PowerDNS Cloud Control

Monitoring

Apr 14, 2024

Release 2.6.0

Contents

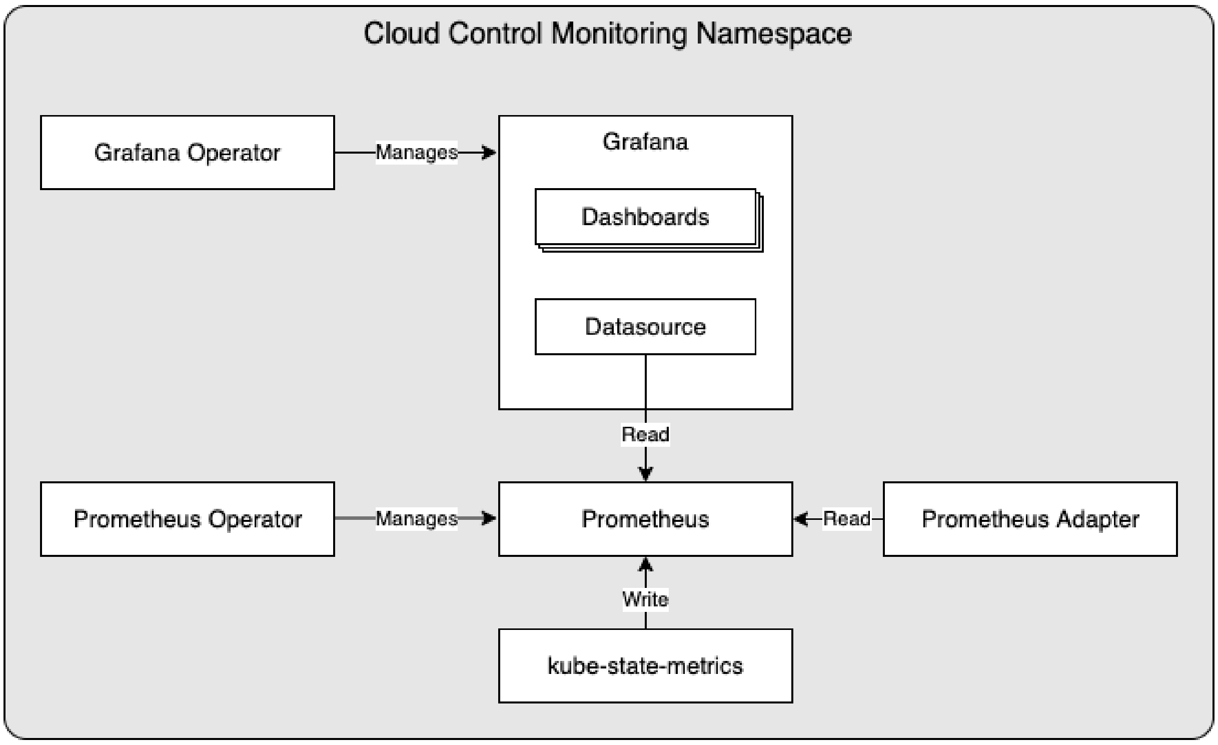
1 Overview

1.1 Cloud Control Monitoring

Cloud Control Monitoring helps provide insight into Cloud Control deployments. The monitoring stack is built using the following components:

- **Grafana** - Visualisation
- **Prometheus** - Metrics gathering & storage
- **kube-state-metrics (KSM)** - Service that exposes Kubernetes metrics to prometheus
- **Prometheus Adapter** - Exposes Prometheus metrics via Kubernetes Metrics API

The stack can be deployed in its entirety, or partially depending on your existing monitoring infrastructure. When the full stack is deployed, the result will be a namespace containing the following components:



2 Helm Charts

2.1 Charts

The Helm charts which are available to deploy this stack are as follows:

- **monitoring-operators** - Deploy Grafana & Prometheus operators + accompanying CRDs
- **monitoring** - Deploy Prometheus & Grafana (including dashboards + datasource) using above mentioned operators. Also deploys KSM & Prometheus Adapter.

Since the monitoring chart depends on the availability of the operators, the monitoring-operators chart needs to be installed prior to the monitoring chart if you intend to deploy Grafana and/or Prometheus.

Note: Due to a restriction in the Grafana Operator you must deploy both charts into the same namespace if you are deploying the Grafana Operator

2.2 Usage

2.2.1 Install Tools

You will need the following software on the machine from which you want to deploy Cloud Control Monitoring:

- Kubectl (Configured for your target Kubernetes cluster)
- Helm (3.8.0 or newer - `https://helm.sh/docs/intro/install/`)

2.2.2 Download Helm Charts

Cloud Control Monitoring Helm Charts are available on the Open-Xchange registry, located at: registry.open-xchange.com.

There are several methods for obtaining Helm Charts using Helm's CLI, in this chapter we are using a method that copies the chart locally to your filesystem prior to using it. Any Helm-supported method will work, but you will need to adjust the commands in this guide accordingly if you wish to utilise a different method.

First step will be to login to the OX registry (replace username & password with your OX registry credentials):

```
helm registry login registry.open-xchange.com --username=REGISTRY_USERNAME_HERE \
--password=REGISTRY_PASSWORD_HERE
```

Once helm has been logged in to the OX registry you can access the Cloud Control Monitoring Helm Charts. To pull the monitoring Helm Charts and export them to your current working directory use the following commands:

```
# Pull & unpack Operators chart
helm pull oci://registry.open-xchange.com/cloudcontrol/monitoring-operators \
--version=2.6.0 --untar

# Pull & unpack Monitoring chart
helm pull oci://registry.open-xchange.com/cloudcontrol/monitoring \
--version=2.6.0 --untar
```

2.2.3 Deploying Cloud Control Monitoring

To deploy the monitoring stack without any customization you can use the following steps:

```
# The namespace
CC_MON_NAMESPACE=ccmon
HELM_RELEASE=ccmon

# Deploy the monitoring operators & CRDs
helm install $HELM_RELEASE-operators ./monitoring-operators --namespace $CC_MON_NAMESPACE \
--create-namespace

# Deploy the monitoring stack
helm install $HELM_RELEASE ./monitoring --namespace $CC_MON_NAMESPACE
```

Note: you can remove `--create-namespace` if you have an existing namespace to deploy into

2.2.4 Accessing Grafana

You can use kubectl's `port-forwarding` to quickly access the Grafana service:

```
# The namespace
CC_MON_NAMESPACE=ccmon

kubectl --namespace=$CC_MON_NAMESPACE port-forward svc/grafana 3000:grafana
```

You can now visit Grafana at: `http://localhost:3000/`

When prompted for a username/password, you can login using the username configured in 'grafana.admin.username' and based on the 'grafana.admin.password' setting a static or dynamically generated password. To customize this behaviour, you can modify the following block in the helm values:

```
grafana:
  # UI Access
  admin:
    # Grafana admin credentials
    username: admin
    # password: some_password
```

If no password is specified (as in the example above), a random password will be generated and stored in Secret: grafana-credentials

For a more permanent method of accessing Grafana, refer to the *Configuration* chapter to configure an Ingress object.

2.2.5 Accessing Prometheus

You can use kubectl's `port-forwarding` to quickly access the Prometheus service:

```
# The namespace
CC_MON_NAMESPACE=ccmon

kubectl --namespace=$CC_MON_NAMESPACE port-forward svc/prometheus 9090:web
```

You can now visit Prometheus at: `http://localhost:9090/`

For a more permanent method of accessing Prometheus, refer to the *Configuration* chapter to configure an Ingress object.

3 OpenShift

3.1 OpenShift

3.1.1 Compatibility mode

Cloud Control Monitoring provides an OpenShift compatibility mode which will make sure default settings adhere to a set compatible with policies present on a default OpenShift cluster.

To enable this compatibility mode, ensure the following is present in your Helm values overrides:

```
global:
  openshift:
    enabled: true
```

Note: This should be used for both the 'Monitoring' and 'Monitoring Operators' Helm charts.

When enabled, Cloud Control Monitoring will have a minimal set of configuration objects adjusted in terms of default settings:

- Default Security Context on Pod objects will no longer set these parameters: 'fsGroup', 'runAsUser' and 'runAsGroup'

For each affected set of configuration items in the above list, you can still apply your own customisations. Below sections explain how you can modify these when you are running with OpenShift compatibility mode enabled.

podSecurityContext

By default, all pods will have a 'podSecurityContext' enforcing 'runAsNonRoot' and nothing else in OpenShift compatibility mode.

To adjust the podSecurityContext configured for pods in the OpenShift compatibility mode, you can use the following configuration:

```
global:
  openshift:
    enabled: true
    podSecurityContext:
      <Contents of Pod Security Context>
```

3.1.2 Conflicts with OpenShift-managed components

Many OpenShift clusters have pre-installed components which conflict with the Cloud Control Monitoring landscape, these likely include:

- Monitoring Operators: Prometheus
- Monitoring: Kube-state-metrics, Prometheus Adapter

If this is the case on your cluster, you can set the *enabled* flag to *false* for these components as documented in the 'Configuration' chapter.

4 Prometheus Adapter

4.1 Adapter

Included in the monitoring stack is the Prometheus Adapter (`https://github.com/kubernetes-sigs/prometheus-adapter`). This component is used to expose metrics via the Kubernetes API, based on data stored in Prometheus. Exposing these metrics via the Kubernetes API enables the use of Horizontal Pod Autoscalers in Cloud Control deployments.

4.2 Predefined Metrics

By default a few basic metrics are exposed via the adapter, which are configured in the helm values under the section 'prometheus-adapter.rules.custom'. The format that these metrics must follow is defined here: https://github.com/kubernetes-sigs/prometheus-adapter/blob/master/docs/config.md

One of the default values available is the amount of queries received by dnsdist, defined as follows:

```
- seriesQuery: 'dnsdist_queries'
  resources:
    overrides:
      namespace: {resource: "namespace"}
      pod: {resource: "pod"}
  name:
    matches: "^(.*)"
    as: "${1}_per_second"
  metricsQuery: 'sum(rate(<<.Series>>{<<.LabelMatchers>>}[1m])) by (<<.GroupBy>>)'
```

Based on the 'counter' type metric 'dnsdist_queries' in Prometheus, the rate (ie: the increase of the metric over a period of time) at which it increases per second is calculated, giving the amount of queries handled per second over the interval. This makes a metric available via the Kubernetes API named "dnsdist_queries_per_second"

Another example focuses on a 'guage' type metric, exposing the average latency reported for a dnsdist instance:

```
- seriesQuery: '{__name__=~"^dnsdist_latency_avg.*$"}'
  resources:
    overrides:
      namespace: {resource: "namespace"}
      pod: {resource: "pod"}
  name:
    matches: ""
    as: ""
  metricsQuery: 'avg(<<.Series>>{<<.LabelMatchers>>}) by (<<.GroupBy>>)'
```

Since there are several guages that match the query expression, this leads to the following metrics being available via the Kubernetes API:

- dnsdist_latency_avg100
- dnsdist_latency_avg1000
- dnsdist_latency_avg10000
- dnsdist_latency_avg1000000

To see all the metrics exposed via the Prometheus Adapter, you can use kubectl:

```
# Note: If you have other sources for metrics available on your cluster this might be a␣
↪long list
kubectl get --raw /apis/custom.metrics.k8s.io/v1beta1/ | jq
```

4.3 Horizontal Pod Autoscaler

These metrics are made available to allow for using Kubernetes' Horizontal Pod Autoscaler to be used to automatically scale up & down deployments. Since the Horizontal Pod Autoscaling functionality is part of the actual deployment of dnsdist & Recursor, you can find more about this part of the configuration in the Cloud Control deployment documentation.

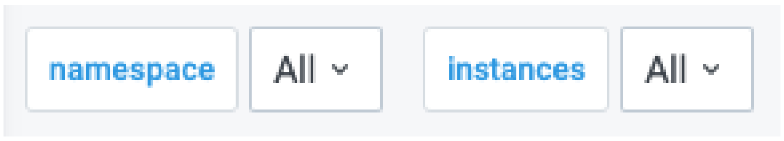
5 Grafana Dashboards

5.1 Dashboards

When the ‘monitoring’ chart is used to deploy the stack, several monitoring dashboards will be provisioned automatically. Currently, this includes the following dashboards:

- **PowerDNS dnsdist** Detailed insight into running dnsdist instances
- **PowerDNS Recursor overview** High-level overview of running Recursor instances
- **PowerDNS Recursor details** Detailed insight into running Recursor instances
- **PowerDNS Authoritative details** Detailed insight into running Authoritative instances
- **PowerDNS Authoritative - Lightningstream** Detailed insight into running Lightningstream instances
- **PowerDNS dstore-dist** Detailed insight into running dstore-dist instances

All dashboards have selectors available, which you can use to view details regarding specific instances or instances within a namespace. These selectors are located top-left of each dashboard and look as follows:



5.2 Download

If you opt to utilize an existing Grafana installation, you can download the dashboards from the the Open-Xchange registry, located at: registry.open-xchange.com

The dashboards are stored inside an OCI artifact, so you will have to use a compatible client to obtain them. Recommended client to use for this is: ORAS (`https://oras.land/`)

With a CLI such as oras available, you can download the dashboards artifact via:

```
oras pull registry.open-xchange.com/cloudcontrol/monitoring-dashboards:2.6.0
```

The result will be a tar archive which contains all the dashboards in JSON format.

6 Prometheus Alert Rules

6.1 Alert Rules

When the ‘monitoring’ chart is used to deploy the stack, several prometheus alert rules will be provisioned automatically. Currently, this includes alert rules for the following:

- **PowerDNS Dnsdist**
- **PowerDNS Recursor**
- **PowerDNS Authoritative Server**

6.2 Download

If you opt to utilize an existing Prometheus installation, you can download the alert rules from the Open-Xchange registry, located at: registry.open-xchange.com

The alert rules are stored inside an OCI artifact, so you will have to use a compatible client to obtain them. Recommended client to use for this is: ORAS (`https://oras.land/`)

With a CLI such as oras available, you can download the alert rules artifact via:

```
oras pull registry.open-xchange.com/cloudcontrol/monitoring-alertrules:2.6.0
```

The result will be a tar archive which contains all the alert rules in YAML format.

7 Component Reference

7.1 Operators

The following operators are utilized in the monitoring stack:

Prometheus Operator: `https://github.com/prometheus-operator/prometheus-operator`

Grafana Operator: `https://github.com/grafana-operator/grafana-operator`

7.2 Components

The following components are utilized in the monitoring stack:

Prometheus: `https://prometheus.io/`

Grafana: `https://grafana.com/oss/grafana/`

Prometheus Adapter: `https://github.com/kubernetes-sigs/prometheus-adapter`

kube-state-metrics: `https://github.com/kubernetes/kube-state-metrics`

8 Configuration

8.1 Configuration

8.1.1 Enable/Disable components

The helm charts have 'enabled' flags available for all major components. The following sections describe how you can use these to enable/disable components.

Monitoring Operators

This chart installs the following operators & accompanying CRDs:

- Grafana Operator
- Prometheus Operator

By default all components are installed, this can be controlled using the 'enabled' settings in the Helm values:

```yaml
# Grafana Operator
grafana-operator:
    # Enable the Grafana Operator deployment
    enabled: true
```

```yaml
# Prometheus Operator
prometheus-operator:
    # Enable the Prometheus Operator deployment
    enabled: true
```

Setting these to 'false' will stop the operator (& CRDs) from being deployed.

Monitoring

This chart installs the following components:

- Grafana
- Grafana Dashboards
- Prometheus
- Prometheus Adapter
- kube-state-metrics (KSM)

By default all components are installed, this can be controlled using the 'enabled' settings in the Helm values:

```yaml
# Grafana
grafana:
    # Enable the Grafana deployment
    enabled: true

    # Create GrafanaDashboard objects for PowerDNS products
    dashboards: true
```

```yaml
# Prometheus
prometheus:
    # Enable the Prometheus deployment
    enabled: true
```

```yaml
# Prometheus Adapter
prometheus-adapter:
    # Enable the Prometheus Adapter deployment
    enabled: true
```

```yaml
# kube-state-metrics
kube-state-metrics:
    # Enable the Kube State Metrics deployment
    enabled: true
```

Setting these to 'false' will stop the component from being deployed.

8.1.2 Ingresses & Loadbalancers

The following components have configuration options to add an Ingress and/or Loadbalancer to expose them outside of the cluster:

- Grafana
- Prometheus

To configure an Ingress and/or Loadbalancer, you can override the 'ingress' and 'service' configuration under the 'prometheus' & 'grafana' sections in the helm values. The following sections describe how you can use these to configure them.

Ingress

By default, the ingress is disabled:

```yaml
# Ingress configuration
ingress:
  # Enable the Ingress
  enabled: false
```

To create an ingress which only serves HTTP, set 'enabled' to true and add the hosts on which you want the ingress to listen.

Example which exposes Prometheus on '`http://prometheus.example.com`' using the NGINX Ingress Controller:

```
ingress:
  enabled: true
  ingressClassName: "nginx"
  hosts:
    - prometheus.example.com
```

To create an ingress which serves HTTPS (and has an HTTP->HTTPS redirect), provide a 'tls' configuration block.

Example which exposes Prometheus on '`https://prometheus.example.com`' via the NGINX Ingress Controller and a pre-existing certificate in a secret named 'prometheus-cert':

```
ingress:
  enabled: true
  ingressClassName: "nginx"
  hosts:
    - prometheus.example.com
  tls:
    - secretName: prometheus-cert
      hosts:
      - prometheus.example.com
```

Example which exposes Prometheus on '`https://prometheus.example.com`' via the NGINX Ingress Controller and an on-demand certificate provisioned by 'cert-manager' in a secret named 'prometheus-cert' (note the additional annotation):

```
ingress:
  enabled: true
  ingressClassName: "nginx"
  annotations:
    cert-manager.io/cluster-issuer: ca-issuer
  hosts:
    - prometheus.example.com
  tls:
    - secretName: prometheus-cert
      hosts:
      - prometheus.example.com
```

Loadbalancer

By default, services of type 'ClusterIP' are created. To expose the service using a loadbalancer, set the type to 'LoadBalancer' and add the necessary additional configuration, based on your LoadBalancer provider. Example Grafana service configuration in a cluster with MetalLB:

```
# Grafana Service
service:
  type: LoadBalancer
  annotations:
    metallb.universe.tf/address-pool: name_of_pool
  loadBalancerIP: 12.34.56.78 # Omit this to have a random IP assigned from the pool
  ports:
    grafana-http:
      port: 3000
```

For a NodePort service, set the type to 'NodePort' and if desirable, specify the nodePort:

```
# Grafana Service
service:
  type: NodePort
  ports:
    grafana-http:
      port: 3000
      nodePort: 30003
```

8.1.3 Private Registries

All images referenced by the monitoring & monitoring-operators charts are available on public registries. If you intend to run the monitoring stack on a kubernetes cluster which makes use of a local registry, you can use one or more of the following settings in your helm values to configure that registry:

```
# Monitoring - global overrides
global:
  # Override image-related settings for this chart and all subcharts
  image:
    # Override registry for all images
    registry: "myregistry.local:8085"

    # Override repository for all images
    repository: "myrepository"

    # Override pullPolicy for all images
    pullPolicy: "IfNotPresent"

  # Add imagePullSecrets for this chart and all subcharts
  imagePullSecrets:
    myIPSSecret:
      registry: registry.example.com:5000
      username: regUsername
      password: regPassword
      email: admin@registry.example.com

  # Reference existing Image Pull Secrets for this chart and all subcharts
  imagePullSecretsList:
    - "my-existing-IPS-secret"
```

Each setting explained:

global.image.registry

All images will be attempted to be pulled from this registry (format: host:port)

global.image.repository

All images will be attempted to be pulled from this repository, on above configured registry

global.image.pullPolicy

This pull policy will be specified for each image

global.imagePullSecrets

For each entry a Secret will be created and assigned to each Pod

global.imagePullSecretsList

Each pre-existing Secret referenced in this list (by name) will be assigned to each Pod

If you have a need to override the above settings for specific images, you can find the corresponding 'image:' configuration blocks in the values file.

8.1.4 Security contexts

By default Cloud Control Monitoring deploys all Pods with a security context which configures the following items:

```yaml
securityContext:
  fsGroup: 2000
  runAsUser: 2000
  runAsNonRoot: true
  runAsGroup: 1000
```

Note: Some Pods have different numerical values for fsGroup, runAsUser & runAsGroup as they have been kept at the defaults provided by open source projects

And all containers have the following applied:

```yaml
securityContext:
  readOnlyRootFilesystem: true
  allowPrivilegeEscalation: false
  seccompProfile:
    type: RuntimeDefault
  capabilities:
    drop:
      - "ALL"
```

Monitoring Operators - Pod Security Context

To overwrite the Pod securityContext for the operators you can add a *podSecurityContext* at the component-level:

```yaml
# Prometheus Operator
prometheus-operator:
  podSecurityContext:
    runAsUser: 1000953

# Grafana Operator
grafana-operator:
  podSecurityContext:
    runAsUser: 1000953
```

Since the configuration of this podSecurityContext overwrites the full default Pod securityContext, this leads to the following differences compared to the defaults:

- fsGroup - This is no longer present, so it is dropped from the Pod security context
- runAsUser - This is now set to '1000953'
- runAsGroup - This is no longer present, so it is dropped from the Pod security context

- runAsNonRoot - This is no longer present, so it is dropped from the Pod security context

Monitoring Operators - Container Security Context

To overwrite the Container securityContext for the operators you can add a *containerSecurityContext* at the component-level:

```yaml
# Prometheus Operator
prometheus-operator:
  containerSecurityContext:
    seccompProfile:
      type: Localhost
      localhostProfile: profiles/audit.json

# Grafana Operator
grafana-operator:
  containerSecurityContext:
    seccompProfile:
      type: Localhost
      localhostProfile: profiles/audit.json
```

Since the configuration of this containerSecurityContext overwrites the full default Container securityContext, this leads to the following differences compared to the defaults:

- allowPrivilegeEscalation - This is no longer present, so it is dropped from the Container security context
- capabilities - This is no longer present, so it is dropped from the Container security context
- readOnlyRootFilesystem - This is no longer present, so it is dropped from the Container security context
- secCompProfile - This is now set to a Localhost profile configuration

Monitoring - Pod Security Context

To overwrite the Pod securityContext for the monitoring Pods you can add a *podSecurityContext* at the component-level:

```yaml
# Grafana
grafana:
  podSecurityContext:
    runAsUser: 1000953

# Prometheus
prometheus:
  podSecurityContext:
    runAsUser: 1000953

# KSM
kube-state-metrics:
  podSecurityContext:
    runAsUser: 1000953

# Prometheus Adapter:
```

(continues on next page)

(continued from previous page)

```
prometheus-adapter:
  podSecurityContext:
    runAsUser: 1000953
```

Since the configuration of this podSecurityContext overwrites the full default Pod securityContext, this leads to the following differences compared to the defaults:

- fsGroup - This is no longer present, so it is dropped from the Pod security context
- runAsUser - This is now set to '1000953'
- runAsGroup - This is no longer present, so it is dropped from the Pod security context
- runAsNonRoot - This is no longer present, so it is dropped from the Pod security context

Monitoring - Container Security Context

To overwrite the Container securityContext for the monitoring Pods you can modify the following parameters at the component-level:

```
# Grafana
grafana:
  containerSecurityContext:
    seccompProfile:
      type: Localhost
      localhostProfile: profiles/audit.json

# Prometheus
prometheus:
  containers:
    configReloader:
      containerSecurityContext:
        seccompProfile:
          type: Localhost
          localhostProfile: profiles/audit.json
    prometheus:
      containerSecurityContext:
        seccompProfile:
          type: Localhost
          localhostProfile: profiles/audit.json
    thanos:
      containerSecurityContext:
        seccompProfile:
          type: Localhost
          localhostProfile: profiles/audit.json

# KSM
kube-state-metrics:
  containerSecurityContext:
    seccompProfile:
      type: Localhost
      localhostProfile: profiles/audit.json

# Prometheus Adapter:
prometheus-adapter:
```

(continues on next page)

(continued from previous page)

```
  containerSecurityContext:
    seccompProfile:
      type: Localhost
      localhostProfile: profiles/audit.json
```

Note: The Prometheus Pod has multiple containers which can be modified individually

Since the configuration of this containerSecurityContext overwrites the full default Container securityContext, this leads to the following differences compared to the defaults:

- allowPrivilegeEscalation - This is no longer present, so it is dropped from the Container security context
- capabilities - This is no longer present, so it is dropped from the Container security context
- readOnlyRootFilesystem - This is no longer present, so it is dropped from the Container security context
- secCompProfile - This is now set to a Localhost profile configuration

8.1.5 Scheduling controls

You can configure the scheduling of Monitoring components using the standard Kubernetes controls:

- Affinity
- Node Selector
- Tolerations

Overriding Affinity

Affinity can be configured via the 'affinity' override at component-level. This override should hold a standard Kubernetes configuration, for example:

```
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
        - key: topology.kubernetes.io/zone
          operator: In
          values:
          - antarctica-east1
          - antarctica-west1
```

Note: This appends to a default podAntiAffinity which prefers to schedule multiple Pods of the same type across different nodes

More information regarding the syntax can be found at: https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node

Overriding Node Selector

Node Selector can be configured via the 'nodeSelector' override at component-level. This override should hold a standard Kubernetes configuration, for example:

```
nodeSelector:
  some_node_label: some-value
```

More information regarding the syntax can be found at: https://kubernetes.io/docs/tasks/configure-pod-container/assign-pods-nodes/

Overriding Tolerations

Tolerations can be configured via the 'tolerations' override at component-level. This override should hold a standard Kubernetes configuration, for example:

```
tolerations:
- key: "example-key"
  operator: "Exists"
  effect: "NoSchedule"
```

Note: This appends to the default tolerations which are set by the Kubernetes cluster on which you deploy.

More information regarding the syntax can be found at: https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/

Monitoring Operators - Pod Scheduling

To overwrite the Pod scheduling configuration for the operators you can use the following component-level overrides:

For example:

```
# Prometheus Operator
prometheus-operator:
  affinity:
    {}
  nodeSelector:
    {}
  tolerations:
    []

# Grafana Operator
grafana-operator:
  affinity:
    {}
  nodeSelector:
    {}
  tolerations:
    []
```

Monitoring - Pod Scheduling

To overwrite the Pod scheduling configuration for the monitoring Pods you can use the following component-level overrides:

For example:

```
# Grafana
grafana:
  affinity:
    {}
  nodeSelector:
    {}
  tolerations:
    []

# Prometheus
prometheus:
  affinity:
    {}
  nodeSelector:
    {}
  tolerations:
    []

# KSM
kube-state-metrics:
  affinity:
    {}
  nodeSelector:
    {}
  tolerations:
    []

# Prometheus Adapter:
prometheus-adapter:
  affinity:
    {}
  nodeSelector:
    {}
  tolerations:
    []
```

Deploying dashboards in existing Grafana deployment

If you have an existing deployment of Grafana managed by the Grafana Operator, you can use this chart to deploy only the Grafana dashboards. Additionally, you can add labels to the dashboards to ensure your Grafana instance sees them as eligible dashboards.

You can use the following configuration to deploy only the dashboards, with a custom label added to each dashboard:

```
# Grafana
grafana:
  enabled: false
```

(continues on next page)

(continued from previous page)

```
  # Create GrafanaDashboard objects for PowerDNS products with custom labels
  dashboards: true
  dashboardLabels:
    monitoring.my.company/grafana_label: some_value

# Prometheus
prometheus:
  enabled: false

# KSM
kube-state-metrics:
  enabled: false

# Prometheus Adapter:
prometheus-adapter:
  enabled: false
```

8.1.6 Prometheus: Storage

The Prometheus deployment can be configured to use persistent storage:

```
# Prometheus
prometheus:
  # Storage configuration options
  storage:
    persistent: true
```

By default, this will use the cluster's default storage class to create a 5 GB persistent volume to store Prometheus data.

To further configure the persistent storage, the following options are available:

- annotations - Annotations to assign to the persistent volume claim
- labels - Labels to assign to the persistent volume claim
- size - Size of the persistent volume
- storageClassName - Storage class to request persistent volume from

For example, you could configure the persistent storage as follows:

```
# Prometheus
prometheus:
  # Storage configuration options
  storage:
    persistent: true
    size: 10Gi
    labels:
      label1: some_value
      label2: some_other_value
    storageClassName: my_storage-class
```

8.1.7 Prometheus: Thanos

The Prometheus deployment can be configured to include a Thanos sidecar. By default, this is disabled and no external labels are configured for deduplication:

```yaml
# Prometheus
prometheus:
  # External Labels
  externalLabels: {}

  # Thanos configuration
  thanos:
    # toggle for thanos sidecar
    enabled: false
```

A simple deployment of the Thanos sidecar can be done via:

```yaml
# Prometheus
prometheus:
  # External Labels
  externalLabels:
    datacenter: dc1

  # Thanos configuration
  thanos:
    # toggle for thanos sidecar
    enabled: true
```

This will cause the Prometheus Pods to have:

- An additional container named 'thanos-sidecar'
- External label 'datacenter=dc1' for data accessed via the Thanos sidecar
- An inbound GRPC Service
- An inbound HTTP Service

To further configure the Thanos sidecar you can refer to the following subchapters.

Inbound GRPC traffic

Inbound GRPC traffic can be configured primarily via the following options:

```yaml
# Prometheus
prometheus:
  # External Labels
  externalLabels:
    datacenter: dc1

  # Thanos configuration
  thanos:
    # toggle for thanos sidecar
    enabled: true

    # Thanos GRPC
    grpc:
```

(continues on next page)

(continued from previous page)

```
      # GRPC Service
      service:
        << Service config here >>

      # TLS config
      tls:
        << TLS config here >>

      # Client CA config
      clientca:
        << Client CA config here >>
```

The Service can be configured to allow inbound traffic from outside of the cluster. For example using a LoadBalancer provided by MetalLB:

```
# Prometheus
prometheus:
  # External Labels
  externalLabels:
    datacenter: dc1

  # Thanos configuration
  thanos:
    # toggle for thanos sidecar
    enabled: true

    # Thanos GRPC
    grpc:
      # GRPC Service
      service:
        type: LoadBalancer
        annotations:
          metallb.universe.tf/address-pool: name_of_pool
```

Inbound TLS can be configured in 2 ways:

- Referring to an existing Secret of type 'kubernetes.io/tls'
- Providing a 'key' and 'cert' inline in the values overrides

Example using an existing Secret:

```
# Prometheus
prometheus:
  # External Labels
  externalLabels:
    datacenter: dc1

  # Thanos configuration
  thanos:
    # toggle for thanos sidecar
    enabled: true

    # Thanos GRPC
    grpc:
      tls:
        secret: thanos-cert
```

This will attempt to use the 'tls.key' & 'tls.crt' from the 'thanos-cert' Secret to enforce TLS on the inbound GRPC traffic.

Example using inline key & cert:

```
# Prometheus
prometheus:
  # External Labels
  externalLabels:
    datacenter: dc1

  # Thanos configuration
  thanos:
    # toggle for thanos sidecar
    enabled: true

    # Thanos GRPC
    grpc:
      tls:
        key: |-
          -----BEGIN RSA PRIVATE KEY-----
          KEY_HERE
          -----END RSA PRIVATE KEY-----
        cert: |-
          -----BEGIN CERTIFICATE-----
          CERT_1_HERE
          -----END CERTIFICATE-----
          -----BEGIN CERTIFICATE-----
          CERT_2_ HERE
          -----END CERTIFICATE-----
```

This will ensure Helm creates a Secret containing the key & cert, which will then be used to enforce TLS on the inbound GRPC traffic.

If TLS is enabled, verification of clients using a certificate can also be performed. To do this, the Thanos sidecar must be configured with a CA against which the clients are verified. Similar to TLS traffic, this can be configured in two ways:

- Referring to an existing Secret
- Providing a 'ca' inline in the values overrides

Example using an existing Secret:

```
# Prometheus
prometheus:
  # External Labels
  externalLabels:
    datacenter: dc1

  # Thanos configuration
  thanos:
    # toggle for thanos sidecar
    enabled: true

    # Thanos GRPC
    grpc:
      clientca:
```

(continues on next page)

(continued from previous page)

```
        secret: ca-cert
        caKey: ca.crt
```

This will attempt to use the CA in the 'ca.crt' key from the 'ca-cert' Secret to enforce client certificate checks against.

Example using inline ca:

```
# Prometheus
prometheus:
  # External Labels
  externalLabels:
    datacenter: dc1

  # Thanos configuration
  thanos:
    # toggle for thanos sidecar
    enabled: true

    # Thanos GRPC
    grpc:
      clientca:
        ca: |-
          -----BEGIN CERTIFICATE-----
          CA_CERT_HERE
          -----END CERTIFICATE-----
```

This will ensure Helm creates a Secret containing the CA, which will then be used to enforce client certificate checks against.

Object Storage

The Thanos sidecar can be configured to write data to object storage using the 'uploadConfig' parameter:

```
# Prometheus
prometheus:
  # External Labels
  externalLabels:
    datacenter: dc1

  # Thanos configuration
  thanos:
    # toggle for thanos sidecar
    enabled: true

    uploadConfig:
      << Object storage config here >>
```

Object storage can be configured in 2 ways:

- Providing configuration inline in the values overrides
- Referring to an existing Secret containing the object storage configuration

Example using inline configuration:

```
# Prometheus
prometheus:
  # External Labels
  externalLabels:
    datacenter: dc1

  # Thanos configuration
  thanos:
    # toggle for thanos sidecar
    enabled: true

    uploadConfig:
      type: s3
      config:
        bucket: NAME_OF_BUCKET
        endpoint: S3_ENDPOINT
        access_key: ACCESS_KEY
        secret_key: SECRET_KEY
```

The above will configure Thanos to store metrics into an S3 bucket. The syntax is identical to the configuration described in Thanos documentation for the parameter '–objstore.config'.

Example using an existing Secret:

```
# Prometheus
prometheus:
  # External Labels
  externalLabels:
    datacenter: dc1

  # Thanos configuration
  thanos:
    # toggle for thanos sidecar
    enabled: true

    uploadSecret: thanos-objstore-config
    uploadSecretKey: objstore
```

The above will attempt to load the Object Storage config from a key named 'objstore' in a secret named 'thanos-objstore-config'.

Additional Configuration

The Thanos sidecar can be configured in more detail using the 'config' parameter:

```
# Prometheus
prometheus:
  # External Labels
  externalLabels:
    datacenter: dc1

  # Thanos configuration
  thanos:
    # toggle for thanos sidecar
    enabled: true
```

(continues on next page)

(continued from previous page)

```
  config:
    {}
```

Parameters available under 'config' are:

- blockSize - This duration controls the size of TSDB blocks produced by Prometheus. (Default is '2h')
- logLevel - Log level for the Thanos sidecar (Default is 'info')
- logFormat - Log format for the Thanos sidecar (Default is 'logfmt')