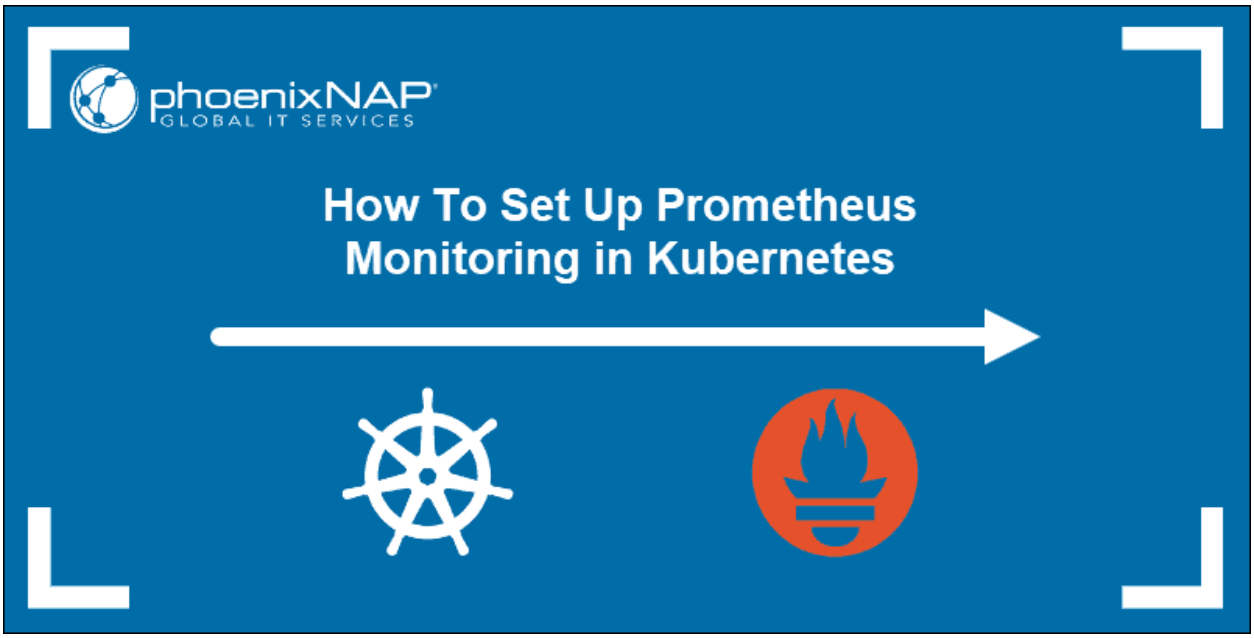
Introduction

Prometheus is an open-source instrumentation framework. Prometheus can absorb massive amounts of data every second, making it well suited for complex workloads.

Use Prometheus to monitor your servers, VMs, databases, and draw on that data to analyze the performance of your applications and infrastructure.

This article explains how to set up Prometheus monitoring in a Kubernetes cluster.



Prerequisites

- A Kubernetes cluster
- A fully configured `kubectl` command-line interface on your local machine

Monitoring Kubernetes Cluster with Prometheus

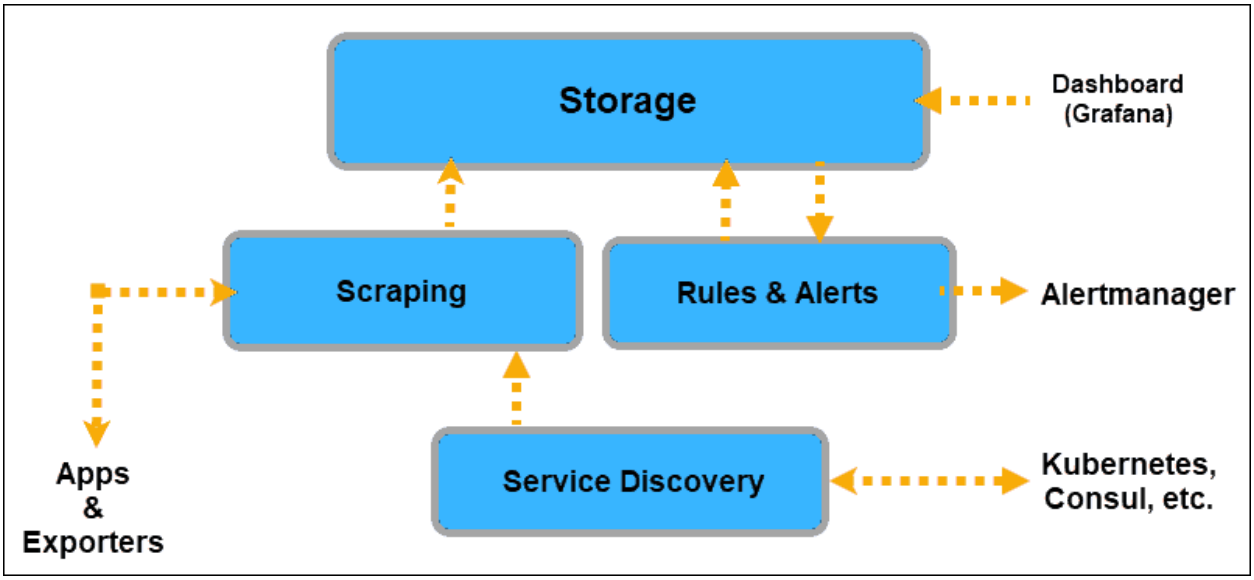
Prometheus is a pull-based system. It sends an HTTP request, a so-called `scrape`, based on the configuration defined in the ***deployment file***. The response to this `scrape` request is stored and parsed in storage along with the metrics for the scrape itself.

The storage is a custom database on the Prometheus server and can handle a massive influx of data. It's possible to monitor thousands of machines simultaneously with a single server.

Note: With so much data coming in, disk space can quickly become an issue. The collected data has great short-term value. If you are planning on keeping extensive long-term records, it might be a good idea to provision additional persistent storage volumes.

The data needs to be appropriately exposed and formatted so that Prometheus can collect it. Prometheus can access data directly from the app's client libraries or by using exporters.

Exporters are used for data that you do not have full control over (for example, kernel metrics). An exporter is a piece of software placed next to your application. Its purpose is to accept HTTP requests from Prometheus, make sure the data is in a supported format, and then provide the requested data to the Prometheus server.



All your applications are now equipped to provide data to Prometheus. We still need to inform Prometheus where to look for that data. Prometheus discovers targets to scrape from by using ***Service Discovery***.

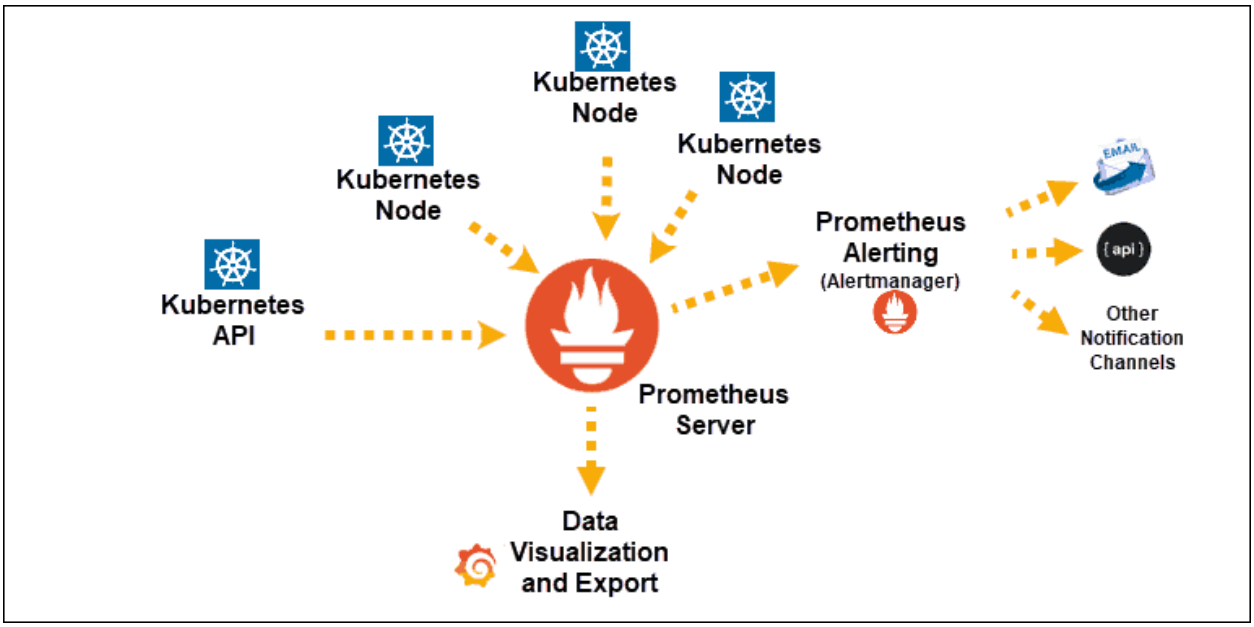
Your Kubernetes cluster already has labels and annotations and an excellent mechanism for keeping track of changes and the status of its elements. Hence, Prometheus uses the Kubernetes API to discover targets.

The Kubernetes service discoveries that you can expose to Prometheus are:

- ***node***
- ***endpoint***
- ***service***
- ***pod***
- ***ingress***

Prometheus retrieves machine-level metrics separately from the application information. The only way to expose memory, disk space, CPU usage, and bandwidth metrics is to use a ***node exporter***. Additionally, metrics about cgroups need to be exposed as well.

Fortunately, the cAdvisor exporter is already embedded on the Kubernetes node level and can be readily exposed.



Once the system collects the data, you can access it by using the PromQL query language, export it to graphical interfaces like Grafana, or use it to send alerts with the Alertmanager.

Install Prometheus Monitoring on Kubernetes

Prometheus monitoring can be installed on a Kubernetes cluster by using a set of YAML (Yet Another Markup Language) files. These files contain configurations, permissions, and services that allow Prometheus to access resources and pull information by scraping the elements of your cluster.

YAML files are easily tracked, edited, and can be reused indefinitely. The files presented in this tutorial are readily and freely available in online repositories such as GitHub.

Note: The **.yml** files below, in their current form, are not meant to be used in a production environment. Instead, you should adequately edit these files to fit your system requirements.

Create Monitoring Namespace

All the resources in Kubernetes are started in a namespace. Unless one is specified, the system uses the default namespace. To have better control over the cluster monitoring process, we are going to specify a monitoring namespace.

The name of the namespace needs to be a label compatible with DNS. For easy reference, we are going to name the namespace: ***monitoring***.

There are two ways to create a monitoring namespace for retrieving metrics from the Kubernetes API.

Option 1:

Enter this simple command in your command-line interface and create the ***monitoring*** namespace on your host:

```
kubectl create namespace monitoring
```

Option 2:

Create and apply a .yml file:

```
apiVersion: v1
kind: Namespace
metadata:
  name: monitoring
```

This method is convenient as you can deploy the same file in future instances. Apply the file to your cluster by entering the following command in your command terminal:

```
kubectl -f apply namespace monitoring.yml
```

Regardless of the method used, list existing namespaces by using this command:

```
kubectl get namespaces
```

Configure Prometheus Deployment File

The following section contains the necessary elements to successfully set up Prometheus scraping on your Kubernetes cluster and its elements.

The sections can be implemented as individual **.yml** files executed in sequence. After you create each file, it can be applied by entering the following command:

```
kubectl -f apply [name_of_file].yml
```

In this example, all the elements are placed into a single **.yml** file and applied simultaneously.

The *prometheus.yml* file in our example instructs the **kubectl** to submit a request to the Kubernetes API server. The file contains:

1. Permissions that allow Prometheus to access all pods and nodes.
2. The Prometheus **configMap** that defines which elements should be scrapped.
3. Prometheus deployment instructions.
4. A service that gives you access to the Prometheus user interface.

Cluster Role, Service Account and Cluster Role Binding

Namespaces are designed to limit permissions of default roles if we want to retrieve cluster-wide data we need to give Prometheus access to all resources of that cluster. A basic Prometheus .yml file that provides cluster-wide access has the following elements:

1. Define Cluster Role

The verbs on each rule define the actions the role can take on the apiGroups.

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRole
metadata:
  name: prometheus
rules:
- apiGroups: [""]
  resources:
```

```
  - nodes
  - services
  - endpoints
  - pods
  verbs: ["get", "list", "watch"]
- apiGroups:
  - extensions
  resources:
  - ingresses
  verbs: ["get", "list", "watch"]
```

2. Create Service Account

Additionally, we need to create a service account to apply this role to:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: prometheus
  namespace: monitoring
```

3. Apply ClusterRoleBinding

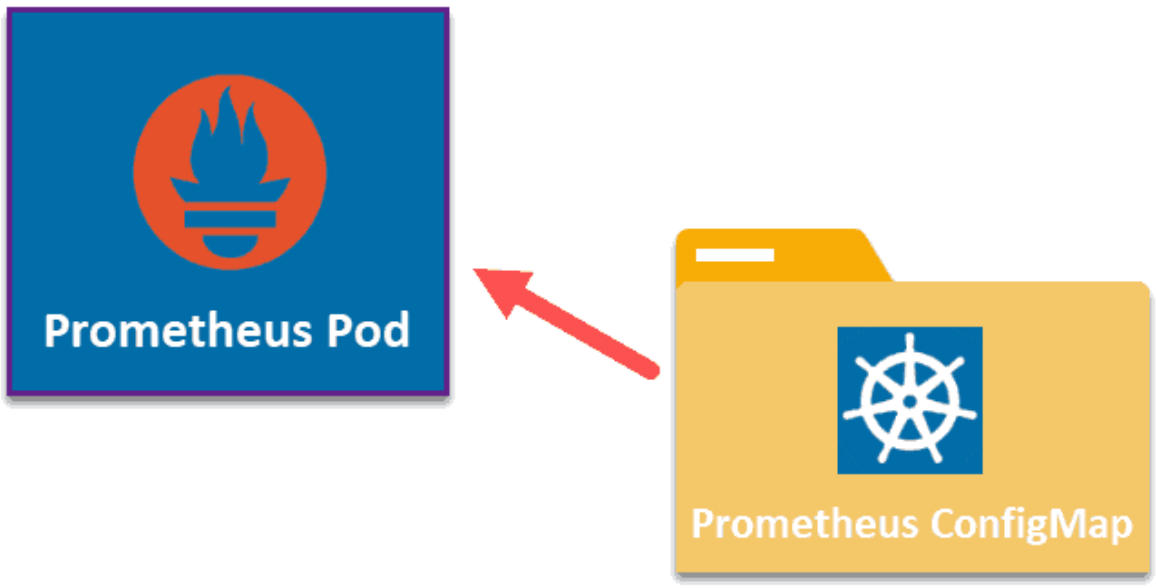
Finally, we need to apply a ***ClusterRoleBinding***. This action is going to bind the Service Account to the Cluster Role created previously.

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: prometheus
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: prometheus
subjects:
- kind: ServiceAccount
  name: prometheus
  namespace: monitoring
```

By adding these resources to our file, we have granted Prometheus cluster-wide access from the ***monitoring*** namespace.

Prometheus ConfigMap

This section of the file provides instructions for the scraping process. Specific instructions for each element of the Kubernetes cluster should be customized to match your monitoring requirements and cluster setup.



1. Global Scrape Rules

```
apiVersion: v1
data:
  prometheus.yml: |
    global:
      scrape_interval: 10s
```

2. Scrape Node

This service discovery exposes the nodes that make up your Kubernetes cluster. The kubelet runs on every single node and is a source of valuable information.

2.1 Scrape kubelet

```
    scrape_configs:
    - job_name: 'kubelet'
      kubernetes_sd_configs:
      - role: node
      scheme: https
```

```
      tls_config:
        ca_file: /var/run/secrets/kubernetes.io/serviceaccount/c
a.crt
        insecure_skip_verify: true  # Required with Minikube.
```

2.2 Scrape cAdvisor (container level information)

The **kubelet** only provides information about itself and not the containers. To receive information from the container level, we need to use an exporter. The **cAdvisor** is already embedded and only needs a **metrics_path**: **/metrics/cadvisor** for Prometheus to collect container data:

```
    - job_name: 'cadvisor'
      kubernetes_sd_configs:
      - role: node
      scheme: https
      tls_config:
        ca_file: /var/run/secrets/kubernetes.io/serviceaccount/c
a.crt
        insecure_skip_verify: true  # Required with Minikube.
      metrics_path: /metrics/cadvisor
```

3. Scrape APIServer

Use the endpoints role to target each application instance. This section of the file allows you to scrape API servers in your Kubernetes cluster.

```
    - job_name: 'k8apiserver'
      kubernetes_sd_configs:
      - role: endpoints
      scheme: https
      tls_config:
        ca_file: /var/run/secrets/kubernetes.io/serviceaccount/c
a.crt
        insecure_skip_verify: true  # Required if using Minikube
.
      bearer_token_file: /var/run/secrets/kubernetes.io/servicea
ccount/token
      relabel_configs:
   - source_labels: [__meta_kubernetes_namespace, __meta_kuberne
tes_service_name, __meta_kubernetes_endpoint_port_name]
        action: keep
```

```
        regex: default;kubernetes;https
```

4. Scrape Pods for Kubernetes Services (excluding API Servers)

Scrape the pods backing all Kubernetes services and disregard the API server metrics.

```
- job_name: 'k8services'
      kubernetes_sd_configs:
      - role: endpoints
      relabel_configs:
      - source_labels:
          - __meta_kubernetes_namespace
          - __meta_kubernetes_service_name
        action: drop
        regex: default;kubernetes
      - source_labels:
          - __meta_kubernetes_namespace
        regex: default
        action: keep
      - source_labels: [__meta_kubernetes_service_name]
        target_label: job
```

5. Pod Role

Discover all pod ports with the name metrics by using the container name as the job label.

```
- job_name: 'k8pods'
      kubernetes_sd_configs:
      - role: pod
      relabel_configs:
      - source_labels: [__meta_kubernetes_pod_container_port_nam
e]
        regex: metrics
        action: keep
      - source_labels: [__meta_kubernetes_pod_container_name]
        target_label: job
kind: ConfigMap
metadata:
  name: prometheus-config
```

6. Configure ReplicaSet

Define the number of replicas you need, and a template that is to be applied to the defined set of pods.

```
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: prometheus
spec:
  selector:
    matchLabels:
      app: prometheus
  replicas: 1
  template:
    metadata:
      labels:
        app: prometheus
    spec:
      serviceAccountName: prometheus
      containers:
      - name: prometheus
        image: prom/prometheus:v2.1.0
        ports:
        - containerPort: 9090
          name: default
        volumeMounts:
        - name: config-volume
          mountPath: /etc/prometheus
      volumes:
      - name: config-volume
        configMap:
         name: prometheus-config
```

7. Define nodePort

Prometheus is currently running in the cluster. Adding the following section to our *prometheus.yml* file is going to give us access to the data Prometheus has collected.

```
kind: Service
apiVersion: v1
metadata:
  name: prometheus
spec:
  selector:
```

```
    app: prometheus
  type: LoadBalancer
  ports:
  - protocol: TCP
    port: 9090
    targetPort: 9090
    nodePort: 30909
```

Apply prometheus.yml File

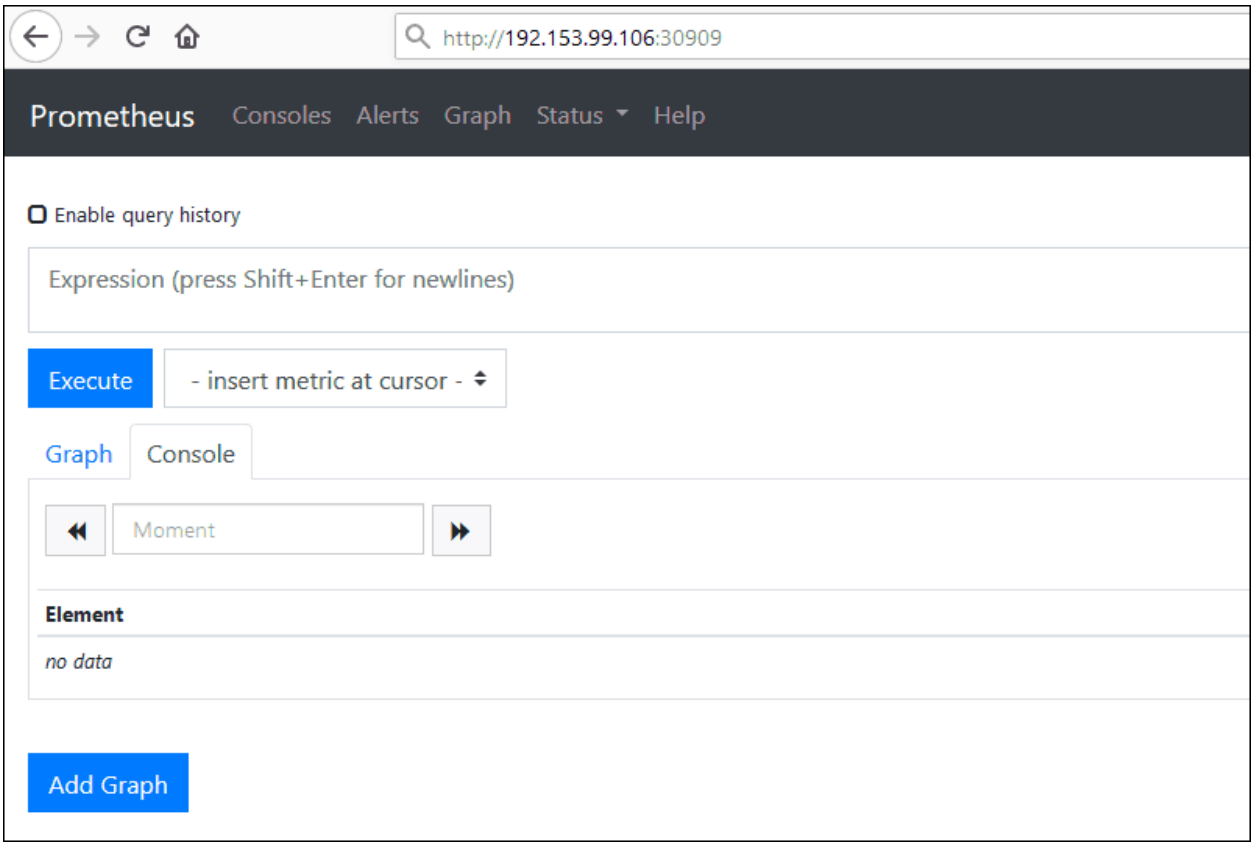
The configuration map defined in the file gives configuration data to every pod on the deployment:

```
kubectl apply -f prometheus.yml
```

Use the individual node URL and the nodePort defined in the *prometheus.yml* file to access Prometheus from your browser. For example:

```
http://192.153.99.106:30909
```

By entering the URL or IP of your node, and by specifying the port from the yml file, you have successfully gained access to Prometheus Monitoring.



Note: If you need a comprehensive dashboard system to graph the metrics gathered by Prometheus, one of the available options is Grafana. It uses data sources to retrieve the information used to create graphs.

How to Monitor kube-state-metrics? (Optional)

You are now able to fully monitor your Kubernetes infrastructure, as well as your application instances. However, this does not include metrics on the information Kubernetes has about the resources in your cluster.

The kube-state-metrics is an exporter that allows Prometheus to scrape that information as well. Create a YAML file for the kube-state-metrics exporter:

```
---
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: kube-state-metrics
```

```
spec:
  selector:
    matchLabels:
      app: kube-state-metrics
  replicas: 1
  template:
    metadata:
      labels:
        app: kube-state-metrics
    spec:
      serviceAccountName: prometheus
      containers:
      - name: kube-state-metrics
        image: quay.io/coreos/kube-state-metrics:v1.2.0
        ports:
        - containerPort: 8080
          name: monitoring
---
kind: Service
apiVersion: v1
metadata:
  name: kube-state-metrics
spec:
  selector:
    app: kube-state-metrics
  type: LoadBalancer
  ports:
  - protocol: TCP
    port: 8080
    targetPort: 8080
```

Apply the file by entering the following command:

```
kubectl apply -f kube-state-metrics.yml
```

Once you apply the file, access Prometheus by entering the node IP/URL and defined nodePort as previously defined.

Conclusion

Now that you have successfully installed Prometheus Monitoring on a Kubernetes cluster, you can track the overall health, performance, and behavior of your system. No matter how large and complex your operations are, a metrics-based monitoring system

such as Prometheus is a vital DevOps tool for maintaining a distributed microservices-based architecture.