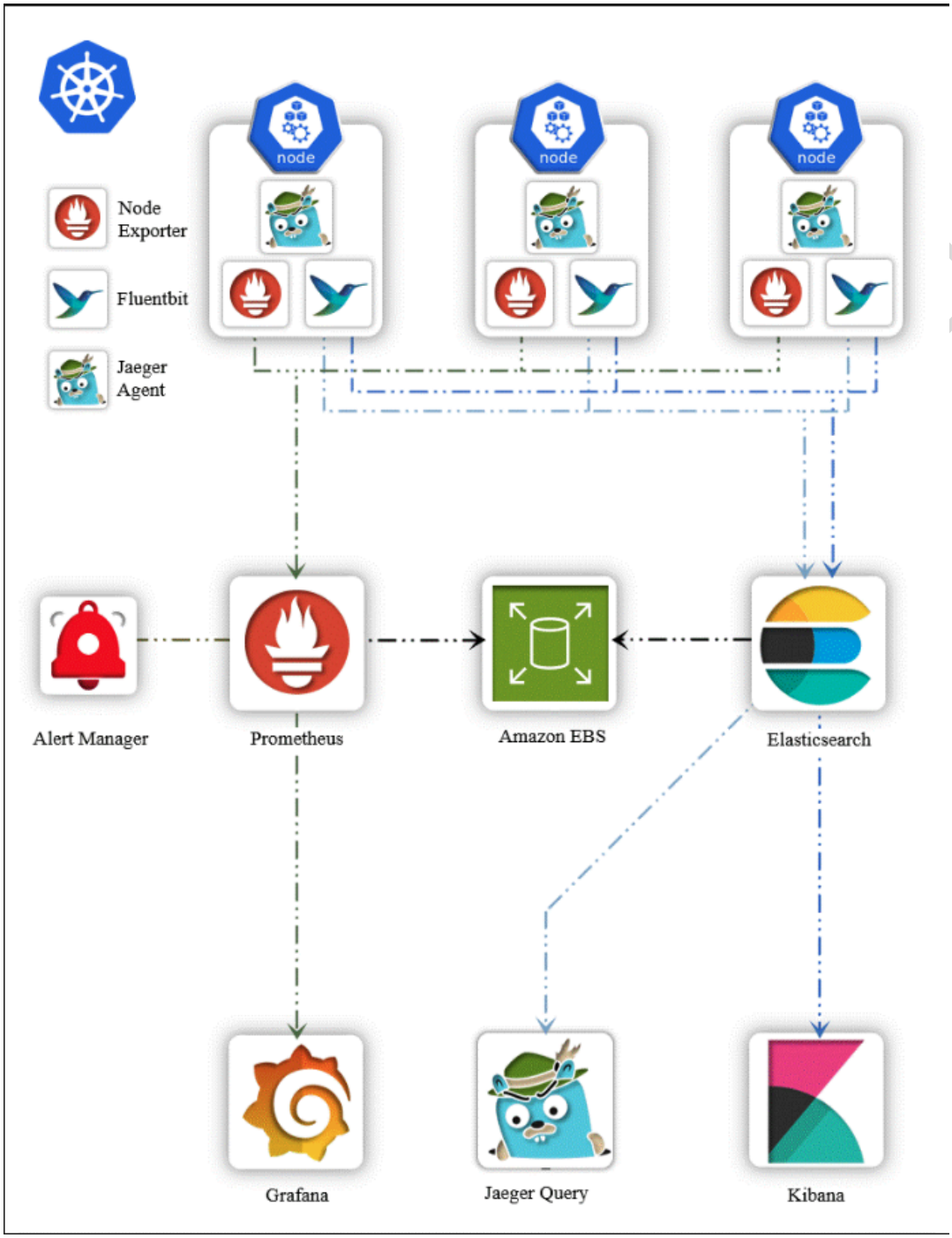
🛠️ Tools Available for Logging in Kubernetes

- 🗂️ EFK Stack (Elasticsearch, Fluentbit, Kibana)
- 🗂️ EFK Stack (Elasticsearch, FluentD, Kibana)
- 🗂️ ELK Stack (Elasticsearch, Logstash, Kibana)
- 📊 Promtail + Loki + Grafana

📦 EFK Stack (Elasticsearch, Fluentbit, Kibana)

- EFK is a popular logging stack used to collect, store, and analyze logs in Kubernetes.
- **Elasticsearch**: Stores and indexes log data for easy retrieval.
- **Fluentbit**: A lightweight log forwarder that collects logs from different sources and sends them to Elasticsearch.
- **Kibana**: A visualization tool that allows users to explore and analyze logs stored in Elasticsearch.

🏠 Architecture



📝 Step-by-Step Setup

Clone the GitHub Repo - https://github.com/iam-veeramalla/observability-zero-to-hero
cd observability-zero-to-hero

1) Create IAM Role for Service Account

```
eksctl create iamserviceaccount \
    --name ebs-csi-controller-sa \
    --namespace kube-system \
    --cluster observability \
    --role-name AmazonEKS_EBS_CSI_DriverRole \
    --role-only \
    --attach-policy-arn
arn:aws:iam::aws:policy/service-role/AmazonEBSCSIDrive
rPolicy \
    --approve
```

- This command creates an IAM role for the EBS CSI controller.
- IAM role allows EBS CSI controller to interact with AWS resources, specifically for managing EBS volumes in the Kubernetes cluster.
- We will attach the Role with service account

2) Retrieve IAM Role ARN

```
ARN=$(aws iam get-role --role-name
AmazonEKS_EBS_CSI_DriverRole --query 'Role.Arn'
--output text)
```

- Command retrieves the ARN of the IAM role created for the EBS CSI controller service account.

3) Deploy EBS CSI Driver

```
eksctl create addon --cluster observability --name
aws-ebs-csi-driver --version latest \
```

```
  --service-account-role-arn $ARN --force
```

- Above command deploys the AWS EBS CSI driver as an addon to your Kubernetes cluster.
- It uses the previously created IAM service account role to allow the driver to manage EBS volumes securely.

4) Create Namespace for Logging

```
kubectl create namespace logging
```

5) Install Elasticsearch on K8s

```
helm repo add elastic https://helm.elastic.co

helm install elasticsearch \
 --set replicas=1 \
 --set volumeClaimTemplate.storageClassName=gp2 \
 --set persistence.labels.enabled=true
elastic/elasticsearch -n logging
```

- Installs Elasticsearch in the `logging` namespace.
- It sets the number of replicas, specifies the storage class, and enables persistence labels to ensure data is stored on persistent volumes.

6) Retrieve Elasticsearch Username & Password

```
# for username
kubectl get secrets --namespace=logging
elasticsearch-master-credentials
-ojsonpath='{.data.username}' | base64 -d
# for password
```

```
kubectl get secrets --namespace=logging
elasticsearch-master-credentials
-ojsonpath='{.data.password}' | base64 -d
```

- Retrieves the password for the Elasticsearch cluster's master credentials from the Kubernetes secret.
- The password is base64 encoded, so it needs to be decoded before use.
- 👉 **Note**: Please write down the password for future reference

7) Install Kibana

```
helm install kibana --set service.type=LoadBalancer
elastic/kibana -n logging
```

- Kibana provides a user-friendly interface for exploring and visualizing data stored in Elasticsearch.
- It is exposed as a LoadBalancer service, making it accessible from outside the cluster.

8) Install Fluentbit with Custom Values/Configurations

- 👉 **Note**: Please update the `HTTP_Passwd` field in the `fluentbit-values.yml` file with the password retrieved earlier in step 6: (i.e NJyO47UqeYBsoaEU)"

```
helm install fluent-bit fluent/fluent-bit -f
fluentbit-values.yaml -n logging
```

✅ Conclusion

- We have successfully installed the EFK stack in our Kubernetes cluster, which includes Elasticsearch for storing logs, Fluentbit for collecting and forwarding logs, and Kibana for visualizing logs.

- To verify the setup, access the Kibana dashboard by entering the `LoadBalancer DNS name followed by :5601 in your browser.
    - `http://LOAD_BALANCER_DNS_NAME:5601`
- Use the username and password retrieved in step 6 to log in.
- Once logged in, create a new data view in Kibana and explore the logs collected from your Kubernetes cluster.

🧼 Clean Up

```
helm uninstall monitoring -n monitoring

helm uninstall fluent-bit -n logging

helm uninstall elasticsearch -n logging

helm uninstall kibana -n logging

cd day-4

kubectl delete -k kubernetes-manifest/

kubectl delete -k
alerts-alertmanager-servicemonitor-manifest/


eksctl delete cluster --name observability
```

🕵️ Chapter 6: Tracing using Jaeger

Jaeger is an open-source, end-to-end distributed tracing system used for monitoring and troubleshooting microservices-based architectures. It helps developers understand how requests flow through a complex system, by tracing the path a request takes and measuring how long each step in that path takes.

❓ Why Use Jaeger?

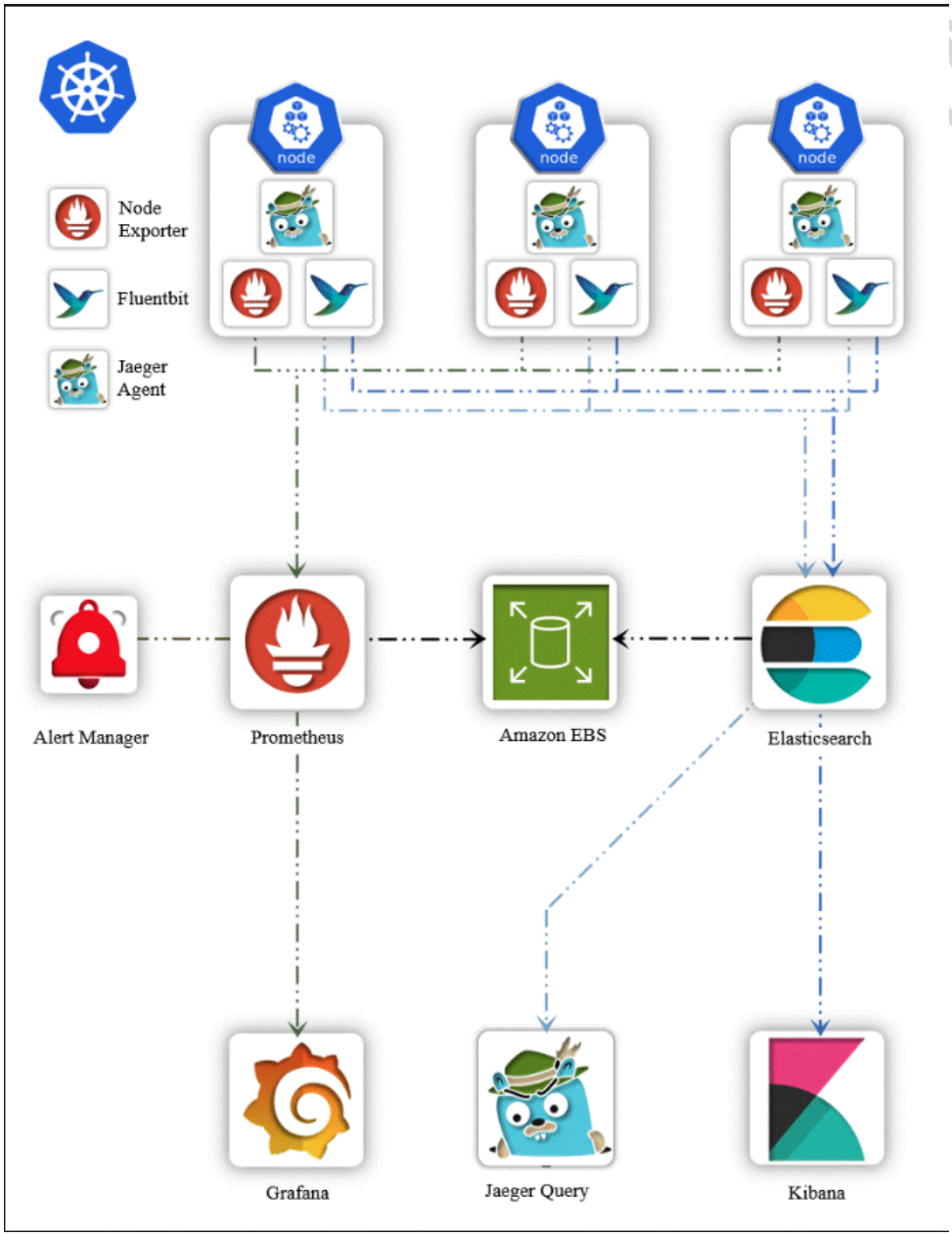
- In modern applications, especially microservices architectures, a single user request can touch multiple services. When something goes wrong, it's challenging to pinpoint the source of the problem. Jaeger helps by:
- 🐢 **Identifying bottlenecks**: See where your application spends most of its time.
- 🔍 **Finding root causes of errors**: Trace errors back to their source.
- ⚡ **Optimizing performance**: Understand and improve the latency of services.

📚 Core Concepts of Jaeger

- 🛤️ **Trace**: A trace represents the journey of a request as it travels through various services. Think of it as a detailed map that shows every stop a request makes in your system.
- 📏 **Span**: Each trace is made up of multiple spans. A span is a single operation within a trace, such as an API call or a database query. It has a start time and a duration.
- 🏷️ **Tags**: Tags are key-value pairs that provide additional context about a span. For example, a tag might indicate the HTTP method used (GET, POST) or the status code returned.
- 📝 **Logs**: Logs in a span provide details about what's happening during that operation. They can capture events like errors or important checkpoints.

- 🔗 **Context Propagation**: For Jaeger to trace requests across services, it needs to propagate context. This means each service in the call chain passes along the trace information to the next service.

🏠 Architecture



⚙️ Setting Up Jaeger

- Clone the GitHub Repo - https://github.com/iam-veeramalla/observability-zero-to-hero
- cd observability-zero-to-hero

Step 1: Instrumenting Your Code

- To start tracing, you need to instrument your services. This means adding tracing capabilities to your code. Most popular programming languages and frameworks have libraries or middleware that make this easy.
- We have already instrumented our code using OpenTelemetry libraries/packages. For more details, refer to `day-4/application/service-a/tracing.js` or `day-4/application/service-b/tracing.js`.

Step 2: Components of Jaeger

- Jaeger consists of several components:
- Agent: Collects traces from your application.
- Collector: Receives traces from the agent and processes them.
- Query: Provides a UI to view traces.
- Storage: Stores traces for later retrieval (often a database like *Elasticsearch*).

Step 3: Export Elasticsearch CA Certificate

- This command retrieves the CA certificate from the Elasticsearch master certificate secret and decodes it, saving it to a ca-cert.pem file.

```
kubectl get secret elasticsearch-master-certs -n
logging -o jsonpath='{.data.ca\.crt}' | base64
--decode > ca-cert.pem
```

Step 4: Create Tracing Namespace

- Creates a new Kubernetes namespace called tracing if it doesn't already exist, where Jaeger components will be installed.

```
kubectl create ns tracing
```

Step 5: Create ConfigMap for Jaeger's TLS Certificate

- Creates a ConfigMap in the tracing namespace, containing the CA certificate to be used by Jaeger for TLS.

```
kubectl create configmap jaeger-tls
--from-file=ca-cert.pem -n tracing
```

Step 6: Create Secret for Elasticsearch TLS

- Creates a Kubernetes Secret in the tracing namespace, containing the CA certificate for Elasticsearch TLS communication.

```
kubectl create secret generic es-tls-secret
--from-file=ca-cert.pem -n tracing
```

Step 7: Add Jaeger Helm Repository

- adds the official Jaeger Helm chart repository to your Helm setup, making it available for installations.

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

helm repo update
```

Step 8: Install Jaeger with Custom Values

- 👉 **Note**: Please update the `password` field and other related field in the `jaeger-values.yaml` file with the password retrieved earlier in day-4 at step 6: (i.e NJyO47UqeYBsoaEU)"
- Command installs Jaeger into the tracing namespace using a custom jaeger-values.yaml configuration file. Ensure the password is updated in the file before installation.

```
helm install jaeger jaegertracing/jaeger -n tracing
--values jaeger-values.yaml
```

Step 9: Port Forward Jaeger Query Service

- Command forwards port 8080 on your local machine to the Jaeger Query service, allowing you to access the Jaeger UI locally.

```
kubectl port-forward svc/jaeger-query 8080:80 -n
tracing
```

🧼 Clean Up

```
helm uninstall jaeger -n tracing
helm uninstall elasticsearch -n logging

# Also delete PVC created for elasticsearch
helm uninstall monitoring -n monitoring

cd day-4
kubectl delete -k kubernetes-manifest/

kubectl delete -k
alerts-alertmanager-servicemonitor-manifest/

# Delete cluster
eksctl delete cluster --name observability
```

📊 Chapter 7: OpenTelemetry

- OpenTelemetry is an open-source observability framework for generating, collecting, and exporting telemetry data (traces, metrics, logs) to help monitor applications.

🛠️ How is it Different from Other Libraries?

- OpenTelemetry offers a unified standard for observability across multiple tools and vendors, unlike other libraries that may focus only on a specific aspect like tracing or metrics.

⏳ What Existed Before OpenTelemetry?

- Before OpenTelemetry, observability was typically managed using a combination of specialized tools for different aspects like
  - `Tracing`: Tools like Jaeger and Zipkin were used to track requests
  - `Metrics`: Solutions like Prometheus and StatsD were popular for collecting metrics
  - `Logging`: Tools like ELK Stack (Elasticsearch, Logstash, Kibana) or Fluentd were used to aggregate and analyze logs.
- OpenTelemetry unified these by standardizing how telemetry data is collected and exported.
- Prior to OpenTelemetry, there were OpenTracing and OpenCensus, which OpenTelemetry merged to provide a more comprehensive and standardized observability solution.

🌐 Supported Programming Languages

OpenTelemetry supports several languages, including:

- **Go**
- **Java**
- **JavaScript**
- **Python**
- **C#**
- **C++**
- **Ruby**
- **PHP**
- **Swift**
- ...and others.

END TO END PROJECT USING OPEN TELEMETRY

- NEXT PAGE

END TO END PROJECT USING OPEN TELEMETRY

- Clone the GitHub Repo - https://github.com/iam-veeramalla/observability-zero-to-hero
- cd observability-zero-to-hero

🖥️ Step 1: Create EKS Cluster

```
eksctl create cluster --name=observability \
                      --region=us-east-1 \
                      --zones=us-east-1a,us-east-1b \
                      --without-nodegroup

eksctl utils associate-iam-oidc-provider \
    --region us-east-1 \
    --cluster observability \
    --approve

eksctl create nodegroup --cluster=observability \
                        --region=us-east-1 \

--name=observability-ng-private \
                        --node-type=t3.medium \
                        --nodes-min=2 \
                        --nodes-max=3 \
                        --node-volume-size=20 \
                        --managed \
                        --asg-access \
                        --external-dns-access \
                        --full-ecr-access \
                        --appmesh-access \
                        --alb-ingress-access \
                        --node-private-networking
```

```
# Update ./kube/config file
aws eks update-kubeconfig --name observability
```

🔐 Step 2: Create IAM Role for Service Account

```
eksctl create iamserviceaccount \
    --name ebs-csi-controller-sa \
    --namespace kube-system \
    --cluster observability \
    --role-name AmazonEKS_EBS_CSI_DriverRole \
    --role-only \
    --attach-policy-arn
arn:aws:iam::aws:policy/service-role/AmazonEBSCSIDrive
rPolicy \
    --approve
```

- This command creates an IAM role for the EBS CSI controller.
- The IAM role allows EBS CSI controllers to interact with AWS resources, specifically for managing EBS volumes in the Kubernetes cluster.
- We will attach the Role with service account

📝 Step 3: Retrieve IAM Role ARN

```
ARN=$(aws iam get-role --role-name
AmazonEKS_EBS_CSI_DriverRole --query 'Role.Arn'
--output text)
```

- Command retrieves the ARN of the IAM role created for the EBS CSI controller service account.

📦 Step 4: Deploy EBS CSI Driver

```
eksctl create addon --cluster observability --name
aws-ebs-csi-driver --version latest \
```

```
--service-account-role-arn $ARN --force
```

- Above command deploys the AWS EBS CSI driver as an addon to your Kubernetes cluster.
- It uses the previously created IAM service account role to allow the driver to manage EBS volumes securely.

🧩 Step 5: Understand the Application

- We have two very simple microservice A (`microservice-a`) & B (`microservice-a`), Built with Golang using the Gin web framework for handling HTTP requests.
- **Microservice A** API Endpoints:
  - `GET /hello-a` – Returns a greeting message
  - `GET /call-b` – Calls another service (Service B) and returns its response
  - `GET /getme-coffee` – Fetches and returns data from an external coffee API
- **Microservice B** API Endpoints:
  - `GET /hello-b` – Returns a greeting message
  - `GET /call-a` – Calls another service (Service A) and returns its response
  - `GET /getme-coffee` – Fetches and returns data from an external coffee API
- Observability:
  - OpenTelemetry SDK integrated for tracing and metrics.
  - Metrics and traces are exported to the OpenTelemetry Collector via OTLP over HTTP.
- Instrumentation:
  - Uses OpenTelemetry middleware (otelgin) for automatic request tracing.
  - Instruments HTTP clients with otelhttp for distributed tracing of outbound requests.

🐳 Step 6: Dockerize & push it to the registry

```
# Dockerize microservice - a
docker build -t <<NAME_OF_YOUR_REPO>>:<<TAG>>
microservice-a/

# Dockerize microservice - b
docker build -t <<NAME_OF_YOUR_REPO>>:<<TAG>>
microservice-b/

# push both images
docker push  <<NAME_OF_YOUR_REPO>>:<<TAG>>
docker push  <<NAME_OF_YOUR_REPO>>:<<TAG>>
```

🗂️ Step 7: Create Namespace for observability components

```
kubectl create namespace olly
```

📚 Step 8: Install Elasticsearch on K8s

helm repo add elastic https://helm.elastic.co

helm install elasticsearch
--set replicas=1
--set volumeClaimTemplate.storageClassName=gp2
--set persistence.labels.enabled=true elastic/elasticsearch -n olly

📜 Step 9: Export Elasticsearch CA Certificate

- This command retrieves the CA certificate from the Elasticsearch master certificate secret and decodes it, saving it to a ca-cert.pem file.

```
kubectl get secret elasticsearch-master-certs -n olly
-o jsonpath='{.data.ca\.crt}' | base64 --decode >
ca-cert.pem
```

🔑 Step 10: Create ConfigMap for Jaeger's TLS Certificate

- Creates a ConfigMap in the olly namespace, containing the CA certificate to be used by Jaeger for TLS.

```
kubectl create configmap jaeger-tls
--from-file=ca-cert.pem -n olly
```

🛡️ Step 11: Create Secret for Elasticsearch TLS

- Creates a Kubernetes Secret in the tracing namespace, containing the CA certificate for Elasticsearch TLS communication.

```
kubectl create secret generic es-tls-secret
--from-file=ca-cert.pem -n olly
```

🔍 Step 12: Retrieve Elasticsearch Username & Password

```
# for username
kubectl get secrets --namespace=olly
elasticsearch-master-credentials
-ojsonpath='{.data.username}' | base64 -d
# for password
kubectl get secrets --namespace=olly
elasticsearch-master-credentials
-ojsonpath='{.data.password}' | base64 -d
```

- Retrieves the password for the Elasticsearch cluster's master credentials from the Kubernetes secret.
- 👉 **Note**: Please write down the password for future reference

🕵️ Step 13: Install Jaeger with Custom Values

- 👉 **Note**: Please update the `password` field and other related field in the `jaeger-values.yaml` file with the password retrieved previous step at step 12: (i.e NJyO47UqeYBsoaEU)"
- Command installs Jaeger into the olly namespace using a custom jaeger-values.yaml configuration file. Ensure the password is updated in the file before installation.

```
helm repo add jaegertracing
https://jaegertracing.github.io/helm-charts
helm repo update

helm install jaeger jaegertracing/jaeger -n olly
--values jaeger-values.yaml
```

🌐 Step 14: Access UI - Port Forward Jaeger Query Service

kubectl port-forward svc/jaeger-query 8080:80 -n olly

📈 Step 15: Install Opentelemetry-collector

helm install otel-collector open-telemetry/opentelemetry-collector -n olly --values otel-collector-values.yaml

📊 Step 16: Install prometheus

```
helm repo add prometheus-community
https://prometheus-community.github.io/helm-charts
helm repo update

helm install  prometheus
prometheus-community/prometheus -n olly --values
prometheus-values.yaml
```

🚀 Step 17: Deploy the application

- ***Note:*** - Review the Kubernetes manifest files located in `./k8s-manifest`. and you should change image name & tag with your own image

```
kubectl apply -k k8s-manifests/
```

- 👉 ***Note***: wait for 5 minutes till you load balancer comes in running state

🔄 Step 18: Generate Load

- Script: `test.sh` takes two load balancer DNS addresses as input arguments and alternates requests between them using curl.
- `test.sh` Continuously sends random HTTP requests every second to predefined routes on two provided load balancer DNSs
- ***Note:*** Keep the script running in another terminal to quickly gather metrics & traces.

```
./test.sh http://Microservice_A_LOAD_BALANCER_DNS
http://Microservice_B_LOAD_BALANCER_DNS
```

📊 Step 19: Access the UI of Prometheus

```
kubectl port-forward svc/prometheus-server 9090:80 -n
olly
```

- Look for your application's metrics like `request_count`, `request_duration_ms`, `active_requests` and other to monitor request rates & performance.

🕵️ Step 20: Access the UI of Jaeger

```
kubectl port-forward svc/jaeger-query 8080:80 -n olly
```

- Look for traces from the service name microservice-a, microservice-b and operations such as `[/hello-a, /call-b, and /getme-coffee]` or `[/hello-b, /call-a, and /getme-coffee]` to monitor request flows and dependencies.

✅ Conclusion

- By following the above steps, you have successfully set up an observability stack using OpenTelemetry on an EKS cluster. This setup allows you to monitor your microservices effectively through integrated tracing, metrics, and logging.

🧼 Clean Up

```
helm uninstall prometheus -n olly
helm uninstall otel-collector -n olly
helm uninstall jaeger -n olly
helm uninstall elasticsearch -n olly

<!-- Delete all the pvc & pv -->
kubectl delete -k k8s-manifests/

kubectl delete ns olly
eksctl delete cluster --name observability
```