NetApp

NetApp Orchestration Solution with Run:AI
NetApp Solutions

NetApp
September 06, 2022

Table of Contents

TR-4858: NetApp Orchestration Solution with Run:AI

Rick Huang, David Arnette, Sung-Han Lin, NetApp
Yaron Goldberg, Run:AI

NetApp AFF storage systems deliver extreme performance and industry-leading hybrid cloud data-management capabilities. NetApp and Run:AI have partnered to demonstrate the unique capabilities of the NetApp ONTAP AI solution for artificial intelligence (AI) and machine learning (ML) workloads that provides enterprise-class performance, reliability, and support. Run:AI orchestration of AI workloads adds a Kubernetes-based scheduling and resource utilization platform to help researchers manage and optimize GPU utilization. Together with the NVIDIA DGX systems, the combined solution from NetApp, NVIDIA, and Run:AI provide an infrastructure stack that is purpose-built for enterprise AI workloads. This technical report gives directional guidance to customers building conversational AI systems in support of various use cases and industry verticals. It includes information about the deployment of Run:AI and a NetApp AFF A800 storage system and serves as a reference architecture for the simplest way to achieve fast, successful deployment of AI initiatives.

The target audience for the solution includes the following groups:

- Enterprise architects who design solutions for the development of AI models and software for Kubernetes-based use cases such as containerized microservices
- Data scientists looking for efficient ways to achieve efficient model development goals in a cluster environment with multiple teams and projects
- Data engineers in charge of maintaining and running production models
- Executive and IT decision makers and business leaders who would like to create the optimal Kubernetes cluster resource utilization experience and achieve the fastest time to market from AI initiatives

Solution Overview

NetApp ONTAP AI and AI Control Plane

The NetApp ONTAP AI architecture, developed and verified by NetApp and NVIDIA, is powered by NVIDIA DGX systems and NetApp cloud-connected storage systems. This reference architecture gives IT organizations the following advantages:

- Eliminates design complexities
- Enables independent scaling of compute and storage
- Enables customers to start small and scale seamlessly
- Offers a range of storage options for various performance and cost points

NetApp ONTAP AI tightly integrates DGX systems and NetApp AFF A800 storage systems with state-of-the-art networking. NetApp ONTAP AI and DGX systems simplify AI deployments by eliminating design complexity and guesswork. Customers can start small and grow their systems in an uninterrupted manner while intelligently managing data from the edge to the core to the cloud and back.

NetApp AI Control Plane is a full stack AI, ML, and deep learning (DL) data and experiment management solution for data scientists and data engineers. As organizations increase their use of AI, they face many

challenges, including workload scalability and data availability. NetApp AI Control Plane addresses these challenges through functionalities, such as rapidly cloning a data namespace just as you would a Git repo, and defining and implementing AI training workflows that incorporate the near-instant creation of data and model baselines for traceability and versioning. With NetApp AI Control Plane, you can seamlessly replicate data across sites and regions and swiftly provision Jupyter Notebook workspaces with access to massive datasets.

Run:AI Platform for AI Workload Orchestration

Run:AI has built the world's first orchestration and virtualization platform for AI infrastructure. By abstracting workloads from the underlying hardware, Run:AI creates a shared pool of GPU resources that can be dynamically provisioned, enabling efficient orchestration of AI workloads and optimized use of GPUs. Data scientists can seamlessly consume massive amounts of GPU power to improve and accelerate their research while IT teams retain centralized, cross-site control and real-time visibility over resource provisioning, queuing, and utilization. The Run:AI platform is built on top of Kubernetes, enabling simple integration with existing IT and data science workflows.

The Run:AI platform provides the following benefits:

- **Faster time to innovation.** By using Run:AI resource pooling, queueing, and prioritization mechanisms together with a NetApp storage system, researchers are removed from infrastructure management hassles and can focus exclusively on data science. Run:AI and NetApp customers increase productivity by running as many workloads as they need without compute or data pipeline bottlenecks.
- **Increased team productivity.** Run:AI fairness algorithms guarantee that all users and teams get their fair share of resources. Policies around priority projects can be preset, and the platform enables dynamic allocation of resources from one user or team to another, helping users to get timely access to coveted GPU resources.
- **Improved GPU utilization.** The Run:AI Scheduler enables users to easily make use of fractional GPUs, integer GPUs, and multiple nodes of GPUs for distributed training on Kubernetes. In this way, AI workloads run based on your needs, not capacity. Data science teams are able to run more AI experiments on the same infrastructure.

Solution Technology

This solution was implemented with one NetApp AFF A800 system, two DGX-1 servers, and two Cisco Nexus 3232C 100GbE-switches. Each DGX-1 server is connected to the Nexus switches with four 100GbE connections that are used for inter-GPU communications by using remote direct memory access (RDMA) over Converged Ethernet (RoCE). Traditional IP communications for NFS storage access also occur on these links. Each storage controller is connected to the network switches by using four 100GbE-links. The following figure shows the ONTAP AI solution architecture used in this technical report for all testing scenarios.

Hardware Used in This Solution

This solution was validated using the ONTAP AI reference architecture two DGX-1 nodes and one AFF A800 storage system. See NVA-1121 for more details about the infrastructure used in this validation.

The following table lists the hardware components that are required to implement the solution as tested.

Hardware	Quantity
DGX-1 systems	2
AFF A800	1
Nexus 3232C switches	2

Software Requirements

This solution was validated using a basic Kubernetes deployment with the Run:AI operator installed. Kubernetes was deployed using the NVIDIA DeepOps deployment engine, which deploys all required components for a production-ready environment. DeepOps automatically deployed NetApp Trident for persistent storage integration with the k8s environment, and default storage classes were created so containers leverage storage from the AFF A800 storage system. For more information on Trident with Kubernetes on ONTAP AI, see TR-4798.

The following table lists the software components that are required to implement the solution as tested.

Software	Version or Other Information
NetApp ONTAP data management software	9.6p4
Cisco NX-OS switch firmware	7.0(3)I6(1)
NVIDIA DGX OS	4.0.4 - Ubuntu 18.04 LTS

Software	Version or Other Information
Kubernetes version	1.17
Trident version	20.04.0
Run:AI CLI	v2.1.13
Run:AI Orchestration Kubernetes Operator version	1.0.39
Docker container platform	18.06.1-ce [e68fc7a]

Additional software requirements for Run:AI can be found at Run:AI GPU cluster prerequisites.

Next: Optimal Cluster and GPU Utilization with Run AI

Optimal Cluster and GPU Utilization with Run:AI

The following sections provide details on the Run:AI installation, test scenarios, and results performed in this validation.

We validated the operation and performance of this system by using industry standard benchmark tools, including TensorFlow benchmarks. The ImageNet dataset was used to train ResNet-50, which is a famous Convolutional Neural Network (CNN) DL model for image classification. ResNet-50 delivers an accurate training result with a faster processing time, which enabled us to drive a sufficient demand on the storage.

Next: Run AI Installation.

Run:AI Installation

To install Run:AI, complete the following steps:

1. Install the Kubernetes cluster using DeepOps and configure the NetApp default storage class.
2. Prepare GPU nodes:
    a. Verify that NVIDIA drivers are installed on GPU nodes.
    b. Verify that `nvidia-docker` is installed and configured as the default docker runtime.
3. Install Run:AI:
    a. Log into the Run:AI Admin UI to create the cluster.
    b. Download the created `runai-operator-<clustername>.yaml` file.
    c. Apply the operator configuration to the Kubernetes cluster.

    ```
    kubectl apply -f runai-operator-<clustername>.yaml
    ```

4. Verify the installation:
    a. Go to https://app.run.ai/.
    b. Go to the Overview dashboard.
    c. Verify that the number of GPUs on the top right reflects the expected number of GPUs and the GPU nodes are all in the list of servers.For more information about Run:AI deployment, see installing Run:AI on an on-premise Kubernetes cluster and installing the Run:AI CLI.

Run:AI Dashboards and Views

After installing Run:AI on your Kubernetes cluster and configuring the containers correctly, you see the following dashboards and views on https://app.run.ai in your browser, as shown in the following figure.

There are 16 total GPUs in the cluster provided by two DGX-1 nodes. You can see the number of nodes, the total available GPUs, the allocated GPUs that are assigned with workloads, the total number of running jobs, pending jobs, and idle allocated GPUs. On the right side, the bar diagram shows GPUs per Project, which summarizes how different teams are using the cluster resource. In the middle is the list of currently running jobs with job details, including job name, project, user, job type, the node each job is running on, the number of GPU(s) allocated for that job, the current run time of the job, job progress in percentage, and the GPU utilization for that job. Note that the cluster is under-utilized (GPU utilization at 23%) because there are only three running jobs submitted by a single team (team-a).

In the following section, we show how to create multiple teams in the Projects tab and allocate GPUs for each team to maximize cluster usage and manage resources when there are many users per cluster. The test scenarios mimic enterprise environments in which memory and GPU resources are shared among training, inferencing, and interactive workloads.

Creating Projects for Data Science Teams and Allocating GPUs

Researchers can submit workloads through the Run:AI CLI, Kubeflow, or similar processes. To streamline resource allocation and create prioritization, Run:AI introduces the concept of Projects. Projects are quota entities that associate a project name with GPU allocation and preferences. It is a simple and convenient way to manage multiple data science teams.

A researcher submitting a workload must associate a project with a workload request. The Run:AI scheduler

compares the request against the current allocations and the project and determines whether the workload can be allocated resources or whether it should remain in a pending state.

As a system administrator, you can set the following parameters in the Run:AI Projects tab:

- **Model projects.** Set a project per user, set a project per team of users, and set a project per a real organizational project.

- **Project quotas.** Each project is associated with a quota of GPUs that can be allocated for this project at the same time. This is a guaranteed quota in the sense that researchers using this project are guaranteed to get this number of GPUs no matter what the status in the cluster is. As a rule, the sum of the project allocation should be equal to the number of GPUs in the cluster. Beyond that, a user of this project can receive an over-quota. As long as GPUs are unused, a researcher using this project can get more GPUs. We demonstrate over-quota testing scenarios and fairness considerations in Achieving High Cluster Utilization with Over-Quota GPU Allocation, Basic Resource Allocation Fairness, and Over-Quota Fairness.

- Create a new project, update an existing project, and delete an existing project.

- **Limit jobs to run on specific node groups**. You can assign specific projects to run only on specific nodes. This is useful when the project team needs specialized hardware, for example, with enough memory. Alternatively, a project team might be the owner of specific hardware that was acquired with a specialized budget, or when you might need to direct build or interactive workloads to work on weaker hardware and direct longer training or unattended workloads to faster nodes. For commands to group nodes and set affinity for a specific project, see the Run:AI Documentation.

- **Limit the duration of interactive jobs**. Researchers frequently forget to close interactive jobs. This might lead to a waste of resources. Some organizations prefer to limit the duration of interactive jobs and close them automatically.

The following figure shows the Projects view with four teams created. Each team is assigned a different number of GPUs to account for different workloads, with the total number of GPUs equal to that of the total available GPUs in a cluster consisting of two DGX-1s.

Next: Submitting Jobs in Run AI CLI

Submitting Jobs in Run:AI CLI

This section provides the detail on basic Run:AI commands that you can use to run any Kubernetes job. It is divided into three parts according to workload type. AI/ML/DL workloads can be divided into two generic types:

- **Unattended training sessions**. With these types of workloads, the data scientist prepares a self-running workload and sends it for execution. During the execution, the customer can examine the results. This type of workload is often used in production or when model development is at a stage where no human

intervention is required.

- **Interactive build sessions**. With these types of workloads, the data scientist opens an interactive session with Bash, Jupyter Notebook, remote PyCharm, or similar IDEs and accesses GPU resources directly. We include a third scenario for running interactive workloads with connected ports to reveal an internal port to the container user..

Unattended Training Workloads

After setting up projects and allocating GPU(s), you can run any Kubernetes workload using the following command at the command line:

```
$ runai project set team-a runai submit hyper1 -i gcr.io/run-ai-
demo/quickstart -g 1
```

This command starts an unattended training job for team-a with an allocation of a single GPU. The job is based on a sample docker image, `gcr.io/run-ai-demo/quickstart`. We named the job `hyper1`. You can then monitor the job's progress by running the following command:

```
$ runai list
```

The following figure shows the result of the `runai list` command. Typical statuses you might see include the following:

- `ContainerCreating`. The docker container is being downloaded from the cloud repository.
- `Pending`. The job is waiting to be scheduled.
- `Running`. The job is running.

To get an additional status on your job, run the following command:

```
$ runai get hyper1
```

To view the logs of the job, run the `runai logs <job-name>` command:

```
$ runai logs hyper1
```

In this example, you should see the log of a running DL session, including the current training epoch, ETA, loss function value, accuracy, and time elapsed for each step.

You can view the cluster status on the Run:AI UI at https://app.run.ai/. Under Dashboards > Overview, you can monitor GPU utilization.

To stop this workload, run the following command:

```
$ runai delte hyper1
```

This command stops the training workload. You can verify this action by running `runai list` again. For more detail, see launching unattended training workloads.

Interactive Build Workloads

After setting up projects and allocating GPU(s) you can run an interactive build workload using the following command at the command line:

```
$ runai submit build1 -i python -g 1 --interactive --command sleep --args
  infinity
```

The job is based on a sample docker image python. We named the job build1.

> (i) The `-- interactive` flag means that the job does not have a start or end. It is the researcher's responsibility to close the job. The administrator can define a time limit for interactive jobs after which they are terminated by the system.

The `--g 1` flag allocates a single GPU to this job. The command and argument provided is `--command sleep —args infinity`. You must provide a command, or the container starts and then exits immediately.

The following commands work similarly to the commands described in Unattended Training Workloads:

- `runai list`: Shows the name, status, age, node, image, project, user, and GPUs for jobs.
- `runai get build1`: Displays additional status on the job build1.
- `runai delete build1`: Stops the interactive workload build1.To get a bash shell to the container, the following command:

```
$ runai bash build1
```

This provides a direct shell into the computer. Data scientists can then develop or finetune their models within the container.

You can view the cluster status on the Run:AI UI at https://app.run.ai. For more detail, see starting and using interactive build workloads.

Interactive Workloads with Connected Ports

As an extension of interactive build workloads, you can reveal internal ports to the container user when starting a container with the Run:AI CLI. This is useful for cloud environments, working with Jupyter Notebooks, or connecting to other microservices. Ingress allows access to Kubernetes services from outside the Kubernetes cluster. You can configure access by creating a collection of rules that define which inbound connections reach which services.

For better management of external access to the services in a cluster, we suggest that cluster administrators install Ingress and configure LoadBalancer.

To use Ingress as a service type, run the following command to set the method type and the ports when submitting your workload:

```
$ runai submit test-ingress -i jupyter/base-notebook -g 1 \
   --interactive --service-type=ingress --port 8888 \
   --args="--NotebookApp.base_url=test-ingress" --command=start-notebook.sh
```

After the container starts successfully, execute `runai list` to see the `SERVICE URL(S)` with which to access the Jupyter Notebook. The URL is composed of the ingress endpoint, the job name, and the port. For example, see https://10.255.174.13/test-ingress-8888.

For more details, see launching an interactive build workload with connected ports.

Next: Achieving High Cluster Utilization

Achieving High Cluster Utilization

In this section, we emulate a realistic scenario in which four data science teams each submit their own workloads to demonstrate the Run:AI orchestration solution that achieves high cluster utilization while maintaining prioritization and balancing GPU resources. We start by using the ResNet-50 benchmark described in the section ResNet-50 with ImageNet Dataset Benchmark Summary:

```
$ runai submit netapp1 -i netapp/tensorflow-tf1-py3:20.01.0 --local-image
--large-shm  -v /mnt:/mnt -v /tmp:/tmp --command python --args
"/netapp/scripts/run.py" --args "--
dataset_dir=/mnt/mount_0/dataset/imagenet/imagenet_original/" --args "--
num_mounts=2"  --args "--dgx_version=dgx1" --args "--num_devices=1" -g 1
```

We ran the same ResNet-50 benchmark as in NVA-1121. We used the flag `--local-image` for containers not residing in the public docker repository. We mounted the directories `/mnt` and `/tmp` on the host DGX-1 node to `/mnt` and `/tmp` to the container, respectively. The dataset is at NetApp AFFA800 with the `dataset_dir` argument pointing to the directory. Both `--num_devices=1` and `-g 1` mean that we allocate one GPU for this job. The former is an argument for the `run.py` script, while the latter is a flag for the `runai submit` command.

The following figure shows a system overview dashboard with 97% GPU utilization and all sixteen available GPUs allocated. You can easily see how many GPUs are allocated for each team in the GPUs/Project bar chart. The Running Jobs pane shows the current running job names, project, user, type, node, GPUs consumed, run time, progress, and utilization details. A list of workloads in queue with their wait time is shown in Pending Jobs. Finally, the Nodes box offers GPU numbers and utilization for individual DGX-1 nodes in the cluster.

Next: Fractional GPU Allocation for Less Demanding or Interactive Workloads

Fractional GPU Allocation for Less Demanding or Interactive Workloads

When researchers and developers are working on their models, whether in the development, hyperparameter tuning, or debugging stages, such workloads usually require fewer computational resources. It is therefore more efficient to provision fractional GPU and memory such that the same GPU can simultaneously be allocated to other workloads. Run:AI's orchestration solution provides a fractional GPU sharing system for containerized workloads on Kubernetes. The system supports workloads running CUDA programs and is especially suited for lightweight AI tasks such as inference and model building. The fractional GPU system transparently gives data science and AI engineering teams the ability to run multiple workloads simultaneously on a single GPU. This enables companies to run more workloads, such as computer vision, voice recognition, and natural language processing on the same hardware, thus lowering costs.

Run:AI's fractional GPU system effectively creates virtualized logical GPUs with their own memory and computing space that containers can use and access as if they were self-contained processors. This enables several workloads to run in containers side-by-side on the same GPU without interfering with each other. The solution is transparent, simple, and portable and it requires no changes to the containers themselves.

A typical usecase could see two to eight jobs running on the same GPU, meaning that you could do eight times the work with the same hardware.

For the job `frac05` belonging to project `team-d` in the following figure, we can see that the number of GPUs allocated was 0.50. This is further verified by the `nvidia-smi` command, which shows that the GPU memory available to the container was 16,255MB: half of the 32GB per V100 GPU in the DGX-1 node.

```
root@run-deploy:~# runai bash frac05 -p team-d
root@frac05-0:/workload# nvidia-smi
Tue Jul 28 15:17:03 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 450.51.05    Driver Version: 450.51.05    CUDA Version: 11.0      |
|-------------------------------+----------------------+----------------------+
| 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  Tesla V100-SXM2...  On   | 00000000:07:00.0 Off |                    0 |
| N/A   57C    P0   240W / 300W |  15525MiB / 16255MiB |    100%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A       156      C   python3                          15525MiB |
+-----------------------------------------------------------------------------+
```

Achieving High Cluster Utilization with Over-Quota GPU Allocation

In this section and in the sections Basic Resource Allocation Fairness, and Over-Quota Fairness, we have devised advanced testing scenarios to demonstrate the Run:AI orchestration capabilities for complex workload management, automatic preemptive scheduling, and over-quota GPU provisioning. We did this to achieve high cluster-resource usage and optimize enterprise-level data science team productivity in an ONTAP AI environment.

For these three sections, set the following projects and quotas:

Project	Quota
team-a	4
team-b	2
team-c	2
team-d	8

In addition, we use the following containers for these three sections:

- Jupyter Notebook: `jupyter/base-notebook`
- Run:AI quickstart: `gcr.io/run-ai-demo/quickstart`

We set the following goals for this test scenario:

- Show the simplicity of resource provisioning and how resources are abstracted from users

- Show how users can easily provision fractions of a GPU and integer number of GPUs

- Show how the system eliminates compute bottlenecks by allowing teams or users to go over their resource quota if there are free GPUs in the cluster

- Show how data pipeline bottlenecks are eliminated by using the NetApp solution when running compute-intensive jobs, such as the NetApp container

- Show how multiple types of containers are running using the system

  - Jupyter Notebook

  - Run:AI container

- Show high utilization when the cluster is full

For details on the actual command sequence executed during the testing, see Testing Details for Section 4.8.

When all 13 workloads are submitted, you can see a list of container names and GPUs allocated, as shown in the following figure. We have seven training and six interactive jobs, simulating four data science teams, each with their own models running or in development. For interactive jobs, individual developers are using Jupyter Notebooks to write or debug their code. Thus, it is suitable to provision GPU fractions without using too many cluster resources.

```
root@run-deploy:~# runai list -A
NAME            STATUS   AGE  NODE     IMAGE                             TYPE         PROJECT  USER  GPUs  CREATED BY CLI  SERVICE URL(S)
b-4-gg          Running  2m   dgx1-2   gcr.io/run-ai-demo/quickstart     Train        team-b   root  2     true
c-5-g           Running  2m   dgx1-2   gcr.io/run-ai-demo/quickstart     Train        team-c   root  1     true
c-4-gg          Running  2m   dgx1-1   gcr.io/run-ai-demo/quickstart     Train        team-c   root  2     true
b-3-g           Running  2m   dgx1-1   gcr.io/run-ai-demo/quickstart     Train        team-b   root  1     true
c-3-g02         Running  2m   dgx1-1   gcr.io/run-ai-demo/quickstart     Interactive  team-c   root  0.2   true
d-1-gggg        Running  2m   dgx1-2   gcr.io/run-ai-demo/quickstart     Train        team-d   root  4     true
c-2-g03         Running  2m   dgx1-1   gcr.io/run-ai-demo/quickstart     Interactive  team-c   root  0.3   true
c-1-g05         Running  2m   dgx1-1   gcr.io/run-ai-demo/quickstart     Interactive  team-c   root  0.5   true
a-2-gg          Running  3m   dgx1-1   gcr.io/run-ai-demo/quickstart     Train        team-a   root  2     true
b-2-g04         Running  3m   dgx1-2   gcr.io/run-ai-demo/quickstart     Interactive  team-b   root  0.4   true
a-1-g           Running  3m   dgx1-1   gcr.io/run-ai-demo/quickstart     Train        team-a   root  1     true
b-1-g06         Running  3m   dgx1-2   gcr.io/run-ai-demo/quickstart     Interactive  team-b   root  0.6   true
a-1-1-jupyter   Running  3m   dgx1-1   jupyter/base-notebook             Interactive  team-a   root  1     true           http://10.61.218.134/a-1-1-jupyter,
https://10.61.218.134/a-1-1-jupyter
```

The results of this testing scenario show the following:

- The cluster should be full: 16/16 GPUs are used.

- High cluster utilization.

- More experiments than GPUs due to fractional allocation.

- team-d is not using all their quota; therefore, team-b and team-c can use additional GPUs for their experiments, leading to faster time to innovation.

Next: Basic Resource Allocation Fairness

Basic Resource Allocation Fairness

In this section, we show that, when team-d asks for more GPUs (they are under their quota), the system pauses the workloads of team-b and team-c and moves them into a pending state in a fair-share manner.

For details including job submissions, container images used, and command sequences executed, see the section Testing Details for Section 4.9.

The following figure shows the resulting cluster utilization, GPUs allocated per team, and pending jobs due to automatic load balancing and preemptive scheduling. We can observe that when the total number of GPUs

requested by all team workloads exceeds the total available GPUs in the cluster, Run:AI's internal fairness algorithm pauses one job each for `team-b` and `team-c` because they have met their project quota. This provides overall high cluster utilization while data science teams still work under resource constraints set by an administrator.

Nodes	Total GPUs	Allocated GPUs	GPU Utilization	GPUs / Project
2	16	16	86%	
Running Jobs	Pending Jobs	Idle Allocated GPUs		
13	2	1		

Running Jobs

Job	Project	User	Type	Node	GPUs	Run Time	Progress	Utilization
c-3-g02	team-c	root	Interactive	dgx1-1	0.20	00:11:40	16%	99%
a-1-g	team-a	root	Train	dgx1-1	1.00	00:11:49	23%	100%
b-2-g04	team-b	root	Interactive	dgx1-2	0.40	00:11:48	13%	100%
c-5-a	team-c	root	Train	dgx1-2	1.00	00:11:38	24%	66%

Pending Jobs

Job	Project	User	Type	Requested GPUs	Wait Time
b-4-gg	team-b	root	Train	2.00	00:01:34
c-4-gg	team-c	root	Train	2.00	00:04:49

Nodes

Node ^	Total GPUs	Allocated GPUs	Utilization
dgx1-1	8	8	85%
dgx1-2	8	8	89%

The results of this testing scenario demonstrate the following:

- **Automatic load balancing.** The system automatically balances the quota of the GPUs, such that each team is now using their quota. The workloads that were paused belong to teams that were over their quota.
- **Fair share pause.** The system chooses to stop the workload of one team that was over their quota and then stop the workload of the other team. Run:AI has internal fairness algorithms.

Over-Quota Fairness

In this section, we expand the scenario in which multiple teams submit workloads and exceed their quota. In this way, we demonstrate how Run:AI's fairness algorithm allocates cluster resources according to the ratio of preset quotas.

Goals for this test scenario:

- Show queuing mechanism when multiple teams are requesting GPUs over their quota.
- Show how the system distributes a fair share of the cluster between multiple teams that are over their quota according to the ratio between their quotas, so that the team with the larger quota gets a larger share of the spare capacity.

At the end of Basic Resource Allocation Fairness, there are two workloads queued: one for `team-b` and one

for `team-c`. In this section, we queue additional workloads.

For details including job submissions, container images used, and command sequences executed, see Testing Details for section 4.10.

When all jobs are submitted according to the section Testing Details for section 4.10, the system dashboard shows that `team-a`, `team-b`, and `team-c` all have more GPUs than their preset quota. `team-a` occupies four more GPUs than its preset soft quota (four), whereas `team-b` and `team-c` each occupy two more GPUs than their soft quota (two). The ratio of over-quota GPUs allocated is equal to that of their preset quota. This is because the system used the preset quota as a reference of priority and provisioned accordingly when multiple teams request more GPUs, exceeding their quota. Such automatic load balancing provides fairness and prioritization when enterprise data science teams are actively engaged in AI model development and production.

Nodes	Total GPUs	Allocated GPUs	GPU Utilization	GPUs / Project
2	**16**	**16**	**86%**	
Running Jobs	Pending Jobs	Idle Allocated GPUs		
14	**2**	**1**		

Running Jobs

Job	Project	User	Type	Node	GPUs	Run Time	Progress	Utilization
c-3-g02	team-c	root	Interactive	dgx1-1	0.20	00:22:01	29%	100%
a-1-g	team-a	root	Train	dgx1-1	1.00	00:22:10	42%	99%
b-2-g04	team-b	root	Interactive	dgx1-2	0.40	00:22:09	24%	100%

Pending Jobs

Job	Project	User	Type	Requested GPUs	Wait Time
c-6-gg	team-c	root	Train	2.00	00:07:45
b-5-gg	team-b	root	Train	2.00	00:07:46

Nodes

Node	Total GPUs	Allocated GPUs	Utilization
dgx1-1	8	8	68%
dgx1-2	8	8	89%

The results of this testing scenario show the following:

- The system starts to de-queue the workloads of other teams.
- The order of the dequeuing is decided according to fairness algorithms, such that `team-b` and `team-c` get the same amount of over-quota GPUs (since they have a similar quota), and `team-a` gets a double amount of GPUs since their quota is two times higher than the quota of `team-b` and `team-c`.
- All the allocation is done automatically.

Therefore, the system should stabilize on the following states:

Project	GPUs allocated	Comment
team-a	8/4	Four GPUs over the quota. Empty queue.

Project	GPUs allocated	Comment
team-b	4/2	Two GPUs over the quota. One workload queued.
team-c	4/2	Two GPUs over the quota. One workload queued.
team-d	0/8	Not using GPUs at all, no queued workloads.

The following figure shows the GPU allocation per project over time in the Run:AI Analytics dashboard for the sections Achieving High Cluster Utilization with Over-Quota GPU Allocation, Basic Resource Allocation Fairness, and Over-Quota Fairness. Each line in the figure indicates the number of GPUs provisioned for a given data science team at any time. We can see that the system dynamically allocates GPUs according to workloads submitted. This allows teams to go over quota when there are available GPUs in the cluster, and then preempt jobs according to fairness, before finally reaching a stable state for all four teams.

Saving Data to a Trident-Provisioned PersistentVolume

NetApp Trident is a fully supported open source project designed to help you meet the sophisticated persistence demands of your containerized applications. You can read and write data to a Trident-provisioned Kubernetes PersistentVolume (PV) with the added benefit of data tiering, encryption, NetApp Snapshot technology, compliance, and high performance offered by NetApp ONTAP data management software.

Reusing PVCs in an Existing Namespace

For larger AI projects, it might be more efficient for different containers to read and write data to the same Kubernetes PV. To reuse a Kubernetes Persistent Volume Claim (PVC), the user must have already created a PVC. See the NetApp Trident documentation for details on creating a PVC. Here is an example of reusing an existing PVC:

```
$ runai submit pvc-test -p team-a --pvc test:/tmp/pvc1mount -i gcr.io/run-
ai-demo/quickstart -g 1
```

Run the following command to see the status of job `pvc-test` for project `team-a`:

```
$ runai get pvc-test -p team-a
```

You should see the PV /tmp/pvc1mount mounted to `team-a` job `pvc-test`. In this way, multiple containers can read from the same volume, which is useful when there are multiple competing models in development or in production. Data scientists can build an ensemble of models and then combine prediction results by majority voting or other techniques.

Use the following to access the container shell:

```
$ runai bash pvc-test -p team-a
```

You can then check the mounted volume and access your data within the container.

This capability of reusing PVCs works with NetApp FlexVol volumes and NetApp ONTAP FlexGroup volumes, enabling data engineers more flexible and robust data management options to leverage your data fabric powered by NetApp.

Conclusion

NetApp and Run:AI have partnered in this technical report to demonstrate the unique capabilities of the NetApp ONTAP AI solution together with the Run:AI Platform for simplifying orchestration of AI workloads. The preceding steps provide a reference architecture to streamline the process of data pipelines and workload orchestration for deep learning. Customers looking to implement these solutions are encouraged to reach out to NetApp and Run:AI for more information.

Testing Details for Section 4.8

This section contains the testing details for the section Achieving High Cluster Utilization with Over-Quota GPU Allocation.

Submit jobs in the following order:

Project	Image	# GPUs	Total	Comment
team-a	Jupyter	1	1/4	–
team-a	NetApp	1	2/4	–
team-a	Run:AI	2	4/4	Using all their quota

Project	Image	# GPUs	Total	Comment
team-b	Run:AI	0.6	0.6/2	Fractional GPU
team-b	Run:AI	0.4	1/2	Fractional GPU
team-b	NetApp	1	2/2	–
team-b	NetApp	2	4/2	Two over quota
team-c	Run:AI	0.5	0.5/2	Fractional GPU
team-c	Run:AI	0.3	0.8/2	Fractional GPU
team-c	Run:AI	0.2	1/2	Fractional GPU
team-c	NetApp	2	3/2	One over quota
team-c	NetApp	1	4/2	Two over quota
team-d	NetApp	4	4/8	Using half of their quota

Command structure:

```
$ runai submit <job-name> -p <project-name> -g <#GPUs> -i <image-name>
```

Actual command sequence used in testing:

```
$ runai submit a-1-1-jupyter -i jupyter/base-notebook -g 1 \
  --interactive --service-type=ingress --port 8888 \
  --args="--NotebookApp.base_url=team-a-test-ingress" --command=start
-notebook.sh -p team-a
$ runai submit a-1-g -i gcr.io/run-ai-demo/quickstart -g 1 -p team-a
$ runai submit a-2-gg -i gcr.io/run-ai-demo/quickstart -g 2 -p team-a
$ runai submit b-1-g06 -i gcr.io/run-ai-demo/quickstart -g 0.6
--interactive -p team-b
$ runai submit b-2-g04 -i gcr.io/run-ai-demo/quickstart -g 0.4
--interactive -p team-b
$ runai submit b-3-g -i gcr.io/run-ai-demo/quickstart -g 1 -p team-b
$ runai submit b-4-gg -i gcr.io/run-ai-demo/quickstart -g 2 -p team-b
$ runai submit c-1-g05 -i gcr.io/run-ai-demo/quickstart -g 0.5
--interactive -p team-c
$ runai submit c-2-g03 -i gcr.io/run-ai-demo/quickstart -g 0.3
--interactive -p team-c
$ runai submit c-3-g02 -i gcr.io/run-ai-demo/quickstart -g 0.2
--interactive -p team-c
$ runai submit c-4-gg -i gcr.io/run-ai-demo/quickstart -g 2 -p team-c
$ runai submit c-5-g -i gcr.io/run-ai-demo/quickstart -g 1 -p team-c
$ runai submit d-1-gggg -i gcr.io/run-ai-demo/quickstart -g 4 -p team-d
```

At this point, you should have the following states:

Project	GPUs Allocated	Workloads Queued
team-a	4/4 (soft quota/actual allocation)	None
team-b	4/2	None
team-c	4/2	None
team-d	4/8	None

See the section Achieving High Cluster Utilization with Over-uota GPU Allocation for discussions on the proceeding testing scenario.

Testing Details for Section 4.9

This section contains testing details for the section Basic Resource Allocation Fairness.

Submit jobs in the following order:

Project	# GPUs	Total	Comment
team-d	2	6/8	Team-b/c workload pauses and moves to `pending`.
team-d	2	8/8	Other team (b/c) workloads pause and move to `pending`.

See the following executed command sequence:

```
$ runai submit d-2-gg -i gcr.io/run-ai-demo/quickstart -g 2 -p team-d$
runai submit d-3-gg -i gcr.io/run-ai-demo/quickstart -g 2 -p team-d
```

At this point, you should have the following states:

Project	GPUs Allocated	Workloads Queued
team-a	4/4	None
team-b	2/2	None
team-c	2/2	None
team-d	8/8	None

See the section Basic Resource Allocation Fairness for a discussion on the proceeding testing scenario.

Testing Details for Section 4.10

This section contains testing details for the section Over-Quota Fairness.

Submit jobs in the following order for `team-a`, `team-b`, and `team-c`:

Project	# GPUs	Total	Comment
team-a	2	4/4	1 workload queued
team-a	2	4/4	2 workloads queued
team-b	2	2/2	2 workloads queued
team-c	2	2/2	2 workloads queued

See the following executed command sequence:

```
$ runai submit a-3-gg -i gcr.io/run-ai-demo/quickstart -g 2 -p team-a$
runai submit a-4-gg -i gcr.io/run-ai-demo/quickstart -g 2 -p team-a$ runai
submit b-5-gg -i gcr.io/run-ai-demo/quickstart -g 2 -p team-b$ runai
submit c-6-gg -i gcr.io/run-ai-demo/quickstart -g 2 -p team-c
```

At this point, you should have the following states:

Project	GPUs Allocated	Workloads Queued
team-a	4/4	Two workloads asking for GPUs two each
team-b	2/2	Two workloads asking for two GPUs each
team-c	2/2	Two workloads asking for two GPUs each
team-d	8/8	None

Next, delete all the workloads for `team-d`:

```
$ runai delete -p team-d d-1-gggg d-2-gg d-3-gg
```

See the section Over-Quota Fairness, for discussions on the proceeding testing scenario.

Next: Where to Find Additional Information

Where to Find Additional Information

To learn more about the information that is described in this document, see the following resources:

- NVIDIA DGX Systems

- NVIDIA DGX-1 System
  https://www.nvidia.com/en-us/data-center/dgx-1/

- NVIDIA V100 Tensor Core GPU
  https://www.nvidia.com/en-us/data-center/tesla-v100/

- NVIDIA NGC
  https://www.nvidia.com/en-us/gpu-cloud/

• Run:AI container orchestration solution

- Run:AI product introduction
  https://docs.run.ai/home/components/

- Run:AI installation documentation
  https://docs.run.ai/Administrator/Cluster-Setup/Installing-Run-AI-on-an-on-premise-Kubernetes-Cluster/
  https://docs.run.ai/Administrator/Researcher-Setup/Installing-the-Run-AI-Command-Line-Interface/

- Submitting jobs in Run:AI CLI
  https://docs.run.ai/Researcher/Walkthroughs/Walkthrough-Launch-Unattended-Training-Workloads-/
  https://docs.run.ai/Researcher/Walkthroughs/Walkthrough-Start-and-Use-Interactive-Build-Workloads-/

- Allocating GPU fractions in Run:AI CLI
  https://docs.run.ai/Researcher/Walkthroughs/Walkthrough-Using-GPU-Fractions/

• NetApp AI Control Plane

- Technical report
  https://www.netapp.com/us/media/tr-4798.pdf

- Short-form demo
  https://youtu.be/gfr_sO27Rvo

- GitHub repository
  https://github.com/NetApp/kubeflow_jupyter_pipeline

• NetApp AFF systems

- NetApp AFF A-Series Datasheet
  https://www.netapp.com/us/media/ds-3582.pdf

- NetApp Flash Advantage for All Flash FAS
  https://www.netapp.com/us/media/ds-3733.pdf

- ONTAP 9 Information Library
  http://mysupport.netapp.com/documentation/productlibrary/index.html?productID=62286

- NetApp ONTAP FlexGroup Volumes technical report
  https://www.netapp.com/us/media/tr-4557.pdf

• NetApp ONTAP AI

- ONTAP AI with DGX-1 and Cisco Networking Design Guide
  https://www.netapp.com/us/media/nva-1121-design.pdf

- ONTAP AI with DGX-1 and Cisco Networking Deployment Guide
  https://www.netapp.com/us/media/nva-1121-deploy.pdf

- ONTAP AI with DGX-1 and Mellanox Networking Design Guide
  http://www.netapp.com/us/media/nva-1138-design.pdf

- ONTAP AI with DGX-2 Design Guide
  https://www.netapp.com/us/media/nva-1135-design.pdf