azul

Optimizer Hub Documentation

Table of Contents

About Optimizer Hub

Documentation for Optimizer Hub, version 1.9.5

Optimizer Hub is a component of Azul Platform Prime that makes your Java programs start fast and stay fast. It consists of two services:

- Cloud Native Compiler: Provides a server-side optimization solution that offloads JIT compilation to separate and dedicated service resources, providing more processing power to JIT compilation while freeing your client JVMs from the burden of doing JIT compilation locally.

- ReadyNow Orchestrator: Records and serves ReadyNow profiles. This greatly simplifies the operational use of the ReadyNow, and removes the need to configure any local storage for writing the profile. ReadyNow Orchestrator can record multiple profile candidates from multiple JVMs and promote the best recorded profile.

NOTE	You can run both services with the default installation, or ReadyNow Orchestrator only, depending on your use case. Check the Architecture Overview to understand the components within the Optimizer Hub system.

Interaction Between Optimizer Hub and JVMs

1. ReadyNow in the JVM asks ReadyNow Orchestrator in Optimizer Hub for a profile.

2. In the JVM, ReadyNow instructs Falcon what to compile based on the profile.

3. ReadyNow in the JVM sends back a new version of the profile to ReadyNow Orchestrator in Optimizer Hub.

4. Falcon in the JVM asks the Cloud Native Compiler in Optimizer Hub to compile the code (optional).

5. Cloud Native Compiler in Optimizer Hub sends the compiled code back to Falcon in the JVM (optional).

 https://www.azul.com/wp-content/uploads/AZL106-ReadyNow-Orchestrator-Video-

[AW.mp4](AW.mp4) *(video)*

About Cloud Native Compiler

Cloud Native Compiler is a component of Optimizer Hub that provides a server-side optimization solution that offloads JIT compilation to separate and dedicated service resources, providing more processing power to JIT compilation while freeing your client JVMs from the burden of doing JIT compilation locally.

JIT Optimization

JIT optimization provides a multitude of benefits, including the ability to use speculative optimizations that lead to faster eventual code. However, traditional on-JVM JIT compilers must share the JVMs local CPU resources and compete with the application logic in using that capacity. This presents several challenges:

- Optimization limitations:

  - The JIT compiler is limited in resources. Resulting optimizations take time to arrive at, and benefits are limited by the practical amount of time that applications can wait for optimization and warm-up to complete.

  - The JIT compiler is limited in how aggressively it can afford to optimize code. The resulting optimizations are not as fast as they could be if the optimizer had more resources available.

- Application performance limitations during warm-up:

  - Optimization takes time to complete, and application code runs significantly slower and less efficiently until the JIT compilers optimize it.

  - JIT compilation work competes with the application for resources. Not all CPU and memory resources are devoted to application threads.

- Resource allocation and utilization:

  - Resources (CPU and memory capacity) used for JIT optimization are only needed and utilized during warm-up, which is a fraction of the overall lifetime of each Java process. Instances must reserve (and customers pay for) these under-utilized

resources for the duration of the run of each application instance.

Falcon JIT

Azul Zulu Prime Builds of OpenJDK replace OpenJDK's C2 JIT compiler with the Falcon JIT compiler. The Falcon JIT compiler can run different levels of optimizations, and its upper tier of optimizations produces optimized code that can run significantly faster than code produced by the OpenJDK C2 compiler.

Using more aggressive optimization levels requires more resources, and when using JVM-local JIT compilers for optimization, resource tradeoffs can often lead to a choice of lowering optimization levels in favor of improved warmup times. Cloud Native Compiler eliminates these tradeoffs by removing JIT compilation work from individual JVMs, and shifting the work of the Falcon JIT compiler to a separate shared service. This shift of work and associated resources allows the Cloud Native Compiler to apply even the most aggressive Falcon JIT optimization levels without disrupting individual JVM behavior. The Cloud Native Compiler can bring to bear practically unlimited Falcon JIT compilation resources when a JVM needs them, and later scale those resources down when they are unused and unneeded. This results in JVMs that can consistently serve higher amounts of traffic in smaller footprint.

About ReadyNow Orchestrator

ReadyNow Orchestrator is a component of Optimizer Hub that records and serves ReadyNow profiles. This greatly simplifies the operational use of ReadyNow when using in large fleets of containerized environments.

- **Centralized Profile Storage**: You can configure your runtimes, using JVM command-line parameters, to use ReadyNow Orchestrator for profile recording. ReadyNow Orchestrator then records profiles from a meaningful subset of your JVMs, saving your profiles either on Optimizer Hub's built-in storage or on your S3-like object storage.

- **Profile Training and Optimization**: ReadyNow Orchestrator also takes care of recording multiple training generations of your profile to produce the best possible

optimization profile. ReadyNow Orchestrator then picks the best profile out of all the possible candidates and streams it to any new JVM that is configured to request that profile.

- **Providing Profiles to JVMs**: ReadyNow Orchestrator automatically serves the best profile to newly started JVMs.

Key Strengths of ReadyNow Orchestrator

- No change to your deployment profile to manually record and distribute your ReadyNow profiles. Everything is configured with a few JVM command-line parameters.

- ReadyNow Orchestrator monitors your entire fleed of JVMs and picks the best optimization profile rather than just using the profile produced by one JVM.

- Easy streaming of profiles into and out of containers, removing the need to configure persistent storage or bake profiles into images each time you build a new image.

Optimizer Hub Architecture Overview

Optimizer Hub is shipped as a Helm chart and a set of docker images to be deployed into a Kubernetes cluster. The Helm chart deploys different components based on the use case.

Architecture Overview

Full Installation

In a full installation, all Optimizer Hub components are available and gateway, compile-broker, and cache are scaled when needed.

The Management Gateway component is optional, as it depends on your use case. See management-gateway-parameters for more info.

ReadyNow Orchestrator Only

When only ReadyNow Orchestrator is needed, a reduced set of the Optimizer Hub components is deployed in the Kubernetes cluster.

Deployment Overview

With the default AWS setup (`values-aws.yaml`), the setup is divided into three node types (four if you also want to use the optional monitoring stack). Each node has a `role` label used to set the affinity for the nodes. If you set up your cluster on AWS EKS using the Azul-provided "cluster config file", nodes are created with these labels.

NOTE	Make sure that the instances on which you run your Optimizer Hub on have enough CPU to handle your requests. For example, for AWS m5.2xlarge instances can be used, and on Google Cloud Platform c2-standard-8 instances.

The nodes in a Optimizer Hub instance are as follows:

- Compile Broker - Performs JIT compilations.

  - AWS node type: `role=opthubserver`

  - System Requirements: CPU 8, RAM 32GB, HDD 100GB

- Cache - Stores information about the JVM that the compiler needs to perform compilations.

  - AWS node type: `role=opthubcache`

  - System Requirements: CPU 8, RAM 32GB, HDD 100GB

  - There is one pod per Cache node. To scale up, create more replicas.

- Infrastructure - Provides supporting functionality.

  - AWS node type: `role=opthubinfra`

  - System Requirements: CPU 8, RAM 32GB, HDD 100GB. Make sure the disk connection is fast (use SSD) and that the storage volume is persistent between runs.

  - The pods included in this node are:

    - db

- gateway

- storage

- Infrastructure - Non-Optimizer Hub supporting functionality, such as monitoring.

  - AWS node type: `role=infra`

  - System Requirements: CPU 8, RAM 32GB, HDD 100GB.

  - Pods included in this node:

    - grafana

    - prometheus

Optimizer Hub Release Notes

Optimizer Hub 1.9.5

Release Date: November 5, 2024

New Features

- Includes bug fixes for Optimizer Hub 1.9.4.

- Faster down-scaling to release resources that are no longer needed.

Default Configuration Changes

- The stabilization window in the scaling configuration changed from 1 to 2 minutes.

- On the first connection to Optimizer Hub, each JVM was getting a certain about of compile-broker capacity allocated. This could cause a high number of compile-brokers when many JVMs connect but don't have enough compilations. This feature is now disabled by default.

Optimizer Hub 1.9.4

Release Date: September 16, 2024

New Features

- Includes bug fixes for Optimizer Hub 1.9.3.

- When using Optimizer Hub with AWS, you can now use an K8S ServiceAccount for S3 permissions. For more info, check out the documentation at service-accounts.

- Profile download errors, caused by any reason, are now reflected in metrics and visible in the Grafana dashboard.

Bug Fixes

- Improved stability of the readiness probes for the Optimizer Hub Components.

- The limit of incoming connections for single instances of the gateway (Envoy proxy) is increased from 1024 to 3072.

- You can now configure the AWS region of S3 with the Helm value `storage.s3.region`.

- Implemented stricter data consistency validation during the upload of RNO profiles to prevent BlobNotFound errors later.

Known Issue

- Old Prime JVMs (pre-23.08), using an earlier protocol version, can send profile chunks in an incorrect order. This can lead to some chunks getting lost, e.g., due to reconnections.

- With newer Prime JVMs, using the latest protocol, this same issue has been noticed very rarely, and research is ongoing.

Optimizer Hub 1.9.3

Release Date: August 12, 2024

New Features

- Includes bug fixes for Optimizer Hub 1.9.2.

- Because of configuration changes, `--set version` is no longer supported during installation using Helm.

- You can now also specify Service labels, as described in the installing-optimizer-hub

Grafana Dashboard Update

A new version of the Grafana dashboard is included in opthub-install.zip

Bug Fixes

- Improved cleanup policy for profiles written with `continueRecordingOnPromotion` to avoid profiles to grow too much.

- Fixed a bug where MariaDB deployed with an empty password, potentially allowing unauthorized root connections. MariaDB is now deployed with a randomly set password.

Known Issue

- In some configurations with cross-region syncing, the sync task can get stuck because of incorrect configurations.

Optimizer Hub 1.9.2

Release Date: April 30, 2024

New Features

API improvements

The API endpoint `/rno/names` is extended with:

- Extra flag `cncEnabled` in the returned result, indicating that the creator of a profile used or didn't use CNC.

- Optional request filter to define a date range.

See api-methods for more info.

Grafana Dashboard Update

A new version of the Grafana dashboard is included in opthub-install.zip

Known Issue

Profile Sync Running Indefinitely

The profile synchronization task may hang indefinitely if it is erroneously configured with an gRPC endpoint URL, instead of an HTTP endpoint URL in `synchronization.peers`. Please review your configuration in case you encounter such a hang.

Optimizer Hub 1.9.1

Release Date: April 12, 2024

New Features

Grafana Dashboard Update

A new version of the Grafana dashboard is included in opthub-install.zip

Configurable Minimal Client Version

Optimizer Hub can now be configured to only allow clients with a specific minimal version of Azul Zing Builds of OpenJDK to connect to and use Optimizer Hub. By default, all versions are allowed. To limit, for example, to 24.02.1+, add the following setting to your `values-override.yaml`:

```
compilations:
  minVmVersionForCNCCompilation: "24.2.1.0"
```

Increased Number of Concurrent Recordings

The default value of `readyNowOrchestrator.producers.maxConcurrentRecordings` has been increased from 5 to 10, ensuring that enough long-lived producers are detected over short-lived ones.

Continuous Recording

With the new flag `readyNowOrchestrator.producers.continueRecordingOnPromotion`, you can define if profiles must still be recorded after the maxGeneration has been reached.

You can use this flag for debugging purposes. See readynow-orchestrator-defaults for more info.

Optimizer Hub 1.9.0

Release Date: February 1, 2024

New Features

Cross-Region Synchronization of ReadyNow Orchestrator Profiles

A new feature in ReadyNow Orchestrator allows you to synchronize profile names between Optimizer Hub instances in different regions so that each instance contains at least one promoted profile for each profile name.

See cross-region-sync-parameters for configuration options.

Database Changes

Optimizer Hub 1.9 includes an update to the Code Cache database schema. After upgrading, Optimizer Hub dumps old Code Cache data and recreates it the next time you run your application.

New Location of REST APIs and ReadyNow Profile Cleaner

The REST APIs and ReadyNow Profile Cleaner moved to the new Management Gateway component, and the APIs are now exposed on a different address. The Management Gateway is disabled by default, see management-gateway-parameters as this component is not required in all use-cases.

Prioritization of Profile Generations

ReadyNow Orchestrator allows you to set different minimum size and recording durations for different generations of your profiles. Often you want to promote the first generation of your profile as quickly as possible so new JVMs are not starting with nothing, but you want your second generation to record for a longer time before promotion, so it is more complete.

New configuration settings: `minProfileSize`, `minProfileDuration`,

`minProfileSizePerGeneration`, and `minProfileDurationPerGeneration`.
Check readynow-orchestrator-defaults for more info.

Grafana Dashboard

The Grafana Dashboard has been updated with more information for greater visibility into Optimizer Hub performance.

Support for Zing Running on ARM

Optimizer Hub now supports connections from Zing JVMs running on both x86 and ARM 64-bit machines. Optimizer Hub itself still needs to run on x86 only.

Bug Fixes

The message "Error occurred while executing task for trigger IntervalTrigger" may be seen during initialization. This resolves automatically after some time and works as expected.

Optimizer Hub 1.8.2

Release Date: December 19, 2023

New Features

- Fixes an issue in 1.8.1 where the cache component is not able to scale up.

- Fixes an issue that caused unexpected HTTP/1.x requests for `GET /q/metrics` to be reported in the logging.

Optimizer Hub 1.8.1

Release Date: December 6, 2023

New Features

Includes bug fixes for Optimizer Hub 1.8.0.

Known Issues

The message "Error occurred while executing task for trigger IntervalTrigger" may be seen during initialization. This resolves automatically after some time and work as

expected.

Optimizer Hub 1.8.0

Release Date: September 12, 2023

As Cloud Native Compiler expands its scope to offer more functionality than just offloading compilations, it is time to rebrand the offering to better reflect what it does. Starting with release 1.8, we are using the following naming:

- Optimizer Hub (was Cloud Native Compiler) - The name of the overall component that you install on your Kubernetes cluster.

  - Cloud Native Compiler (was Compiler Service) - The feature that performs the compilation on Optimizer Hub.

  - ReadyNow Orchestrator (was Profile Log Service) - The feature that records and serves ReadyNow profiles to JVMs.

In Optimizer Hub 1.8, all major artifacts and command line switches use the updated branding. This includes, but is not limited to:

- Command-line JVM options to configure Cloud Native Compiler and readynow-orchestrator-jvm-options.

- Helm repository locations, names, and parameter names: github.com/AzulSystems/opthub-helm-charts.

- REST API URLs.

If you are using release 1.7 and earlier, all of the previous spellings of artifacts still work. Additionally, all of the pre-1.8 command-line arguments continue to work for a period of one year from the release of 1.8.

New Features

- Monitoring with Prometheus and Grafana is no longer included in the Optimizer Hub Helm charts, but must be configured separately as described on Monitoring Optimizer Hub.

- In the past, each release was bundled with the most likely JVM compiler engine. This is no longer the cause, resulting in smaller images.

- Session rebalancing has been improved with an (optional) Envoy proxy, or any other gRPC-aware load balancer/ingress in your Kubernetes cluster. More information can be found on Configuring gRPC Proxy.

- Documentation has been extended with installation instructions for Google Cloud.

Known Issues

Fixed Ports for gRPC

The helm chart values contain the keys `gateway.service.httpEndpoint.port` and `gateway.service.grpc.port` to change the default ports 50051 and 8080. But these values are hardcoded for the gRPC Envoy proxy, at this moment, and cannot be changed with the mentioned helm chart keys.

Cloud Native Compiler 1.7.1

Release Date: June 30, 2023

New Features

- Profile Log Service now stores profile metadata in the blob storage. This means that you can use AWS S3 or Azure Blob Storage to persist profile metadata and no longer need to back up the database pod with persistent storage. This change also means that when you upgrade from any release prior to 1.7.1 your previously collected profiles are no longer available.

  - Because of this change, the db component (MariaDB) is no longer needed when running CNC in Profile Log Service-only mode.

- Profile Log service automatically cleans-up unused profile names when not requested for a defined time. You can configure the duration with `profileLogService.cleaner.keepUnrequestedProfileNamesFor`. See readynow-orchestrator-defaults for more configuration information.

- New version of the Grafana monitoring dashboard with additional charts, and

updates related to changes in the metrics reported by CNC components.
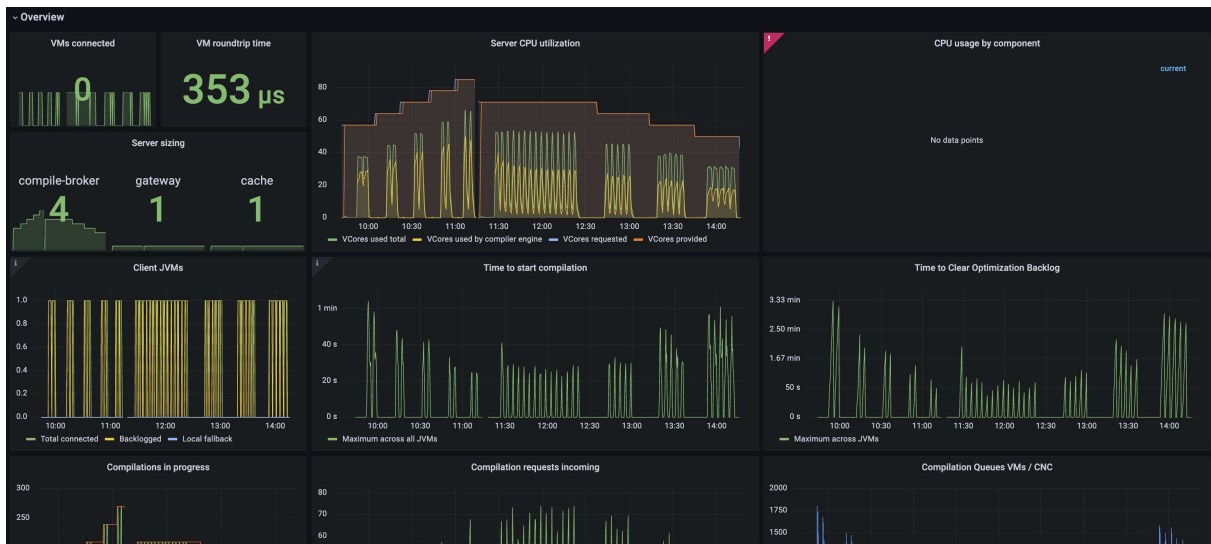
- You can define the profile log name with a Java property specified in the command line, in the format `%prop={PROPERTY}%`. For more info, see substitution-macros.

- Improved setup for Profile Log Service-only deployment.

- CNC can automatically recover from DB pod restarts with loss of schema. To enable this feature, set the following value in `values-override.yaml`:

```
dbschema.auto-recreate.enabled=true
```

- The `hostPort` attribute is no longer required and included for the storage pod.

Cloud Native Compiler 1.7.0

Release Date: May 3, 2023

New Features

- Improved performance of autoscaling for the Compiler Service.

- Usability improvements to the Profile Log Service Admin REST API.

- Native blob storage on Azure and AWS. Extra documentation is provided on:

  ○ configuring-aws-s3-storage

○ configuring-azure-blob-storage

- Added documentation of the [CNC API](#).

Cloud Native Compiler 1.6.3

Release Date: May 24, 2023

New Feature

Fix to prevent the storage pod from crashing with persistent volume enabled on CNC 1.6.2.

Cloud Native Compiler 1.6.2

Release Date: April 27, 2023

New Features

- The CNC helm charts now use full names for the Docker images to prevent issues in environments where a Docker Hub mirror is used.

- CNC pods can now be run as non-root user. The Docker images have a non-root user and the Helm chart is instructing Kubernetes to use this non-root user for CNC pods.

Upgrade

Follow the steps described on ["Upgrading Cloud Native Compiler"](#).

Cloud Native Compiler 1.6.1

Release Date: March 1, 2023

New Features

- To avoid restarts of the Gateway pod when a large number of clients try to write profile logs at the same time, a default limit has been configured.

- Upgrade from version 1.6.0 can be done with a helm upgrade, as described on [Upgrading Cloud Native Compiler](#).

Bug Fixes

- Gateway pod gets restarted when large number of clients try to write profile simultaneously.

Known Issues

- JVMs released before CNC 1.6.1 use HTTP for uploads of the compiler engine. Since version 1.6.1, gRPC is used and the HTTP port is disabled by default in values.yaml. Because of this, these JVMs are not able to upload their appropriate compiler engine to CNC.

  When a CNC version prior to 1.6.1 already has been used and upgraded, the older JVMs keep working with CNC, because the upload is not needed anymore.

- The first attempt to download a previously existing profile, after CNC upgrade to 1.6.1 can fail with a timeout.

Cloud Native Compiler 1.6.0

Release Date: January 30, 2023

New Features

- Cloud Native Compiler has a new Profile Log Service. This service allows you to read and write ReadyNow profile logs to Cloud Native Compiler. This simplifies getting profile logs in and out of containers and other environments without persistent storage. For more information on Profile Log Service configuration, see "Using the Profile Log Service".

- Introduced ReadyNow-only deployment to helm charts.

Bug Fixes

- Multiple APIs failed with empty response.

- Cache requests latency increased manifold resulting in an increase in wait time and overall compilation duration.

Known Issues

- In case of heavy applications, if you see anomalies in TTCOB, the problem can be resolved by increasing the number of cache pods. For more info, see cloud_native_compiler_troubleshooting.

Cloud Native Compiler 1.5.0

Release Date: October 31, 2022

New Features

- Compiler Cache on by default.

- New Time to Clear Optimization Backlog metric in Grafana dashboard.

Known Issues

- Multiple pods can get evicted because of low ephemeral storage in a long-running Code Cache cluster.

Cloud Native Compiler 1.4.0

Release Date: July 8, 2022

New Features

- Early access of the Compiler Cache. The Compiler Cache stores previously performed optimizations and serves them from the cache rather than recompiling whenever possible. Running your workloads with a Compiler Cache leads to lower CNC CPU usage and faster warmup time.

Known Issues

- Compiler Cache is not scalable and too many connections overload the database.

- Multiple pods can get evicted because of low ephemeral storage in a long-running Code Cache cluster.

Cloud Native Compiler 1.3.0

Release Date: May 9, 2022

New Features

- Simplified installation and configuration with Helm charts.

Known Issues

- ZVM-23070 - Using Cloud Native Compiler with local ReadyNow can dramatically increase the CPU required to deliver the compilations in time. Monitor your compiler output and look for connections being rejected and the JVM switching to local compilation, and scale out your CNC instance accordingly.

Cloud Native Compiler 1.2.0

Release Date: February 24, 2021

New Features

- Fallback to local JIT compilation when Cloud Native Compiler is unreachable or underperforming.

- You can now provide an existing ReadyNow profile as the input of the `-XX:ProfileLogIn={file}` flag. Note that generating a ReadyNow profile using the `-XX:ProfileLogOut={file}` is not supported with Cloud Native Compiler yet.

Cloud Native Compiler 1.1.0

Release Date: December 20, 2021

New Features

- Built-in monitoring stack with Prometheus and Grafana.

- JDK 17 support.

Known Issues

- The CNC gateway is currently configured with one instance. Do not attempt to increase the number of gateway instances.

- Extremely slow disk I/o configurations (with latencies in the multiple seconds) can lead to internal crashes and data loss within CNC (due to Artemis crashes). Avoid

configuring CNC with pods using very slow HDD or network volumes.

Cloud Native Compiler 1.0.0

Release Date: October 15, 2021

This is the first release of Cloud Connected Compiler (CNC), and we are really excited about it!

New Features

- Cloud Native Compiler server able to provide JIT compilations to Azul Zulu Prime Builds of OpenJDK 12.09.1.0 and later.

- Configuration files to provision an AWS Elastic Kubernetes Service cluster for your CNC server.

- A sample Grafana dashboard for monitoring your CNC server.

Azul Platform Core Third Party Licenses

This page contains links to the documents with licenses for third party software included in Optimizer Hub.

Version	Optimizer Hub TPL
1.9.5	PDF
1.9.4	PDF
1.9.3	PDF
1.9.2	PDF
1.9.1	PDF
1.9.0	PDF

Optimizer Hub Installation Instructions

Installing Optimizer Hub

Optimizer Hub is shipped as a Kubernetes cluster which you provision and run on your cloud or on-premise servers.

Supported Platforms

Optimizer Hub is available for **x64** platforms only, however, supports connections from Zing JVMs running on both x86 and ARM 64-bit machines.

Supported Kubernetes Environments

You can install Optimizer Hub on any Kubernetes cluster:

- **Kubernetes** clusters that you manually configure with [kubeadm](#).

  "Installing Optimizer Hub on Kubernetes".

- A single-node **minikube** cluster.

  "Installing Optimizer Hub on Minikube".

- **Managed cloud Kubernetes** services such as Amazon Web Services Elastic Kubernetes Service (EKS), Google Kubernetes Engine, and Microsoft Azure Managed Kubernetes Service.

  "Installing Optimizer Hub on Elastic Kubernetes Service".

NOTE	By downloading and using Optimizer Hub, you agree with the [Azul Platform Prime Evaluation Agreement](#).

Installing Optimizer Hub on Kubernetes

Optimizer Hub uses Helm as the deployment manifest package manager. There is no need to manually edit any Kubernetes deployment manifests. You can configure the installation overriding the default settings from [values.yaml](#) in a custom values file. Here we refer to the file as `values-override.yaml` but you can give it any name.

	This section describes setting up an evaluation or developer version of Optimizer Hub without SSL authentication. To set up a production version with full SSL authentication, see "Configuring Optimizer Hub with SSL Authentication".
NOTE	

You should install Optimizer Hub in a location to which the JVM machines have unauthenticated access. You can run Optimizer Hub in the same Kubernetes cluster as the client VMs or in a separate cluster.

	If you are upgrading an existing installation, make sure to check "Upgrading Optimizer Hub".
NOTE	

Optimizer Hub Helm Charts

Azul provides Optimizer Hub Helm Charts on GitHub. You can download the full package as a zip.

Installing Optimizer Hub

These instructions are for installing a full Optimizer Hub instance with both Cloud Native Compiler and ReadyNow Orchestrator. In case you only want to install the full Optimizer Hub, but only a part of the services, see "Configuring the Active Optimizer Hub Services".

1. Install Azul Zulu Prime Builds of OpenJDK 21.09.1.0 or newer on your client machine.

2. Make sure your Helm version is `v3.8.0` or newer.

3. Add the Azul Helm repository to your Helm environment:

```
helm repo add opthub-helm https://azulsystems.github.io/opthub-helm-charts/
helm repo update
```

4. Create a namespace (i.e. `my-opthub`) for Optimizer Hub.

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

5. Create the `values-override.yaml` file in your local directory.

6. If you have a custom cluster domain name, specify it in `values-override.yaml`:

```
clusterName: "example.org"
```

7. If you want specific labels being added to your Kubernetes objects, define them in your `values-override.yaml`, for example as follows:

```
gateway:
  applicationLabels: # Additional labels for Deployment/StatefulSet
  podTemplateLabels: # Additional labels for POD
  serviceLabels: # Additional labels for Service
```

8. Configure sizing and autoscaling of the Optimizer Hub components according to the "sizing guide". By default, autoscaling is on and Optimizer Hub can scale up to 10 Compile Brokers. For example, you could set the following:

```
simpleSizing:
  vCores: 32
  minVCores: 32
  maxVCores: 106
```

9. If needed, configure external access in your cluster. If your JVMs are running within the same cluster as Optimizer Hub, you can ignore this step. Otherwise, it is necessary to configure an external load balancer in `values-override.yaml`.

   For clusters running on AWS [an example configuration file is available on Azul's GitHub](#).

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

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

   ○ In case you need a specific Optimizer Hub version, please use `--version` `1.9.5` flag.

○ The command should produce output similar to this:

```
NAME: opthub
LAST DEPLOYED: Wed Jan 31 12:19:58 2024
NAMESPACE: my-opthub
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

11. Verify that all started pods are ready:

```
kubectl get all -n my-opthub
```

Configuring Persistent Storage

By default, Optimizer Hub pods allocate data directories on the root disk or in an `emptyDir` volume, both residing in the pod's ephemeral storage. If the pod dies, all data is lost and has to be regenerated after restart.

You can move the pods' data directories to persistent volumes, so the data survives pod crashes, restarts and even scale down/up events. Furthermore, this allows you to lower the local storage sizing of target Kubernetes worker nodes, since large data directories are stored in separate volumes outside of these worker nodes.

When you use persistent volumes, you create 2 additional Kubernetes objects per pod:

- `persistentVolumeClaim` (PVC), whose name is derived from parent pod

- `persistentVolume` (PV), which is allocated automatically by chosen the storage class and has an auto-generated name.

PV and PVC objects lifecycles are separate from other Optimizer Hub Kubernetes objects. When you uninstall Optimizer Hub using the helm chart, these objects remain in cluster for as long as the installation namespace exists. Removal of namespace or manual deletion of PVCs within the namespace automatically removes their associated PVs from the Kubernetes cluster as well.

You can configure persistent volumes for the `db` and `builtinStorage` components. The configuration is the same for both components. Your target Kubernetes cluster needs to have at least one storage class configured. By default, Optimizer Hub uses the default configured storage class.

NOTE	If you are using **AWS EBS Storage** for your persistent storage, use gp3 volumes instead of gp2 volumes. gp2 volumes have limited IOPS which can affect Optimizer Hub performance. Additional configuration info for **AWS S3 Storage** is configuring-aws-s3-storage.

NOTE	If you are using **Azure Blob Storage**, please check "Installing Optimizer Hub on Azure" for additional settings.

Configuration with Custom Resources Values

Example pod sizing with 10GiB for root volume and 100GiB for data volume:

```
db:
  resources:
    requests:
      cpu: "5"
      memory: "20Gi"
      ephemeral-storage: "10Gi"
    limits:
      cpu: "5"
      memory: "20Gi"
      ephemeral-storage: "10Gi"
  persistentDataVolume:
    enabled: true
    size: "100Gi"
```

If you want to use recommended sizing of pods, you still need to explicitly override the default size of the ephemeral storage. This is in order to not waste resources and increase pod schedulability on smaller sized nodes.

```
db:
  resources:
    requests:
      ephemeral-storage: "10Gi"
    limits:
      ephemeral-storage: "10Gi"
  persistentDataVolume:
    enabled: true
    size: "100Gi"
```

Configuration with Custom Storage Class

If your cluster has multiple configured storage classes, and you want to use a non-default storage class, do the following:

```
db:
  resources:
  persistentDataVolume:
    enabled: true
    storageClassName: "my-storage-class"
```

Enabling the Management Gateway

The Management Gateway enables two pieces of functionality:

- Access to REST APIs for managing ReadyNow profiles

- Cross-region synchronization of ReadyNow Profiles

To enable the Management Gateway, set `mgmtGateway.enabled` to true in `value-override.yaml`, see management-gateway-parameters for more info.

Cleaning Up

To uninstall a deployed Optimizer Hub, run the following command:

```
helm uninstall opthub -n my-opthub
kubectl delete namespace my-opthub
```

Installing Optimizer Hub on AWS Elastic Kubernetes Service

If you are using Amazon Web Services, you can simplify the process of starting and

maintaining your cluster considerably by using the [Elastic Kubernetes Service](#) (EKS).

Provisioning on EKS

To provision Optimizer Hub on EKS:

1. Install and configure the `eksctl` and `aws` command-line tools.

   If you don't have permissions to set up networking components, have your administrator create the Virtual Public Cloud.

2. Download [opthub-install-1.9.5.zip](#). Navigate to the `opthub-install/eks` directory. You can pass the `opthub_eks.yaml` file to the `eksctl` to create the cluster. For more information, look at the [eskctl config file schema](#).

3. Replace the placeholders `{your-cluster-name}`, `{your-region}`, and `{path-to-your-key}` with the correct values.

4. If you are working with an existing VPC and do not want eksctl to create one, uncomment the `vpc` section and replace `{your-vpc}` and `{your-subnet}` with the correct values.

5. Apply the file with the following command:

   ```
   eksctl create cluster -f opthub_eks.yaml
   ```

   This command takes several minutes to execute.

   Successful command output:

   ```
   2021-08-20 20:09:53 [¬] eksctl version 0.60.0
   2021-08-20 20:09:53 [¬] using region eu-central-1
   2021-08-20 20:09:54 [¬] setting availability zones to [eu-central-
   1a eu-central-1b eu-central-1c]
   2021-08-20 20:09:54 [¬] subnets for eu-central-1a -
   public:192.168.0.0/19 private:192.168.96.0/19
   2021-08-20 20:09:54 [¬] subnets for eu-central-1b -
   public:192.168.32.0/19 private:192.168.128.0/19
   2021-08-20 20:09:54 [¬] subnets for eu-central-1c -
   public:192.168.64.0/19 private:192.168.160.0/19
   2021-08-20 20:09:54 [¬] nodegroup "infra" will use "ami-
   ```

```
05f67790af078876f" [AmazonLinux2/1.19]
2021-08-20 20:09:54 [¬] using SSH public key
"/Users/XXXXXXXX/.ssh/id_rsa.pub" as "eksctl-eks-opthub-cluster-
nodegroup-infra-19:01:7b:fb:83:19:12:bb:17:59:40:37:22:dc:82:86"
2021-08-20 20:09:54 [¬] nodegroup "opthubservice" will use "ami-
05f67790af078876f" [AmazonLinux2/1.19]
2021-08-20 20:09:54 [¬] using SSH public key
"/Users/XXXXXXXX/.ssh/id_rsa.pub" as "eksctl-eks-opthub-cluster-
nodegroup-opthubserver-
19:01:7b:fb:83:19:12:bb:17:59:40:37:22:dc:82:86"
2021-08-20 20:09:54 [¬] nodegroup "opthubcache" will use "ami-
05f67790af078876f" [AmazonLinux2/1.19]
2021-08-20 20:09:54 [¬] using SSH public key
"/Users/XXXXXXXX/.ssh/id_rsa.pub" as "eksctl-eks-opthub-cluster-
nodegroup-opthubcache-
19:01:7b:fb:83:19:12:bb:17:59:40:37:22:dc:82:86"
2021-08-20 20:09:54 [¬] nodegroup "opthubinfra" will use "ami-
05f67790af078876f" [AmazonLinux2/1.19]
2021-08-20 20:09:54 [¬] using SSH public key
"/Users/XXXXXXXX/.ssh/id_rsa.pub" as "eksctl-eks-opthub-cluster-
nodegroup-opthubinfra-
19:01:7b:fb:83:19:12:bb:17:59:40:37:22:dc:82:86"
2021-08-20 20:09:55 [¬] using Kubernetes version 1.19
2021-08-20 20:09:55 [¬] creating EKS cluster "eks-opthub-cluster"
in "eu-central-1" region with un-managed nodes
2021-08-20 20:09:55 [¬] 4 nodegroups (opthubcache, opthubinfra,
opthubserver, infra) were included (based on the include/exclude
rules)
2021-08-20 20:09:55 [¬] will create a CloudFormation stack for
cluster itself and 4 nodegroup stack(s)
2021-08-20 20:09:55 [¬] will create a CloudFormation stack for
cluster itself and 0 managed nodegroup stack(s)
2021-08-20 20:09:55 [¬] if you encounter any issues, check
CloudFormation console or try 'eksctl utils describe-stacks
--region=eu-central-1 --cluster=eks-opthub-cluster'
2021-08-20 20:09:55 [¬] CloudWatch logging will not be enabled for
cluster "eks-opthub-cluster" in "eu-central-1"
2021-08-20 20:09:55 [¬] you can enable it with 'eksctl utils
update-cluster-logging --enable-types={SPECIFY-YOUR-LOG-TYPES-HERE
(e.g. all)} --region=eu-central-1 --cluster=eks-opthub-cluster'
2021-08-20 20:09:55 [¬] Kubernetes API endpoint access will use
default of {publicAccess=true, privateAccess=false} for cluster
"eks-opthub-cluster" in "eu-central-1"
2021-08-20 20:09:55 [¬] 2 sequential tasks: { create cluster
control plane "eks-opthub-cluster", 3 sequential sub-tasks: { wait
for control plane to become ready, 1 task: { create addons },
4 parallel sub-tasks: { create nodegroup "infra", create nodegroup
"opthubserver", create nodegroup "opthubcache", create nodegroup
"opthubinfra" } } }
2021-08-20 20:09:55 [¬] building cluster stack "eksctl-eks-opthub-
```

```
cluster-cluster"
2021-08-20 20:09:55 [¬] deploying stack "eksctl-eks-opthub-cluster-
cluster"
2021-08-20 20:10:25 [¬] waiting for CloudFormation stack "eksctl-
eks-opthub-cluster-cluster"
2021-08-20 20:10:55 [¬] waiting for CloudFormation stack "eksctl-
eks-opthub-cluster-cluster"
2021-08-20 20:19:57 [¬] waiting for CloudFormation stack "eksctl-
eks-opthub-cluster-cluster"
...
2021-08-20 20:20:58 [¬] waiting for CloudFormation stack "eksctl-
eks-opthub-cluster-cluster"
2021-08-20 20:25:06 [¬] building nodegroup stack "eksctl-eks-
opthub-cluster-nodegroup-opthubinfra"
2021-08-20 20:25:06 [¬] building nodegroup stack "eksctl-eks-
opthub-cluster-nodegroup-opthubcache"
2021-08-20 20:25:06 [¬] building nodegroup stack "eksctl-eks-
opthub-cluster-nodegroup-opthubserver"
2021-08-20 20:25:06 [¬] building nodegroup stack "eksctl-eks-
opthub-cluster-nodegroup-infra"
2021-08-20 20:25:07 [¬] deploying stack "eksctl-eks-opthub-cluster-
nodegroup-infra"
2021-08-20 20:25:07 [¬] waiting for CloudFormation stack "eksctl-
eks-opthub-cluster-nodegroup-infra"
2021-08-20 20:25:07 [¬] deploying stack "eksctl-eks-opthub-cluster-
nodegroup-opthubserver"
2021-08-20 20:25:07 [¬] waiting for CloudFormation stack "eksctl-
eks-opthub-cluster-nodegroup-opthubserver"
2021-08-20 20:25:07 [¬] deploying stack "eksctl-eks-opthub-cluster-
nodegroup-opthubcache"
2021-08-20 20:25:07 [¬] waiting for CloudFormation stack "eksctl-
eks-opthub-cluster-nodegroup-opthubcache"
2021-08-20 20:25:07 [¬] deploying stack "eksctl-eks-opthub-cluster-
nodegroup-opthubinfra"
2021-08-20 20:25:07 [¬] waiting for CloudFormation stack "eksctl-
eks-opthub-cluster-nodegroup-opthubinfra"
2021-08-20 20:25:23 [¬] waiting for CloudFormation stack "eksctl-
eks-opthub-cluster-nodegroup-infra"
2021-08-20 20:25:24 [¬] waiting for CloudFormation stack "eksctl-
eks-opthub-cluster-nodegroup-opthubcache"
...
2021-08-20 20:32:16 [¬] waiting for CloudFormation stack "eksctl-
eks-opthub-cluster-nodegroup-opthubcache"
2021-08-20 20:32:16 [¬] waiting for the control plane
availability...
2021-08-20 20:32:16 [¬] saved kubeconfig as
"/Users/XXXXXXXX/.kube/config"
2021-08-20 20:32:16 [¬] no tasks
2021-08-20 20:32:16 [¬] all EKS cluster resources for "eks-opthub-
cluster" have been created
```

```
2021-08-20 20:32:16 [¬] adding identity
"arn:aws:iam::912192438162:role/eksctl-eks-opthub-cluster-
nodegroup-infra-NodeInstanceRole-9VFWHMM30SSV" to auth ConfigMap
2021-08-20 20:32:16 [¬] nodegroup "infra" has 0 node(s)
2021-08-20 20:32:16 [¬] waiting for at least 1 node(s) to become
ready in "infra"
2021-08-20 20:32:49 [¬] nodegroup "infra" has 1 node(s)
2021-08-20 20:32:49 [¬] node "ip-192-168-90-183.eu-central-
1.compute.internal" is ready
2021-08-20 20:32:49 [¬] adding identity
"arn:aws:iam::912192438162:role/eksctl-eks-opthub-cluster-
nodegroup-opthubser-NodeInstanceRole-16JA2COTZHLWQ" to auth
ConfigMap
2021-08-20 20:32:49 [¬] nodegroup "opthubserver" has 0 node(s)
2021-08-20 20:32:49 [¬] waiting for at least 1 node(s) to become
ready in "opthubserver"
2021-08-20 20:33:49 [¬] nodegroup "opthubserver" has 1 node(s)
2021-08-20 20:33:49 [¬] node "ip-192-168-90-115.eu-central-
1.compute.internal" is ready
2021-08-20 20:33:49 [¬] adding identity
"arn:aws:iam::912192438162:role/eksctl-eks-opthub-cluster-
nodegroup-opthubcac-NodeInstanceRole-5KIIEOTU3ELU" to auth
ConfigMap
2021-08-20 20:33:49 [¬] nodegroup "opthubcache" has 0 node(s)
2021-08-20 20:33:49 [¬] waiting for at least 1 node(s) to become
ready in "opthubcache"
2021-08-20 20:34:21 [¬] nodegroup "opthubcache" has 1 node(s)
2021-08-20 20:34:21 [¬] node "ip-192-168-70-66.eu-central-
1.compute.internal" is ready
2021-08-20 20:34:21 [¬] adding identity
"arn:aws:iam::912192438162:role/eksctl-eks-opthub-cluster-
nodegroup-opthubinf-NodeInstanceRole-103G0W4M1XCZ7" to auth
ConfigMap
2021-08-20 20:34:21 [¬] nodegroup "opthubinfra" has 0 node(s)
2021-08-20 20:34:21 [¬] waiting for at least 1 node(s) to become
ready in "opthubinfra"
2021-08-20 20:35:37 [¬] nodegroup "opthubinfra" has 1 node(s)
2021-08-20 20:35:37 [¬] node "ip-192-168-46-62.eu-central-
1.compute.internal" is ready
2021-08-20 20:37:39 [¬] kubectl command should work with
"/Users/XXXXXXXX/.kube/config", try 'kubectl get nodes'
2021-08-20 20:37:39 [¬] EKS cluster "eks-opthub-cluster" in "eu-
central-1" region is ready
```

Here is everything that `opthub_eks.yaml` creates in your AWS account:

- CloudFormation stacks for the main EKS cluster and each of the NodeGroups in the cluster.

- A Virtual Private Cloud called eksctl-{cluster-name}-cluster/VPC. If you chose to use an existing VPC, this is not created. You can explore the VPC and its related networking components in the AWS VPC console. The VPC has all of the required networking components configured:

  - A set of three public subnets and three private subnets

  - An Internet Gateway

  - Route Tables for each of the subnets

  - An Elastic IP Address for the cluster

  - A NAT Gateway

- An EKS Cluster, including four nodegroups with one m5.2xlarge instance provisioned:

  - `infra` - For running Grafana and Prometheus.

  - `opthubinfra` - For running the Optimizer Hub infrastructure components.

  - `opthubcache` - For running the Optimizer Hub cache.

  - `opthubserver` - For running the Optimizer Hub compile broker settings.

- IAM artifacts for the Autoscaling Groups:

  - Roles for the Autoscaler groups for the cluster and for each subnet

  - Policies for the EKS autoscaler

Setting Up an External Load Balancer

If you need to connect to Optimizer Hub from outside the Kubernetes cluster, you need to setup up a load balancer in front of the gateway instances:

To set up a load balancer, please follow AWS documentation regarding load balancer controller setup.

Installing Optimizer Hub on EKS

Because `opthub_eks.yaml` file creates the nodegroups in the cluster, you have to pass in an additional configuration file when installing via Helm. The

`opthub_eks.yaml` file is located in `opthub-install/eks/values-eks.yaml` and includes the nodegroup affinity settings and other settings EKS expects.

To continue with the full installation instructions for Optimizer Hub, please refer to "Installing Optimizer Hub on Kubernetes". In case you don't want to install the full Optimizer Hub, but only a part of the services, check "Configuring the Active Optimizer Hub Services".

To install using the `values-eks.yaml` config file, run the following command:

```
helm install opthub opthub-helm/azul-opthub -n my-opthub -f values-
eks.yaml -f values-override.yaml
```

When adding multiple values files, remember the last one takes precedence.

Configuring AWS S3 Storage

You can configure your Optimizer Hub to use AWS S3 storage instead of the internal blob storage in the internal builtInStorage pod. When you use AWS S3 storage, the builtInStorage pod is not created at all.

You can configure S3 storage by adding the following to `values-override.yaml`:

```
storage:
  blobStorageService: s3 # available options: builtin-storage, azure-
blob, s3
  s3:
    # opthub-* buckets examples: opthub-sandbox, opthub-demo
    commonBucket: opthub-storage0
```

Using Kubernetes Nodes and Permissions

To configure AWS S3 storage, use the following configuration. Ensure that your Kubernetes nodes with `opthub-compilebroker` and `opthub-gateway` have RW permissions to S3 bucket(s), and the target buckets exist.

A role with the below policy must be assigned to instances (EC2, EC2 ASG, Fargate, etc) for the `opthub-compilebroker` and `opthub-gateway` pods.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "s3:ListBucket"
      ],
      "Resource": [
        "arn:aws:s3:::opthub-*"
      ],
      "Effect": "Allow"
    },
    {
      "Action": [
        "s3:*Object"
      ],
      "Resource": [
        "arn:aws:s3:::opthub-*/*"
      ],
      "Effect": "Allow"
    }
  ]
}
```

Using AWS Service Accounts

If your security practices do not allow you to give nodes access to S3 buckets, you can also grant access to just the key services in Optimizer Hub. You can do this by configuring AWS IAM, roles, and permissions as described in the AWS documentation.

In the next steps, Optimizer Hub assumes the role name is `opthub-s3-role`. The IAM role trust relationship entry needs the following additional settings in AWS (you will need to change the IDs in this example to align with your configuration):

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Federated": "arn:aws:iam::163957972732:oidc-
provider/oidc.eks.us-west-
2.amazonaws.com/id/F7E8B430691CFE3B776B8CA663896762"
            },
            "Action": "sts:AssumeRoleWithWebIdentity",
            "Condition": {
                "StringLike": {
                    "oidc.eks.us-west-
2.amazonaws.com/id/F7E8B430691CFE3B776B8CA663896762:sub":
"system:serviceaccount:*:opthub*",
                    "oidc.eks.us-west-
2.amazonaws.com/id/F7E8B430691CFE3B776B8CA663896762:aud":
"sts.amazonaws.com"
                }
            }
        }
    ]
}
```

After creating the Service Accounts, add the following settings to your `values-override.yaml` file:

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

The Helm chart of Optimizer Hub creates the following Service Accounts:

- `opthub-cache`

- `opthub-compile-broker`

- `opthub-gateway`

- `opthub-operator`

Storage for ReadyNow Orchestrator

You can limit the usage of persistent storage by ReadyNow Orchestrator with the readynow-orchestrator-defaults.

Cleaning Up

Run the following command:

```
eksctl delete cluster -f opthub_eks.yaml
```

Installing Optimizer Hub on Microsoft Azure

To install Optimizer Hub on Azure, follow the general "Kubernetes" instructions. This document provides additional configurations specific for Azure.

Configuring Azure Blob Storage

Following Helm values activate Azure Blob Storage. Currently, the default configuration uses MinIO which is deployed as part of Optimizer Hub.

```
storage:
  blobStorageService: azure-blob
  azureBlob:
    endpoint: https://{yourendpoint}.blob.core.windows.net
    container: {your-container}
    authMethod: {method} # sas-token, connection-string, or default-
credentials
```

- When using `authMethod:sas-token`:

```
secrets:
  azure:
    blobStorage:
      sasToken: "{your-token}"
```

- When using `authMethod:connection-string`:

```
secrets:
  azure:
    blobStorage:
      connectionString: "{your-connection-string}"
```

Storage for ReadyNow Orchestrator

You can limit the usage of persistent storage by ReadyNow Orchestrator with the readynow-orchestrator-defaults.

Installing Optimizer Hub on Google Cloud

To install Optimizer Hub on Google Cloud, please follow the instructions on "Installing Optimizer Hub on Kubernetes".

If you want to install Optimizer Hub on Google Cloud with S3 compatibility mode, instead of the builtin storage pod, you need the following additional settings.

Configuring Storage

Use the `S3` compatible storage and specify a bucket name in your `values-override.yaml`:

```
storage:
  blobStorageService: s3
  s3:
    commonBucket: opthub-storage0
```

Configuring Compile Broker

Add the following `extraArgumentsMap` section under `compileBroker` in your `values-override.yaml`:

```
compileBroker:
  extraArgumentsMap:
    "quarkus.s3.endpoint-override": "https://storage.googleapis.com"
    "quarkus.s3.aws.credentials.type": static
    "quarkus.s3.aws.credentials.static-provider.access-key-id":
  "{your access key}"
    "quarkus.s3.aws.credentials.static-provider.secret-access-key":
  "{your secret key}"
```

Configuring Gateway

Add the following `extraArgumentsMap` section `gateway` in your `values-override.yaml`:

```
gateway:
  extraArgumentsMap:
    "quarkus.s3.endpoint-override": "https://storage.googleapis.com"
    "quarkus.s3.aws.credentials.type": static
    "quarkus.s3.aws.credentials.static-provider.access-key-id":
"{your access key}"
    "quarkus.s3.aws.credentials.static-provider.secret-access-key":
"{your secret key}"
```

Configuring Cache

Add the following `extraArgumentsMap` section `cache` in your `values-override.yaml`:

```
cache:
  extraArgumentsMap:
    "quarkus.s3.endpoint-override": "https://storage.googleapis.com"
    "quarkus.s3.aws.credentials.type": static
    "quarkus.s3.aws.credentials.static-provider.access-key-id":
"{your access key}"
    "quarkus.s3.aws.credentials.static-provider.secret-access-key":
"{your secret key}"
```

Installing Optimizer Hub on Minikube

Minikube can be used for testing and evaluating Optimizer Hub.

You should run Optimizer Hub on minikube only for evaluation purposes. Make sure your minikube meets the 18 vCore minimum for running Optimizer Hub. Although minikube can run on multiple platforms, Optimizer Hub is only available for the **x64** platform, so not on macOS with M1/2.

Installing Minikube

Install minikube for your platform following this installation guide.

Installing Optimizer Hub

Optimizer Hub uses Helm as the deployment manifest package manager. There is no need to manually edit any Kubernetes deployment manifests.

1. Make sure your Helm version is `v3.8.0` or newer.

2. Add the Azul Helm repository to your Helm environment:

```
helm repo add opthub-helm https://azulsystems.github.io/opthub-
helm-charts/
helm repo update
```

3. Create a namespace (i.e. `my-opthub`) for Optimizer Hub.

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

4. Create a configuration file `values-minikube.yaml`.

   An example file is available on [GitHub in the Azul "opthub-helm-charts" project](), to disable all resource definitions.

   As the supplied values file for minikube resets pod resources to null, we can simply add only the persistent volume section:

```
db:
  resources:
  persistentDataVolume:
    enabled: true
```

   You can also set the volume size if the default 200Gi is too big for local testing:

```
db:
  resources:
  persistentDataVolume:
    enabled: true
    size: "50Gi"
```

5. Install using Helm, passing in the `values-minikube.yaml`. In case you don't want

to install the full Optimizer Hub, but only a part of the services, first check "Configuring the Active Optimizer Hub Services".

```
helm install opthub opthub-helm/azul-opthub -n my-opthub -f values-
minikube.yaml
```

The command should produce output similar to this:

```
NAME: opthub
LAST DEPLOYED: Mon Jan 30 14:35:29 2023
NAMESPACE: my-opthub
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

6. Verify that all started pods are ready:

```
minikube kubectl -- get all -n my-opthub
```

Uninstalling Optimizer Hub from Minikube

Optimizer Hub can be removed from minikube using `helm`, after which the namespace can also be deleted.

```
helm uninstall opthub -n my-opthub
minikube kubectl -- delete namespace my-opthub
```

Upgrading Optimizer Hub

Upgrade an existing Optimizer Hub installation to a newer version with the following commands:

```
helm repo update
helm upgrade opthub opthub-helm/azul-opthub -n my-opthub -f values-
override.yaml
kubectl get all -n my-opthub
```

When upgrading an existing Optimizer Hub installation, make sure to validate your

`values-override.yaml` file, as parameters might have changed.

Changed Values in 1.8

In Optimizer Hub 1.8, all major artifacts and command line switches use the updated branding. This includes, but is not limited to:

- Command-line JVM options to configure "Cloud Native Compiler" and readynow-orchestrator-jvm-options.

- Helm repository locations, names, and parameter names:
  github.com/AzulSystems/opthub-helm-charts.

- "REST API URLs".

NOTE	If you are using release 1.7 and earlier, all of the previous spellings of artifacts still work. Additionally, all of the pre-1.8 command-line arguments continue to work for a period of one year from the release of 1.8.

Upgrade From Specific Versions

From Version 1.7.0

If you are upgrading **from versions prior to 1.7.0**, and you are using a custom `values.yaml` file with parameters for the storage component, rename the yaml block from `storage` to `builtinStorage`. For example:

Before

```
storage:
  persistentDataVolume:
    enabled: true
    size: "200Gi"
    storageClassName: ""
```

After

```
builtinStorage:
  persistentDataVolume:
    enabled: true
    size: "200Gi"
    storageClassName: ""
```

From Version 1.6.1

If you are upgrading **from version 1.6.1** with persistent storage, follow these steps before running the helm upgrade:

1. Connect into the storage pod:

   ```
   kubectl exec --stdin --tty storage-0 -- /bin/sh
   ```

2. Inside the pod run the following command to change permissions:

   ```
   chown -R 10001 /data && chmod u+rxw /data
   ```

3. After the `chmod` command is completed, you can exit the pod shell with `ctrl-d` and continue with the helm upgrade.

Configuring Optimizer Hub

Optimizer Hub Generic Defaults

Optimizer Hub is shipped as a Helm chart with all the defaults as specified in the values.yaml file. Here you find a list of the most important generic values that can be modified to match Optimizer Hub to your environment.

Specific settings can be found on the configuration pages of the service itself, for example, readynow-orchestrator-defaults.

Database Parameters

Option	Description	Default
db.enabled	Define wether the database node is installed. Is set to false in ReadyNow Orchestrator only mode, using `values-disable-compiler.yaml`	true

Database Schema Parameters

Option	Description	Default
dbschema.auto-recreate.enabled	If enabled, automatically recovers from database pod restarts with loss of schema. If you have a database and this database is not using a persistence volume, this setting must be set to true, otherwise you need a manual interaction if the pod is restored.	false

Management Gateway Parameters

Option	Description	Default
mgmtGateway.enabled	Define if the Management Gateway needs to be enabled to expose the REST APIs for ReadyNow Orchestrator and/or Cross-Region Sync.	false
mgmtGateway.service.httpEndpoint.port	The port used by the Management Gateway.	8080

Cross-Region Sync Parameters

Option	Description	Default
synchronization.enabled	Define if Cross-Region Sync needs to be enabled. You must also enable the [Management Gateway](#) for this setting to become effective.	true
synchronization.peers	A comma separated list of peer Management Gateway URLs from other Optimizer Hub instances to include in the syncing process.	
synchronization.initialDelay	Initial delay for the periodic synchronization task.	PT180s
synchronization.period	Defines a periodicity of a synchronization with the specified Optimizer Hub peers.	PT30s

Simple Sizing Parameters

See configuring-capacity.

SSL Parameters

See "Configuring Optimizer Hub with SSL Authentication".

Storage Parameters

Storage parameters depend on the platform of your deployment:

- storage

- configuring-aws-s3-storage

- configuring-azure-blob-storage

Configuring the Active Optimizer Hub Services

Optimizer Hub can run in different modes:

- **Full**: both the Cloud Native Compiler and ReadyNow Orchestrator are available.

  This is the default configuration.

- **ReadyNow only**: only ReadyNow Orchestrator is available.

  Use the installation instructions below.

Install Only ReadyNow Orchestrator

To install with **only** ReadyNow Orchestrator, pass in `values-disable-compiler.yaml`, together with your `values-override.yaml`:

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

Disabling Cloud Native Compiler on a Full Optimizer Hub Installation

If you installed a full installation of full Optimizer Hub with Cloud Native Compiler and ReadyNow Orchestrator, you can still disable Cloud Native Compiler by:

- Reinstalling as specified above.
- Or disable the Cloud Native Compiler globally using the `compilations.parallelism.limitPerVm` setting, with the value `0`, to override the default value of `500`.

Enabling the Management Gateway

The Management Gateway enables two pieces of functionality:

- Access to REST APIs for managing ReadyNow profiles
- Cross-region synchronization of ReadyNow Profiles

44

To enable the Management Gateway, set `mgmtGateway.enabled` to true in `value-override.yaml`, see management-gateway-parameters for more info.

Configuring Optimizer Hub Host and Port

As an Optimizer Hub administrator, you must provide users the host and ports for connecting to the service. Customers should use the host and port name in the `OptHubHost` JVM parameter.

Determining the Optimizer Hub Endpoint

Use the IP address of the Optimizer Hub `gateway` service as the connection endpoint for your JVMs.

Using an External Load Balancer

It is strongly recommended to use an external load balancer. If you correctly defined the load-balancer in `values-override.yaml` as described in "Installing Optimizer Hub", you can discover the external IP of the service using the following command:

```
$ kubectl describe service gateway -n my-opthub | grep 'LoadBalancer
Ingress:'
LoadBalancer Ingress:     internal-add1ff3e1591e4f93a49af3523b68e3b-
1321158844.us-west-2.elb.amazonaws.com
```

JVM customers then connect using the following command:

```
java -XX:OptHubHost=internal-add1ff3e1591e4f93a49af3523b68e3b-
1321158844.us-west-2.elb.amazonaws.com \
    -XX:+EnableRNO \
    -jar my-app.jar
```

Connecting Without an External Load Balancer

If you did not set up an external load balancer, you can find the endpoint using the following steps:

1. Run the following command:

```
kubectl -n my-opthub get services
```

2. Look for the `gateway` service and note the ports corresponding to port 50051 inside the container. This is the port to use for connecting VMs to this Optimizer Hub cluster.

```
service/gateway NodePort     10.233.15.55     <none>
8080:31951/TCP,50051:30926/TCP  52d
```

In this example the ports is `30926`.

NOTE	Only the internal ports `8080` and `50051` in Optimizer Hub are fixed. The port in each setup is a random value. You need to use this lookup to find the port of your Optimizer Hub instance.

3. Run the `kubectl get nodes` command and note the IP address or name of any node.

4. Concatenate node IP with service ports to get something like `10.22.20.131:30926`. Do not prefix it with `http://`.

5. JVM customers set `-XX:OptHubHost=host:port` flag to the port mapped to 50051.

```
java -XX:OptHubHost=10.22.20.131:30926 \
    -XX:+EnableRNO \
    -jar my-app.jar
```

Specifying a Custom Compiler Engine Upload Port

Cloud Native Compiler uses compiler engines to provide instructions for working with a specific version of the JVM. These compiler engines are not shipped with Cloud Native Compiler. When attempting to use Cloud Native Compiler for compilation, the JVM checks if the right compiler engine is present and, if not, automatically uploads it to Cloud Native Compiler.

If your Optimizer Hub instance is using default 8080 HTML ports, or you are fronting it with a load balancer, then there is nothing the user needs to do to configure uploads correctly. If you are connecting without a loadbalancer and are not using the default 8080 ports, follow the process described above to provide the JVM user with the host and port mapped to 8080. The JVM user must specify this host/port in the `-XX:CNCEngineUploadAddress=host:port`. In the above example, the host/port combination is `10.22.20.131:30926`.

Configuring gRPC Proxy

Optimizer Hub comes with Envoy as the default gRPC proxy for optimal session balancing. But it's recommended to use your own preferred load balancers, consistent with how you dispatch HTTP traffic to your other applications. In such a case, disable Envoy in Optimizer Hub and use your own instance.

Disabling Envoy in Optimizer Hub

Add the following to `values-override.yaml`:

```
gwProxy.enabled=false
```

Useful links

- [Deploy a gRPC-based application on an Amazon EKS cluster and access it with an Application Load Balancer](#).

- [Deploy gRPC services on the backend of the NGINX Ingress controller](#).

- [Kubernetes > Ingress-Nginx Controller > gRPC](#).

Configuring Optimizer Hub with SSL Authentication

While you can use Optimizer Hub without SSL authentication for development and evaluation, it is highly recommended that you run your production Optimizer Hub with SSL authentication.

To enable SSL authentication on your Optimizer Hub:

1. Establish your SSL certificate. To enable SSL encryption of the communication between the JVM and Optimizer Hub, you need to provide a certificate and a corresponding private key in the `pem` format.

> **NOTE**
>
> The common name field in the certificate must match the name of the Optimizer Hub service as provided to client JVMs via the `-XX:OptHubHost` flag. Otherwise there may be issues when connecting.

2. Enable SSL in your `values-overrride.yaml` file:

```
ssl:
  enabled: true
```

3. Add your certificate and private key. This can be done in several ways:

   a. The most secure way to add certificates is using a separate chain that manages your certificate. You can then point the deployment to a custom secret in the installation namespace. Such a secret needs to have keys named `cert.pem` and `key.pem`.

   ```
   ssl:
     secretName: "my-custom-secret"
   ```

   b. You can add the certificate and private keys directly to the values.yaml as values. This is the simplest way to run quick experiments in a controlled environment, especially when you're installing from the Helm repository. We do not recommend this approach in production as it embeds private security credentials in a config file:

```
ssl:
  value:
    cert: |-
      -----BEGIN CERTIFICATE-----
      ...
      -----END CERTIFICATE-----
    key: |-
      -----BEGIN PRIVATE KEY-----
      ...
      -----END PRIVATE KEY-----
```

c. If you downloaded and unpacked the Helm chart to a local directory, you can just place files named `cert.pem` and `key.pem` into the root directory of your Helm chart.

4. Perform Helm installation as shown in the "general installation guide".

Running Azul Zulu Prime JDK Clients with SSL

By default, the Azul Zulu Prime JDK connects to Optimizer Hub using SSL. If you installed without enabling SSL, you must use the `-XX:-OptHubUseSSL` flag to instruct the Azul Zulu Prime JDK to allow unsecured connections to Optimizer Hub.

NOTE	Before version 1.8.0 the flag was called `-XX:+/-CNCInsecure`. Because of this change, you need to review your settings.

If you attempt to connect to a Optimizer Hub that is running without SSL and do not specify the `-XX:-OptHubUseSSL` flag, you get the following error:

```
E1011 13:16:23.198074100        29 ssl_transport_security.cc:1446]
Handshake failed with fatal error SSL_ERROR_SSL:
error:1408F10B:SSL routines:ssl3_get_record:wrong version number.
```

To connect to Optimizer Hub using SSL, make sure the service certificate is trusted by the client server where you run Azul Zulu Prime JDK. This can be achieved by having the certificate signed by a publicly trusted certificate authority. If you have an internal CA trusted within the company infrastructure, make sure it is trusted.

The exact process depends on your OS distribution. Follow the instructions for your OS

distribution to register the certificate on your client server. For example, on Ubuntu-based distributions you run the following command:

```
sudo openssl x509 -in {path to cert.pem} -inform PEM -out
/usr/local/share/ca-certificates/cert.crt
sudo update-ca-certificates
```

Alternatively, you can explicitly instruct Azul Zulu Prime JDK to use and trust a specified certificate on the filesystem by using the `-XX:OptHubSSLRootsPath={path to cert.pem}` flag.

If certificate validation fails, your `.pem` file is missing or does not match the certificate that you uploaded to Optimizer Hub, you get the following error:

```
[1.856s][info][concomp] [gRPCEvent] read error!
[1.856s][info][concomp] [gRPC processing] BidiStreamWrapper is dying,
finishing stream 0x7fbec00180f0 with status: failed to connect to all
addresses (14)
```

Configuring ReadyNow Orchestrator

When you use ReadyNow Orchestrator, JVMs all write profile log candidates to unique profile names on the service. ReadyNow Orchestrator gathers all of the candidates for a profile name and decides which is th best candidate to serve to JVM clients requesting that profile name.

When considering what settings are set on the client versus on the service:

- Individual JVMs decide when ReadyNow Orchestrator should consider their profile log is a suitable candidate for sharing with other JVMs. They can also override server-side defaults for profile log nomination candidates and maximum profile log size.

- ReadyNow Orchestrator also controls the rules for where to store ReadyNow profile logs, when to clean up old logs, and service-wide defaults for profile log candidate nomination and maximum profile log size.

Duration Configuration

When you need to specify the duration in time a process takes, use the `PnDTnHnMn.nS` format, where n is the relevant days, hours, minutes or seconds part of the duration.

Configuring Clean Up of Old Profile Logs

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.

You can also configure ReadyNow Orchestrator to delete unused profile names completely after a [given duration](#) using the `readyNowOrchestrator.cleaner.keepUnrequestedProfileNamesFor` property in your `values-override.yaml`. For example, to keep unused profiles for 5 days, use the following:

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

ReadyNow Orchestrator triggers cleanup when you have used around 60% of the available space in your storage. If you are using a persistent volume to back up your storage, ReadyNow Orchestrator calculates the threshold for triggering clean up automatically. If you are using S3 or Azure Blob Storage, there is no reliable way for ReadyNow Orchestrator to calculate the size of the blob storage. You must therefore specify the size using the `readyNowOrchestrator.cleaner.externalPersistentStorageSoftLimit` property, which by default, is 10Gb.

Note that depending on your usage, ReadyNow Orchestrator's clean-up mechanism may not be able to keep the actual size of your stored profiles below the size of your storage. When you reach 90% usage, a warning is printed in the log of the gateway service.

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

Configuring Cross-Region Synchronization of Profiles

When you deploy a separate instance of Optimizer Hub in each region, you can configure Optimizer Hub to synchronize profile names between Optimizer Hub instances so that each instance contains at least one promoted profile for each profile name. For example, when deploying a new version of your program, you may first do a canary run in one of your regions. This canary run populates the first generation of the profile for the new version's profile name. Upon success, you want to launch a full fleet update in your other regions without doing a canary run in each region. By enabling cross-region synchronization, the profile that you wrote in the first region is available when you launch your fleet restarts in other regions.

To enable cross-region synchronization of profiles:

1. Enable the Management Gateway component by setting the following values in values-override.yaml:

   - Set `mgmtGateway.enabled` to true

   - If necessary, assign a different port than the default 8080 using `mgmtGateway.service.httpEndpoint.port`

2. Define the Optimizer Hub instances that you want to synchronize by entering a comma-separated list of URLs in `synchronization.peers`.

3. If necessary, adjust the number of profile generations that Optimizer Hub synchronizes. By default, Optimizer Hub synchronizes the first two generations of the profile.

See cross-region-sync-parameters for more information about the configuration options.

ReadyNow Orchestrator Defaults

Optimizer Hub admins can set the following global defaults for ReadyNow profiles in `values-override.yaml`:

Option	Description	Default
readyNowOrchestrator.debugInfoHistoryLength	Limit of rolling profile history entries	100
readyNowOrchestrator.cache.enabled	Enabling of caching the chunk content on the gateway	true
readyNowOrchestrator.cache.maxSizeBytes	The fixed size of chunk content cache on the gateway	500000000
readyNowOrchestrator.completedAfter	Time required after the last profile update, after which the profile is considered completed and updates are no longer possible, [duration specified in format `PnDTnHnMn.nS`]().	PT24H
readyNowOrchestrator.producers.continueRecordingOnPromotion	Flag to define if profiles must still be recorded after the maxGeneration has been reached. This can be used for debugging purposes.	false
readyNowOrchestrator.producers.maxConcurrentRecordings	The number of concurrent copies of a specific generation ReadyNow Orchestrator accepts before it tells other JVMs trying to write the same generation of the same profile name to stop	10
readyNowOrchestrator.producers.maxPromotableGeneration	Maximum number of generations ReadyNow Orchestrator accepts for a profile name. Note that here is no 'unlimited' value available	3

Option	Description	Default
readyNowOrchestrator.producers.maxProfileSize	Limit on the input profile size, in bytes. No limit by default	0
readyNowOrchestrator.cleaner.enabled	Enabling of automatic repository clean-up	true
readyNowOrchestrator.cleaner.externalPersistentStorageSoftLimit	When your storage is backed by `azure-blob` or `s3` storage, this determines the threshold for the blob data usage, at which ReadyNow Orchestrator initiates its cleanup process. When your storage is backed by a persistent storage volume, this threshold is calculated automatically.	10Gi
readyNowOrchestrator.cleaner.keepUnrequestedProfileNamesFor	Time limit after which the profile name gets removed if it was not requested within the given [duration specified in format `PnDTnHnMn.nS`](). By default, no limit is defined.	0
readyNowOrchestrator.promotion.minProfileSize	Minimal size (bytes) threshold for all generations unless per-generation flags are specified. Per-generation flags take precedence over the global setting, but the global might be used as a generation 0 setting in case it is not specified in the corresponding per-generation setting.	1000000

Option	Description	Default
readyNowOrchestrator.promotion.minProfileSizePerGeneration	Minimal size thresholds for each generation. In case a generation is missing in the list, it inherits a value from the previously specified generation or the global setting, if there is no previous generation specified. List of pair <generation>:<size>, separated by `\,`. For more information, check "Understanding ReadyNow Orchestrator Generations".	0:1000000\,1:10000000\,2:25000000\,3:50000000
readyNowOrchestrator.promotion.minProfileDuration	See previous. [Duration specified in format `PnDTnHnMn.nS`.](#)	PT2M
readyNowOrchestrator.promotion.minProfileDurationPerGeneration	See `minProfileSizePerGeneration`. List of pair <generation>:<duration>, separated by `\,`. The duration must be specified in the [format `PnDTnHnMn.nS`.](#) For more information, check "Understanding ReadyNow Orchestrator Generations".	0:PT2M\,1:PT15M\,2:PT30M\,3:PT60M

Option	Description	Default
readyNowOrchestrator.producers.maxSynchronizedGeneration	Defines the maximum number of profile generations to be synced from peers. Profiles with a higher generation are not synced from peers.	2

Configuring Prometheus and Grafana

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.

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.

NOTE	Monitoring Optimizer Hub assumes you have a Prometheus and Grafana available, or install one within your Kubernetes cluster.

Prometheus Configuration Instructions

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:

```yaml
# 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
  - 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
```

Grafana Configuration Instructions

Once Prometheus is available and collection data from the Optimizer Hub Components, a dashboard can be added. You can find a Grafana configuration file `cnc_dashboard.json` in opthub-install-1.9.5.zip.

This dashboard expects the following labels to be attached to all application metrics, refering 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`.

Sizing and Scaling your Optimizer Hub Installation

In order for the Optimizer Hub to perform the JIT compilation in time, you need to make sure the installation is sized correctly. You scale Optimizer Hub by specifying the total number of vCores you wish to allocate to the service. The Helm chart automatically sets the sizing of the individual Optimizer Hub components.

Scaling Overview

Scaling Optimizer Hub is controlled by how much capacity Optimizer Hub has to process compilation requests. This is controlled by the amount of vCores Optimizer Hub has been provisioned. Note that scaling is primarily a concern when discussing Cloud Native Compiler. ReadyNow Orchestrator consumes much fewer resources than Cloud Native Compiler and often never needs to scale beyond its minimum installation.

A critical metric to measure whether your Cloud Native Compiler is responding to compilation requests in time is the Time to Clear Optimization Backlog (TCOB). When you start a Java program, there is a burst of compilation activity as a large amount of optimization requests are put on the compilation queue. Eventually, the compiler catches up with the optimization backlog and all new compilation requests are started within 2 seconds of being put on the compilation queue. The TCOB is the measurement, for each individual JVM, of how long it took from the start of the compilation activity to when the optimization backlog is cleared.

To find the right amount of vCores to provision for a job, first determine an acceptable TCOB for your application. Different applications find different TCOBs acceptable depending on how many optimizations the program requests and how quickly you need to warm it up. As a starting point, set the amount of time you want to wait before the application is ready to accept requests as your target TCOB.

Perform a test run of a single JVM against your Cloud Native Compiler. In Grafana, check the Time to Clear Optimization Backlog and Compilations in Progress graphs.
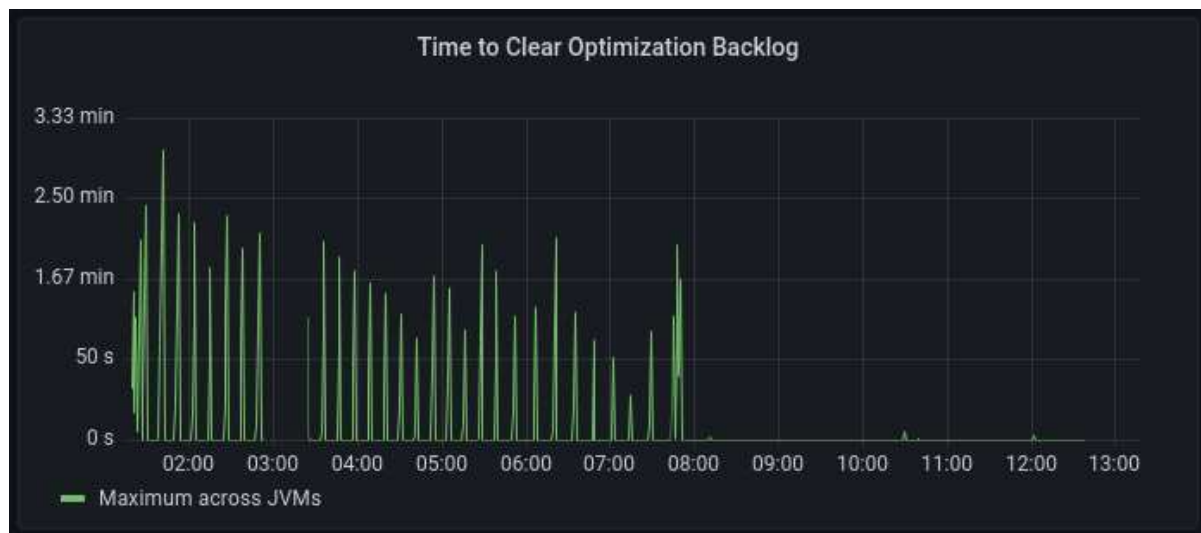
- If the maximum TCOB during your application's warmup is lower than your target, you can scale down the number of vCores provisioned for the job.

- If the maximum TCOB is higher than your target, check the Compilations in Progress metric in Grafana. This metric shows you actual compilations in progress versus Cloud Native Compiler capacity. If you are using the full capacity, add more vCores to the capacity.

You should also check the client JVM logs to see whether the JVM fallback-to-local-jit-compilation. JVMs switch to local compilation when Cloud Native Compiler becomes unresponsive or tells the JVM that it cannot handle any new requests. You can also see the number of local fallbacks in the Grafana dashboard.
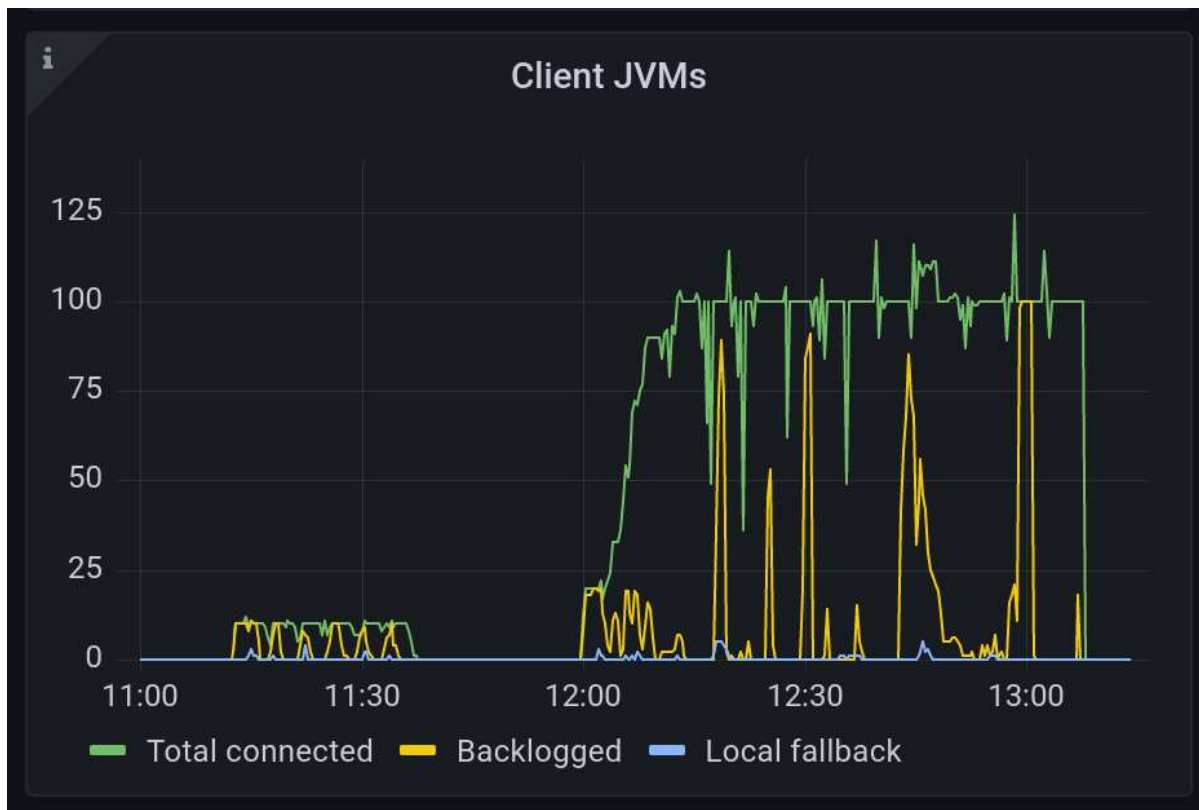
Configuring Capacity

Depending on your autoscaling settings, there are three variables you need to set:

```
simpleSizing:
  vCores: 32
  minVCores: 32
  maxVCores: 106
```

- `vCores` - Total number of vCores that are allocated. This does NOT include resources required by monitoring, if you enable it. The minimum amount of vCores for provisioning Cloud Native Compiler is 29.

- `minVCores` - The minimum amount of resources that are always allocated when autoscaling is enabled.

- `maxVCores` - The maximum amount of resources that are allocated when autoscaling is enabled.

Configuring Autoscaling

Autoscaling is enabled by default in the Helm chart. To disable autoscaling, add the

following to `values-override.yaml`:

```
autoscaler: false
```

If you use the Azul-provided "cluster config file", the pre-defined node groups for the `gateway`, `compile-broker` and `cache` components already contain instructions to work with Autoscaler. If the Autoscaler Node sees any unused nodes, it deletes them. If a replication controller, deployment, or replica set tries to start a container and cannot do it due to lack of resources, the Autoscaler Node knows which service is needed and adds this service to the Kubernetes cluster. For more information, see https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

In order to use HPA autoscaling, you need install the Metrics Server component in Kubernetes.

JVM Connections to Optimizer Hub

Connecting a JVM to Optimizer Hub

Whether you are using an Optimizer Hub instance to provide compilations, ReadyNow profiles, or both, the first step is creating a connection between the JVM and the Optimizer Hub instance. Ask your Optimizer Hub instance admin for the "host address and port of the Optimizer Hub host" and enter it in the `-XX:OptHubHost=host:port` JVM parameter flag.

Establishing a connection to Optimizer Hub does not force the JVM to fetch compilations from Optimizer Hub and not perform compilations locally by default. See the configuration flags on "Using the Cloud Native Compiler".

| NOTE | In some cases, you may also need to enter an additional `-XX:CNCEngineUploadAddress=host:port` flag. Ask your Optimizer Hub service admin if this is needed. |

Using the Cloud Native Compiler

You configure an Azul Zulu Prime Build of OpenJDK (Azul Zulu Prime JVM) to request compilations from Cloud Native Compiler by specifying the IP address of the service along with other command-line options. If the Cloud Native Compiler cannot respond to the compilation requests in time, the Azul Zulu Prime JVM switches to local JIT compilation until the service recovers.

Cloud Native Compiler JVM Options

NOTE	The minimum JVM options to request compilations from Cloud Native Compiler are `-XX:OptHubHost={host:port}` and `-XX:+CNCEnableRemoteCompiler`.

Command Line Option	Description	Default
-XX:OptHubHost={host:port}	Address where Optimizer Hub is listening. The default is `localhost:50051`. See "Configuring Optimizer Hub Host and Port" for instructions on determining the correct host and port.	
-XX:[+/-]CNCEnableRemoteCompiler	Allows usage of the remote compiler when Cloud Native Compiler has established a connection.	`false`
-XX:CNCEngineUploadAddress={host:port}	Address to upload the compiler engine. Only needed when your Optimizer Hub has non-standard ports. See custom-port.	

Command Line Option	Description	Default
-XX:[+/-]CNCAbortOnBadChannel	With this flag, the JVM crashes if it loses connection with a Cloud Native Compiler.	`false`
-XX:[+/-]OptHubUseSSL	Instructs the Azul Zulu Prime JVM to communicate directly with Optimizer Hub without using SSL. Use this option if you installed Optimizer Hub without SSL.	`true`
-XX:OptHubSSLRootsPath={path to cert.pem}	Instructs the Azul Zulu Prime JVM to use and trust a specified SSL certificate on the filesystem.	
-Xlog:[+/-]concomp	Display messages describing communication with Optimizer Hub.	`false`

Fallback to Local JIT Compilation

When you connect an Azul Zulu Prime JVM to a Cloud Native Compiler, the JVM attempts to fetch all JIT compilations from the service. If the Cloud Native Compiler cannot meet the JVM's requests in time, the JVM automatically falls back to performing optimizations on the client. Factors that can cause a Cloud Native Compiler to not meet optimization demand include:

- The service does not have the corresponding "Compiler Engine" installed. To force an Azul Zulu Prime JVM to fail when requesting optimizations from a Cloud Native Compiler that doesn't have the corresponding Compiler Engine installed, use the `-XX:+CNCAbortOnBadChannel` flag.

- The service is down or cannot be reached.

- The service does not have enough capacity to meet the optimization requests. If you have autoscaling enabled, this is often a temporary problem as new resources come online. See "Sizing and Scaling your Optimizer Hub Installation" for more info.

When an Azul Zulu Prime JVM switches to local JIT compilation, it keeps checking whether Cloud Native Compiler is ready to perform optimizations. Once Cloud Native Compilation is back online and healthy, the Azul Platform Prime JVM switches back to requesting optimizations from the service.

The following output in the JVM concomp log show when fallback to local JIT compilation is enabled and disabled:

```
[110,991s][info    ][concomp] [LocalFallback] local compilation queue
disabled
[111,018s][info    ][concomp] [LocalFallback] local compilation queue
enabled
```

Logging and SSL

To view compiler info and ensure that the JVM is correctly connecting to Optimizer Hub, use the `-Xlog:concomp` flag.

By default the Azul Zulu Prime JDK connects to Optimizer Hub using SSL. If you did not enable SSL during Optimizer Hub deployment, you must use the `-XX:-OptHubUseSSL` flag to instruct the Azul Zulu Prime JDK to connect without SSL.

If you attempt to connect to Optimizer Hub, running without SSL, and do not specify the `-XX:-OptHubUseSSL` flag, you get the following error (visible with the `-Xlog:concomp` flag):

```
E1011 13:16:23.198074100      29 ssl_transport_security.cc:1446]
Handshake failed with fatal error SSL_ERROR_SSL: error:1408F10B:SSL
routines:ssl3_get_record:wrong version number.
```

Registering a New Compiler Engine in Cloud Native Compiler

Since different versions of Azul Zulu Prime JVMs may require different compiled code, Optimizer Hub's Cloud Native Compiler must be able to produce different versions of compiled code simultaneously. You do not need to create a separate Optimizer Hub instance for each application or different Java version.

Cloud Native Compiler does not have its own compiler - it is just server-side infrastructure for running the JIT compiler that ships inside of Azul Zulu Prime Builds of OpenJDK. This compiler is uploaded to Cloud Native Compiler from the JVM in the form of a Compiler Engine.

Each version of Azul Zulu Prime JVM contains a signed Compiler Engine distributable. The JVM auto-uploads any missing compiler engine on startup. Compiler Engines are signed to prevent malicious versions of Compiler Engines from being installed.

If an Azul Zulu Prime JVM connects to a Cloud Native Compiler service that does not have the corresponding Compiler Engine installed, the JVM automatically switches to performing the optimizations on the client VM.

NOTE	Cloud Native Compiler does not keep any persistent record of compiler engines. If a JVM requests compilations from Cloud Native Compiler that does not have the corresponding compiler engine, the JVM switches to local JIT compilation and starts auto-uploading the compiler engine for future use.

Auto-Uploading Compiler Engines

For JVMs connecting to Cloud Native Compiler in the same Kubernetes cluster, or connecting to Cloud Native Compiler that is fronted by an external load-balancer, auto-uploading works with no additional configuration.

For JVMs connecting to Cloud Native Compiler in an external Kubernetes cluster with no external load-balancer, pass the IP address and port of Cloud Native Compiler's gateway service in the `--XX:CNCEngineUploadAddress` flag. See "Connecting a

JVM to a Cloud Native Compiler" for how to get the IP address of the `gateway` service. Make sure you use the port that is mapped to 8080 in the `gateway` service.

Inspecting the Installed Compiler Engines

Each Compiler Engine has a Compiler Engine ID. You can view all of the Compiler Engines that are installed on a Cloud Native Compiler by calling the `/compiler-engines` REST API on the `gateway` service's `8080` port when calling from inside the cluster or the external port that is mapped to `8080` when calling from outside the cluster.

Using ReadyNow Orchestrator

Using [ReadyNow](#) involves two distinct phases:

- Recording a good profile log that accurately captures the usage pattern you want to warm up. Recordings can be refined automatically through repetitive training cycles.

- Using the profile log as the input to newly started VMs.

Using the Optimizer Hub ReadyNow Orchestrator to record and serve profile logs, greatly simplifies the operational use of ReadyNow.

- There is no need to configure any local storage for writing the profile log.

- ReadyNow Orchestrator handles recording multiple profile candidates from multiple JVMs and promoting the best recorded profile log. You no longer need to manually prepare a profile and then distribute it before rolling out new versions of your code. Instead, you can generate the profile automatically in production as part of your fleet restart.

- ReadyNow Orchestrator monitors the optimization profiles of an entire fleet of JVMs rather than just one JVM, intelligently picking the best one.

Creating and Writing To a New Profile Name

You use ReadyNow Orchestrator by "creating a connection to the Optimizer Hub" and specifying the criteria for reading and writing profile logs. All of the necessary options

can be specified as command-line arguments to the Java process at the time of deployment.

The basic lifecycle of using ReadyNow profile logs is as follows:

- The JVM streams profile log output to ReadyNow Orchestrator, giving the output a unique profile name.

- Based on basic criteria specified in the command-line arguments, the JVM nominates the output profile log as a candidate for sharing with other JVMs.

- ReadyNow Orchestrator deals with candidate profile logs arriving from various JVMs that use the same profile name.

- Whenever the service receives a request for a profile log with a given profile name, it examines the candidates it has collected and serves up the best one. This can change over time as ReadyNow Orchestrator receives new and more complete profile log candidates.

- JVMs can request multiple generations of a profile log. Rather than starting with no input profile log and recording its output log based on the regular JIT profiling process, the JVM can take a profile log as the input and further refine the profiling information, recording its experience as a new generation of that profile log. If you need to minimize the chances of having any deoptimizations through the life of your Java program, it is sometimes beneficial to record several generations. ReadyNow Orchestrator always serves the newest generation for a profile name to JVMs. JVMs can cap the number of generations that they write out to avoid developing the profile forever.

ReadyNow Orchestrator JVM Options

The following options are available in Azul Prime when using ReadyNow Orchestrator with Optimizer Hub:

Command Line Option	Description	Default
-XX:OptHubHost={host:port}	Address where Optimizer Hub is listening. The default is `localhost:50051`. See "Connecting a JVM to Optimizer Hub" for instructions to determine the correct host and port.	null
-XX:+EnableRNO	Enables ReadyNow read and write to ReadyNow Orchestrator, using `ProfileName` as the name for the profile log.	
-XX:ProfileName={profilePath}	Name of the profile that the JVM both reads from and writes to. Use of this flag is equivalent to using `-XX:ProfileLogIn={profilePath}` `-XX:ProfileLogOut={profilePath}`, and is the preferred way to specify profile names when different input and output names are not needed. If prefixed with `opthub://`, `{profilePath}` is used as the profile name in ReadyNow Orchestrator. If not prefixed with `opthub://`, `{profilePath}` is interpreted as a file path on the JVM.	null

Command Line Option	Description	Default
-XX:ProfileLogOut={profilePath}	The ProfileLogOut enables Azul Zulu Prime JVM to record compilations from the current run. `{profilePath}` is the name of the profile that the JVM reads as input to ReadyNow. If prefixed with `opthub://`, `{profilePath}` is used as the profile name in ReadyNow Orchestrator. If not prefixed with `opthub://`, `{profilePath}` is interpreted as a file path on the JVM. **NOTE** For local ReadyNow you often have to specify different names for `ProfileLogIn` and `ProfileLogOut`. But for ReadyNow Orchestrator you must only use `ProfileName`.	null

Command Line Option	Description	Default
-XX:ProfileLogIn={profilePath}	The ProfileLogIn allows Azul Zulu Prime JVM to base its decisions on the information from a previous run. The current ProfileLogIn file information is read in its entirety - before Azul Zulu Prime JVM starts to create a new ProfileLogOut log. `{profilePath}` is the name of the profile that the JVM reads as input to ReadyNow. If prefixed with `opthub://`, `{profilePath}` is used as the profile name in ReadyNow Orchestrator. If not prefixed with `opthub://`, `{profilePath}` is interpreted as a file path on the JVM. NOTE For local ReadyNow you often have to specify different names for `ProfileLogIn` and `ProfileLogOut`. But for ReadyNow Orchestrator you must only use `ProfileName`.	null

71

Command Line Option	Description	Default
-XX:ProfileLogOutNominationMinSize	Indicate to server that the produced profile is eligible for promotion after specified amount of bytes recorded. `0` = any size eligible `-1` = never gets promoted	1M
-XX:ProfileLogOutNominationMinSizePerGeneration	Define minimum acceptable amount of bytes per generation which the profile size should reach to become eligible for promotion. List of pair <generation>:<size>, separated by `,`. For example: ``` 0:1000000,1:10000000,2:25000000,3:50000000 ``` For more information, check "Understanding ReadyNow Orchestrator Generations".	null

Command Line Option	Description	Default
-XX:ProfileLogOutNominationMinTimeSec	When used with ReadyNow Orchestrator, the minimum time, in seconds, a profile must record before ReadyNow Orchestrator can nominate it as a candidate. `0` = any duration eligible `-1` = never gets promoted	120
-XX:ProfileLogOutNominationMinTimeSecPerGeneration	When used with ReadyNow Orchestrator, the minimum time, in seconds, per generation during which the profile should be recorded in order to become eligible for promotion. List of pair <generation>:<duration>, separated by `,`. For example: ``` 0:100,2:150 ``` For more information, check "Understanding ReadyNow Orchestrator Generations".	null

Command Line Option	Description	Default
-XX:ProfileLogOutMaxNominatedGenerationCount	When used with ReadyNow Orchestrator, specifies the maximum generation of a profile that a VM nominates. This JVM command line parameter overrides the serverside default to configure ReadyNow Orchestrator. `0` = unlimited For more information, check "Understanding ReadyNow Orchestrator Generations".	0

Command Line Option	Description	Default
-XX:ProfileLogMaxSize={value in bytes}	Specifies the maximum size that a ReadyNow profile log is allowed to reach. Profiles get truncated at this size, regardless of whether the application has actually been completely warmed up. This JVM command line parameter overrides the serverside default to configure ReadyNow Orchestrator. It is recommended to either not set this size explicitly, or set it generously if required, for example: `-XX:ProfileLogMaxSize=1G` `0` = unlimited	0

Command Line Option	Description	Default
-XX:ProfileLogTimeLimitSeconds={value in seconds}	Instructs ReadyNow to stop adding to the profile log after a period of N seconds regardless of where the application has been completely warmed up. It is recommended to either not set this size explicitly, or set it generously if required. `0` = unlimited	0
-XX:ProfileLogDumpInputToFile={name}	Dumps input profile to the specified path. For debugging purposes only.	null
-XX:ProfileLogDumpOutputToFile={name}	Dumps output profile to the specified path. For debugging purposes only.	null
-XX:RNOConnectionTimeoutMillis	Timeout on establishing remote connection and timeout on interval between downloading two chunks. Specified in milliseconds.	5000

Command Line Option	Description	Default
-XX:RNOProfileFallbackInput	**Experimental feature**. Local filesystem path which gets used in case no profile data is downloaded. E.g., in case of a missing connection or the requested profile name doesn't exist on the server.	null
-XX:ProfileLogOutVerbose	Enables logging of verbose, optional tracing information in `-XX:ProfileLogOut`	true

Substitution Macros

The profile name is the central organizing attribute that ReadyNow Orchestrator uses to group together profile logs. ReadyNow Orchestrator regards all candidates it receives that contain the same profile name as being for the same application, with no further knowledge of what code was actually runs. This poses the danger of accidentally using the same profile name for two different applications. For example, if a user copies and pastes the command-line arguments, including the profile name, from a production application and uses it to run HelloWorld, the HelloWorld profile could, in some cases, replace your valid production application profile.

To avoid this danger, you can use substitution macros in your profile name to limit the likelihood of profile name clashes between different applications. Each macro unfolds to a 4-byte hash string taken from a particular plain-text string corresponding to a property:

Macro	Description
%classpathhash	Hashed user-defined Java class path string

Macro	Description
%vmargshash	Hashed JVM arguments string
%vmflagshash	Hashed JVM flags string
%cmdlinehash	Hashed string containing all plain-text values from above macros. Input values are concatenated to one string: Java class path string + JVM arguments string + JVM flags string. Afterwards, 4-bytes hash is applied to concatenated result.
%jdkver	Hashed JDK version number converted to string
%jvmver	Hashed JVM version number converted to string
%prop={PROPERTY}%	Substition macro defining the profile log name. This gets replaced with the value of the corresponding Java system property. Provide these properties to the JVM on startup with `-Dprop=value`. For example: ``` -Dmyprofilename=test-profileout \ -XX:ProfileLogOut=opthub://%prop=myprofilename% ```

Using a Previous Profile as the Basis of a New Profile Recording

When you're deploying version 3 of MyApp, you often have a valid profile for version 2. In most cases, you change a small portion of your code between versions and most of the previous profile is still valid for your new version. When you feed in the previous version of the profile as input to recording the new version of the profile, you can in most cases eliminate the need to do multiple training iterations.

Using our above example, perform one run of the full ten minutes in a canary with the following settings:

```
java -XX:OptHubHost={host:port} \
    -XX:+EnableRNO \
    -XX:ProfileLogIn=opthub://MyApp-v2 \
    -XX:ProfileLogOut=opthub://MyApp-v3 \
    -XX:ProfileLogTimeLimitSeconds=600 \
    -XX:ProfileLogOutMaxNominatedGenerationCount=1 \
    -jar myapp.jar
```

To restart the rest of your fleet with the following settings:

```
java -XX:OptHubHost={host:port} \
    -XX:+EnableRNO \
    -XX:ProfileName=opthub://MyApp-v3 \
    -XX:ProfileLogTimeLimitSeconds=600 \
    -XX:ProfileLogOutMaxNominatedGenerationCount=1 \
    -jar myapp.jar
```

Understanding ReadyNow Orchestrator Generations

When using ReadyNow, you get the best results if you perform several training runs of your application to generate an optimal profile. For example, to generate a good profile for an application instance called MyApp-v1.5, you perform three training runs of your application to record three generations of the ReadyNow profile log. For each training run, you read in the generation recorded by the last training run as the input profile for the current training run.
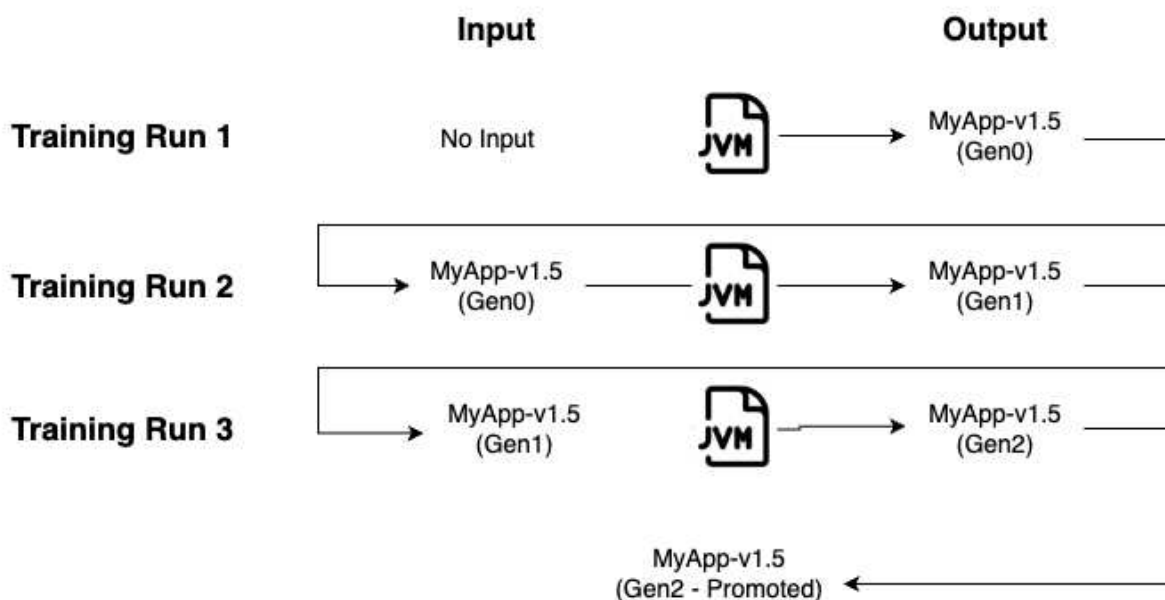
You want to make sure that the output of each training run meets minimum criteria to be promoted as the input for the next level. These promotion criteria can be:

- The duration (time) of the training run

- The size of the candidate profile log

The ReadyNow Orchestrator feature in Optimizer Hub automatically takes care of creating a promoted profile as you run your application. Users specify the promotion criteria for each version of their application and a unique ProfileName as Java command line parameters. Administrators can also specify server-side global promotion criteria that must also be met. When deploying a new version of an application, ReadyNow Orchestrator automatically collects candidates from many JVMs running the same ProfileName and performs the training runs to generate the best promoted profile.

Configuring Generations

ReadyNow Orchestrator allows you to set different minimum size and recording durations for different generations of your profiles. Often you want to promote the first generation of your profile as quickly as possible so new JVMs are not starting with nothing, but you want your second generation to record for a longer time before promotion, so it is more complete.

You can use configuration settings for both readynow-orchestrator-defaults itself and the readynow-orchestrator-jvm-options to change this behavior if the default values don't deliver the desired result.

Basic Profile Recording with Default Generations

In its most basic form, you let the defaults do all the work. By default, ReadyNow Orchestrator nominates profile logs after three full generations and doesn't place a limit on log size. Suppose you want to record a new profile while deploying code to a fleet running in production. Run with the following options:

```
java -XX:OptHubHost={host:port} \
    -XX:+EnableRNO \
    -XX:ProfileName=opthub://MyApp-v3 \
    -jar myapp.jar
```

In this case, all JVMs nominate their logs based on the defaults (e.g. 2 minutes for the first generation, 10 minutes for the second generation, 120 minutes for the third generation) and keep recording until the JVM shuts down. For best results, do a test run in a canary instance for at least two minutes and if possible a full ten minutes. This creates generation 1 of your profile. Then restart your fleet as normal. As JVMs start up, they receive a profile from ReadyNow Orchestrator and check the generation number. If that number is less than the server-side default maximum of 3, the JVM writes out the next generation of the profile. Once there is a valid generation 3 of the profile on ReadyNow Orchestrator, none of the JVMs write any more output.

You can overrule the server-side defaults, by providing extra options, for example:

```
java -XX:OptHubHost={host:port} \
    -XX:+EnableRNO \
    -XX:ProfileName=opthub://MyApp-v3 \
    -XX:ProfileLogOutNominationMinSizePerGeneration=0:1000000
\,1:10000000\,2:25000000\,3:50000000 \
    -XX:ProfileLogOutNominationMinTimeSec=0:PT2M\,1:PT15M\,2:PT30M
\,3:PT60M \
    -jar myapp.jar
```

Capping Profile Log Recording and Maximum Generations

We can make our example above more complex:

- After 10 minutes you want to stop recording.

- You want to record two generations of the profile.

Start your JVM with the following parameters:

```
java -XX:OptHubHost={host:port} \
    -XX:+EnableRNO \
    -XX:ProfileName=opthub://MyApp-v3 \
    -XX:ProfileLogTimeLimitSeconds=600 \
    -XX:ProfileLogOutMaxNominatedGenerationCount=2 \
    -jar myapp.jar
```

Priority of Generation Settings

Please take the default values into account when you define your own generation settings as these can be overruled by other default settings. Let's look at an example:

- You define `-XX:ProfileLogOutNominationMinTimeSec=900`, but don't change other settings.

- The server-side default for the promotion of different generations, specifies the following default `0:PT2M\,1:PT15M\,2:PT30M\,3:PT60M` for `readyNowOrchestrator.promotion.minProfileDurationPerGeneration`.

- As a result, the 2nd and 3rd generation aren't promoted after 900 seconds, but after 30 and 60 minutes as specified in the defaults.

When you want to overrule the default settings, make sure to specify all appropriate options.

Detailed Information

Optimizer Hub API

Optimizer Hub provides an administration API with the following methods.

ReadyNow Orchestrator Admin API

These methods are available on `{MANAGEMENT_GATEWAY_IP}:{SERVICE_PORT}/rno/…` and can be accessed without authentication. The service port typically is 8080, but can be different based on the used configuration. For security reasons, by default, the API is not exposed outside the cluster.

Configure the API Endpoint

Apply the required changes in this section of the cluster configuration.

```
gateway:
  service:
    type: "NodePort"
    httpEndpoint:
      enabled: false
      port: 8080
```

Overview of the API Methods

Method	Url	Description
GET	/rno/names	Returns a list of all profile names with summary information. The request can optionally be extended with a date range filter (`from`, `to`) in the format `YYYY-MM-DD`. For example: `from=2024-04-18&to=2024-04-23`. Both these parameters are optional, so you can filter profiles by only one of them. The data is returned as a JSON array of profile names in the following example format: ```{\n "name": "name-of-profile",\n "numberOfProfiles": 1,\n "numberOfChunks": 88,\n "totalDiskSize": 17273029,\n "lastAccess": "2024-04-18T16:06:27.842Z",\n "cncEnabled": true\n}``` You can use the value provided in `name` to retrieve more info with the following APIs.

Method	Url	Description
GET	/rno/names/{name}	Returns summary information for the requested profile name.
DELETE	/rno/names/{name}	Deletes given profile name and all profiles belonging to it.
GET	/rno/names/{name}/profiles	Returns summary information for all of the profiles within a given profile name. Use the `?status=promoted` query parameter to see only the promoted profile.
GET	/rno/names/{name}/profiles/{Id}	Returns summary information for a specific profile.
DELETE	/rno/names/{name}/profiles/{Id}	Deletes a specific profile.

Method	Url	Description
GET	/rno/names/{name}/export	Exports all of the profiles in a specific profile name. Each profile's directory has the Id of the VM that created it. The promoted profile, meaning the profile that PLS sends to new clients requesting the profile name, is stored in `profilePromoted.json`. The README has instructions for unifying the profile chunks into a single profile file that can be used as a local input to ReadyNow. You can see the iteration of a given profile in the profileIteration property in the profile's `profileInfo.json` file.

| | NOTE | The profile export fails if the resulting data stream is larger than 2GB. If this happens, consider just exporting the promoted profile using the `?status=promoted` query parameter. |

Method	Url	Description
POST	/rno/names/{name}/import	Imports a profile log to this instance of Optimizer Hub. This API is mostly used for moving a promoted profile from one Optimizer Hub instance to another. The uploaded file should be a zip archive in the format produced by the `/profile/export` API. Do not rename directories in the profile structure or edit the profile metadata. **NOTE** If the profile name already exists, the import fails.
GET	/rno/profiles/{id}/content	Returns a profile by the specified id. You can use the returned profile as an input for Prime JVM for the ReadyNow ProfileLogIn flag value. For example: ```curl {endpoint}:{port}/rno/profiles/{id}/content > {RETURNED_PROFILE}\n\n-XX:ProfileLogIn={RETURNED_PROFILE}```
GET	/rno/statistics	Returns service-wide statistics for this instance of Cloud Native Compiler.

Monitoring Optimizer Hub

You can monitor your Optimizer Hub using the standard Kubernetes monitoring tools (Prometheus and Grafana) and through log files.

Using Prometheus and Grafana

The Optimizer Hub components are already configured to expose key metrics for scraping by Prometheus. Follow "Configuring Prometheus and Grafana" to set up these monitoring tools and check "Using the Grafana Dashboard" for more info about the different sections of the dashboard.

Retrieving Optimizer Hub Logs

All Optimizer Hub components, including third-party ones, log some information to `stdout`. These logs are very important for diagnosing problems.

You can extract individual logs with the following command:

```
kubectl -n my-opthub logs {pod}
```

However by default Kubernetes keeps only the last 10 MB of logs for every container, which means that in a cluster under load the important diagnostic information can be quickly overwritten by subsequent logs.

You should configure log aggregation from all Optimizer Hub components, so that logs are moved to some persistent storage and then extracted when some issue needs to be analyzed. You can use any log aggregation One suggested way is to use Loki. You can query the Loki logs using the logcli tool.

Here are some common commands you can run to retrieve logs:

- Find out host and port where Loki is listening

```
export LOKI_ADDR=http://{ip-adress}:{port}
```

- Get logs of all pods in the selected namespace

```
logcli query --since 24h --forward --limit=10000 '{namespace="zvm-dev-3606"}'
```

- Get logs of a single application in the selected namespace

```
logcli query --since 24h --forward --limit=10000 '{namespace="zvm-
dev-3606" app="compile-broker"}'
```

- Get logs of a single pod in the selected namespace

```
logcli query --since 24h --forward --limit=10000 '{namespace="zvm-
dev-3606",pod="compile-broker-5fd956f44f-d5hb2"}'
```

Extracting Compilation Artifacts

Optimizer Hub uploads compiler engine logs to the blob storage. By default, only logs from failed compilations are uploaded.

You can retrieve the logs from your blob storage, which uses the directory structure `<compilationId>/<artifactName>`. The `<compilationId>` starts with the `VM-Id` which you can find in `connected-compiler-%p.log`:

```
# Log command-line option
-Xlog:concomp=info:file=connected-compiler-%p.log::filesize
=500M:filecount=20

# Example:
[0.647s][info ][concomp] [ConnectedCompiler] received new VM-Id:
4f762530-8389-4ae9-b64a-69b1adacccf2
```
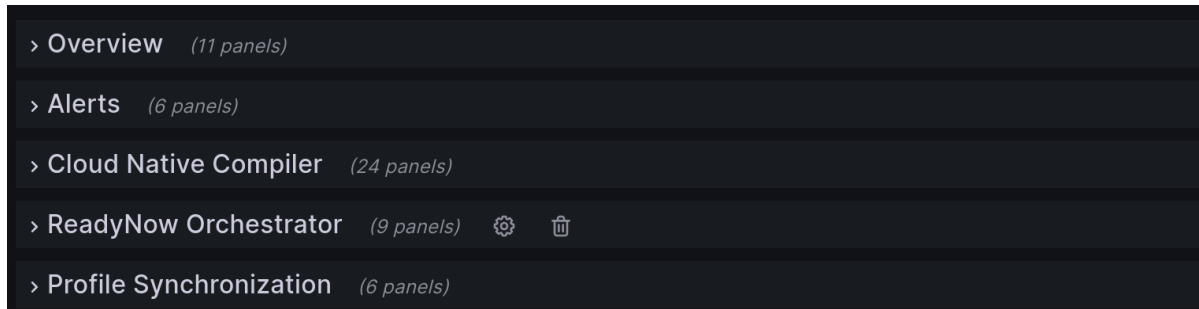
Note About `gw-proxy` Metrics

The `gw-proxy` component in Optimizer Hub uses, by default, `/stats/prometheus` as target HTTP endpoint to provide metrics. Most other Optimizer Hub components use `/q/metrics`. If you make manual changes in the configuration of the metrics for individual Kubernetes Deployments in the Optimizer Hub installation, make sure that you don't use the `/q/metrics` for the `gw-proxy` deployment. Doing so would lead to confusion when metrics are processed.

Using the Grafana Dashboard

A Grafana dashboard is available after it has been "configured", to understand how your Optimizer Hub instance is performing. This dashboard is divided into several sections. The most important sections from user-perspective are described here. The other sections are more oriented towards maintaining and troubleshooting of the installation.

> Overview   *(11 panels)*
> Alerts   *(6 panels)*
> Cloud Native Compiler   *(24 panels)*
> ReadyNow Orchestrator   *(9 panels)*   ⚙   🗑
> Profile Synchronization   *(6 panels)*

Overview

Provides a high-level view of the Optimizer Hub instance:

- The number of running Optimizer Hub components.

- The number of connected JVMs, with basic overview (in local fallback, backlogged,…)

- Basic metrics about compilations, with "Time to clear optimization backlog" as the most important value to monitor.
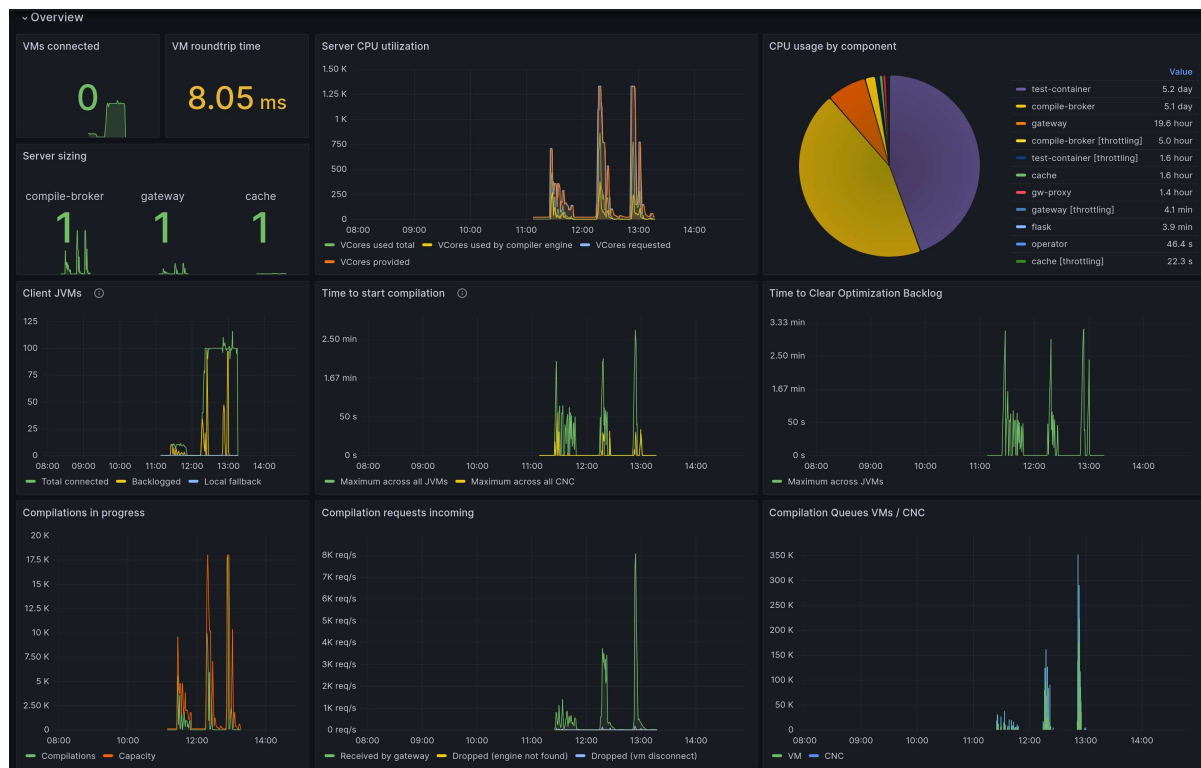
Alerts

Basic monitoring of problems in the Optimizer Hub cluster.
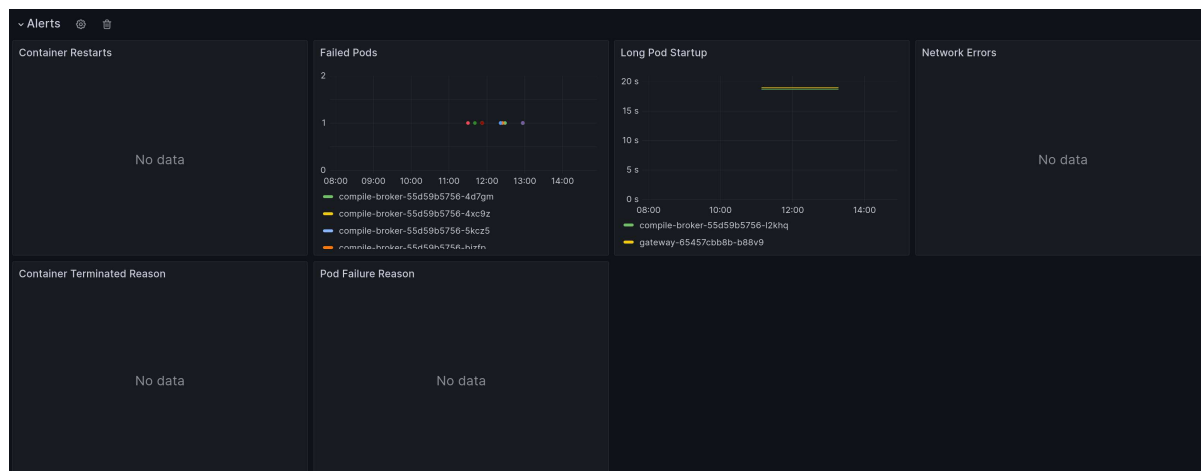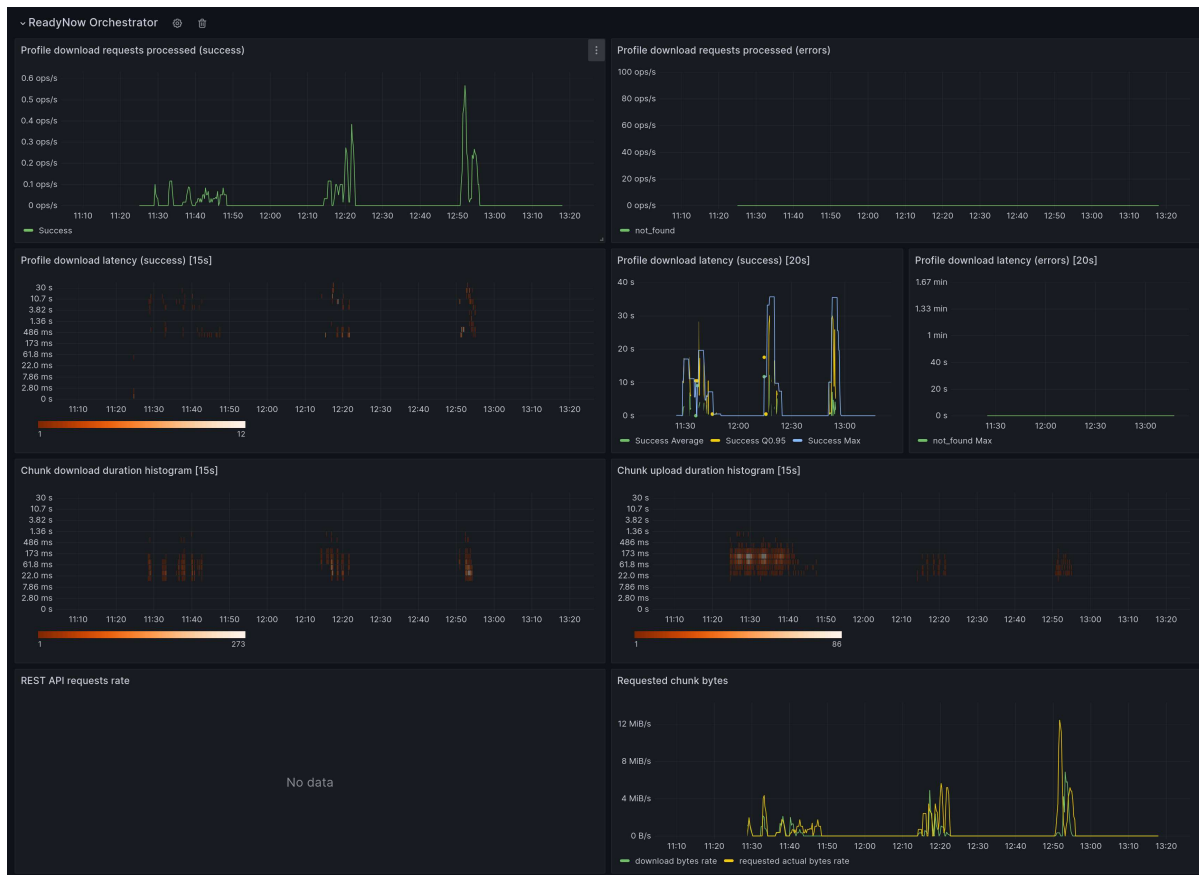
Cloud Native Compiler

Monitoring of the Cloud Native Compiler feature. This is applicable for JVMs using the Cloud Native Compiler.

ReadyNow Orchestrator

Monitoring of the ReadyNow Orchestrator feature. This is applicable for JVMs using

Optimizer Hub ReadyNow Orchestrator.

Profile Synchronization

Applicable when you have multiple Optimizer Hub clusters and profiles are synchronized between them. This section provides the following info:

- Profile sync latency

- Profile sync rate: speed of value changes of the metric

- Profile sync tasks finished

- Synced bytes total: sum of the synced bytes

- Sync task duration: the duration of each task

- Number of synced profiles rate
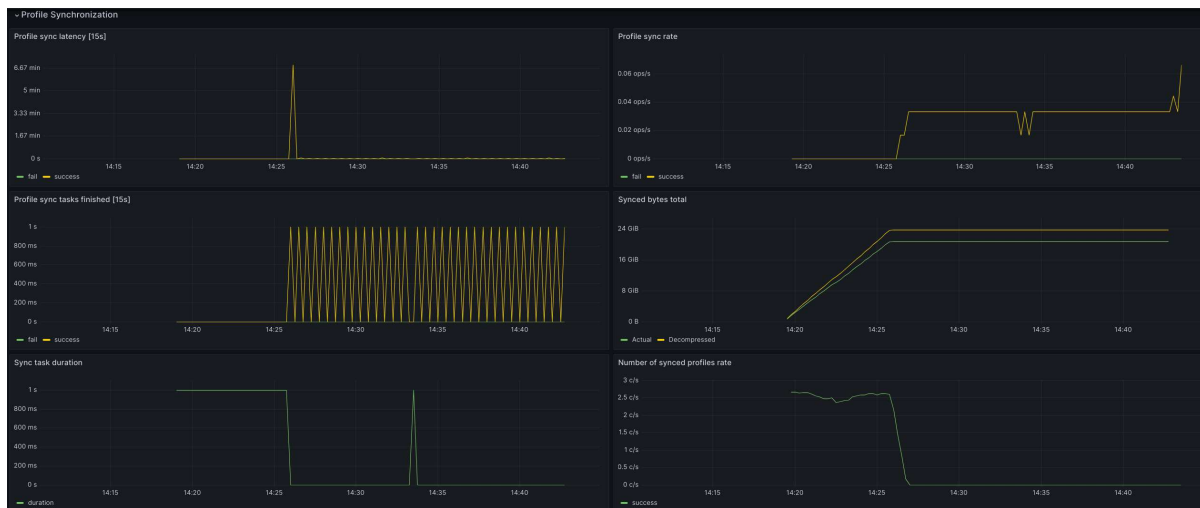
Troubleshooting Optimizer Hub

This page shows how to troubleshoot a misbehaving Optimizer Hub and any Azul Zulu Prime Builds of OpenJDK (Azul Zulu Prime JVM) instances using Optimizer Hub.

Client VM Troubleshooting

My application running in a Cloud Native Compiler-enabled VM shows worse performance than usually. What can I do?

1. Double-check VM arguments. Ensure that VM is started with `-XX:OptHubHost=` parameter pointing to the address of the Optimizer Hub gateway.

   See "Connecting a JVM to a Cloud Native Compiler" for more details on Optimizer Hub-related VM parameters and "Installing Optimizer Hub" for finding out the gateway address.

2. Enable Optimizer Hub logging in VM using `-Xlog:concomp` parameter and look for log messages that show the JVM connecting to and disconnecting from Optimizer Hub.

   - If the log says that the VM fails to connect to the service, check that the service is up and running, check the network connectivity between JVM and service, and check the value of `-XX:OptHubHost=`.

   - If the log says that VM disconnects from the service soon after connecting, the log

should also give the reason for disconnecting. The most frequent reason for such disconnects is a missing Compiler Engine on the service, indicated by the `FAILED_PRECONDITION` error code and message `Compiler engine … not found`. See "Registering a New Compiler Engine" for more information.

- If the connection between the VM and service is established and does not break, then proceed to item #3.

3. Collect VM GC log, open it in GCLA and see top-tier compilation statistics. Top-tier compilation stats can also be seen in VM compilation log (`-XX:+PrintCompilation`).

- If stats show high top-tier compilation failure ratio, then it's time to troubleshoot Cloud Native Compiler.

- Write down the VM ID seen in the VM concomp log, it can be used to filter service events related to this particular VM.

   You can find the VM ID in `connected-compiler-%p.log`:

   ```
   # Log command-line option
   -Xlog:concomp=info:file=connected-compiler-%p.log::filesize
   =500M:filecount=20

   # Example:
   [0.647s][info ][concomp] [ConnectedCompiler] received new VM-Id:
   4f762530-8389-4ae9-b64a-69b1adacccf2
   ```

- Proceed to [Cloud Native Compiler Server Troubleshooting](#).

4. Use the TTCOB metric to research possible problems.

   An overloaded client (the JVM) can cause worse performance of Cloud Native Compiler. This could be seen as a too high TTCOB metric. One example of such overload is CPU saturation on JVM side. This can cause smaller amounts of compilations being sent to Cloud Native Compiler but also a worse performance of Cloud Native Compiler compilation because an overloaded JVM affects the communication between the CNC Compiler and JVM itself.

- If TTCOB is over the threshold:

  - Look at the "Compilations in progress" chart.

  - If "Compilations" value hits the capacity, then the server is the bottleneck and should be scaled.

  - Otherwise the bottleneck is related to the per-VM limit on concurrent compilations. It should be increased. Scaling server without increasing that per-VM limit doesn't help.

- If TTCOB is below threshold:

  - How much below threshold is it?

  - If there is a gap between the actual TTCOB and the threshold, then Optimizer Hub can be downscaled proportionally to the gap.

  - Otherwise relax and don't touch anything.

5. If scaling compile-brokers doesn't improve TTCOB, the culprit may be the cache.

   A typical symptom is cache CPU usage hitting the ceiling, depending on the workload. An example can be seen in this graph:
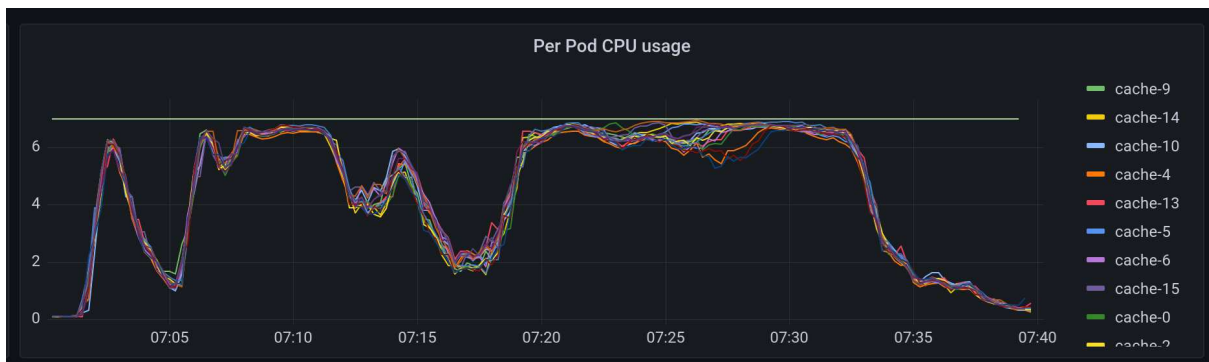
   If that's the case, one can modify simple sizing relationships to have more caches. This is the relevant section in the values.yaml:

```
simpleSizing:
  relationships:
    brokersPerGateway: 30
    brokersPerCache: 20
```

Settings brokersPerCache to a lower value (e.g. 15) results in having more cache instances relative to compile-brokers.

I see occasional "compiler timeout" errors in service logs and/or grafana dashboard. What's that?

Every compilation on Cloud Native Compiler has a time limit. By default it's 500 seconds.

- If that limit is exceeded, the first thing to check is network latency between VM and Cloud Native Compiler using `ping {opthub_host}`. Latency should not exceed single-digit milliseconds. If the latency is higher, CNC can't deliver its best performance. Make sure to locate VMs close enough to CNC.

- You can use the "VM rountrip" widget in the Grafana dashboard to detect if this limit is exceeded.

- In rare cases there are very large compilations that actually require that long. If that's the case, compilation timeout can be changed by adding `-Dcompiler.timeout={N}` flag to compile-broker, where `{N}` is the number in seconds.

My application running in a Optimizer Hub-enabled VM behaves incorrectly or crashes. What can I do?

1. Collect all VM logs and the `hs_err*` file and send it to Azul for analysis.

2. Run the application without the `-XX:OptHubHost` flag to verify that the problem is specific to connecting to Optimizer Hub.

I sometimes see entries about failed compilations because of "ConnectedCompiler is not yet ready", but I see it is compiling fine. Is that ok?

This may happen when running with SSL enabled. The VM keeps an open connection to the service, but sometimes the connection can be reset or re-established. It may happen that the VM tries to send a compilation request in the very moment. With SSL, the VM and the service need to do a handshake to make sure the connection is trusted. It is very quick, but it is possible the VM hits this small window. It is harmless as the compilation is resubmitted the next moment.

Cloud Native Compiler Troubleshooting

JVM compilation log shows that top-tier compilations are started, but never finished. What can I do?

This can be caused by one of these reasons:

- No compile-broker pods are running in the Optimizer Hub cluster. Make sure that at least one compile-broker is up and running.

- Cloud Native Compiler has too many compilation requests enqueued due to too many VMs connected and it takes too long to provide compiled code. To confirm, check the "Compilation Queues" chart in Grafana. Increase the number of compile-broker replicas.

I see occasional "vm unreachable" in service logs and/or grafana dashboard. What's that?

This is caused by the service's inability to receive some information necessary for the compilation from the JVM. It usually happens when the JVM disconnects from the service for any reason, e.g. JVM termination or a network error. It's harmless. The service just skips the compilation and proceeds to the next one.

ReadyNow Orchestrator Troubleshooting

ReadyNow profile reading timed-out with pre-main exceeding 60 seconds.

In case of a service misconfiguration with the Optimizer Hub not being deployed, and `compilation.limit.per.vm` setting being set to a value higher than `0`, Prime may

attempt to use the service for compilations to no avail. It might take some time for Prime to automatically switch to the local Falcon compiler. This can severely impact the ability of ReadyNow to pre-compile methods before the application load is started thus limiting the overall effect of ReadyNow.

Known Issues

- VM crashes when there is not enough memory available on the system. The exact amount of memory needed depends on the environment and the application. If you see VM crashing, please try freeing memory (e.g. killing some memory-hungry processes) or moving to a machine with more memory.