Optimizer Hub Documentation

Table of Contents

About Optimizer Hub **1**

Interaction Between Optimizer Hub and JVMs 1

About Cloud Native Compiler 2

JIT Optimization 2

Falcon JIT with CNC 4

About ReadyNow Orchestrator 4

Key Strengths of ReadyNow Orchestrator 5

Video Introduction of ReadyNow Orchestrator 5

Optimizer Hub Architecture Overview 5

Architecture Overview 5

Recommended Pods and Nodes Sizing 7

High Availability of Optimizer Hub 8

Optimizer Hub Release Notes **9**

Optimizer Hub 1.11.2 9

New Features 9

Bug Fixes 9

Optimizer Hub 1.11.1 10

New Features 10

Bug Fixes 10

CVE Fixes 10

Optimizer Hub 1.11.0 10

New Features 11

Bug Fixes 12

Optimizer Hub 1.10.2 12

New Features 12

CVE Fixes 12

Optimizer Hub 1.10.1 13

New Features . . . 13
Optimizer Hub 1.10.0 . . . 13
New Features . . . 13
Bug Fixes . . . 14
Known Issues . . . 15
Optimizer Hub 1.9.5 . . . 15
New Features . . . 15
Default Configuration Changes . . . 15
Optimizer Hub 1.9.4 . . . 15
New Features . . . 15
Bug Fixes . . . 16
Known Issue . . . 16
Optimizer Hub 1.9.3 . . . 16
New Features . . . 16
Bug Fixes . . . 17
Known Issue . . . 17
Optimizer Hub 1.9.2 . . . 17
New Features . . . 17
Known Issue . . . 17
Optimizer Hub 1.9.1 . . . 18
New Features . . . 18
Optimizer Hub 1.9.0 . . . 18
New Features . . . 19
Bug Fixes . . . 20
Optimizer Hub 1.8.2 . . . 20
New Features . . . 20
Optimizer Hub 1.8.1 . . . 20
New Features . . . 20

Known Issues 20
Optimizer Hub 1.8.0 20
New Features 21
Known Issues 22
Cloud Native Compiler 1.7.1 22
New Features 22
Cloud Native Compiler 1.7.0 23
New Features 23
Cloud Native Compiler 1.6.3 24
New Feature 24
Cloud Native Compiler 1.6.2 24
New Features 24
Upgrade 24
Cloud Native Compiler 1.6.1 24
New Features 24
Bug Fixes 24
Known Issues 25
Cloud Native Compiler 1.6.0 25
New Features 25
Bug Fixes 25
Known Issues 25
Cloud Native Compiler 1.5.0 26
New Features 26
Known Issues 26
Cloud Native Compiler 1.4.0 26
New Features 26
Known Issues 26
Cloud Native Compiler 1.3.0 26

New Features . . . 26
Known Issues . . . 27
Cloud Native Compiler 1.2.0 . . . 27
New Features . . . 27
Cloud Native Compiler 1.1.0 . . . 27
New Features . . . 27
Known Issues . . . 27
Cloud Native Compiler 1.0.0 . . . 28
New Features . . . 28
Azul Platform Core Third Party Licenses . . . 28
Optimizer Hub Installation Instructions . . . 29
Installing Optimizer Hub . . . 29
Supported Platforms . . . 29
Load Balancing . . . 29
Supported Kubernetes Environments . . . 30
Using Internal Docker Registry . . . 30
Installing Optimizer Hub on Kubernetes . . . 31
Prerequisites . . . 31
Installing Optimizer Hub . . . 31
Cleaning Up . . . 34
Installing Optimizer Hub on AWS Elastic Kubernetes Service . . . 34
Configuring AWS S3 Storage . . . 34
Installing Optimizer Hub on EKS . . . 37
Setting Up an External Load Balancer . . . 39
Cleaning Up . . . 39
Installing Optimizer Hub on Microsoft Azure . . . 40
Configuring Azure Blob Storage . . . 40
Installing Optimizer Hub on Google Cloud . . . 41

Configuring GCP Blob Storage . . . 41
Installing on an S3 Compatible Environment . . . 42
Configuring Storage . . . 43
Configuring S3 Authentication . . . 43
Installing Optimizer Hub on MicroK8s . . . 43
Installing MicroK8s . . . 43
Installing Optimizer Hub . . . 44
Upgrade Optimizer Hub on MicroK8s . . . 47
Uninstalling Optimizer Hub from MicroK8s . . . 48
Installing Optimizer Hub on Minikube . . . 49
Installing Minikube . . . 49
Installing Optimizer Hub . . . 49
Uninstalling Optimizer Hub from Minikube . . . 52
Upgrading Optimizer Hub . . . 52
Rolling Back to a Previous Version . . . 53
Configuring Optimizer Hub . . . 53
Optimizer Hub Generic Defaults . . . 53
Management Gateway Parameters . . . 53
Cross-Region Sync Parameters . . . 53
Blob Storage Auto Cleanup Parameters . . . 54
Simple Sizing Parameters . . . 54
SSL Parameters . . . 54
Storage Parameters . . . 54
Using Externally Defined Secrets . . . 54
Defining Your Secrets . . . 55
Configuring the Active Optimizer Hub Services . . . 56
Install Only ReadyNow Orchestrator . . . 57
Disabling Cloud Native Compiler on a Full Optimizer Hub Installation . . . 57

Configuring Optimizer Hub Host 57
Host for Single Optimizer Hub service 57
Host for High Availability and Failover 59
Configuring ReadyNow Orchestrator 60
Duration Configuration 60
Configuring Cross-Region Synchronization of Profiles. 60
ReadyNow Orchestrator Defaults 61
Configuring Blob Storage Auto Cleanup 63
Code Cache Cleanup. 64
ReadyNow Profile Log Cleanup 64
Configuring Optimizer Hub SSL Authentication 66
SSL Configuration in Optimizer Hub 66
SSL Configuration for Clients. 68
Configuring Prometheus and Grafana 69
Prometheus Configuration Instructions 70
Grafana Configuration Instructions. 71
Sizing and Scaling your Optimizer Hub Installation 72
Service Scaling 72
How Optimizer Hub Scales 73
Scaling API. 75
JVM Connections to Optimizer Hub. 75
Connecting a JVM to Optimizer Hub 75
Using Cloud Native Compiler 76
Cloud Native Compiler JVM Options 76
Fallback to Local JIT Compilation. 77
Logging and SSL 78
Using ReadyNow Orchestrator 78
Advantages of ReadyNow Orchestrator 78

Creating and Writing To a New Profile Name. . . . 79
ReadyNow Orchestrator JVM Options 80
Registering a New Compiler Engine in Cloud Native Compiler. . . . 87
Auto-Uploading Compiler Engines. . . . 88
Understanding ReadyNow Orchestrator Generations 88
Configuring Generations. . . . 89
Basic Profile Recording with Default Generations. . . . 90
Capping Profile Log Recording and Maximum Generations 91
Priority of Generation Settings. . . . 91
Detailed Information. . . . 91
Optimizer Hub API 91
API Methods 92
Monitoring Optimizer Hub. . . . 92
Using Prometheus and Grafana 92
Retrieving Optimizer Hub Logs 92
Extracting Compilation Artifacts. . . . 93
Note About `gw-proxy` Metrics 94
Using the Grafana Dashboard. . . . 94
Overview. . . . 94
Alerts 95
Cloud Native Compiler 95
ReadyNow Orchestrator 97
Profile Synchronization. . . . 97
Troubleshooting Optimizer Hub 98
Client VM Troubleshooting. . . . 98
Cloud Native Compiler Troubleshooting. . . . 103
ReadyNow Orchestrator Troubleshooting 104
Known Issues 104

About Optimizer Hub

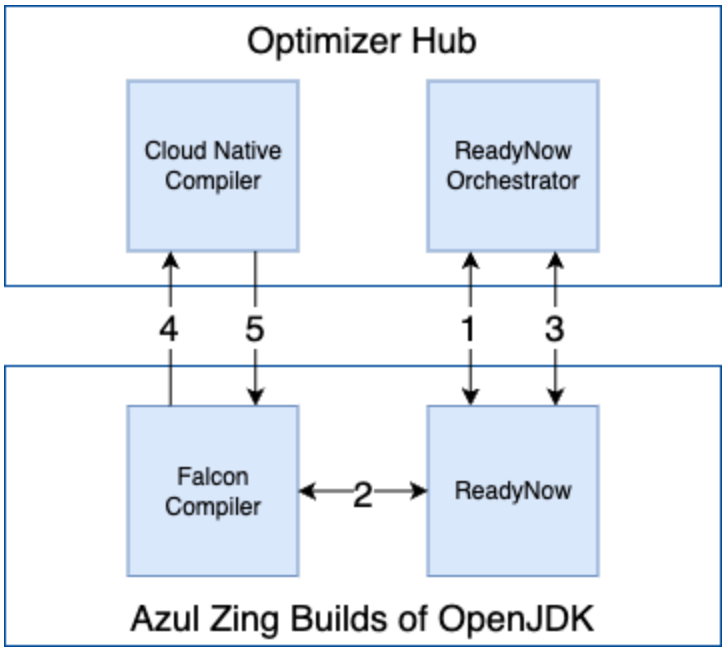
Documentation for Optimizer Hub, version 1.11.2

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 from Zing's Falcon JIT compiler 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).

About Cloud Native Compiler

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

JIT Optimization

Thanks to CNC, organizations can achieve faster, smarter, and more cost-effective application performance. This transforms the traditional limitations of on-JVM JIT compilation into strategic opportunities for performance gains, cost savings, and

operational efficiency.

Enhanced Optimization Capabilities

CNC enables the use of advanced speculative optimizations that result in significantly faster application code execution. Offloading the JIT compilation process to an external optimizer, unlocks several key benefits:

- **Access to Better Compute Resources**: Unlike traditional on-JVM JIT compilers, CNC has access to dedicated, scalable compute resources. This allows it to execute more sophisticated, aggressive optimizations that deliver higher performance outcomes.
- **Faster Optimization**: Since the external compiler is not constrained by the application's runtime environment, optimizations are applied more rapidly, enabling applications to achieve peak performance sooner.

Improved Application Performance from the Start

With CNC, applications experience a shorter warm-up period, leading to faster and more efficient execution right from the start.

- **Immediate Performance Gains**: By offloading the JIT compilation to an external service, applications avoid the typical "slow-start" period caused by on-JVM compilation. Applications run closer to optimal performance right after launch.
- **Resource Efficiency**: The application's compute and memory resources only execute your business logic, leading to faster response times and more consistent performance during critical early phases of execution.

Optimized Resource Allocation and Cost Efficiency

CNC provides an opportunity to reduce wasteful resource allocation and optimize for efficiency:

- **Resource Cost Savings**: Since JIT compilation happens on CNC, JVM instances can run with lower resource overhead, reducing operational and cloud infrastructure costs.

- **On-Demand Compiler Resources**: The resources used by CNC for JIT compilation are provisioned only when needed, rather than being reserved for the entire lifecycle of a process. This ensures more efficient utilization of compute capacity.

Falcon JIT with CNC

Azul Zing 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.

Video Introduction of ReadyNow Orchestrator

 https://www.azul.com/wp-content/uploads/AZL106-ReadyNow-Orchestrator-Video-AW.mp4 *(video)*

Optimizer Hub Architecture Overview

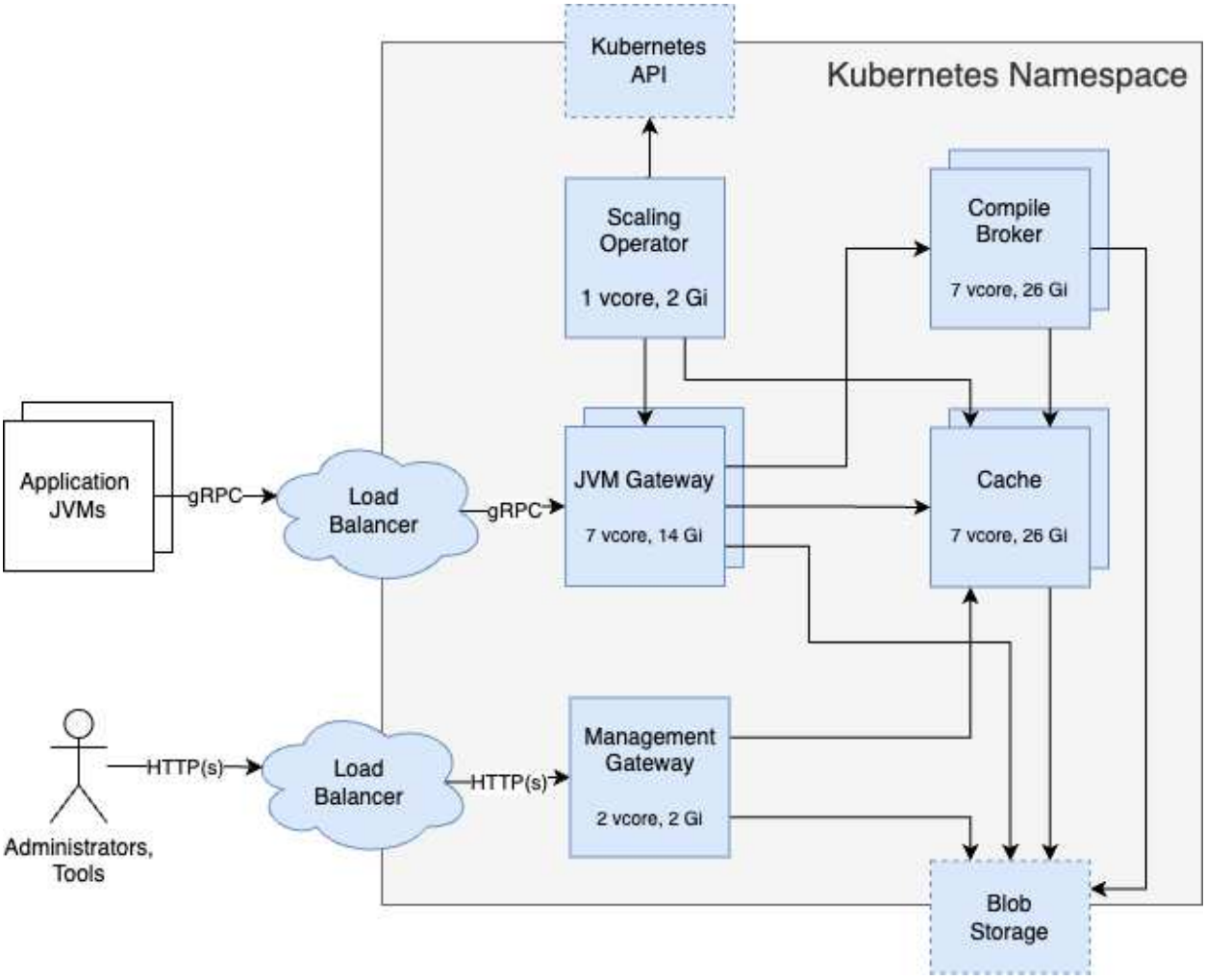
Optimizer Hub is shipped as a Helm chart and a set of Docker images to be deployed into a Kubernetes cluster. The Helm chart deploys different components based on the use case.

Architecture Overview

Optimizer Hub offers two deployment options: a full installation of all components or a ReadyNow Orchestrator-only installation.

Full Installation

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

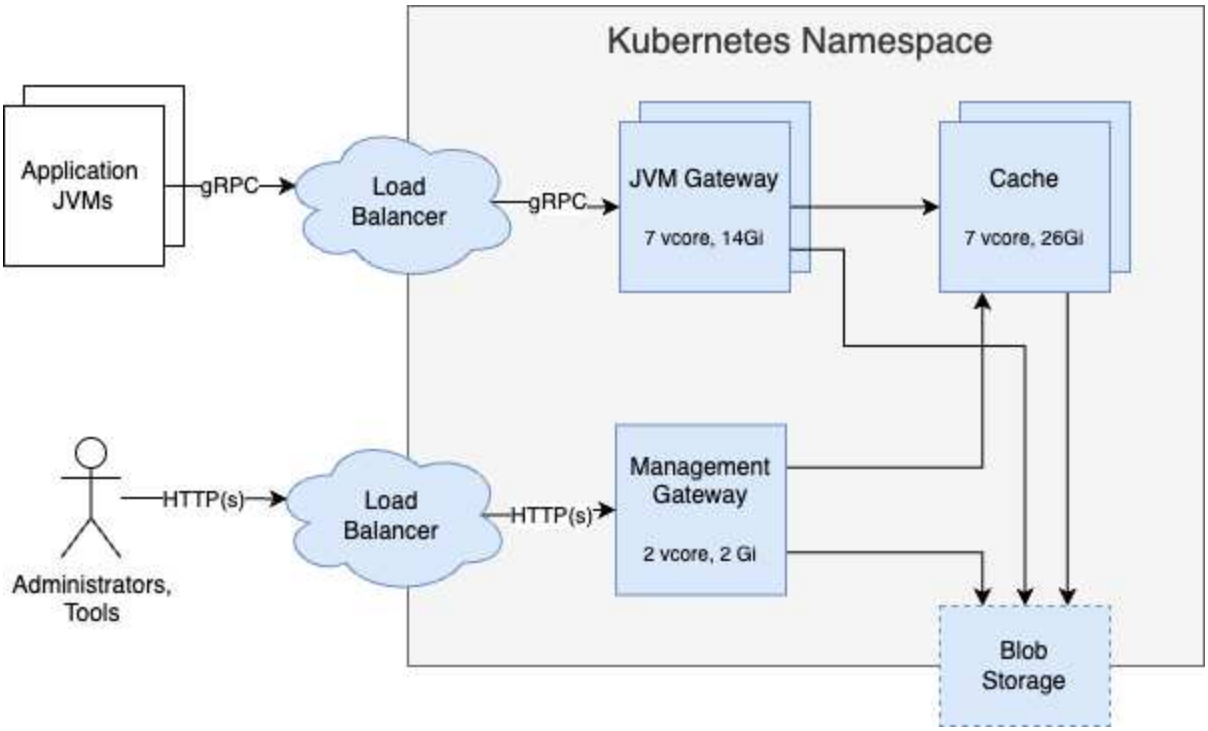


Remarks:

- All services use one pod, except Cache uses two pods by default.
- The load balancer is either your own solution (recommended), or the optional `gw-proxy` included in Optimizer Hub. See "Configuring Optimizer Hub Host" 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.



Recommended Pods and Nodes Sizing

Please follow these guidelines to set up your Kubernetes environment.

Kubernetes Pods

The sizes for the pods are specified in `values.yaml` file, as part of the Helm chart:

- Gateway: CPU 7, RAM 28GB
- Compile Broker: CPU 7, RAM 28GB
- Cache: CPU 7, RAM 28GB
- gwProxy: CPU 7, RAM 1GB
- Management Gateway: CPU 2, RAM 2GB
- Operator: CPU 1, RAM 2GB

Requirements for ephemeral storage (temporary storage space allocated to Kubernetes pods that is non-persistent and exists only for the lifetime of the pod):

- Compile Broker: 8GB
- All other pods: 1GB

Kubernetes Nodes

The underlying Kubernetes nodes (either cloud instances or physical computers) have to be large enough to fit one or more of the pods. This means they need to provide a multiple of 8 vCores with 4GB of RAM per vCore. For example: 8 vCore with 32GB, 16 vCores with 64GB, etc.

NOTE	Ensure the instances you run your Optimizer Hub have enough CPU to handle your requests. For example, on AWS use `m6` and `m7` instances, and on Google Cloud Platform c2-standard-8 instances.

High Availability of Optimizer Hub

Optimizer Hub is designed with High Availability (HA) as a fundamental architectural principle:

- The system architecture prioritizes continuous update and service reliability.
- Built-in redundancy at multiple levels ensures business continuity.

High Availability in Clusters

HA is guaranteed inside and between clusters.

Inside a Cluster

The nodes inside the Optimizer Hub service have failover built-in:

- Automatic redistribution of workload when a node fails.
- The system maintains full functionality even if individual nodes crash.
- A seamless transition between nodes prevents service interruption.

Between Clusters

HA is also integrated in configurations with multiple clusters:

- Clusters have health check endpoints to declare their readiness to accept traffic.
- You can add a DNS-based load balancer or service mesh to route the requests to the nearest available cluster.

- Custers can sync important information.

High Availability Configuration

Follow these recommendations to ensure High Availability (HA) of Optimizer Hub.

- Install multiple instances of the Optimizer Hub service, for example, one per region of availability zone.
- Front the Optimizer Hub service with either:
  - A DNS-based load balancer (i.e. Amazon Route 53)
  - Kubernetes service mesh (i.e. Istio)
  - See high-availability for more info.
- Let the clients connect to the load balancer or service mesh.
- Use the health check APIs to only route requests to instances that are ready to handle traffic.
- Route the requests to the Optimizer Hub service that is nearest to the JVMs.
- Set-up cross-region-sync.

NOTE	Cloud Native Compiler artifacts are not synced. They can be easily regenerated without compromising application performance.

Optimizer Hub Release Notes

Optimizer Hub 1.11.2

Release Date: June 10, 2025

New Features

- Includes bug fixes for Optimizer Hub 1.11.1.

Bug Fixes

- CNC compilation failures on deployments in IPv6-only networks.

- Race condition between ReadyNow Orchestrator profile deletion and cross-region syncing, that could result in an inconsistent state of the data, leading to subsequent `BlobNotFound` exceptions. In case such inconsistencies already happened within your setup, you need to manually delete the affected profiles.

Optimizer Hub 1.11.1

Release Date: May 13, 2025

New Features

- Includes bug fixes for Optimizer Hub 1.11.0.
- Memory settings for the Gateway pod are updated:
  - Memory requests/limits: from 14GB to 28GB.
  - `-XX:MaxRAMPercentage`: from 80 to 60.

Bug Fixes

- Typo fix in the response of the REST API `/api/currentlyConnectedProfileNames` for `resourceUsage`.

CVE Fixes

CVE	Base Score	Component
CVE-2024-8176	7.5	expat (2.4.7-1ubuntu0.5), libexpat1 (2.4.7-1ubuntu0.5)
CVE-2024-47606	7.5	Zulu (17.56.15)
CVE-2024-54534	7.5	Zulu (17.56.15)
CVE-2025-21587	7.4	Zulu (17.56.15)
CVE-2025-30691	4.8	Zulu (17.56.15)
CVE-2025-30698	5.6	Zulu (17.56.15)

Optimizer Hub 1.11.0

Release Date: April 7, 2025

New Features

- Includes bug fixes for Optimizer Hub 1.10.1.
- The REST API in Optimizer Hub has been improved and extended:
  - Previous endpoints have moved to a new address with new data formatting and paging.
  - Attributes were added to the existing entities to provide more data.
  - More information is available per JVM instance.
  - REST endpoints are documented based on the OpenAPI standard.
- A new readiness endpoint `/api/opthub-health/healthy` is available and replaces `/q/health`. You can use this endpoint to determine when it is safe to route traffic to a newly started Optimizer Hub cluster. This endpoint is provided by the gateway component, instead of the other ones, which are provided by the mgmt-gateway.

  Returned states:

  - `200`: OK
  - `300` and higher: Not OK
- With the new api-methods, you can instrument Optimizer Hub to temporarily increase the minimum number of vCPUs between a start and end timestamp. Multiple calls can be made to this API and Optimizer Hub will take all given timestamps and potential overlaps into account to start and stop the extra resources.

  See Sizing and Scaling your Optimizer Hub Installation for more info.
- The Management Gateway component in Optimizer Hub is now installed by default. It has become an essential part of any deployment as it provides the REST APIs.
- Azul Zing Builds of OpenJDK, version 25.02 and newer, provide a new command line option `-XX:CNCLocalFallbackOptLevelLimit=<3,2,1,0>` to define the

OptLevel for local fallback. See cloud-native-compiler-jvm-options.

- Secrets can be externally defined to allow you to manage Kubernetes secrets independent of the Optimizer Hub configuration. See Using Externally Defined Secrets for more info.

Bug Fixes

- Optimizer Hub now requires you to specify enough vCores to provision at least one Compile Broker if you have used the full installation. The minimum number of vCores is 39. When this is not the case, one or more of the following messages can be found in the log files:

```
The compile-broker's minimum replicas must be greater than 0
The gateway's minimum replicas must be greater than 0
The cache's minimum replicas must be greater than 0
```

- Improvement in ReadyNow Orchestrator to reduce the time before it can serve profiles during a restart of Optimizer Hub with a pre-populated storage.

Optimizer Hub 1.10.2

Release Date: May 7, 2025

New Features

- Includes bug fixes for Optimizer Hub 1.10.1.
- Memory settings for the Gateway pod are updated:
  - Memory requests/limits: from 14GB to 28GB.
  - `-XX:MaxRAMPercentage`: from 80 to 60.

CVE Fixes

CVE	Base Score	Component
CVE-2024-8176	7.5	expat (2.4.7-1ubuntu0.5), libexpat1 (2.4.7-1ubuntu0.5)

CVE	Base Score	Component
CVE-2024-47606	7.5	Zulu (17.56.15)
CVE-2024-54534	7.5	Zulu (17.56.15)
CVE-2025-1247	7.2	Quarkus (3.15.3.1)
CVE-2025-21587	7.4	Zulu (17.56.15)
CVE-2025-27363	8.1	libfreetype6 (2.11.1+dfsg-1ubuntu0.2)
CVE-2025-30691	4.8	Zulu (17.56.15)
CVE-2025-30698	5.6	Zulu (17.56.15)

Optimizer Hub 1.10.1

Release Date: March 13, 2025

New Features

- Includes bug fixes for Optimizer Hub 1.10.0.
- Fix for a problem where cross-region syncing of ReadyNow Orchestrator got stuck indefinitely. A better timeout implementation was added to remote calls (fetching profiles from another region), to prevent such blocks in case of network glitches.

Optimizer Hub 1.10.0

Release Date: December 20, 2024

New Features

- Azul Zing Builds of OpenJDK, version 24.08 introduced the new `-XX:ProfileName=<name>` option that allows a JVM instance to specify a profile name to Optimizer Hub. This name allows multiple JVM instances that use the same profile name to share the Optimizer Hub functionality, such as the use of ReadyNow Orchestrator profiles and Cloud Native Compiler caching.

  This new feature also introduces the new command line option `-XX:+EnableRNO` that enables ReadyNow read and writes in the JVMs against ReadyNow Orchestrator,

using `ProfileName` as the name for the profile log. More info is provided in readynow-orchestrator-jvm-options.

The previously used options (`OptHubHost` and `ProfileLogName`) is still supported to existing configurations don't break. But we advise to use the new settings with `OptHubHost`, `EnableRNO`, and `ProfileName`.

- Native support for Google Cloud Platform Blob Storage is added. See Installing Optimizer Hub on Google Cloud.
- MinIO has been removed from the helm chart as it is mainly intended for testing and demos. For production, cloud-managed blob storage is recommended (configuring-aws-s3-storage, configuring-azure-blob-storage, configuring-gcp-blob-storage, or S3-compatible (e.g. for Alibaba)).

  See configure-blob-storage for more info.
- The database storage for Code Cache is deprecated because blob storage is the best production-friendly option as it is more scalable, highly available, and durable. At the same time, blob storage is simpler to maintain.

  See Optional Database Pod Configuration if you want to keep using the database.
- Avoid cleanup of profiles from systems with low restarts by using the last time the JVM requesting that profile was seen alive, rather than the last time the profile was requested.

Bug Fixes

- Improved time-outs for Profile Sync Task. In some configurations with cross-region syncing, the sync task could get stuck because of incorrect configurations. This has been fixed with improved time-outs.
- Cross-region syncing of ReadyNow Orchestrator profiles is improved to let new instances check if a later promoted generation becomes available.

- Improved the handling of unloaded or unknown classes.

Known Issues

- A large amount of stored data may require a significant grace period after a restart of Optimizer Hub to avoid issues with profile download.

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 Overview of the 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 grpc-proxy.

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

Known Issues

Fixed Ports for gRPC

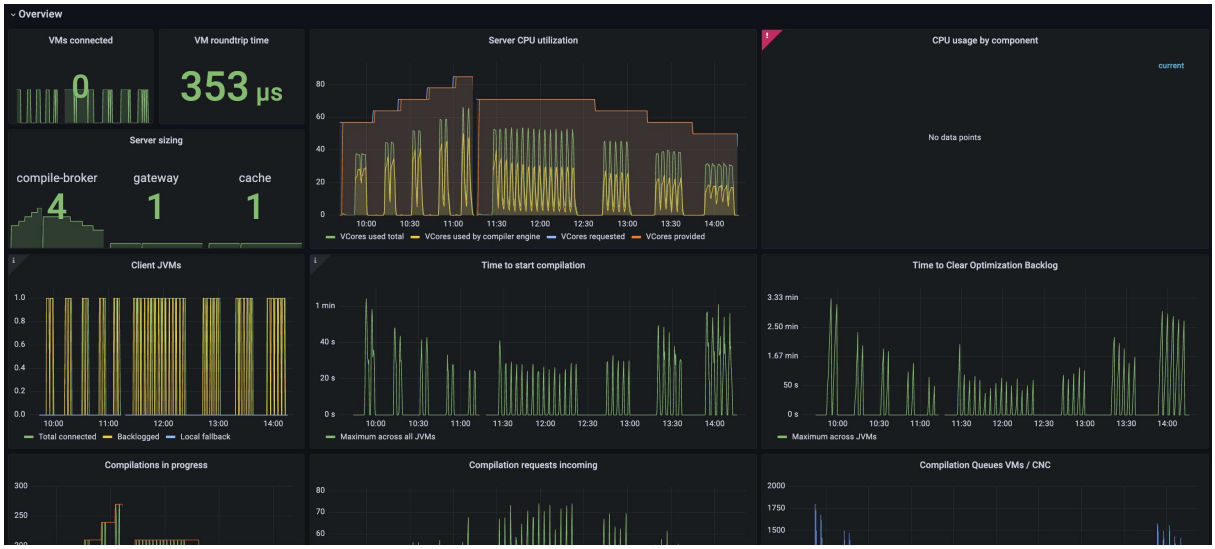
The helm chart values contain the keys `gateway.service.httpEndpoint.port` and `gateway.service.grpc.port` to change the default ports 50051 and 8080. But these values are hardcoded for the gRPC Envoy proxy, at this moment, and cannot be changed with the mentioned helm chart keys.

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 Zing 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
1.11.2	PDF
1.11.1	PDF
1.11.0	PDF
1.10.2	PDF
1.10.1	PDF
1.10.0	PDF
1.9.5	PDF
1.9.4	PDF
1.9.3	PDF
1.9.2	PDF
1.9.1	PDF

Version	Optimizer Hub TPL
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.

Load Balancing

It's recommended to use a load balancer or service mesh to set up a high-availability system, optionally with a secondary fallback system. JVMs connecting to Optimizer Hub need a stable, single entry point to communicate with the service.

It's recommended to use your own load balancer and configure the DNS of the system that must be used by the JVMs to connect. See load-balancer and the "Readiness (healthy) API". If no load balancer is available, you can use the optional gw-proxy.

Benefits of a Load Balancer

A load balancer provides this external access point while also potentially offering benefits like:

- SSL configuration in the load balancer
- Traffic distribution across Optimizer Hub components
- High availability
- Network isolation
- Consistent endpoint for clients regardless of internal pod IP changes

Load Balancer Requirements

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

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

NOTE	By downloading and using Optimizer Hub, you agree with the Azul Platform Prime Evaluation Agreement.

Using Internal Docker Registry

The Optimizer Hub components, by default, are installed from the Docker Hub `azul/opthub-gateway`. These images can also be provided by your own registry, e.g. if your IT or security policies require this.

We advise to preserve the image name and version tag when you save the Docker images to your internal registry, to minimize the customization in the helm chart and

avoid confusion. For example, when saving a copy of `azul/compile-broker:1.11.1`, save it as `<your-registry>/docker-external/azul-zing/compile-broker:1.11.1`.

When you preserve the image names and version tags, the only extra configuration needed in your `values-override.yaml` file is:

```
registry:
    opthub: "<your-registry>/docker-external/azul-zing"
```

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.

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.

Prerequisites

Before installing Optimizer Hub, make sure to check the following prerequisites:

- Setting up a load balancer or services mesh to handle SSL is recommended, see "Configuring Optimizer Hub with SSL Authentication".
- Check recommended-sizing to define the sizing of your cluster.
- Helm is required to set up your Optimizer Hub instance. Azul provides Optimizer Hub Helm Charts on GitHub, which you can download as a zip package.
- If you are upgrading an existing installation, make sure to check "Upgrading Optimizer Hub".

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 Zing Builds of OpenJDK 24.02 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 of the Optimizer Hub components according to the "sizing guide". By default, autoscaling is enabled and Optimizer Hub requires 39 vCores and can scale up to 10 Compile Brokers. For example, you could set the following in your `values-override.yaml` file:

```
simpleSizing:
  vCores: 39
  minVCores: 39
```

```
  maxVCores: 113
```

9. Configure the blob storage according to your environment in your `values-override.yaml` file:

```
storage:
  # Available options: s3, azure-blob, gcp-blob
  blobStorageService: s3
  # Depending on the type of storage, configure the extra settings
  s3:
    ...
  azureBlob:
    ...
  gcpBlob:
    ...
```

   For more detailed blob storage instructions, please check:

   - configuring-aws-s3-storage
   - configuring-azure-blob-storage
   - configuring-gcp-blob-storage
   - "S3-compatible (e.g. for Alibaba)"

10. 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.

11. 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 1.11.2` 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
```

12. Verify that all started pods are ready:

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

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).

Configuring AWS S3 Storage

Optimizer Hub requires a bucket and R/W permissions to the bucket.

1. Within the AWS system, create the bucket and R/W permissions.
2. Configure the Optimizer Hub storage by adding the following to your `values-override.yaml` file:

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

3. Configure the permissions by adding the following to your `values-override.yaml` file:

```
deployment:
    serviceAccount:
```

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

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.

Installing Optimizer Hub on EKS

Please download opthub-install-1.11.2.zip for additional files to configure Optimizer Hub on AWS EKS.

Cluster Requirements

You can create a cluster following the steps in Cluster Provisioning on EKS, or use a cluster created by any other means according to these requirements:

- ReadyNow Orchestrator requires on-demand EC2 instances. Don't use spot instances.
- All the nodes must have at least 8 vCores and 32 GB RAM to fit the Optimizer Hub pods.
- The suggested EC2 instance types are `m6` or `m7`. Using instances with less powerful CPUs may negatively impact the performance of Optimizer Hub.

Cluster Provisioning on EKS

If you don't have a cluster available to run Optimzer Hub, you can provision one on EKS with the following steps:

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. In the downloaded package, navigate to the `eks` directory.
3. In `opthub_eks.yaml`, 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. Pass the modified `opthub_eks.yaml` file to `eksctl` to create the cluster. For more information, look at the eskctl config file schema. Apply the file with the following

command:

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

This command takes several minutes to execute, and when successful ends with the following output:

```
[¬] EKS cluster "eks-opthub-cluster" in "eu-central-1" region is ready
```

By using `opthub_eks.yaml`, the following gets created 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

Install Optimizer Hub on an EKS Cluster

Follow the installation instructions on "Installing Optimizer Hub on Kubernetes" to install all services of Optimizer Hub. Or, 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".

In the step where Helm is used to create the node groups in the cluster, pass in the additional configuration file `eks/values-eks.yaml`, located in the installation package. This file includes the nodegroup affinity settings and other settings expected by EKS.

The Helm installation command needs to be extended with the `values-eks.yaml` config file:

```
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.

Setting Up an External Load Balancer

As described on "Configuring Optimizer Hub Host", it is highly recommend to use a load-balanced setup.

To set up an AWS load balancer, please follow the documentation on Route internet traffic with AWS Load Balancer Controller.

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

Optimizer Hub requires a bucket and R/W permissions to the bucket.

1. Within the Azure system, create the bucket and R/W permissions.
2. Configure the Optimizer Hub storage by adding the following to your `values-override.yaml` file:

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

3. Configure the permissions by adding the following to your `values-override.yaml` file:
   - 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".

Configuring GCP Blob Storage

Optimizer Hub requires a bucket and R/W permissions to the bucket.

1. Within the Google Cloud system, create the bucket and R/W permissions.
2. Configure the Optimizer Hub storage by adding the following to your `values-override.yaml` file:

```
storage:
  blobStorageService: gcp-blob
  gcpBlob:
    commonBucket: opthub-storage0
```

3. Configure the permissions by adding the following to your `values-override.yaml` file:

```
deployment:
  serviceAccount:
    annotations:
      iam.gke.io/gcp-service-account: <YOUR_SERVICE_ACCOUNT>
```

IAM Policy Update

An IAM policy update is required to add the role to the service account to assign the required permissions for the bucket :

```
>> gsutil iam get gs://<YOUR_BUCKET>
{
  "bindings": [
    ...
    {
      "members": [
        "serviceAccount:<YOUR_SERVICE_ACCOUNT>"
      ],
      "role": "roles/storage.objectAdmin"
    }
  ],
  "etag": "CAM="
}
```

You can use the following CLI command to assign the required roles to a bucket:

```
>>gsutil iam ch serviceAccount:<YOUR_SERVICE_ACCOUNT>:roles/storage.objectAdmin
gs://<YOUR_BUCKET>
```

IAM Policy Binding

```
>>gcloud iam service-accounts get-iam-policy <YOUR_SERVICE_ACCOUNT>
bindings:
- members:
- serviceAccount:<YOUR_PROJECT_ID>.svc.id.goog[<YOUR_NAMESPACE>/opthub-cache]
- serviceAccount:<YOUR_PROJECT_ID>.svc.id.goog[<YOUR_NAMESPACE>/opthub-compile-
broker]
- serviceAccount:<YOUR_PROJECT_ID>.svc.id.goog[<YOUR_NAMESPACE>/opthub-gateway]
- serviceAccount:<YOUR_PROJECT_ID>.svc.id.goog[<YOUR_NAMESPACE>/opthub-mgmt-
gateway]
role: roles/iam.workloadIdentityUser
etag: BwYo0_53sDw=
version: 1
```

You can use the following CLI command to add `workloadIdentity` to the Kubernetes service account names for the server components (`opthub-cache`, `opthub-compile-broker`, `opthub-gateway`, and `opthub-mgmt-gateway`):

```
gcloud iam service-accounts \
  add-iam-policy-binding <YOUR_SERVICE_ACCOUNT> \
  --role roles/iam.workloadIdentityUser \
  --member "serviceAccount:<YOUR_PROJECT_ID>.svc.id.goog[<YOUR_NAMESPACE>/opthub-
gateway]"
gcloud iam service-accounts \
   add-iam-policy-binding <YOUR_SERVICE_ACCOUNT> \
   --role roles/iam.workloadIdentityUser \
   --member "serviceAccount:<YOUR_PROJECT_ID>.svc.id.goog>[<YOUR_NAMESPACE>/opthub-
cache]"
gcloud iam service-accounts \
   add-iam-policy-binding <YOUR_SERVICE_ACCOUNT> \
   --role roles/iam.workloadIdentityUser \
   --member "serviceAccount:<YOUR_PROJECT_ID>.svc.id.goog[<YOUR_NAMESPACE>/opthub-
compile-broker]"
gcloud iam service-accounts \
   add-iam-policy-binding <YOUR_SERVICE_ACCOUNT> \
   --role roles/iam.workloadIdentityUser \
   --member "serviceAccount:<YOUR_PROJECT_ID>.svc.id.goog[<YOUR_NAMESPACE>/opthub-
mgmt-gateway]"
```

Installing on an S3 Compatible Environment

If you want to install Optimizer Hub on a platform which provides S3 compatibility mode, instead of cloud native blob storage, 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
```

Additional settings may be needed, for example, when using MinIO and Minikube:

```
storage:
  blobStorageService: s3
  s3:
    commonBucket: opthub
    credentialsType: "static"
    storageEndpoint: http://minio.minio-dev.svc.cluster.local:9000
```

Configuring S3 Authentication

Add the `S3` authentication keys in your `values-override.yaml` or check "Using Externally Defined Secrets" for more options.

```
secrets:
  blobStorage:
    s3:
      accesskey: KEY
      secretkey: SECRET
```

Installing Optimizer Hub on MicroK8s

MicroK8s can be used for testing, evaluating, and non-cloud-managed blob storage use of Optimizer Hub.

Make sure your MicroK8s meets the 18 vCore minimum for running Optimizer Hub. Although MicroK8s can run on multiple platforms, Optimizer Hub is only available for the **x64** platform, so not on macOS with M-processor.

Installing MicroK8s

Install MicroK8s for your platform following this installation guide.

- Install MicroK8s on Linux with Snap:

```
sudo snap install microk8s --classic
```

- Check the status while Kubernetes starts:

```
$ microk8s status --wait-ready
microk8s is running
high-availability: no
  datastore master nodes: 127.0.0.1:19001
  datastore standby nodes: none
```

- Turn on the services you need, in our case, only `dashboard`:

```
microk8s enable dashboard
```

- Access the Kubernetes dashboard:

```
$ microk8s dashboard-proxy

Checking if Dashboard is running.
Infer repository core for addon dashboard
Waiting for Dashboard to come up.
Trying to get token from microk8s-dashboard-token
Waiting for secret token (attempt 0)
Dashboard will be available at https://127.0.0.1:10443
Use the following token to login:
eyJhbGciOiJSUzI1NiIs...
```

- You can now open the dashboard in the browser, using the token you got in the previous step.

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, check it with `helm version`.
2. Add the Azul Helm repository to your Helm environment:

```
microk8s helm repo add opthub-helm https://azulsystems.github.io/opthub-helm-charts/
```

```
microk8s helm repo update
```

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

```
microk8s kubectl create namespace my-opthub
```

4. Clone or download the files from the GitHub opthub-helm-charts repository.

```
git clone https://github.com/AzulSystems/opthub-helm-charts.git
```

5. Create a directory for your configuration files

```
mkdir ~/my-opthub/
```

6. Create MinIO storage:

   - Copy minio-dev.yaml and replace `minio-dev` with `my-opthub` or the namespace you created in the previous step.

   ```
   cp ~/opthub-helm-charts/minio-dev.yaml ~/my-opthub/minio-dev.yaml
   sed -i 's/minio-dev/my-opthub/g' ~/my-opthub/minio-dev.yaml
   # `-i` modifies the original file
   # `s/old/new/g`: s = substitute, g = all occurrences
   ```

   - Copy minio-setup-job.yaml and again replace `minio-dev`.

   ```
   cp ~/opthub-helm-charts/minio-setup-job.yaml ~/my-opthub/minio-setup-job.yaml
   sed -i 's/minio-dev/my-opthub/g' ~/my-opthub/minio-setup-job.yaml
   ```

   - Install the S3 compatible storage with:

   ```
   microk8s kubectl apply -f ~/my-opthub/minio-dev.yaml -f ~/my-opthub/minio-
   setup-job.yaml
   ```

7. Create a configuration file `values-minikube.yaml`, based on the example file, to disable all resource definitions.

```
cp ~/opthub-helm-charts/values-minikube.yaml ~/my-opthub/values-minikube.yaml
```

8. 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".

```
microk8s helm install opthub opthub-helm/azul-opthub -n my-opthub -f ~/my-
opthub/values-minikube.yaml
```

   The command should produce output similar to this:

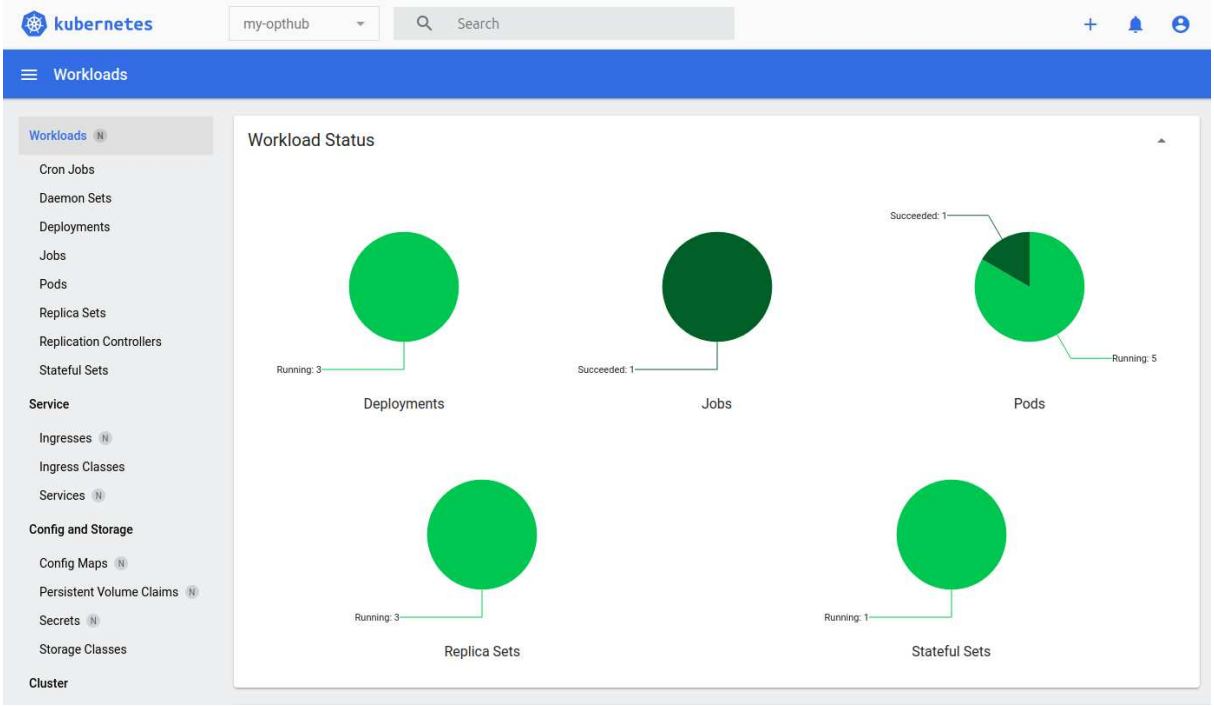
```
NAME: opthub
LAST DEPLOYED: Tue Jun 17 16:52:18 2025
NAMESPACE: my-opthub
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

9. Verify that all started pods are ready:

```
$ microk8s kubectl get all -n my-opthub

NAME                                  READY   STATUS      RESTARTS   AGE
pod/cache-0                           1/1     Running     0          2m58s
pod/compile-broker-59b99b66d6-njmwg   1/1     Running     0          2m58s
pod/gateway-8445f9d959-zvp5f          1/1     Running     0          2m58s
pod/gw-proxy-6c7b66bb6f-t4675         1/1     Running     0          2m58s
pod/minio                             1/1     Running     0          3m42s
pod/minio-setup-job-hqpw7             0/1     Completed   1          3m42s

NAME                        TYPE           CLUSTER-IP       EXTERNAL-IP    PORT(S)
AGE
service/cache               ClusterIP      None             <none>
5701/TCP            2m58s
service/compile-broker      ClusterIP      10.152.183.188   <none>
50051/TCP           2m58s
service/gateway             LoadBalancer   10.152.183.250   <pending>
50051:31401/TCP     2m58s
service/gateway-headless    ClusterIP      None             <none>
50051/TCP           2m58s
service/minio               ClusterIP      10.152.183.157   <none>
9000/TCP,9090/TCP   3m42s

NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/compile-broker      1/1     1            1           2m58s
deployment.apps/gateway             1/1     1            1           2m58s
deployment.apps/gw-proxy            1/1     1            1           2m58s

NAME                                           DESIRED   CURRENT   READY   AGE
replicaset.apps/compile-broker-59b99b66d6      1         1         1       2m58s
replicaset.apps/gateway-8445f9d959             1         1         1       2m58s
replicaset.apps/gw-proxy-6c7b66bb6f            1         1         1       2m58s
```

```
NAME                          READY   AGE
statefulset.apps/cache        1/1     2m58s

NAME                              STATUS     COMPLETIONS   DURATION   AGE
job.batch/minio-setup-job         Complete   1/1           12s        3m42s
```

10. You can also verify the status on the dashboard we opened before, by selecting the `my-opthub` namespace in the header:



Upgrade Optimizer Hub on MicroK8s

You can upgrade your Optimizer Hub instance running on MicroK8s, and keep all profile data and compile caches as the S3 compatible minio storage won't be upgraded.

1. List the currently installed version:

```
$ microk8s helm list --namespace my-opthub
NAME    NAMESPACE   REVISION    UPDATED                                 STATUS
CHART               APP VERSION
opthub  my-opthub   1           2025-06-17 16:59:26.33115804 +0200 CEST deployed
azul-opthub-1.11.2  1.11.2
```

2. List the available versions on the helm repo:

```
$ microk8s helm repo update
$ microk8s helm search repo opthub-helm --versions
```

```
NAME                      CHART VERSION   APP VERSION DESCRIPTION
opthub-helm/azul-opthub 1.11.2          1.11.2      Azul Intelligence Cloud:
Optimizer Hub
opthub-helm/azul-opthub 1.11.1          1.11.1      Azul Intelligence Cloud:
Optimizer Hub
opthub-helm/azul-opthub 1.11.0          1.11.0      Azul Intelligence Cloud:
Optimizer Hub
opthub-helm/azul-opthub 1.10.2          1.10.2      Azul Intelligence Cloud:
Optimizer Hub
opthub-helm/azul-opthub 1.10.1          1.10.1      Azul Intelligence Cloud:
Optimizer Hub
opthub-helm/azul-opthub 1.10.0          1.10.0      Azul Intelligence Cloud:
Optimizer Hub
opthub-helm/azul-opthub 1.9.5           1.9.5       Azul Intelligence Cloud:
Optimizer Hub
...
```

3. Upgrade and use the same settings and files from the installation, just by replacing `install` with `upgrade`:

```
microk8s helm upgrade opthub opthub-helm/azul-opthub -n my-opthub -f ~/my-
opthub/values-minikube.yaml
```

NOTE To upgrade or downgrade to a specific version, add `--version <version>`.

4. After upgrading, verify the version with the helm list command again.

5. You can check the version of the MinIO storage with this command:

```
$ microk8s kubectl exec -n my-opthub pod/minio -- minio --version
minio version RELEASE.2024-12-18T13-15-44Z (commit-id
=16f8cf1c52f0a77eeb8f7565aaf7f7df12454583)
```

Uninstalling Optimizer Hub from MicroK8s

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

```
microk8s helm uninstall opthub -n my-opthub
microk8s kubectl delete namespace my-opthub
microk8s helm repo remove opthub-helm
```

Installing Optimizer Hub on Minikube

Minikube can be used for testing, evaluating, and non-cloud-managed blob storage use of Optimizer Hub.

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 M-processor.

Blob storage is required for Optimizer Hub (since 1.10) and can be added to your Minikube setup with MinIO.

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, check it with `helm version`.
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. Clone or download the files from the GitHub opthub-helm-charts repository.

```
git clone https://github.com/AzulSystems/opthub-helm-charts.git
```

5. Create a directory for your configuration files

```
mkdir ~/my-opthub/
```

6. Create MinIO storage:
   - Copy minio-dev.yaml and replace `minio-dev` with `my-opthub` or the namespace you created in the previous step.

     ```
     cp ~/opthub-helm-charts/minio-dev.yaml ~/my-opthub/minio-dev.yaml
     sed -i 's/minio-dev/my-opthub/g' ~/my-opthub/minio-dev.yaml
     # `-i` modifies the original file
     # `s/old/new/g`: s = substitute, g = all occurrences
     ```

   - Copy minio-setup-job.yaml and again replace `minio-dev`.

     ```
     cp ~/opthub-helm-charts/minio-setup-job.yaml ~/my-opthub/minio-setup-job.yaml
     sed -i 's/minio-dev/my-opthub/g' ~/my-opthub/minio-setup-job.yaml
     ```

   - Install the S3 compatible storage with:

     ```
     minikube kubectl -- apply -f ~/my-opthub/minio-dev.yaml -f ~/my-opthub/minio-
     setup-job.yaml
     ```

7. Create a configuration file `values-minikube.yaml`, based on the example file, to disable all resource definitions.

   ```
   cp ~/opthub-helm-charts/values-minikube.yaml ~/my-opthub/values-minikube.yaml
   ```

8. 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 ~/my-opthub/values-
   minikube.yaml
   ```

   The command should produce output similar to this:

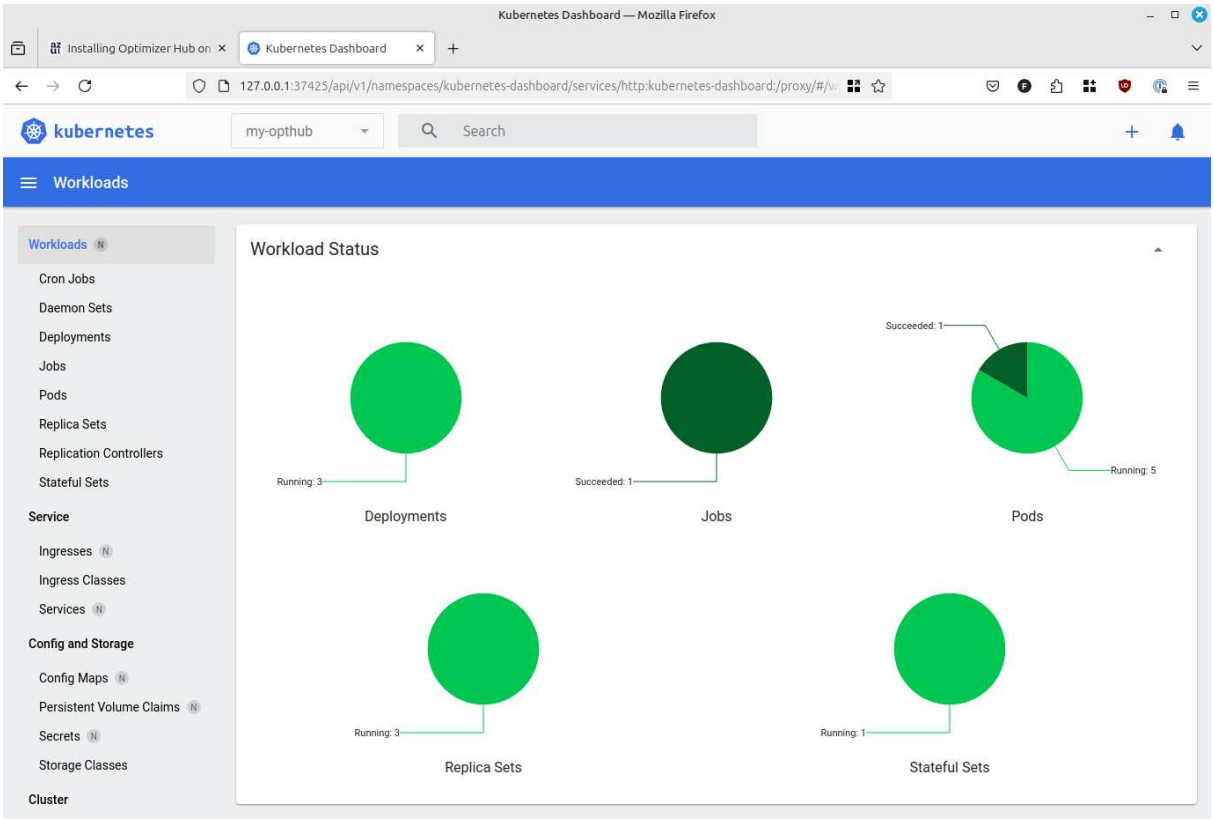
   ```
   NAME: opthub
   LAST DEPLOYED: Mon Jan 30 14:35:29 2023
   NAMESPACE: my-opthub
   STATUS: deployed
   REVISION: 1
   TEST SUITE: None
   ```

9. Verify that all started pods are ready:

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

NAME                                  READY   STATUS      RESTARTS   AGE
pod/cache-0                           1/1     Running     0          65s
pod/compile-broker-59b99b66d6-bg86g   1/1     Running     0          65s
pod/gateway-8445f9d959-p5l6k          1/1     Running     0          65s
pod/gw-proxy-6c7b66bb6f-829t8         1/1     Running     0          65s
pod/minio                             1/1     Running     0          2m50s
pod/minio-setup-job-zcbpz             0/1     Completed   0          2m50s

NAME                        TYPE           CLUSTER-IP      EXTERNAL-IP   PORT(S)
AGE
service/cache               ClusterIP      None            <none>        5701/TCP
65s
service/compile-broker      ClusterIP      10.105.247.81   <none>
50051/TCP           65s
service/gateway             LoadBalancer   10.105.68.63    <pending>
50051:31368/TCP     65s
service/gateway-headless    ClusterIP      None            <none>
50051/TCP           65s
service/minio               ClusterIP      10.108.8.165    <none>
9000/TCP,9090/TCP   2m50s

NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/compile-broker      1/1     1            1           65s
deployment.apps/gateway             1/1     1            1           65s
deployment.apps/gw-proxy            1/1     1            1           65s

NAME                                           DESIRED   CURRENT   READY   AGE
replicaset.apps/compile-broker-59b99b66d6      1         1         1       65s
replicaset.apps/gateway-8445f9d959             1         1         1       65s
replicaset.apps/gw-proxy-6c7b66bb6f            1         1         1       65s

NAME                       READY   AGE
statefulset.apps/cache     1/1     65s

NAME                           STATUS     COMPLETIONS   DURATION   AGE
job.batch/minio-setup-job      Complete   1/1           11s        2m50s
```

10. You can also verify the status on a dashboard in the browser with the following command. Select your namespace (`my-opthub`) from the dropdown in the header of the page.

```
minikube dashboard
```



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
helm repo remove opthub-helm
```

Upgrading Optimizer Hub

Follow these steps to upgrade your Optimizer Hub instance:

1. Shut down the existing instance.
2. Create a complete backup of the blob storage content.
3. Validate your `values-override.yaml` file for any parameter changes.
4. Install the new version "following the installation guide".

This procedure will cause a temporary service outage. For zero-downtime upgrades, consider implementing a failover system. See high-availability.

Rolling Back to a Previous Version

In case a rollback is needed after upgrading, follow these steps:

1. Shutdown the upgraded instance.
2. Restore the blob storage content from your pre-upgrade backup.
3. Re-install the previous version.

> **NOTE** The blob storage metadata format may vary between releases. Performing a rollback with a blob storage containing files modified by a newer version may result in incorrect behavior.

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.

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.	true
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

Blob Storage Auto Cleanup Parameters

See how-scales.

Simple Sizing Parameters

See "Configuring Blob Storage Auto Cleanup".

SSL Parameters

See "Configuring Optimizer Hub with SSL Authentication".

Storage Parameters

Storage parameters depend on the platform of your deployment:

- configure-blob-storage
- configuring-aws-s3-storage
- configuring-azure-blob-storage
- configuring-gcp-blob-storage
- "S3-compatible (e.g. for Alibaba)"

Using Externally Defined Secrets

Secrets can be externally defined to allow you to manage Kubernetes secrets

independent of the Optimizer Hub configuration.

Defining Your Secrets

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

- blobstorage.s3.accesskey
- blobstorage.s3.secretkey
- azure.connectionString
- azure.sasToken

Default Settings

These are the default settings with descriptions to describe how they are used.

```
secrets:
  blobStorage:
    s3:
      # name of existing Secret object to use. New Secret is created if name is
empty
      existingSecret: ""

      # name of *key* for `accessKey` value in K8S Secret. It can be renamed to
match names in existing Secret
      accessKeySecretKey: blob-storage-accesskey

      # default value for accesskey - used when new secret is created
      accesskey: <yourAccessKey>

      # name of *key* for `secretkey` value in K8S Secret. It can be renamed to
match names in existing Secret
      secretAccessKeySecretKey: blob-storage-secretkey

      # default value for s3.secretkey - used when new secret is created
      secretkey: <yourSecretKey>

    azure:
      # name of existing Secret object to use. New Secret is created if name is
empty
      existingSecret: ""

      # name of *key* for `connectionStringSecretKey` value in K8S Secret. It can
be renamed to match names in existing Secret
      connectionStringSecretKey: azure-storage-connection-string
      # connectionString: "<connection-string>" . For authMethod: connection-
string,
      # get connection-string on Azure Portal > Storage accounts >
{storage_account_name} > Access keys
```

```
      # name of *key* for `sasTokenSecretKey` in K8S Secret. It can be renamed to
match names in existing Secret
      sasTokenSecretKey: azure-storage-sas-token
      # sasToken: "<sas-token>"  # For authMethod: sas-token,
      # Get sas-token on Azure Portal > Storage accounts > {storage_account_name} >
{blob_container} > Shared access tokens
```

How To Use

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

Example

For example, if you have an existing secret with S3 credentials, and the name of this K8S secret Object is `awsS3secretsForOpthub`, it should contain the following values:

```
MyKeyID: key123455
MyKey: xyzabcdef
```

Then you can configure Optimizer Hub with the following values in your `values-override.yaml` file:

```
secrets:
  blobStorage:
    s3:
      existingSecret: awsS3secretsForOpthub
      accessKeySecretKey: MyKeyID
      secretAccessKeySecretKey: MyKey
```

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`.

Configuring Optimizer Hub Host

As an Optimizer Hub administrator, you must provide users the "host (DNS or IP) and optional port of the Optimizer Hub service" of the Optimizer Hub service or the (DNS) load balancer the JVMs must connect to. The JVMs need this for the value to be specified in the `-XX:OptHubHost=<host>[:<port>]` option.

We highly recommend using a load-balanced setup, configured with a DNS address, that redirects requests from JVMs to one or more Optimizer Hub services.

Host for Single Optimizer Hub service

In a setup with a single Optimizer Hub service, you can either use the included `gw-`

`proxy` component, or add your own load balancer.

Using your Own Load Balancer

It's recommended to use your own preferred load balancer, consistent with how you dispatch HTTP traffic to your other applications. In such a case, disable `gw-proxy` in Optimizer Hub and use your own instance, by adding the following to your `values-override.yaml` file:

```
gwProxy.enabled=false
```

Your load balancer must be aware of gRPC calls and avoid affinity to a single gateway and not interrupt long calls.

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
```

Using the Included gw-proxy

NOTE	We recommend using your own load balancer.

The `gw-proxy` pod which is deployed in the Optimizer Hub namespace is the default load balancer. It uses Envoy as the default gRPC proxy for optimal session balancing. You can find the endpoint of `gw-proxy` 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 port 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
```

Host for High Availability and Failover

When you have multiple Optimizer Hub services to guarantee high availability (HA) and provide a failover system, you can use the following approaches.

- Use a (DNS) load balancer of your choice, e.g. Route 53.
- Use the readiness state of each Optimizer Hub service by using the Kubernetes check available on `/q/health`.

- Configure your (DNS) load balancer with the host info of each Optimizer Hub service.

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 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. If necessary, assign a different port to the Management Gateway component 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 content on the gateway	true
readyNowOrchestrator.cache.maxSizeBytes	The fixed size of content cached 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

Option	Description	Default
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
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	Determines the threshold for the blob data usage, at which ReadyNow Orchestrator initiates its cleanup process.	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. You need to specify a value if you want to enable complete cleanup of unused profiles.	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
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 Blob Storage Auto Cleanup

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

- ReadyNow Orchestrator: saved profile logs.
- Code Cache: previously performed compilations.

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

Code Cache Cleanup

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

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

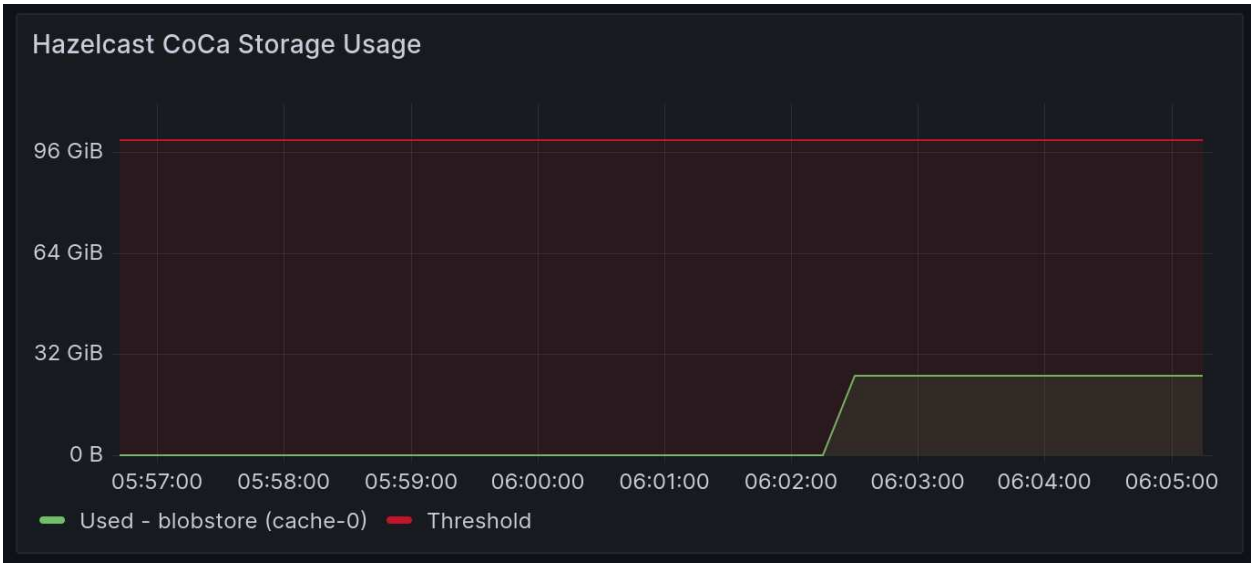
```
codeCache:
  cleaner:
    enabled: true
    targetSize: "107374182400" # 100GiB, use quotes for large numbers
    interval: PT2H # 2 hours
```

How Code Cache Storage Gets Cleaned

At a regular interval, the Code Cache cleaner checks whether the current usage is bigger than `targetSize`. If so, the cleanup process is triggered. This process deletes the Code Cache items that are least recently used to get below the `targetSize`.

Check Used Code Cache Storage Size

You can check the used storage size in the Grafana dashboard in the "Details: cache" section.



ReadyNow Profile Log Cleanup

ReadyNow Orchestrator performs automatic cleanup of unused profile logs in order to

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

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

```
readyNowOrchestrator:
  cleaner:
    enabled: true
    externalPersistentStorageSoftLimit: "10Gi"
    targetSize: 0 # use only to override auto-settings
    warningSize: 0 # use only to override auto-settings
    keepUnrequestedProfileNamesFor: 0
    keepDebugOnlyGenerationProfilesFor: "P7D"
```

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

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

NOTE

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 not enough profiles can be cleaned up. When you reach 90% usage, a warning is printed in the log of the gateway service. In that case, you need to increase the `externalPersistentStorageSoftLimit`.

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

How ReadyNow Storage Gets Cleaned

By default, the ReadyNow profile log auto cleaner follows these steps:

- For each profileName: deletes all the profiles that are not promoted at this time. The cleaner keeps the five last-used profileLogs. It only deletes enough profileLogs to get under `targetSize`.
  - The currently promoted profiles will never be deleted.
- Deletes all debug only profileLogs, meaning profileLogs with a generation higher than `readyNowOrchestrator.producers.maxPromotableGeneration`, if they have not been accessed longer than the period defined in `keepDebugOnlyGenerationProfilesFor`.
  - Can delete all of these debug profileLogs, but only deletes enough to get under `targetSize`.
- Deletes any completely unrequested profileNames.
  - Will delete all of them, regardless of `targetSize`.

Configuring Optimizer Hub SSL Authentication

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

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

SSL Configuration in Optimizer Hub

Follow these steps to configure the SSL configuration within the Optimizer Hub service.

1. Create or use an existing 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".

SSL Configuration for Clients

Azul Zing Builds of OpenJDK (Zing) can connect both with (default) or without SSL to Optimizer Hub.

Running Zing Clients with SSL

By default, Zing connects to Optimizer Hub using SSL.

Make sure the service certificate is trusted by the client machine where you run Zing. 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.

To make sure an authority is trusted usually involves copying its certificate file to `/usr/local/share/ca-certificates/` or `/etc/ssl/certs/`. The exact path and process depends on your OS distribution. Follow the instructions for your OS distribution to register the certificate on your client machine. 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 Zing 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)
```

Running Zing Clients without SSL

NOTE	Using Optimizer Hub without SSL must only be used for testing.

If you installed Optimizer Hub without enabling SSL, you must use the `-XX:-OptHubUseSSL` flag to instruct Zing to allow unsecured connections to Optimizer Hub.

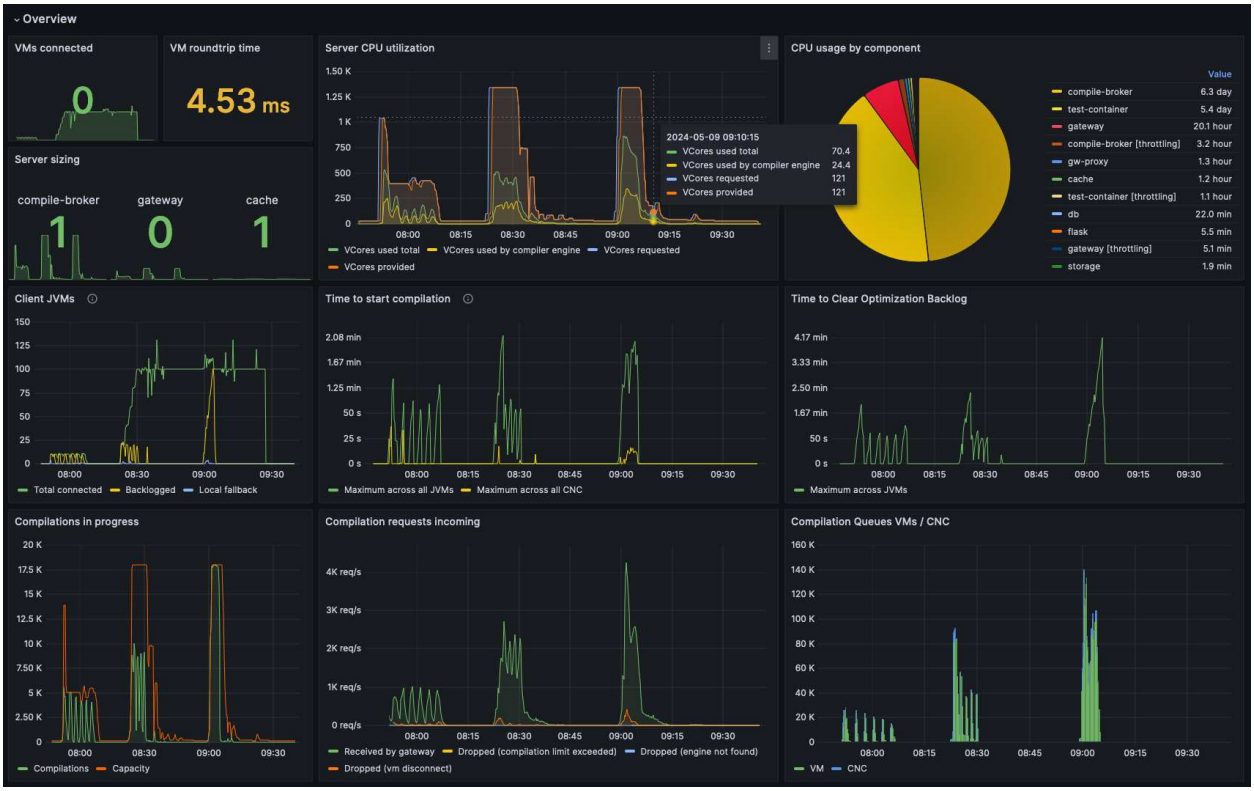
> **NOTE**
>
> Before version 1.8.0 the flag was called `-XX:+/-CNCInsecure`. Because of this change, you need to review your settings.

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.
```

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:

```
# 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. In opthub-install-1.11.2.zip > `grafana/opthub_dashboard.json`, you can find a Grafana configuration file.

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 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 minimum and maximum number of vCores you wish to allocate to the service. The Helm chart automatically sets the sizing of the individual Optimizer Hub components.

Service Scaling

Optimizer Hub can be configured to run one ore multiple services, see "Configuring the Active Optimizer Hub Services". According to the selected services, different scaling approaches are required.

Cloud Native Compiler (CNC)

The CNC service must be able to autoscale rapidly to handle resource demands effectively. From time-to-time, depending on the number of starting applications, it needs a large amount of resources to be able to perform all requested compilations in time. As such, it must scale up according to the needs, but also scale down quickly when resources are no longer needed as it's prohibitively expensive to keep those resources always on.

ReadyNow Orchestrator (RNO)

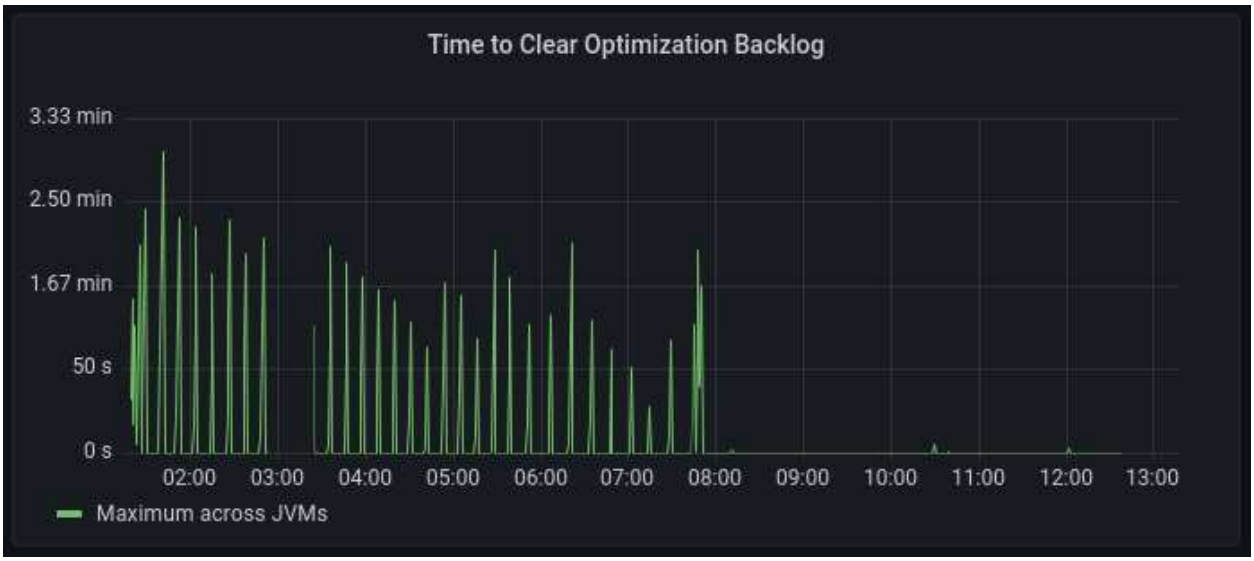
When Optimizer Hub is configured on RNO-only mode (using `values-disable-compiler.yaml`, see "Configuring the Active Optimizer Hub Services"), it doesn't need to scale. The predefined sizing will be able to handle full RNO functionality.

Combined Services

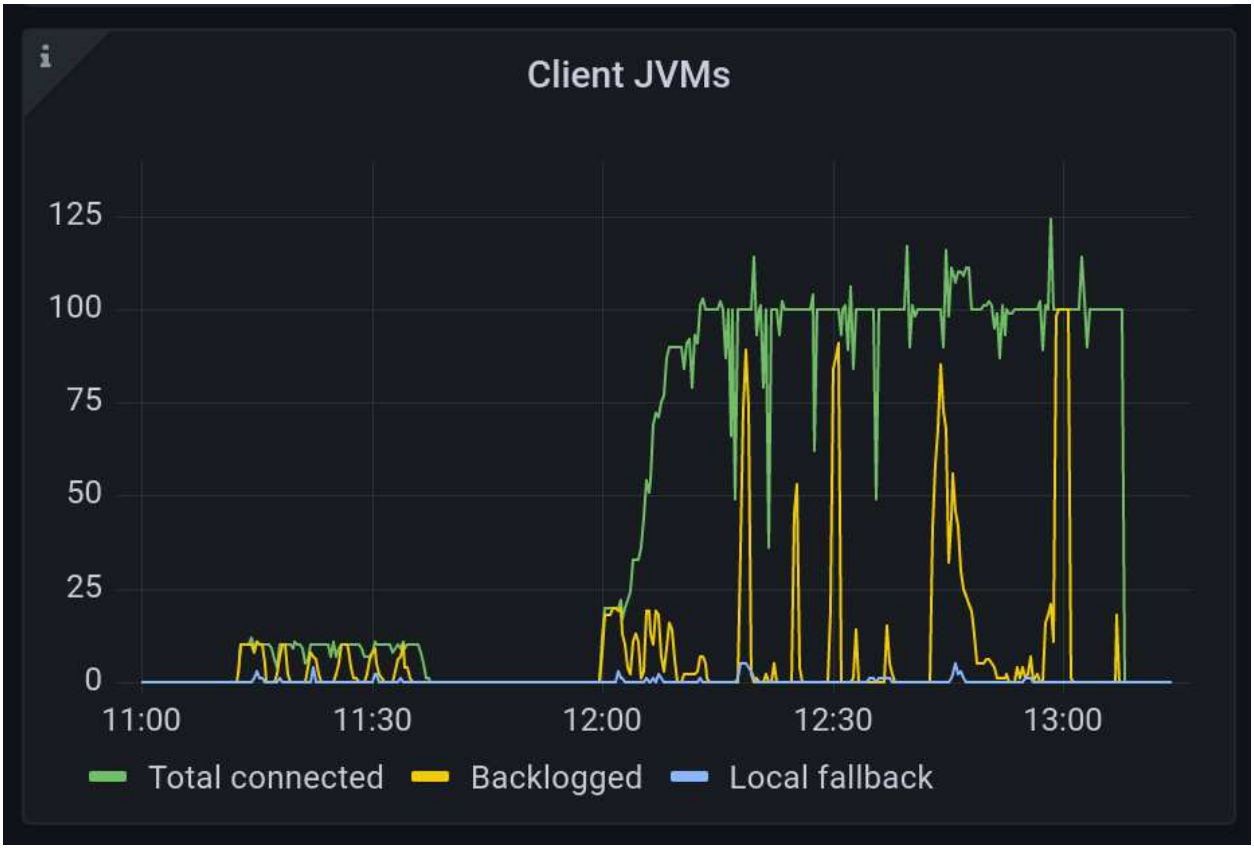
When both CNC and RNO are enabled, but you only use RNO, the Optimizer Hub service may in some rare cases of extreme traffic scale up more instances.

How Optimizer Hub Scales

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 and all requests are started within 2 seconds.



By default, Optimizer Hub is configured to use autoscaling. You can control autoscaling by specifying the minimum number of vCores for the entire Optimizer Hub installation. The minimum vCores for an Optimizer Hub installation, including a management-gateway pod and one compile-broker pod, is 39 vCores. If you want more compilation capacity, increase `minVCores`.

The maximum number of vCores, configured by `maxVCores`, defines the maximum number of vCores over which the Optimizer Hub service will not scale regardless of how much load it is under.

These values can be defined by overriding the default values in your `values-override.yaml` file.

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

The minimum and maximum number of vCores is used by the Optimizer Hub service to

adjust the sizing of the instance to try to meet your `timeToClearOptimizationBacklog` limit for all the JVMs that request compilations.

NOTE	Optimizer Hub uses a custom Kubernetes operator to scale and does not use Kubernetes Horizontal Pod Autoscalers.

Scaling API

The api-methods allows you to instrument Optimizer Hub to temporary increase the minimum number of vCPUs between a start and end timestamp. Multiple calls can be made to this API and Optimizer Hub will take all given timestamps and potential overlaps into account to start and stop the extra resources.

JVM Connections to Optimizer Hub

Connecting a JVM to Optimizer Hub

You can use an Optimizer Hub instance to provide compilations with Cloud Native Compiler, reading and writing profile logs with ReadyNow Orchestrator, or both.

Ask your Optimizer Hub instance admin for the "host (DNS or IP) and optional port of the Optimizer Hub service" and enter it in the `-XX:OptHubHost=<host>[:<port>]` JVM parameter flag to create a connection between the JVM and the Optimizer Hub instance.

For additional flags, see:

- "Using Cloud Native Compiler"
- "Using ReadyNow Orchestrator"

NOTE	Establishing a connection to Optimizer Hub does not force the JVM to fetch compilations from Optimizer Hub and not perform compilations locally by default.

Using Cloud Native Compiler

You configure an Azul Zing Build of OpenJDK (Zing) 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 Zing 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={value}` and `-XX:+CNCEnableRemoteCompiler`.

Command Line Option	Description	Default
-XX:OptHubHost=<host>[:<port>]	`<host>` is the DNS name or IP address of the Optimizer Hub service. The part `:<port>` is optional, with port 50051 being used by default. See "Connecting a JVM to Optimizer Hub" for instructions to determine the correct host and port.	null
-XX:[+/-]CNCEnableRemoteCompiler	Allows usage of the remote compiler when Cloud Native Compiler has established a connection. Requires `OptHubHost`.	`false`
-XX:CNCEngineUploadAddress=<host:port>	Address to upload the compiler engine. Only needed when your Optimizer Hub has non-standard ports. **NOTE** Obsolete for 1.10 and above.	

Command Line Option	Description	Default
-XX:CNCLocalFallbackOptLevelLimit=<3,2,1,0>	Limit the OptLevel for local compilations when Cloud Native Compiler is enabled. See the Zing documentation > Falcon Compiler Options > -XX:FalconOptimizationLevel.	`3`
-XX:[+/-]OptHubUseSSL	Instructs the Zing 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 Zing 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 a Zing 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.
- 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 a Zing 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 Zing 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 Zing 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.
```

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.

Advantages of ReadyNow Orchestrator

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.
- JVMs provide profile logs, outputting them to Optimizer Hub ReadyNow Orchestrator on an ongoing basis as the JVM experience evolves.
    - JVMs designate a set of criteria for nominating their individual profile logs as candidates for promotion with criteria chosen by the JVM that are configurable, with some defaults.
- ReadyNow Orchestrator handles recording multiple profile candidates from multiple JVMs and promoting the best recorded profile log.
    - The ReadyNow Orchestrator considers all provided logs that meet their specific JVM's nomination criteria and meet the ReadyNow Orchestrator's own promotion criteria that are configurable, with some defaults.
    - 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.
    - Within the qualifying candidates (eligible for nomination and promotion, per criteria), Optimizer Hub ReadyNow Orchestrator picks a specific log as the "currently prompted" profile log, based on rules for picking the promoted log from within the qualifying candidates are. For example, the longest qualifying profile log in the largest generation.
- When a JVM connects to the Optimizer Hub ReadyNow Orchestrator service with a profile name, it is provided with the currently promoted profile log (if one exists with that profile name) as an input.
- 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>]	`<host>` is the DNS name or IP address of the Optimizer Hub service. The part `:<port>` is optional, with port 50051 being used by default. See "Connecting a JVM to Optimizer Hub" for instructions to determine the correct host and port.	null
-XX:ProfileName=<value>	Name of the profile that the JVM both reads from and writes to. Use of this flag is equivalent to using `-XX:ProfileLogIn=<value>` `-XX:ProfileLogOut=<value>`, and is the preferred way to specify profile names when different input and output names are not needed.	null
-XX:[+/-]EnableRNO	Enables ReadyNow read and write to ReadyNow Orchestrator. Requires `OptHubHost` and uses `ProfileName` as the name for the profile log.	`false`

Command Line Option	Description	Default
-XX:ProfileLogOut=<value>	The ProfileLogOut enables Zing to record compilations from the current run. `<value>` is the name of the profile that the JVM reads as input to ReadyNow. If prefixed with `opthub://`, `<value>` is used as the profile name in ReadyNow Orchestrator. If not prefixed with `opthub://`, `<value>` 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=<value>	The ProfileLogIn allows Zing to base its decisions on the information from a previous run. The current ProfileLogIn file information is read in its entirety - before Zing starts to create a new ProfileLogOut log. `<value>` is the name of the profile that the JVM reads as input to ReadyNow. If prefixed with `opthub://`, `<value>` is used as the profile name in ReadyNow Orchestrator. If not prefixed with `opthub://`, `<value>` 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
-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

Command Line Option	Description	Default
-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
-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 parameter has a server-side counterpart `readyNowOrchestrator.producers.maxPromotableGeneration`. The profile has to satisfy both settings to be promoted. `0` = unlimited For more information, check "Understanding ReadyNow Orchestrator Generations".	0
-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 parameter has a server-side counterpart `readyNowOrchestrator.producers.maxProfileSize`. The profile will be allowed to reach whatever is the smallest of both settings. 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
-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
%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=%prop=myprofilename%`

Registering a New Compiler Engine in Cloud Native Compiler

Since different versions of Azul Zing Builds of OpenJDK (Zing) 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 Zing. This compiler is uploaded to Cloud Native Compiler from the JVM in the form of a Compiler Engine.

Each version of Zing 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 a Zing 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.

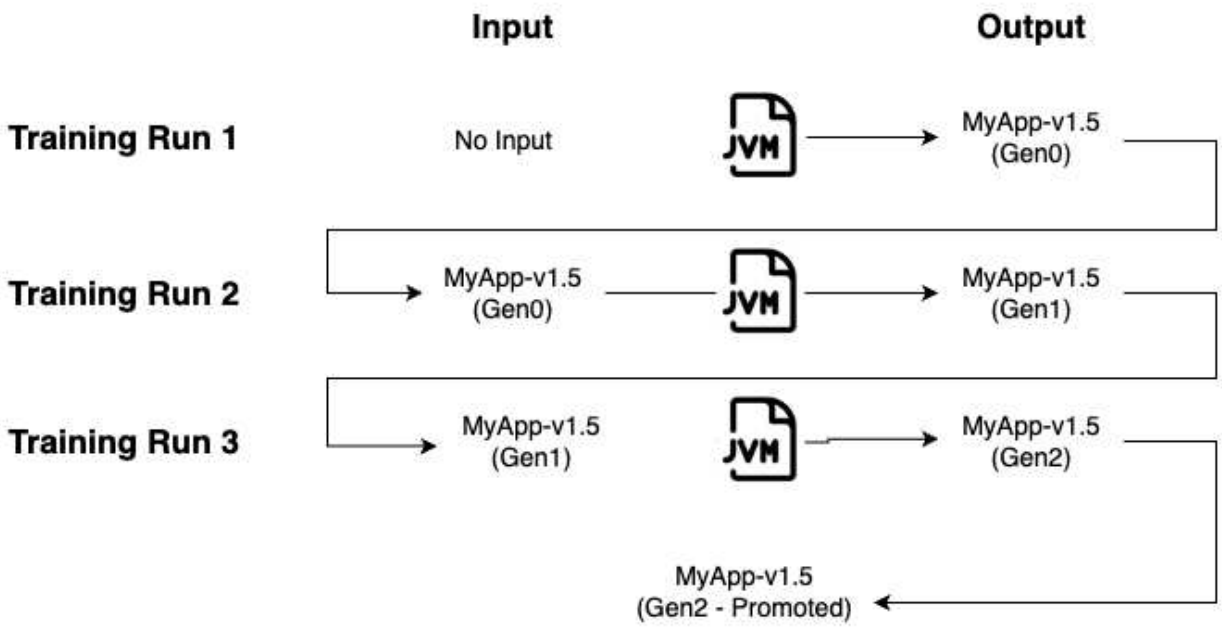
NOTE	If a JVM requests compilations from Cloud Native Compiler that does not have the corresponding compiler engine, the JVM switches to local JIT compilation and starts auto-uploading the compiler engine for future use.

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.

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=MyApp-v3 \
    -jar myapp.jar
```

In this case, all JVMs nominate their logs based on the defaults (2 minutes for the first generation, 15 minutes for the second, 30 minutes for the third, 60 minutes for the fourth) 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=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=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 API for several management tasks.

These methods are available on `{MANAGEMENT_GATEWAY_IP}:{SERVICE_PORT}/…`

and can be accessed without authentication. The service port typically is 8080. For security reasons, by default, the API is not exposed outside the cluster. Your network administrator can provide you secure access to this endpoint.

API Methods

Please check the documentation website (docs.azul.com) for the OpenAPI documentation.

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`:

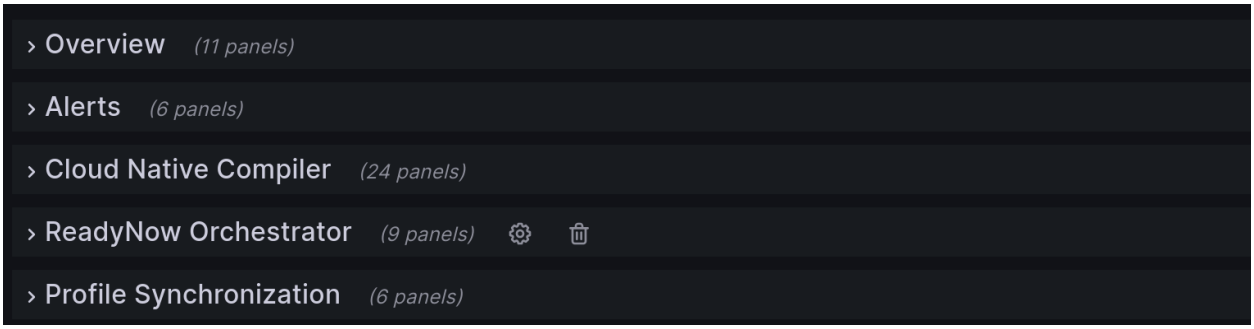
```
# Log command-line option
-Xlog:concomp=info:file=connected-compiler-%p.log::filesize=500M:filecount=20

# 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

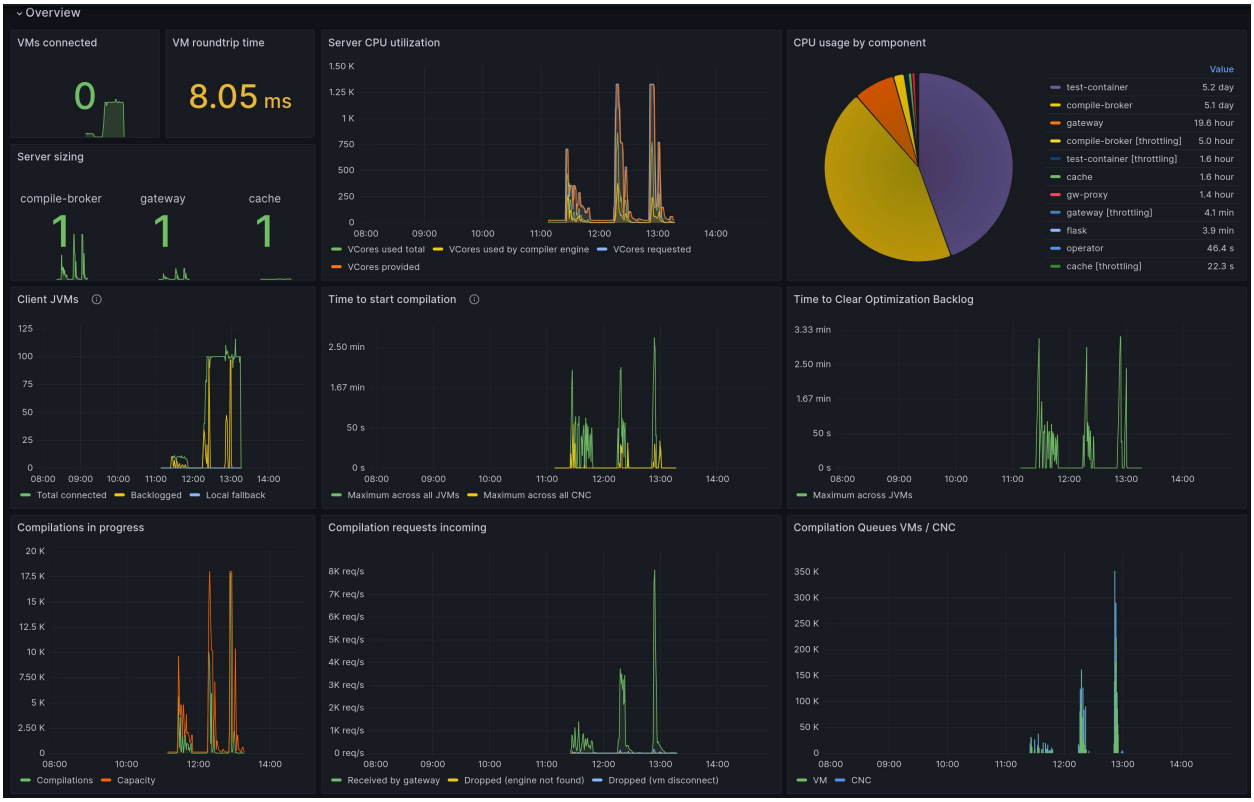
A Grafana dashboard is available after it has been "configured", to understand how your Optimizer Hub instance is performing. This dashboard is divided into several sections. The most important sections from user-perspective are described here. The other sections are more oriented towards maintaining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



Overview

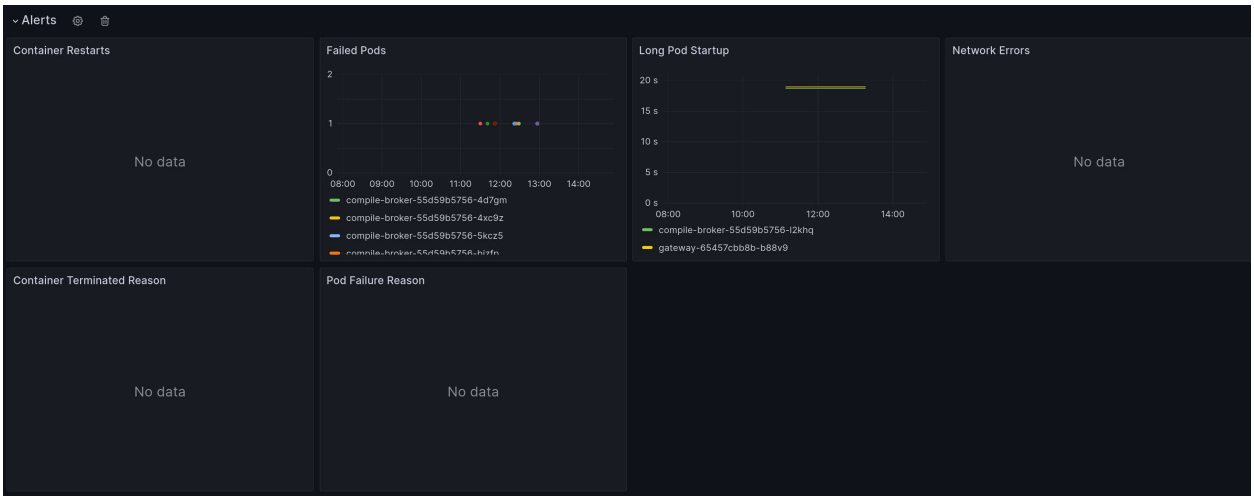
Provides a high-level view of the Optimizer Hub instance:

- 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.



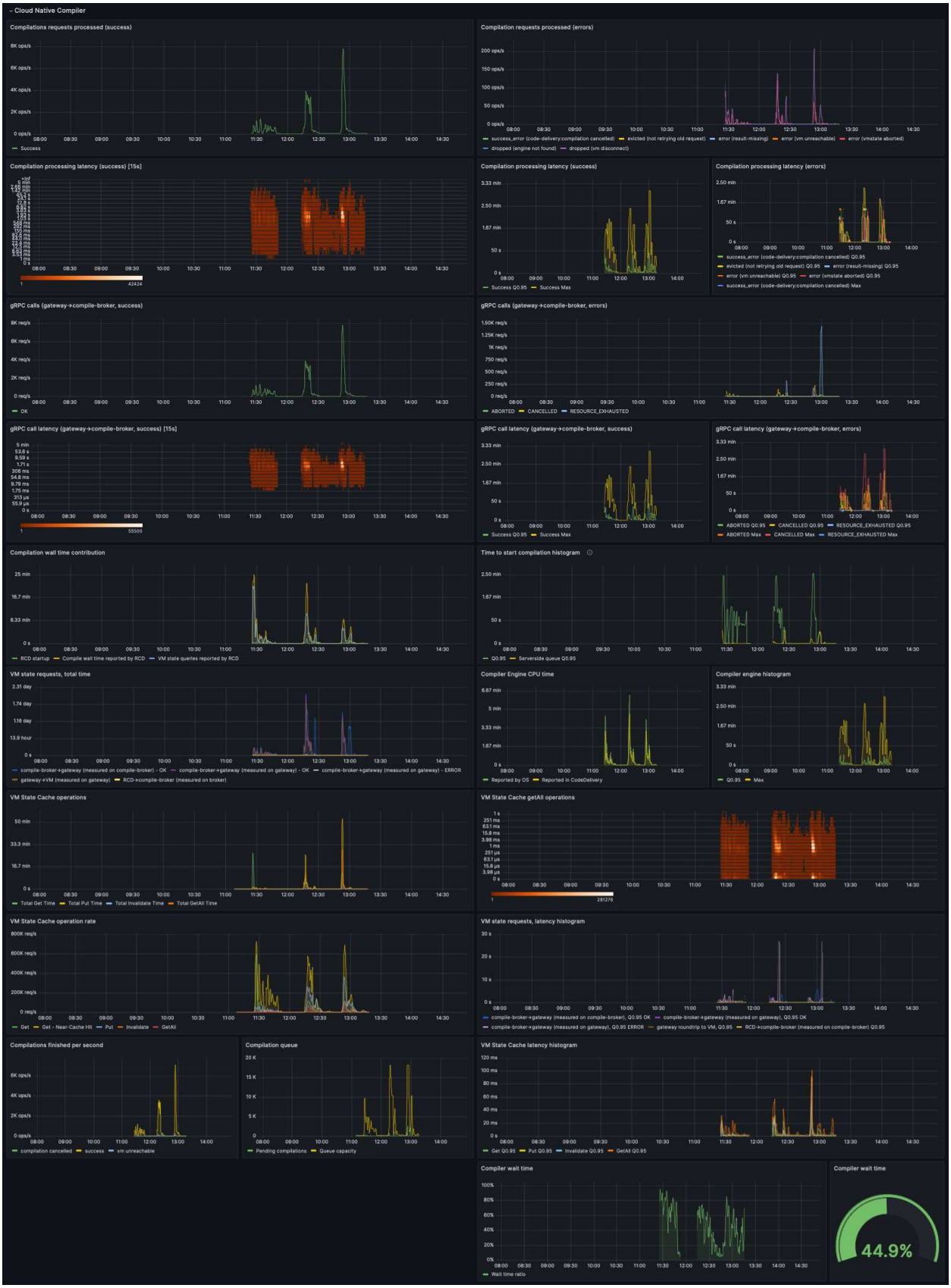
Alerts

Basic monitoring of problems in the Optimizer Hub cluster.



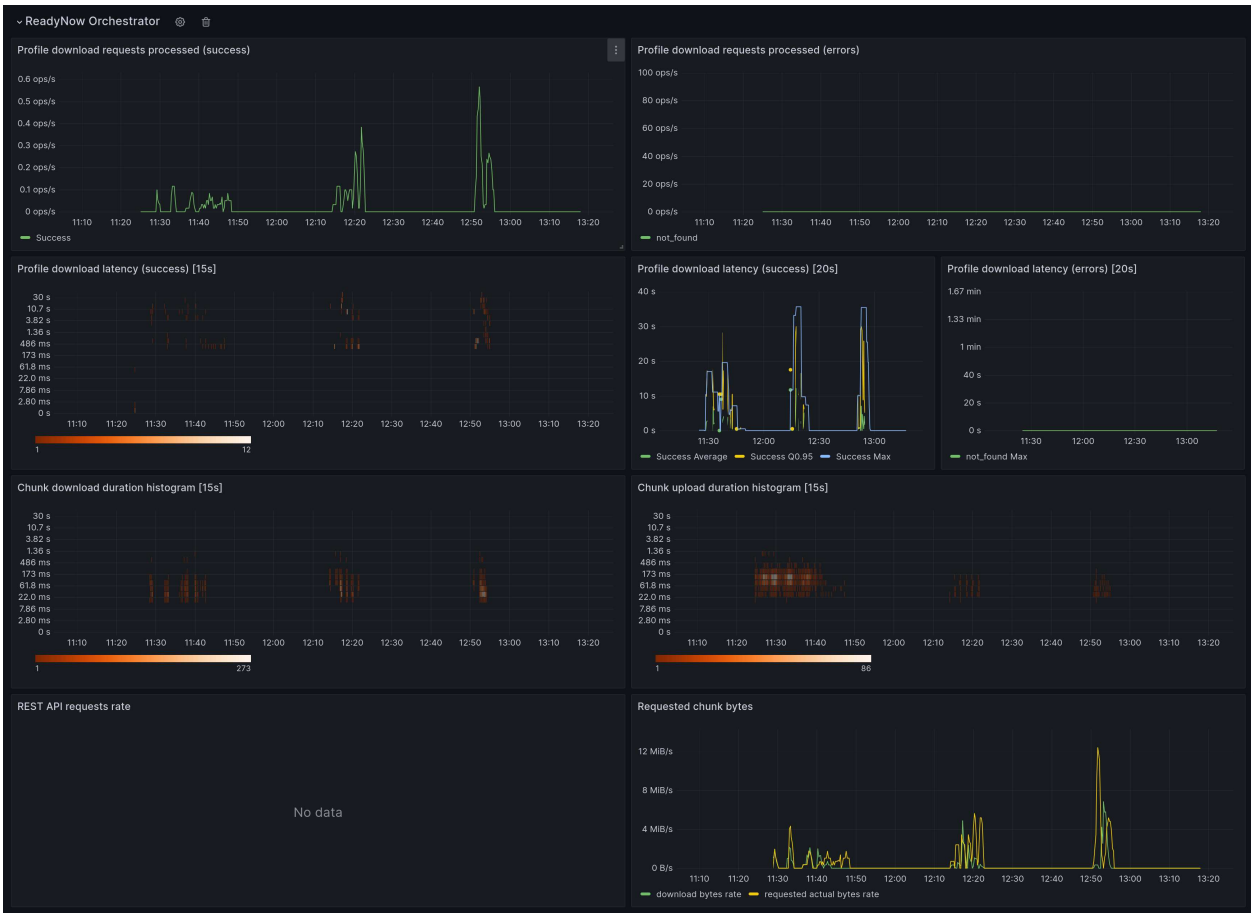
Cloud Native Compiler

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


Cloud Native Compiler
Compilations requests processed (success)
Compilation requests processed (errors)
Compilation processing latency (success) [15s]
Compilation processing latency (success)
Compilation processing latency (errors)
gRPC calls (gateway→compile-broker, success)
gRPC calls (gateway→compile-broker, errors)
gRPC call latency (gateway→compile-broker, success) [15s]
gRPC call latency (gateway→compile-broker, success)
gRPC call latency (gateway→compile-broker, errors)
Compilation wall time contribution
Time to start compilation histogram
VM state requests, total time
Compiler Engine CPU time
Compiler engine histogram
VM State Cache operations
VM State Cache getAll operations
VM State Cache operation rate
VM state requests, latency histogram
Compilations finished per second
Compilation queue
VM State Cache latency histogram
Compiler wait time
Compiler wait time
44.9%

ReadyNow Orchestrator

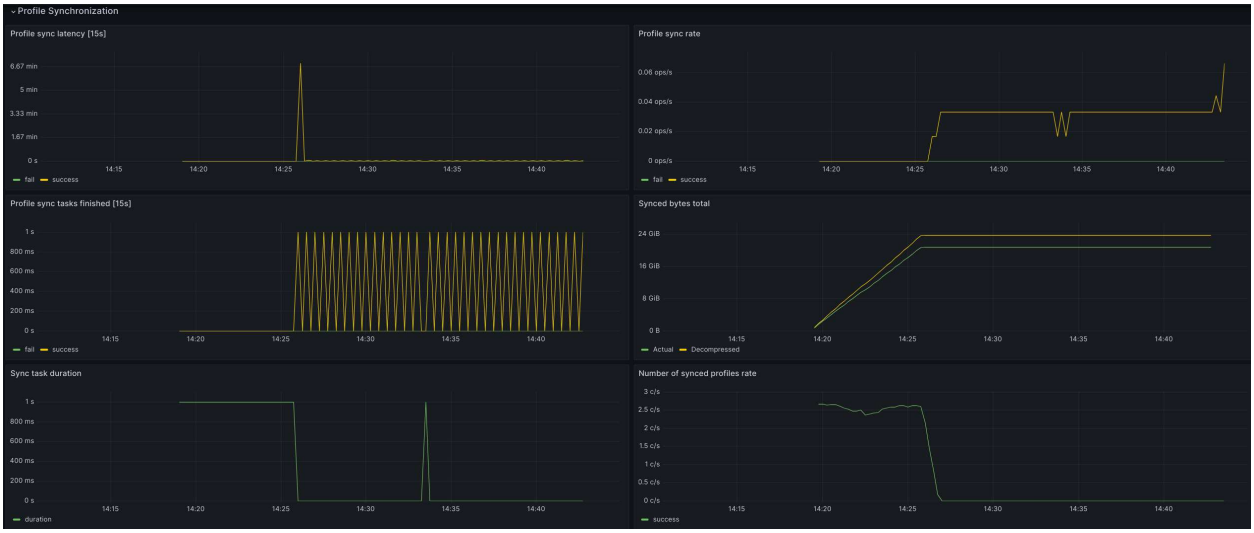
Monitoring of the ReadyNow Orchestrator feature. This is applicable for JVMs using Optimizer Hub ReadyNow Orchestrator.



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 Zing Build of OpenJDK (Zing) instances using Optimizer Hub.

Client VM Troubleshooting

My application gc.log contains `PROFERR Failed to connect with the server` and/or `PROFERR Unable to load remote profile`.

There is probably no `-XX:OptHubHost` specified, or an incorrect address of the server is provided. If no host is specified, the default value `localhost:50051` is used instead of the correct address of the Optimizer Hub service. Please check "Using ReadyNow Orchestrator" for more information.

This can also be caused by trying to establish a TLS-encrypted connection with `-XX:+OptHubUseSSL` to a server which expects unencrypted connections, or vice versa.

Double-check your VM arguments. Ensure that VM is started with the `-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.

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 the `-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`:

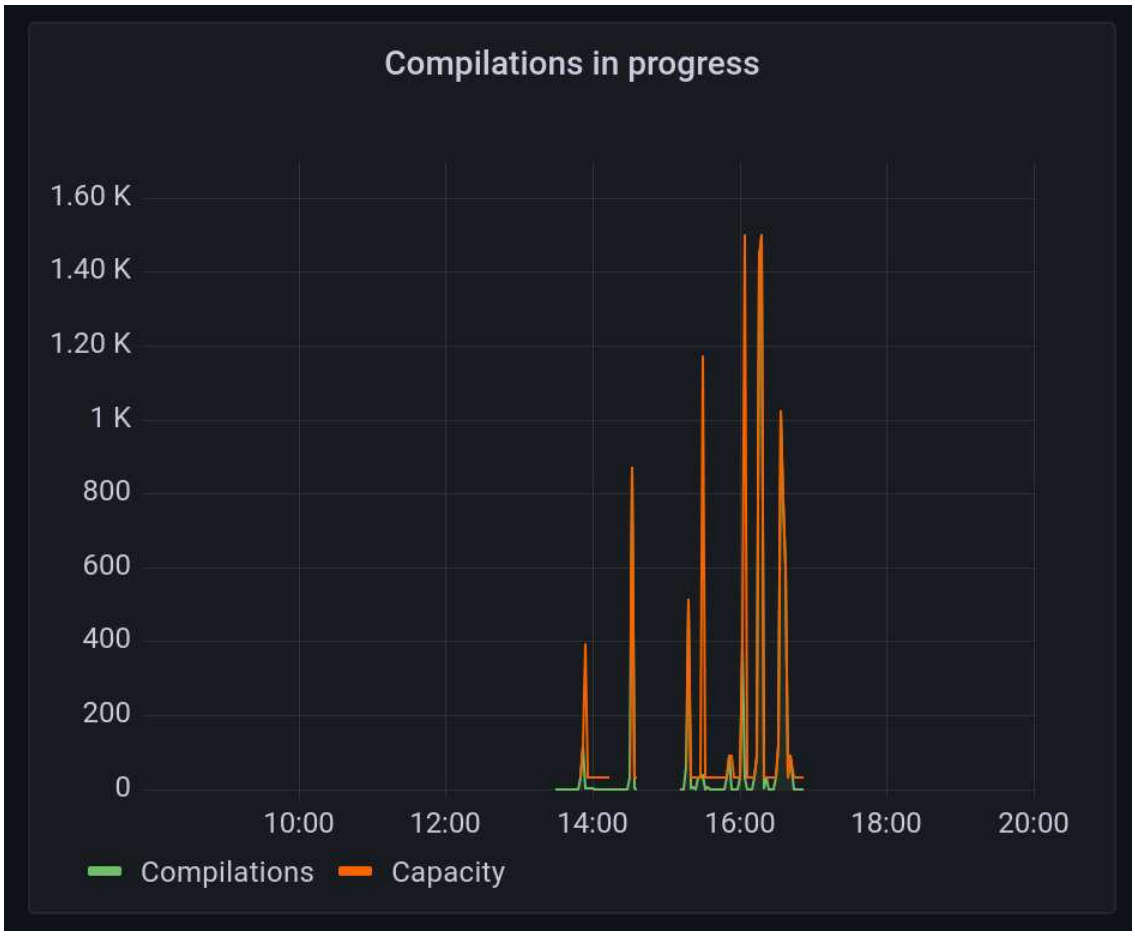
```
# Log command-line option
-Xlog:concomp=info:file=connected-compiler-%p.log::filesize=500M:filecount=20

# 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.

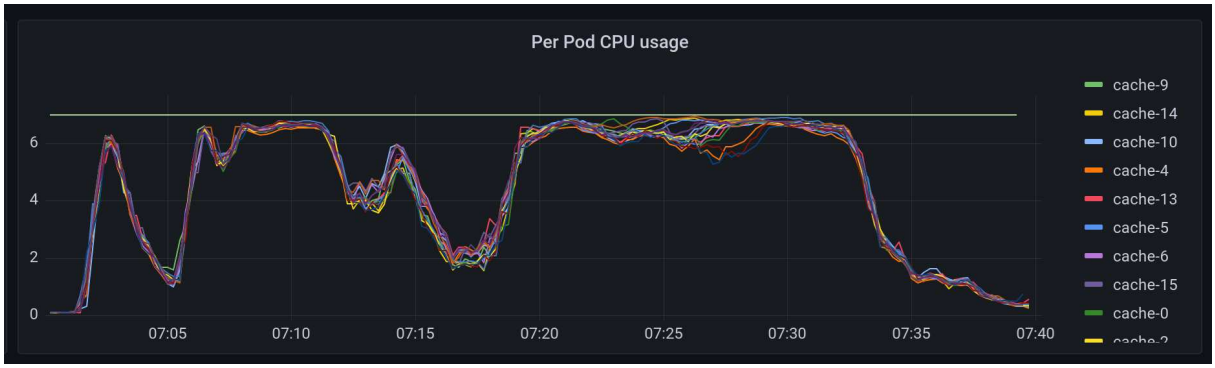
   An overloaded client (the JVM) can cause worse performance of Cloud Native Compiler. This could be seen as a too high TTCOB metric. One example of such overload is CPU saturation on JVM side. This can cause smaller amounts of compilations being sent to Cloud Native Compiler but also a worse performance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