

Optimizer Hub

Release 26.05.5

Table of Contents

About Optimizer Hub 1

Architecture Overview 1

Interaction Between Optimizer Hub and JVMs 1

Deployment Options 2

Full Installation 2

ReadyNow Orchestrator Only 3

About Cloud Native Compiler 4

JIT Optimization 4

Enhanced Optimization Capabilities 4

Improved Application Performance from the Start 4

Optimized Resource Allocation and Cost Efficiency 5

Falcon JIT with CNC 5

About ReadyNow Orchestrator 6

Key Strengths of ReadyNow Orchestrator 6

Video Introduction of ReadyNow Orchestrator 7

About Compilation Streaming 7

Objectives 7

How Streaming Changes Warm-up 7

Components 7

Stream Build Triggers 8

What to Expect on the First Runs 8

Compatibility 9

Memory 9

Also Read 9

High Availability 9

High Availability in Clusters 9

Inside a Cluster 10

Between Clusters 10
High Availability Configuration 10
Installation Prerequisites **11**
Kubernetes Requirements 11
Deployment Modes 11
Kubernetes Cluster 11
Kubernetes Pods 12
Kubernetes Nodes 13
Network and Gateway Requirements 13
General Network Requirements 13
Load Balancing 14
Benefits of a Load Balancer 14
Load Balancer Requirements 14
Gateway Connection Capacity 14
Network Latency and Upload Throughput 15
Configuring the Optimizer Hub Host 16
Host for Single Optimizer Hub service 16
Host for High Availability and Failover 18
Configuring the Gateway 18
Gateway Health Check 19
Storage Requirements 20
Configuring AWS S3 Storage 20
Using Kubernetes Nodes and Permissions 21
Using AWS Service Accounts 21
Storage for ReadyNow Orchestrator 23
Configuring GCP Blob Storage 23
IAM Policy Update 23
IAM Policy Binding 24

Configuring Azure Blob Storage 24
Storage for ReadyNow Orchestrator 25
Configuring S3 Compatible Storage 25
Specifying Bucket Location 26
Required Roles and Permissions 26
Operator Pod Permissions 27
Cache Pod Permissions 27
Using Externally Defined Secrets 28
How To Use 28
Example 28
Using an Internal Docker Registry 29
Installation, Upgrade, Uninstallation 29
Standard Optimizer Hub Installation Procedure on Kubernetes 29
Installation Steps 30
Step 1: Prepare Helm Repository 30
Step 2: Create Namespace 30
Step 3: Create Configuration File 30
Step 4: Decide on the Monitoring Approach 31
Step 5: Execute Installation 31
Step 6: Verify Installation 32
Installing Optimizer Hub without Cloud Native Compiler 32
Resource Requirements 32
Install Only ReadyNow Orchestrator 33
Disabling Cloud Native Compiler on a Full Optimizer Hub Installation 33
Installing Optimizer Hub with Compilation Streaming 34
Step 1: Enable Compilation Streaming on the Server 34
New Installation 35
Existing Installation 35

Step 2: Verify the Server . . . 35
Step 3: Enable Compilation Streaming on the JVM . . . 35
Step 4: Verify the Client . . . 36
Compilation Streaming Requires a Promoted Profile . . . 36
Platform-Specific Installation Steps . . . 36
AWS EKS . . . 36
AWS EKS Cluster Requirements . . . 37
Microsoft Azure . . . 37
Google Cloud Platform . . . 37
Other Kubernetes Platform . . . 37
Local Installation . . . 38
Installing on MicroK8s . . . 38
Installing MicroK8s . . . 38
Installing Optimizer Hub on MicroK8s . . . 39
Upgrading Optimizer Hub on MicroK8s . . . 44
Uninstalling Optimizer Hub from MicroK8s . . . 45
Installing on minikube . . . 45
Installing minikube . . . 45
Installing Optimizer Hub on minikube . . . 46
Uninstalling Optimizer Hub from minikube . . . 51
Upgrading or Uninstalling Optimizer Hub . . . 51
Upgrading or Rolling Back . . . 51
Uninstalling . . . 52
Configuration Management . . . 52
Optimizer Hub Generic Defaults . . . 52
Management Gateway Parameters . . . 52
Additional Labels for Kubernetes Objects . . . 53
ReadyNow Orchestrator Defaults . . . 53

Cross-Region Sync Parameters . . . 55
Compile Artifacts Auto Cleanup Parameters . . . 56
Blob Storage Auto Cleanup Parameters . . . 56
Externally Defined Secrets Parameters . . . 57
Simple Sizing Parameters . . . 58
Compilations Parameters . . . 58
Compilation Streaming Parameters . . . 59
SSL Parameters . . . 60
Size and Duration Formats . . . 61
Specifying Size Values . . . 61
Specifying Durations . . . 61
Configuring ReadyNow Orchestrator . . . 61
Understanding ReadyNow Orchestrator Generations . . . 62
Configuring Generations . . . 62
Basic Profile Recording with Default Generations . . . 63
Capping Profile Log Recording and Maximum Generations . . . 64
Priority of Generation Settings . . . 64
Uploading a Profile through the API . . . 65
Indirect Delivery of ReadyNow Profiles . . . 66
Configuring Cross-Region Synchronization of Profiles . . . 67
Configuring Blob Storage Auto Cleanup . . . 68
Code Cache Cleanup . . . 68
How the Code Cache Cleaner Works . . . 68
Check Used Code Cache Storage Size . . . 68
ReadyNow Profile Log Cleanup . . . 69
How the ReadyNow Cleaner Works . . . 70
Size Value Formats . . . 70
Configuring Optimizer Hub SSL Authentication . . . 71

SSL Configuration in Optimizer Hub 71
SSL Configuration for Clients. 72
Running Zing Clients with SSL 73
Running Zing Clients without SSL 73
Sizing and Scaling your Optimizer Hub Installation 74
Service Scaling 74
Full Optimizer Hub with Cloud Native Compiler (CNC) 74
ReadyNow Orchestrator (RNO) Only 75
How Optimizer Hub Scales 75
Controlling the Scaling with vCores 76
Scaling of the Gateway Service 77
Scaling API. 78
Connecting JVM to Optimizer Hub. 78
Connecting a JVM to Optimizer Hub 78
Using Cloud Native Compiler 79
Fallback to Local JIT Compilation. 79
Fallback Reasons 79
Local Fallback Baseline and Configuration 80
Performance-Based Local Fallback 82
Automatic Switch Back to Cloud Native Compiler 83
Logging and SSL 84
Cloud Native Compiler JVM Options 84
Using ReadyNow Orchestrator 86
Advantages of ReadyNow Orchestrator 86
Creating and Writing To a New Profile Name. 87
ReadyNow Orchestrator JVM Options 88
Substitution Macros. 94
Using Compilation Streaming. 95

Enabling Compilation Streaming . . . 96
Configuration Options . . . 97
Storage Requirements . . . 97
Stream Segments . . . 98
Compilation Streaming and Indirect Delivery . . . 98
Monitoring . . . 99
Registering a New Compiler Engine in Cloud Native Compiler . . . 99
Auto-Uploading Compiler Engines . . . 100
Monitoring and Alerting . . . 100
Monitoring Optimizer Hub . . . 100
Using Prometheus and Grafana . . . 100
Configuring Prometheus Path . . . 100
Using the REST APIs . . . 100
Retrieving Optimizer Hub Logs . . . 103
Extracting Compilation Artifacts . . . 104
Note About `gw-proxy` Metrics . . . 105
Configuring Prometheus and Grafana . . . 105
Using the Integrated Prometheus and Grafana . . . 106
Using Your Own Prometheus and Grafana . . . 107
Prometheus Configuration Instructions . . . 107
Grafana Configuration Instructions . . . 108
Using the Grafana Dashboard . . . 109
Overview . . . 109
Alerts . . . 110
Cloud Native Compiler . . . 110
ReadyNow Orchestrator . . . 111
Profile Synchronization . . . 112
Optimizer Hub API . . . 112

API Methods . . . 112
Troubleshooting Optimizer Hub . . . 113
Client VM Troubleshooting . . . 113
Cloud Native Compiler Troubleshooting . . . 118
ReadyNow Orchestrator Troubleshooting . . . 119
Optimizer Hub API Troubleshooting . . . 120
Known Issues . . . 120
Optimizer Hub Release Notes . . . 120
Optimizer Hub 26.08.4 . . . 120
New Features . . . 120
CVE Fixes . . . 120
Optimizer Hub 26.08.3 . . . 121
New Features . . . 121
Known Issues . . . 121
CVE Fixes . . . 121
Optimizer Hub 26.08.2 . . . 122
New Features . . . 122
Known Issues . . . 122
Optimizer Hub 26.08.1 . . . 122
New Features . . . 122
Bug Fixes . . . 122
CVE Fixes . . . 123
Optimizer Hub 26.08.0 . . . 123
New Features . . . 124
CVE Fixes . . . 125
Optimizer Hub 26.05.5 . . . 126
New Features . . . 126
CVE Fixes . . . 126

Optimizer Hub 26.05.4 126
New Features. 126
CVE Fixes 126
Optimizer Hub 26.05.3 126
New Features. 127
Known Issues 127
CVE Fixes 127
Optimizer Hub 26.05.2 127
New Features. 127
Known Issues 127
Optimizer Hub 26.05.1 128
New Features. 128
CVE Fixes 128
Optimizer Hub 26.05.0 129
New Features. 129
Bug Fixes 129
CVE Fixes 129
Optimizer Hub 26.02.1 131
New Features. 131
CVE Fixes 131
Optimizer Hub 26.02.0 131
New Features. 131
CVE Fixes 132
Optimizer Hub 25.11.1 132
New Features. 132
CVE Fixes 132
Optimizer Hub 25.11.0 133
New Features. 133

CVE Fixes 134
Optimizer Hub 25.08.1 134
New Features. 134
Bug Fixes 134
CVE Fixes 135
Optimizer Hub 25.08.0 135
New Features. 135
Optimizer Hub 25.05.2 136
New Features. 136
Bug Fixes 136
CVE Fixes 136
Optimizer Hub 25.05.1 137
New Features. 137
Bug Fixes 137
CVE Fixes 137
Optimizer Hub 25.05.0 138
New Features. 138
API Changes 138
Bug Fixes 139
Optimizer Hub 1.11.6. 139
New Features. 139
Bug Fixes 139
CVE Fixes 139
Optimizer Hub 1.11.5. 140
New Features. 140
CVE Fixes 140
Optimizer Hub 1.11.4. 140
New Features. 140

Bug Fixes 140
Optimizer Hub 1.11.3 141
New Features 141
Bug Fixes 141
CVE Fixes 141
Optimizer Hub 1.11.2 141
New Features 141
Bug Fixes 141
Optimizer Hub 1.11.1 142
New Features 142
Bug Fixes 142
CVE Fixes 142
Optimizer Hub 1.11.0 142
New Features 143
Bug Fixes 144
Optimizer Hub 1.10.2 144
New Features 144
CVE Fixes 144
Optimizer Hub 1.10.1 145
New Features 145
Optimizer Hub 1.10.0 145
New Features 145
Bug Fixes 146
Known Issues 147
Optimizer Hub 1.9.5 147
New Features 147
Default Configuration Changes 147
Optimizer Hub 1.9.4 147

New Features. 147
Bug Fixes 147
Known Issue 148
Optimizer Hub 1.9.3. 148
New Features. 148
Grafana Dashboard Update. 148
Bug Fixes 148
Known Issue 149
Optimizer Hub 1.9.2. 149
New Features. 149
API improvements 149
Grafana Dashboard Update. 149
Known Issue 149
Profile Sync Running Indefinitely 149
Optimizer Hub 1.9.1. 149
New Features. 150
Grafana Dashboard Update. 150
Configurable Minimal Client Version 150
Increased Number of Concurrent Recordings 150
Continuous Recording 150
Optimizer Hub 1.9.0. 150
New Features. 150
Cross-Region Synchronization of ReadyNow Orchestrator Profiles 150
Database Changes. 151
New Location of REST APIs and ReadyNow Profile Cleaner 151
Prioritization of Profile Generations. 151
Grafana Dashboard 151
Support for Zing Running on ARM 151

Bug Fixes 152
Optimizer Hub 1.8.2 152
New Features 152
Optimizer Hub 1.8.1 152
New Features 152
Known Issues 152
Optimizer Hub 1.8.0 152
New Features 153
Known Issues 153
Fixed Ports for gRPC 154
Cloud Native Compiler 1.7.1 154
New Features 154
Cloud Native Compiler 1.7.0 155
New Features 155
Cloud Native Compiler 1.6.3 156
New Feature 156
Cloud Native Compiler 1.6.2 156
New Features 156
Upgrade 156
Cloud Native Compiler 1.6.1 156
New Features 156
Bug Fixes 156
Known Issues 157
Cloud Native Compiler 1.6.0 157
New Features 157
Bug Fixes 157
Known Issues 157
Cloud Native Compiler 1.5.0 158

New Features. 158
Known Issues 158
Cloud Native Compiler 1.4.0 158
New Features. 158
Known Issues 158
Cloud Native Compiler 1.3.0 158
New Features. 158
Known Issues 159
Cloud Native Compiler 1.2.0 159
New Features. 159
Cloud Native Compiler 1.1.0 159
New Features. 159
Known Issues 159
Cloud Native Compiler 1.0.0 160
New Features. 160
Azul Prime Optimizer Hub Third Party Licenses 160

About Optimizer Hub

Architecture Overview

Optimizer Hub is a component of Azul 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 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.

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

Interaction Between Optimizer Hub and JVMs

Azul Zing Builds of OpenJDK (Azul Zing) integrates with Optimizer Hub to offload JIT compilation and manage ReadyNow profiles. The following diagram shows how the ReadyNow and Falcon Compiler components in Azul Zing interact with the ReadyNow Orchestrator and Cloud Native Compiler services in Optimizer Hub.

```
sequenceDiagram
    box Azul Zing
    participant RN as ReadyNow
    participant FC as Falcon Compiler
    end
    box Optimizer Hub
    participant RNO as ReadyNow Orchestrator
    participant CNC as Cloud Native Compiler
    end

    RN->>RNO: 1. Ask for a profile
    RN->>FC: 2. Instruct what to compile
    RN->>RNO: 3. Send back new profile version
    FC->>CNC: 4. Ask to compile the code (optional)
```

```
    CNC->>FC: 5. Send back the compiled code (optional)
```

Deployment Options

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 scale the gateway, compile-broker, and cache when needed. Use the "Standard Optimizer Hub Installation Procedure on Kubernetes".

```
flowchart LR
    subgraph EXT[" "]
        direction TB
        JVMs["Application JVMs"]
        Admins(["Administrators,<br/>Tools"])
    end

    LB1(["Load Balancer"])
    LB2(["Load Balancer"])

    subgraph NS["Kubernetes Namespace"]
        direction LR
        GW["JVM Gateway<br/>7 vcore, 14 Gi<br/>(Scaling)"]
        MG["Management Gateway<br/>2 vcore, 2 Gi"]
        SO["Scaling Operator<br/>1 vcore, 2 Gi"]
        CB["Compile Broker<br/>7 vcore, 26 Gi<br/>(Scaling)"]
        Cache["Cache<br/>7 vcore, 26 Gi<br/>(Scaling)"]
        TE["Task Executor<br/>4 vcore, 16 Gi"]
        Blob[["Blob Storage"]]

        SO -.->|control| GW
        SO -.->|control| Cache
        SO -.->|control| CB
        GW --> Cache
        CB <--> Cache
        TE --> Blob
        TE --> Cache
        MG --> Blob
        MG --> Cache
        Cache --> Blob
    end

    K8sAPI[["Kubernetes API"]]
    SO -.->|control| K8sAPI

    JVMs -->|gRPC| LB1
    Admins -.->|"control over HTTP(s)"| LB2
    LB1 -->|gRPC| GW
    LB2 -.->|"control over HTTP(s)"| MG
```

```
    classDef dashed stroke-dasharray: 5 5;
    class K8sAPI,Blob,LB1,LB2 dashed;
    style EXT fill:none,stroke:none
```

Remarks:

- Dotted arrows indicate control-plane traffic (administration, scaling decisions, and Kubernetes API calls); solid arrows indicate data-plane traffic.
- 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 optimizer-host for more info.
- Task Executor runs the background jobs that build "compilation streams" and refresh the ReadyNow profiles used by indirect-delivery. Like the Scaling Operator and Management Gateway, it is a fixed-size component that does not autoscale.

ReadyNow Orchestrator Only

When you need only ReadyNow Orchestrator, you can deploy a reduced set of the Optimizer Hub components in the Kubernetes cluster. See "Installing without Cloud Native Compiler".

```
flowchart LR
    JVMs["Application JVMs"]
    Admins(["Administrators,<br/>Tools"])
    LB1(["Load Balancer"])
    LB2(["Load Balancer"])

    subgraph NS["Kubernetes Namespace"]
        direction LR
        GW["JVM Gateway<br/>7 vcore, 14 Gi<br/>(Scaling)"]
        MG["Management Gateway<br/>2 vcore, 2 Gi"]
        Cache["Cache<br/>7 vcore, 26 Gi<br/>(Scaling)"]
        TE["Task Executor<br/>4 vcore, 16 Gi"]
        Blob[["Blob Storage"]]

        GW --> Cache
        TE --> Cache
        TE --> Blob
        MG --> Cache
        MG --> Blob
        Cache --> Blob
    end

    JVMs -->|gRPC| LB1
    Admins -.->|"control over HTTP(s)"| LB2
```

```
    LB1 -->|gRPC| GW
    LB2 -.->|"control over HTTP(s)"| MG

    classDef dashed stroke-dasharray: 5 5;
    class Blob,LB1,LB2 dashed;
```

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, it can apply optimizations 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**: CNC provisions the resources for JIT compilation 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 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 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. You can configure this with a few JVM command-line parameters.
- ReadyNow Orchestrator monitors your entire fleet 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)*

About Compilation Streaming

Compilation Streaming (CoCa Streaming) delivers pre-compiled machine code to the JVM at startup. Compilations covered by the stream are installed directly, without local JIT compilation or a Cloud Native Compiler (CNC) round-trip. Compilations not covered by the stream fall back to CNC as usual.

Compilation Streaming is available starting with Optimizer Hub 26.08.0 and requires Azul Zing 26.08.1.0 or later.

Objectives

The primary goal is to reduce Optimizer Hub resource consumption and scaling spikes under load. The secondary goal is to improve warm-up.

How Streaming Changes Warm-up

Without streaming, the JVM requests each compilation from Optimizer Hub individually during warm-up. Even with a high Code Cache (CoCa) hit rate, every compilation must be requested, validated, and installed one by one. This lengthens warm-up and creates load spikes on Optimizer Hub.

With streaming, Optimizer Hub builds a stream of ready-made compilations for the application in advance. The JVM downloads it segment by segment and does not need the whole stream at once.

Components

- **Cloud Native Compiler (CNC)** compiles code on the server.
- **Code Cache (CoCa)** stores the compilations that are produced by CNC.
- **ReadyNow Orchestrator (RNO)** collects ReadyNow profile data from JVMs, develops the profile over generations, and promotes it. The profile contains profile data and

identifiers of compilations, not the code itself.

- The **Stream Builder** takes the promoted profile from RNO, finds the matching compilations in CoCa, and assembles them into a stream.

```
flowchart LR
    JVM["Azul Zing JVM"]
    subgraph OH["Optimizer Hub"]
        CNC["Cloud Native Compiler"]
        CoCa[("Code Cache")]
        RNO["ReadyNow Orchestrator"]
        SB["Stream Builder"]
    end

    CNC -->|stores compilations| CoCa
    JVM -->|sends profile data| RNO
    RNO -->|promoted profile| SB
    CoCa -->|matching compilations| SB
    SB -->|stream, segment by segment| JVM
```

Stream Build Triggers

All existing streams of a profile name are rebuilt when ReadyNow Orchestrator promotes a new profile-generations. They are not rebuilt between promotions. A new stream is also built on demand when a JVM with a new VM Description requests one and none exists.

What to Expect on the First Runs

The stream contains only ReadyNow-triggered compilations, and only those are looked up in it. Compilations that were not triggered by ReadyNow are never served from the stream and fall back to CNC as usual.

The generation-0 stream is built from compilations that were not triggered by ReadyNow, so it brings almost no benefit.

After JVMs run with the generation-0 profile, Code Cache accumulates ReadyNow-triggered compilations, and the stream rebuilt after the next promotion contains them. The earliest run that can benefit is therefore the third one. The exact timing depends on promotion thresholds and stream build time.

See profile-generations for more information.

Compatibility

Streams are served only to JVMs with a compatible VM Description (Zing version, CPU and its features, JVM flags, compile commands), because these factors influence the generated code. If no compatible stream exists, the JVM runs normally and a build is scheduled.

Streaming cannot be combined with indirect-delivery.

Memory

Streaming uses additional native (non-heap) memory on the JVM during warm-up for the active stream segments and buffers, with a short peak while segments are downloaded and parsed. By default, streaming stops and releases this memory 600 seconds after JVM start.

The overhead depends mainly on the number of active segments and the application size. It is visible in Native Memory Tracking, not in the GC log.

Also Read

- "Installing Optimizer Hub with Compilation Streaming"
- compilation-streaming
- "Using Compilation Streaming"
- Prime Documentation: Azul Zing Command Line Options

High Availability

Optimizer Hub has High Availability (HA) as a fundamental architectural principle in its design:

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

High Availability in Clusters

The system guarantees High Availability (HA) 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.
- Seamless transitions between nodes prevent service interruption.

Between Clusters

Configurations with multiple clusters also integrate HA:

- 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.
- Clusters 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	The system does not sync Cloud Native Compiler artifacts, but can easily regenerate them without compromising application performance.

Installation Prerequisites

Kubernetes Requirements

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

Deployment Modes

The resources Optimizer Hub requires depend on the mode you deploy:

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

  This is the default configuration. See "Standard Optimizer Hub Installation Procedure on Kubernetes".

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

  Cloud Native Compiler is disabled, so the pods that serve compilation are not deployed and the overall footprint is smaller. See "Installing Optimizer Hub without Cloud Native Compiler".

The Kubernetes Pods table shows which pods each mode requires.

Kubernetes Cluster

You can install Optimizer Hub on any Kubernetes cluster:

- **Kubernetes** clusters that you manually configure with kubeadm:
  - "Standard Optimizer Hub Installation Procedure on Kubernetes"
  - "Installing Optimizer Hub without Cloud Native Compiler"
- **Managed cloud Kubernetes** services such as:
  - aws-eks

  - gcp
  - azure
- A **single-node cluster**:
  - microk8s
  - minikube

Kubernetes Pods

The values below are the default, recommended sizes for each pod, as defined in the `values.yaml` file that is part of the Helm chart. Treat them as a starting point for a production deployment rather than as hard minimums.

The **Required in** column shows which deployment mode needs each pod. Pods marked **Full only** serve Cloud Native Compiler and are not deployed when you "install without Cloud Native Compiler".

Pod	CPU	RAM	Ephemeral storage	Required in
Gateway	7	28GB	1GB	All modes
gwProxy	7	1GB	1GB	All modes
Management Gateway	2	2GB	1GB	All modes
Cache	7	28GB	1GB	All modes
Compile Broker	7	28GB	8GB	Full only
Operator	1	2GB	1GB	Full only
Task Executor	4	16GB	10GB	All modes

Ephemeral storage is temporary storage space allocated to Kubernetes pods that is non-persistent and exists only for the lifetime of the pod.

NOTE	These are default, recommended values, not minimums. Optimizer Hub can run below these settings, and how far you can reduce them depends on your specific load profile, such as the number of connected JVMs, the

> compilation rate, and the size of the ReadyNow profiles you store. Reducing CPU allocations in particular can reduce the performance of the Optimizer Hub environment. If you are unsure how far you can scale down for your workload, contact Azul support before changing these values in production.

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.

Network and Gateway Requirements

To achieve peak performance and seamless compilation offloading, a robust and high-speed network foundation is essential. The connection between your Optimizer Hub instance and the environments running your Java applications is the backbone of the optimization process.

General Network Requirements

Because Optimizer Hub relies on real-time, bidirectional communication to manage JIT compilation tasks, the stability and speed of this link directly impact application latency and throughput. A reliable network ensures that JVMs can maintain the long-lived gRPC streams necessary for continuous optimization without interruption.

JVMs that run your Java applications and want to make use of the Optimizer Hub services, require unauthenticated access to the Optimizer Hub gateway.

Load Balancing

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.

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 (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 as long as the VM stays alive. The load balancer must not kill these long-lived calls.

Gateway Connection Capacity

Each Gateway instance can accept only a limited number of simultaneously connected JVMs. This limit is controlled by the infrastructure in front of or inside Optimizer Hub (for example, Envoy, ingress, or external load balancer settings).

If this connection limit is reached, additional JVMs cannot connect, even if existing connected JVMs are mostly idle and do not request compilations. In this situation, Optimizer Hub may look stuck at the minimum size because it does not see compilation demand from the JVMs that are not able to connect.

As part of installation and sizing, do the following:

- Find the connection limit configured in your infrastructure.
- Consider increasing that limit, based on your platform guidance.
- Estimate the maximum number of concurrently connected JVMs (all connected JVMs, not only JVMs currently requesting compilations).
- Make sure your minimum Gateway size can accept that number of connections; see gateway-connection-impact.

Network Latency and Upload Throughput

High network latency between your JVM and Optimizer Hub reduces upload throughput, and can prevent cloud compilation altogether. This mainly affects deployments split across environments, such as on-premises JVMs with Optimizer Hub in a cloud region.

The cause is the buffer size of the gRPC proxy between your JVMs and the Gateway, which limits how much data is in flight. This is Envoy by default, running in the `gw-proxy` pod, with a 32 KB buffer. If each buffer takes 220 ms to transfer, a 54 MB upload takes around 6 minutes 20 seconds against a default upload timeout of 30 seconds. The condition is not reported directly; it surfaces as an `Engine not found` failure.

Over a high-latency link, either increase the proxy buffer sizes or increase the upload timeout.

To increase the timeout, set `compilerEngineUploadTimeout` in your `values-override.yaml` file. The value is in seconds:

```
gateway:
  compilerEngineUploadTimeout: 600
```

NOTE Optimizer Hub does not require you to use the included `gw-proxy`. If you substitute your own gRPC-aware load balancer or ingress, can tune the equivalent settings at your own discretion.

Configuring the 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 or the (DNS) load balancer the JVMs must connect to. The JVMs need this for the value in the `-XX:OptHubHost=<host>[:<port>]` option.

Host for Single Optimizer Hub service

In a setup with a single Optimizer Hub service, you can either add your own load balancer (recommended), or use the included `gw-proxy` component.

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 define the load-balancer in `values-override.yaml` as described in "Standard Optimizer Hub Installation Procedure on Kubernetes", you can get 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 deploys 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. Use this lookup to find the port of your Optimizer Hub instance.

   You can change the gRPC-port the Gateway pod uses, in your `values-override.yaml` file in case you want to override the default values:

```
gateway:
  ports:
    serviceGrpcPort: 50051
    internalGrpcPort: 50052
cache:
  ports:
    internalGrpcPort: 50071
```

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 that maps 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`, see "Readiness (healthy) API".
- Configure your (DNS) load balancer with the host info of each Optimizer Hub service.

> **NOTE** `/q/health` runs on the management gateway and is intended for internal/Kubernetes use only. For a load balancer or other external probe checking the gateway itself, use `/opthub-health` instead; see Gateway Health Check.

Configuring the Gateway

You must expose the Optimizer Hub gRPC/HTTP endpoints outside the Kubernetes cluster, so that the client JVMs can reach them. Specify the Kubernetes Service type you use as a loadbalancer in your `values-override.yaml` file:

```
gateway:
  service:
    type: <your-service-type>
```

Available types:

- `ClusterIP`: Exposes the Service on a cluster-internal IP. Choosing this value makes the Service only reachable from within the cluster. This is the default when you don't explicitly specify a type for a Service. You can expose the Service to the public internet using an Ingress or a Gateway.
- `NodePort`: Exposes the Service on each Node's IP at a static port (the NodePort). To make the node port available, Kubernetes sets up a cluster IP address, the same as if you had requested a Service of type: `ClusterIP.`
- `LoadBalancer`: Exposes the Service externally using an external load balancer. Kubernetes does not directly offer a load balancing component. You must provide one, or you can integrate your Kubernetes cluster with a cloud provider.

You may need additional settings, depending on the type of service and your environment.

For an AWS EKS cluster, the Helm chart ships `values-awslb.yaml`, which exposes both the `gateway` and `mgmt-gateway` services through an AWS Network Load Balancer. Pass it alongside your own overrides:

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

The file sets the following for the `gateway` service, and the same type and annotations for `mgmtGateway`:

```
gateway:
  service:
    type: 'LoadBalancer'
    annotations:
      service.beta.kubernetes.io/aws-load-balancer-nlb-target-type: 'ip'
      service.beta.kubernetes.io/aws-load-balancer-target-group-attributes:
'preserve_client_ip.enabled=false'
      service.beta.kubernetes.io/aws-load-balancer-type: 'nlb'
```

Gateway Health Check

The gateway exposes a public `/opthub-health` endpoint for monitoring the health of

the gRPC channel itself. It lets a load balancer or GSLB that cannot speak gRPC (and so cannot probe the gRPC service directly) verify over plain HTTP that the gateway's gRPC server is actually up and reachable, instead of relying on a reflexive HTTP check that could report success even when gRPC is not functioning.

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

Storage Requirements

This document describes different configurations for the storage and related roles and permissions, depending on the environment on which you deploy Optimizer Hub.

Optimizer Hub stores ReadyNow profiles, compilation caches, and other data in one single blob storage location, depending on the environment, a `bucket` or `container`.

The system stores all objects in this blob by name, using a filesystem-path-like approach:

- `<bucket_or_container>/<namespace>/code-cache/`
- `<bucket_or_container>/<namespace>/compiler-artifacts/`
- `<bucket_or_container>/<namespace>/compiler-engines/`
- `<bucket_or_container>/<namespace>/inventory/`
- `<bucket_or_container>/<namespace>/persistent-profile/`

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.

You must assign a role with the below policy 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 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 Optimizer Hub Helm chart 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.

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

You need to add the role to the service account in the IAM policy 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]"
```

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.

Configuring S3 Compatible Storage

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

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

You may need additional settings, for example, when using MinIO and minikube:

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

Specifying Bucket Location

By default, the Kubernetes namespace defines the path inside the bucket defined in `commonBucket`:

```
storage:
  pathPrefix: "%namespace%"
```

You can change this value in your `values-override.yaml` file, some examples:

- Extend it with a subdirectory:

  `pathPrefix: "%namespace%/test1"`

- Use a custom subdirectory:

  `pathPrefix: "custom-path-for-opthub"`

Required Roles and Permissions

Optimizer Hub requires specific Kubernetes roles and permissions to function properly. All required permissions are scoped to the namespace level, and no cluster-level permissions are needed.

When you deploy Optimizer Hub to a Kubernetes cluster, you need to configure the permissions for the following pods:

- **Operator**: Manages deployment scaling and observability.
- **Cache**: Enables Hazelcast cluster formation and service discovery.

These components require only namespace-scoped permissions, making Optimizer Hub suitable for environments with strict security policies.

Operator Pod Permissions

The Optimizer Hub helm chart includes the following Kubernetes API permissions:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: opthub-operator-role
  namespace: {{ .Release.Namespace }}
rules:
  - apiGroups:
      - apps
    resources:
      - deployments/scale
      - deployments
      - statefulsets/scale
      - statefulsets
    verbs:
      - get
      - update
      - patch
  - apiGroups:
      - ""
      - "metrics.k8s.io"
    resources:
      - pods
    verbs:
      - get
      - list
```

NOTE

The `metrics.k8s.io` permissions are optional and only required if you want the operator to observe and optimize based on runtime metrics.

Cache Pod Permissions

The Optimizer Hub helm chart includes the following Kubernetes API permissions:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: opthub-cache-role
  namespace: {{ .Release.Namespace }}
rules:
  - apiGroups:
      - ""
    resources:
      - endpoints
      - pods
      - services
    verbs:
      - get
      - list
```

Using Externally Defined Secrets

You can externally define secrets to manage Kubernetes secrets independent of the Optimizer Hub configuration.

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

- S3
- Azure

How To Use

- If you keep the default values, the Optimizer Hub helm chart defines and uses its own Kubernetes secret objects.
- 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
```

Using an Internal Docker Registry

The Optimizer Hub components, by default, are installed from the Docker Hub `azul/opthub-gateway`. But, you can also provide these images from 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/opthub-compile-broker:26.08.0`, save it as `<your-registry>/docker-external/azul-zing/opthub-compile-broker:26.08.0`. Replace the tag with the version of Optimizer Hub you are installing.

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"
```

Installation, Upgrade, Uninstallation

Standard Optimizer Hub Installation Procedure 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.

Before you begin, check "Kubernetes Requirements" to confirm your cluster can accommodate the pods a full installation deploys. If you only need ReadyNow Orchestrator, see deployment-modes for the smaller ReadyNow-only footprint.

Installation Steps

These instructions are for installing the latest version of a full Optimizer Hub instance with both Cloud Native Compiler and ReadyNow Orchestrator. In case you don't want to install all the Optimizer Hub services, check "Installing Optimizer Hub without Cloud Native Compiler".

Step 1: Prepare Helm Repository

Make sure your Helm version is `v3.8.0` or newer and add the Azul Helm repository to your Helm environment.

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

Step 2: Create Namespace

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

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

Step 3: Create Configuration File

Create a file `values-override.yaml`. Use this file to override all the default settings and adjust Optimizer Hub to your use case and environment.

NOTE

The following represents a minimal example of the override configuration required to deploy Optimizer Hub. To fully configure Optimizer Hub to match your use case and environment, more overrides are needed. Check "Optimizer Hub Generic Defaults" for more info.

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

# Configure storage
storage:
  # Available options: s3, azure-blob, gcp-blob
  blobStorageService: s3
  # Depending on the type of storage, configure the extra settings
  s3:
    commonBucket: opthub-storage0
```

Step 4: Decide on the Monitoring Approach

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

- Use external instances managed by you. You can configure this later, after your Optimizer Hub instance is started. See external for more info.
- Configure the integrated Prometheus and Grafana instances now, as the next step needs these configurations. See integrated for more info.

Step 5: Execute Installation

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

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

NOTE	If you don't need all Optimizer Hub services, see "Installing Optimizer Hub without Cloud Native Compiler" for the available modes, and kubernetes-pods for how each mode changes the resources you need to provision.

NOTE	To enable Compilation Streaming, check "Installing Optimizer Hub with Compilation Streaming".

```
helm install opthub opthub-helm/azul-opthub \
  -n my-opthub \
  -f values-override.yaml \
  --version 26.05.5
```

Or, when you configured the integrated Prometheus and Grafana in the previous step:

```
helm install opthub opthub-helm/azul-opthub \
  -n my-opthub \
  -f values-override.yaml \
  -f ~/my-opthub/values-monitoring.yaml \
  --version 26.05.5
```

This 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
```

Step 6: Verify Installation

Verify that all pods start and reach ready state:

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

Installing Optimizer Hub without Cloud Native Compiler

Optimizer Hub can run in different modes:

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

  This is the default configuration. See "Standard Optimizer Hub Installation Procedure on Kubernetes".

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

  Use the installation instructions below.

Before you install, check kubernetes-pods to confirm your cluster can accommodate the pods this mode deploys.

Resource Requirements

In ReadyNow-only mode, the pods that serve Cloud Native Compiler are not deployed, so the footprint is smaller than a full installation. The pods you do need are sized the same as in a full installation:

Pod	CPU	RAM	Ephemeral storage
Gateway	7	28GB	1GB

Pod	CPU	RAM	Ephemeral storage
gwProxy	7	1GB	1GB
Management Gateway	2	2GB	1GB
Cache	7	28GB	1GB

For the complete pod list, the full-mode comparison, and guidance on scaling these values for your load profile, see "Kubernetes Requirements".

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`.

NOTE

Disabling Cloud Native Compiler with `compilations.parallelism.limitPerVm` alone stops compilation requests but does not remove the Compile Broker pods, so the resource footprint stays the same as a full installation. Reinstall with `values-disable-compiler.yaml` to reclaim those resources.

Installing Optimizer Hub with Compilation Streaming

Compilation Streaming lets a JVM download a stream of previously performed compilations from Optimizer Hub at startup and install them directly, instead of requesting each compilation individually as the application warms up. This shortens the time needed to clear the compilation queue and reduces load on Cloud Native Compiler.

For background on how the feature works, see "About Compilation Streaming".

Compilation Streaming must be enabled on both the Optimizer Hub server and the JVM. This page gives the complete commands for both. For how the feature works, its full configuration options, and its metrics, see "Using Compilation Streaming".

Compilation Streaming requires:

- Optimizer Hub 26.08.0 or newer on the server.
- Azul Zing 26.08.1.0 or newer on the client.
- A promoted ReadyNow profile for the profile name your JVMs use. See Compilation Streaming Requires a Promoted Profile.

Step 1: Enable Compilation Streaming on the Server

Compilation Streaming requires one setting. Add it to your `values-override.yaml`:

```
compilationStreaming:
  enabled: true
```

The `task-executor` pod that builds the streams is deployed by default, so no second setting is needed.

IMPORTANT

If you disabled `task-executor` in your `values-override.yaml`, re-enable it with `taskExecutor.enabled: true`. Without it, the streaming endpoints start correctly but no streams are ever built, and JVMs continue without streaming and without reporting an error.

New Installation

```
helm install opthub opthub-helm/azul-opthub \
  -n my-opthub \
  -f values-override.yaml \
  --version 26.05.5
```

Existing Installation

To enable Compilation Streaming on a running Optimizer Hub instance, use the same files and settings as your original installation and replace `install` with `upgrade`:

```
helm upgrade opthub opthub-helm/azul-opthub \
  -n my-opthub \
  -f values-override.yaml \
  --version 26.05.5
```

Step 2: Verify the Server

Check that the `task-executor` pod is running:

```
kubectl get pods -n my-opthub | grep task-executor
```

Step 3: Enable Compilation Streaming on the JVM

Add `-XX:+CNCEnableCompilationStreaming` to the JVM options of your application, alongside the options that connect it to Optimizer Hub and set its ReadyNow profile name:

```
java -XX:OptHubHost=<host>:<port> \
    -XX:+EnableRNO \
    -XX:ProfileName=<profile-name> \
    -XX:+CNCEnableCompilationStreaming \
    -jar my-app.jar
```

Optimizer Hub builds each stream from its Code Cache and the promoted ReadyNow profile for a given profile name, so the JVM must use the same `-XX:ProfileName` as the profile the stream was built from. For the other options that connect a JVM to Optimizer Hub, see "Connecting a JVM to Optimizer Hub" and "Using ReadyNow Orchestrator".

At startup the JVM asks whether a stream exists for its profile. If one does, the JVM

uses it. If not, the JVM runs normally without streaming and no error is reported.

Step 4: Verify the Client

Streaming statistics appear in the JVM's GC log in the `COMPSTREAMSTATS` line, covering lookups, hits, misses, installation outcomes, segments loaded and evicted, and bytes downloaded. The hit rate derived from these values is the main indicator of whether streaming is working well for an application.

Compilation Streaming Requires a Promoted Profile

Optimizer Hub can only provide a stream after a ReadyNow profile has been promoted for the given profile name. If the promotion criteria configured for your deployment are never met, no stream is ever built, and JVMs continue without streaming.

This is not reported as an error. A JVM that finds no stream for its profile starts normally and the application behaves correctly, so an instance that appears healthy may not be using Compilation Streaming at all. If `COMPSTREAMSTATS` shows no hits, confirm that a promoted profile exists for your profile name before investigating anything else.

Promotion criteria are set per generation and are configurable, so how long promotion takes depends on your deployment. See profile-generations.

NOTE	A profile that you upload manually is promoted immediately, regardless of the configured promotion rules. See upload-profile.
IMPORTANT	A single JVM cannot use both Compilation Streaming and indirect-delivery. See streaming-and-indirect-delivery.

Platform-Specific Installation Steps

AWS EKS

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).

- Check the instructions on configuring-aws-s3-storage.

- Create a Kubernetes cluster within your AWS EKS environment using the AWS EKS Cluster Requirements as a guideline.
- To expose the `gateway` and `mgmt-gateway` services through an AWS Network Load Balancer, pass the shipped `values-awslb.yaml` during installation. See configuring-gateway.
- Follow the "Standard Optimizer Hub Installation Procedure on Kubernetes".

AWS EKS Cluster 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.

Microsoft Azure

- Check the instructions on configuring-azure-blob-storage.
- Create a Kubernetes cluster within your Microsoft Azure environment.
- Follow the "Standard Optimizer Hub Installation Procedure on Kubernetes".

Google Cloud Platform

- Check the instructions on configuring-gcp-blob-storage.
- Create a Kubernetes cluster within your Google Cloud Platform environment.
- Follow the "Standard Optimizer Hub Installation Procedure on Kubernetes".

Other Kubernetes Platform

- Check the instructions on configuring-s3-compatible-storage.
- Create a Kubernetes cluster within your environment.
- Follow the "Standard Optimizer Hub Installation Procedure on Kubernetes".

Local Installation

You can use MicroK8s or minikube for testing, evaluating, and non-cloud-managed blob storage use of Optimizer Hub.

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

Optimizer Hub requires blob storage. You can add this to your local setup with MinIO.

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

Installing on MicroK8s

Installing MicroK8s

Install MicroK8s for your platform following this installation guide.

1. Install MicroK8s on Linux with Snap:

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

2. Or, when already installed, start it:

```
microk8s start
```

3. 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
```

4. Turn on these additional services:

```
# Built-in Kubernetes dashboard
microk8s enable dashboard
```

```
# Metrics used by Optimizer Hub to provide a Grafana dashboard
microk8s enable metrics-server
```

5. 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...
```

6. You can now open the dashboard in the browser, using the token from the previous step.

Installing Optimizer Hub on MicroK8s

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
```

Modify the following value to match your environment:

```
# Existing
storageEndpoint: http://minio.minio-dev.svc.cluster.local:9000
# Change to
storageEndpoint: http://minio.my-opthub:9000
```

8. Create the configuration files for the monitoring tools (Prometheus and Grafana).
   - A file is needed for the role creation: `metrics-rbac.yaml`
     - Copy the file from the GitHub opthub-helm-charts repository.
     - Replace the namespace in two places `namespace: opthub` in the example file, or execute:

```
sed -i 's/opthub$/my-opthub/g' ~/my-opthub/metrics-rbac.yaml
```

- If you create your service accounts separately, you need to match these names with the ones in this file.
- Apply these roles to your MicroK8s cluster:

```
microk8s kubectl apply -f ~/my-opthub/metrics-rbac.yaml
```

- Create a file with the monitoring configuration: `values-monitoring.yaml`
  - Copy the file from the GitHub opthub-helm-charts repository.
  - If you created your own service account, update this value in the file: `name: metrics-server`.
- Install the Kubernetes State Metrics service:

```
microk8s helm install kube-state-metrics \
    oci://ghcr.io/prometheus-community/charts/kube-state-metrics
```

9. 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 "Installing Optimizer Hub without Cloud Native Compiler". To enable Compilation Streaming, check "Installing Optimizer Hub with Compilation Streaming".

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

10. Verify that all started pods are ready:

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

```
NAME                                    READY   STATUS              RESTARTS   AGE
pod/cache-0                             0/1     Running             0          26s
pod/compile-broker-68d58bbc8d-nqj2z     0/1     Running             0          26s
pod/gateway-68d84f7fff-zwpx4            0/1     Running             0          26s
pod/grafana-56c957b967-4r9w2            0/2     ContainerCreating   0          26s
pod/gw-proxy-5bf785bdbc-fl586           1/1     Running             0          26s
pod/mgmt-gateway-5c556bd89f-rxd24       0/1     Running             0          26s
pod/minio                               1/1     Running             0
4m23s
pod/minio-setup-job-t29vv               0/1     Completed           1
4m23s

NAME                          TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(
S)             AGE
service/cache                 ClusterIP      None             <none>
5701/TCP            26s
service/compile-broker        ClusterIP      10.152.183.211   <none>
50051/TCP           26s
service/gateway               LoadBalancer   10.152.183.161   <pending>
50051:30759/TCP     26s
service/gateway-headless      ClusterIP      None             <none>
50051/TCP           26s
service/grafana               NodePort       10.152.183.49    <none>
80:32146/TCP        26s
service/grafana-headless      ClusterIP      None             <none>
9094/TCP            26s
service/mgmt-gateway          LoadBalancer   10.152.183.185   <pending>
8080:32188/TCP      26s
service/minio                 ClusterIP      10.152.183.106   <none>
9000/TCP,9090/TCP   4m23s
service/prometheus-server     ClusterIP      10.152.183.235   <none>        80/TCP
26s

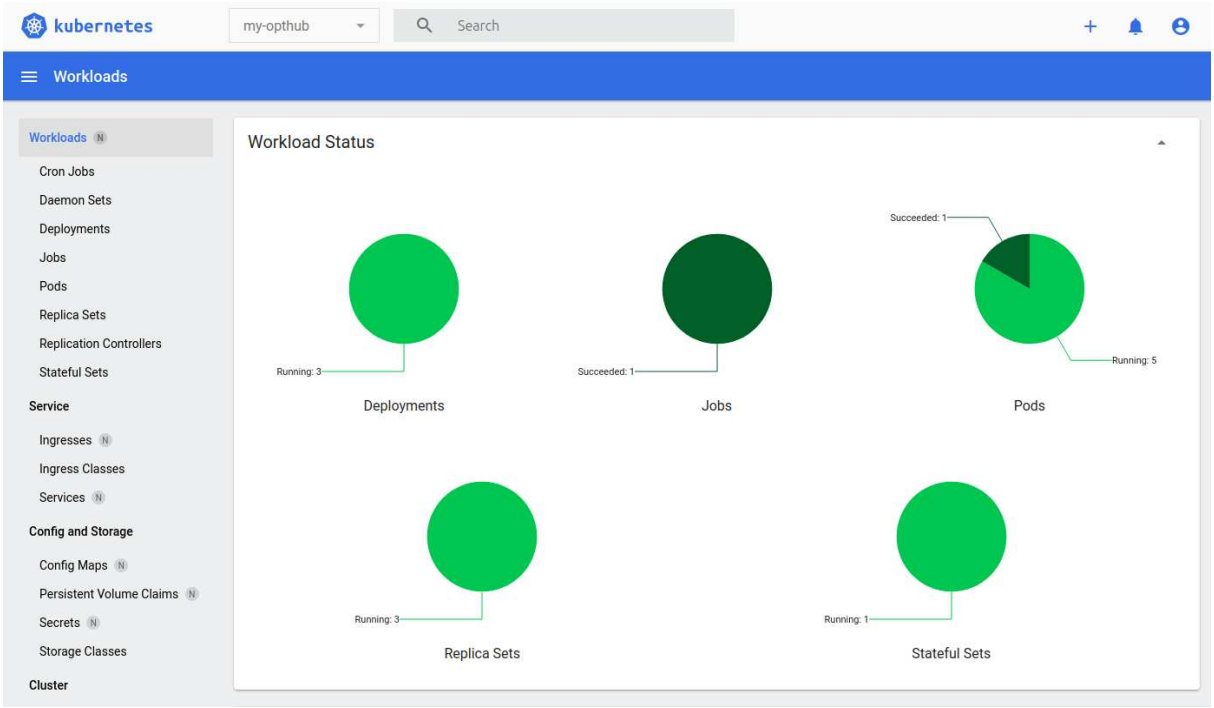
NAME                                  READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/compile-broker        0/1     1            0           26s
deployment.apps/gateway               0/1     1            0           26s
deployment.apps/grafana               0/1     1            0           26s
deployment.apps/gw-proxy              1/1     1            1           26s
deployment.apps/mgmt-gateway          0/1     1            0           26s
deployment.apps/prometheus-server     0/1     0            0           26s

NAME                                             DESIRED   CURRENT   READY   AGE
replicaset.apps/compile-broker-68d58bbc8d        1         1         0       26s
replicaset.apps/gateway-68d84f7fff               1         1         0       26s
replicaset.apps/grafana-56c957b967               1         1         0       26s
replicaset.apps/gw-proxy-5bf785bdbc              1         1         1       26s
replicaset.apps/mgmt-gateway-5c556bd89f          1         1         0       26s
replicaset.apps/prometheus-server-59588969fc     1         0         0       26s

NAME                       READY   AGE
statefulset.apps/cache     0/1     26s

NAME                           STATUS     COMPLETIONS   DURATION   AGE
job.batch/minio-setup-job      Complete   1/1           11s        4m23s
```

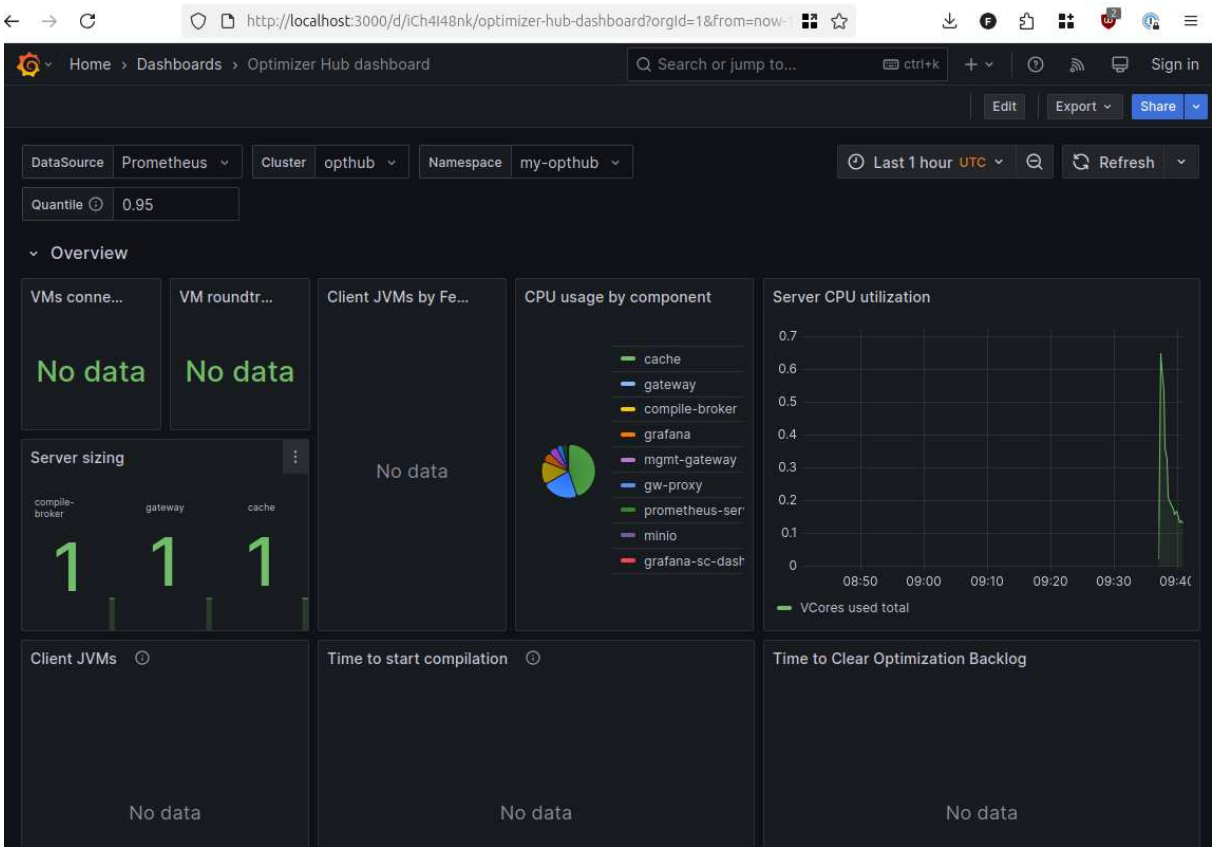
11. You can also verify the status on the dashboard we opened before, by selecting the `my-opthub` namespace in the header:



12. Add port forwarding to have access to Grafana:

```
microk8s kubectl port-forward -n my-opthub deployments/gw-proxy 3000:3000
```

13. The Grafana dashboard is now available on `localhost:3000`. At this moment, it only shows minimal data. Use this dashboard later, when you have connected your first clients.



Upgrading Optimizer Hub on MicroK8s

You can upgrade your Optimizer Hub instance running on MicroK8s, keep all profile data and compile caches as the upgrade does not affect the S3 compatible minio storage.

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

You can remove Optimizer Hub from MicroK8s using `helm`, after which you can also delete the namespace.

```
microk8s helm uninstall opthub -n my-opthub
microk8s kubectl delete namespace my-opthub
microk8s kubectl delete -f testing-rbac.yaml
microk8s helm repo remove opthub-helm
```

Installing on minikube

Installing minikube

1. Install minikube for your platform following this installation guide.
2. Start minikube with some extra options to add the metrics server.

```
minikube start --addons=metrics-server --container-runtime=containerd
```

> **NOTE**
>
> The `--container-runtime=containerd` option is required for the Grafana dashboard to work correctly.

Installing Optimizer Hub on minikube

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
```

Modify the following value to match your environment:

```
# Existing
storageEndpoint: http://minio.minio-dev.svc.cluster.local:9000
# Change to
storageEndpoint: http://minio.my-opthub:9000
```

8. Create the configuration files for the monitoring tools (Prometheus and Grafana).
   - A file is needed for the role creation: `metrics-rbac.yaml`
     - Copy the file from the GitHub opthub-helm-charts repository.
     - Replace the namespace in two places `namespace: opthub` in the example file, or execute:

```
sed -i 's/opthub$/my-opthub/g' ~/my-opthub/metrics-rbac.yaml
```

     - If you create your service accounts separately, you must match these names with the ones in this file.
     - Apply these roles to your minikube:

```
minikube kubectl -- apply -f ~/my-opthub/metrics-rbac.yaml
```

   - Create a file with the monitoring configuration: `values-monitoring.yaml`

- Copy the file from the GitHub opthub-helm-charts repository.
- If you created your own service account, update this value in the file: `name: metrics-server`.

- Install the Kubernetes State Metrics service:

```
helm install kube-state-metrics \
    oci://ghcr.io/prometheus-community/charts/kube-state-metrics
```

9. Install using Helm, passing in the `values-minikube.yaml` and `values-monitoring.yaml`. In case you don't want to install the full Optimizer Hub, but only a part of the services, first check "Installing Optimizer Hub without Cloud Native Compiler". To enable Compilation Streaming, check "Installing Optimizer Hub with Compilation Streaming".

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

The command produces 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
```

10. 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
111s
pod/compile-broker-9d8755469-hnc86      1/1     Running     0
111s
pod/gateway-64dc66c9cd-8r87r            1/1     Running     0
111s
pod/grafana-6dccb6d5bd-5wct5            2/2     Running     0
111s
pod/gw-proxy-d9bc9f69d-mzqlb            1/1     Running     0
111s
```

```
pod/mgmt-gateway-5fb49f767d-4rqnz        1/1     Running     0
111s
pod/minio                                1/1     Running     1 (2m17s ago)
3d20h
pod/minio-setup-job-867vj                0/1     Completed   0
3d20h
pod/prometheus-server-65d89b44d4-nshgg   1/1     Running     0
111s

NAME                          TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(
S)             AGE
service/cache                 ClusterIP      None             <none>
5701/TCP            111s
service/compile-broker        ClusterIP      10.101.186.209   <none>
50051/TCP           111s
service/gateway               LoadBalancer   10.108.230.242   <pending>
50051:31551/TCP     111s
service/gateway-headless      ClusterIP      None             <none>
50051/TCP           111s
service/grafana               NodePort       10.109.252.245   <none>
80:30529/TCP        111s
service/grafana-headless      ClusterIP      None             <none>
9094/TCP            111s
service/mgmt-gateway          LoadBalancer   10.108.223.142   <pending>
8080:30912/TCP      111s
service/minio                 ClusterIP      10.107.144.95    <none>
9000/TCP,9090/TCP   3d20h
service/prometheus-server     ClusterIP      10.101.23.244    <none>        80/TCP
111s

NAME                                    READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/compile-broker          1/1     1            1           111s
deployment.apps/gateway                 1/1     1            1           111s
deployment.apps/grafana                 1/1     1            1           111s
deployment.apps/gw-proxy                1/1     1            1           111s
deployment.apps/mgmt-gateway            1/1     1            1           111s
deployment.apps/prometheus-server       1/1     1            1           111s

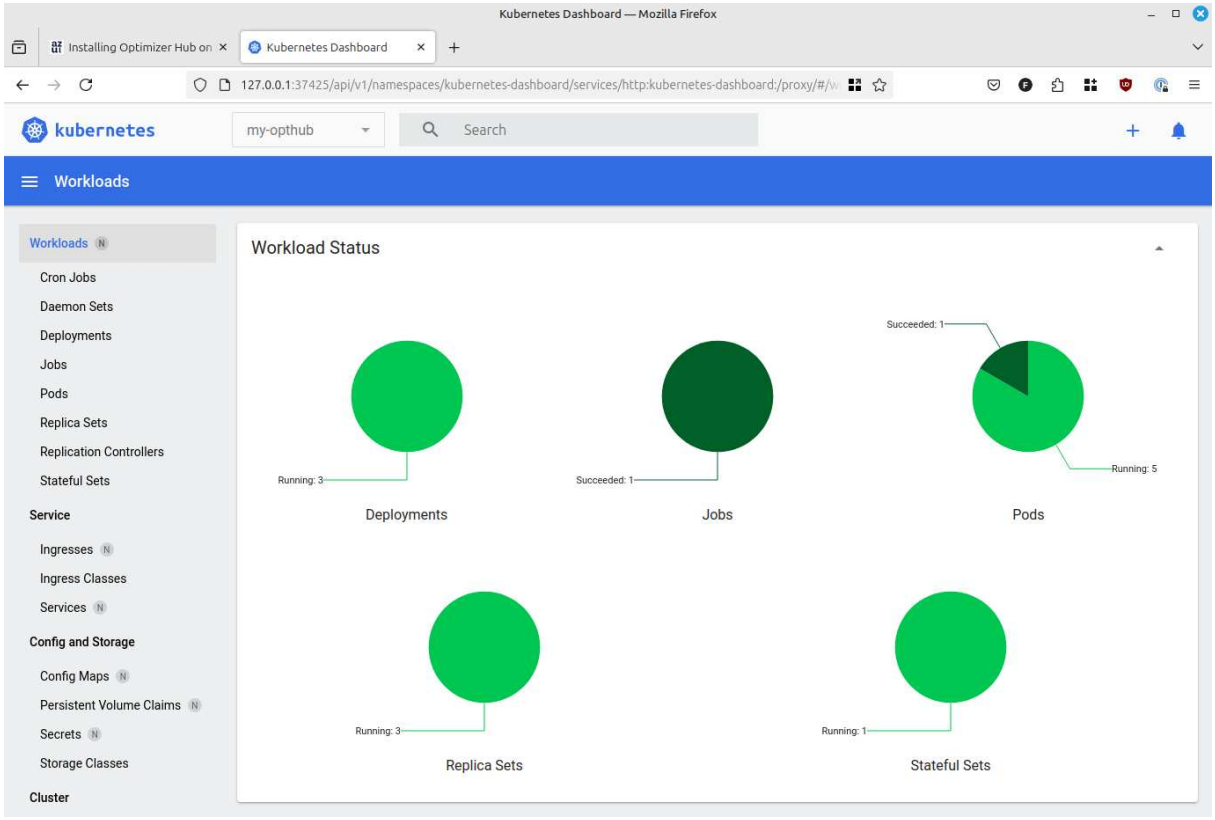
NAME                                               DESIRED   CURRENT   READY   AGE
replicaset.apps/compile-broker-9d8755469           1         1         1       111s
replicaset.apps/gateway-64dc66c9cd                 1         1         1       111s
replicaset.apps/grafana-6dccb6d5bd                 1         1         1       111s
replicaset.apps/gw-proxy-d9bc9f69d                 1         1         1       111s
replicaset.apps/mgmt-gateway-5fb49f767d            1         1         1       111s
replicaset.apps/prometheus-server-65d89b44d4       1         1         1       111s

NAME                       READY   AGE
statefulset.apps/cache     1/1     111s

NAME                           STATUS     COMPLETIONS   DURATION   AGE
job.batch/minio-setup-job      Complete   1/1           12s        3d20h
```

11. 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.

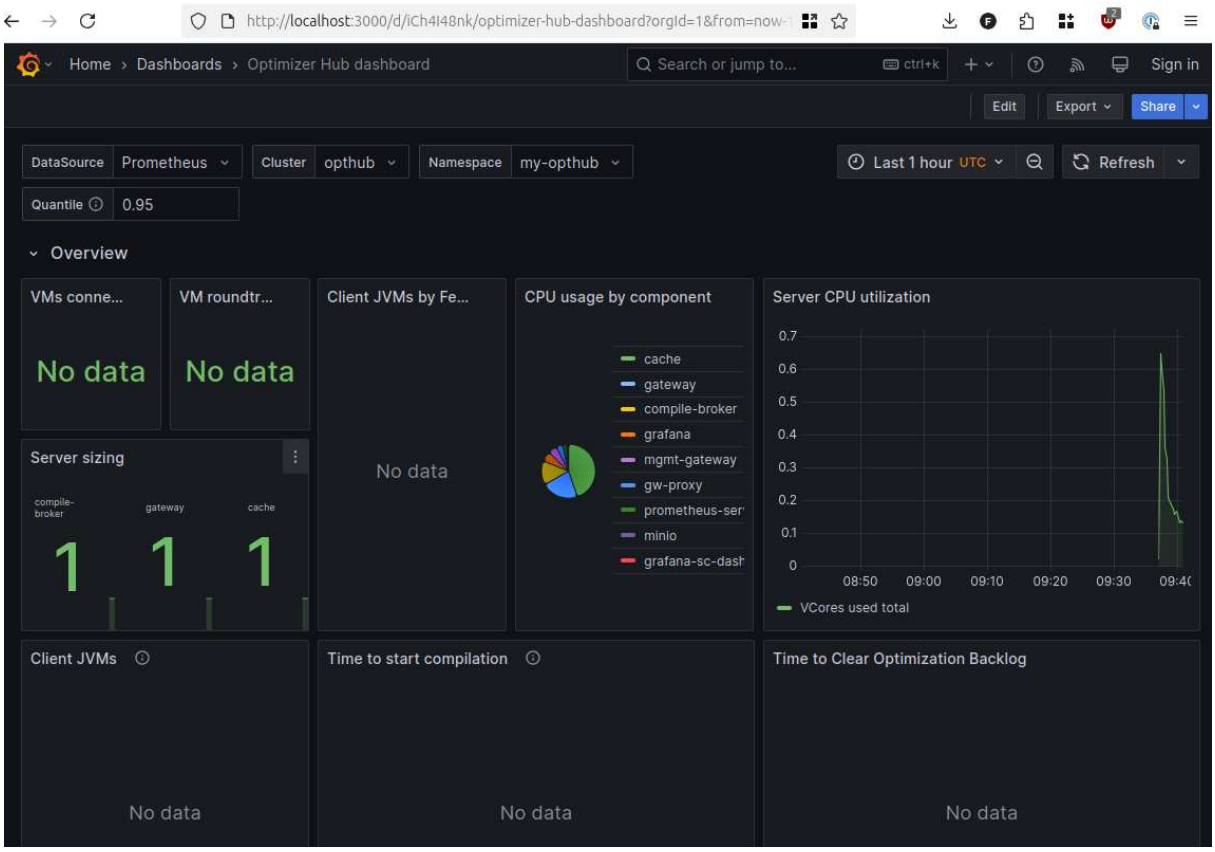
```
minikube dashboard
```



12. Add port forwarding to have access to Grafana:

```
kubectl port-forward -n my-opthub deployments/gw-proxy 3000:3000
```

13. The Grafana dashboard now becomes available on `localhost:3000`. At this moment, it only shows minimal data. Use this dashboard later, when you have connected your first clients.



Uninstalling Optimizer Hub from minikube

You can remove Optimizer Hub from minikube using `helm`, after which you can also delete the namespace.

```
helm uninstall opthub -n my-opthub
minikube kubectl -- delete namespace my-opthub
minikube kubectl -- delete -f testing-rbac.yaml
helm repo remove opthub-helm
```

Upgrading or Uninstalling Optimizer Hub

Follow the below instructions to upgrade, downgrade, or uninstall Optimizer Hub.

Upgrading or Rolling Back

Follow these steps to upgrade or rollback your Optimizer Hub instance to a newer or previous version:

1. Shut down the existing instance.
2. Validate your `values-override.yaml` file for any parameter changes.

3. Install the new version "following the installation guide".

NOTE	Rolling and in-place upgrades are not supported. Shut the instance down completely, including the `cache` pods, before you install the new version. If a `cache` pod keeps running, the new version can inherit data it is unable to read, and the Optimizer Hub API returns HTTP 500 on the affected endpoints. See api-500-after-upgrade for how to recover.
NOTE	The blob storage metadata format may vary between releases. Performing an upgrade or rollback with a blob storage containing files modified by another version, may result in incorrect behavior.

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

Uninstalling

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

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

Configuration Management

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 by overriding the default values in your `values-override.yaml` file.

Management Gateway Parameters

Option	Description	Default
`mgmtGateway.enabled`	Flag to define if the Management Gateway is 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

Additional Labels for Kubernetes Objects

If needed, you can specify additional labels for the Kubernetes objects.

Option	Description	Default
`gateway.applicationLabels`	For Deployment/StatefulSet	
`gateway.applicationAnnotations`	For Deployment/StatefulSet	
`gateway.podTemplateLabels`	For POD	
`gateway.podTemplateAnnotations`	For POD	
`gateway.serviceLabels`	For Service	

ReadyNow Orchestrator Defaults

See profile-generations for more information.

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

Option	Description	Default
`readyNowOrchestrator.producers.continueRecordingOnPromotion`	Flag to define if profiles must be recorded after the maxGeneration has been reached. You can use this flag for debugging purposes.	false
`readyNowOrchestrator.producers.maxConcurrentRecordings`	The number of concurrent copies of a specific generation ReadyNow Orchestrator accepts before it tells other JVMs trying to write the same generation of the same profile name to stop	10
`readyNowOrchestrator.producers.maxPromotableGeneration`	Maximum number of generations ReadyNow Orchestrator accepts for a profile name. Note that there is no 'unlimited' value available	3
`readyNowOrchestrator.producers.maxProfileSize`	Limit on the input profile size, in bytes.	300000000
`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 system removes the profile name if no process requested it during the 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 profile-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 profile-generations.	0:PT2M\,1:PT15M\,2:PT30M\,3:PT60M
`readyNowOrchestrator.producers.maxSynchronizedGeneration`	Defines the maximum number of profile generations to be synced from peers. The system does not sync profiles with a higher generation from peers.	2

Cross-Region Sync Parameters

See cross-region-sync for more information.

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

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

Option	Description	Default
`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 periodicity of a synchronization with the specified Optimizer Hub peers.	PT30s

Compile Artifacts Auto Cleanup Parameters

Artifacts from compiler crashes, used for debugging purposes, can be automatically cleaned up. If the profile cleaner deletes the profile, the compiler artifacts for the profile also get deleted. These artifacts are also cleaned based on age.

Option	Description	Default
`compileArtifact.cleaner.enabled`	Define if the automatic cleanup of compile artifacts should be enabled.	true
`compileArtifact.cleaner.keepCompilerCrashArtifactsFor`	Defines how after what duration the artifacts are deleted.	P30D
`compileArtifact.cleaner.cleanupInterval`	Defines periodicity of the cleaning job.	P1D
`compileArtifact.cleaner.firstRunDelay`	Defines the delay before the first run of the cleaner.	PT6H

Blob Storage Auto Cleanup Parameters

See "Configuring Blob Storage Auto Cleanup" for more info.

Option	Description	Default
`codeCache.cleaner.enabled`	Flag to specify if the automatic Code Cache cleaner must be enabled.	true

Option	Description	Default
`codeCache.cleaner.targetSize`	Target size for the Code Cache cleaner.	"100GiB" # Or "107374182400"
`codeCache.cleaner.interval`	Interval at which 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.	PT2H # 2 hours

Externally Defined Secrets Parameters

See "Using Externally Defined Secrets" for more info.

Option	Description	Default
`secrets.blobStorage.s3.existingSecret`	Name of the existing Secret object to use. The system creates a new secret if the name is empty.	""
`secrets.blobStorage.s3.accessKeySecretKey`	Name of the key for the `accessKey` value in Kubernetes secret. It can be renamed to the match names in an existing secret.	blob-storage-accesskey
`secrets.blobStorage.s3.accesskey`	The default value for accesskey that the system uses when it creates a new secret.	<yourAccessKey>
`secrets.blobStorage.s3.secretAccessKeySecretKey`	The name of the key for the `secretkey` value in Kubernetes secret. It can be renamed to match names in an existing secret.	blob-storage-secretkey
`secrets.blobStorage.s3.secretkey`	The default value for `s3.secretkey` that the system uses when it creates a new secret.	<yourSecretKey>
`secrets.blobStorage.azure.existingSecret`	The name of an existing secret object to use. The system creates a new secret if the name is empty.	""

Option	Description	Default
`secrets.blobStorage.azure.connectionStringSecretKey`	The name of the key for the `connectionStringSecretKey` value in Kubernetes secret. It can be renamed to match names in an existing secret.	azure-storage-connection-string
`secrets.blobStorage.azure.sasTokenSecretKey`	The name of the key for `sasTokenSecretKey` in K8S Secret. It can be renamed to match names in existing Secret	azure-storage-sas-token

Simple Sizing Parameters

See how-scales for more info.

Option	Description	Default
`simpleSizing.vCores`	The total number of vCores that the Optimizer Hub service uses. The default value is the number of vCores necessary to start one instance of Optimizer Hub, so that the service is operational.	32
`simpleSizing.minVCores`	`minVCores` and `maxVCores` have the same formula as `vCores`, but apply when you enable autoscaling. The Optimizer Hub service uses the value to adjust the sizing of the instance to try to meet your `timeToClearOptimizationBacklog` limit for all the JVMs that request compilations.	32
`simpleSizing.maxVCores`	The default max value allocates 10 compile brokers.	106

Compilations Parameters

See how-scales for more info.

Option	Description	Default
`compilations.parallelism.limitPerVm`	Maximum concurrent compilations per VM (VMs will not send more than this amount at the same time).	500
`compilations.parallelism.limitPerCompileBroker`	Maximum number of parallel compiler engine processes running at the same time per compile broker.	30
`compilations.parallelism.lookupParallelism`	Is calculated as `limitPerCompileBroker * 4`	120

Compilation Streaming Parameters

See "About Compilation Streaming" and "Using Compilation Streaming" for more info.

Option	Description	Default
`compilationStreaming.enabled`	Enables Compilation Streaming. Optimizer Hub builds streams from Code Cache and ReadyNow Orchestrator content, and the gateway's streaming endpoints become available. Requires `taskExecutor.enabled`.	false
`compilationStreaming.eviction.enabled`	Evicts stale compilation stream build requests.	true
`compilationStreaming.eviction.buildRequestRetention`	How long a stream build request is retained before eviction, in ISO-8601 duration format. Applies only when `compilationStreaming.eviction.enabled` is true.	P7D
`taskExecutor.enabled`	Deploys the `task-executor` pod, which builds compilation streams and refreshes the ReadyNow profiles used by indirect-delivery. Without it, stream build requests are recorded but never executed.	true
`taskExecutor.terminationGracePeriodSeconds`	How long Kubernetes waits for the `task-executor` pod to shut down cleanly before killing it.	120

SSL Parameters

See "Configuring Optimizer Hub with SSL Authentication" for more info.

Option	Description	Default
`ssl.enabled`	Flag to specify if SSL must be enabled for Optimizer Hub connections from JVMs.	false

`ssl.value.cert` and `ssl.value.key`	Specify the certificate and private keys directly as values. This is the simplest way to run quick experiments in a controlled environment, especially when you're installing from the Helm repository.
	
	Example:
	
	`ssl:`
	`  value:`
	`  cert: |-`
	`  -----BEGIN CERTIFICATE-----`
	`  ...`
	`  -----END CERTIFICATE-----`
	`  key: |-`
	`  -----BEGIN PRIVATE KEY-----`
	`  ...`
	`  -----END PRIVATE KEY-----`
	
	NOTE This is not the recommended approach in production as it embeds private security credentials in a config file.	

`ssl.path.cert` and `ssl.path.key`	Specify the certificate and private key file paths.	
`ssl.existingSecret`	When you use a separate chain to manage your certificate, you can point the deployment to a custom secret in the installation namespace. Such a secret needs to have keys named `cert.pem` and `key.pem`. This is the most secure way to add certificates.	

Size and Duration Formats

Specifying Size Values

You can define size values in helm charts, e.g. `values-override.yaml`, with different notations:

- Number format

  Example: `5000000`

- Decimal format: `M`, `G`,...

  Example: `5M` instead of `5000000`

- Binary format: `Mi`, `Gi`,...

  Example: `20Mi` instead of `20971520`

Specifying Durations

You can specify the duration in time using the ISO-8601 format `PnDTnHnMn.nS` where n is the relevant days, hours, minutes or seconds part of the duration.

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 the candidates for a profile name and decides which is the 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.

Understanding ReadyNow Orchestrator Generations

With ReadyNow, you get the best results if you perform several runs of your application to generate an optimal profile. The ReadyNow Orchestrator feature in Optimizer Hub automatically takes care of creating a promoted profile as you run your application. With the default configuration, Optimizer Hub uses four runs to reach the optimal profile.

```
sequenceDiagram
    participant JVM as JVM
    participant RNO as ReadyNow Orchestrator

    Note over RNO,JVM: First run of new app
    JVM->>RNO: MyApp-v1.5 (Gen0)

    Note over RNO,JVM: After 15 minutes
    JVM->>RNO: MyApp-v1.5 (Gen1)

    Note over RNO,JVM: After 30 minutes
    JVM->>RNO: MyApp-v1.5 (Gen2)

    Note over RNO,JVM: After 60 minutes
    JVM->>RNO: MyApp-v1.5 (Gen3)

    Note over RNO: MyApp-v1.5 Gen3 gets promoted

    Note over RNO,JVM: All following runs
    RNO->>JVM: MyApp-v1.5 (Gen3)
```

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

When you deploy 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 a profile is available when new JVMs start. 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 and let ReadyNow Orchestrator nominate profile logs after three full generations according to these default values:

```
promotion:
    minProfileSize: 1M
    minProfileDuration: "PT2M"
    minProfileSizePerGeneration: "0:1M,1:10M,2:25M,3:50M"
    minProfileDurationPerGeneration: "0:PT2M,1:PT15M,2:PT30M,3:PT60M"
```

For example, if 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 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 falls below 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.

If needed, you can overrule the server-side defaults, by providing extra options to the

JVM. For example:

```
java -XX:OptHubHost={host:port} \
    -XX:+EnableRNO \
    -XX:ProfileName=MyApp-v3 \
    -XX:ProfileLogOutNominationMinSizePerGeneration=0:1M\,1:25M\,2:50M\,3:75M \
    -XX:ProfileLogOutNominationMinTimeSecPerGeneration=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

Take the default values into account when you define your own generation settings as other default settings can overrule these. 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 system does not promote the 2nd and 3rd generation after 900 seconds, but after 30 and 60 minutes as the defaults specify.

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

options.

Uploading a Profile through the API

In certain scenarios, you may need to upload a profile to your Optimizer Hub instance. Some example use cases include:

- Migrating profiles between different Optimizer Hub instances.
- Re-uploading profiles after offline post-processing or analysis.
- Restoring profiles from backups.
- Importing profiles collected through alternative methods.

The REST API provides an upload endpoint to support these use cases.

```
POST /api/profileNames/{profileName}/profileLogs:upload
```

Example to call this API with `curl`, replace `{profileName}` with your actual profile name:

```
curl -X POST \
  http://{host:port}/api/profileNames/{profileName}/profileLogs:upload \
  -H "Content-Type: text/plain" \
  --data-binary @profile.log
```

If your profile log is already compressed, you can indicate this with the `Accept-Encoding` header:

```
curl -X POST \
  http://{host:port}/api/profileNames/{profileName}/profileLogs:upload \
  -H "Content-Type: text/plain" \
  -H "Accept-Encoding: gzip" \
  --data-binary @profile.log.gz
```

NOTE Uploaded profiles are immediately promoted, regardless of the current configuration of the promotion rules.

You can find more details about the use of the API in the documentation of the

"Optimizer Hub API".

Indirect Delivery of ReadyNow Profiles

By default, a JVM receives its input ReadyNow profile from Optimizer Hub over gRPC. With Indirect Delivery, Optimizer Hub instead stores the profile in blob storage and hands the JVM a pre-signed URI, which the JVM uses to download the profile over HTTP.

Indirect Delivery is useful when you want to reduce the volume of traffic through the Optimizer Hub gateway, or when your network topology makes a direct gRPC transfer of large profiles impractical.

Configure Indirect Delivery under the `indirectDelivery` key:

```
indirectDelivery:
  enabled: true
```

The first copy of a profile is prepared for Indirect Delivery when the profile is promoted. After that, the `task-executor` component periodically re-prepares any profile whose underlying ReadyNow profile log has grown significantly, so that JVMs keep receiving an up-to-date profile. As of Optimizer Hub 26.08.0, `task-executor` is deployed by default. If you disabled it with `taskExecutor.enabled: false`, the initial copy is still prepared, but it is never refreshed and JVMs continue to download an increasingly stale profile.

On the JVM side, the transport is controlled by `-XX:RNOProfileTransportPriority`, which defaults to `uri`, meaning the JVM attempts an HTTP download first and falls back to gRPC.

Even when Optimizer Hub offers a pre-signed URI, a JVM does not always use it. The JVM receives the profile over gRPC instead when:

- You explicitly set `-XX:RNOProfileTransportPriority=grpc`.
- The HTTP download fails for any reason. The JVM then falls back to gRPC

automatically.

- You enable Compilation Streaming on the JVM. A compilation stream is identified by the `ProfileLogId` of the profile it was built from, and a profile downloaded from a pre-signed URI does not carry that identifier, so the JVM prints a warning and switches the transport to gRPC. See streaming-and-indirect-delivery.

Take this into account when you size your Optimizer Hub gateway, as Indirect Delivery only reduces gateway traffic for the profiles that are actually delivered over HTTP.

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 now 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.

Check cross-region-sync-parameters for the available configuration options.

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. You must configure cleanup for ReadyNow profile logs and Code Cache entries separately.

Code Cache Cleanup

You specify the target size for the blob storage that Optimizer Hub may not exceed, as well as how often Optimizer Hub must 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: "100GiB" # Or "107374182400"
    interval: PT2H # 2 hours
```

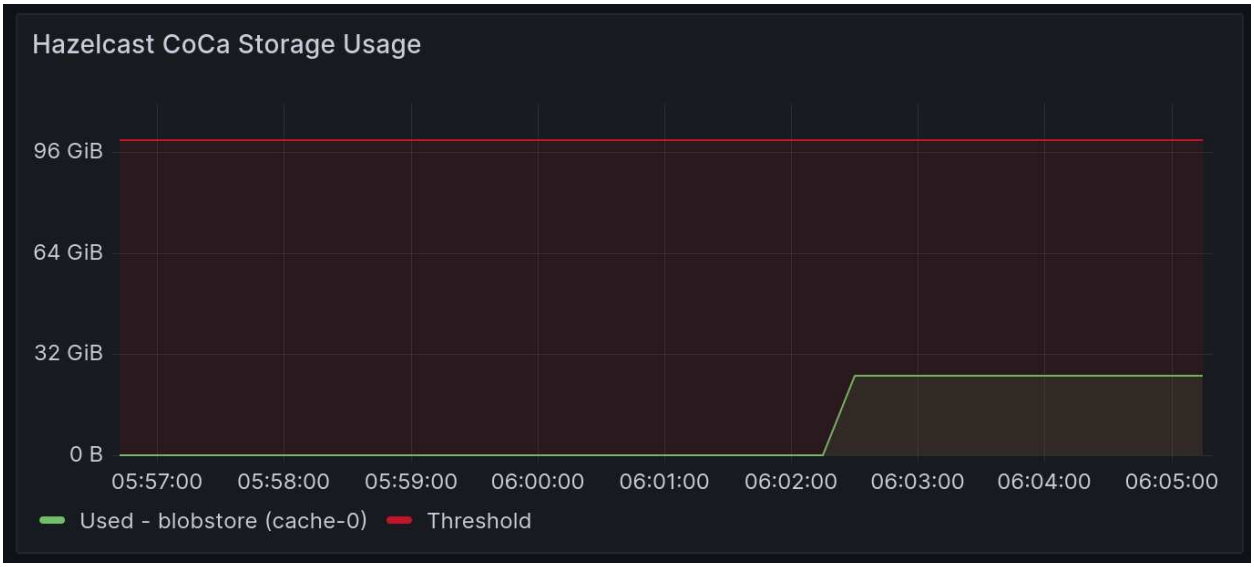
Check blob-storage-cleanup for the available configuration options.

How the Code Cache Cleaner Works

At a regular interval, the Code Cache cleaner checks whether the current usage is bigger than `targetSize`. If so, the system triggers the cleanup process. This process deletes the Code Cache items that applications used least recently 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 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 the system makes a promoted profile log 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 the system does not clean up unused profiles 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, the gateway service prints a warning in its log. 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 the ReadyNow Cleaner Works

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

- For each profileName: deletes all the profiles that the system does not promote at this time. The cleaner keeps the five last-used profileLogs. It only deletes enough profileLogs to get under `targetSize`.
  - The cleaner never deletes the currently promoted profiles.
- Deletes all debug-only profileLogs, meaning profileLogs with a generation higher than `readyNowOrchestrator.producers.maxPromotableGeneration`, if applications have not accessed them 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.
  - Deletes all of them, regardless of `targetSize`.

Size Value Formats

You can define size values in helm charts with different notations:

- Number format (existing)

  Example: `5000000`

- Decimal format (new): `M`, `G`,...

  Example: `5M` instead of `5000000`

- Binary format (new): `Mi`, `Gi`,...

  Example: `20Mi` instead of `20971520`

Configuring Optimizer Hub SSL Authentication

The recommended setup has a load balancer or service mesh in front of the Optimizer Hub service, see load-balancer. This serves 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, you can configure Optimizer Hub itself 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 enable SSL 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 you provide to client JVMs via the `-XX:OptHubHost` flag. Otherwise there may be issues when connecting.

2. Enable SSL in your `values-override.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:
     existingSecret: "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 "Standard Optimizer Hub Installation Procedure on Kubernetes".

Check ssl-parameters for the available configuration options.

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 client machine where you run Zing trusts the service certificate. You can achieve this by having a publicly trusted certificate authority sign the certificate.

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, the system cannot find your `.pem` file, or the file doesn't 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 Only use Optimizer Hub without SSL 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 had the name `-XX:+/-CNCInsecure`. Because of this change, you need to review your settings.

If you attempt to connect an Optimizer Hub that runs 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.
```

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 with all available services or "without Cloud Native Compiler". According to the selected services, different scaling approaches become necessary.

Full Optimizer Hub with Cloud Native Compiler (CNC)

The CNC service must be able to autoscale rapidly to handle resource demands effectively. When under heavy load, for example, during a fleet restart, CNC 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.

You can control how many compilation requests each JVM can send simultaneously to Optimizer Hub with the `compilations.parallelism.limitPerVm` setting. This client-side rate limiter prevents any single VM from overwhelming the compilation service. The default value of 500 ensures that VMs do not send more than this amount at the same time, balancing client throughput with server capacity.

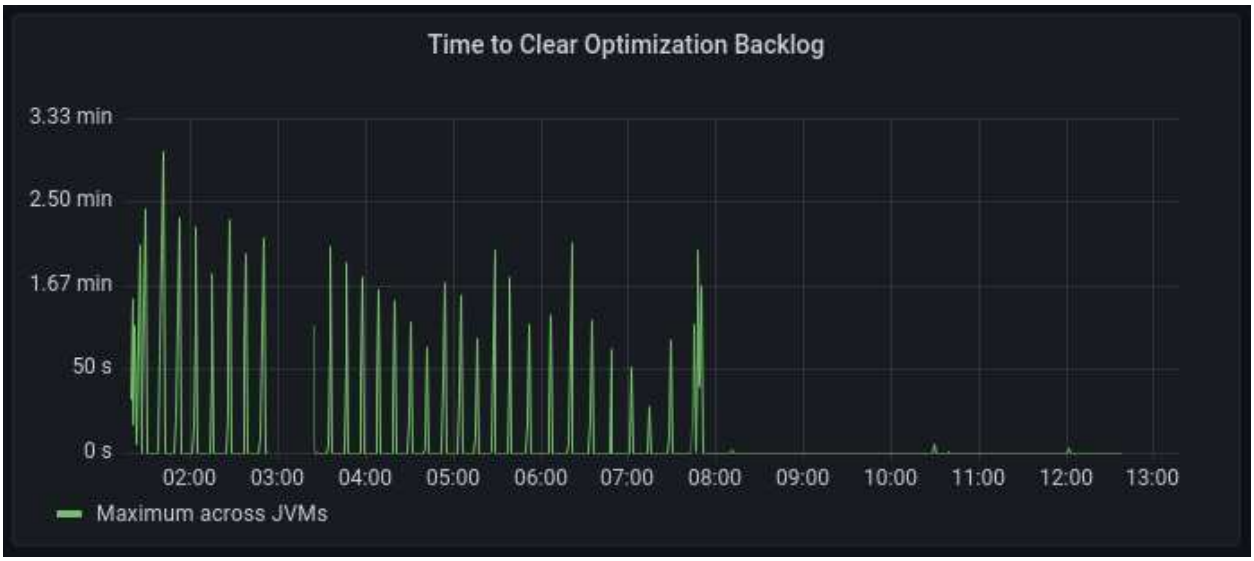
Check compilations for the available configuration options.

ReadyNow Orchestrator (RNO) Only

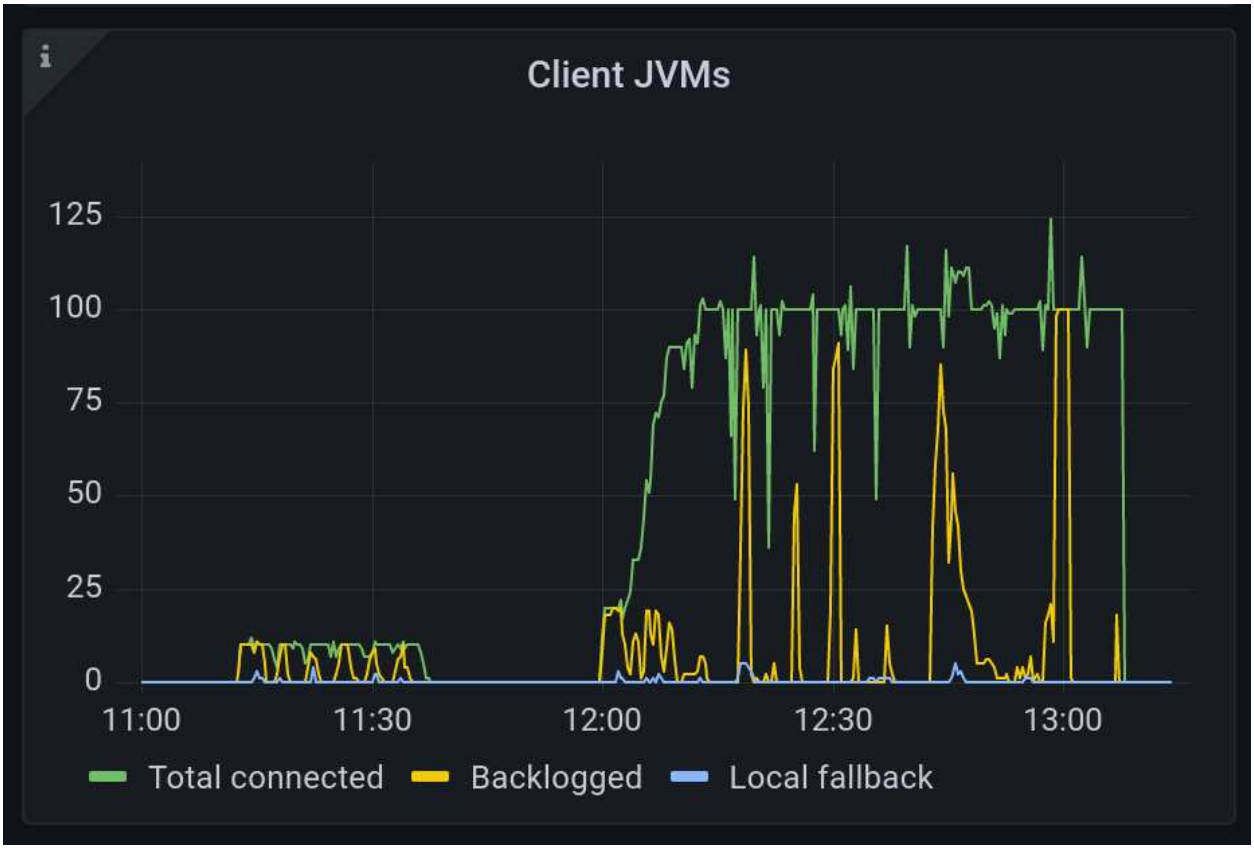
When you configure Optimizer Hub in RNO-only mode (using `values-disable-compiler.yaml`, see "Installing Optimizer Hub without Cloud Native Compiler"), it does not need to scale. The predefined sizing handles full RNO functionality.

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 number of optimization requests are put in the compilation queue. Eventually, the compiler catches up with the optimization backlog and all new compilation requests start within 2 seconds after the system has put them in 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 clears and all requests start within 2 seconds.



Controlling the Scaling with vCores

By default, Optimizer Hub uses 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 does 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. For example, based on your needs, you can configure a higher maximum:

```
simpleSizing:
  maxVCores: 500
```

Optimizer Hub uses the minimum and maximum number of vCores 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.

Check simple-sizing-parameters for the available configuration options.

Scaling of the Gateway Service

By default, Optimizer Hub has simple sizing enabled and one Gateway service. Depending on your use case, you can increase this by overriding the default values in your `values-override.yaml` file.

- For systems using Cloud Native Compiler (= when you enable all services) that can scale up and down:
  - With simple sizing enabled, an additional Gateway pod requires 7 vCores extra:

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

  - With simple sizing disabled:

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

- For systems with ReadyNow Orchestrator only, which don't scale up and where simple sizing is disabled:

```
gateway:
  replicas: 2
```

Connection Limit Impact on Scaling

Gateway scaling is not only about CPU and compilation throughput, you also need to take the connection capacity into account. When the per-gateway connection limit is reached, for example, by many long-lived JVM connections, new JVMs cannot connect to the Gateway. If those new JVMs are the ones requesting compilations, Optimizer Hub does not receive those requests and stays at a minimum size even though it needs to scale-up.

You need to configure the minimum number of Gateway instances to guarantee enough connection slots for all concurrently connected JVMs, including JVMs that are currently idle.

See gateway-connection-capacity for the infrastructure side of this limit.

Scaling API

The api-methods allows you to 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 takes all given timestamps and potential overlaps into account to start and stop the extra resources.

Connecting JVM 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.

Make sure you install Azul Zing Builds of OpenJDK (Zing) 24.02 or newer on your client machines. Starting from release 26.02 of Zing, OpenSSL is no longer included. If your Optimizer Hub instance is configured with SSL, you must install your own SSL library in your application. We recommend simply adding an OpenSSL implementation to your Maven dependencies such as Netty or Conscrypt.

Ask your Optimizer Hub instance admin for the optimizer-host 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.

NOTE	Connected JVMs still consume Gateway connection capacity, even when they are not currently requesting compilations. For capacity planning guidance, see gateway-connection-capacity.

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.

NOTE	Google Cloud Profiler is not supported with Optimizer Hub and Cloud Native Compiler. See gcp-profiler for details.

Fallback to Local JIT Compilation

When the Cloud Native Compiler component becomes unavailable or is unable to deliver compilations at acceptable speeds, the Zing JVM falls back to local compilation to protect application performance.

Fallback Reasons

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 trigger local fallback include:

- The service does not have the corresponding "Compiler Engine".
- The service is down or the JVM cannot reach it.
- The service is delivering compilations at an unacceptably slow rate for an extended period. This can occur in several scenarios:
  - Your Optimizer Hub instance is underprovisioned and does not have enough capacity to handle the number of incoming compilation requests. Too many clients connecting at the same time causes compilations to queue up, and application productivity can drop below what local JIT compilation would provide. The Zing JVM may switch to local fallback in this situation. See "Sizing and Scaling your Optimizer Hub Installation" for more info.
  - Even with autoscaling enabled, new resources take time to come online during traffic burst. During deployment waves or other sudden spikes in client connections, a large compilation backlog can build up before Optimizer Hub can catch up. Application productivity may temporarily drop during this period, and the Zing JVM may use local fallback until capacity recovers.
  - When you deploy a new version of your application under a different profile, Optimizer Hub treats it as an unknown application and starts profiling from scratch. A newly started profile tends to have lower hit rates and slower optimizations for the first few instances involved.

When the JVM detects that Cloud Native Compiler is consistently delivering compilations slower than the client's local JIT capabilities, it automatically switches to local fallback to maintain acceptable service levels.

Local Fallback Baseline and Configuration

Before deploying Cloud Native Compiler in production, you must establish a **Local Fallback Baseline** for your application. This baseline represents the acceptable performance characteristics when running with local JIT compilation and depends on the CPU resources you have allocated to your JVM instances.

Understanding Your Local Resource Classification

The viability of local fallback depends on the CPU resources available in your deployment for the JIT compiler threads:

- **No CPU/Minimal CPU**: Some deployments have aggressively reduced CPU allocation after migrating to Cloud Native Compiler. In these cases, any use of local JIT can cause critical issues. If this describes your environment, you may need to disable performance-based fallback or ensure an adequate CPU is available for emergency fallback scenarios.
- **1 vCPU or limited compiler threads**: This is the most common case for mature Cloud Native Compiler deployments. Local compilation is typically slower than optimal but can serve as a disaster recovery mechanism.
- **Sufficient CPU for local JIT**: Applications that recently migrated to Cloud Native Compiler or retained their original CPU allocation. These deployments can effectively use local fallback without service degradation.

Establishing Your Local Fallback Baseline

You need to verify two scenarios:

1. **Cloud Native Compiler Success Criteria**: Your application meets acceptable performance when Cloud Native Compiler is fully operational:
    - Response latencies meet your SLAs
    - CPU utilization is within expected ranges
    - No critical outages or problems when restarting fleets under normal operations
2. **Local Fallback Baseline**: Your application can survive when switching to local fallback:
    - Response latencies and CPU utilization are higher but still within acceptable limits
    - Time to full speed is longer, and the carrying capacity of each server is lower (requiring more instances)
    - No critical outages related to pods starting without fully operational Cloud Native

Compiler

Configure your Local Fallback Baseline with `-XX:CNCLocalFallbackOptLevelLimit=<3,2,1,0>` and `-XX:CIMaxCompilerThreads=<n>` to tune local compilation behavior. For more information, check the Zing documentation > Falcon Compiler Options > -XX:FalconOptimizationLevel.

NOTE	You must have monitoring in place to measure response latencies, CPU utilization, and detect outages during fleet restarts.

Performance-Based Local Fallback

The Zing JVM continuously monitors the rate at which Cloud Native Compiler delivers compiled code to the client. The JVM enters the local fallback when:

- The number of compilations that Cloud Native Compiler delivers per second falls below a threshold based on your local compiler thread configuration and optimization level.
- The remote compilation queue has reached a significant depth.
- Both conditions persist for several seconds (default 5 seconds).

When in local fallback, the JVM performs both local and remote compilations simultaneously. This allows the JVM to:

- Provide immediate compilation results from local resources.
- Continue monitoring Cloud Native Compiler performance to detect when it recovers.
- Use the first available compilation from either source.

The JVM exits local fallback and returns to Cloud Native Compiler-only mode when both of the following conditions hold true:

- The remote compilation queue clears below the threshold.

- The rate of incoming code deliveries from Cloud Native Compiler consistently exceeds the local capability threshold.

Check the available `Productivity` flags in the list of Cloud Native Compiler JVM Options to configure Performance-Based Local Fallback.

Automatic Switch Back to Cloud Native Compiler

The following output in the JVM `concomp` log shows when the JVM enables and disables fallback to local JIT compilation:

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

When local fallback is active, the JVM continues to request and receive compilations from Cloud Native Compiler while also compiling locally. You may observe both local and remote compilation activity in the logs during this period.

Recovering Methods Compiled During Local Fallback

Methods compiled while local fallback is active can be compiled at a lower optimization level than the one they were intended to reach, as set by `-XX:CNCLocalFallbackOptLevelLimit`. Switching back to Cloud Native Compiler does not by itself restore those methods, so an application can keep running capped code long after the service has recovered.

Starting with Azul Zing 26.08.0.0, the JVM periodically checks for methods still capped from a fallback period and recompiles them at their intended optimization level once Cloud Native Compiler is reachable again. This happens automatically when `-XX:CNCLocalFallbackOptLevelLimit` is used.

You can observe this activity in the GC log with `-XX:+GCLogPrintRecompStatistics` (enabled by default), which reports the number of methods checked, the number queued for recompilation, and the time spent checking. See Command Line Options in the Azul Prime documentation for details.

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.
```

Cloud Native Compiler JVM Options

NOTE

The minimum JVM options to request compilations from Cloud Native Compiler are `-XX:OptHubHost={value}` and `-XX:+CNCEnableRemoteCompiler`. It's also advised to specify `-XX:ProfileName=<value>` for improved Cloud Native Compiler caching.

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 as the 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`

Command Line Option	Description	Default
`-XX:ProfileName=<value>`	Name of the profile to identify the application and its version, for example, `my-spring-app-v1.2.3`. This name allows multiple JVM instances that use the same profile name to share the Optimizer Hub functionality, such as the Cloud Native Compiler caching. The ProfileName may only contain alphanumeric characters, and `-`, `_`, `.`, `~`.	null
`-XX:CNCEngineUploadAddress=<host:port>`	Address to upload the compiler engine. Only needed when your Optimizer Hub has non-standard ports. **NOTE** | Obsolete for Cloud Native Compiler 1.10 and above.	
`-XX:[+/-]CNCStartWithLocalCompiler`	Start the JVM with the local compiler and disables it when the connection to Cloud Native Compiler succeeds.	true
`-XX:CNCLocalFallbackOptLevelLimit=<3,2,1,0>`	Limit the OptLevel for local compilations when you enable Cloud Native Compiler. For more information, check the Zing documentation > Falcon Compiler Options > -XX:FalconOptimizationLevel.	`3`
`-XX:[+/-]CNCLocalFallbackProductivityBasedTriggering`	Enables and disables Local Fallback based on productivity checks. This flag does not control other existing local fallback mechanisms like connection loss and server circuitbreaker. Doesn't impact telemetry collection or telemetry reporting.	`true`

Command Line Option	Description	Default
`-XX:CNCProductivityLocalCompilationRatePerCompilerThread=<value>`	The number of compilations per second that the JVM presumes a local compiler thread can deliver when local productivity is being estimated for local fallback triggering purposes. The default value is derived from `FalconOptimizationLevel` and `CNCLocalFallbackOptLevelLimit`.	`5`
`-XX:CNCProductivityMinEvaluationWindowSeconds=<value>`	The minimal size of the time window in seconds for productivity evaluation.	`5`
`-XX:CNCProductivityMinOutstandingCompilesForEvaluation=<value>`	Determines if there is enough work on the server to be able to trigger local fallback.	`100`
`-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`

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 promoted" profile log, based on rules for picking the best log from within the qualifying candidates. 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 as the default. See "Connecting a JVM to Optimizer Hub" for instructions to determine the correct host and port.	null
`-XX:[+/-]EnableRNO`	Enables ReadyNow read and write to ReadyNow Orchestrator. Requires `OptHubHost` and uses `ProfileName` as the name for the profile log. **NOTE** Since Zing 26.08.0.0, `-XX:+EnableRNO` cannot be combined with `-XX:ProfileLogNam e`. If both are supplied, the JVM fails to start. Use `-XX:ProfileName` instead.	`false`
`-XX:ProfileName=<value>`	Name of the profile to identify the application and its version, for example, `my-spring-app-v1.2.3`. 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. The ProfileName may only contain alphanumeric characters, and `-`, `_`, `.`, `~`.	null

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://`, the `<value>` gets 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` [1]	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
	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`	null
`-XX:ProfileLogOutNominationMinTimeSec` [1]	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
	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`	null

Command Line Option	Description	Default
	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	0
`-XX:ProfileLogMaxSize=<value in bytes>` [1]	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 is 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
	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 is actually running. 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>%	Substitution 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%`

Using Compilation Streaming

Compilation Streaming lets a JVM download a stream of previously performed compilations from Optimizer Hub at startup and install them directly, instead of requesting each compilation individually as the application warms up. This shortens the time needed to clear the compilation queue and reduces load on Cloud Native Compiler.

Optimizer Hub builds each stream from its Code Cache and the promoted ReadyNow profile for a given profile name. Stream generation is automatic: when a profile is promoted, Optimizer Hub records a build request and the `task-executor` component produces the stream. At startup a JVM asks whether a stream exists for its profile. If one does, the JVM uses it. If not, the JVM runs normally without streaming and no error is reported.

Because promotion is what triggers a stream build, no stream exists for a profile name until a profile has been promoted for it. See promoted-profile.

Compilation Streaming requires Azul Zing 26.08.1.0 or later on the client side, and must be enabled on both the server and the JVM. On the JVM, add `-XX:+CNCEnableCompilationStreaming`. See Command Line Options in the Azul Prime documentation for the other JVM options that control it.

TIP	For background on how Compilation Streaming works, see "About Compilation Streaming".
TIP	For the complete Helm and JVM commands to enable Compilation Streaming on a new or existing installation, see "Installing Optimizer Hub with Compilation Streaming".

Enabling Compilation Streaming

Compilation Streaming requires one setting in your `values-override.yaml`:

```
compilationStreaming:
  enabled: true
```

`compilationStreaming.enabled` turns on the feature and its endpoints. The `task-executor` pod that builds the streams is deployed by default, so no second setting is needed.

IMPORTANT	If you disabled `task-executor` in your `values-override.yaml`, re-enable it with `taskExecutor.enabled:`

> `true`. Without it, the streaming endpoints start correctly but no streams are ever built, and JVMs continue without streaming and without reporting an error.

Configuration Options

Name	Type	Default	Description
`compilationStreaming.enabled`	boolean	false	Enables Compilation Streaming. Optimizer Hub builds streams from Code Cache and ReadyNow Orchestrator content, and the gateway's streaming endpoints become available. Requires `taskExecutor.enabled`, which is true by default.
`taskExecutor.enabled`	boolean	true	Deploys the `task-executor` pod, which builds streams from Code Cache content and the promoted ReadyNow profile. Without it, stream build requests are recorded but never executed.
`compilationStreaming.eviction.enabled`	boolean	true	Evicts stale compilation stream build requests.
`compilationStreaming.eviction.buildRequestRetention`	duration	P7D	How long a stream build request is retained before eviction, in ISO-8601 duration format. Applies only when `compilationStreaming.eviction.enabled` is true.

By default, the `task-executor` pod is sized at 4 CPU and 16 GiB of memory, with 10 GiB of ephemeral storage for temporary stream files. It is a fixed-size component that does not autoscale, and its cores are included in the `simpleSizing` vCore budget. See how-scales.

Storage Requirements

Stream segments are held in blob storage alongside your existing Code Cache and

ReadyNow profile data, so you need to allow for additional capacity when you enable the feature. See "Storage Requirements" for sizing guidance.

Stream Segments

A stream is divided into segments, and a JVM holds a limited number of segments in memory at any one time. The default segment size is 500 compilations. You do not need to configure segmentation, as Optimizer Hub produces segmented streams automatically.

If a specific workload needs a different segment size, you can override it per build using the `maxCompilationsPerSegment` builder option when submitting a stream build request.

On the client side, the number of segments held in memory at once is controlled by `-XX:CNCCompilationStreamingMaxActiveSegments`. Larger values raise the hit rate slightly at a significant cost in JVM memory, so a value of 3 to 5 suits most production workloads.

Compilation Streaming and Indirect Delivery

A single JVM cannot use both Compilation Streaming and indirect-delivery. You can leave both features enabled on the Optimizer Hub server. Each JVM resolves the conflict for itself at startup:

- If `-XX:RNOProfileTransportPriority` is left at its default, the JVM prints a warning and sets the transport to `grpc`. Compilation Streaming is used, and the profile arrives over gRPC.
- An explicit `-XX:RNOProfileTransportPriority=uri` causes a startup error.
- An explicit `-XX:RNOProfileSource=fallback` causes a startup error.

If you adopted Indirect Delivery to reduce gateway traffic, enabling Compilation Streaming returns profile transport to gRPC. Account for that when sizing your Optimizer Hub connection.

Monitoring

Optimizer Hub records the following metrics for Compilation Streaming:

Metric	Description
`compstream.delivery.requests`	Number of stream delivery requests received from JVMs.
`compstream.delivery.duration`	Time taken to serve stream delivery requests.
`compstream.build.task.duration`	Time taken to build a stream.
`compstream.build.file.size`	Size of the streams that were built.
`compstream.queue.requests.total`	Total number of stream build requests queued.
`compstream.queue.wait.duration`	Time that build requests spend waiting in the queue.

On the client side, streaming statistics appear in the JVM's GC log in the `COMPSTREAMSTATS` line, covering lookups, hits, misses, installation outcomes, segments loaded and evicted, and bytes downloaded. The hit rate derived from these values is the main indicator of whether streaming is working well for an application.

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 produces 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. The JVM uploads this compiler to Cloud Native Compiler 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. The system signs Compiler Engines to prevent the installation of malicious versions.

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 later 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.

Monitoring and Alerting

Monitoring Optimizer Hub

You can monitor your Optimizer Hub using one or more of the tools described below.

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.

Configuring Prometheus Path

By default, Prometheus metrics are provided at the path `/q/metrics`. You can specify a custom path in your `values-override.yaml` file:

```
metricsPath: "/your_path"
```

Using the REST APIs

The "Optimizer Hub REST APIs" are mostly intended for operational concerns, but you can use them to get information about how your Optimizer Hub instance is used. We

recommend writing a script that scrapes this API regularly for a historical view of the Optimizer Hub usage over time.

Remember that each app is identified by a profile name, that you can configure on the JVM-side with `-XX:ProfileName=<name>`. 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.

Some important questions you can answer with the API:

- *Of the currently connected client JVMs, how many are running which applications?*

  Use `/api/currentlyConnectedProfileNames` to retrieve a list of currently connected profile names and check the value of `currentlyConnectedVMInstanceCount`.

  An example: you can use this approach to find out which two clients are stuck in a reboot loop, causing Cloud Native Compiler to not scale down.

- *Which application groups are using ReadyNow Orchestrator and which ones are using Cloud Native Compiler?*

  Call `/api/profileNames` recursively to get the full list of profile names in an Optimizer Hub instance's memory then sort by `"cncActive": false, "rnoActive": false`

- *How much compute power is each application using?*

  You may need to charge back the cost for your Optimizer Hub environment to the individual teams, each of which may consume a different amount of resources. Since Cloud Native Compiler is the major consumer of resources, you only need to report per-profile name resource consumption for Cloud Native Compiler. Below you can find two example scripts to achieve this.

  You must run this first script periodically (e.g., once a minute) to scrape data from `/api/currentlyConnectedProfileNames`, and save the result to separate

JSON files.

```
import requests
import json

api_base_url = 'http://<your-instance>:8120'

def get_currently_connected_profiles(page_token = None):
    params = {'pageToken': page_token} if page_token is not None else {}
    response = requests.get(f'{api_base_url}/api/currentlyConnectedProfileNames
', params)
    response.raise_for_status()
    return response.json()

all_profiles = []
page_token = None

while True:
    response = get_currently_connected_profiles(page_token)
    all_profiles += response['data']
    page_token = response['nextPage']
    if page_token is None:
        break

print(json.dumps(all_profiles, indent = 2))
```

Once enough JSON files are generated, you can aggregate them (important: you need to do this in the order they were taken) to produce a table of resource usage per profile name for the period of time covered by those scrapes.

```
import sys
import json
from dataclasses import dataclass
from typing import Dict

@dataclass
class Accumulator:
    prev_sample: float = 0
    total: float = 0
    def add(self, sample: float) -> None:
        if sample >= self.prev_sample:
            self.total += sample - self.prev_sample
        else:
            self.total += sample
        self.prev_sample = sample

def read_scrape(path: str) -> Dict[str, float]:
    profile_usage: Dict[str, float] = {}
    with open(path, 'r') as f:
        data = json.load(f)
    for profile in data:
        name = profile['name']
        usage = (profile['cnc'] or {}).get('resouceUsage', 0)
        profile_usage[name] = usage
```

```
    return profile_usage

def main(scrape_paths) -> None:
    usage_by_profile: Dict[str, Accumulator] = {}
    for path in scrape_paths:
        scrape = read_scrape(path)
        for name, usage in scrape.items():
            usage_by_profile.setdefault(name, Accumulator())
            usage_by_profile[name].add(usage)

    total_usage = sum(acc.total for acc in usage_by_profile.values())

    for name, acc in sorted(usage_by_profile.items(), key=lambda item: item[1
].total, reverse=True):
        percentage = 100 * acc.total / total_usage if total_usage else 0
        print(f'{name:<30} {acc.total:<15.0f} {percentage:>6.2f}%')

if __name__ == "__main__":
    main(sys.argv[1:])
```

This lists the detected profiles with a percentage of use:

```
exampleProfileName1       7500048012       81.08%
exampleProfileName2       1750024006       18.92%
```

You can apply these percentages to the total amount your organization has spent on the hosting infrastructure for Optimizer Hub for the same period to split the costs between the applications.

Retrieving Optimizer Hub Logs

All Optimizer Hub components, including third-party ones, log some information to `stdout`. These logs are 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 later logs.

Configure log aggregation from all Optimizer Hub components so that the system moves logs to some persistent storage. Then you can extract them from the storage

when you need to analyze some issue. 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="my-opthub"}'
```

- Get logs of a single application in the selected namespace

```
logcli query --since 24h --forward --limit=10000 '{namespace="my-opthub"
app="compile-broker"}'
```

- Get logs of a single pod in the selected namespace

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

Extracting Compilation Artifacts

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

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

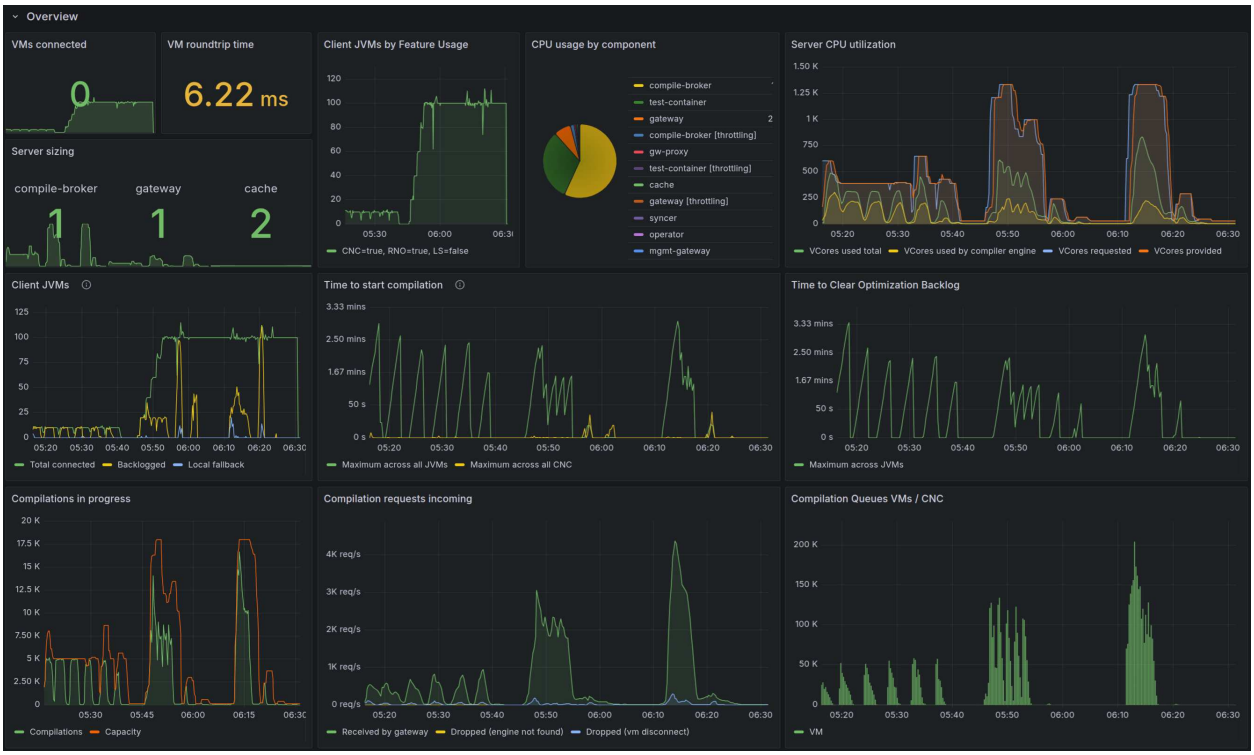
# Example:
[0.647s][info ][concomp] [ConnectedCompiler] received new VM-Id: 4f762530-8389-
4ae9-b64a-69b1adacccf2
```

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 the system processes metrics.

Configuring Prometheus and Grafana

The Optimizer Hub components expose key metrics for scraping by Prometheus. But to be able to monitor this info in a Grafana dashboard, you must perform some additional configuration.



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 can use the integrated services in Optimizer Hub (available since 25.11.0). For testing and evaluation, specific instructions are provided in the installation instructions for minikube and microk8s.

Using the Integrated Prometheus and Grafana

You need these additional steps to configure the integrated Prometheus and Grafana in Optimizer Hub. The system does not include them by default.

- Create a file for the role creation: `metrics-rbac.yaml`
  - Copy the file from the GitHub opthub-helm-charts repository.
  - Replace the namespace in two places `namespace: opthub` in the example file, or execute:

```
sed -i 's/opthub$/my-opthub/g' ~/my-opthub/metrics-rbac.yaml
```

  - If you create your service accounts separately, you need to match these names with the ones in this file.
  - Apply these roles:

```
kubectl apply -f ~/my-opthub/metrics-rbac.yaml
```

- Create a file with the monitoring configuration: `values-monitoring.yaml`
  - Copy the file from the GitHub opthub-helm-charts repository.
  - If you created your own service account, update this value in the file: `name: metrics-server`.
- Make sure you have the Kubernetes State Metrics service running.
  - Example output:

```
$ kubectl get deployments --all-namespaces | grep kube-state-metrics
monitoring  kube-state-metrics  1/1     1    1   2y27d

$ kubectl get services --all-namespaces | grep kube-state-metrics
kube-state-metrics  ClusterIP   172.20.61.221   <none>  8080/TCP    2y27d
```

  - If not available yet, install it with:

```
helm install kube-state-metrics \
    oci://ghcr.io/prometheus-community/charts/kube-state-metrics
```

Using Your Own Prometheus and Grafana

Use the following instructions to configure external Prometheus and Grafana instances in a production environment.

Prometheus Configuration Instructions

Optimizer Hub components expose their metrics on HTTP endpoints in a format compatible with Prometheus. The Helm chart includes annotations 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 becomes available and collects data from the Optimizer Hub Components, you can add a dashboard. In the GitHub opthub-helm-charts repository > config-files > grafana, you can find a Grafana configuration file.

This dashboard expects the following labels that you attach to all application metrics, referring to the Prometheus configuration above:

- `cluster_id`: The identifier of the Kubernetes cluster where you install Optimizer Hub. This allows you to switch between Optimizer Hub instances in different clusters.
- `kubernetes_namespace`: The Kubernetes namespace where you install Optimizer Hub. 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 the `labelmap` action supplies from the example Prometheus configuration mentioned below.

You need to manually edit the dashboard file if your environment names these labels differently.

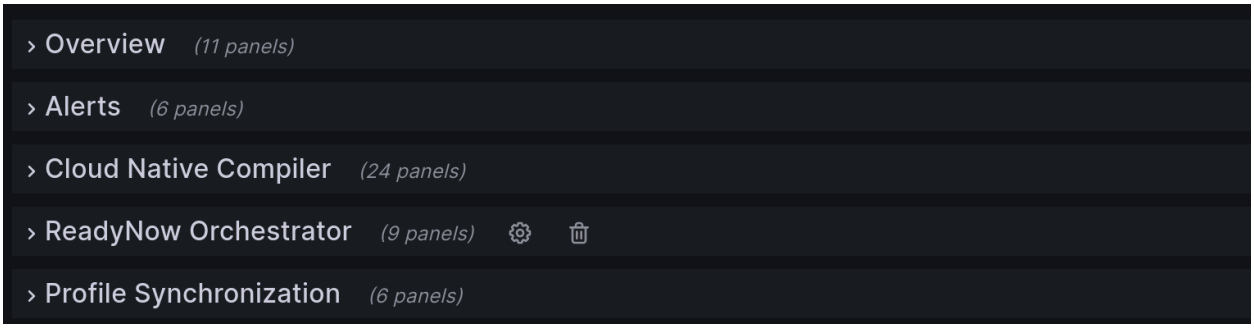
The dashboard also relies on some infrastructure metrics from Kubernetes and cAdvisor, such as `kube_pod_container_resource_requests` and `container_cpu_usage_seconds_total`. Therefore, the **containerd** container runtime is needed on your Kubernetes nodes to be able to correctly filter the data in Grafana.

Using the Grafana Dashboard

A Grafana dashboard becomes available after "Configuring Prometheus and Grafana". This helps you to understand how your Optimizer Hub instance is performing. The dashboard is divided into several sections. This document describes the most important sections from user-perspective. The other sections are more oriented towards maintaining and troubleshooting of the installation.

NOTE

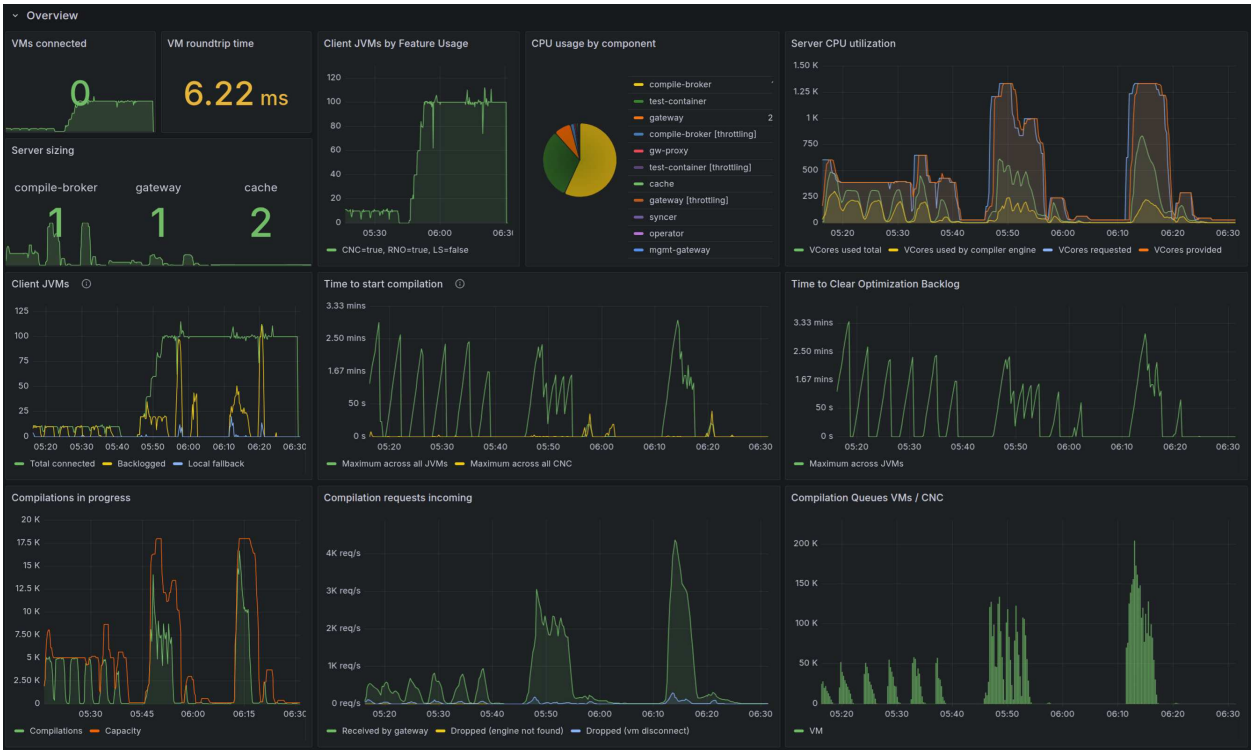
The Grafana dashboard relies on cAdvisor infrastructure metrics that require the **containerd** container runtime on your Kubernetes nodes for correct filtering in Grafana. For minikube, start the cluster with `--container-runtime=containerd`. MicroK8s uses containerd by default.



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.
- Compare the connected JVM count with your Gateway connection capacity to detect potential connection saturation early. See gateway-connection-capacity.



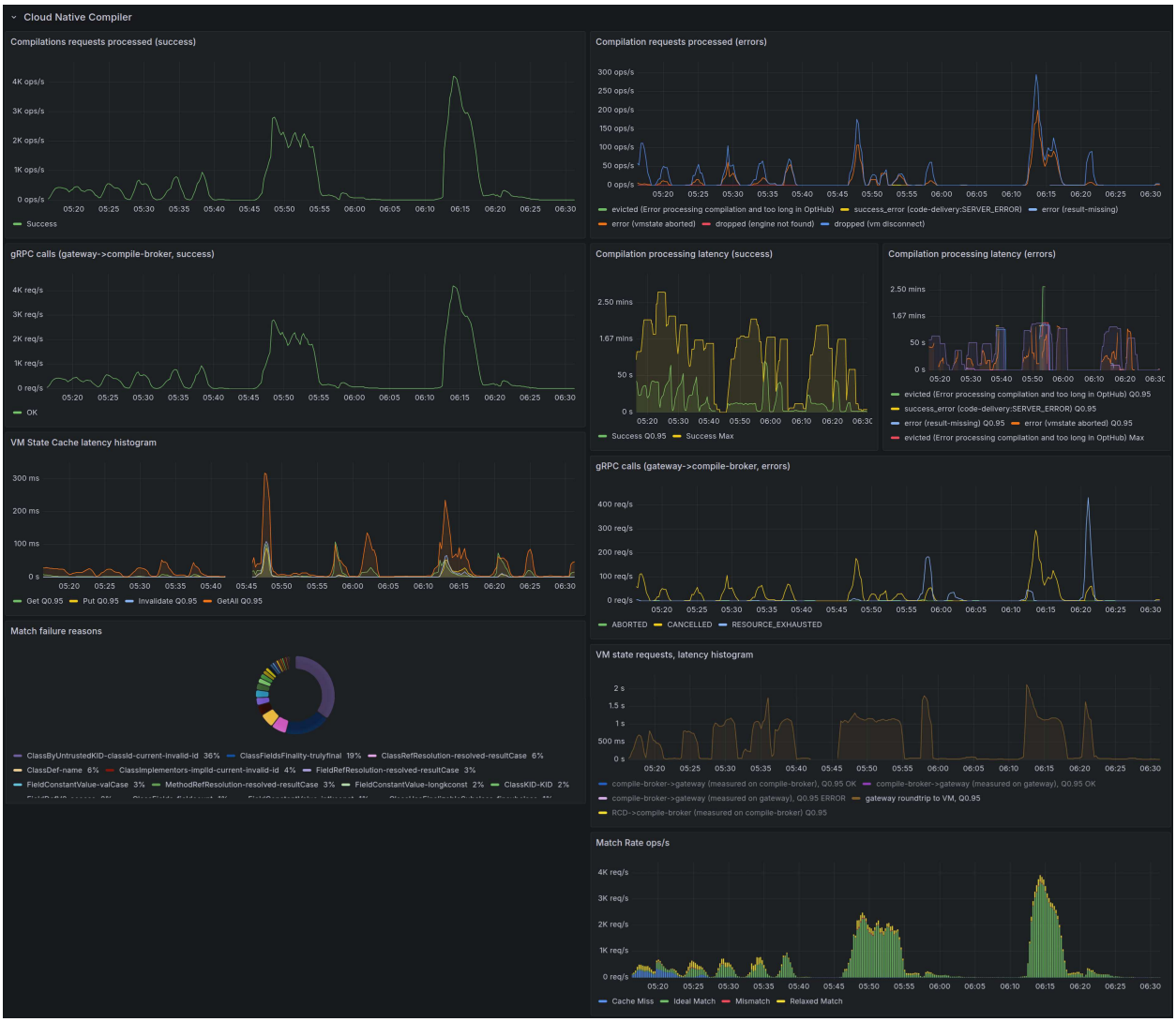
Alerts

Basic monitoring of problems in the Optimizer Hub cluster.



Cloud Native Compiler

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



ReadyNow Orchestrator

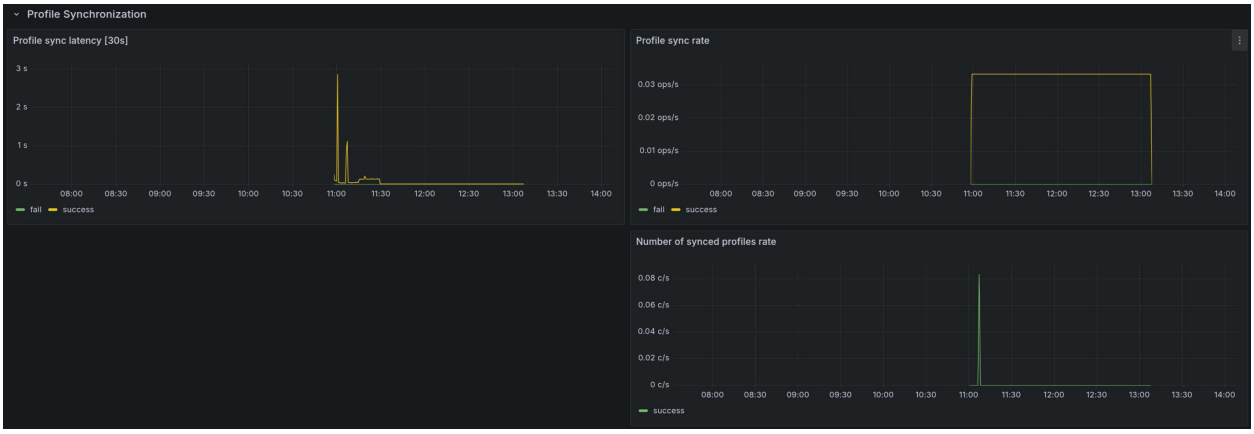
Monitoring of the ReadyNow Orchestrator feature. This is applicable for JVMs using Optimizer Hub ReadyNow Orchestrator.



Profile Synchronization

Applicable when you have multiple Optimizer Hub clusters and profiles synchronize 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



Optimizer Hub API

Optimizer Hub provides an API for several management tasks.

These methods are available on `{MANAGEMENT_GATEWAY_IP}:{SERVICE_PORT}/...` and you can access them without authentication. The service port typically is 8080. For security reasons, by default, the API does not expose its endpoints 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.

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 has been configured. If no host is specified, the default value `localhost:50051` gets 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 optimizer-host 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 optimizer-host 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

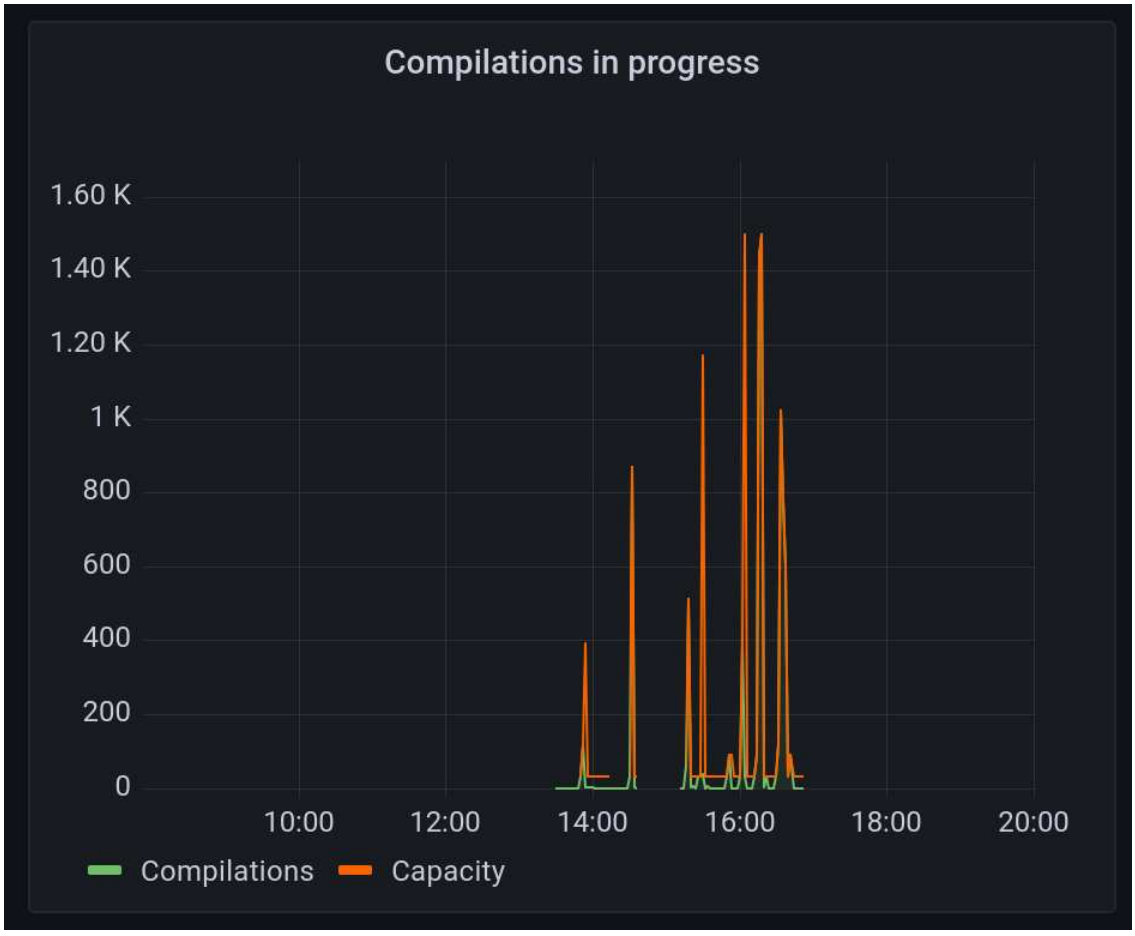
     # Example:
     [0.647s][info ][concomp] [ConnectedCompiler] received new VM-Id: 4f762530-
     8389-4ae9-b64a-69b1adacccf2
     ```

   - 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 you should scale it.

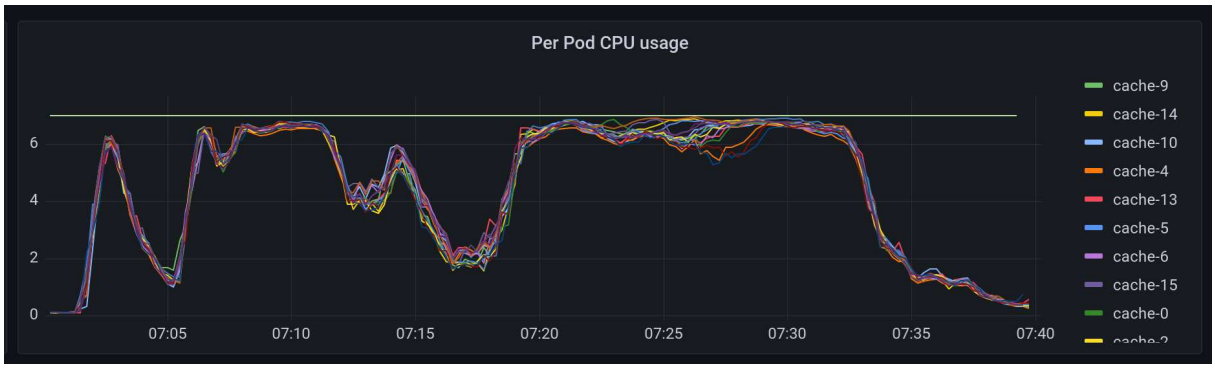


  - Otherwise the bottleneck is related to the per-VM limit on concurrent compilations. You should increase it. Scaling the server without increasing that per-VM limit doesn’t help.
- If TTCOB is below the threshold:

- 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.

On Google Cloud, the Cloud Native Compiler CodeCache hit rate suddenly collapses and Optimizer Hub CPU spikes.

Running JVMs with Google Cloud Profiler (`-agentpath:…/profiler_java_agent.so`) is not supported with Optimizer Hub Cloud Native Compiler and should be avoided.

The profiler can trigger JVMTI state changes such as `canhotswap`. Because such a

state is global per JVM (not per method), this can invalidate cache assumptions in Cloud Native Compiler forcing recompilation work that would otherwise be served from cache.

If you run on Google Cloud Platform, disable the Google Cloud Profiler agent on JVMs that use Cloud Native Compiler.

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 roundtrip" 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 an 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.

Optimizer Hub looks idle or stuck at minimum size, while some JVMs cannot get compilations. What can I do?

This can happen when the Gateway connection limit is saturated by already connected JVMs. New JVMs then cannot connect, so Optimizer Hub cannot see their compilation demand.

1. Check the number of connected JVMs in Grafana and compare it with your estimated Gateway connection capacity.
2. Check Gateway and proxy/load-balancer logs for connection limit or rejected-connection errors.
3. Confirm your minimum Gateway size (`gateway.autoscaler.min` or `gateway.replicas`) and compare it with your expected maximum number of

connected JVMs.

If the connection limit is the root cause:

- Increase the minimum number of Gateway instances as described in gateway-connection-impact.
- Consider increasing the per-gateway connection limit in your infrastructure.

See gateway-connection-capacity for planning guidance.

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 starts the load, thus limiting the overall effect of ReadyNow.

ReadyNow cannot download profiles with many parallel requests

In case the gateway log shows multiple errors , like `PoolAcquirePendingLimitException: Pending acquire queue has reached its maximum size of 1000" in gateway log`, you must increase the number of gateways. See scaling-gateway for more info.

Optimizer Hub API Troubleshooting

The Optimizer Hub API returns HTTP 500 on some endpoints after an upgrade. What can I do?

Restart the `cache` pods. Optimizer Hub does not support rolling or in-place upgrades, so the upgraded instance can be left holding data written by the previous version that it is unable to read. When this happens, the `mgmt-gateway` log reports a `ClassCastException`.

Scale the `cache` statefulset down to zero replicas, wait until all `cache` pods have terminated, then scale it back up. If the operator recreates the pods immediately, scale the operator down to zero as well. This clears only in-memory data, which is rebuilt as your JVMs reconnect.

To avoid the problem, shut the instance down completely before upgrading, as described in upgrading-or-rolling-back.

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.

Optimizer Hub Release Notes

Optimizer Hub 26.08.4

Release Date: November 12, 2026

New Features

- Includes bug fixes for Optimizer Hub 26.08.3.

CVE Fixes

CVE	Base Score	Component
CVE-2026-110879	CRITICAL 9.8	jackson-databind
CVE-2026-107726	CRITICAL 9.3	hazelcast
CVE-2026-87744	CRITICAL 9.1	quarkus
eMudhra intermediate fix	CRITICAL	envoy
CVE-2026-107725	HIGH 8.7	hazelcast
CVE-2026-93879	HIGH 7.5	quarkus
CVE-2026-94451	MEDIUM 6.5	quarkus

Optimizer Hub 26.08.3

Release Date: October 27, 2026

New Features

- Includes bug fixes for Optimizer Hub 26.08.2.
- Related to a second eMudhra CVE, CVE-2026-114820, the JDK and Docker base images are updated. There are no changes in the Optimizer Hub functionality.

Known Issues

- The optional `gw-proxy` component in the Optimizer Hub Helm chart uses Envoy 1.35, which contains known security vulnerabilities. Envoy 1.40.1 contains fixes for these vulnerabilities and has been tested with Optimizer Hub. If you keep using `gw-proxy`, we recommend using Envoy 1.40.1. Check which Envoy version your deployment runs. A Helm chart update with Envoy 1.40.1 will be available soon.

  Alternatively, you can replace `gw-proxy` with your own load balancer or ingress, see load-balancer.

CVE Fixes

CVE	Base Score	Component
CVE-2026-114820	CRITICAL 9.1	JVM

Optimizer Hub 26.08.2

Release Date: October, 2026

New Features

- Includes bug fixes for Optimizer Hub 26.08.1.
- Related to eMudhra Intermediate CA Compromise, the JDK and Docker base images are updated. There are no changes in the Optimizer Hub functionality.

Known Issues

- The optional `gw-proxy` component in the Optimizer Hub Helm chart uses Envoy 1.35, which contains known security vulnerabilities. We recommend replacing `gw-proxy` with your own load balancer or ingress, see load-balancer. If you keep using `gw-proxy`, check which Envoy version your deployment runs.

  Envoy 1.40.1 contains fixes for these vulnerabilities. Azul has not yet validated it with Optimizer Hub, so if you decide to use it, test it in a non-production environment first. A Helm chart update with a newer Envoy version will be available soon.

Optimizer Hub 26.08.1

Release Date: October 13, 2026

New Features

- Includes bug fixes for Optimizer Hub 26.08.0.
- Added the `gateway.compilerEngineUploadTimeout` Helm value (in seconds) to configure the timeout for compiler engine uploads through the Gateway. Increase it when the JVMs connect to Optimizer Hub over a high-latency network, where the default 30-second timeout can cause compilations to fail with an "Engine not found" error. See Network and Gateway Requirements.

Bug Fixes

- Fix for compilation streaming when used with compilation multi tiering (`-XX:+UseMultiTiering`). Before this fix, a compilation stream could not be built

from a profile recorded with multi tiering enabled, and the task executor failed with `No compilations were found in CoCa for profile …; stream can't be built`.

CVE Fixes

CVE	Base Score	Component
CVE-2026-75595	9.1	Netty
CVE-2026-8763	9.1	BouncyCastle
CVE-2026-58062	9.1	BouncyCastle
CVE-2026-59650	9.1	BouncyCastle
CVE-2026-12185	8.6	BouncyCastle
CVE-2026-58061	7.5	BouncyCastle
CVE-2026-58059	7.5	BouncyCastle
CVE-2026-13506	7.5	BouncyCastle
CVE-2026-59651	7.5	BouncyCastle
CVE-2026-14682	7.5	BouncyCastle
CVE-2026-12803	7.5	BouncyCastle
CVE-2026-58060	7.5	BouncyCastle
CVE-2026-12816	7.5	BouncyCastle
CVE-2026-13586	7.5	BouncyCastle
CVE-2026-40983	7.5	Micrometer
CVE-2026-40984	7.5	Micrometer
CVE-2026-12860	5.3	BouncyCastle
CVE-2026-58063	5.3	BouncyCastle

Optimizer Hub 26.08.0

Release Date: August, 2026

New Features

- Optimizer Hub now supports **Compilation Streaming**. Optimizer Hub builds a stream of previously performed compilations from its Code Cache and the promoted ReadyNow profile, and serves that stream to JVMs at startup. A JVM installs the streamed compilations directly instead of requesting them one at a time as the application warms up, which shortens the time needed to clear the compilation queue and reduces load on Cloud Native Compiler.

  Compilation Streaming requires Azul Zing 26.08.1.0 or later, and must be enabled on both the server and the JVM. On the server, set `compilationStreaming.enabled`. On the JVM, add `-XX:+CNCEnableCompilationStreaming`. See Installing Optimizer Hub with Compilation Streaming for the complete commands, and Using Compilation Streaming for the full reference.

NOTE	A single JVM cannot use both Compilation Streaming and Indirect Delivery of ReadyNow profiles. You can leave both enabled on the server: a JVM that uses Compilation Streaming prints a warning and receives its profile over gRPC instead.

- The `task-executor` component is now **deployed by default**. It builds the compilation streams used by Compilation Streaming, and refreshes the ReadyNow profiles used by Indirect Delivery, so neither feature needs a separate opt-in anymore. The pod is sized at 4 CPU and 16 GiB of memory, with 10 GiB of ephemeral storage for temporary stream files. Like `gw-proxy`, `mgmt-gateway`, and the operator, `task-executor` is a fixed-size component that does not autoscale, and its cores are now included in the `simpleSizing` vCore budget. If you use neither Compilation Streaming nor Indirect Delivery, you can disable it with `taskExecutor.enabled: false`.

- Optimizer Hub now supports **Indirect Delivery** of ReadyNow profiles. Instead of sending the input profile to the JVM over gRPC, Optimizer Hub stores the profile in

blob storage and hands the JVM a pre-signed URI, which the JVM uses to download the profile over HTTP. Use this to reduce the volume of traffic through the Optimizer Hub gateway, or when your network topology makes a direct gRPC transfer of large profiles impractical. Enable it on the server with `indirectDelivery.enabled: true`. See indirect-delivery.

- In Azul Zing Builds of OpenJDK, version 26.08.0.0 and newer, methods compiled at a reduced optimization level during a local fallback period, as capped by `-XX:CNCLocalFallbackOptLevelLimit`, are guaranteed to be recompiled at their intended level once Cloud Native Compiler is reachable again. See automatic-switch-back.
- Includes bug fixes for Optimizer Hub 26.05.0.

CVE Fixes

CVE	Base Score	Component
CVE-2026-54512	8.1	jackson-databind
CVE-2026-54513	8.1	jackson-databind
CVE-2026-56819	7.5	Netty
CVE-2026-59898	7.5	Netty
CVE-2026-59899	7.5	Netty
CVE-2026-59901	7.5	Netty
CVE-2026-56746	6.5	Netty
CVE-2026-59888	6.5	jackson-databind
CVE-2026-59921	6.5	Netty
CVE-2026-54514	5.3	jackson-databind
CVE-2026-54515	5.3	jackson-databind
CVE-2026-59900	5.3	Netty

Optimizer Hub 26.05.5

Release Date: November 19, 2026

New Features

- Includes bug fixes for Optimizer Hub 26.05.4.

CVE Fixes

CVE	Base Score	Component
CVE-2026-114938	8.1	JVM

Optimizer Hub 26.05.4

Release Date: November 12, 2026

New Features

- Includes bug fixes for Optimizer Hub 26.05.3.

CVE Fixes

CVE	Base Score	Component
CVE-2026-110879	CRITICAL 9.8	jackson-databind
CVE-2026-107726	CRITICAL 9.3	hazelcast
CVE-2026-87744	CRITICAL 9.1	quarkus
eMudhra intermediate fix	CRITICAL	envoy
CVE-2026-107725	HIGH 8.7	hazelcast
CVE-2026-93879	HIGH 7.5	quarkus
CVE-2026-94451	MEDIUM 6.5	quarkus

Optimizer Hub 26.05.3

Release Date: October 27, 2026

New Features

- Includes bug fixes for Optimizer Hub 26.05.2.
- Related to a second eMudhra CVE, CVE-2026-114820, the JDK and Docker base images are updated. There are no changes in the Optimizer Hub functionality.

Known Issues

- The optional `gw-proxy` component in the Optimizer Hub Helm chart uses Envoy 1.35, which contains known security vulnerabilities. Envoy 1.40.1 contains fixes for these vulnerabilities and has been tested with Optimizer Hub. If you keep using `gw-proxy`, we recommend using Envoy 1.40.1. Check which Envoy version your deployment runs. A Helm chart update with Envoy 1.40.1 will be available soon.

  Alternatively, you can replace `gw-proxy` with your own load balancer or ingress, see load-balancer.

CVE Fixes

CVE	Base Score	Component
CVE-2026-114820	CRITICAL 9.1	JVM

Optimizer Hub 26.05.2

Release Date: October, 2026

New Features

- Includes bug fixes for Optimizer Hub 26.05.1.
- Related to eMudhra Intermediate CA Compromise, the JDK and Docker base images are updated. There are no changes in the Optimizer Hub functionality.

Known Issues

- The optional `gw-proxy` component in the Optimizer Hub Helm chart uses Envoy 1.35, which contains known security vulnerabilities. We recommend replacing `gw-proxy` with your own load balancer or ingress, see load-balancer. If you keep using `gw-proxy`, check which Envoy version your deployment runs.

Envoy 1.40.1 contains fixes for these vulnerabilities. Azul has not yet validated it with Optimizer Hub, so if you decide to use it, test it in a non-production environment first. A Helm chart update with a newer Envoy version will be available soon.

Optimizer Hub 26.05.1

Release Date: October 13, 2026

New Features

- Includes bug fixes for Optimizer Hub 26.05.0.

CVE Fixes

CVE	Base Score	Component
CVE-2026-75595	9.1	Netty
CVE-2026-8763	9.1	BouncyCastle
CVE-2026-58062	9.1	BouncyCastle
CVE-2026-59650	9.1	BouncyCastle
CVE-2026-12185	8.6	BouncyCastle
CVE-2026-58061	7.5	BouncyCastle
CVE-2026-58059	7.5	BouncyCastle
CVE-2026-13506	7.5	BouncyCastle
CVE-2026-59651	7.5	BouncyCastle
CVE-2026-14682	7.5	BouncyCastle
CVE-2026-12803	7.5	BouncyCastle
CVE-2026-58060	7.5	BouncyCastle
CVE-2026-12816	7.5	BouncyCastle
CVE-2026-13586	7.5	BouncyCastle
CVE-2026-40983	7.5	Micrometer
CVE-2026-40984	7.5	Micrometer

CVE	Base Score	Component
CVE-2026-12860	5.3	BouncyCastle
CVE-2026-58063	5.3	BouncyCastle

Optimizer Hub 26.05.0

Release Date: June 25, 2026

New Features

- Includes bug fixes for Optimizer Hub 26.02.1.

Bug Fixes

- The Optimizer Hub ConfigMap is now immutable, which prevents Kubernetes from repeatedly syncing it across pods during scale up. This avoids overwhelming the Kubernetes API with sync requests that could previously cause `failed to sync configmap cache` errors and pod readiness issues when many pods started simultaneously.
- Server-side log messages now report the VM ID and profile name in a consistent format. Previously these values were logged inconsistently across different messages. They are now injected into the logging context (via MDC), making it easier to correlate and filter log entries by VM ID or profile.

CVE Fixes

CVE	Base Score	Component
CVE-2026-47691	10.0	Netty
CVE-2026-45674	10.0	Netty
CVE-2026-42581	9.8	Netty
CVE-2026-42584	9.1	Netty
CVE-2026-42579	9.1	Netty
CVE-2026-44249	8.1	Netty

CVE	Base Score	Component
CVE-2026-50011	7.5	Netty
CVE-2026-50010	7.5	Netty
CVE-2026-48059	7.5	Netty
CVE-2026-48043	7.5	Netty
CVE-2026-48006	7.5	Netty
CVE-2026-46340	7.5	Netty
CVE-2026-45416	7.5	Netty
CVE-2026-44890	7.5	Netty
CVE-2026-44250	7.5	Netty
CVE-2026-44248	7.5	Netty
CVE-2026-42587	7.5	Netty
CVE-2026-42585	7.5	Netty
CVE-2026-42583	7.5	Netty
CVE-2026-42578	7.5	Netty
CVE-2026-33871	7.5	Netty
CVE-2026-33870	7.5	Netty
CVE-2025-55163	7.5	Netty
CVE-2026-42586	7.1	Netty
CVE-2026-42580	6.5	Netty
CVE-2026-45673	6.8	Netty
CVE-2026-50560	5.3	Netty
CVE-2026-50020	5.3	Netty
CVE-2026-47244	5.3	Netty
CVE-2026-41417	5.3	Netty
CVE-2026-45536	4.0	Netty

CVE	Base Score	Component
CVE-2026-6860	5.3	Eclipse Vert.x (io.vertx):

Optimizer Hub 26.02.1

Release Date: May 27, 2026

New Features

- Includes bug fixes for Optimizer Hub 26.02.0.
- The liveness probe in the cache service is extended and detects if a Hazelcast member shuts down.

CVE Fixes

CVE	Base Score	Component
CVE-2026-39852	8.2	Quarkus

Optimizer Hub 26.02.0

Release Date: April 3, 2026

New Features

- Includes bug fixes for Optimizer Hub 25.11.0.
- In certain scenarios where you need to upload a profile to Optimizer Hub instances, you can now use the new upload-profile.
- Several improvements in the OpenAPI documentation. See the Optimizer Hub API documentation for more details.
- The default path for Prometheus metrics, `/q/metrics` can now be configured. See configuring-prometheus-path.
- The profile cleaner can now automatically clean up artifacts from compiler crashes. When the profile cleaner deletes a profile, it also deletes the compiler artifacts for

that profile. The cleaner also removes artifacts based on age. You can configure this behavior, see compile-storage-cleanup for more details.

- General cleanup of the Helm charts. The charts have been restructured and simplified to make it easier for teams that do not use Helm to extract the configuration and recreate it in their own Kubernetes deployment process. Some configuration options have also been removed. If you use `storage.s3.credentialsType` in your `values-override.yml` file, it should be removed. Check Storage Requirements for the up-to-date information.
- The different sections in the Grafana dashboard got reorganized and simplified to provide more clear views on the behavior of your Optimizer Hub instance. See Using the Grafana Dashboard for more details.

CVE Fixes

CVE	Base Score	Component
CVE-2025-66560	7.5	arc
CVE-2026-1002	5.3	vertx-web-client
CVE-2025-53057	5.9	zulu
CVE-2025-53066	7.5	zulu

Optimizer Hub 25.11.1

Release Date: April 28, 2026

New Features

- Includes bug fixes for Optimizer Hub 25.11.0.

CVE Fixes

CVE	Base Score	Component
CVE-2025-66560	7.5	arc
CVE-2026-1002	5.3	vertx-web-client

CVE	Base Score	Component
CVE-2025-53057	5.9	zulu
CVE-2025-53066	7.5	zulu

Optimizer Hub 25.11.0

Release Date: December 15, 2025

New Features

- Includes bug fixes for Optimizer Hub 25.08.1.
- The ReadyNow Orchestrator API now returns the duration of a profile, next to its size. Both values are important characteristics for profile promotion, and their thresholds can be configured. By providing these in the API result, it becomes easier to identify why a profile was not promoted.
- Extra configuration files and installation steps are provided to install Prometheus and Grafana:
  - integrated.
  - As part of a minikube and microk8s setup.
- Annotations can now also be added to Deployment objects, next to the existing custom labels-kubernetes-objects. Examples from `values.yaml`:

```
# Existing
applicationLabels: # Additional labels for Deployment/StatefulSet
applicationAnnotations: # Additional annotations for Deployment/StatefulSet
podTemplateLabels: # Additional labels for POD
serviceLabels: # Additional labels for Service

# New
podTemplateAnnotations: # Additional annotations for POD
```

- In some deployments, ReadyNow profiles with sizes of multiple GBs, were stored. Such big profiles are not needed and can impact system performance and stability. Therefore, the default value of `readyNowOrchestrator.producers.maxProfileSize` is changed from

unlimited to 300MB.

- Similar to ReadyNow Orchestrator, you can now also configure the Code Cache cleaner with human-readable parameters for `codeCache.cleaner.targetSize`. For example:
  - Number format (existing)

    Example: `5000000`
  - Decimal format (new): `M`, `G`,...

    Example: `5M` instead of `5000000`
  - Binary format (new): `Mi`, `Gi`,...

    Example: `20Mi` instead of `20971520`

CVE Fixes

CVE	Base Score	Component
CVE-2025-11965	6.3	vertx-web (4.5.18)
CVE-2025-11966	2.3	vertx-web (4.5.18)

Optimizer Hub 25.08.1

Release Date: November 18, 2025

New Features

- Includes bug fixes for Optimizer Hub 25.08.0, aligning with the changes in 1.11.6.
- A new metric `hz_operations_pending_invocations` exposes the number of pending invocations on the "cache" (Hazelcast) node. A high number can indicate a problem with the processing, and can be used to trigger alerts.

Bug Fixes

- The Optimizer Hub Operator can be configured to perform regular restarts of cache

Kubernetes pods. This is disabled by default but can be enabled in case of stability issues in cache pods. Please contact our support team if you want to make use of this feature.

- A fix was applied for cases where the cache component of Optimizer Hub froze, causing disruption of service.

CVE Fixes

CVE	Base Score	Component
CVE-2025-58057	6.9	Netty
CVE-2025-58056	2.9	Netty
CVE-2025-48924	5.3	Apache Commons Lang
CVE-2025-27817	7.5	Apache Kafka Client

Optimizer Hub 25.08.0

Release Date: September 17, 2025

New Features

- Includes bug fixes for Optimizer Hub 1.11 and 25.05.
- You can now define size values in helm charts with different notations. The changes are backwards compatible, so you can still use the previous values.
  - Number format (existing)

    Example: `5000000`
  - Decimal format (new): `M`, `G`,...

    Example: `5M` instead of `5000000`
  - Binary format (new): `Mi`, `Gi`,...

    Example: `20Mi` instead of `20971520`

- The gRPC-port used by the Gateway pod can be changed in your `values-override.yaml` file in case you want to override the default values:

```
gateway:
  ports:
    serviceGrpcPort: 50051
    internalGrpcPort: 50052
cache:
  ports:
    internalGrpcPort: 50071
```

Optimizer Hub 25.05.2

Release Date: November 12, 2025

New Features

- Includes bug fixes for Optimizer Hub 25.05.1, aligning with the changes in 1.11.6.
- A new metric `hz_operations_pending_invocations` exposes the number of pending invocations on the "cache" (Hazelcast) node. A high number can indicate a problem with the processing, and can be used to trigger alerts.
- Improvements in:
  - Tracking of JVM to Optimizer Hub connections.
  - Detection of stale connections.

Bug Fixes

- The Optimizer Hub Operator can be configured to perform regular restarts of cache Kubernetes pods. This is disabled by default but can be enabled in case of stability issues in cache pods. Please contact our support team if you want to make use of this feature.

CVE Fixes

CVE	Base Score	Component
CVE-2025-58057	6.9	Netty

CVE	Base Score	Component
CVE-2025-58056	2.9	Netty
CVE-2025-48924	5.3	Apache Commons Lang
CVE-2025-27817	7.5	Apache Kafka Client

Optimizer Hub 25.05.1

Release Date: September 8, 2025

New Features

- Includes bug fixes for Optimizer Hub 25.05.0, and contains the new features and bug fixes also present in 1.11.4 and 1.11.5.
- Stability improvement: improved handling of OutOfMemory condition - management gateway and operator should restart immediately when OOM happens.
- Cache service stability improvement: IO operations offloaded to avoid blocking Hazelcast partition threads.
- Fix for an important regression in Quarkus:

  A regression in Vert.x introduced in 4.5.18 can lead to a pool HTTP client connection that does not have a correct state and stop making progress when receiving bytes, so the application will not observe the entirety of the HTTP response and therefore hang when receiving the data. It also means that clients of the library might not obtain a connection in a timely manner.

Bug Fixes

- Reduced possibility of unnecessary scale-up of compile-broker pods when many Zing JVMs with multiple different RNO-only profile names connect to OptHub.

CVE Fixes

CVE	Base Score	Component
CVE-2025-8671		

CVE	Base Score	Component
CVE-2025-55163	8.2	netty/quarkus

Optimizer Hub 25.05.0

Release Date: July 1, 2025

New Features

- This release switches to the numbering format `YY.MM` to align with the Azul Prime releases.
- Includes bug fixes for Optimizer Hub 1.11.2.
- More frequent gRPC keepalive messages are allowed to prevent time-outs reported on Azure deployments.
- By default, Helm chart creates a new service account for every Optimizer Hub component. With this release, you can use an existing service account which will be used for all components. Set the Helm value `deployment.serviceAccount.existingServiceAccount` to the name of the existing service account.

API Changes

See Optimizer Hub API for more info.

- New endpoints to get ReadyNow Orchestrator profile logs.
  - `/api/vmInstances/{id}/logs`: List logs produced by JVM instance
  - `/api/vmInstances/{id}/logs:export`: Export logs produced by JVM instance
- ReadyNow Orchestrator profile logs are now exported in one single file from the API `/api/profileNames/{profile}/profileLogs:export`.
- Info about profiles has been extended with the value `"isPromoted": true/false`.

Bug Fixes

- Because the management gateway is a required component, the setting to disable it, has been removed.
- Improved Optimizer Hub restarts with pre-populated storage to be able to serve profiles faster.

Optimizer Hub 1.11.6

Release Date: November 4, 2025

New Features

- Includes bug fixes for Optimizer Hub 1.11.5.
- A new metric `hz_operations_pending_invocations` exposes the number of pending invocations on the "cache" (Hazelcast) node. A high number can indicate a problem with the processing, and can be used to trigger alerts.

Bug Fixes

- The Optimizer Hub Operator can be configured to perform regular restarts of cache Kubernetes pods. This is disabled by default but can be enabled in case of stability issues in cache pods. Please contact our support team if you want to make use of this feature.
- A fix was applied for cases where the cache component of Optimizer Hub froze, causing disruption of service.

CVE Fixes

CVE	Base Score	Component
CVE-2025-58057	6.9	Netty
CVE-2025-58056	2.9	Netty
CVE-2025-48924	5.3	Apache Commons Lang
CVE-2025-27817	7.5	Apache Kafka Client

Optimizer Hub 1.11.5

Release Date: September 5, 2025

New Features

- Includes bug fixes for Optimizer Hub 1.11.4.
- Fix for an important regression in Quarkus:

  A regression in Vert.x introduced in 4.5.18 can lead to a pool HTTP client connection that does not have a correct state and stop making progress when receiving bytes, so the application will not observe the entirety of the HTTP response and therefore hang when receiving the data. It also means that clients of the library might not obtain a connection in a timely manner.

CVE Fixes

CVE	Base Score	Component
CVE-2025-8671		
CVE-2025-55163	8.2	netty/quarkus

Optimizer Hub 1.11.4

Release Date: August 18, 2025

New Features

- Includes bug fixes for Optimizer Hub 1.11.3.
- Stability improvement: improved handling of OutOfMemory condition - management gateway and operator should restart immediately when OOM happens.
- Cache service stability improvement: IO operations offloaded to avoid blocking Hazelcast partition threads.

Bug Fixes

- Reduced possibility of unnecessary scale-up of compile-broker pods when many Zing JVMs with multiple different RNO-only profile names connect to OptHub.

Optimizer Hub 1.11.3

Release Date: July 29, 2025

New Features

- Includes bug fixes for Optimizer Hub 1.11.2.
- Improvements in:
  - Tracking of JVM to Optimizer Hub connections.
  - Detection of stale connections.

Bug Fixes

- Fix for Grafana charts showing cache pods have unusually high CPU usage when a high number of compiler-broker and cache pods was active. Fixed by improving cache server overload checks.
- Fix for unexpected errors in the compile broker for gRPC calls that fail with code CANCELLED.

CVE Fixes

CVE	Base Score	Component
CVE-2025-4598	4.7	libsystemd0

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`) are still supported so 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 gcp.
- 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 configuring-s3-compatible-storage).

  See Storage Requirements 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 Standard Installation Procedure on Kubernetes

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 gw-proxy.
- Documentation has been extended with gcp.

Known Issues

Fixed Ports for gRPC

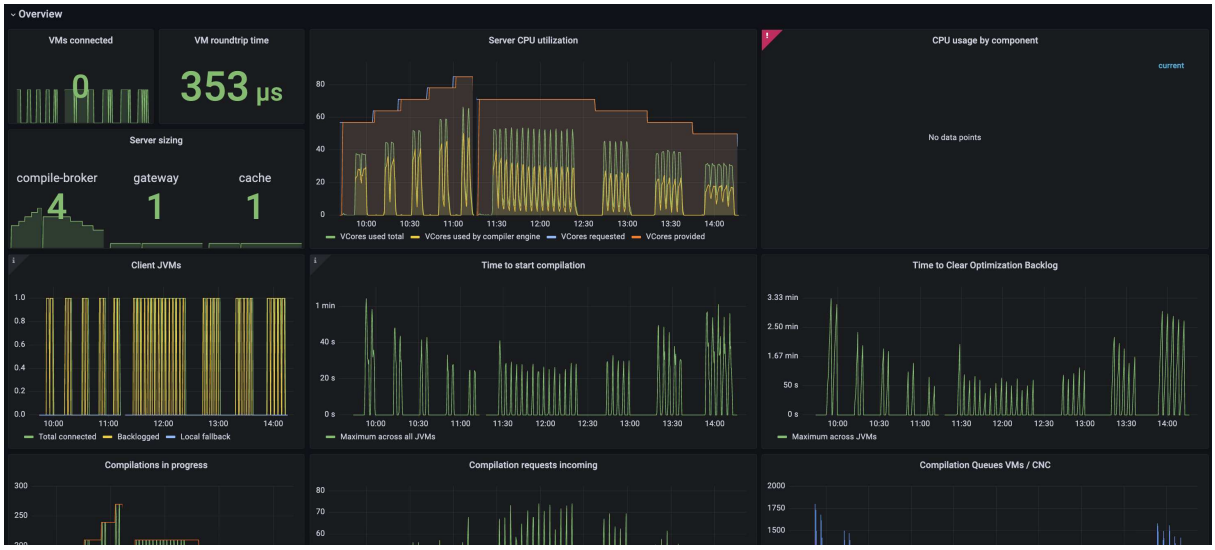
The helm chart values contain the keys `gateway.service.httpEndpoint.port` and `gateway.service.grpc.port` to change the default ports 50051 and 8080. But these values are hardcoded for the gRPC Envoy proxy, at this moment, and cannot be changed with the mentioned helm chart keys.

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 without Cloud Native Compiler.
- 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 Prime Optimizer Hub Third Party Licenses

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

Version	Optimizer Hub TPL
26.08.4	PDF
26.08.3	PDF
26.08.2	PDF
26.08.1	PDF
26.08.0	PDF
26.05.5	PDF
26.05.4	PDF
26.05.3	PDF
26.05.2	PDF
26.05.1	PDF
26.05.0	PDF

Version	Optimizer Hub TPL
26.02.1	PDF
26.02.0	PDF
25.11.1	PDF
25.11.0	PDF
25.08.1	PDF
25.08.0	PDF
25.05.2	PDF
25.05.1	PDF
25.05.0	PDF
1.11.6	PDF
1.11.5	PDF
1.11.4	PDF
1.11.3	PDF
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
1.9.0	PDF