Hyperscalers Run:ai Appliance

Reference Guide

Friday, 19 August 2022

Contents

1 Introduction 4

Featured Hardware from Hyperscalers 6

Hyperscalers Run:ai Appliance 7

Audience and Purpose 10

Digital IP Appliance Design Process 11

Appliance Optimizer Utility AOU 11

Features of our hardware 12

Important Considerations 14

2 Base Product Deployment 15

Preinstallation Requirements 15

Installation Components 15

Hardware Deployment 15

Software Deployment 17

Installation of operating system 17

Installation of Kubernetes 17

Prerequisites 17

Run on All Nodes 17

Permanently disable swap on all nodes 20

Avoiding Accidental Upgrades 21

Run:ai Software Prerequisites 21

Install the Run:ai Control Plane (Backend) 23

3 Configure the Appliance 24

4 Updating the Appliance 30

Prerequisites for updating 30

5 Testing the Appliance 31

6 Integration 39

7 Maintenance 45

8 Addendum 48

9 References 49

List Of Figures

Figure 1 Hyperscalers Run:ai Appliance stack 5
Figure 2 Hyperscalers Run:ai Appliance Workflow Architecture 8
Figure 3 Digital IP-Appliance Design Process 11
Figure 6 Front view of HSGP1 Figure 7 Top view and Internals of HSGP1 12
Figure 4 Front view of S5N Figure 5 Rear view and Internals of S5N 13
Figure 8 Front Internal view of GZ2 Figure 9 SXM GPUs of GZ2 13
Figure 10 Run:ai Atlas Lab as a Service Appliance Architecture from Hyperscalers (can be changed on customer's request) 16
Figure 11 Stages of Machine Learning Application development [15] 31
Figure 12 Assigning one complete GPU to a job 32
Figure 13 Deployment of the job with one complete GPU 32
Figure 14 Fractional GPU virtualization in Run:ai Atlas [15] 33
Figure 15 Allocating 10% of GPU to a job 34
Figure 16 Run:ai Atlas Dashboard training utilisation 35
Figure 17 Run:ai Atlas Dashboard interactive utilisation 35
Figure 18 Run:ai Atlas scheduler features [15] 36
Figure 19 Creating a training job for distributed training (multi-node) 37
Figure 20 Creating a training job for multi-GPU training 37
Figure 21 Run:ai Atlas Machine learning model deployment [15] 38
Figure 22 Autoscaling feature of Run:ai Atlas for any model deployment [15] 39
Figure 23 A sample screenshot to demonstrate clustered GPUs [15] 48

1 Introduction

Modern organisations expect rapid, efficient, and accurate results from their investments into AI technology. The ability to stand up appropriate models quickly can be paramount in terms of delivering critical insights responsively across key business and research dimensions.

AI infrastructure hardware is an expensive resource, yet most AI workload scheduling environments are not able to utilise all GPU resources optimally - meaning that unfortunately, AI hardware infrastructure must often be significantly over-provisioned.

The Hyperscalers Run:ai Appliance stands out against this backdrop due to its ability to perform fine-grained sub-allocation of GPU resources between multiple simultaneous workloads. In addition, the Hyperscalers Run:ai Appliance can dynamically re-allocate GPU resources against any workload while it is being processed. This may mean actively releasing GPU resources if they are no longer required or adding GPU resources whenever they become needed.

The dynamic partitioning capability of Run:ai moves beyond traditional methods in which GPU profile size must be pre-configured to a pre-set fixed value in advance. This older method often fails to match the profile sizing requirements of workloads whenever they are eventually scheduled, further reducing the ability to fully utilise GPU resources.

Inclusion of fine-grained, dynamic sub- allocation capabilities within the Hyperscalers Run:ai Appliance is an industry leading approach that automatically optimises utlisation of your valuable AI infrastructure. Coupled additionally with the powerful deployment context of NGC containers (and/or Helm charts), the Hyperscalers Run:ai Appliance is truly a game-changer in the field of agile AI platform enablement.

Hyperscalers [1] in partnership with Run:ai [2] is offering the HyperScalers Run:ai Appliance as a fully integrated and qualified appliance to quickly, reliably and efficiently support the needs of AI users using hardware supplied by Hyperscalers and software stack from Run:ai.

The following diagram illustrates the high-level relationship between Nvidia NGC, MLOps & Model Serving tools supported by Run:ai and additionally its ability to operate within the context of numerous well-known Kubernetes implementations using qualified, reliable state of the art equipment by Hyperscalers:

Nvidia NGC AI Model Library

Language Modelling
Recommender Systems
Image Segmentation
Translation
Object Detection
HPC

MLOps Tools
Kubeflow
Apache Airflow

Model Serving Tools
KServe
Seldon

Run:ai
run:ai

run:ai
AI Workload Orchestration

VMware Tanzu
OpenShift
Rancher
HPE Ezmeral
EKS
AKS
GKE

NVIDIA

HYPER SCALERS

Figure 1 Hyperscalers Run:ai Appliance stack

Featured Hardware from Hyperscalers

Hyperscalers have identified the hardware configurations that can be categorised based on the customer use cases as the below. Run:ai enterprise appliance can be deployed in any of these high performance ultra-dense GPU servers.

1. Cost efficient (Up to 4 x PCIe 80GB GPUs)
2. Balanced performance (Up to 4 x PCIe 80GB GPUs)
3. High Performance Compute (Up to 8 x SXM 80GB GPUs)

High Performance Compute (Up to 8 x SXM 80GB GPUs)

Balanced performance solution← (Up to 4 x PCIe 80GB GPUs)→ Cost efficient solution

Hyperscalers Run:ai Appliance

Assign the Right Amount of AI Compute Power to Users, Automatically

The Hyperscalers Run:ai Appliance is a Kubernetes-based software platform for orchestration of containerized AI workloads that enables GPU clusters to be utilized for different Deep Learning workloads dynamically - from building AI models, to training, to inference. With Run:ai, jobs at any stage can obtain access to the compute power they need, automatically [3].

Run:ai's compute management platform speeds up data science initiatives by pooling available resources and then dynamically allocating resources optimally as needed. These powerful capabilities maximise AI compute power utilisation and therefore return on investment for your organisation.

Key Features

- Fair-share scheduling to allow users to share clusters of GPUs easily and automatically
- Fractional GPU allocation for interactive/ training workloads
- Simplified workflows for building, training (including multi-GPU and distributed training) and deployment of AI models
- Visibility into workloads and resource utilization to improve user productivity
- Control for cluster admin and ops teams, to align priorities to business goals
- On-demand access to Multi-Instance GPU (MIG) instances for the A100 GPU

Key Benefits

Advanced Kubernetes-based Scheduling Eliminates Static GPU Allocation

The *Run:ai Scheduler* manages tasks in batches using multiple queues on top of Kubernetes, allowing system admins to define different rules, policies, and requirements for each queue based on business priorities. Combined with an over-quota system and configurable fairness policies, the allocation of resources can be automated and optimized to allow maximum utilization of cluster resources.

Because it was built as a plug-in to K8s, Run:ai's scheduler requires no advanced setup, and is certified to integrate with any number of Kubernetes "flavors" including Red Hat OpenShift and HPE Ezmeral.

The following diagram illustrates key architecture and functional elements of the fully integrated Hyperscalers Run:ai Appliance:

Figure 2 Hyperscalers Run:ai Appliance Workflow Architecture

The following illustration shows Run:ai management dashboard monitoring UI capabilities supporting both high-level overview and detailed observation of AI workload status and resource utilisation:

run:
ai

Figure 3 Overview of the Hyperscalers Run:ai Management Dashboard

No More Idle Resources

Run:ai's over-quota system allows users to automatically access idle resources when available based on configurable fairness policies. The platform allocates resources dynamically, for full utilization of cluster resources. Our customers see improvements in utilization from around 25% from when we start working with them to over 75% once more fully optimised.

Bridge Between HPC and AI

The Run:ai Scheduler allows users to easily make use of integer GPUs, multi-node/multi GPUs, and even GPU Multi-Instance GPU (MIG) instances for distributed training on Kubernetes. In this way, AI workloads run based on needs, not available capacity. Run:ai empowers you to combine the benefits and efficiency of High-Performance Computing with the simplicity of Kubernetes.

Accelerate AI

By using Run:ai resource pooling, queueing, and prioritization mechanisms, researchers are shielded from infrastructure management hassles and can focus exclusively on data science. Many workloads can be run in parallel without compute bottlenecks. Run:ai delivers real time and historical views on all resources managed by the platform, such as jobs, deployments, projects, users, GPUs and clusters.

The Hyperscalers Run:ai Appliance can accelerate your time to reach productive AI results, in particular as it is delivered as a turn-key solution including all master, worker and storage nodes pre-configured and ready to start running your AI workloads.

Streamline AI

Run:ai can support all types of workloads required within the AI lifecycle (build, train, inference) to easily start experiments, run large-scale training jobs and take AI models to production without ever worrying about the underlying infrastructure. The Run:ai Atlas platform allows MLOps and AI Engineering teams to quickly operationalize AI pipelines at scale and run production machine learning models anywhere while using the built-in ML toolset or simply integrating their existing 3rd party toolset.

Productize AI

Run:ai's unique GPU Abstraction capabilities effectively "virtualize" all available GPU resources to maximize infrastructure efficiency and increase ROI. The platform pools expensive compute resources and makes them accessible to researchers on-demand for a simplified, cloud-like experience.

Audience and Purpose

Engineers, Enthusiasts, Executives and IT professionals with background in Computer Science/ Electronics/ Information Technology with understanding in Linux commands, Python language and basic electronics who intend to study, explore, deploy the

Hyperscalers Run:ai Appliance with NVIDIA A100 and A6000 (or other qualified GPUs in featured hardware) on Ubuntu 20.04.

The purpose of this document is to create a Hyperscalers Run:ai Appliance with NVIDIA A100 and A6000 as GPUs with QuantaGrid D43N-3U server (worker node) using the Ubuntu 20.04 operating system.

Digital IP Appliance Design Process

Hyperscalers has developed a Digital- IP-Appliance Design Process and associated Appliance Optimizer Utility which can enable the productization of IT-appliances for Digital-IP owners needing to hyperscale their services very quickly, reliably and at a fraction of traditional costs.

Appliance Optimizer Utility AOU

The Appliance Optimizer Utility (AOU) automates the discovery of appliance bottlenecks by pinging all layers in the proposed solution stack. A live dashboard unifies all key performance characteristics to provide a head-to-head performance assessment between all data-path layers in the appliance, as well as a comparison between holistic appliances.

Figure 3 Digital IP-Appliance Design Process

Features of our hardware

Cost efficient Run:ai Appliance configuration

Our *cost efficient* solution with HSGP1 [4] [5] (TN83B8251) supports two AMD EPYC™ 7003 top bin CPUs up to 280W each and 16x DIMM slots, which is powerful enough to feed four (4) double width, high throughput GPU cards such as NVIDIA: A100, A40 GPU, or eight (8) single width GPU such as NVIDIA T4 and A4000. The HSGP1 can deliver up to 45.6 Tera FLOPS HPC performance, which makes the HSGP1 a Hyperscale TCO-optimized solution for service providers running Artificial Intelligence AI, Machine Learning, High Performance Computing HPC and graphical applications in molecular dynamics, life sciences and many more fields.

Figure 4 Front view of HSGP1

Figure 5 Top view and Internals of HSGP1

Balanced Performance Run:ai Appliance configuration

Our *balanced performance* solution with S5N [6](D43N-3U) can support two AMD EPYC™ 7003 top bin CPUs up to 280W and 32x DIMM slots, which is powerful enough to feed four high throughput GPU cards such as NVIDIA A100 Tensor Core GPU. It can deliver up to 45.6 Tera Flops HPL performance, which makes D43N-3U a TCO-optimized solution for data centres running HPC applications in molecular dynamics, life sciences and more fields.

Figure 6 Front view of S5N

Figure 7 Rear view and Internals of S5N

High Performance Compute Run:ai Appliance configuration

Our *High Performance Compute (HPC)* solution with GZ2 [7] [8](G492-ZD2) based on PCIe Gen 4.0, NVIDIA HGX GPU, and AMD EPYC processors offers: Eight (8) A100 40GB and 80GB SXM GPU, two (2) CPU Sockets for up to 120 cores using AMD Milan (7003) processors 60 cores each, 32 Memory slots, six (6) front Storage drive bays 2.5" hot-plug U.2 NVMe or SATA SSD and two (2) M.2 onboard storage in 4RU.

Figure 8 Front Internal view of GZ2

Figure 9 SXM GPUs of GZ2

Important Considerations

This appliance documentation is qualified and valid only for the featured hardware and software configuration.

2 Base Product Deployment

Preinstallation Requirements

Hardware and Software requirements

(Production only) Run:ai System Nodes: To reduce downtime and save CPU cycles on expensive GPU Machines we recommend that production deployments will contain two or more worker machines, designated for Run:ai Software. The nodes do not have to be dedicated to Run:ai, but for Run:ai purposes we would need:

- 4 CPUs
- 8GB of RAM
- 120GB of Disk space

Worker machines with CUDA enabled GPUs

The control plane (backend) installation of Run:ai will require the configuration of Kubernetes Persistent Volumes of a total size of 110GB.

An active DNS Server is needed to deploy Run:ai Atlas.

Installation Components

Hardware Deployment

Run:ai Atlas appliance will have a minimum of two nodes with one master and one worker (GPU Node). The following diagram shows the architecture of our LaaS appliance and its hardware configuration.

Data Network

Container Applications
NVIDIA. NGC

RunAI + Kubernetes

S5SP (Master)

S5N (Worker)

DNS Server

Management Network

Figure 10 Run:ai Atlas Lab as a Service Appliance Architecture from Hyperscalers (can be changed on customer's request)

Server	Nodes	Designation	CPU	RAM	Mezzanine NIC	GPU
S5SP	1	master	Intel 4210 [9]x 2	32 GB 2400MT/s x 4	ConnectX-4 25Gb	N/A
S5N-3U	1	worker	AMD 7543 [10]x 2	32 GB 3200MT/s x 8	ConnectX-6 25Gb	NVIDIA A6000 [11], A100 [12]

Table 1 Run:ai Atlas Lab as a Service Appliance from Hyperscalers (can be changed on customer's request)

Software Deployment

Installation of operating system

We will begin by installing with Ubuntu 20.04 Server [13] in master and worker nodes. While installing Ubuntu 20.04 Server, ensure that "Static IP/ DNS server" options are set up. This document will follow air gapped installation of Run:ai Atlas.

Installation of Kubernetes

Kubernetes is composed of master(s) and workers. The instructions below are for creating a bare-bones installation of a single master and several workers for testing purposes. For a more complex, production-grade, Kubernetes installation, use tools such as Kubespray https://kubespray.io/, Rancher Kubernetes Engine, or review Kubernetes documentation to learn how to customize the native installation [14].

Prerequisites

The script below assumes all machines have Ubuntu 20.04.

Run on All Nodes

If not yet installed, install docker by performing the following commands.

```
curl -fsSL https://get.docker.com -o get-docker.sh
sudo sh get-docker.sh
```

Change Docker to use systemd by editing `/etc/docker/daemon.json` and adding:

```
/etc/docker/daemon.json

{
  "exec-opts": ["native.cgroupdriver=systemd"]
}
```

Restart the docker service

```
sudo systemctl restart docker
```

Run on Master Node

Install Kubernetes master:

```
sudo sh -c 'cat <<EOF >  /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
net.ipv4.ip_forward = 1
```

```
EOF'

sudo apt-get update && sudo apt-get install -y apt-transport-https curl
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -
cat <<EOF | sudo tee /etc/apt/sources.list.d/kubernetes.list
deb https://apt.kubernetes.io/ kubernetes-xenial main
EOF

sudo apt-get update
sudo apt-get install -y kubelet=1.23.5-00 kubeadm=1.23.5-00 kubectl=1.23.5-00

sudo swapoff -a
sudo kubeadm init --pod-network-cidr=10.244.0.0/16 --kubernetes-version=v1.23.5 --token-ttl 180h
```

The `kubeadm init` command above has emitted as output a `kubeadm join` command. Save it for joining the workers below.

Copy the Kubernetes configuration file which provides access to the cluster:

```
mkdir .kube
sudo cp -i /etc/kubernetes/admin.conf .kube/config
sudo chown $(id -u):$(id -g) .kube/config
```

Add Kubernetes networking:

```
kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-
flannel.yml
```

Test that Kubernetes is up and running:

```
kubectl get nodes
```

Verify that the master node is ready

Run on Kubernetes Workers

For Kubernetes workers with GPU, you must install the NVIDIA prerequisites. We recommend using the NVIDIA GPU Operator on top of Kubernetes.Install Kubernetes worker (any machine):

```
sudo sh -c 'cat <<EOF >  /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
net.ipv4.ip_forward = 1
EOF'

sudo apt-get update && sudo apt-get install -y apt-transport-https curl
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -
cat <<EOF | sudo tee /etc/apt/sources.list.d/kubernetes.list
deb https://apt.kubernetes.io/ kubernetes-xenial main
EOF
sudo apt-get update
sudo apt-get install -y kubelet=1.23.5-00 kubeadm=1.23.5-00

sudo swapoff -a
```

Replace the following `join` command with the one saved from the init command above:

```
sudo kubeadm join 10.0.0.3:6443 --token <token> \
    --discovery-token-ca-cert-hash sha256:<hash>
```

Note *The default token expires after 24 hours. If the token has expired, go to the master node and run sudo kubeadm token create --print-join-command. This will produce an up-to-date join command.*

Return to the master node. Re-run `kubectl get nodes` and verify that the new node is ready.

NVIDIA GPU Operator

Install the operator after Kubernetes is installed.

Prerequisites

Before installing the GPU Operator, you should ensure that the Kubernetes cluster meets some prerequisites.

1. Nodes must be configured with a container engine such as Docker CE/EE, cri-o, or containerd. For docker, follow the official install instructions.
2. Node Feature Discovery (NFD) is a dependency for the Operator on each node. By default, NFD master and worker are automatically deployed by the Operator. If NFD is already running in the cluster prior to the deployment of the operator, then the Operator can be configured to not to install NFD.
3. For monitoring in Kubernetes 1.13 and 1.14, enable the kubelet KubeletPodResources feature gate. From Kubernetes 1.15 onwards, its enabled by default.

Install Helm

The preferred method to deploy the GPU Operator is using helm.

```
curl -fsSL -o get_helm.sh https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3 \
   && chmod 700 get_helm.sh \
   && ./get_helm.sh
```

Now, add the NVIDIA Helm repository:

```
helm repo add nvidia https://helm.ngc.nvidia.com/nvidia \
   && helm repo update
```

Install the GPU Operator

The GPU Operator Helm chart offers a number of customizable options that can be configured depending on your environment.

Namespace

Starting with GPU Operator v1.9, both the operator and operands get installed in the same namespace. The namespace is configurable and is determined during installation. For example, to install the GPU Operator in the gpu-operator namespace:

```
helm install --wait --generate-name \
     -n gpu-operator --create-namespace \
     nvidia/gpu-operator
```

If a namespace is not specified during installation, all GPU Operator components will be installed in the default namespace.

Operands

By default, the GPU Operator operands are deployed on all GPU worker nodes in the cluster. GPU worker nodes are identified by the presence of the label feature.node.kubernetes.io/pci-10de.present=true, where 0x10de is the PCI vendor ID assigned to NVIDIA.

To disable operands from getting deployed on a GPU worker node, label the node with nvidia.com/gpu.deploy.operands=false. This can be useful when dedicating a GPU worker node for non-container workloads (i.e. KubeVirt VMs).

```
kubectl label nodes $NODE nvidia.com/gpu.deploy.operands=false
```

Bare metal/Passthrough with default configurations on Ubuntu

In this POC, the default configuration options are used:

```
helm install --wait --generate-name \
     -n gpu-operator --create-namespace \
     nvidia/gpu-operator
```

Permanently disable swap on all nodes

1. Edit the file /etc/fstab
2. Comment out any swap entry if such exists

Avoiding Accidental Upgrades

To avoid accidental upgrade of Kubernetes binaries, it is recommended to hold the version. Run the following on all nodes:

```
sudo apt-mark hold kubeadm kubelet kubectl
```

Run:ai Software Prerequisites

Air gapped deployment

You should receive a single file Run:ai-<version>.tar from Run:ai customer support

Kubernetes

Run:ai requires Kubernetes. Supported versions are 1.19 through 1.24.

Network

1. Shared Storage. Network address and a path to a folder in a Network File System
2. All Kubernetes cluster nodes should be able to mount NFS folders. Usually, this requires the installation of the nfs-common package on all machines (sudo apt install nfs-common or similar)
3. IP Address. An available, internal IP Address that is accessible from Run:ai Users' machines (referenced below as <RUN:AI_IP_ADDRESS>)
4. DNS entry Create a DNS A record such as Run:ai.<company-name> or similar. The A record should point to <RUN:AI_IP_ADDRESS>
5. A certificate for the endpoint. The certificate(s) must be signed by the organization's root CA.

Other

Private Docker Registry. For air gapped installation, Run:ai assumes the existence of a Docker registry for images. Most likely installed within the organization. The installation requires the network address and port for the registry (referenced below as <REGISTRY_URL>)

Pre-install Script

Once you believe that the Run:ai prerequisites are met, we highly recommend installing and running the Run:ai pre-install diagnostics script. The tool:

```
chmod +x preinstall-diagnostics-<platform>
./preinstall-diagnostics-<platform> --domain <dns-entry>
```

- Tests the below requirements as well as additional failure points related to Kubernetes, NVIDIA, storage, and networking.
- Looks at additional components installed and analyzes their relevancy to a successful Run:ai installation.

To use the script, download the latest version of the script and run:

```
chmod +x preinstall-diagnostics-<platform>
./preinstall-diagnostics-<platform> --domain <dns-entry>
```

If the script fails, or if the script succeeds but the Kubernetes system contains components other than Run:ai, locate the file Run:ai-preinstall-diagnostics.txt in the current directory and send it to Run:ai technical support.

Preperations

Prepare Installation Artifacts

Run:ai Software Files

SSH into a node with kubectl access to the cluster and Docker installed.

To extract Run:ai files, replace <VERSION> in the command below and run:

```
tar xvf Run:ai-<version>.tar.gz
cd deploy
```

Upload images to Docker Registry. Set the Docker Registry address in the form of NAME:PORT (do not add https):

```
export REGISTRY_URL=<Docker Registry address>
```

Run the following script (you must have at least 20GB of free disk space to run):

```
kubectl create namespace Run:ai-backend
sudo -E ./prepare_installation.sh
```

(If docker is configured to run as non-root then sudo is not required).

Run:ai Administration CLI

Install the Run:ai Administrator Command-line Interface by following the steps. Use the image under `deploy/Run:ai-admin-cli-<version>-linux-amd64.tar.gz`

Download the Run:ai Administrator Command-line Interface by running:

```
wget --content-disposition https://app.run.ai/v1/k8s/admin-cli/linux
chmod +x Run:ai-adm
sudo mv Run:ai-adm /usr/local/bin/Run:ai-adm
```

To verify the installation run:

```
Run:ai-adm version
```

Mark Run:ai System Workers

The Run:ai control plane (backend) should be installed on a set of dedicated Run:ai system worker nodes rather than GPU worker nodes. To set system worker nodes run:

```
kubectl label node <NODE-NAME> node-role.kubernetes.io/Run:ai-system=true
```

To avoid single-point-of-failure issues, we recommend assigning more than one node in production environments.

Install the Run:ai Control Plane (Backend)

Create a Control Plane Configuration

Create a configuration file to install the Run:ai control plane:

Generate a values file by running the following under the deploy folder:

```
Run:ai-adm generate-values
    --external-ips <ip> \ #
    --domain <dns-record> \ #
    --tls-cert <file-name>  --tls-key <file-name> \ #
    --nfs-server <nfs-server-address> --nfs-path <path-in-nfs> \ #
    --airgapped
```

Note *In cloud environments, the flag --external-ips should contain both the internal and external IPs (comma separated)*

A file called Run:ai-backend-values.yaml will be created.

Install the Control Plane (Backend)

Run the helm command below:

```
helm install Run:ai-backend Run:ai-backend-<version>.tgz -n \
    Run:ai-backend -f Run:ai-backend-values.yaml
```

(replace <version> with the Run:ai control plane version)

Connect to Run:ai User Interface

Go to: `Run:ai.<company-name>`. Log in using the default credentials: User: `test@run.ai`, Password: `password`

3 Configure the Appliance

Designating Specific Role Nodes

When installing a production cluster, you may want to:

- Set one or more Run:ai system nodes. These are nodes dedicated to Run:ai software.
- Machine learning frequently requires jobs that require CPU but not GPU. You may want to direct these jobs to dedicated nodes that do not have GPUs, so as not to overload these machines.
- Limit Run:ai monitoring and scheduling to specific nodes in the cluster.

To perform these tasks, you will need the Run:ai Administrator CLI. See Installing the Run:ai Administrator Command-line Interface.

Dedicated Run:ai System Nodes

Find out the names of the nodes designated for the Run:ai system by running kubectl get nodes. For each such node run:

```
Run:ai-adm set node-role --Run:ai-system-worker <node-name>
```

If you re-run kubectl get nodes you will see the node role of these nodes changed to Run:ai-system

To remove the Run:ai-system node role run:

```
Run:ai-adm remove node-role --Run:ai-system-worker <node-name>
```

Dedicated GPU & CPU Nodes

Separate nodes into those that:

- Run GPU workloads
- Run CPU workloads
- Do not run Run:ai at all. these jobs will not be monitored using the Run:ai Administration User interface.

Review nodes names using kubectl get nodes. For each such node run:

```
Run:ai-adm set node-role --gpu-worker <node-name>
```

or

```
Run:ai-adm set node-role --cpu-worker <node-name>
```

Nodes not marked as GPU worker or CPU worker will not run Run:ai at all.

To set all workers not running Run:ai-system as GPU workers run:

```
Run:ai-adm set node-role --all <node-name>
```

To remove the CPU or GPU worker node role run:

```
Run:ai-adm remove node-role --cpu-worker <node-name>
```

or

```
Run:ai-adm remove node-role --gpu-worker <node-name>
```

Install the Run:ai Command-line Interface

The Run:ai Command-line Interface (CLI) is one of the ways for a Researcher to send deep learning workloads, acquire GPU-based containers, list jobs, etc.

The instructions below will guide you through the process of installing the CLI. The Run:ai CLI runs on Mac and Linux. You can run the CLI on Windows by using Docker for Windows.

Researcher Authentication

When enabled, Researcher authentication requires additional setup when installing the CLI. To configure authentication, see Setup Project-based Researcher Access Control. Use the modified Kubernetes configuration file described in the article.

Prerequisites

When installing the command-line interface, it is worth considering future upgrades:

- Install the CLI on a dedicated Jumpbox machine. Researchers will connect to the Jumpbox from which they can submit Run:ai commands
- Install the CLI on a shared directory that is mounted on Researchers' machines.
- A Kubernetes configuration file obtained from the Kubernetes cluster installation.

Setup

Kubernetes Configuration

- On the Researcher's root folder, create a directory `.kube`. Copy the Kubernetes configuration file into the directory. Each Researcher should have a separate copy of the configuration file. The Researcher should have write access to the configuration file as it stores user defaults.
- If you choose to locate the file at a different location than `~/.kube/config`, you must create a shell variable to point to the configuration file as follows:

```
export KUBECONFIG=<Kubernetes-config-file>
```

Test the connection by running:

```
kubectl get nodes
```

Install Run:ai CLI

- Go to the Run:ai user interface. On the top right select Researcher Command Line Interface.
- Select Mac or Linux.
- Download directly using the button or copy the command and run on a remote machine
- Run:

```
chmod +x Run:ai
```

```
sudo mv Run:ai /usr/local/bin/Run:ai
```

To verify the installation run:

```
Run:ai list jobs
```

Install Command Auto-Completion

It is possible to configure your Linux/Mac shell to complete Run:ai CLI commands. This feature works on bash and zsh shells only.

Zsh

Edit the file `~/.zshrc`. Add the lines:

```
autoload -U compinit; compinit -i
source <(Run:ai completion zsh)
```

Bash

Install the bash-completion package:

- Mac: brew install bash-completion
- Ubuntu/Debian: sudo apt-get install bash-completion
- Fedora/Centos: sudo yum install bash-completion

Edit the file `~/.bashrc`. Add the lines:

```
[[ -r “/usr/local/etc/profile.d/bash_completion.sh” ]] && .
“/usr/local/etc/profile.d/bash_completion.sh”
source <(Run:ai completion bash)
```

Use Run:ai on Windows

- Install Docker for Windows.
- Get the following folder from GitHub: `https://github.com/run-ai/docs/tree/master/cli/windows`.
- Replace config with your Kubernetes Configuration file.
- Run: `build.sh` to create a docker image named `Run:ai-cli`.
- Test the image by running:

```
docker run -it Run:ai-cli bash
```

- Try and connect to your cluster from inside the docker by running a Run:ai CLI command. E.g. `Run:ai list projects`

- Distribute the image to Windows users.
    - In case you want to use port-forward feature please use the following command

```
docker run -it -p <PORT>:<PORT> Run:ai-cli bash
```

- And when using `Run:ai submit` command add the following flag:

```
--address 0.0.0.0
```

Setup Researcher Access Control

Introduction

The following instructions explain how to complete the configuration of access control for Researchers. Run:ai access control is at the Project level. When you assign Users to Projects - only these users are allowed to submit Jobs and access Jobs details.

This requires several steps:

- Assign users to their Projects
- (Mandatory) Modify the Kubernetes entry point (called the Kubernetes API server) to validate credentials of incoming requests against the Run:ai Authentication authority.
- (Command-line Interface usage only) Modify the Kubernetes profile to prompt the Researcher for credentials when running Run:ai login (or oc login for OpenShift).

Administration User Interface Setup

Enable Researcher Authentication

- Open the Run:ai user interface and navigate to General | Settings.
- Enable the flag Researcher Authentication (should be enabled by default for new tenants).
- There are values for Realm, client configuration, and server configuration which appear on the screen. Use them as below.

Assign Users to Projects

Assign Researchers to Projects:

- Open the Run:ai user interface and navigate to Users. Add a Researcher and assign it with a Researcher role.

- Navigate to Projects. Edit or create a Project. Use the Access Control tab to assign the Researcher to the Project.
- If you are using Single Sign-on, you can also assign Groups. For more information see the Single Sign-on documentation.

(Mandatory) Kubernetes Configuration

One must direct the Kubernetes API server to authenticate via Run:ai. This requires adding flags to the Kubernetes API Server. Modfiying the API Server configuration differs between Kubernetes distributions, For native Kubernetes,

- Locate the Kubernetes API Server configuration file. The file's location may defer between different Kubernetes distributions. The location for vanilla Kubernetes is /etc/kubernetes/manifests/kube-apiserver.yaml
- Edit the document, under the command tag, add the server configuration text from General | Settings | Researcher Authentication described above.
- Verify that the kube-apiserver-<master-node-name> pod in the kube-system namespace has been restarted and that changes have been incorporated. Run the below and verify that the oidc flags you have added:

```
kubectl get pods -n kube-system kube-apiserver-<master-node-name> -o yaml
```

Command-line Interface Access

To control access to Run:ai (and Kubernetes) resources, you must modify the Kubernetes configuration file. The file is distributed to users as part of the Command-line interface installation.

When making changes to the file, keep a copy of the original file to be used for cluster administration. After making the modifications, distribute the modified file to Researchers.

- Under the `~/.kube` directory edit the `config` file, remove the administrative user, and replace it with the client configuration text from `General | Settings | Researcher Authentication` described above.
- Under `contexts | context | user` change the user to `Run:ai-authenticated-user`

Test via Command-line interface

- Run: `Run:ai login`

- You will be prompted for a username and password. In a single sign-on flow, you will be asked to copy a link to a browser, log in and return a code.
- Once login is successful, submit a Job.
- If the Job was submitted with a Project for which you have no access, your access will be denied.
- If the Job was submitted with a Project for which you have access, your access will be granted.

You can also submit a Job from the Run:ai User interface and verify that the new job shows on the job list with your user name.

Test via User Interface

- Open the Run:ai user interface. Go to Jobs
- On the top-right, select Submit Job.

4 Updating the Appliance

Prerequisites for updating

Ask for a tar file Run:ai-air-gapped-<new-version>.tar from Run:ai customer support. The file contains the new version you want to upgrade to. new-version is the updated version of the Run:ai backend.

Prepare the installation artifact as described earlier (untar the file and run the script to upload it to the local container registry).

Upgrade Control Plane (Backend)

Run the helm command below.

```
kubectl delete deployment -n Run:ai-backend Run:ai-backend-nginx-ingress-controller
helm get values Run:ai-backend -n Run:ai-backend > be-values.yaml
helm upgrade Run:ai-backend Run:ai-backend/Run:ai-backend-<version>.tgz -n \
    Run:ai-backend  -f be-values.yaml
```

(replace <version> with the control plane (backend) version)

Upgrade Cluster

To upgrade the cluster, follow the instructions here.

```
helm get values Run:ai-cluster -n Run:ai > values.yaml
helm upgrade Run:ai-cluster -n Run:ai Run:ai-cluster-<version>.tgz -f values.yaml
```

(replace <version> with the cluster version)

5 Testing the Appliance

Run:ai Atlas appliance offers researchers to build, train and infer from their machine learning application development. The following image shows the stages of the development and their key steps involved and resource allocation. Run:ai Atlas also offers fixed quota, over provisioning of quota (if available/idle in the cluster) through its scheduler to researchers to use the cluster resources anytime, anywhere.

Build	Train	Inference
Development & debugging	Training & Hyperparameter Optimization	Model serving in real time
Interactive sessions	Remote execution	Long-running services
Short cycles	Long cycles	Short compute bursts
Throughput is less important	Throughput is highly important	Latency and throughput is important
Low GPU utilization	High GPU utilization	Low GPU utilization

Figure 11 Stages of Machine Learning Application development [15]

Build phase

In the build phase of the development, one can have a live interactive environment (remote / local) with the job created with necessary GPU resources.

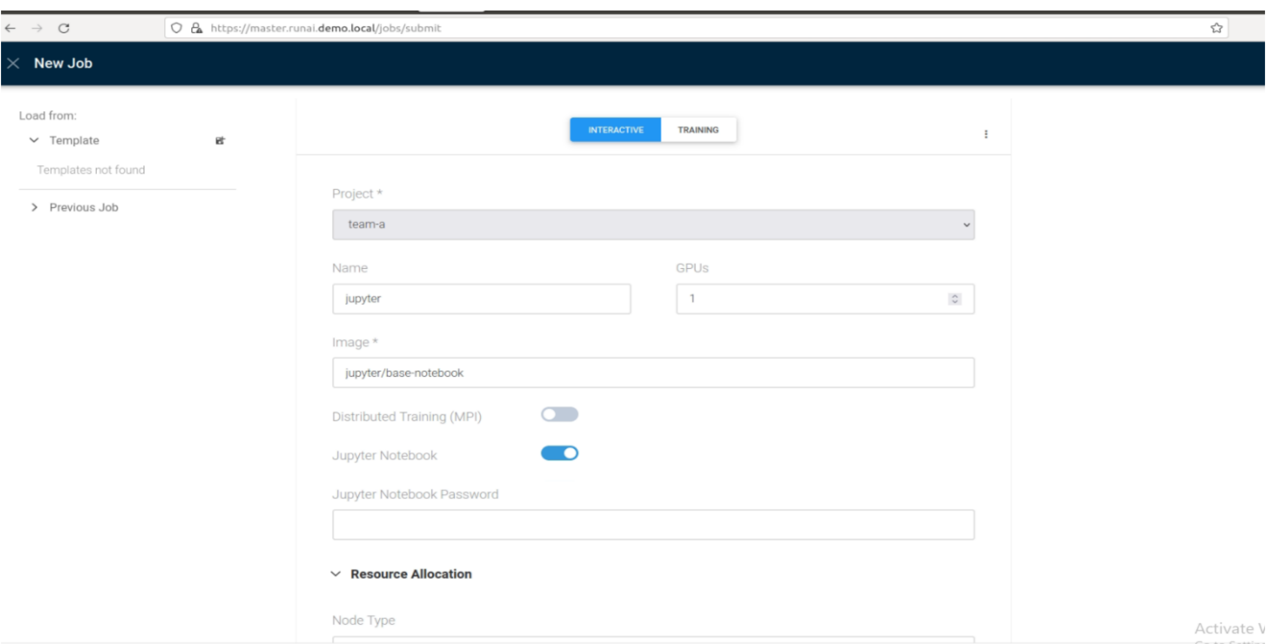


Figure 12 Assigning one complete GPU to a job

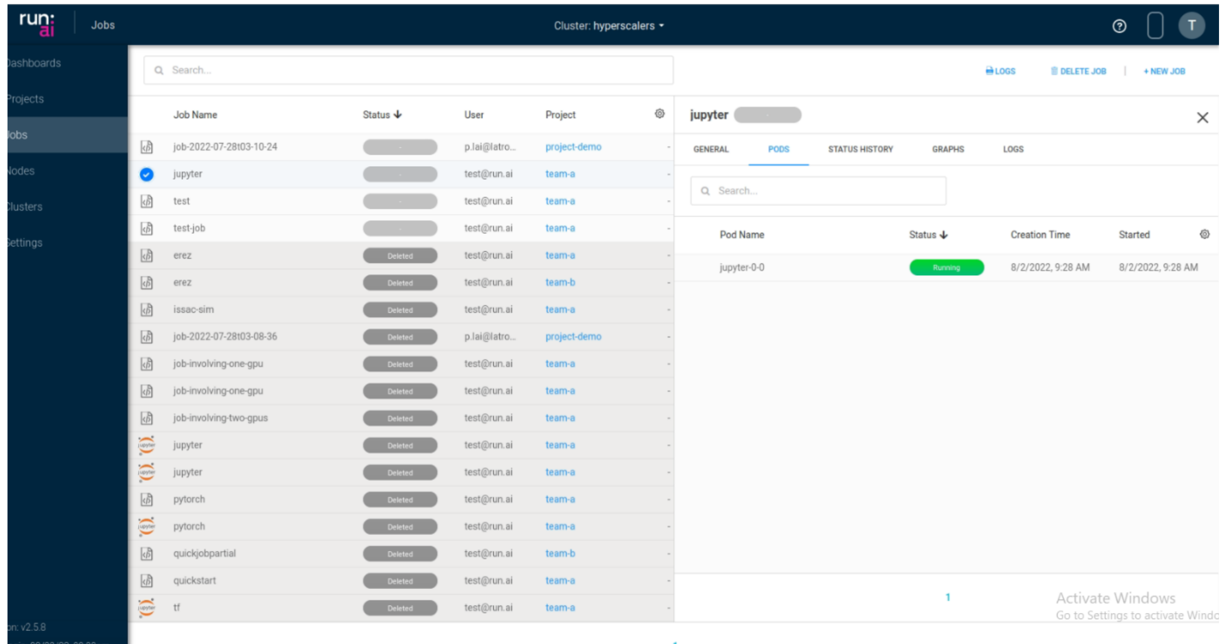
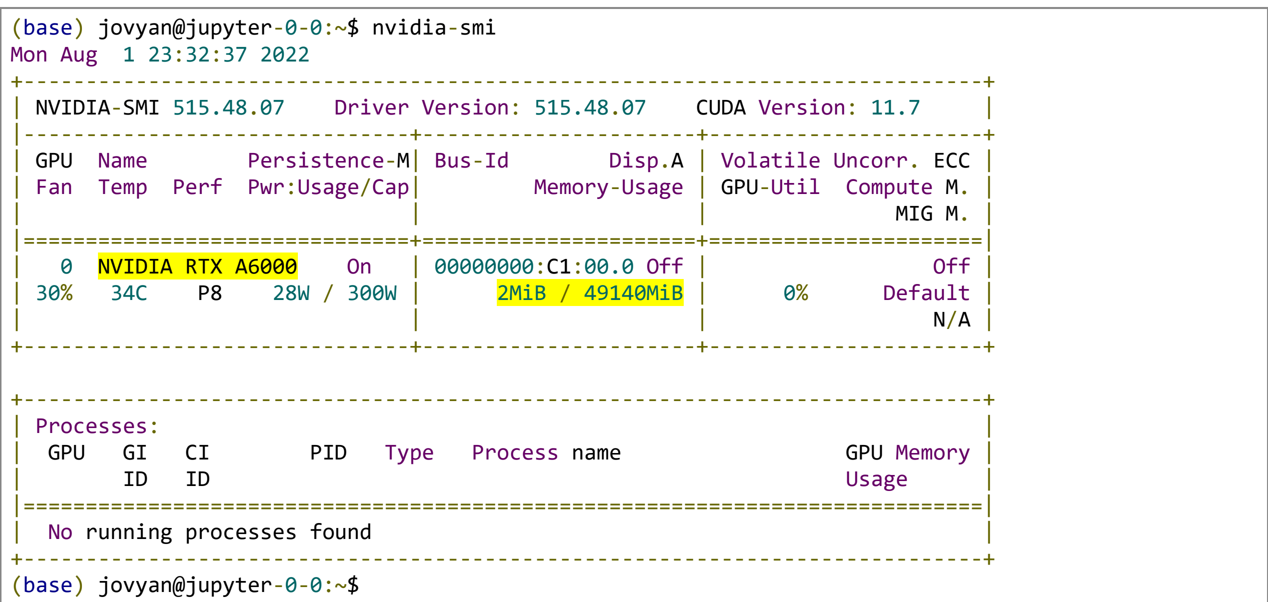


Figure 13 Deployment of the job with one complete GPU



```
(base) jovyan@jupyter-0-0:~$ nvidia-smi
Mon Aug  1 23:32:37 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 515.48.07    Driver Version: 515.48.07    CUDA Version: 11.7     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA RTX A6000     On  | 00000000:C1:00.0 Off |                  Off |
| 30%   34C    P8    28W / 300W |      2MiB / 49140MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
(base) jovyan@jupyter-0-0:~$
```

Fractional GPUs

One of the key features of Run:ai Atlas is to partition a single GPU (CUDA enabled) in fractions to improve the utilization of resources.

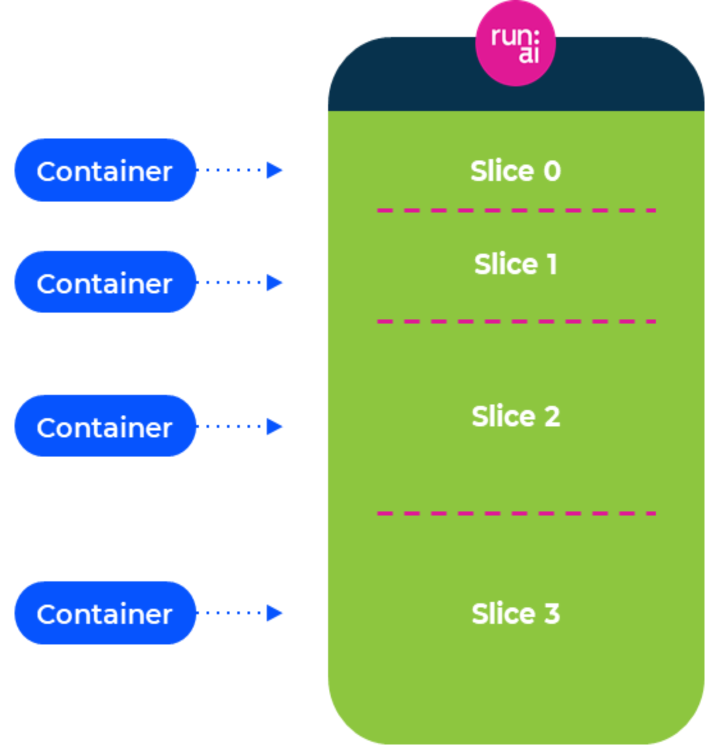


Figure 14 Fractional GPU virtualization in Run:ai Atlas [15]

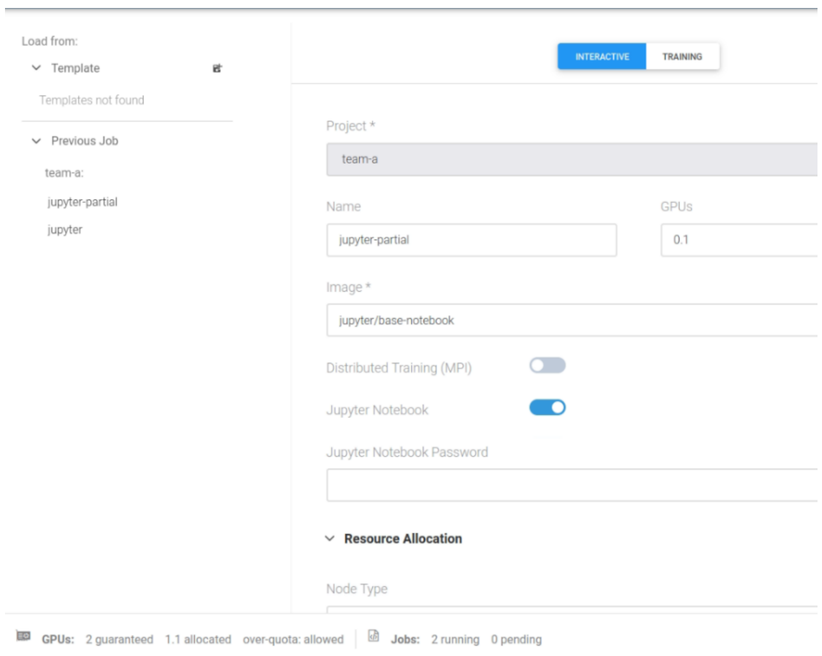
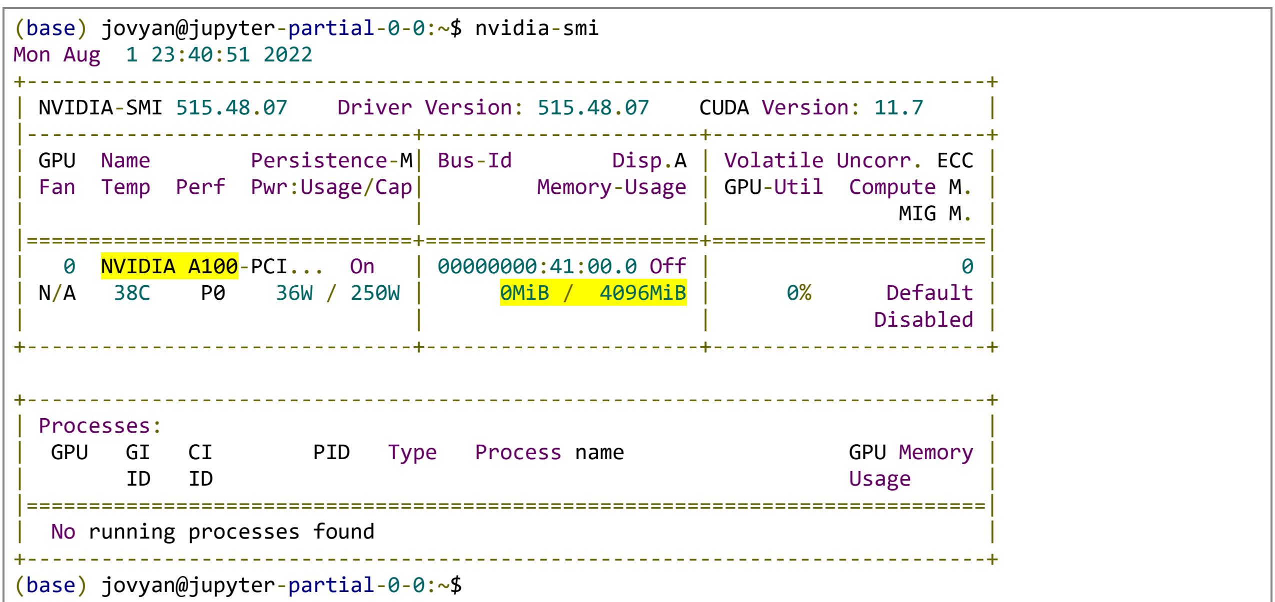


Figure 15 Allocating 10% of GPU to a job



```
(base) jovyan@jupyter-partial-0-0:~$ nvidia-smi
Mon Aug  1 23:40:51 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 515.48.07    Driver Version: 515.48.07    CUDA Version: 11.7     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA A100-PCI...  On   | 00000000:41:00.0 Off |                    0 |
| N/A   38C    P0    36W / 250W |      0MiB /  4096MiB |      0%      Default |
|                               |                      |             Disabled |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
(base) jovyan@jupyter-partial-0-0:~$
```

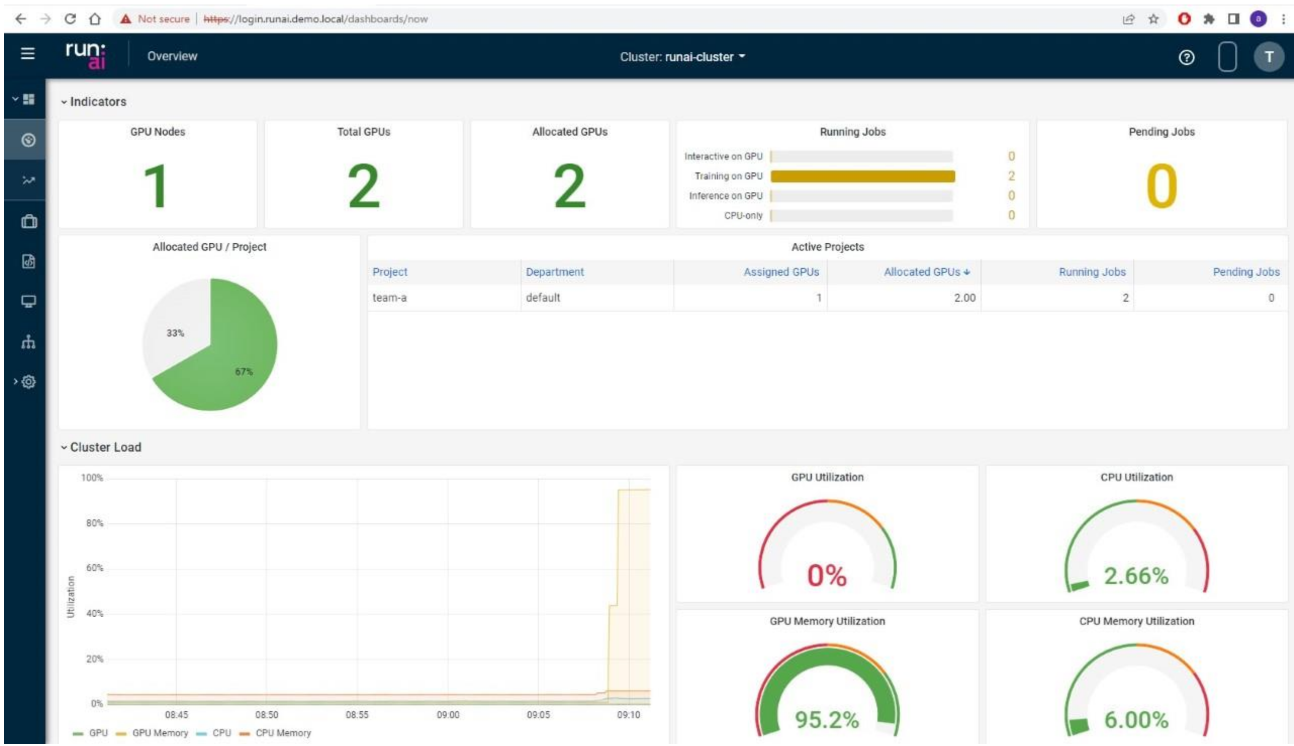


Figure 16 Run:ai Atlas Dashboard training utilisation

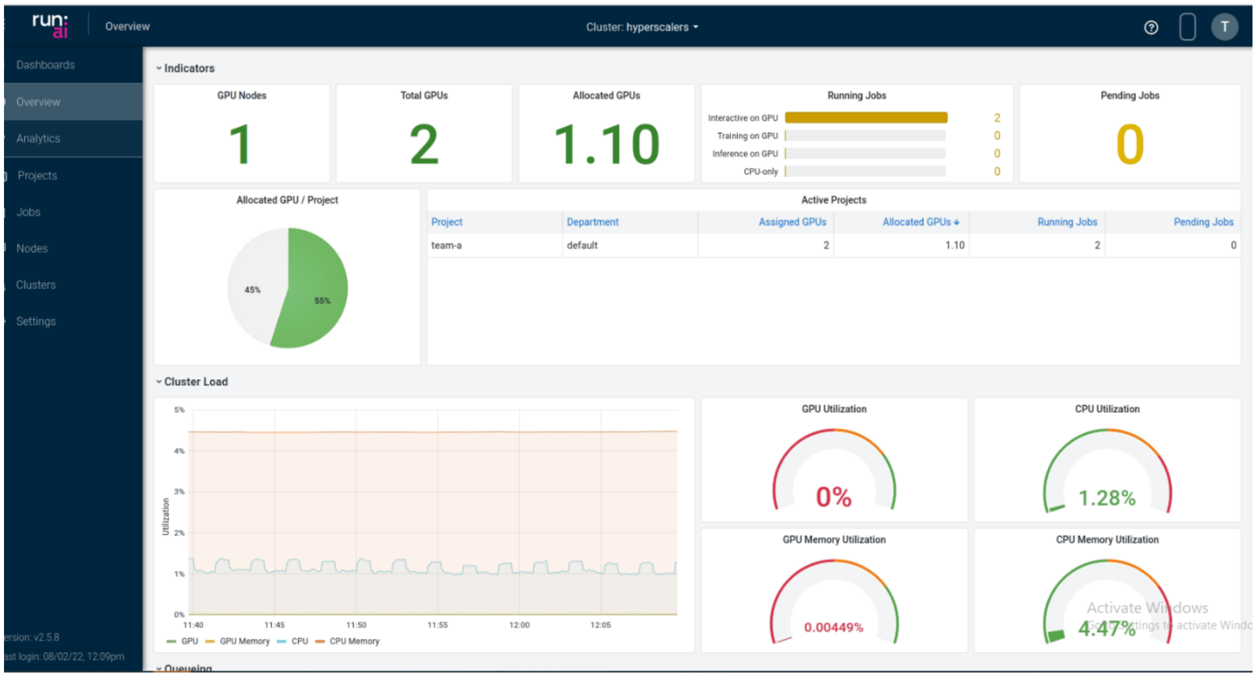


Figure 17 Run:ai Atlas Dashboard interactive utilisation

Train phase

AI Workload Scheduler

- Batch Scheduling
- Gang Scheduling
- Preemption

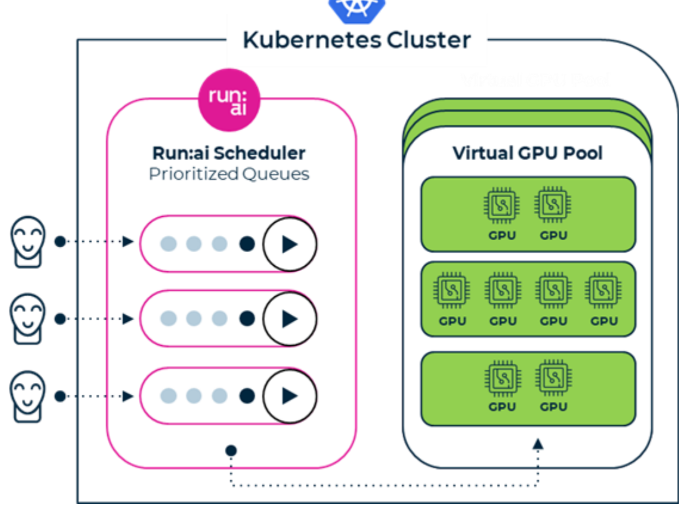


Figure 18 Run:ai Atlas scheduler features [15]

Batch scheduling – Queuing of jobs as they enter the system; can be scheduled as unattended

Gang scheduling – Multiple containers at once, parallel execution, unattended, Hyperparameter optimization and distributed training (multiple nodes and not multiple GPUs)

While creating a training job, one can request a node type/ specific GPU (Ex: A100) from the cluster to utilize higher CPU/ GPU memory resources required for training.

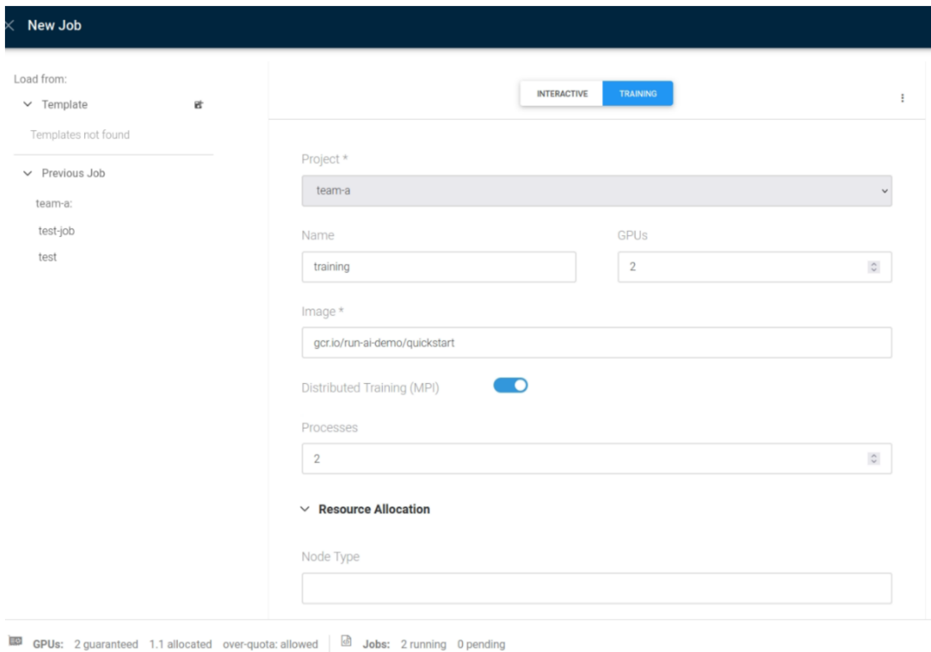


Figure 19 Creating a training job for distributed training (multi-node)

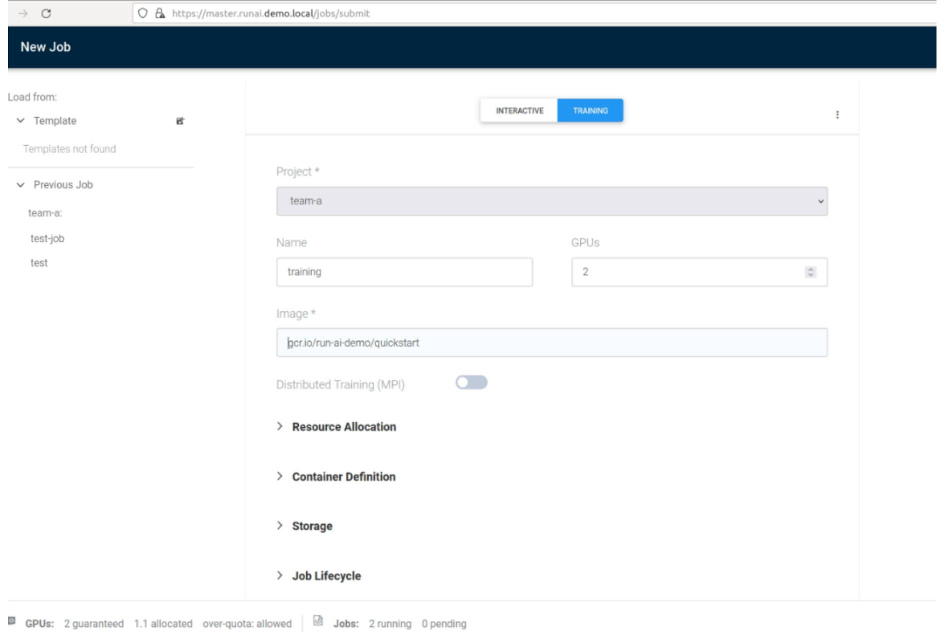


Figure 20 Creating a training job for multi-GPU training

Pre-emption

With pre-emption GPU resources are prioritized to high priority jobs by the scheduler and pauses any over provisioned GPU resources. When a pre-empted job continues, it starts at the closest checkpoint (It is recommended to integrate checkpoints into researcher's code). Generally, Interactive workloads (remote web sessions) take higher priority.

Inference phase

The key features to Run:ai Atlas in inference phase includes

1. Model Deployment
2. Deployments monitoring and management
3. Auto Scaling
4. Scale to Zero

With Run:ai Atlas, Models can be deployed with one GPU or multimodel servers or fractionalized GPUs, meaning, one can allocate right size of GPU resources to their jobs.

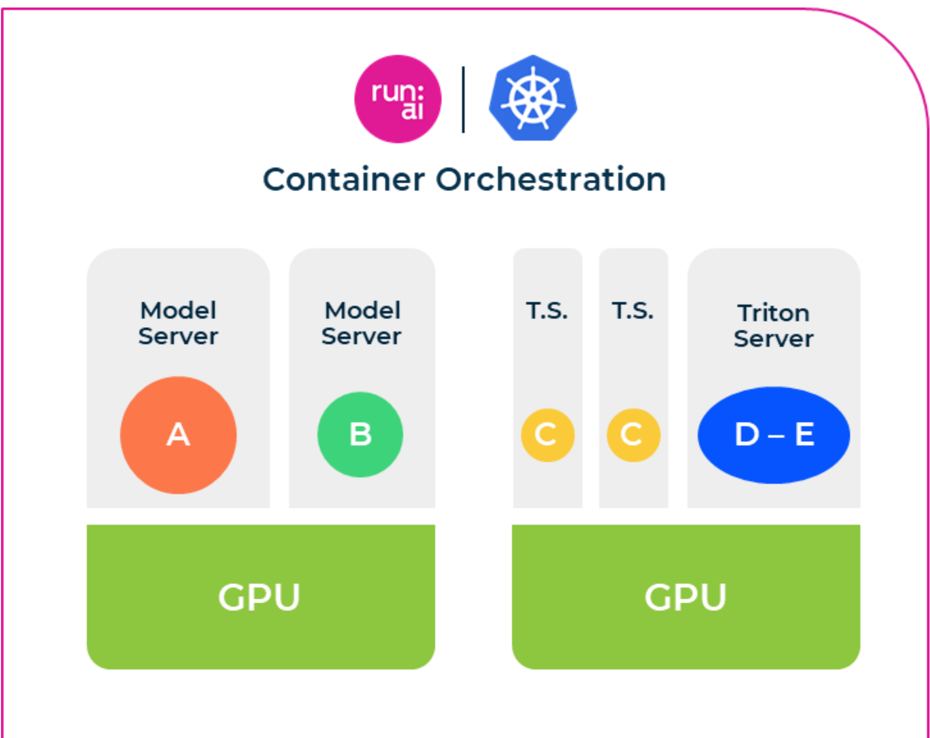


Figure 21 Run:ai Atlas Machine learning model deployment [15]

Auto scaling or setting scaling parameters for any deployed model. The following metrics influence the scaling up or down of any deployed pod (job).

1. GPU Utilization
2. CPU Utilization
3. Latency
4. Throughput
5. Concurrency

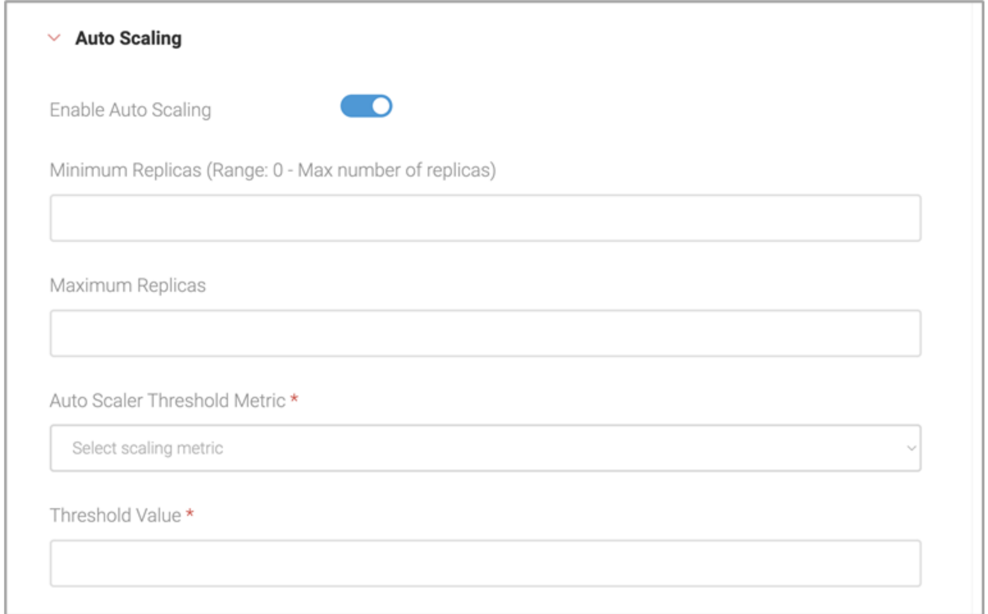


Figure 22 Autoscaling feature of Run:ai Atlas for any model deployment [15]

6 Integration

Run:ai Atlas offers integration into 3rd party MLOps and model serving tools like MLflow, Kubeflow and many more [16].

Integrate Run:ai with MLflow

MLflow is an open-source platform to manage the ML lifecycle, including experimentation, reproducibility, deployment, and a central model registry. The purpose of this section is to explain how to run Jobs with MLflow using the Run:ai scheduler.

Overview

MLflow concepts and alternative architectures are discussed here [17]. MLflow can run on various platforms. To work with Run:ai we would use the MLflow Kubernetes integration.

The MLflow documentation describes the Kubernetes integration as such:

When you run an MLflow Project on Kubernetes, MLflow constructs a new Docker image containing the Project's contents; this image inherits from the Project's Docker environment. MLflow then pushes the new Project image to your specified Docker registry and starts a Kubernetes Job on your specified Kubernetes cluster. This Kubernetes Job downloads the Project image and starts a corresponding Docker container. Finally, the container invokes your Project's entry point, logging parameters, tags, metrics, and artifacts to your MLflow tracking server.

To run an MLflow job via Kubernetes, you specify an MLflow Kubernetes configuration file that contains a template. Here is an example from the MLflow documentation:

```
{
  "kube-context": ...,
  "repository-uri": ...,
  "kube-job-template-path": "/username/path/to/kubernetes_job_template.yaml"
}
```

The essence of the Run:ai integration is the modification of the kubernetes_job_template.yaml file. Specifically adding the Run:ai scheduler name and the Run:ai Project (Kubernetes namespace).

Step by Step Instructions

Prerequisites

- Install MLflow [18].
- Make sure you have push access to a Docker repository from your local machine.
- Make sure you are connected to Run:ai via the Run:ai Command-line interface.

The sample MLflow Project

The relevant sample files are here [19]. These contain:

- A Dockerfile. This file builds a base docker image containing python3 and the required MLflow dependencies. The Docker file is already compiled and available at gcr.io/run-ai-demo/mlflow-demo.
- An MLflow project file MLproject. The project file contains the base image above as well as the python command-line to run.
- The training python code train.py
- MLflow Kubernetes configuration files as in the MLflow documentation.
- Kubernetes configuration file `kubernetes_config.json`
- An MLflow Kubernetes Job template `kubernetes_job_template.yaml`

Preparations

- Edit kubernetes_config.json.
    - Set kube-context to the name of the Kubernetes context. You can find the context name by running runai list clusters or kubectl config get-contexts.
    - Set repository-uri to a repository and name of a docker image that will be used by MLflow (this is a different image than the base docker image described above). Your local machine needs permissions to be able to push this image to the Docker registry.
- Edit kubernetes_job_template.yaml.
    - Set the value of namespace to runai-<name of Run:ai project>.
    - Note the last line which adds the Run:ai scheduler to the configuration.
    - Do not change the lines marked by {replaced with....
    - Set the requested resources including GPUs. You can use the --dry-run flag of the runai submit command to gain insight on additional configurations

Running

Perform docker login if required.

Run:

```
mlflow run mlproject -P alpha=5.0  -P l1-ratio=0.1  \
    --backend kubernetes --backend-config kubernetes_config.json
```

MLflow Tracking

The sample training code above does not contain references to an MLflow tracking server. This has been done to simplify the required setup. With MLflow-Kubernetes you will need a remote server architecture. Once you have such an architecture set up, you can use MLflow Tracking in your code.

Using Interactive Workloads

With Run:ai you can also run interactive workloads. To run the Job as interactive, add the following to kubernetes_job_template.yaml:

```
metadata:
  labels:
    priorityClassName: "build"
```

Integrate Run:ai with Kubeflow

Kubeflow is a platform for data scientists who want to build and experiment with ML pipelines. Kubeflow is also for ML engineers and operational teams who want to deploy ML systems to various environments for development, testing, and production-level serving.

This section describes the process of using Kubeflow in conjunction with Run:ai. Kubeflow submits jobs that are scheduled via Run:ai.

Kubeflow is a set of technologies. This document discusses Kubeflow Notebooks [20] and Kubeflow Pipelines [21].

Install Kubeflow

Use the default installation to install Kubeflow.

Install Run:ai Cluster

When installing Run:ai, customize the cluster installation as follows:

Set `createNamespaces` to `false`, as Kubeflow uses its own namespace convention.

Create Run:ai Projects

Kubeflow uses the namespace convention `kubeflow-<username>`. Use the 4 steps here to set up Run:ai projects and link them with Kubeflow namespaces.

Verify that the association has worked by running:

```
kubectl get rolebindings -n <KUBEFLOW-NAMESPACE>
```

See that role bindings starting with `runai-` were created.

Kubeflow, Users and Kubernetes Namespaces

Kubeflow has a multi-user architecture. A user has a Kubeflow profile which maps to a Kubernetes Namespace. This is similar to the Run:ai concept where a Run:ai Project is mapped to a Kubernetes namespace.

Kubeflow Notebooks

When starting a Kubeflow Notebook [22], you select a Kubeflow configuration. A Kubeflow configuration allows you to inject additional settings into the notebook, such as environment variables. To use Kubeflow with Run:ai you will use configurations to inject:

- The name of the Run:ai project
- Allocation of a fraction of a GPU, if required

Whole GPUs

To use Run:ai with whole GPUs (no fractions), apply the following configuration:

```
apiVersion: kubeflow.org/v1alpha1
kind: PodDefault
metadata:
  name: runai-non-fractional
  namespace: <KUBEFLOW-NAMESPACE>
spec:
  desc: "Use Run:ai scheduler (whole GPUs)"
  env:
    - name: RUNAI_PROJECT
      value: "<PROJECT>"
  selector:
    matchLabels:
      runai-non-fractional: "true"  # key must be identical to metadata.name
```

Where `<KUBEFLOW-NAMESPACE>` is the name of the namespace associated with the Kubeflow user and `<PROJECT>` is the name of the Run:ai project.

Within the Kubeflow Notebook creation form, select the new configuration as well as the number of GPUs required.

Fractions

The Kubeflow Notebook creation form only allows the selection of 1, 2, 4, or 8 GPUs. It is not possible to select a portion of a GPU (e.g. 0.5). As such, within the form, select `None` in the GPU box together with the following configuration:

```
apiVersion: kubeflow.org/v1alpha1
kind: PodDefault
metadata:
  name: runai-half-gpu
  namespace: <KUBEFLOW-NAMESPACE>
spec:
  desc: "Allocate 0.5 GPUs via Run:ai scheduler"
  env:
    - name: RUNAI_PROJECT
      value: "<PROJECT>"
    - name: RUNAI_GPU_FRACTION
      value: "0.5"
  selector:
    matchLabels:
      runai-half-gpu: "true"  # key must be identical to metadata.name
```

Similar configurations can be created for fractional configurations, other than 0.5.

Kubeflow Pipelines

Kubeflow Pipelines [23] is a platform for building and deploying portable, scalable machine learning (ML) workflows based on Docker containers.

As with Kubeflow Notebooks, the goal of this section is to run pipelines jobs within the context of Run:ai.

To create a Kubeflow pipeline, you:

- Write code using the Kubeflow Pipeline SDK [24].
- Package it into a single compressed file.
- Upload the file into Kubeflow and set it up.

The example code provided here shows how to augment pipeline code to use Run:ai

Whole GPUs

To the pipeline code add:

```
_training = training_op()
...
_training.add_pod_label('runai', 'true')
_training.add_pod_label('project', '<PROJECT>')
```

Where `<Project>` is the Run:ai project name. See example code here [25]

Compile the code by running:

```
dsl-compile --py kubeflow-runai-one-gpu.py --output kubeflow-runai-one-gpu.tar.gz
```

(dsl-compile is part of the Kubeflow Pipeline Python SDK).

Fractions

To allocate half a GPU, add the following to the pipeline code:

```
_training = training_op()
...
_training.add_pod_label('runai', 'true')
_training.add_pod_label('project', '<PROJECT>')
_training.add_pod_annotation('gpu-fraction', '0.5')
```

Where `<Project>` is the Run:ai project name. See example code here [26].

Compile the code as described above.

7 Maintenance

Introduction

Nodes (Machines) that are part of the cluster are susceptible to occasional downtime. This can be either as part of *planned maintenance* where we bring down the node for a specified time in an orderly fashion or an *unplanned downtime* where the machine abruptly stops due to a software or hardware issue.

The purpose of this section is to provide a process for retaining the Run:ai service and Researcher workloads during and after the downtime.

Self-hosted installation

The Self-hosted installation differs from the Classic (SaaS) installation of Run:ai in that it includes the Run:ai control-plane (backend). The control plane contains data that must be preserved during downtime. As such, you must first follow the disaster recovery planning process.

Node Types

The document differentiates between Run:ai System Worker Nodes and GPU Worker Nodes:

- Worker Nodes - are where Machine Learning workloads run.
- Run:ai System Nodes - In a production installation Run:ai software runs on one or more Run:ai System Nodes on which the Run:ai software runs.

Worker Nodes

Worker Nodes are where machine learning workloads run. Ideally, when a node is down, whether for planned maintenance or due to an abrupt downtime, these workloads should migrate to other available nodes or wait in the queue to be started when possible.

Training vs. Interactive

Run:ai differentiates between Training and Interactive workloads. The key difference at node downtime is that Training workloads will automatically move to a new node while Interactive workloads require a manual process. The manual process is recommended for Training workloads as well, as it hastens the process -- it takes time for Kubernetes to identify that a node is down.

Planned Maintenance

Before stopping a Worker node, perform the following:

- Stop the Kubernetes scheduler from starting new workloads on the node and drain node from all existing workloads. Workloads will move to other nodes or await on queue for renewed execution:

```
kubectl taint nodes <node-name> Run:ai=drain:NoExecute
```

- Shut down the node and perform the required maintenance.
- When done, start the node and then run:

```
kubectl taint nodes <node-name> Run:ai=drain:NoExecute-
```

Unplanned Downtime

- If a node has failed and has immediately restarted, all services will automatically start.
- If a node is to remain down for some time, you will want to drain the node so that workloads will migrate to another node:

```
kubectl taint nodes <node-name> Run:ai=drain:NoExecute
```

- When the node is up again, run:

```
kubectl taint nodes <node-name> Run:ai=drain:NoExecute-
```

- If the node is to be permanently shut down, you can remove it completely from Kubernetes. Run:

```
kubectl delete node <node-name>
```

- However, if you plan to bring back the node, you will need to rejoin the node into the cluster.

Run:ai System Nodes

In a production installation, Run:ai software runs on one or more Run:ai system nodes. As a best practice, it's best to have more than one such node so that during planned maintenance or unplanned downtime of a single node, the other node will take over. If a second node does not exist, you will have to designate an arbitrary node on the cluster as a Run:ai system node to complete the process below.

Protocols for planned maintenance and unplanned downtime are identical to Worker Nodes. See the section above.

Rejoin a Node into the Kubernetes Cluster

To rejoin a node to the cluster follow the following steps:

- On the master node, run:

```
kubeadm token create --print-join-command
```

- This would output a kubeadm join command. Run the command on the worker node for it to re-join the Kubernetes cluster. * Verify that the node is joined by running:

```
kubectl get nodes
```

- When the machine is up you will need to re-label nodes according to their role

8 Addendum

rntop

rntop [27] (pronounced as runtop) is a tool which helps you analyse an existing cluster (Not managed by Run:ai) of GPUs and their utilization. The tool requires root ssh access to all the GPU nodes that are to be tested.

```
Cluster
    GPU utilization:  24%    Used GPU memory:  7600 MiB / 78368 MiB (10%)

Nodes
    192.168.1.60            465.19.01    GPU utilization:  96%    Used GPU memory:  7600 MiB / 7979  MiB (95%)
    raz-qa-3                470.82.01    GPU utilization:   0%    Used GPU memory:     0 MiB / 7611  MiB (0%)
    dev-raz-worker-gpu-1    470.103.01   GPU utilization:   0%    Used GPU memory:     0 MiB / 30218 MiB (0%)
    menny-kernel-p100       470.57.02    GPU utilization:   1%    Used GPU memory:     0 MiB / 32560 MiB (0%)

GPUs
    192.168.1.60            0    GPU utilization:  96%    Used GPU memory:  7600 MiB / 7979  MiB (95%)
    raz-qa-3                0    GPU utilization:   0%    Used GPU memory:     0 MiB / 7611  MiB (0%)
    dev-raz-worker-gpu-1    0    GPU utilization:   0%    Used GPU memory:     0 MiB / 15109 MiB (0%)
    dev-raz-worker-gpu-1    1    GPU utilization:   0%    Used GPU memory:     0 MiB / 15109 MiB (0%)
    menny-kernel-p100       0    GPU utilization:   0%    Used GPU memory:     0 MiB / 16280 MiB (0%)
    menny-kernel-p100       1    GPU utilization:   2%    Used GPU memory:     0 MiB / 16280 MiB (0%)
```

Figure 23 A sample screenshot to demonstrate clustered GPUs [15]

9 References

[1] Hyperscalers, "About HS," [Online]. Available: https://www.hyperscalers.com/about-us-hyperscalers.

[2] RunAI, "RunAI Homepage," [Online]. Available: https://www.run.ai/. [Accessed 2022].

[3] NVIDIA Corporation, "Run:AI," [Online]. Available: https://developer.nvidia.com/run-ai. [Accessed 2022].

[4] Hyperscalers, "TYAN TN83-B8251," [Online]. Available: https://www.hyperscalers.com/nvidia-hyperscale-computing-gpu-server-A100-A6000-A40-TYAN-transport-TN83-B8251?search=HS%20GP1.

[5] TYAN, "TN83B8251," [Online]. Available: https://www.tyan.com/Barebones_TN83B8251_B8251T83E8HR-2T-N. [Accessed 2022].

[6] Hyperscalers, "S5N-D43N-3U," [Online]. Available: https://www.hyperscalers.com/quanta-qct-GPU-big-data-analytics-server-service-support-Xeon-Phi-NVidea/S5N-D43N-3U-Optimized-Accelerated-Server-hyperscalers-HS-buy. [Accessed 2022].

[7] Hyperscalers, "GZ2-G492-ZD2," [Online]. Available: https://www.hyperscalers.com/quanta-qct-GPU-big-data-analytics-server-service-support-Xeon-Phi-NVidea/hyperscalers-gigabyte-GZ2-G492-ZD2-nvidia-HGX-a100-80GB-40GB-SXM-GPU-buy-configure. [Accessed 2022].

[8] Gigabyte, "G492-ZD2-rev-A00," [Online]. Available: https://www.gigabyte.com/Enterprise/GPU-Server/G492-ZD2-rev-A00. [Accessed 2022].

[9] Intel Corporation, "Intel® Xeon® Silver 4210 Processor," [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/193384/intel-xeon-silver-4210-processor-13-75m-cache-2-20-ghz.html. [Accessed 2022].

[10] AMD Inc, "AMD EPYC™ 7543," [Online]. Available: https://www.amd.com/en/products/cpu/amd-epyc-7543. [Accessed 2022].

[11] Nvidia Corp, "RTX A6000 Datasheet," [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/quadro-product-literature/proviz-print-nvidia-rtx-a6000-datasheet-us-nvidia-1454980-r9-web%20(1).pdf. [Accessed 2022].

[12] NVIDIA Corporation, "NVIDIA A100 TENSOR CORE GPU," [Online]. Available: https://www.nvidia.com/en-au/data-center/a100/. [Accessed 2022].

[13] Canonical, "Ubuntu 20.04.4 LTS (Focal Fossa)," [Online]. Available: https://releases.ubuntu.com/20.04.4/. [Accessed 2022].

[14] Run:ai, "Run:ai Documentation Library," [Online]. Available: https://docs.run.ai/admin/overview-administrator/. [Accessed 2022].

[15] RunAI, "RunAI Beer with Engineers series," [Online]. Available: https://www.youtube.com/watch?v=q6fUcM2dHZ8&list=PLcZaJVeIq4kbcYhcVV6QmjRA1rgPil5fT . [Accessed 2022].

[16] Run:ai, "Run:ai Integration," [Online]. Available: https://docs.run.ai/admin/integration/mlflow/. [Accessed 2022].

[17] MLflow, "MLflow concepts," [Online]. Available: https://www.mlflow.org/docs/latest/concepts.html. [Accessed 2022].

[18] MLflow, "Install MLflow," [Online]. Available: https://www.mlflow.org/docs/latest/quickstart.html#installing-mlflow. [Accessed 2022].

[19] Run:ai, "MLflow Integration," [Online]. Available: https://github.com/run-ai/docs/tree/master/integrations/mlflow. [Accessed 2022].

[20] Kubeflow, "Kubeflow Notebooks - Documentation," [Online]. Available: https://www.kubeflow.org/docs/components/notebooks/. [Accessed 2022].

[21] Kubeflow, "Kubeflow Pipelines - Documentation," [Online]. Available: https://www.kubeflow.org/docs/components/pipelines/.

[22] Kubeflow, "Kubeflow Notebook setup," [Online]. Available: https://www.kubeflow.org/docs/components/notebooks/setup/. [Accessed 2022].

[23] Kubeflow, "Kubeflow Pipelines - Overview," [Online]. Available: https://www.kubeflow.org/docs/components/pipelines/overview/pipelines-overview/.

[24] Kubeflow, "Kubeflow Pipelines - Install SDK," [Online]. Available: https://www.kubeflow.org/docs/components/pipelines/sdk/install-sdk/. [Accessed 2022].

[25] Run:ai, "Kubeflow Integration -Sample," [Online]. Available: https://github.com/run-ai/docs/blob/master/integrations/kubeflow/kubeflow-runai-one-gpu.py. [Accessed 2022].

[26] Run:ai, "Kubeflow Integration - Half GPU Sample," [Online]. Available: https://github.com/run-ai/docs/blob/master/integrations/kubeflow/kubeflow-runai-half-gpu.py. [Accessed 2022].

[27] R. Rotenberg, "rntop," [Online]. Available: https://github.com/run-ai/rntop. [Accessed 2022].

Index

A

Addendum 49
Appliance Optimizer Utility AOU 11
Audience and Purpose 10

B

Base Product Deployment 16

C

Configure the Appliance 25

D

Digital IP Appliance Design Process 11

I

Important Considerations 15
Introduction 4

P

Preinstallation Requirements.. 18
Prerequisites for updating... 31

T

Testing the Appliance 32

U

Updating the Appliance.. 31