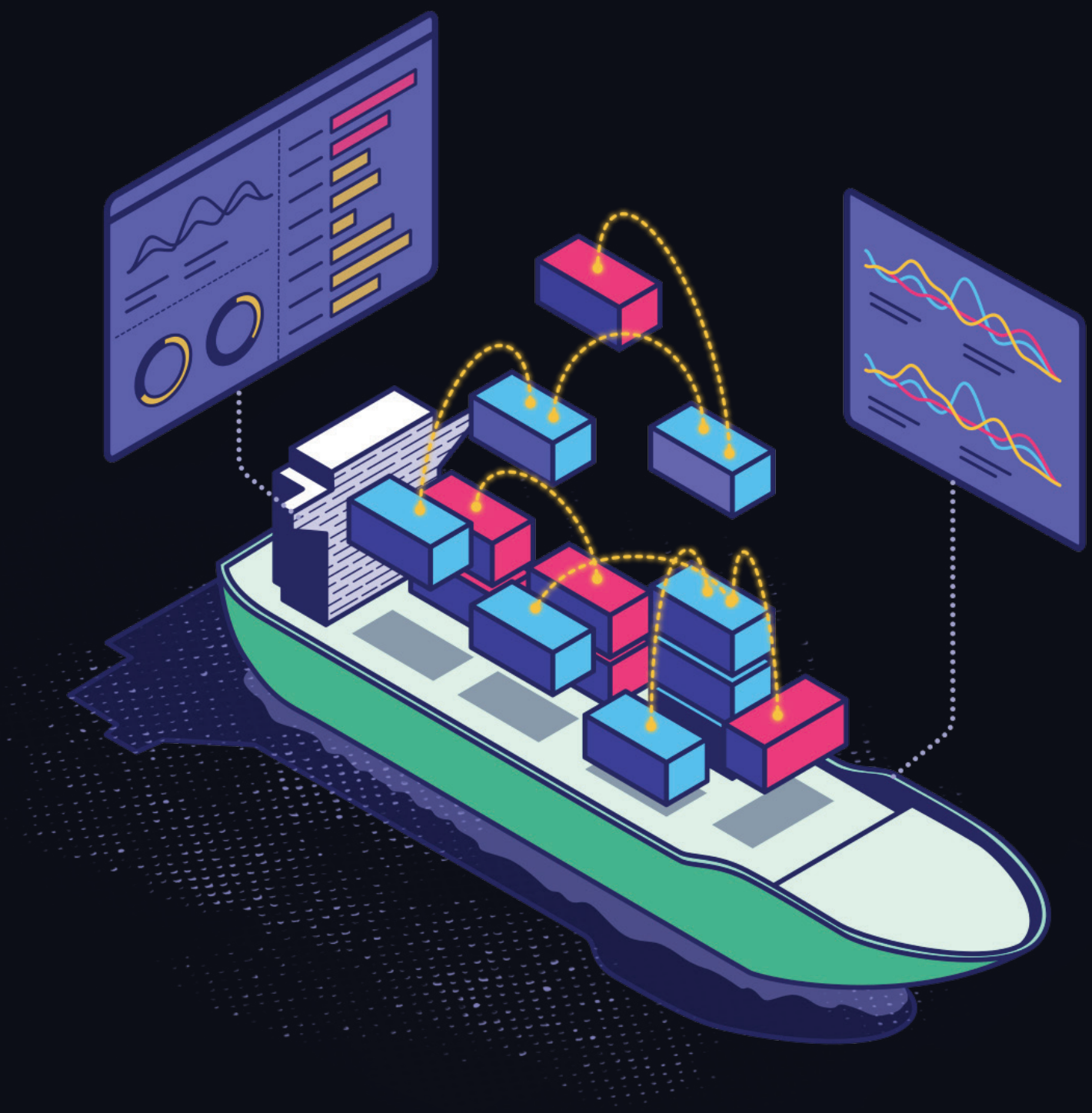
Coralogix

Metrics and Traces in Kubernetes



BY JOANNA WALLACE

INTRODUCTION

The era of the monolith has come to an end, though companies are still working on migrating their black-box code to microservices. Microservices give flexibility in scaling, deployments without service disruptions, and fast feature integrations previously unobtainable.

As the industry moved toward microservices, developers found the need for containers that could hold unique runtimes. Containers save the cost of owning a server for each runtime and ensure all developers and all container deployments have the same setup. Containers are portable, isolated, and consistent, allowing code to run from privately owned hardware. More features were added to container orchestration tools, making them even more appealing.

Container orchestration tools like Kubernetes automate containers' provisioning, deployment, scaling, networking, and lifecycle. While these tasks may be done manually or using scripts, developers would have difficulty managing hundreds or even thousands of containers as busy production environments require. Container orchestration tools were designed to handle this work easily and report back information about the health of clusters.

Kubernetes, a tool maintained by the CNCF, is the industry standard, open-source container orchestration tool. Recording, visualizing, and analyzing this information, coming in the form of metrics, traces, and logs, is critical for maintaining the health of any production environment.

<PART 1>

WHAT WE'LL COVER

This eBook will focus on metrics forming the backbone of alerting and monitoring in Kubernetes. It will also discuss how traces are sorely needed and an often-forgotten tool that is crucial for container observability. Examples will discuss exporting metrics using Prometheus, and examples using traces will use Jaeger.

THE ROLE THAT METRICS AND TRACES PLAY

Metrics

Kubernetes clusters are composed of several components. Generally, a cluster will contain a primary node and one or more running worker nodes. Metrics for Kubernetes indicate how nodes in the cluster are running. Traces give insight into the flow of data and commands throughout a cluster.

Autoscaling is a feature of container orchestration tools like Kubernetes that draws software engineers. Metrics are used in both cases to detect the cluster's health and automatically react to ensure the core functionality of the cluster is maintained. Horizontal autoscaling will adjust the number of replicas in an application, vertical autoscaling adjusts the limits of a container and the resource requests it can handle, and cluster autoscaling adjusts the number of nodes present in a cluster. Each of these autoscalers relies on monitored metrics to know when to take action.

There are built-in metrics that can provide information about when to scale, such as the number of requests per second hitting a particular app. When the number of requests increases beyond a target value, Kubernetes will automatically horizontally scale the app by adding more replicas.

Custom metrics may be necessary for complex applications to ensure your app functions appropriately. For example, you may have a queue internal to a node that can handle a certain amount of traffic. If the traffic hitting a particular node reaches a threshold, autoscaling should kick in again to adjust the number of nodes present to handle the increased traffic. Since this is a custom solution, the default Kubernetes metrics will not handle this case. Developers can use external metrics tools like Prometheus to handle custom cases.

Traces

While metrics help identify problems, traces enable teams to troubleshoot the issue. Distributed systems are difficult to troubleshoot since logs are stored separately for each component. Traces provide data that will link logs

together so logs for the same event can be connected across components.

Traces are unique to microservice architectures like Kubernetes since they were not needed to troubleshoot monolithic software. Trace data enables teams to follow specific requests throughout the software, visualizing what different services it calls and what dependencies it requires.

Teams can use visualizations of trace data to simplify error detection. Since traces track requests throughout the system, traces can be compared to design and expected behavior to indicate where the system is healthy and detect where it is not. Traces show when and where a service or request failed and when the data processing is slow.

WHY YOU NEED OBSERVABLE SOFTWARE

Observable software allows teams to reduce product downtime, improve security, optimize cost, and enhance functionality. Observability becomes especially important with microservices because the system is constantly changing. Services will automatically scale to handle both increases and decreases in traffic, and each of these services requires monitoring.

Metrics provide at-a-glance data indicating if your cluster is healthy, performing effectively, and using resources efficiently. Generally, low-level, static metrics without context do not provide sufficient information for developers to fix issues or enhance architecture. When visualizations show metrics in context with other relevant metrics, logs, and traces, and even with historical behavior, developers can quickly solve problems and may avoid downtime altogether. With a complete set of information including all pillars of observability, troubleshooting and maintaining the system is less cumbersome.

BUILD VS. BUY

We'll explore whether it's worth it to build your solutions from the ground up, or to buy off the shelf products. We'll look at some of the pros and cons for each approach and why you need to tailor your strategy to the specific needs of your business.

<PART 2>

PROMETHEUS FOR KUBERNETES

DEPLOYING YOUR PROMETHEUS INSTANCE

There are several ways to deploy Prometheus to monitor Kubernetes. Here, we will deploy Prometheus in a monitoring namespace of an existing Kubernetes node. Before deploying your Prometheus instance, you should have a Kubernetes cluster running with kubectl. You can find the latest version of Prometheus as a download available through docker.

1. Create a Namespace

Namespaces provide a way for Kubernetes to isolate groups of resources in a single cluster. Prometheus designers intended namespaces for environments with many users or multiple teams, such as development and DevOps teams. By default, deployed objects on Kubernetes will use the default namespace. Configure a new namespace for monitoring using Prometheus using the command below.

```
kubectl create namespace monitoring
```

2. Create a Cluster Role

Prometheus collects logs by making requests against Kubernetes APIs. These APIs hold metrics from Nodes, Deployments, and other Kubernetes constructs and any custom metrics configured in code. Kubernetes recommends that Kublets run with authentication and authorization enabled, so for Prometheus to read from these APIs, it must have the appropriate permissions. Kubelet needs to be configured with role-based access control (RBAC) to grant these permissions. The setup here is then used in the following step defining the configuration for Prometheus.

Use the following YAML file to configure the Kubelet for Prometheus. Name the file rbac.yaml and use it to create the role with the following command:

```
kubectl create -f rbac.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: prometheus
rules:
  - apiGroups: [“”]
    resources:
      - nodes
      - nodes/metrics
      - services
```

```
      - endpoints
      - pods
    verbs: ["get", "list", "watch"]
  - apiGroups:
      - extensions
      - networking.k8s.io
    resources:
      - ingresses
    verbs: ["get", "list", "watch"]
  - nonResourceURLs: ["/metrics", "/metrics/cadvisor"]
    verbs: ["get"]
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: prometheus
  namespace: default
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: prometheus
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: prometheus
subjects:
  - kind: ServiceAccount
    name: prometheus
    namespace: default
```

3. Create External Prometheus Configurations

Prometheus configuration can be done using both the command line and a configuration file. The configuration file defines settings related to scraping

jobs and loaded rule files containing settings for alerts and recordings.

Create the config map in Kubernetes. Setting up Prometheus this way ensures you don't need to build the Prometheus image when a configuration change is needed. Restart the Prometheus pods to apply configuration changes. Create a config_map.yaml file based on the required Prometheus settings, and use the following command to create the config map inside the container. Put both the rules and config into the same file to be loaded into the Prometheus server by the following command:

```
kubectl create -f config_map.yaml
```

The outline of the config_map.yaml file is shown below. This setup will scrape for the services in your Kubernetes cluster. You can customize the configuration according to your needs in the monitoring setup. Prometheus provides a complete document for what can be configured.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: prometheus_configuration
data:
  prometheus.rules: |-
    groups:
    - name: devopscube demo alert
      rules:
      - alert: High Pod Memory
        expr: sum(container_memory_usage_bytes) > 1
        for: 1m
        labels:
          severity: slack
        annotations:
          summary: High Memory Usage
```

```
  prometheus.yml: |-
    global:
      scrape_interval: 10s          # How frequently to
scrape targets (default = 1 minute)
      evaluation_interval: 10s     # How frequently to
evaluate rules (default = 1 minute)

    rule_files:                    # Rules will be read
from all matching files listed below
      - /etc/prometheus/prometheus.rules

    alerting:                      # Settings related to
the alert manager
      alertmanagers:
      - scheme: http
        static_configs:
        - targets:
          - "alertmanager.monitoring.svc:9093"

    remote_write:                  # Use remote_write to
send logs to long-term storage on a third-party tool
                                   # Contact Coralogix
for specific setup to use with Prometheus
      url: <coralogix.org>
      name: test_coralogix
      remoteTimeout: 120s
      bearerToken: <private_key>

    scrape_configs:                # A list of scrape
configurations
      - job_name: 'node-exporter' # Job name must be
unique across all scrape_configs definitions
        scrape_interval: 5s        # Overwrite the
global scrape_interval above if needed
```

```
        kubernetes_sd_configs:
          - role: endpoints
        relabel_configs:           # Dynamically rewrite
the label set of a target
        - source_labels: [__meta_kubernetes_endpoints_
name]
          regex: 'node-exporter'   # Regular expression
against which the extracted value is matched
          action: keep             # Action to perform
based on the regex matched (default = replace)

      - job_name: 'kubernetes-apiservers'
        kubernetes_sd_configs:
        - role: endpoints
        scheme: https
        tls_config:
          ca_file: /var/run/secrets/kubernetes.io/
serviceaccount/ca.crt
        bearer_token_file: /var/run/secrets/kubernetes.
io/serviceaccount/token
        relabel_configs:
        - source_labels: [__meta_kubernetes_namespace,
__meta_kubernetes_service_name, __meta_kubernetes_
endpoint_port_name]
          action: keep
          regex: default;kubernetes;https

      - job_name: 'kubernetes-nodes'
        scheme: https
        tls_config:
          ca_file: /var/run/secrets/kubernetes.io/
serviceaccount/ca.crt
        bearer_token_file: /var/run/secrets/kubernetes.
io/serviceaccount/token
```

```
        kubernetes_sd_configs:
        - role: node
        relabel_configs:
        - target_label: __address__
          replacement: kubernetes.default.svc:443
# Replacement value to use in place of regex-matched
values

      - job_name: 'kubernetes-pods'
        kubernetes_sd_configs:
        - role: pod

      - job_name: 'kube-state-metrics'
        static_configs:
          - targets: ['kube-state-metrics.kube-system.
svc.cluster.local:8080']

      - job_name: 'kubernetes-cadvisor'
        scheme: https
        tls_config:
          ca_file: /var/run/secrets/kubernetes.io/
serviceaccount/ca.crt
        bearer_token_file: /var/run/secrets/kubernetes.
io/serviceaccount/token
        kubernetes_sd_configs:
        - role: node

      - job_name: 'kubernetes-service-endpoints'
        kubernetes_sd_configs:
        - role: endpoints
```

4. Create a Prometheus Deployment

Create a file named prometheus_deployment.yaml that will contain the deployment definition. This will link to the configuration file defined in

earlier steps. The YAML file should contain the following:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: prometheus_deployment
  namespace: monitoring
  labels:
    app: prometheus-server
spec:
  replicas: 1         # The number of desired replicas to
set
  strategy:           # Define how updates are performed
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
  selector:           # Defines how the replicate set
targets pods
    matchLabels:
      app: prometheus-server
  template:           # The pod template applied to each
pod in the set
    metadata:
      labels:
        app: prometheus-server
      annotations:
        prometheus.io/port: “9090”
    spec:             # Specifies how each container will
run
      containers:
        - name: prometheus
          image: prom/prometheus      # The docker
image that will run
          args:
```

```
            - "--storage.tsdb.retention.time=24h"
            - "--storage.tsdb.path=/prometheus/"
            - "--config.file=/etc/prometheus/prometheus.
yml"
          ports:
            - containerPort: 9090
          resources:
            requests:
              cpu: 500m
              memory: 500M
            limits:
              cpu: 1
              memory: 1Gi
          volumeMounts:
            - name: prometheus-config-volume
              mountPath: /etc/prometheus/
            - name: prometheus-storage-volume
              mountPath: /prometheus/
      volumes:    # Defines external volumes or
directories mounted into the containers
        - name: prometheus-config-volume
          configMap:
            name: prometheus-config

        - name: prometheus-storage-volume
          emptyDir: {}
```

5. Create the Deployment

Finally, deploy the Prometheus configuration by running the following command:

```
kubectl create  -f prometheus-deployment.yaml
```

Configuring Prometheus is Complicated

During Step 3 of the tutorial above we showed a simple configuration for Prometheus. Whether you are using Prometheus for the first time or running it in a production environment, writing a configuration file is a daunting task. Prometheus' documentation shows just how many settings can be set to customize how Prometheus runs in your application. The learning curve on configuring Prometheus in an ideal way is very steep and operational knowledge of how to decide what configurations to use comes with time and experience.

Kubernetes introduced the Operator Framework to help lower the barrier to setting up tools like Prometheus in clusters. The operator built for Prometheus has a sample configuration built in that will help developers get started with Prometheus quickly.

Deploying Via the Prometheus Operator

The Operator Framework provides a way to build native applications on Kubernetes in an automated way. This open-source software is a software development kit that provides a way to package, deploy, and manage Kubernetes applications. Using this framework, developers have created operators for specific applications like Prometheus so it can be deployed with already-integrated operational knowledge from its designers.

Deploying Prometheus on Kubernetes can easily get complex as you move into a production environment and need to really hone in your needed configuration. The Prometheus operator aims to give developers a fully automated way to deploy a Prometheus server on Kubernetes without needing to configure files as we did in the previous step. The Prometheus operator provides the Prometheus server, Alertmanager, secrets, and config map setup. It will also automatically discover targets that need to be monitored by using common Kubernetes label queries like what was done in the config_map.yaml file above.

To quickly deploy only the Prometheus Operator inside a cluster, simply

choose a release from the repository and run the following command:

```
kubectl create -f bundle.yaml
```

The Prometheus operator project is still in active development by the CNCF, and not all features have been developed at this point. There are still planned features to be delivered to improve how developers can use the Prometheus Operator. All changes are guaranteed to be reverse compatible.

Developers can also enhance the setting on the Prometheus operator with additional configurations like setting a remote write for long-term storage and analysis of metrics.

HOW TO CONFIGURE APPLICATIONS FOR PROMETHEUS SCRAPING

The scrape configuration is set in the Prometheus config-map.yaml file described in step 3 of *Deploying your Prometheus Instance*. The **scrape_config** section of the file specifies how to scrape a set of specified targets. The configuration can be specified for a single job or multiple jobs in more complex architectures. The scrape targets set can be listed statically or dynamically if labels should change at scrape time.

The **scrape_config** section also supports an attribute called ***kubernetes_sd_config***. This array holds a list of Kubernetes service discovery configurations that let users define the targets to scrape on the Kubernetes cluster. Each target on Kubernetes will be deployed with a REST API that stays in sync with the cluster state and will give Prometheus this information when it is polled. Targets include nodes, services, pods, endpoints, endpoint slices, and ingress. Each target includes several various labels giving information about the status of the target.

This scraping configuration is potent, allowing Prometheus to collect what the developer deems most important to observing Kubernetes cluster health. Developers can scrape all available endpoints or just a couple

depending on the use case, giving a lever between data storage and cost.

MAINTAINING PROMETHEUS

Vertical Scaling With More Memory and CPU

Prometheus is a monitoring system, but when you look behind the curtain at how Prometheus functions, it is a time-series database. As with any database, scaling needs to be a first consideration for designing how it will fit into your system. Prometheus's storage is limited to the scalability of memory and CPU of a single node due to its inherent design.

Vertical scaling can be done on your Prometheus implementation, but this type of scaling is limited to local storage on a single node. Also, because Prometheus runs on a single node, locally stored data is not durable and will be lost if the node crashes. Prometheus is known to require a lot of memory since it is directly tied to the number of time-series metrics logged. In production environments, the required memory can snowball. Prometheus provides options for removing data from local storage to clear space to solve this problem. You could also reduce the duration metrics are stored, or remove metrics not needed past checking for alerts.

Since local storage is not clustered or replicated, it should be treated like a single node database where a remote backup is configured, so data is not lost in the case of a node failure. Prometheus offers an interface that writes to and reads from external long-term data stores, and this solves the durability problem and allows developers to clear locally-stored memory earlier. Third-party storage could also use the Prometheus metrics to analyze data and enhance system observability.

Horizontal Scaling With Federated Instances

When starting with a Prometheus server in Kubernetes, it is a common starting point to keep one Prometheus server per cluster. But, as your service grows to multiple data centers, it becomes difficult to review metrics from each Prometheus server since they do not communicate. One solution when this setup is used is to create a global Prometheus server that uses

federation to collect aggregated time-series data from lower-level servers. Another similar federation setup uses this global server to scrape data from other servers.

While the concept of federation is simple, the implementation is more complicated. Several things need to be considered by your architecture when setting up either type of federation in Prometheus, including

- Which metrics will be sent to the global server since memory is limited in each Prometheus server?
- How will the global server scale vertically?
- How will the data transfers be secured since Prometheus uses HTTP calls?
- How much will the data transfers cost?

Rather than scaling with federated instances, use an external tool to collect the data from each of your Prometheus servers and store it. If already using this tool for long-term storage, this can be exceptionally efficient. The data from all Prometheus servers running is not only collected in one location but can also be visualized and analyzed in the same service.

Configuring Your Prometheus SLAs

Before setting up a Prometheus instance, you should have a sense of the limits inherent to the tool. Understanding the limits of the metrics ensures you are aware of what Prometheus can collect with your configuration and when you need to change your configuration or enhance it with other tools. While this list of SLAs is not complete, it is a good indication of the design that developers should address before creating a production-grade Prometheus server.

Local Storage Limits

Prometheus has limits for how many time series chunks (or samples) it can ingest simultaneously. This limit is based on the node's heap size and can be reset with the ***storage.local.target-heap-size*** setting. By default,

Prometheus can handle about 200,000 time-series chunks present on the heap at one time. Beyond this, sample ingestion is throttled, which will cause scrapes to be skipped and alerts to be missed. The recommended value for ***storage.local.target-heap-size*** is two-thirds of the physical memory limit Prometheus should not exceed.

Strategy For Writing to Disk

Under ideal conditions, data would be written to disk as soon as possible, so it is present in permanent storage. However, if this is done too frequently with write operations handling small amounts of data, the I/O bandwidth would be consistently consumed, and the disk behavior would appear slow. By default, Prometheus will batch write operations with an *adaptive* strategy, which attempts to balance I/O bandwidth usage with the potential amount of data lost in case of a crash. To move away from this strategy, use the ***storage.local.series-sync-strategy*** flag.

Using the adaptive strategy, Prometheus will calculate the urgency of writing chunks to disk. The higher the urgency (on a scale of 0 to 1), the more frequently chunks are written to disk. Beyond a value of 0.8, Prometheus will enter *rushed mode*, where it takes shortcuts to disk writing to ensure the heap is cleared. It will stop syncing files after write operations, reduce checkpoint creation, and turn off the throttling of write operations to persist chunks. When the urgency score hits a value of 1, throttling of ingestion will occur, and data will be lost.

Sample Retention Time

The sample retention time can be set depending on your analytics needs. The more extended data is held for, the more memory will be required to hold the data in the local Prometheus store. Long retention is considered anything longer than one month. Retention time can be adjusted with the ***storage.local.retention*** flag.

When relatively small amounts of data are removed from storage as they expire, you can use the ***storage.local.series-file-shrink-ratio*** flag to avoid

rewriting large files. The flag represents the percent of the time series being removed from the file that will trigger a rewrite. For example, if the value is 0.2, then 20% of the time series will need to expire before the file will be rewritten and the data removed. The tradeoff is wasting disk space which will need to be considered in settings.

This All Sounds Difficult

Setting up, monitoring, and tweaking settings on a Prometheus cluster is not a simple task. Developers need to know not only what settings are available for change but how one setting's value will affect other aspects of the server's behavior. Tweaking is not something you want to do in a production environment when data loss can be catastrophic, but unfortunately, sometimes it is unavoidable. To avoid data loss, set up and test in an environment as close to your production setup as possible. With deployments on Kubernetes, spinning up a staging cluster to test can be helpful, albeit costly, for scale testing your cluster.

MONITORING PROMETHEUS

Prometheus is used to monitor distributed systems like Kubernetes. But, developers must not discount that Prometheus itself also needs to be monitored. At any given time, developers should be able to know that the metrics collected are accurate. It is widespread for developers setting up the first Prometheus instance in production to have failures. If you can know how to spot the tell-tale signs of an impending crash, Prometheus usually gives you a way to scale before the crash occurs. The trick is to catch the issue before it happens because if your Prometheus server crashes, it takes your metrics with it.

Prometheus Self-Monitoring

Prometheus exposes data about itself in the same way a Kubernetes cluster does. So, while developers can configure Prometheus servers to monitor Kubernetes, they can also configure the servers to monitor themselves or another Prometheus server.

Prometheus servers expose internal metrics on an endpoint named ***/metrics***. By default, they will appear on port 9090, but developers can configure the port if needed. Metrics that are useful for monitoring to ensure data are sent to remote storage appropriately include:

- ***prometheus_remote_storage_failed_samples_total***: a counter of the total number of samples that failed to send to remote (long-term) storage
- ***prometheus_remote_storage_dropped_samples_total:*** a counter of the total number of samples that Prometheus dropped because the queue was full
- ***prometheus_remote_storage_queue_length:*** The number of processed samples in the queue to send to remote (long-term) storage
- ***prometheus_remote_sent_batch_duration_seconds:*** a histogram of the duration of sample batch calls to remote storage

Dead Man's Switch

A dead man's switch is a system that requires a human to intervene; otherwise, the system triggers some event automatically. In Prometheus, Dead Man's switch is an alert manager webhook service for Prometheus.

Developers create an alert using a simple expression that always returns true. Then, configure an alert manager that will recognize this alert and expect to receive it periodically. Developers can configure the alert manager to notify them if the signal is not received and the Prometheus server has gone down.

The dead man's switch is a single concept with Prometheus, but there are many different ways to implement it depending on your toolkit. But, whatever you use, you must have a tool outside the monitored Prometheus server to send an alert. This tool could be another Prometheus server with an alert manager setup or another cloud service like AWS Lambda that will receive the alert through an API.

USING GRAFANA TO VISUALIZE YOUR METRICS

Grafana is an open-source software stack that includes tools to visualize and alert on your metrics. Prometheus does include a tool called the Prometheus Dashboard, but it is most useful for quick data queries. Grafana is a complete visualization tool allowing developers to customize graphs and set up metric alerts.

Deploying Grafana Using the Prometheus Operator

Just like the Prometheus operator, there is an operator pattern set up for deploying Grafana. Using the Grafana Operator instead of a Helm deployment means developers automatically access operational knowledge from Grafana designs. This includes the ability to comanage and configure Grafana with Kubernetes resources like the config maps we used earlier to set up Prometheus. The Grafana Operator also gives automation of tools like the Grafana dashboard plugins and Oauth proxy. The Grafana Operator is a great tool, but since we also need the Prometheus Operator, we can use the kube-prometheus project to install and configure both tools.

To quickly set up your monitoring, clone kube-prometheus from Github using the following command:

```
git clone https://github.com/prometheus-operator/kube-prometheus.git
```

Once downloaded, cd into the Kubernetes project's root directory. Deploy kube-prometheus using the following commands. Note that it may be necessary to run the *kubectl create* command several times before all components are successfully created.

```
# Create the namespace and custom resource definitions (CRD), and then wait for them to be available before creating the remaining resources
kubectl create -f manifests/setup
```

```
# Wait until the “servicemonitors” CRD is created. The
message “No resources found” means success in this
context.
until kubectl get servicemonitors --all-namespaces ; do
date; sleep 1; echo “”; done

kubectl create -f manifests/
```

You now have access to Grafana and can view your dashboard using the following command. After running the command, point a web browser to *http://localhost:3000*. The default login uses admin for both username and password.

```
kubectl --namespace monitoring port-forward svc/grafana
3000
```

Cross-Reference Your Logs and Your Metrics

For the Grafana dashboard to be effective, you need a way to correlate your logs with your metrics so developers can easily navigate issues. When you do not have all your logs, metrics, and traces in one place, developers need to look to Grafana for metrics but then open other tools to check logs or traces to understand what happened.

Since it is an open-source tool, developers can send data to Grafana from different tools. Data can flow in from Prometheus with our existing setup, but it is helpful to have all your data recorded in one place. If you’ve set up Kubernetes logging, you could also export your logs to Grafana for visualization.

Grafana’s Loki does offer a way to correlate your logs to metrics in Grafana. The tool can be used to view both metrics and logs using ad-hoc queries for incident investigations. Loki was not designed to replace analytical solutions, so cannot replace solutions that offer full observability and analytics on your logs, metrics, and traces.

Declare Your Dashboards

Once you have Grafana deployed, you can customize the dashboard to hold data useful for analyzing your own system. One issue with dashboards is they are destroyed when a Kubernetes pod is rebooted. This can happen during a regular update or after a crash has occurred. To get around this problem, define the GrafanaDashboard custom resource. This definition allows Grafana to detect defined dashboards and instantiate them when needed.

<PART 3>

TRACES: THE FORGOTTEN SIBLING

Developers tend to lean on logs and metrics when setting up microservice observability data collection. While these are both important for gaining visibility into your system, they do not inherently give enough context to understand or troubleshoot problems.

Traces are a construct specifically useful for microservices. They add a snippet of unique data to each request, giving the ability to follow it throughout your cluster and understand where it might run into errors or bottlenecks. In a microservice where a request will travel to different functions or pods, being able to track a request for its entire lifespan is an invaluable tool to developers trying to troubleshoot a problem.

THE TOOLING OPTIONS FOR TRACING

There are currently two open-source solutions for monitoring traces in Kubernetes: Jaeger and the combination of Kiali and Istio. These services can all be used in your Kubernetes cluster since they bring different information to your monitoring solution.

Jaeger

Like Kubernetes, the CNCF maintains Jaeger. It is an open-source distributed tracing system used to monitor and troubleshoot microservice architectures like Kubernetes. Jaeger can propagate the distributed context to other data like logs, monitor distributed transactions, optimize performance across your cluster, and more. Using Jaeger, a DevOps user can follow a request's path throughout a distributed system with a visual representation and identify any problem areas such as where the data's flow is slowed.

How to Deploy Jaeger

Deploying Jaeger is complicated because of its architecture. There are clients, agents, and collectors to set up to start the tracing software. Luckily, Jaeger has built an operator to ease the operational complexity of this deployment. Use the following command to get the Jaeger Operator and add Jaeger to the monitoring namespace.

```
kubectl create -f https://github.com/jaegertracing/
jaeger-operator/releases/download/v1.36.0/jaeger-
operator.yaml -n monitoring
```

By default, the operator is installed in cluster mode and will watch for Jaeger-related events in all namespaces, watch the namespaces themselves, and batch all deployments to inject or remove sidecars. To only watch a specific namespace, change the ***ClusterRole*** and ***ClusterBindingRole*** of the operator manifest to ***Role*** and ***RoleBinding,*** respectively. Also, the ***WATCH_NAMESPACE*** environment variable should be set to the namespace to watch.

ISTIO SERVICE MESH

Istio is a service mesh that manages communications between microservices. Service meshes run on sidecar proxies on software nodes. While Istio was designed to run on any platform, it is commonly used with Kubernetes clusters. Software architectures can include hundreds or thousands of Kubernetes nodes that need to communicate with each other. Istio service mesh enables that communication by tracking, queuing, and securing data as it flows from one node to another.

Istio can provide enhanced traffic resilience, collect service-level metrics, provide distributed tracing, improve security, and more. Istio leverages distributed tracing systems like Jaeger to send traces throughout the microservice architecture.

Features of Istio Service Mesh

Istio packs a large punch for platform observability. It is highly configurable, allowing DevOps teams to record traces, logs, and metrics. The biggest feature of a service mesh is its unique ability to create and collect communication data with traces. Since data communicating between nodes is sent through the Istio proxy sidecar, trace data can consistently be added. This means that the route data takes throughout the Kubernetes cluster can be easily tracked and used in conjunction with other observability tools for troubleshooting.

Whenever nodes are communicating, security must be considered. Istio has built-in encryption, authentication, and authorization capabilities to secure data as it flows through software. Further, by using built-in Istio security features, all these capabilities will be configured in one place. This reduces the engineering time needed to set up a custom configuration and also reduces maintenance since all the setup is done only once.

Istio also improves the reliability of Kubernetes platforms by providing load balancing and traffic routing to the sidecar proxies. Istio is able to detect when a node is down or overloaded and can hold back messages until a

node is ready to receive them.

Deploy Istio to Get a Configured Kiali Instance

Istio is a tool that provides observability of your Kubernetes cluster. It provides metrics, traces, and logs that show how monitored services interact with each other. Developers can also configure Istio to monitor multiple clusters with a single mesh.

Istio can be set up to integrate with other observability tools. Kiali can be used to observe the health of the Istio configuration. It can export data to Grafana dashboards to visualize mesh data and also to Jaegar to log traces.

While Istio is a powerful tool, if you already have other services set up to collect metrics, traces, and logs, it may be more than you need. Since the memory and computational requirements of observability tools can be high, collecting redundant data is not ideal. Istio is a good option for new software builds or software that does not have integrated observability tools.

TRACING CAN HAVE SOME DRAWBACKS

The Sudden Spike in Data

Jaeger data requires external storage. Each event flowing through each microservice will receive trace data that is recorded and stored. Both Elasticsearch and Cassandra are available for storage, but Jaeger recommends using Elasticsearch because of its horizontally-scalable storage and fast search capabilities and Elastic's APM server, which is wire compatible with trace data from the Jaeger agent. Production systems will end up with a large spike in data required for long-term storage to keep traces, more so than metrics require.

Tracing is Very Intrusive

Istio and Jaeger are both deployed as sidecars on Kubernetes pods. The sidecar will intercept traffic entering and exiting the pod and record the trace in the defined storage location. In production, busy applications will have significant traffic and required interceptions. When using Istio, each

microservice data flow will record two spans: one from the client sidecar and one from the server sidecar. These spans are then sent to a system like Jaeger for data storage. The volume of data collected by the sidecar can slow the delivery of the data flows they are trying to record. Further, ensure that connections between client and collector use a reverse proxy in front of collectors to ensure the trace data collected is secure.

HOW YOUR LOGS CAN HELP WITH TRACING

Implementing traces is much easier done when writing new code; traces can be added to endpoint calls from the code's setup. Adding traces is more cumbersome if you have existing code because edits are required on existing functions. Further, if nodes along a data path are missing traces, the nodes with the information cannot be accurately used for observability. A different way to gather trace information is to leverage existing logs to collect the data.

Session IDs in All Logs

Trace data adds identifiers to inputs from endpoints so they can be tracked through a distributed architecture. These identifiers allow tools to visualize how long inputs take to flow through the architecture and will also show if they are dropped in a service. Trace data includes the overhead required to record these sessions and visualize the data.

Assuming the system contains logs, these can be leveraged to include trace data. Each log could have a consistent session identifier added to them. The session identifier would indicate the same information as the collected trace data would have but would have an added advantage of always being associated with the appropriate log data.

An MDC (mapped diagnostic context) tool can extract the session identifier and display the log data in a similar format to a trace. Once extracted, the log timestamp and the identifier data can be used to visualize traces in the same way they would be when using a tool like Jaeger to collect trace data.

<PART 4>

BUILD VS. BUY

When embarking on any new software venture, a good engineer should consider the implications of each decision, to ensure that the finished product serves the needs of the business, in a cost-effective way. This mission inevitably leads to a "build vs buy" conversation.

Open Source Isn't Free

Open source is free of licence and support costs. This means that you don't add any new subscriptions to your operational costs, and you are in full control of the solution that you've built. However, this doesn't mean that your open source solution is free. The costs are very real:

- Initial setup costs are incurred when you deploy and experiment with the product to get it working.
- Ongoing engineering effort to build and maintain the necessary features that the open source product inevitably lacks.
- Operational costs in maintaining your solution, once it reaches a steady state.
- The opportunity cost because as a company, you've focused on building your new observability platform, and not your core product.

The Challenge of Tool Sprawl

In addition to these obvious costs, there is a more insidious cost in the open source landscape. Open source tooling tends to do one thing, and one thing well. If you decide to follow a fully open source observability stack, you're going to be spending time knitting together a variety of tools, in order to build out the feature set you need.

This variety of tools increases the learning curve and complicates your stack, which in turn drives the aforementioned costs - it takes longer to do anything, because with more moving parts in your observability stack comes a higher chance of complications.

When is Open Source a Good Idea?

There are some clear arguments for building your own solution. The most common is that if you build your own platform, you have much more freedom to change and rebuild features as you need. If you're doing something highly specialised, or your primary product is your observability data, it's a good argument for favouring flexibility over speed. In this case,

it's then up to you to look at the economics.

Is the flexibility you get from your open source and custom code solution worth the upfront cost, tool sprawl, maintenance effort and opportunity cost involved in building your own custom observability solution? If the answer is yes, then build away! If you're not so sure, then keep reading, because there are other options on the market.

HOW CAN A SAAS OBSERVABILITY PROVIDER WORK FOR YOU?

SaaS solutions come with some excellent benefits, that directly remedy the major hidden costs of open source and custom solutions:

- There is very little upfront cost, because this cost is spread out in the form of a subscription.
- You should spend far less time configuring your SaaS solution, meaning you don't incur the same opportunity cost.
- There is no ongoing engineering effort to build and maintain new features.

All of these things form a compelling case for why most organizations should focus on SaaS solutions, rather than spending precious time and money working on something that has very little to do with your product.

A Word of Warning With SaaS Observability Providers

Many SaaS observability providers attempt a "land and expand" strategy. This means they make an agreement with a company, install lots of proprietary software onto the customer's infrastructure, and make it as difficult as possible for that customer to switch in the future. If you're concerned with vendor lock-in, then a key measurement is if your provider offers a fully open source integration path.

This enables you to switch back to your existing stack, or to a different provider, if you need to. It's important to ensure that you don't fall prey

to these strategies, which are designed to extract as much profit from companies as possible. Instead, look for a partner that will work with you and provide a service that gives you value for money.

How Can Coralogix Bring You World-Class Observability?

Coralogix comes built with best-in-class observability tooling, including logs, metrics, tracing and security data. It does all of this with a revolutionary pricing model that regularly saves customers 40-70% off their observability costs.

It does all of this while offering full integration with OpenTelemetry and providing managed instances of Grafana, Kibana and Jaeger, so you can interface exclusively with open source tooling, if you so choose. This means there is no proprietary software sitting in your infrastructure, locking you in and holding you back.

Feature	ELK + Prometheus Stack	Coralogix
Logs	✓	✓
Metrics	✓	✓
Tracing	✓	✓
Alerting	✓	✓
Open Source Integration	✓	✓
Machine learning driven alerting	✗	✓
Out of the box cost optimisation	✗	✓
Time sequence Flow Alerts	✗	✓
Fully managed Grafana, Kibana and Jaeger	✗	✓

CONCLUSION

We have talked about how metrics are required to know that Kubernetes clusters are functioning properly. Monitoring Prometheus' health is also important to ensure you are not missing data. Traces are another data set that should be collected from Kubernetes and can assist in troubleshooting issues. Tools like Jaeger and Istio can be difficult to set up and will require monitoring quite a lot of data. Traces could be instantiated in logs instead and give the same effect. We've shown how to install tools using Operators to ease the operational burden of setup.

Even with Operators helping to instantiate tools, monitoring and maintaining is still a burden on engineering teams. Developers will need to keep a close watch on visualized data to ensure they are seeing the full picture of their system's health and are collecting data in a cost-effective way for their system.

To avoid tool sprawl in Kubernetes, consider storing data in Observability platforms that can perform functionality provided by open source tools automatically. Such systems can be used with open source observability data recording systems and in some cases can replace tools.

ABOUT CORALOGIX

We're rebuilding the path to observability using a real-time streaming analytics pipeline that provides monitoring, visualization, and alerting capabilities without the burden of indexing.

By enabling users to define different data pipelines per use case, we provide deep insights for less than half the cost.

In short, we are streaming the future of data.

Built for tomorrow's data scale

2K+
Global Customers

10K+
DevOps and Engineering Users

500K+
Applications Monitored

3M+
Events Processed Per Second

Your data is telling yesterday's story — Find out what it can tell you today.

Create an Account | Get a Demo