



Optimizer Hub Runbook

Runbook 25.08.1

Table of Contents

Document Information	1
Revision History	1
Feedback and Updates	1
Executive Summary	2
Architecture Overview	3
System Components	4
High Availability Architecture	5
Prerequisites and Dependencies	6
Required Software Versions	6
Supported Platforms	6
Network Requirements	7
Storage Requirements	7
Required Roles and Permissions	8
Using Externally Defined Secrets	8
Installation Procedures	10
Prepare a Kubernetes Cluster	10
Standard Installation	10
Step 1: Prepare Helm Repository	10
Step 2: Create Namespace	11
Step 3: Create Configuration File	11
Step 3: Install Optimizer Hub	11
Step 4: Install Monitoring	11
Step 5: Verify Installation	12
ReadyNow Orchestrator Only Installation	12
Configuration Management	13
Configuring Scaling	13
Understanding Optimizer Hub Sizing	13

Manual Scaling	13
Autoscaling Configuration	14
Configuring SSL/TLS	15
Enable SSL in Optimizer Hub	15
Configure JVM Clients	15
Storage Cleanup	16
Code Cache Cleanup	16
ReadyNow Profile Log Cleanup	16
Operational Procedures	18
Daily Operations	18
Checking the System Health	18
Checking the Connected JVMs	18
Upgrade Procedures	18
Pre-Upgrade Checklist	18
Standard Upgrade	18
Disaster Recovery Procedure	19
Monitoring and Alerting	20
Reviewing Logs	20
Extracting Logs with Loki (if configured)	20
Using Prometheus and Grafana	21
Configuring Prometheus	21
Configuring Grafana	22
Metrics Endpoints	23
Legal Notice	24

Document Information

Document Version	1.0
Last Updated	{date}
Optimizer Hub Version	25.08.1
Maintained By	Platform Operations Team

Revision History

Version	Date	Changes
1.0		Initial runbook creation based on Optimizer Hub 25.08.0 documentation

Feedback and Updates

This runbook gets reviewed and updated:

- With each Optimizer Hub release
- When operational procedures change
- After any major incident

To suggest improvements or report issues with this runbook, contact support@azul.com.

Executive Summary

This runbook provides operational procedures for managing Azul Optimizer Hub, a component of Azul Platform Prime that speeds up Java application performance through server-side JIT compilation and ReadyNow profile orchestration.

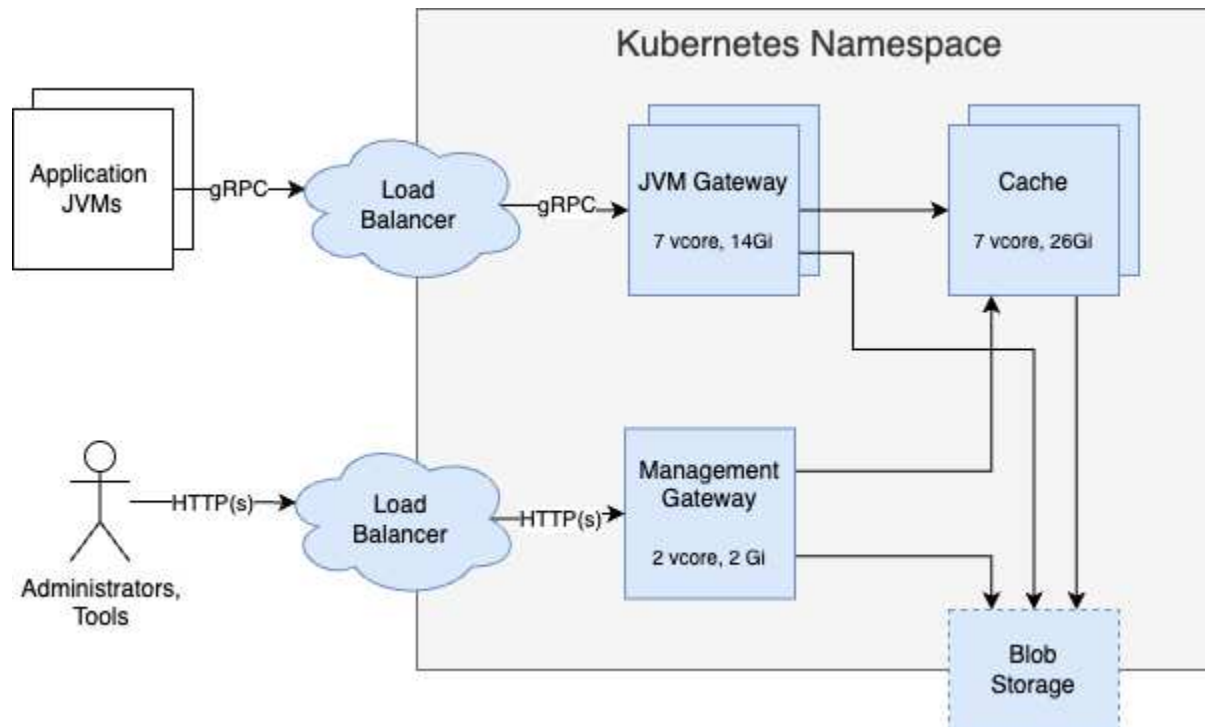
Key Components:

- **Cloud Native Compiler (CNC):** Offloads JIT compilation from client JVMs to dedicated server resources
- **ReadyNow Orchestrator:** Manages and serves ReadyNow profiles for rapid JVM warmup

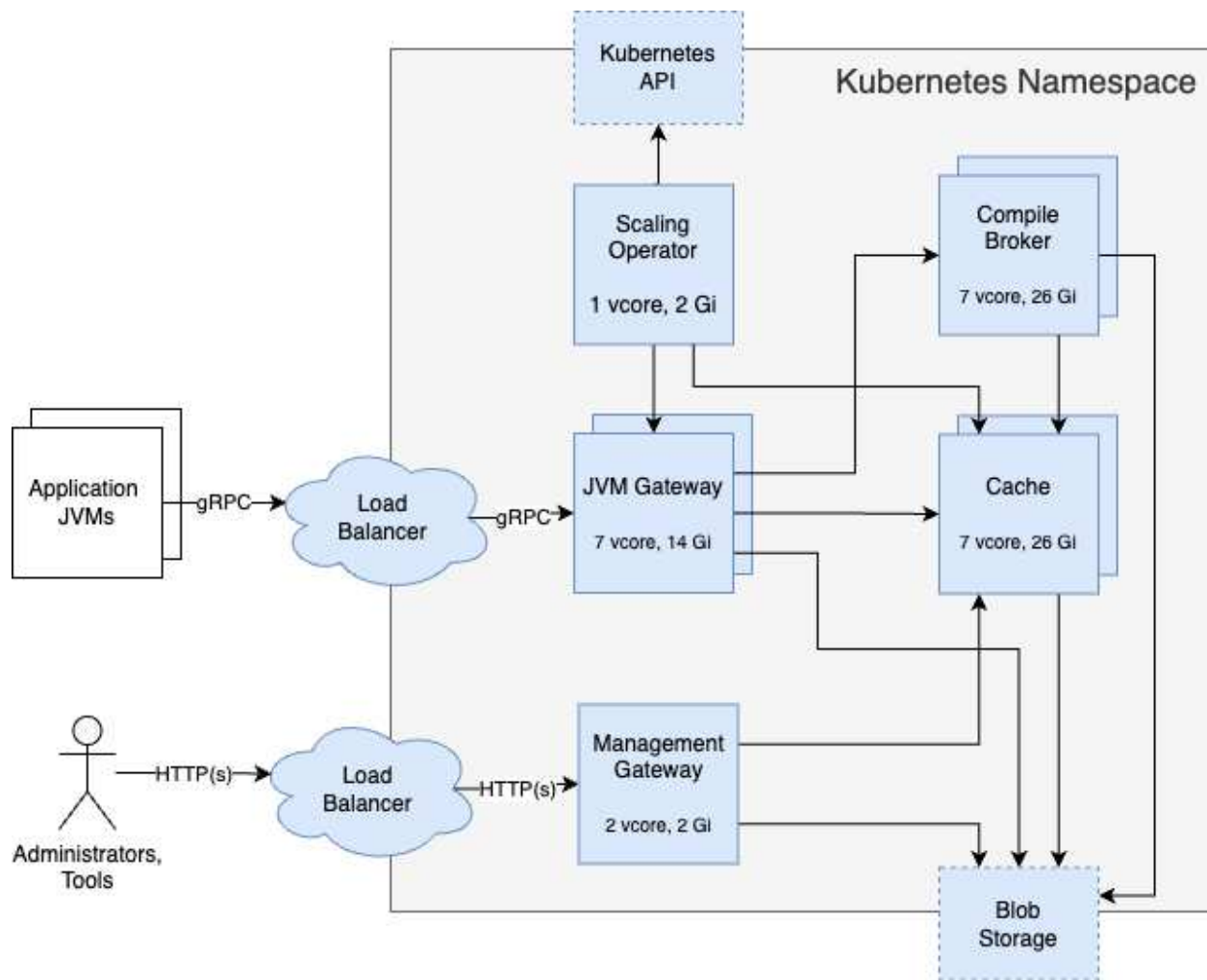
Architecture Overview

You can deploy Optimizer Hub in two configurations:

- Without Cloud Native Compiler: In this configuration Optimizer Hub does not need to scale. There are no Compiler Broker pods, and the Gateway and Cache pods remain at the fixed number you specify when installing.



- With Cloud Native Compiler: In this configuration, Optimizer Hub does need to scale. The Operator component handles all scaling of Compile Broker, Gateway, and Cache pods.



System Components

Optimizer Hub consists of the following Kubernetes-deployed components:

Component	Purpose	Default Replicas
Gateway	Entry point for JVM connections (gRPC on port 50051)	1 (auto-scales)
Compile Broker	Performs actual JIT compilation work	1 (auto-scales)
Cache	Stores compiled code for reuse	2
Management Gateway	Provides REST API for management tasks (port 8080)	1
Operator	Manages autoscaling based on compilation workload	1

In addition to these components, you must also configure the following external components:

- Load balancer to route traffic from JVMs to the Optimizer Hub services.
 - The load balancer must be an application-level load balancer, i.e., it must understand the gRPC protocol (which is built on top of HTTP/2) and load balance each gRPC request independently.
 - The load balancer may not limit the duration of gRPC calls. Optimizer Hub uses streaming gRPC calls, which can last for hours, days, or how long the VM stays alive. These long-lived calls may not be considered as an error and may not be killed.
- Grafana and Prometheus for collecting and displaying monitoring information for the service.

High Availability Architecture

Within a single cluster, Optimizer Hub has built-in redundancy at multiple levels and automatic workload redistribution on node failure.

For full high availability:

- Configure at least two Optimizer Hub clusters, typically one per availability zone.
- Front them with a global load balancer that directs traffic from JVMs to their closest instance.
- Use the provided health-check endpoints to determine whether a cluster is healthy, and route traffic to one of the secondary clusters if it is not.
- Configure cross-region ReadyNow profile synchronization to ensure every cluster has all the profiles needed to serve as a backup. Cloud Native Compiler artifacts do not need to be synchronized as they can be recreated at runtime with minimal disruption to connected JVMs.

Prerequisites and Dependencies

Required Software Versions

For Optimizer Hub:

- Kubernetes 1.21+
 - Self-managed Kubernetes clusters
 - Amazon Elastic Kubernetes Service (EKS)
 - Google Kubernetes Engine (GKE)
 - Microsoft Azure Kubernetes Service (AKS)
 - MicroK8s and Minikube (evaluation only)
- Helm v3.8.0 or newer
- Grafana 10.0 or newer (for monitoring dashboards)

For client machines:

- Azul Zing Builds of OpenJDK 24.02 or newer

Supported Platforms

- x64 architecture (JVMs can run on Arch64 hardware, but all Optimizer Hub pods must run on x64 nodes)
- Reserved or on-demand compute nodes. Running on spot instances, managed container services like AWS Fargate, or serverless systems like AWS Lambdas is not supported.
- Preferred node types:

Platform	Recommended Type	CPU/RAM	Comment
AWS	<code>m6.2xlarge</code> / <code>m7.2xlarge</code>	8 vCPU / 32 GB	Baseline per Azul documentation

Platform	Recommended Type	CPU/RAM	Comment
GCP	<code>n2-standard-8</code> / <code>c3-standard-8</code>	8 vCPU / 32 GB	Balanced, modern CPUs
Azure	<code>D8s_v5</code> / <code>D8plsv5</code>	8 vCPU / 32 GB	Best general-purpose match

Network Requirements

- JVMs require unauthenticated access to Optimizer Hub gateway
- Port 50051 (gRPC) for JVM connections
- Port 8080 (HTTP) for management API and engine uploads
- Network latency between JVM and OptHub should be <10ms for optimal performance
- The load balancer may not limit the duration of gRPC calls. Optimizer Hub uses streaming gRPC calls, which can last for hours, days, or how long the VM stays alive. These long-lived calls may not be considered as an error and may not be killed.

Storage Requirements

Depending on the environment on which you deploy Optimizer Hub, different configurations are documented for the storage:

- [AWS S3](#)
- [Azure Blob Storage](#)
- [Google Cloud Storage](#)
- [S3-compatible](#)
- MinIO (for evaluation only in [MicroK8s](#) and [Minikube](#))

Optimizer Hub uses your cloud provider's blob storage as the main persistence mechanism. Artifacts persisted to blob storage are:

- ReadyNow Orchestrator: saved profile logs.

- Code Cache: previously performed compilations.

Optimizer Hub includes auto cleaner mechanisms to clean up unused data. Cleanup for ReadyNow profile logs and Code Cache entries are configured separately.

Required Roles and Permissions

Depending on the environment on which you deploy Optimizer Hub, different configurations are documented related to roles and permissions for storage and/or other components:

- AWS
 - [Kubernetes Nodes and Permissions](#)
 - [AWS Service Accounts](#)
- Google Cloud:
 - [IAM Policy Update](#)
 - [IAM Policy Binding](#)

The Optimizer Hub Operator functions as the controller for the instance and runs with **namespace-scoped permissions**. It requires a `Role` and `RoleBinding` within its specific namespace to manage resources such as Deployments, Services, and Secrets. It does not require `ClusterRole` (cluster-wide) privileges, ensuring it operates in isolation without affecting other tenants in the cluster.

Using Externally Defined Secrets

Secrets can be externally defined to allow you to manage Kubernetes secrets independent of the Optimizer Hub configuration.

You can define the following secrets by overriding the following default settings in your `values-override.yaml` file:

- ``blobstorage.s3.accesskey``
- `blobstorage.s3.secretkey`

- `azure.connectionString`
- `azure.sasToken`

Usage:

- If you keep the default values, the Optimizer Hub helm chart will define its own Kubernetes secret objects and use these.
- Or you use your existing secrets by:
 - Defining the name of your Kubernetes secret object with `existingSecret`.
 - Optionally you can define the name of the keys in your Kubernetes secret object with, e.g. `accessKeySecretKey`, in case you want something different than what Optimizer Hub expects by default.

More info is [available in the documentation](#).

Installation Procedures

Prepare a Kubernetes Cluster

You can install Optimizer Hub on any Kubernetes cluster. Preparing a cluster for Optimizer Hub is not documented in this document, but you can find detailed information on the Optimizer Hub documentation:

- Managed cloud Kubernetes services such as:
 - [Amazon Web Services Elastic Kubernetes Service \(EKS\)](#)
 - [Google Kubernetes Engine](#)
 - [Microsoft Azure Managed Kubernetes Service](#)
- A single-node minikube cluster:
 - [Installing Optimizer Hub on MicroK8s](#)
 - [Installing Optimizer Hub on Minikube](#)

Standard Installation

The following steps outline the installation of a full Optimizer Hub instance, including Cloud Native Compiler (CNC) and ReadyNow Orchestrator (RNO).

This procedure uses the bundled `gw-proxy` component as a reference implementation for routing traffic. For production deployments, especially those requiring [High Availability \(HA\)](#), you should replace `gw-proxy` with your own Load Balancer or Ingress controller.

The following instructions are applicable for the latest version.

Step 1: Prepare Helm Repository

```
# Add Azul Helm repository
helm repo add opthub-helm https://azulsystems.github.io/opthub-helm-charts/
helm repo update
```

Step 2: Create Namespace

```
kubectl create namespace my-opthub
```

Step 3: Create Configuration File

The following represents the minimal configuration required to deploy Optimizer Hub. This example accepts the default settings for most parameters but is specifically configured to use AWS S3 for storage.

Create `values-override.yaml` with the following settings. This same file can be extended to override all the default settings and adjust Optimizer Hub to your use case and environment.

```
# Custom cluster domain (if applicable)
clusterName: "example.org"

# Storage configuration
storage:
  blobStorageService: s3
  s3:
    commonBucket: opthub-storage0

# Storage authentication
deployment:
  serviceAccount:
    annotations:
      eks.amazonaws.com/role-arn: arn:aws:iam::<...>:role/opthub-s3-role
```

Step 3: Install Optimizer Hub

Install using Helm, passing in the `values-override.yaml` file.

```
# Install full Optimizer Hub (CNC + ReadyNow Orchestrator)
helm install opthub opthub-helm/azul-opthub \
  -n my-opthub \
  -f values-override.yaml
```

Step 4: Install Monitoring

Monitoring is a critical requirement for production environments. It provides essential visibility into the health and performance of the Optimizer Hub components, which is necessary for effective operation and troubleshooting.

For detailed instructions on installing and configuring the monitoring stack, please refer to [Configuring Prometheus and Grafana](#) or "Monitoring and Alerting" in this Runbook.

Step 5: Verify Installation

```
# Check pod status
kubectl get pods -n my-opthub

# Expected output:
# NAME                                READY   STATUS    RESTARTS   AGE
# gateway-xxxxx                      1/1     Running   0           2m
# compile-broker-xxxxx               1/1     Running   0           2m
# cache-xxxxx                        1/1     Running   0           2m
# mgmt-gateway-xxxxx                 1/1     Running   0           2m
# operator-xxxxx                     1/1     Running   0           2m

# Check services
kubectl get svc -n my-opthub
```

ReadyNow Orchestrator Only Installation

For deployments requiring only profile orchestration without compilation:

```
helm install opthub opthub-helm/azul-opthub \
  -n my-opthub \
  -f values-override.yaml \
  -f values-disable-compiler.yaml
```

Configuration Management

Configuring Scaling

Understanding Optimizer Hub Sizing

The sizing strategy for Optimizer Hub depends on your deployment mode:

- **Only ReadyNow Orchestrator (RNO):** This configuration has a fixed size and the components do not auto-scale, as the workload is predictable and lightweight. You manually define the number of replicas.
- **All services:** When using both RNO and Cloud Native Compiler (CNC), automatic sizing is used, managed by the Optimizer Hub Operator and the `simpleSizing` configuration.

NOTE

You should not configure custom auto-scaling rules, as these conflict with the Operator's `simpleSizing` logic. Doing so will lead to instability.

If you need to test CNC against a statically scaled cluster (e.g., during a pilot or for debugging), you must explicitly disable `simpleSizing` and manually set the replica counts.

Manual Scaling

When Optimizer Hub is configured on RNO-only mode (using `values-disable-compiler.yaml`, see [Configuring the Active Optimizer Hub Services](#)), it doesn't need to scale. The predefined sizing will be able to handle full RNO functionality.

When using the full configuration with RNO and Cloud Native Compiler (CNC), Optimizer Hub is configured with simple sizing enabled and one Gateway service. Depending on your use case, you can increase this by overriding the default values in your `values-override.yaml` file.

- For systems using Cloud Native Compiler (= all services enabled) that can scale up and down:

- With simple sizing enabled, 7 vCores extra are required for an additional Gateway pod:

```
gateway:
  autoscaler:
    min: 2
  simpleSizing:
    vCores: 46
    minVCores: 46
```

- With simple sizing disabled:

```
gateway:
  autoscaler:
    min: 2
```

- For systems with ReadyNow Orchestrator only or using a statically scaled CNC config for testing, one gateway and one cache suffice. If needed, you can increase the number of gateways to 2 for redundancy:

```
gateway:
  replicas: 2
```

Autoscaling Configuration

When using the full configuration with ReadyNow Orchestrator and Cloud Native Compiler, the Optimizer Hub operator automatically scales based on compilation workload, taking the configured number of vCores into account.

The following default values can be overriding in your `values-override.yaml` file:

```
simpleSizing:
  vCores: 39
  minVCores: 39
  maxVCores: 113
```

The minimum and maximum number of vCores is used by the Optimizer Hub service to adjust the sizing of the instance to try to meet your

`timeToClearOptimizationBacklog` limit for all the JVMs that request compilations.

Configuring SSL/TLS

The recommended setup is to have a load balancer or service mesh in front of the Optimizer Hub service. This will then be used as the connection point for the JVMs to interact with Optimizer Hub and include the SSL configuration.

In cases where such a load balancer or service mesh is not available, for instance, for development and evaluation, Optimizer Hub itself can be configured to run with or without SSL authentication. Of course, it is highly recommended that you run your production Optimizer Hub with SSL authentication.

Enable SSL in Optimizer Hub

Add to `values-override.yaml`:

```
gateway:
  tls:
    enabled: true
    certificateSecret: ophub-tls-secret
```

Create TLS secret:

```
kubectl create secret tls ophub-tls-secret \
  -n my-ophub \
  --cert=path/to/tls.crt \
  --key=path/to/tls.key
```

Configure JVM Clients

For SSL-enabled Optimizer Hub (default):

```
# Ensure certificate is trusted on client machine
sudo openssl x509 -in cert.pem -inform PEM -out /usr/local/share/ca-
certificates/cert.crt
sudo update-ca-certificates

# Connect with SSL (default)
java -XX:OptHubHost=ophub.example.com:50051 \
  -XX:+EnableRNO \
  -jar my-app.jar
```

For non-SSL Optimizer Hub:

```
java -XX:OptHubHost=opthub.example.com:50051 \
-XX:-OptHubUseSSL \
-XX:+EnableRNO \
-jar my-app.jar
```

Storage Cleanup

Optimizer Hub uses your cloud provider's blob storage as the main persistence mechanism for saved profile logs from ReadyNow Orchestrator, and previously performed compilations from the Code Cache. Optimizer Hub includes auto cleaner mechanisms to clean up unused data. Cleanup for ReadyNow profile logs and Code Cache entries are configured separately.

Code Cache Cleanup

You specify the target size for the blob storage that Optimizer Hub should not exceed, as well as how often Optimizer Hub should check if cleaning is necessary.

The following are the default values, which you can modify in your `values-override.yaml` file:

```
codeCache:
  cleaner:
    enabled: true
    targetSize: "107374182400" # 100GiB, use quotes for large numbers
    interval: PT2H # 2 hours
```

ReadyNow Profile Log Cleanup

ReadyNow Orchestrator performs automatic cleanup of unused profile logs in order to fit collected data in the configured storage. When the data size in your storage exceeds a threshold, ReadyNow Orchestrator deletes old profile logs, thus guaranteeing that a promoted profile log is available for all profile names.

The following are the default values, which you can modify in your `values-override.yaml` file:

```
readyNowOrchestrator:
  cleaner:
    enabled: true
    externalPersistentStorageSoftLimit: "10Gi"
```

```
targetSize: 0 # use only to override auto-settings  
warningSize: 0 # use only to override auto-settings  
keepUnrequestedProfileNamesFor: 0  
keepDebugOnlyGenerationProfilesFor: "P7D"
```

You can configure ReadyNow Orchestrator to delete unused profile names completely after a given duration using the `keepUnrequestedProfileNamesFor` property in your `values-override.yaml`. By default, this value is `0`, meaning unused profiles are not cleaned up completely. For example, to keep unused profiles for 5 days, use the following:

```
readyNowOrchestrator.cleaner.keepUnrequestedProfileNamesFor=P5D
```

If your storage fills up completely, JVMs attempting to write to ReadyNow Orchestrator receive an error.

Operational Procedures

Daily Operations

Checking the System Health

```
# Check pod status
kubectl get pods -n my-opthub

# Check health endpoints
MGMT_GW=$(kubectl get svc mgmt-gateway -n my-opthub -o jsonpath=
' {.spec.clusterIP} ')
curl http://${MGMT_GW}:8080/q/health

# Check readiness for traffic routing
GATEWAY=$(kubectl get svc gateway -n my-opthub -o jsonpath=' {.spec.clusterIP} ')
curl http://${GATEWAY}:8080/api/opthub-health/healthy
```

Checking the Connected JVMs

```
# View connected JVM count via Grafana or API
curl http://${MGMT_GW}:8080/api/connected-jvms

# View profile names in use
curl http://${MGMT_GW}:8080/api/currentlyConnectedProfileNames
```

Upgrade Procedures

Pre-Upgrade Checklist

1. ☐ Review release notes for breaking changes.
2. ☐ Note current version: `helm list -n my-opthub`.
3. ☐ Make a backup of your configuration files, like `values-override.yaml` and the externally defined secrets.
4. ☐ Schedule the maintenance window if needed.
5. ☐ Notify JVM application teams.

Standard Upgrade

Optimizer Hub has releases for separate versions which are still maintained. Therefore, the "latest version" may not be the one you want to upgrade to, and it's recommended to always specify the version you want to install.

```
# Update Helm repository
helm repo update

# Check available versions
helm search repo ophub-helm/azul-ophub --versions

# Upgrade to a specific version
helm upgrade ophub ophub-helm/azul-ophub \
  -n my-ophub \
  -f values-override.yaml \
  --version 25.08.1

# Monitor upgrade progress
kubectl rollout status deployment/gateway -n my-ophub
kubectl rollout status deployment/compile-broker -n my-ophub
```

Disaster Recovery Procedure

In case of a complete cluster loss, follow this procedure to start a new one:

1. Deploy new Kubernetes cluster.
2. Create namespace: `kubectl create namespace my-ophub`.
3. Reuse your configuration files like `values-override.yaml` and the externally defined secrets.
4. Reinstall Optimizer Hub with backed-up values:

```
helm install ophub ophub-helm/azul-ophub \
  -n my-ophub \
  -f values-override.yaml
```

5. Verify blob storage connectivity.
6. Update DNS/load balancer to point to the new cluster.
7. Verify JVM connections: Check connected JVMs count.

Monitoring and Alerting

The Optimizer Hub components are already configured to expose key metrics for scraping by Prometheus. But to be able to monitor this info in a Grafana dashboard, some additional configuration is required.

Reviewing Logs

All Optimizer Hub components follow cloud-native best practices by writing application logs directly to standard output (`stdout`) and standard error (`stderr`). This design eliminates the need for complex log file rotation or management within the pods themselves.

For production environments, you should configure your Kubernetes cluster to capture these streams using your organization's standard log aggregation solution (e.g., Datadog, Splunk, Fluentd/Fluent Bit, or ELK Stack).

To view logs for immediate troubleshooting without an external aggregator:

```
# View logs for a specific component (e.g., gateway)
kubectl logs -l app=gateway -n my-opthub

# Follow logs in real-time
kubectl logs -l app=compile-broker -n my-opthub -f
```

NOTE

The default log retention policy in Kubernetes is often limited (e.g., 10MB or rotated quickly). Relying solely on `kubectl logs` is insufficient for historical analysis or auditing.

Extracting Logs with Loki (if configured)

```
# Get last 24 hours of compile-broker logs
logcli query --since 24h --forward --limit=10000 \
  '{namespace="my-opthub",app="compile-broker"}'

# Get logs from specific pod
logcli query --since 24h --forward --limit=10000 \
  '{namespace="my-opthub",pod="compile-broker-abc123"}'
```

Using Prometheus and Grafana

In your production systems, you likely want to use your existing Prometheus and Grafana instances to monitor Optimizer Hub. If you are just evaluating Optimizer Hub, you may want to install a separate instance of Prometheus and Grafana to just monitor your test instance of Optimizer Hub.

Configuring Prometheus

Optimizer Hub components expose their metrics on HTTP endpoints in a format compatible with Prometheus. Annotations are in place in the Helm chart with the details of the endpoint for every component. For example:

```
annotations:
  prometheus.io/scrape: "true"
  prometheus.io/port: "8080"
  prometheus.io/path: "/q/metrics"
```

The following snippet is an example for the Prometheus configuration to scrape the metrics based on the above annotations:

```
# Example scrape config for pods
#
# The relabeling allows the actual pod scrape endpoint to be configured via the
# following annotations:
#
# * `prometheus.io/scrape`: Only scrape pods that have a value of `true`
# * `prometheus.io/path`: If the metrics path is not `/metrics` override this.
# * `prometheus.io/port`: Scrape the pod on the indicated port instead of the
# pod's declared ports (default is a port-free target if none are declared).
- job_name: 'kubernetes-pods'
  kubernetes_sd_configs:
    - role: pod

  relabel_configs:
    - source_labels: [__meta_kubernetes_pod_annotation_prometheus_io_scrape]
      action: keep
      regex: true
    - source_labels: [__meta_kubernetes_pod_annotation_prometheus_io_path]
      action: replace
      target_label: __metrics_path__
      regex: (.+)
    - source_labels: [__address__,
      __meta_kubernetes_pod_annotation_prometheus_io_port]
      action: replace
      regex: ([^:]+)(?::\d+)?;(\d+)
      replacement: $1:$2
      target_label: __address__
  # mapping of labels, this handles the `app` label
  # mapping of labels, this handles the `app` label
```



```

- action: labelmap
  regex: __meta_kubernetes_pod_label_(.+)
- source_labels: [__meta_kubernetes_namespace]
  action: replace
  target_label: kubernetes_namespace
- source_labels: [__meta_kubernetes_pod_name]
  action: replace
  target_label: kubernetes_pod_name
metric_relabel_configs:
- source_labels:
  - namespace
  action: replace
  regex: (.+)
  target_label: kubernetes_namespace

```

Configuring Grafana

Once Prometheus is available and collecting data from the Optimizer Hub Components, a dashboard can be added:

1. Find the dashboard-file in the latest [opthub-install.zip](#): `opthub_dashboard.json`
2. Import into Grafana
3. Configure Prometheus datasource
4. The dashboard includes:
 - Overview (running components, connected JVMs, compilations)
 - Compilation Queues
 - Compilation Performance (TTCOB metric)
 - Time to Clear Optimization Backlog
 - Resource Usage
 - Profile Statistics
 - Error Rates

This dashboard expects the following labels to be attached to all application metrics, referring to the Prometheus configuration above:

- `cluster_id`: The identifier of the Kubernetes cluster on which Optimizer Hub is installed. This allows you to switch between Optimizer Hub instances in different

clusters.

- `kubernetes_namespace`: The Kubernetes namespace on which Optimizer Hub is installed. This setting allows you to switch between Optimizer Hub instances in different namespaces of the same cluster.
- `kubernetes_pod_name`: The Kubernetes pod name.
- `app`: The value of the `app` label on the pod, which is provided by the `labelmap` action from the example Prometheus configuration mentioned below.

You need to manually edit the dashboard file if these labels are named differently in your environment.

The dashboard also relies on some infrastructure metrics from [Kubernetes](#) and [cAdvisor](#), such as `kube_pod_container_resource_requests` and `container_cpu_usage_seconds_total`.

Metrics Endpoints

```
# Gateway proxy metrics (note different endpoint)
curl http://<gateway-pod-ip>:8080/stats/prometheus

# All other components
curl http://<pod-ip>:8080/q/metrics

# Management gateway API
curl http://${MGMT_GW}:8080/api/metrics
```

Legal Notice

© 2005–2025, Azul Systems, Incorporated, 375 Moffett Park Drive, Suite 115, Sunnyvale, CA 94089. All rights reserved.

Products and specifications discussed in this document may reflect future versions and are subject to change without notice. Azul Systems assumes no responsibility or liability for any errors or inaccuracies that may appear in the informational content contained in this guide.

No part of this document may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, recording, or otherwise, without the prior written permission of Azul Systems. Please note that the content in this document is protected under copyright law even if it is not distributed with software that includes an end user license agreement.

Azul Systems, Azul Zulu, and the Azul logo are trademarks or registered trademarks of Azul Systems, Inc. Linux is a registered trademark of Linus Torvalds. Java is a registered trademark of Oracle Corporation. Microsoft and Windows are registered trademarks of Microsoft Corporation. Other marks are the property of their respective owners and are used here only for identification purposes.