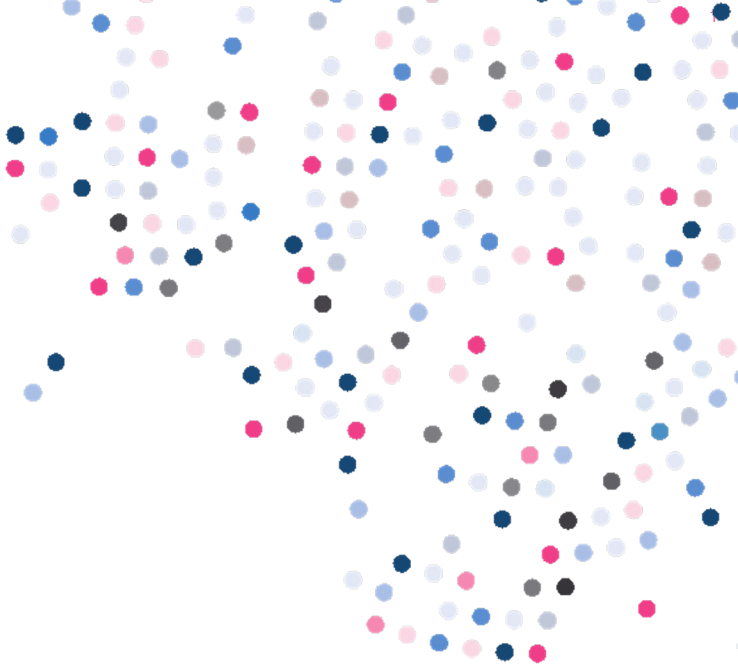
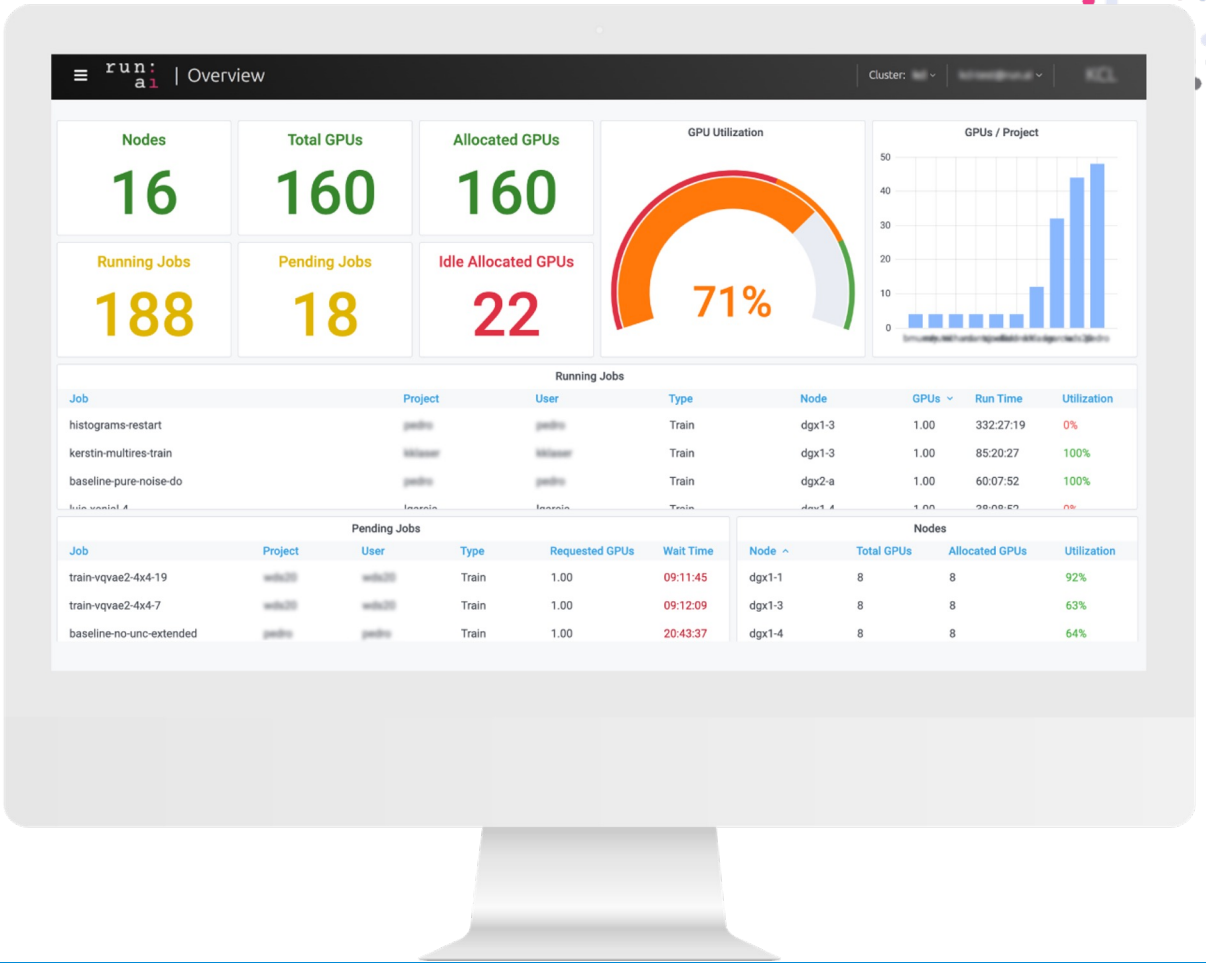


run: ai Researcher Training

Gain visibility and control over AI workloads to increase GPU utilization

Run:AI brings HPC capabilities to Kubernetes with batch scheduling and GPU virtualization, enabling seamless distributed training and full utilization of GPU resources.


run: ai | Overview
Cluster:
Nodes
16
Total GPUs
160
Allocated GPUs
160
Running Jobs
188
Pending Jobs
18
Idle Allocated GPUs
22
GPU Utilization
71%
GPUs / Project
50
40
30
20
10
0
Running Jobs
Job Project User Type Node GPUs Run Time Utilization
histograms-restart Train dgx1-3 1.00 332:27:19 0%
kerstin-multires-train Train dgx1-3 1.00 85:20:27 100%
baseline-pure-noise-do Train dgx2-a 1.00 60:07:52 100%
Pending Jobs
Job Project User Type Requested GPUs Wait Time
train-vqvae2-4x4-19 Train 1.00 09:11:45
train-vqvae2-4x4-7 Train 1.00 09:12:09
baseline-no-unc-extended Train 1.00 20:43:37
Nodes
Node Total GPUs Allocated GPUs Utilization
dgx1-1 8 8 92%
dgx1-3 8 8 63%
dgx1-4 8 8 64%

Concepts

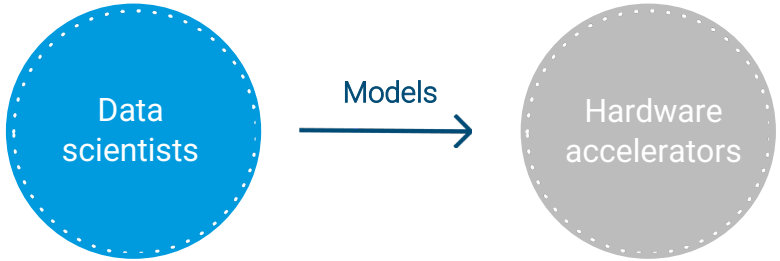
run:
ai

Basics & Assumptions
Locality → Global considerations

- At the heart of Run:AI is the premise that “optimization” requires finding the “right resources for your Job”. With this assumption in mind, the researcher is no longer **permanently** assigned a **local** machine.

- Instead, upon request, Run:AI will allocate resources on different machines according to your needs taking into account global considerations

The Run:Ai Vision
Full Hardware Abstraction



Basics & Assumptions
Containers & Images

- To be able to abstract the resource location, Run:AI uses docker images to instantiate containers on the right machine

- It is assumed that you are already familiar with docker images and are using them today.

Basics & Assumptions
Shared Storage

- As a researcher, you use data: training data, scripts, interim checkpoints, docker image, etc.
- To be able to abstract the resource location, your data must be stored in a location which is *shared by all machines in a uniform way*. You can no longer rely on data being stored on the local machine
- If not already there, as part of the Run:AI implementation, your IT department will make such a shared location available

Basic Run:AI Concepts
Projects

- As a researcher, each request for cluster resources should be accompanied by a reference project. Without a project, resources cannot be allocated
- Depending on your organization's preference, projects can be modeled as **individuals**, as **teams** of people (e.g. team-ny) or as actual **business** activities (e.g. ct-scan-2020)

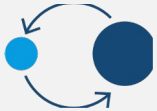
Basic Run:AI Concepts
Guaranteed Quotas

- Projects are assigned with a *guaranteed quota* of GPUs
- Projects can go over quota and consume more GPUs than assigned to them
- The Run:AI scheduler preempts and queues over-quota workloads when there are not enough resources to run under-quota workloads, taking into account fairness and priorities
- For more information on the Run:AI scheduler, including over-quota fairness, preemption, priorities, bin packing, elasticity and more, see here.

Basic Run:AI concepts:

Run:AI can schedule interactive “build” workloads, unattended “train” workloads and “inference” workloads

Build



- **Development & debugging**
- **Interactive** sessions
- **Short** cycles
- Performance is **less** important
- **Low** GPU utilization

Training

- **Model Training**
- **Remote, unattended** execution
- **Long** workloads
- **Throughput** is highly important
- **High** GPU utilization

Inference

- **Run Model in Production**
- **Services multiple users**
- **Typically low GPU memory requirements**
- **Performance is key**

Basic Run:AI Concepts
Build (Interactive)

- Build workloads are meant for interactive work.
- It is the responsibility of the researcher to stop a build workload.
- The Run:AI scheduler will usually not preempt a build workload with two notable exceptions:
  - The administrator has set a duration limit on interactive jobs for your project
  - The researcher has used the flag –preemptible
- Build workloads cannot extend beyond *guaranteed* quota (as they cannot be stopped automatically).
- See Quickstart of how to run a build job.

Basic Run:AI Concepts
Training

- Run:AI allows non-interactive training workloads to extend beyond *guaranteed* quotas and into *over-quota* as long as computing resources are available.
- To achieve this flexibility, the system needs to be able to safely stop a training workload and restart it again later. This means that:
    - The docker image should have an entrypoint instruction that initiates the training automatically upon restart.
    - Highly recommended: save ‘checkpoints’ frequently and allow the training to restart from the latest checkpoint
- See Quickstart on how to run a training job

Basic Run:AI Concepts
Inference

- Run:AI allows submitting inference workloads.
- Inference workloads are considered production workloads and thus take precedence over training workloads.
- It is the responsibility of the researcher to stop an inference workload.
- The Run:AI scheduler will usually not preempt a inference workloads.
- Inference workloads cannot extend beyond *guaranteed* quota (as they cannot be stopped automatically).

- See Quickstart on how to run a inference workload

Components

run:
ai

The Run:AI Components
The Command Line interface

Designed to be used by the researcher in order to do things like:

- Run and delete workloads
- View list of workloads and their status
- View list of available and allocated GPUs
- Access workloads via bash and view online logs
- Similar to Docker API

Docker Syntax

```
docker run --shm-size 16G -it --rm -e
HOSTNAME=`hostname` -v
/raid/public/my_datasets:/root/dataset
nvcr.io/nvidia/pytorch
```

Run:AI Syntax

```
runai submit myjob --large-shdm -e
HOSTNAME=`hostname` -v
/raid/public/my_datasets:/root/dataset -i
nvcr.io/nvidia/pytorch
```

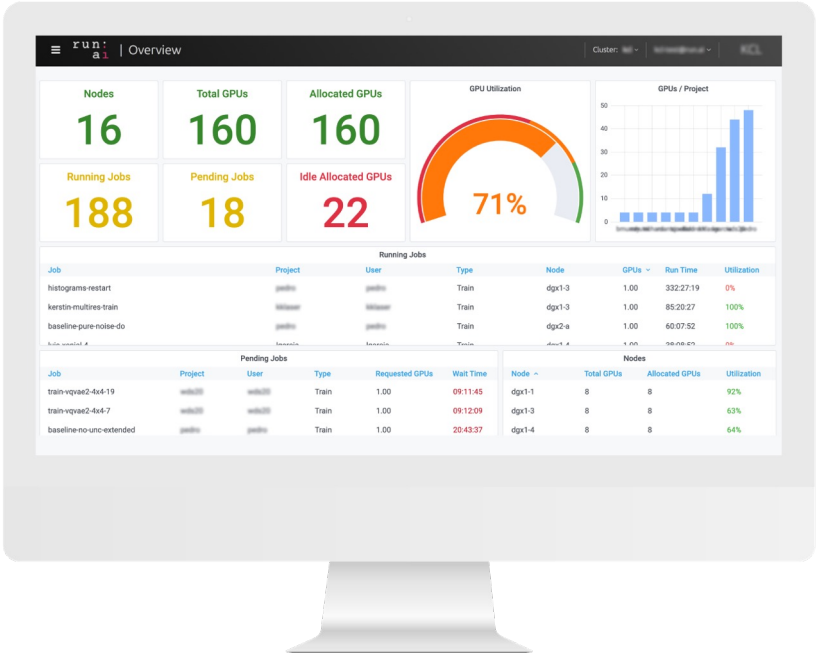
See Run:AI CLI reference

The Run:AI Components
The Run:AI Administrative User Interface

Designed to be used by IT and occasionally by researchers.
Goals are:

- Show holistic view of system resources (nodes, jobs etc). Both current status and long term status
- Allocate resources to researchers via projects
- Resolve conflicts

See: [app.run.ai](app.run.ai)



The Command Line Interface

run:
ai

Let's Try
Interactive Build

```
runai submit --help

runai config project team-a

Interactive Build job:
runai submit build1 -i ubuntu -g 1 --interactive --command -- sleep infinity

Get execution data on job:
runai describe job build1

Get shell:
runai bash build1

nvidia-smi

List all jobs:
runai list jobs

List delete job:
runai delete build1
```

Let's Try
Interactive Build- More Examples

Attach directly to job

```
runai submit build2 -i ubuntu -g 1 --interactive --attach
```

Forgo the name:

```
runai submit -i ubuntu -g 1 --interactive --attach
```

Interactive Build job with attached volume:

```
runai submit build3 -i  ubuntu -g 1 -v /mnt/nfs_share/files:/nfs --attach --interactive
```

Add a load balancing point (for details on Jupyter notebook see next slide):

```
runai submit jupyter2 -i my-image -g 1 --interactive --service-type=ingress --port 8888:8888
```

Many other options:

```
runai submit --help
```

Tooling

It is possible to invoke interactive containers with Run:AI and attach to them via the development tool of your choice.

Some tools support remote development:

- Visual Studio Code: https://docs.run.ai/Researcher/tools/dev-vscode/
- PyCharm: https://docs.run.ai/Researcher/tools/dev-pycharm/

The recommendation in these cases is to use ‘portforwarding’ to expose ports

Some tools, use container based web-development. Most notably jupyter notebooks.

- Jupyter: https://docs.run.ai/Researcher/tools/dev-jupyter/
- TensorBoard: https://docs.run.ai/Researcher/tools/dev-tensorboard/
- Others: https://docs.run.ai/Researcher/Walkthroughs/walkthrough-build-ports/

Let’s Try
Remote PyCharm / VSCode

Start a remote container with port-forwarding

```
runai submit build-remote -i gcr.io/run-ai-demo/pycharm-demo --interactive --service-type=portforward --port 2222:22 -v /mnt/nfs_share/remote:/workspace/mydir
```

Add a new PyCharm interpreter connecting to 127.0.0.1 (root/root)

Run and debug your code remotely

Let's Try
Training: Preparation

Under network file storage: (e.g. **/nfs/testuser**)

- Create a **requirements.txt** file with python dependencies
- Put your python code (e.g. **main.py**)
- Create a startup script (**startup.sh**) that runs both:

startup.sh

```
#!/bin/bash

pip install -r requirements.txt
python main.py
```

Let's Try
Training

Unattended job:

```
runai submit train1 -i tensorflow/tensorflow:1.14.0-gpu-py3 -g 1  -v
/mnt/nfs_share/john:/workspace/mydir --working-dir /workspace/mydir/ --command  -- ./startup.sh
```

Show logs:

```
runai logs train1 --follow
```

Unattended job with parameters

```
runai submit train2 -i tensorflow/tensorflow:1.14.0-gpu-py3  -g 1  -v
/mnt/nfs_share/john:/workspace/mydir --working-dir /workspace/mydir/ --command -- ./startup.sh
-e 'EPOCHS=30' -e 'LEARNING_RATE=0.02'
```

Minimal version:

```
runai submit -i gcr.io/run-ai-demo/quickstart -g 1
```

Let’s Try
Working with Projects

List of projects

```
runai list projects
```

Set a default project

```
runai config project team-b
```

Running on another project

```
Runai submit train-b -i gcr.io/run-ai-demo/quickstart -g 1 -p team-b
```

Listing all projects

```
Runai list jobs -A
```

GPU Fractions

- You can opt to allocate fractions of GPUs
- Run:AI provides memory isolation between multiple workloads using the same GPU
- Fractions can be interactive or none. If none, then all preemptions and consolidation rules apply.

Let's Try
Fractions

Allocate 1/2th of a GPU:

```
runai submit frac05 -i ubuntu --project team-a -g 0.5 --attach --interactive

nvidia-smi
```

Allocate more fractions to fill node:

```
runai submit frac03 -i ubuntu --project team-a -g 0.3 --attach --interactive
runai submit frac02 -i ubuntu --project team-a -g 0.2 --attach --interactive
```

Allocate fractions by memory size

```
runai submit fixed-gpu-mem -i ubuntu --project team-a --gpu-memory 3G --attach --interactive
```

Overquota & Bin-packing

See quickstart for overquota behavior and bin-packing
https://docs.run.ai/Researcher/Walkthroughs/walkthrough-overquota/

See quickstart for queue fairness:
https://docs.run.ai/Researcher/Walkthroughs/walkthrough-queue-fairness/

Hyperparameter Optimization

- Hyperparameter optimization (HPO) is the process of choosing a set of optimal hyperparameters for a learning algorithm (e.g. learning rate, batch size)
- To search for good hyperparameters, Researchers start a series of small runs with different hyperparameter values, let them run for a while, and then examine results to decide what works best.
- Run:AI automates the management and scheduling of HPO
- The Run:AI python Researcher library automates the experiment management and result gathering of HPO.
- See Quickstart for a step by step introduction

Let's Try
Hyperparameter Optimization

HPO run, 12 combinations, only 3 at a time:

```
runai submit hpo1 -i gcr.io/run-ai-demo/quickstart-hpo -g 1 --parallelism 3 --completions 12 -v
/mnt/nfs_share/files:/nfs
```

Watch

```
runai list jobs
runai describe job hpo1
```

Relate to a specific pod

```
runai logs hpo1 --pod <pod-name>
runai bash hpo1 --pod <pod-name>
```

Sample code for usage with Run:AI library

```
# import Run:AI HPO library
import runai.hpo

# select Random search or grid search
strategy = runai.hpo.Strategy.GridSearch

# initialize the Run:AI HPO library. Send the NFS directory used for sync
runai.hpo.init("/hpo")

# pick a configuration for this HPO experiment
# we pass the options of all hyperparameters we want to test
# `config` will hold a single value for each parameter
config = runai.hpo.pick(
grid=dict(
        batch_size=[32, 64, 128],
        lr=[1, 0.1, 0.01, 0.001]),
strategy=strategy)

....

# Use the selected configuration within your code
optimizer = keras.optimizers.SGD(lr=config['lr'])
```

Distributed Training

- The ability to split the training of a model among multiple processors (workers)
- Workers work in parallel, in different containers and perhaps in different nodes, to speed up model training
- Should not be confused with allocating multiple GPUs to a single container
- Distributed Training requires the user program to sync data and timing between different workers and there are a number of Deep Learning frameworks that support this. Horovod is a good example.
- Run:AI automates the management and scheduling of the workers and the communication links between them, providing the ability to easily run, manage and view Distributed Training.
- Run:AI employes gang-scheduling, in the sense that all worker-nodes are scheduled together and if one preempts, all others close down.
- See Quickstart for a step by step introduction

Let's Try
Distributed Training (MPI)

Unattended Distributed Training (2 workers):

```
runai submit-mpi dist1 --processes=3 -g 1 -i gcr.io/run-ai-demo/quickstart-distributed
```

Watch

```
runai list jobs
runai describe dist1
runai logs dist1
```

Interactive Distributed Training (for testing)

```
runai submit-mpi dist-int --processes=2 -g 1  -i gcr.io/run-ai-demo/quickstart-distributed  \
  --interactive --command -- sh -c sleep infinity
```

Bash and run

```
runai bash dist-int
horovodrun -np 2 python scripts/tf_cnn_benchmarks/tf_cnn_benchmarks.py --model=resnet20 --
num_batches=1000000 --data_name cifar10 --data_dir /cifar10 --batch_size=64 --
variable_update=horovod
```

Allocation of CPU & Memory

- GPUs are typically the most critical resource. But memory and cpu resources are no less important
- The *runai submit* command allows allocation of CPU & memory
- CPU *--cpu* command guarantees a quota. Over-allocation of CPU happens automatically if not requested by others
- Memory *--memory* command guarantees a quota. Over-allocation of Memory is possible, but you will receive an OOM exception if requested by others
- Without these flags, the system has defaults based on ratios to the number of requested GPUs

For information on how the Run:AI Scheduler allocates CPU & memory see here.

Let’s Try
Allocation of CPU & Memory

CPU & Memory guarantee

```
runai submit job1 -i ubuntu --gpu 2 --cpu 12 --memory 1G
```

CPU & Memory limits

```
runai submit job1 -i ubuntu --gpu 2 --cpu 12 --cpu-limit 24 --memory 1G --memory-limit 4G
```

(you will never receive more than 24 CPUs, you will get an OOM if you request more than 4GB)

Let’s Try
Other Useful Commands

Information about nodes

```
runai top node
```

Information about jobs

```
runai top job
```

Update CLI to latest version

```
sudo runai update
```

Inference

run:
ai

Inference Overview

- Machine learning (ML) inference is the process of running live data points into a machine-learning algorithm to calculate an output.
- Inference lends itself nicely to the usage of Run:AI Fractions. You can, for example, run 4 instances of an Inference server on a single GPU, each employing a fourth of the memory.
- Inference workloads can be submitted via Run:AI Command-line interface as well as Kubernetes API/YAML. Internally, spawning an Inference workload also creates a Kubernetes Service. The service is an endpoint to which clients can connect.

Let's Try
Basic Inference Commands

Submit an inference workload

```
runai submit --name inference1 --service-type nodeport --port 8888 --inference \
    -i gcr.io/run-ai-demo/quickstart-inference-marian  -g 1
```

Retrieve Host and Port information

```
runai list jobs
```

Connect to inference service

```
runai submit inference-client -i gcr.io/run-ai-demo/quickstart-inference-marian-client \
    -- --hostname <HOSTNAME> --port <PORT>  --processes 1
```

Let's Try
Inference with Fractions

Submit an inference workload. 4 Replicas on a single GPU

```
runai submit inference2 --service-type nodeport --port 8888 --inference \
    -i gcr.io/run-ai-demo/quickstart-inference-marian  --replicas 4 -g 0.25
```

Retrieve Host and Port information

```
runai list jobs
```

Connect to inference service

```
runai submit inference-client -i gcr.io/run-ai-demo/quickstart-inference-marian-client \
    -- --hostname <HOSTNAME> --port <PORT>  --processes 1
```

Let’s Try
Create an Inference Workload using YAML

Submit an inference workload. 1 Replica

```
kubectl apply -f https://raw.githubusercontent.com/run-ai/docs/master/examples/inference-single-replica.yaml
```

Submit an inference workload. 2 Replicas on a single GPU

```
kubectl apply -f https://raw.githubusercontent.com/run-ai/docs/master/examples/inference-two-replicas.yaml
```

Submit an inference (triton) workload. 1 Replicas on a single GPU

```
kubectl apply -f https://raw.githubusercontent.com/run-ai/docs/master/examples/inference-triton-one-replicas.yaml
```

Advanced

run:
ai

CLI Templates

The CLI has a templating mechanism that is useful for shortening the command line. There are two kinds of templates:

- Named templates (e.g. *runai submit…. --template batch_ops*)
- Default template. Command line parameters used when the --template flag is not used.

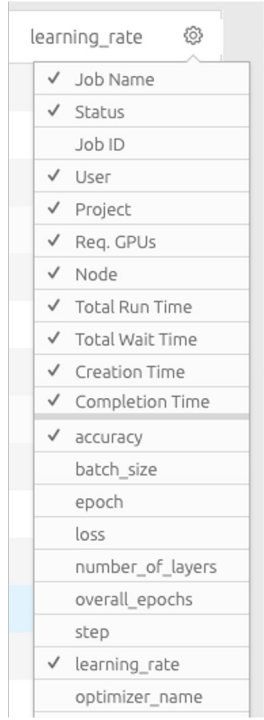
Templates are set by the IT administrator. For further information see: setting up templates

The Run:AI Components
Training Library (Optional)

Run:AI provides a python library which can optionally be installed within your docker image and activated during the deep learning session.

If installed, the library provides:

- Additional progress reporting and metrics (accuracy, loss, batch size, etc.) externalized from the training run and shown on the Run:AI UI as metrics and graphs
- Ability to dynamically stretch and compress jobs according to GPU availability.
- Hyperparameter optimization support



Additional Material

Researcher documentation exists here under the “Researcher” tab. Noteworthy documents:

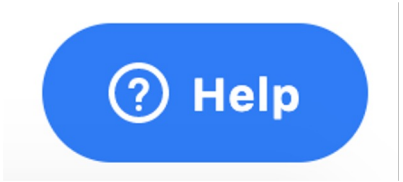
- CLI reference.
- Run:AI Scheduler internals
- Various Quickstart guides.
- Researcher Library.
- This presentation is here

How to Get Help

Write to support@run.ai

or

Use the “help” button at app.run.ai and docs.run.ai



Next Steps

- See document on how to ramp up new researchers. Including installation and configuration.
- Submit your first workload with Run:AI.
  - Start by submitting an ‘interactive’ build workload.
  - After which ssh to the container and run your code.

run:
ai

Thank you

Contact: support@run.ai