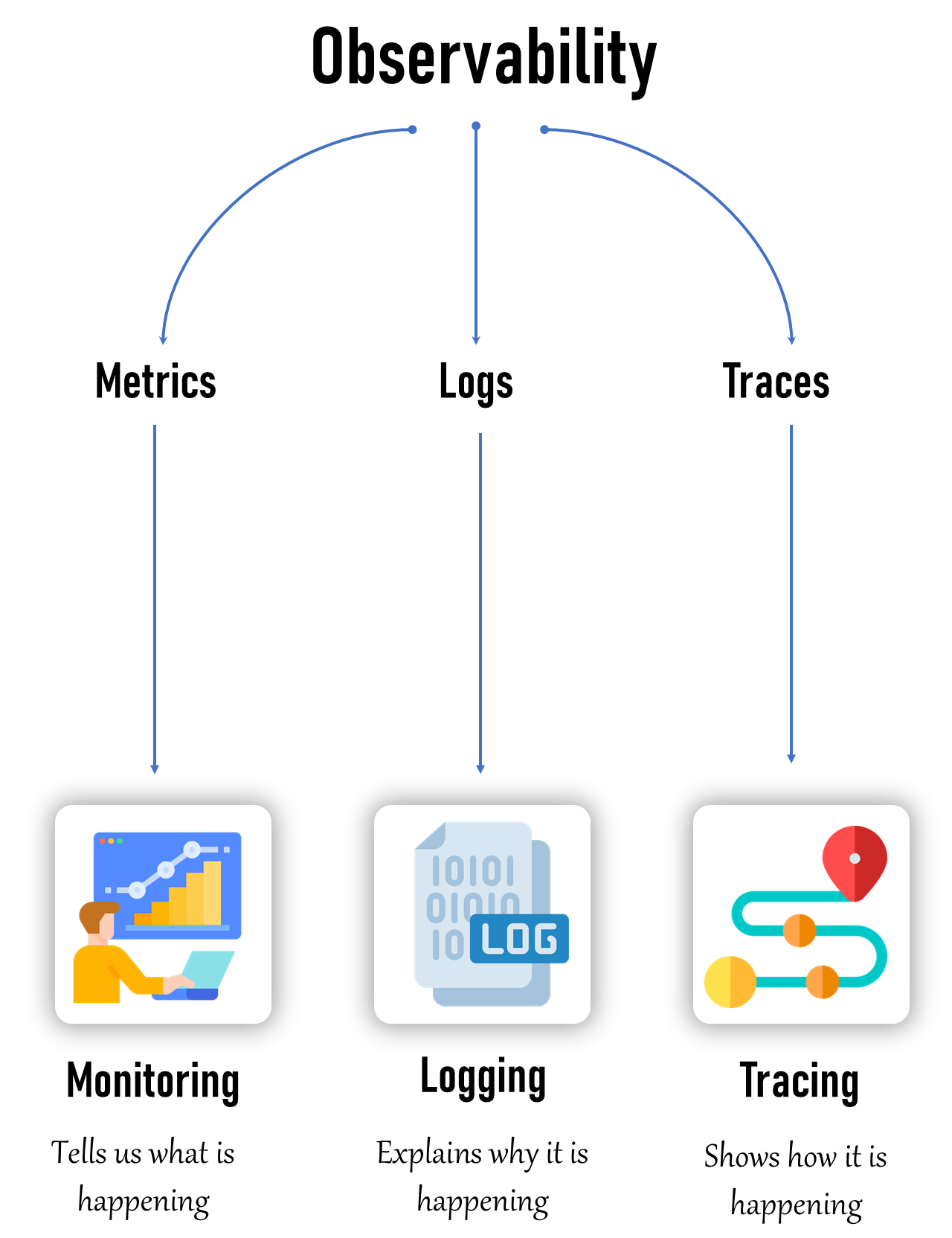
# Introduction to Observability

* 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.

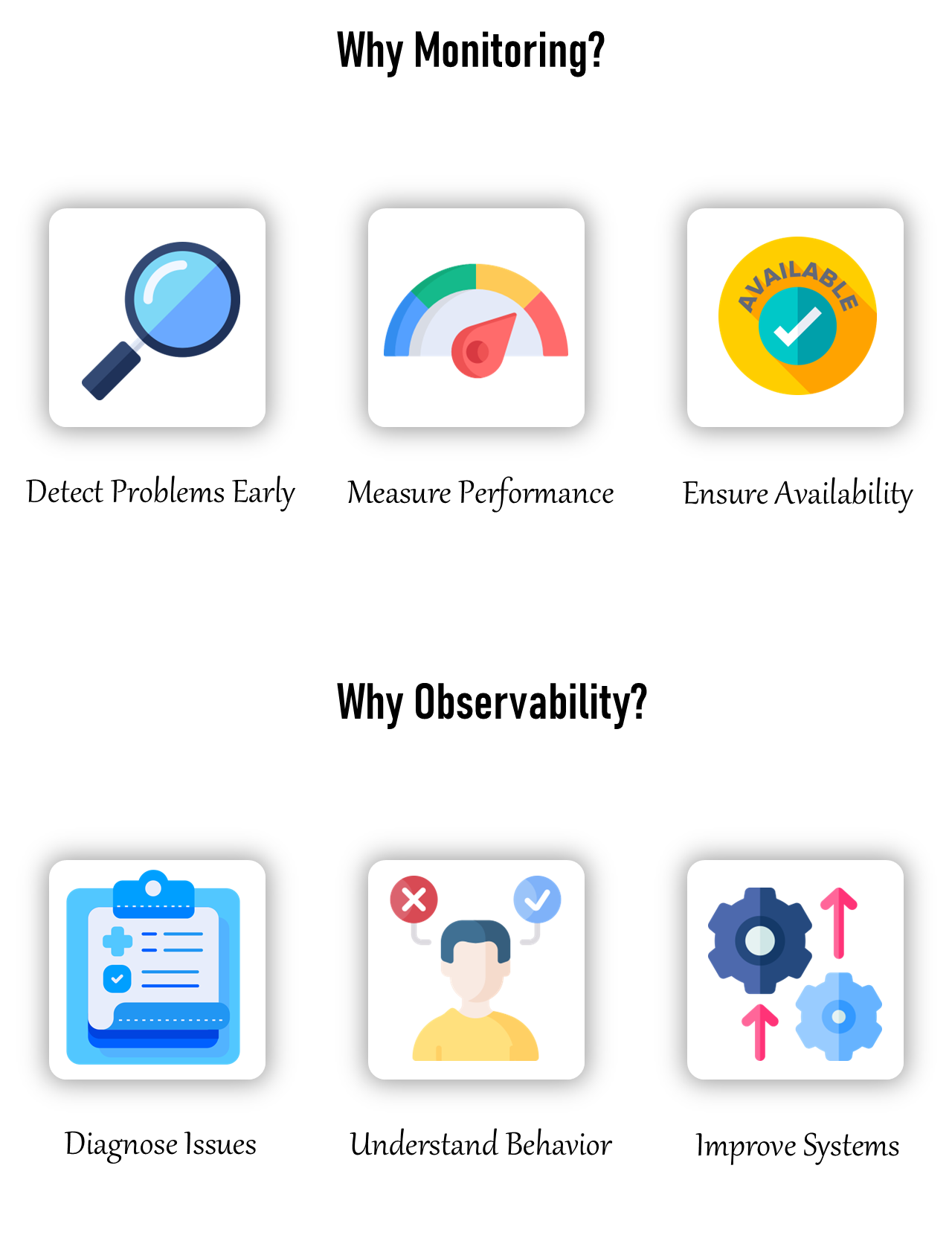
[](https://github.com/iam-veeramalla/observability-zero-to-hero/blob/main/day-1/images/Introduction-to-Observability.png)

**🤔 Why Monitoring?**

* Monitoring helps us keep an eye on our systems to ensure they are working properly.
* Perpose: 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:

[](https://github.com/iam-veeramalla/observability-zero-to-hero/blob/main/day-1/images/why-monitoring-why-observability.png)

**🆚 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.

## Day2

**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.

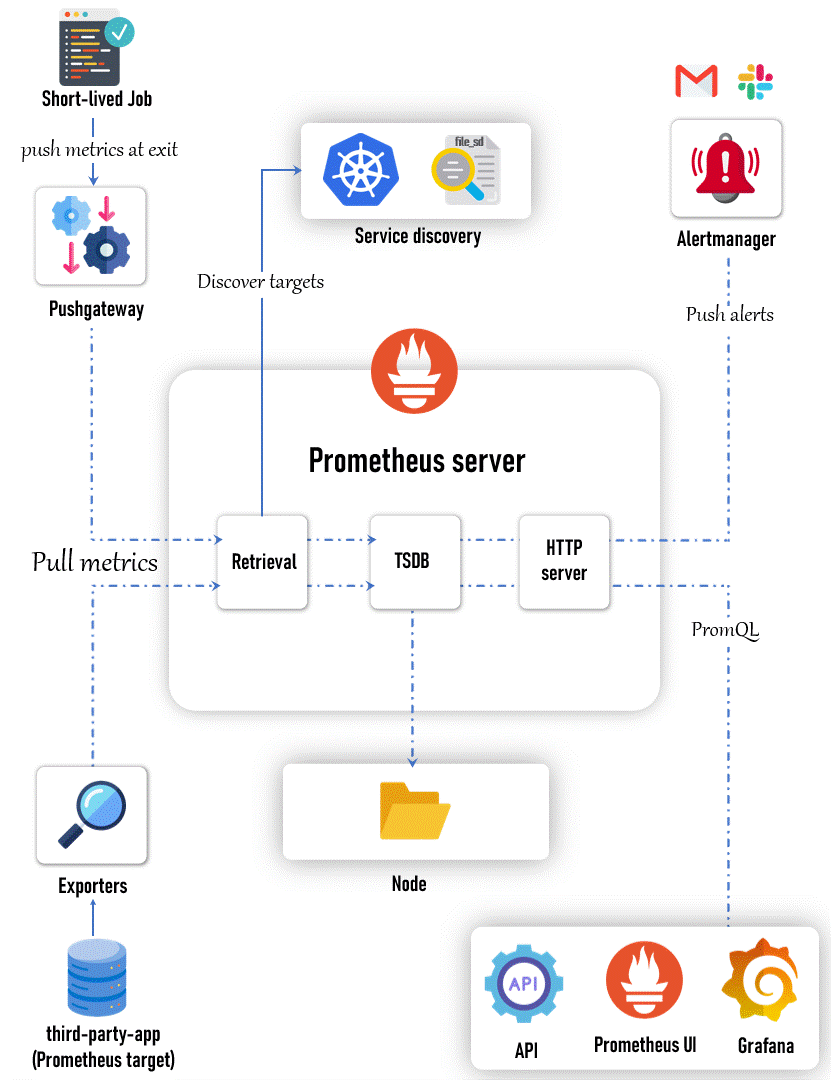
Monitoring is the process of keeping an eye on these metrics over time to understand what’s normal, identify changes, and detect problems. It's like watching your step count daily to see if you're meeting your fitness goal or checking your heart rate to make sure it's in a healthy range.

**🚀 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.

[](https://github.com/iam-veeramalla/observability-zero-to-hero/blob/main/day-2/images/prometheus-architecture.gif)

**🔥 Prometheus Server**

* 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 - Please Refer this link.
* Setup and configure AWS CLI using the aws configure command.
* Install and configure eksctl using the steps mentioned here.
* Install and configure kubectl as mentioned here.

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

**NOTE:** If you are using an EC2 Instance or Cloud VM, you need to pass --address 0.0.0.0 to the above command. Then you can access the UI on instance-ip:port

* **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

alertmanager:

alertmanagerSpec:

# Selects Alertmanager configuration based on these labels. Ensure that the Alertmanager configuration has matching labels.

# ✅ Solves error: Misconfigured Alertmanager selectors can lead to missing alert configurations.

# ✅ Solves error: Alertmanager wasn't able to findout the applied CRD (kind: Alertmanagerconfig)

alertmanagerConfigSelector:

matchLabels:

release: monitoring

# Sets the number of Alertmanager replicas to 3 for high availability.

# ✅ Solves error: Single replica can cause alerting issues during pod failures.

# ✅ Solves error: Alertmanager Cluster Status is Disabled (GitHub issue)

replicas: 2

# Sets the strategy for matching Alertmanager configurations. 'None' means no specific matching strategy.

# ✅ Solves error: Incorrect matcher strategy can lead to unhandled alert configurations.

# ✅ Solves error: Get rid of namespace matchers when creating AlertManagerConfig (GitHub issue)

alertmanagerConfigMatcherStrategy:

type: None

## day3

**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.

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

name: kubernetes-prometheus-stack

annotations:

alb.ingress.kubernetes.io/scheme: internet-facing

alb.ingress.kubernetes.io/listen-ports: '[{"HTTP": 80}]'

alb.ingress.kubernetes.io/target-type: ip

spec:

ingressClassName: alb

rules:

- http:

paths:

- path: /prometheus

pathType: Prefix

backend:

service:

name: prometheus-service # Change this to your Prometheus service name

port:

number: 9090

- path: /grafana

pathType: Prefix

backend:

service:

name: grafana-service # Change this to your Grafana service name

port:

number: 3000

- path: /alertmanager

pathType: Prefix

backend:

service:

name: alertmanager-service # Change this to your Alertmanager service name

port:

number: 9093

## Day4

**Instrumentation**

* Instrumentation refers to the process of adding monitoring capabilities to your applications, systems, or services.
* This involves embedding/Writting 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.

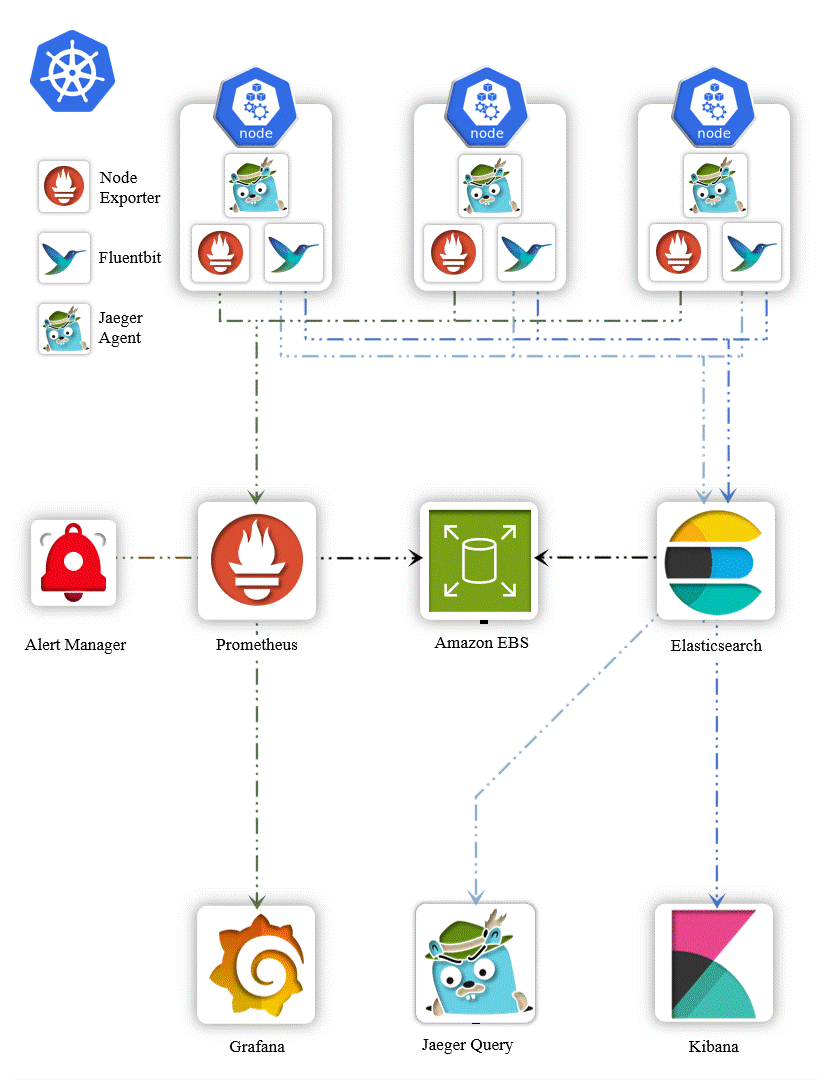
**📈 Types of Metrics in Prometheus**

* 🔄️ **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**

[](https://github.com/iam-veeramalla/observability-zero-to-hero/blob/main/day-4/images/architecture.gif)

**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/

# or use abhishekf5/demoservice-a:v

# Dockerize microservice - b

docker build -t <<NAME\_OF\_YOUR\_REPO>>:<<TAG>> application/service-b/

or use the pre-built images

- abhishekf5/demoservice-a:v

- abhishekf5/demoservice-b:v

**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.