Hewlett Packard Enterprise

Document type name
DISCLOSURE STATEMENT

HPE Machine Learning Development Environment: Deployment Guide

Learn about the basics of HPE ML Machine Learning (MLDE) and how to install the platform within a Kubernetes cluster.

Contents

Terms and Definitions

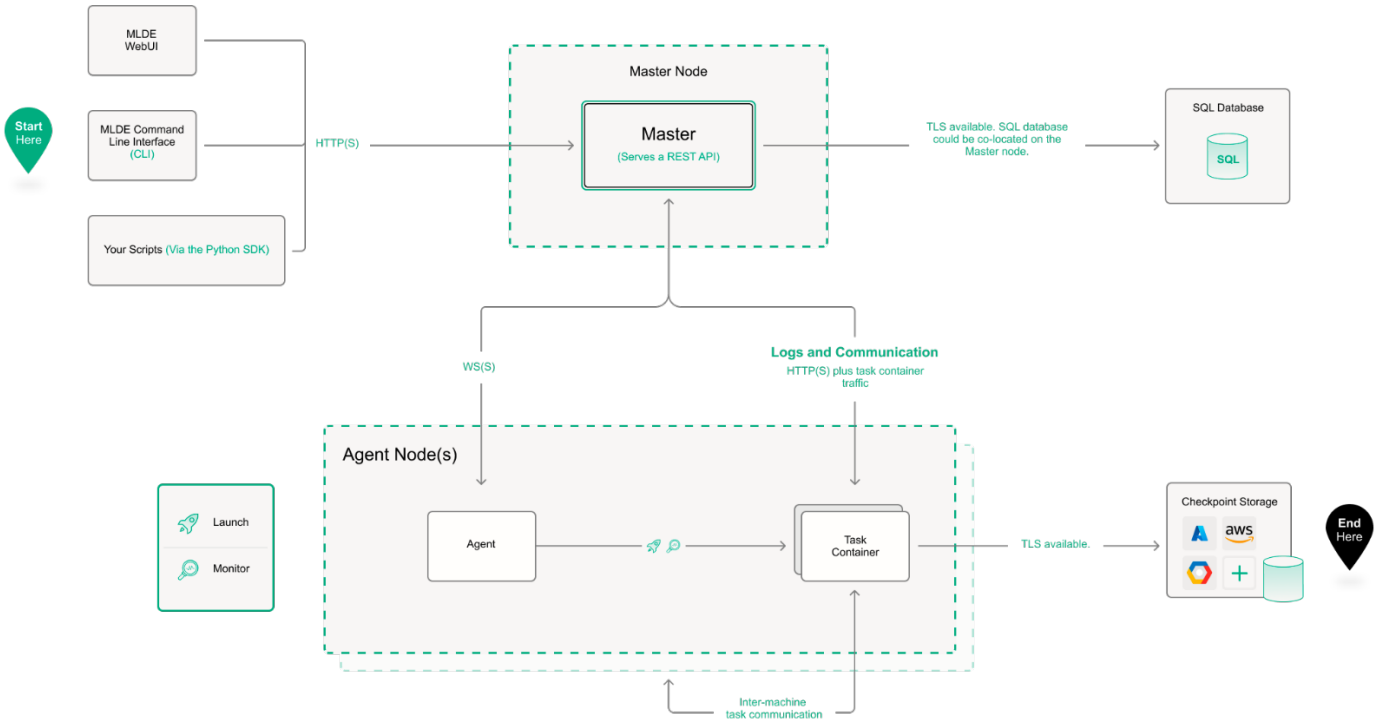
Term	Definition
HPE Machine Learning Development Environment	HPE Machine Learning Development Environment (MLDE) is built upon the open source Determined Training platform. For the latest documentation, visit the HPE Machine Learning Development documentation.
Determined	The open source Determined Training platform

Introduction to MLDE

With MLDE you can:

- Use state-of-the-art distributed training to train models faster without changing model code.
- Automatically find high-quality models using advanced hyperparameter tuning.
- Get more from your GPUs and reduce cloud GPU costs with preemptible instances and smart scheduling.
- Leverage experiment tracking out-of-the-box to track and reproduce your work, tracking code versions, metrics, checkpoints, and hyperparameters.
- Continue using popular deep learning libraries, such as TensorFlow, Keras, and PyTorch by simply integrating the Determined API with your existing model code.

MLDE integrates these features into an easy-to-use, high-performance deep learning environment so you can spend your time building models instead of managing infrastructure.



HPE MLDE System Architecture

Learn more:

- Intro to HPE Machine Learning Development Environment: Conceptual information about MLDE including its features and benefits.
- System Architecture: Learn about the main components of the MLDE system architecture.
- Distributed Training: A conceptual overview of distributed training with MLDE.

Key Features

Interactive Job Configuration

The behavior of interactive jobs, such as TensorBoards, notebooks, commands, and shells, can be influenced by setting a variety of configuration variables. These configuration variables are similar but not identical to the configuration options supported by experiments.

Configuration settings can be specified by passing a YAML configuration file when launching the workload via the Determined CLI.

Configuration variables can also be set directly on the command line when any Determined task, except a TensorBoard, is launched.

Options set via `--config` take precedence over values specified in the configuration file. Configuration settings are compatible with any Determined task unless otherwise specified.

MLDE CLI

One of the key components of the MLDE platform is the command-line interface (CLI). The CLI serves as a primary entry point for interacting with MLDE, providing a way to efficiently manage and control various aspects of the system. The following list describes some of the tasks you can perform with the Determined CLI:

- Experiment management: Running experiments is a fundamental part of the machine learning process. With the CLI, you can effortlessly create, list, and manage experiments, as well as access important experiment metrics and logs.

- Queue management: The CLI enables users to manage their job queues, monitor the progress of ongoing tasks, and even prioritize or cancel jobs as needed.
- Notebook management: Jupyter notebooks are an essential tool for data scientists and machine learning engineers. The CLI simplifies the process of creating, launching, and managing Jupyter notebooks within the platform.
- TensorBoard integration: TensorBoard is a popular visualization tool for TensorFlow projects. The CLI allows users to easily launch and manage TensorBoard instances, making it simple to visualize and analyze the training progress of their models.

Commands and Shells

Commands and shells support free-form tasks.

In the MLDE, a developer uses an experiment to run a trial. Outside of trials, a developer can use the det cmd Command (the capitalization indicates it is a specific feature of MLDE). This Command facilitates the execution of a user-defined program on the cluster. On the other hand, shells initiate SSH servers, enabling the interactive use of cluster resources.

Commands and shells enable developers to use a Determined cluster and its GPUs without having to write code conforming to the trial APIs. Commands are useful for running existing code in a batch manner; shells provide access to the cluster in the form of interactive SSH sessions.

Configuration Templates

At a typical organization, many MLDE configuration files will contain similar settings. There are two places a site can customize configuration, the master.yaml file and configuration templates. The values set in the master.yaml (in Kubernetes this is configured in the heml values.yaml file, see below Customizing a Pod) define the site default values, then groups or projects can utilize configuration template to help coordinate experimental data. For example, all of the training workloads run at a given organization might use the same checkpoint storage configuration. One way to reduce this redundancy is to use *configuration templates*. With this feature, you can move settings that are shared by many experiments into a single YAML file that can then be referenced by configurations that require those settings.

Each configuration template has a unique name and is stored by the Determined master. If a configuration specifies a template, the effective configuration of the task will be the result of merging the two YAML files (configuration file and template). The semantics of this merge operation is described under Configuration Templates: Merge Behavior. MLDE stores this expanded configuration so that future changes to a template will not affect the reproducibility of experiments that used a previous version of the configuration template.

A single configuration file can use at most one configuration template. A configuration template cannot use another configuration template.

Queue Management

The Determined Queue Management system extends scheduler functionality to offer better visibility and control over scheduling decisions. It does this using the Job Queue, which provides better information about job ordering, such as which jobs are queued, and permits dynamic job modification.

Queue Management is a new feature that is available to the fair share scheduler and the priority scheduler. Queue Management, described in detail in the following sections, shows all submitted jobs and their states and lets you modify some configuration options, such as priority, position in the queue, and resource pool.

To begin managing job queues, go to the WebUI Job Queue section or use the det job set of CLI commands.

Model Registry

The Model Registry is a way to group conceptually related checkpoints (including ones across different experiments), store metadata and long-form notes about a model, and retrieve the latest version of a model for use or further development. The Model Registry can be accessed through the WebUI, Python SDK, REST API, or CLI, though the WebUI has some features that the others are missing.

The Model Registry is a top-level option in the navigation bar. This will take you to a page listing all of the models that currently exist in the registry, and allow you to create new models. You can select any of the existing models to go to the Model Details page, where you can view and edit detailed information about the model. There will also be a list of every version associated with the selected model, and you can go to the Version Details page to view and edit that version's information.

For more information about how to use the model registry, see Organizing Models in the Model Registry

Notebooks

Jupyter Notebooks are a convenient way to develop and debug machine learning models, visualize the behavior of trained models, or even manage the training lifecycle of a model manually. MLDE makes it easy to launch and manage notebooks.

MLDE Notebooks have the following benefits:

- Jupyter Notebooks run in containerized environments on the cluster. We can easily manage dependencies using images and virtual environments. The HTTP requests are passed through the master proxy from and to the container.
- Jupyter Notebooks are automatically terminated if they are idle for a configurable duration to release resources. A notebook instance is considered to be idle if it is not receiving any HTTP traffic and it is not otherwise active (as defined by the notebook_idle_type option in the task configuration).

Note

- Once a Notebook is terminated, it is not possible to restore the files that are not stored in the persistent directories. **You need to ensure that the cluster is configured to mount persistent directories into the container and save files in the persistent directories in the container.** See Save and Restore Notebook State for more information.
- If you open a Notebook tab in JupyterLab, it will automatically open a kernel that will not be shut down automatically so you need to manually terminate the kernels.

TensorBoards

TensorBoard is a widely used tool for visualizing and inspecting deep learning models. MLDE makes it easy to use TensorBoard to examine a single experiment or to compare multiple experiments.

TensorBoard instances can be launched via the WebUI or the CLI. To launch TensorBoard instances from the CLI, first install the CLI on your development machine.

Benefits

MLDE is a deep learning training platform that simplifies infrastructure management for domain experts while enabling configuration-based deep learning functionality that engineering-oriented practitioners might find inconvenient to implement. The MLDE cohesive, end-to-end training platform provides best-in-class functionality for deep learning model training, including the following benefits:

Implementation	Benefit
Automated model tuning	Optimize models by searching through conventional hyperparameters or macro- architectures, using a variety of search algorithms. Hyperparameter searches are automatically parallelized across the accelerators in the cluster. See Hyperparameter Tuning.
Cluster-backed notebooks, commands, and shells	Leverage your shared cluster computing devices in a more versatile environment. See Jupyter Notebooks and Commands and Shells.
Cluster management	Automatically manage ML accelerators, such as GPUs, on-premise or in cloud VMs using your own environment, automatically scaling for your on-demand workloads. MLDE runs in either AWS or GCP, so you can switch easily according to your requirements. See Resource Pools, Scheduling, and Elastic Infrastructure.

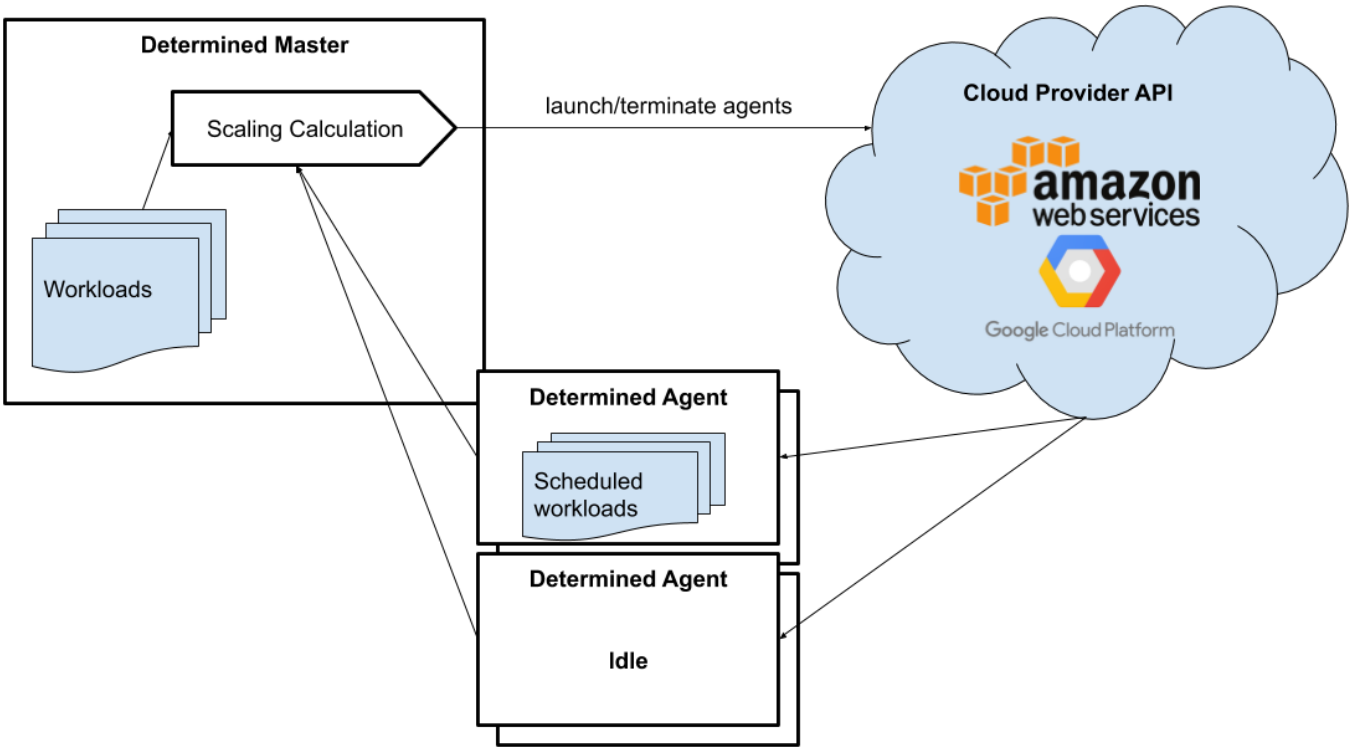
Containerization	Develop and train models in customizable containers that enable simple, consistent dependency management throughout the model development lifecycle. See Customizing Your Environment.
Distributed training	Easily distribute a single training job across multiple accelerators to speed up model training and reduce model development iteration time. MLDE uses synchronous, data-parallel distributed training, with key performance optimizations over other available options. See Distributed Training Concepts.
Experiment collaboration	Automatically track your experiment configuration and environment to facilitate reproducibility and collaboration among teams. See Submit Experiment.
Fault tolerance	Models are checkpointed throughout the training process and can be restarted from the latest checkpoint, automatically. This enables training jobs to automatically tolerate transient hardware or system issues in the cluster.
Framework support	Broad framework support leverages these capabilities using any of the leading machine learning frameworks without needing to manage a different cluster for each. Different frameworks for different models can be used without risking future lock-in. See Training APIs.
Profiling	Out-of-the-box system metrics (measurements of hardware usage) and timings (durations of actions taken during training, such as data loading).
Visualization	Visualize your model and training procedure by using The built-in WebUI and by launching managed Using TensorBoard instances.

Concepts

Elastic Infrastructure

When running in a cloud environment, MLDE can automatically provision and terminate GPU instances as the set of deep learning workloads on the cluster changes. This capability is called *elastic infrastructure*. The agents that are provisioned by the system are called *dynamic agents*.

The diagram below outlines the high-level system architecture when using dynamic agents:



Following the diagram, the execution would be:

1. The master collects information on idle agents (agents with no active workloads) and pending workloads (agents waiting to be scheduled).
2. The master calculates the ideal size of the cluster and decides how many agents to launch and which agents to terminate. The calculation is based on the total resource requests of all jobs submitted to the cluster, configured scaling behavior (minimum and maximum amount of instances the master can spawn), and the specification of the resource pools.
    a. An agent that is not running any containers is considered *idle*. By default, idle dynamic agents will automatically be terminated after 5 minutes of inactivity. This behavior gives agents a chance to run multiple workloads after they have been provisioned.
3. The master makes API calls to agent providers, such as AWS and GCP, to provision and terminate agents as necessary.
4. Once the agent instance has been created, it will automatically connect to the current master. The time it takes to create a new instance depends on the cloud provider and the configured instance type, but >60 seconds is typical.

Experiment

An *experiment* represents the basic unit of running the model training code. An experiment is a collection of one or more trials that are exploring a user-defined hyperparameter space. For example, during a learning rate hyperparameter search, an experiment might consist of three trials with learning rates of .001, .01, and .1.

To run experiments, you need to write your model training code. A *model definition* represents a specification of a deep learning model and its training procedure. It contains training code that implements training APIs. Visit the Training API Guides for more information.

For each experiment, you can configure a *searcher*, also known as a *search algorithm*. The search algorithm determines how many trials will be run for a particular experiment and how the hyperparameters will be set. More information can be found at Hyperparameter Tuning.

Resource Pools

To run tasks such as experiments or notebooks, MLDE needs to have resources (CPUs, GPUs) on which to run the tasks. However, different tasks have different resource requirements and, given the cost of GPU resources, it is important to choose the right resources for specific goals so that you get the most value out of your money. For example, you may want to run your training on beefy H100 GPU machines, while you want your TensorBoards to run on cheap CPU machines with minimal resources.

MLDE has the concept of a *resource pool*, which is a collection of identical resources that are located physically close to each other. MLDE allows you to configure your cluster to have multiple resource pools and to assign tasks to a specific resource pool so that you can use different sets of resources for different tasks. Each resource pool handles scheduling and instance provisioning independently.

When you configure a cluster, you set which pool is the default for auxiliary tasks and which pool is the default for compute tasks. CPU-only tasks such as TensorBoards will run on the default auxiliary pool unless you specify that they should run in a different pool when launching the task. Tasks that require a slot, such as experiments or GPU notebooks, will use the default compute pool unless otherwise specified. For this reason it is recommended that you always create a cluster with at least two pools, one with low-cost CPU instances for auxiliary tasks and one with GPU instances for compute tasks. This is the default setup when launching a cluster on AWS or GCP using det deploy.

Here are some scenarios where it can be valuable to use multiple resource pools:

- *Use GPU for training while using CPUs for TensorBoard.*

  You create one pool, aws-v100, that provisions p3dn.24xlarge instances (large V100 EC2 instances) and another pool, aws-cpu that provisions m5.large instances (small and cheap CPU instances). You train your experiments using the aws-v100 pool, while you run your TensorBoards in the aws-cpu pool. When your experiments complete, the aws-v100 pool can scale down to zero to save money, but you can continue to run your TensorBoard. Without resource pools, you would have needed to keep a p3dn.24xlarge instance running to keep the TensorBoard alive. By default TensorBoard will always run on the default CPU pool.

- *Use GPUs in different availability zones on AWS.*

  You have one pool aws-v100-us-east-1a that runs p3dn.24xlarge in the us-east-1a availability zone and another pool aws-v100-us-east-1b that runs p3dn.24xlarge instances in the us-east-1b availability zone. You can launch an experiment into aws-v100-us-east-1a and, if AWS does not have sufficient p3dn.24xlarge capacity in that availability zone, you can launch the experiment in aws-v100-us-east-1b to check if that availability zone has capacity. Note that the "AWS does not have capacity" notification is only visible in the master logs, not on the experiment itself.

- *Use spot/preemptible instances and fall back to on-demand if needed.*

  You have one pool aws-v100-spot that you use to try to run training on spot instances and another pool aws-v100-on-demand that you fall back to if AWS does not have enough spot capacity to run your job. MLDE will not switch from spot to on-demand instances automatically, but by configuring resource pools appropriately, it should be easy for users to select the appropriate pool depending on the job they want to run and the current availability of spot instances in the AWS region they are using. For more information on using spot instances, refer to Use Spot Instances.

- *Use cheaper GPUs for prototyping on small datasets and expensive GPUs for training on full datasets.*

  You have one pool with less expensive GPUs that you use for initial prototyping on small data sets and another pool that you use for training more mature models on large datasets.

Limitations

Currently resource pools are completely independent from each other so it is not possible to launch an experiment that tries to use one pool and then falls back to another one if a certain condition is met. You will need to manually decide to shift an experiment from one pool to another.

A cluster is not currently allowed to have resource pools in multiple AWS/GCP regions or across multiple cloud providers. If the master is running in one AWS/GCP region, all resource pools must also be in that AWS/GCP region.

If you create a task that needs slots and specify a pool that will never have slots (i.e. a pool with CPU-only instances), that task can never get scheduled. Currently that task will appear to be PENDING permanently.

Set up Resource Pools

Resource pools are configured using the master configuration. For each resource pool, you can configure scheduler and provider information.

If you are using static resource pools and launching agents by hand, you will need to update the agent configuration to specify which resource pool the agent should join.

Resource Pools

The resource_manager section is for cluster level setting such as which pools should be used by default and the default scheduler settings. The resource_pools section is a list of resource pools each of which has a name, description and resource pool level settings. Each resource pool can be configured with a provider field that contains the same information as the provisioner field in the legacy format. Each resource pool can also have a scheduler field that sets resource pool specific scheduler settings. If the scheduler field is not set for a specific resource pool, the default settings are used.

Note that defining resource pool-specific scheduler settings is all-or-nothing. If the pool-specific scheduler field is blank, all scheduler settings will be inherited from the settings defined in resource_manager.scheduler. If any fields are set in the pool-specific scheduler section, no settings will be inherited from resource_manager.scheduler - you need to redefine everything.

Here is an example master configuration illustrating the potential problem.

```
resource_manager:
  type: agent
  scheduler:
    type: round_robin
    fitting_policy: best
  default_aux_resource_pool: pool1
  default_compute_resource_pool: pool1

resource_pools:
  - pool_name: pool1
    scheduler:
      fitting_policy: worst
```

This example sets the cluster-wide scheduler defaults to use a best-fit, round robin scheduler in resource_manager.scheduler. The scheduler settings at the pool level for pool1 are then overwritten. Because scheduler.fitting_policy=worst is set, no settings are inherited from resource_manager.scheduler so pool1 uses a worst-fit, fair share scheduler because for a blank scheduler.type field, the default value is fair_share.

If you want to have pool1 use a worst-fit, round robin scheduler, you need to make sure you redefine the scheduler type at the pool-specific level:

```
resource_manager:
  type: agent
  scheduler:
    type: round_robin
    fitting_policy: best
  default_aux_resource_pool: pool1
  default_compute_resource_pool: pool1

resource_pools:
  - pool_name: pool1
    scheduler:
      type: round_robin
      fitting_policy: worst
```

Launch Tasks into Resource Pools

When creating a task, the job configuration file has a section called “resources”. You can set the resource_pool subfield to specify the resource_pool that a task should be launched into.

```
resources:
  resource_pool: pool1
```

If this field is not set, the task will be launched into one of the two default pools defined in the master configuration. Experiments will be launched into the default compute pool. TensorBoards will be launched into the default auxiliary pool. Commands, shells, and notebooks that request a slot (which is the default behavior if the resources.slots field is not set) will be launched into the default compute pool. Commands, shells, and notebooks that explicitly request 0 slots (for example the "Launch CPU-only Notebook" button in the WebUI) will use the auxiliary pool.

Scheduling

This document covers the supported scheduling policies. The first section describes the native scheduling capabilities supported by MLDE. The next section describes how MLDE schedules tasks when running on Kubernetes.

Native Scheduler

Administrators can configure the desired scheduler in master configuration file. It is also possible to configure different scheduling behavior for different resource pools.

Once the scheduling policy has been defined for the current master and/or resource pool, the scheduling behavior of an individual task is influenced by several task configuration values:

- For the fair-share scheduler, resources.weight lets users set the resource demand of a task relative to other tasks.
- For the priority scheduler, resources.priority lets users assign a priority order to tasks.
- Regardless of the scheduler, searcher.max_concurrent_trials lets users cap the number of slots that an adaptive_asha hyperparameter search experiment will request at any given time.

Note

Zero-slot tasks (e.g., CPU-only notebooks, TensorBoards) are scheduled independently of tasks that require slots (e.g., experiments, GPU notebooks). The fair-share scheduler schedules zero-slot tasks on a FIFO basis. The priority scheduler schedules zero-slot tasks based on priority.

Fair-Share Scheduler

The master allocates cluster resources (*slots*) among the active experiments using a weighted fair-share scheduling policy. This policy aims for fair distribution of resources, taking into account each experiment's request. More specifically, slots are divided among the active experiments according to the demand of each experiment, where *demand* is the number of desired concurrent slots.

For example, in an eight-GPU cluster running two experiments with demands of 10 and 30 respectively, the fair-share scheduler allocates one slot to the first experiment while the second experiment receives the remaining seven slots. As new experiments become active or the resource demand of an active experiment changes, the scheduler appropriately adjusts how slots are allocated to experiments.

You can modify the behavior of the fair-share scheduler by changing the *weight* of a workload. A workload demand for slots is multiplied by the workload weight for scheduling purposes. A workload with a higher weight will be assigned proportionally more resources than a workload with lower weight. The default weight is 1. For example, in the scenario above, if the weight of the first experiment is set to 3 and the weight of the second experiment is set to 1, each experiment will be assigned four slots.

Task Priority

The master allocates cluster resources (*slots*) to active tasks based on their *priority*. High-priority tasks are preferred to low-priority tasks. Low-priority tasks will be preempted to make space for pending high-priority tasks if possible. Tasks of equal priority are scheduled in the order in which they were created.

By default, the priority scheduler does not use preemption. If preemption is enabled in the master configuration file, when a higher priority task is pending and cannot be scheduled because no idle resources are available, the scheduler will attempt to schedule it by preempting lower priority tasks, starting with the task with the lowest priority. If there are no tasks to preempt, lower priority tasks might be backfilled on the idle resources. When a trial is preempted, its state is checkpointed so that the progress of the trial is not lost. Enabling preemption ensures that cluster resources can be reallocated to high priority tasks more promptly and backfilled to

make the most use of the idle resources; however, preemption can also result in additional overhead due to checkpointing low priority tasks, which might be expensive for some models.

Notebooks, TensorBoards, shells, and commands are not preemptible. These tasks will continue to occupy cluster resources until they complete or are terminated.

The priority of any task can be changed after it is created using one of the following commands:

```
det experiment set priority <ID> <priority>
det command set priority <ID> <priority>
det notebook set priority <ID> <priority>
det shell set priority <ID> <priority>
det tensorboard set priority <ID> <priority>
```

However, since only experiments are preemptible, changing the priority of any other kind of task after it is scheduled has no effect. (It can still be useful to change the priorities of such tasks before they are scheduled in order to affect when they ultimately start running.)

An example of priority scheduler behavior with preemption enabled:

1. User submits a priority 2 adaptive_asha experiment with max_concurrent_trials 20 and slots_per_trial 1. 8 trials run and utilize all 8 GPUs.
2. User submits a priority 1 distributed training experiment with slots_per_trial 4. 4 ASHA trials are preempted so the new distributed training experiment can run. Note that if preemption was not enabled, the new experiment would not get scheduled until the ASHA experiment GPU demand becomes <= 4.
3. User starts a priority 3 notebook with resources.slots 1. The notebook has a lower priority than the two active experiments, so it will run as soon as the two active experiments collectively need <= 7 GPUs.
4. ASHA and the distributed training experiment both complete, and the notebook task with priority 3 will run.
5. User submits a priority 1 distributed training experiment with slots_per_trial 8. Although this workload has a higher priority than the active notebook task, it cannot be scheduled because it requires 8 slots, notebooks are not preemptible, and therefore only 7 slots are available.
6. User submits a priority 2 distributed training experiment with slots_per_trial 4. One trial will be scheduled to make use of the idle 7 slots.
7. The notebook is killed. The priority 2 distributed training experiment is preempted. And then the priority 1 distributed training experiment starts running. Once that experiment is complete, distributed training experiment with priority 2 restarts.

The priority scheduler can be used with the Determined job queue, which provides more insight into scheduling decisions.

Scheduling with Kubernetes

When using MLDE on Kubernetes, Determined workloads, such as experiments, notebooks, and shells, are started by launching Kubernetes pods. The scheduling behavior that applies to those workloads depends on how the Kubernetes scheduler has been configured.

Gang Scheduling

By default, the Kubernetes scheduler does not perform gang scheduling or support preemption of pods. While it does take pod priority into account, it greedily schedules pods without consideration for the job each pod belongs to. This can result in problematic behavior for deep learning workloads, particularly for distributed training jobs that use many GPUs. A distributed training job that uses multiple pods requires all pods to be scheduled and running in order to make progress. Because Kubernetes does not support gang scheduling by default, cluster deadlocks can arise. For example, suppose that two experiments are launched simultaneously that each require 16 GPUs on a cluster with only 16 GPUs. It is possible that Kubernetes will assign some GPUs to one experiment and some GPUs to the other. Because neither experiment will receive the resources it needs to begin executing, the system will wait indefinitely.

One way MLDE addresses these problems is through the use of the lightweight coscheduling plugin, which extends the Kubernetes scheduler to support priority-based gang scheduling. To implement gang scheduling, the coscheduling plugin will not schedule a pod

unless there are enough available resources to also schedule the rest of the pods in the same job. To function, the plugin requires special labels to be set that specify the number of nodes that each job needs for execution. MLDE automatically calculates and sets these labels for GPU experiments that it launches.

The coscheduling plugin is in beta and is therefore not enabled by default. To enable it, edit values.yaml in the Determined Helm chart to set the defaultScheduler field to coscheduler.

There are several limitations to the coscheduling plugin to be aware of:

1. The coscheduling plugin does not work with Kubernetes' cluster autoscaling feature. Static node pools must be used to achieve gang scheduling
2. The plugin does not support preemption. For example, if the cluster is full of low priority jobs and a new high priority job is submitted, the high priority job will not be scheduled until one of the low priority jobs finishes.
3. The MLDE capability to automatically set pod labels is restricted to GPU experiments. MLDE does not currently set labels for CPU experiments or user commands.
4. When scheduling experiments that utilize the entire cluster, the plugin may take several minutes to schedule the next job. Because the coscheduler only approves of jobs when all of its pods are available, it may repeatedly reject partially-ready jobs, causing them to wait further.

To enable gang scheduling with commands or CPU experiments, enable the coscheduler in values.yaml and modify the experiment config to contain the following:

```
environment:
  pod_spec:
    metadata:
      labels:
        pod-group.scheduling.sigs.k8s.io/name: <unique task name>
        pod-group.scheduling.sigs.k8s.io/min-available: <# of GPUs required>
    spec:
      schedulerName: coscheduler
```

You can also use schedulerName: default-scheduler to use the default Kubernetes scheduler.

Additionally, please note that when running MLDE on Kubernetes, a higher priority value means a higher priority (e.g. a priority 50 task will run before a priority 40 task).

Priority Scheduling with Preemption

MLDE also makes available a priority scheduler that extends the Kubernetes scheduler to support preemption with backfilling. This plugin will preempt existing pods if higher priority pods are submitted. If there is still space in the cluster, backfilling will attempt to fill the nodes by scheduling lower priority jobs. Additionally, if there are leftover slots on partially-filled nodes, the scheduler will attempt to assign single-slot tasks until the space is filled. This packing behavior only occurs with single-slot tasks.

This plugin is also in beta and is not enabled by default. To enable it, edit values.yaml in the Determined Helm chart to set the defaultScheduler field to preemption. Autoscaling is not supported and MLDE can only automatically set labels for GPU experiments.

MLDE provides a default priority class, determined-medium-priority that has a priority of 50 and is used for all tasks. If users want to set a different priority level for an experiment, they may either specify a priority in the resources field of the experiment config or create a priorityClass and specify it in the pod_spec of the config. If both are specified, the specified priorityClass will take precedence over the priority field. In Kubernetes, a higher priority value means a higher priority (e.g. a priority 50 task will run before a priority 40 task).

Additionally, if using a cluster with tainted nodes or labels, users must specify the tolerations or node selectors in the pod_spec. It is recommended that you use both tolerations and node selectors to better constrain where your experiments can run, especially on clusters that contain multiple GPU types.

Below is an example that illustrates how to set priorities, tolerations, and node selectors.

```
resources:
  priority: 42 # priorityClass, if set, takes precedence over this value
```

```
environment:
  pod_spec:
    apiVersion: v1
    kind: Pod
    spec:
      priorityClassName: determined-medium-priority # don't set if using priority value
      nodeSelector:
        key: value
      tolerations:
      - key: "key1"
        operator: "Equal"
        value: "value"
        effect: "NoSchedule"
```

The Kubernetes priority scheduler can be used with the Determined job queue feature, which allows more insight into scheduling decisions.

Trial

A *trial* is a training task with a defined set of hyperparameters. A common degenerate case is an experiment with a single trial, which corresponds to training a single deep learning model.

Workspaces and Projects

Workspaces and **projects** provide a way to organize experiments. A project is a collection of experiments, and a workspace is a collection of projects. Learn more about workspaces and projects at Workspaces and Projects.

RBAC and User Groups

Role Based Access Control (RBAC) enables administrators to control user access to various actions and data within MLDE. RBAC feature requires the enterprise edition (MLDE) and is not available with the open source edition. Learn more about RBAC and User Group usage at RBAC.

YAML Configuration

YAML is a markup language often used for configuration. MLDE uses YAML for configuring tasks such as experiments and notebooks, as well as configuring the Determined cluster as a whole. This guide describes a subset of YAML that is recommended for use with MLDE. This is not a full description of YAML; see the specification or other online guides for more details.

YAML Types

A value in YAML can be a null or number, string, or Boolean scalar, or an array or map collection. Collections can contain other collections nested to any depth, although, the MLDE YAML files generally have a fixed structure.

A comment in a YAML file starts with a # character and extends to the end of the line.

If you are familiar with JSON, you can think of YAML as an alternative way of expressing JSON objects that is meant to be easier for humans to read and write, since it allows comments and has fewer markup characters around the content.

Maps

Maps represent unordered mappings from strings to YAML values. A map is written as a sequence of key-value pairs. Each key is followed by a colon and the corresponding value. The value can be on the same line as the key if it is a scalar (in which case it must be preceded by a space) or on subsequent lines (in which case it must be indented, conventionally by two spaces).

A map is used in the experiment configuration to configure hyperparameters:

```
hyperparameters:
  base_learning_rate: 0.001
  weight_cost: 0.0001
  global_batch_size: 64
  n_filters1: 40
  n_filters2: 40
```

The snippet above describes a map with one key, hyperparameters; the corresponding value is itself a map whose keys are base_learning_rate, weight_cost, etc.

Arrays

An array contains multiple other YAML values in some order. An array is written as a sequence of values, each one preceded by a hyphen and a space. The hyphens for one list must all be indented by the same amount.

An array is used in the experiment configuration to configure environment variables:

```
environment:
  environment_variables:
    - A=A
    - B=B
    - C=C
```

Scalars

Scalars generally behave naturally: null, true, 2.718, and "foo" all have the same meanings that they would in JSON (and many programming languages). However, YAML allows strings to be unquoted: foo is the same as "foo". This behavior is often convenient, but it can lead to unexpected behavior when small edits to a value change its type. For example, the following YAML block represents a list containing several values whose types are listed in the comments:

```
- true        # Boolean
- grue        # string

- 0.0         # number
- 0.0.        # string

- foo: bar    # map
- foo:bar     # string
- foo bar     # string
```

Example Experiment Configuration

A Determined configuration file consists of a YAML object with a particular structure: a map at the top level that is expected to have certain keys, with the value for each key expected to have a certain structure in turn.

In this example experiment configuration, numbers, strings, maps, and an array are demonstrated:

```
name: mnist_tf_const
data:
  base_url: https://s3-us-west-2.amazonaws.com/determined-ai-datasets/mnist/
  training_data: train-images-idx3-ubyte.gz
  training_labels: train-labels-idx1-ubyte.gz
  validation_set_size: 10000
hyperparameters:
  base_learning_rate: 0.001
  weight_cost: 0.0001
  global_batch_size: 64
  n_filters1: 40
  n_filters2: 40
searcher:
  name: single
  metric: error
  max_length:
    batches: 500
  smaller_is_better: true
environment:
  environment_variables:
    - A=A
    - B=B
```

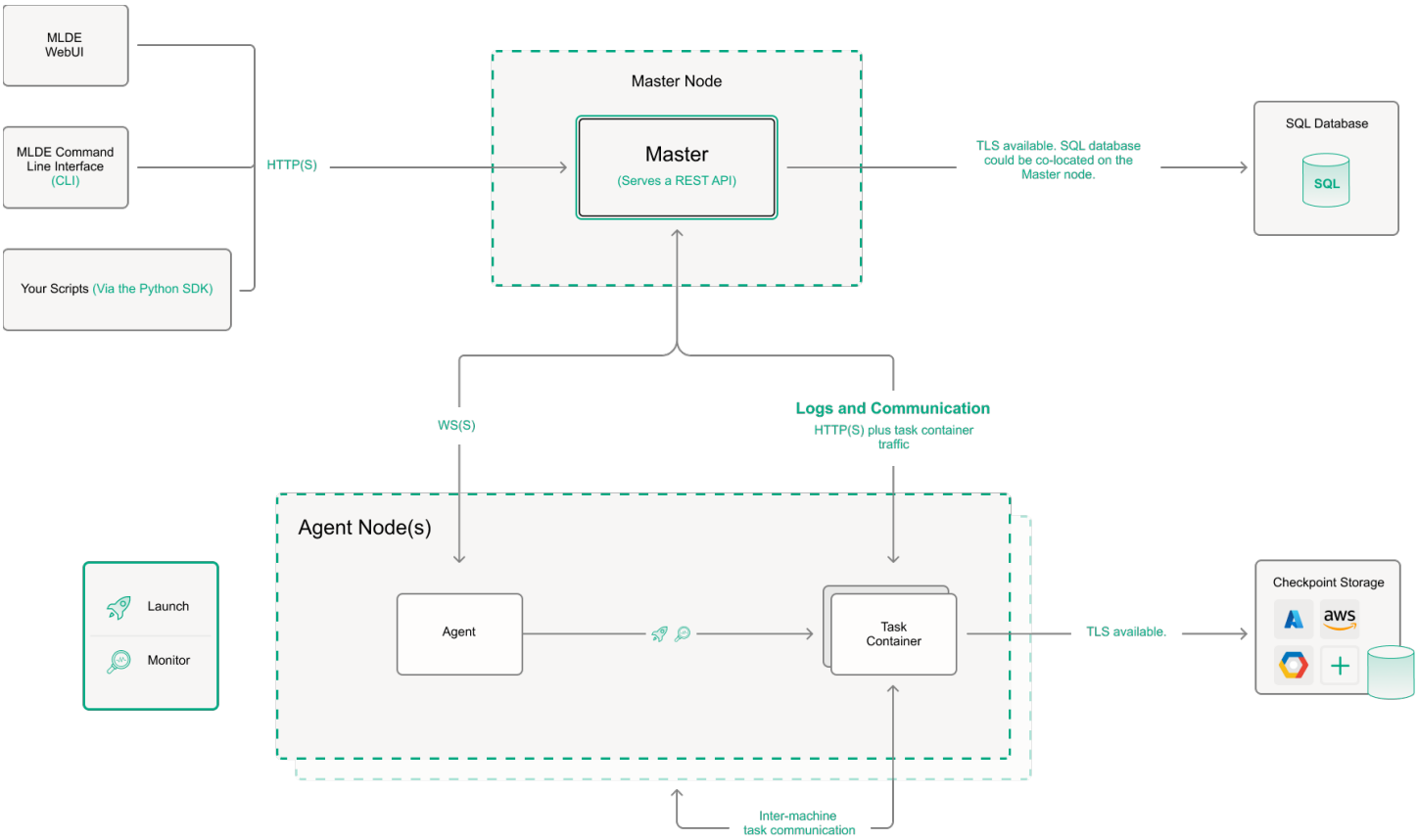
- C=C

Reference

- YAML: https://learnxinyminutes.com/docs/yaml/
- Validate YAML: http://www.yamllint.com/
- Convert YAML to JSON: https://www.json2yaml.com/convert-yaml-to-json

System Architecture

MLDE consists of a single **master** and one or more **agents**. There is typically one agent per compute server; a single machine can serve as both a master and an agent.



HPE MLDE System Architecture

The **master** is the central component of the MLDE system. It is responsible for

- Storing experiment, trial, and workload metadata.
- Scheduling and dispatching work to agents.
- Managing provisioning and deprovisioning of agents in clouds.
- Advancing the experiment, trial, and workload state machines over time.
- Hosting the WebUI and the REST API.

An **agent** manages a number of **slots**, which are computing devices (typically a GPU or CPU). An agent has no state and only communicates with the master. Each agent is responsible for

- Discovering local computing devices (slots) and sending metadata about them to the master.
- Running the workloads that are requested by the master.
- Monitoring containers and sending information about them to the master.

The **task container** runs a training task or other task(s) in a containerized environment. Training tasks are expected to have access to the data that will be used in training. The **agents** are responsible for reporting the status of the **task container** to the master.

Deploy on Kubernetes

This document describes how MLDE runs on Kubernetes. For instructions on installing MLDE on Kubernetes, see the installation guide.

In this topic guide, we will cover:

1. How MLDE works on Kubernetes.
2. Limitations of MLDE on Kubernetes.
3. Useful Helm and Kubectl commands.

How MLDEWorks on Kubernetes

Installing Determined on Kubernetes deploys an instance of the Determined master and a Postgres database in the Kubernetes cluster. Once the master is up and running, you can launch experiments, notebooks, TensorBoards, commands, and shells. When new workloads are submitted to the Determined master, the master launches pods and configMaps on the Kubernetes cluster to execute those workloads. Users of MLDE shouldn't need to interact with Kubernetes directly after installation, as MLDE handles all the necessary interaction with the Kubernetes cluster.

It is also important to note that when running MLDE on Kubernetes, a higher priority value means a higher priority (e.g. a priority 50 task will run before a priority 40 task). This is different from priority scheduling in non-Kubernetes deployments, where lower priority values mean a higher priority (e.g. a priority 40 task will run before a priority 50 task).

Limitations on Kubernetes

This section outlines the current limitations of MLDE on Kubernetes.

Scheduling

By default, the Kubernetes scheduler does not support gang scheduling or preemption. This can be problematic for distributed deep learning workloads that require multiple pods to be scheduled before execution starts. MLDE includes built-in support for the lightweight coscheduling plugin, which extends the default Kubernetes scheduler to support gang scheduling. MLDE also includes support for priority-based preemption scheduling. Neither are enabled by default. For more details and instructions on how to enable the coscheduling plugin, refer to Gang Scheduling and Priority Scheduling with Preemption.

Dynamic Agents

MLDE is not able to autoscale your cluster, but equivalent functionality is available by using the Kubernetes Cluster Autoscaler, which is supported on GKE and EKS.

Pod Security

By default, MLDE runs task containers as root. However, it is possible to associate a MLDE user with a Unix user and group, provided that the Unix user and group already exist. Tasks initiated by the associated MLDE user will run under the linked Unix user rather than root. For more information, see: Run Tasks as Specific Agent Users.

Useful Helm and Kubectl Commands

kubectl is a command-line tool for interacting with a Kubernetes cluster. Helm is used to install and upgrade MLDE on Kubernetes. This section covers some of the useful kubectl and helm commands when running Determined on Kubernetes.

For all the commands listed below, include -n <kubernetes namespace name> if running MLDE in a non-default namespace.

List Installations of MLDE

To list the current installation of MLDE on the Kubernetes cluster:

```
# To list in the current namespace.
helm list

# To list in all namespaces.
helm list -A
```

It is recommended to have just one instance of MLDE per Kubernetes cluster.

Get the IP Address of the Determined Master

To get the IP and port address of the Determined master:

```
# Get all services.
kubectl get services

# Get the master service. The exact name of the master service depends on
# the name given to your helm deployment, which can be looked up by running
# ``helm list``.
kubectl get service determined-master-service-<helm deployment name>
```

Check the Status of the Determined Master

Logs for the Determined master are available via the CLI and WebUI. Kubectl commands are useful for diagnosing any issues that arise during installation.

```
# Get all deployments.
kubectl get deployments

# Describe the current state of Determined master deployment. The exact name
# of the master deployment depends on the name given to your helm deploy
# which can be looked up by running `helm list`.
kubectl describe deployment determined-master-deployment-<helm deployment name>

# Get all pods associated with the Determined master deployment. Note this
# will only include pods that are running the Determined master, not pods
# that are running tasks associated with Determined workloads.
kubectl get pods -l=app=determined-master-<helm deployment name>

# Get logs for the pod running the Determined master.
kubectl logs <determined-master-pod-name>
```

Get All Running Task Pods

These kubectl commands list and delete pods which are running Determined tasks:

```
# Get all pods that are running Determined tasks.
kubectl get pods -l=determined

# Delete all Determined task pods. Users should never have to run this,
# unless they are removing a deployment of Determined.
kubectl get pods --no-headers=true -l=determined | awk '{print $1}' | xargs kubectl delete pod
```

Install MLDE on Kubernetes

Configuration Reference	Helm Chart Configuration Reference

This user guide describes how to install MLDE on Kubernetes using the `Determined Helm Chart`.

When the Determined Helm chart is installed, the following entities will be created:

- Deployment of the Determined master.
- ConfigMap containing configurations for the Determined master.
- LoadBalancer service to make the Determined master accessible. Later in this guide, we describe how it is possible to replace this with a NodePort service.
- ServiceAcccount which will be used by the Determined master.
- Deployment of a Postgres database. Later in this guide, we describe how an external database can be used instead.
- PersistentVolumeClaim for the Postgres database. Omitted if using an external database.
- Service to allow the Determined master to communicate with the Postgres database. Omitted if using an external database.

When installing Determined on Kubernetes using Helm, the deployment should be configured by editing the values.yaml and Chart.yaml files in the `Determined Helm Chart`.

Prerequisites

Before installing MLDE on a Kubernetes cluster, please ensure that the following prerequisites are satisfied:

- The Kubernetes cluster should be running Kubernetes version >= 1.19 and <= 1.21, though later versions may work.
- You should have access to the cluster via kubectl.
- Helm 3 should be installed.
- If you are using a private image registry or the enterprise edition, you should add a secret using kubectl create secret.
- The nodes in the cluster already have or can pull the fluent/fluent-bit:1.9.3 Docker image from Docker Hub.
- Optional: for GPU-based training, the Kubernetes cluster should have GPU support enabled.

You should also download a copy of the `Determined Helm Chart` and extract it on your local machine.

If you do not yet have a Kubernetes cluster deployed and you want to use MLDE in a public cloud environment, we recommend using a managed Kubernetes offering such as Google Kubernetes Engine (GKE) on GCP or Elastic Kubernetes Service (EKS) on AWS. For more info on configuring GKE for use with MLDE, refer to the Instructions for setting up a GKE cluster. For info on configuring EKS, refer to the Instructions for setting up an EKS cluster.

Configuration

When installing MLDE using Helm, first configure some aspects of the MLDE deployment by editing the values.yaml and Chart.yaml files in the Helm chart.

Image Registry Configuration

To configure which image registry of MLDE will be installed by the Helm chart, change imageRegistry in values.yaml. You can specify the Docker Hub public registry determinedai or specify any private registry that hosts the Determined master image.

Image Pull Secret Configuration

To configure which image pull secret will be used by the Helm chart, change imagePullSecretName in values.yaml. You can set it to empty for the Docker Hub public registry or specify any secret that is configured using kubectl create secret.

Version Configuration

To configure which version of MLDE will be installed by the Helm chart, change appVersion in Chart.yaml. You can specify a release version (e.g., 0.13.0) or specify any commit hash from the upstream Determined repo (e.g., b13461ed06f2fad339e179af8028d4575db71a81). You are strongly encouraged to use a released version.

Resource Configuration (GPU-based setups)

For GPU-based configurations, you must specify the number of GPUs on each node (for GPU-enabled nodes only). This is done by setting maxSlotsPerPod in values.yaml. MLDE uses this information when scheduling multi-GPU tasks. Each multi-GPU (distributed training) task will be scheduled as a set of slotsPerTask / maxSlotsPerPod separate pods, with each pod assigned up to maxSlotsPerPod GPUs. Distributed tasks with sizes that are not divisible by maxSlotsPerPod are never scheduled. If you have a cluster of different size nodes, set maxSlotsPerPod to the greatest common divisor of all the sizes. For example, if you have some nodes with 4 GPUs and other nodes with 8 GPUs, set maxSlotsPerPod to 4 so that all distributed experiments will launch with 4 GPUs per pod (with two pods on 8-GPU nodes).

Resource Configuration (CPU-based setups)

For CPU-only configurations, you need to set slotType: cpu as well as slotResourceRequests.cpu: <number of CPUs per slot> in values.yaml. Please note that the number of CPUs allocatable by Kubernetes may be lower than the number of "hardware" CPU cores. For example, an 8-core node may provide 7.91 CPUs, with the rest allocated for the Kubernetes system tasks. If slotResourceRequests.cpu was set to 8 in this example, the pods would fail to allocate, so it should be set to a lower number instead, such as 7.5.

Then, similarly to GPU-based configuration, maxSlotsPerPod needs to be set to the greatest common divisor of all the node sizes. For example, if you have 16-core nodes with 15 allocatable CPUs, it's reasonable to set maxSlotsPerPod: 1 and slotResourceRequests.cpu: 15. If you have some 32-core nodes and some 64-core nodes, and you want to use finer-grained slotResourceRequests.cpu: 15, set maxSlotsPerPod: 2.

Checkpoint Storage

Checkpoints and TensorBoard events can be configured to be stored in shared_fs, AWS S3, Microsoft Azure Blob Storage, or GCS. By default, checkpoints and TensorBoard events are stored using shared_fs, which creates a hostPath Volume and saves to the host file system. This configuration is intended for *initial testing only*; you are strongly discouraged from using shared_fs for actual deployments of MLDE on Kubernetes, because most Kubernetes cluster nodes do not have a shared file system.

Instead of using shared_fs, configure either AWS S3, Microsoft Azure Blob Storage, or GCS:

- **AWS S3**: To configure MLDE to use AWS S3 for checkpoint and TensorBoard storage, you need to set checkpointStorage.type in values.yaml to s3 and set checkpointStorage.bucket to the name of the bucket. The pods launched by the Determined master must have read, write, and delete access to the bucket. To enable this you can optionally configure checkpointStorage.accessKey and checkpointStorage.secretKey. You can optionally configure checkpointStorage.endpointUrl which specifies the endpoint to use for S3 clones (e.g., http://<minio-endpoint>:<minio-port|default=9000>).
- **Microsoft Azure Blob Storage**: To configure MLDE to use Microsoft Azure Blob Storage for checkpoint and TensorBoard storage, you need to set checkpointStorage.type in values.yaml to azure and set checkpointStorage.container to the name of the container to store it in. You must also specify one of connection_string - the connection string associated with the Azure Blob Storage service account to use, or the tuple account_url and credential - where account_url is the URL for the service account to use, and credential is an optional credential.
- **GCS**: To configure MLDE to use Google Cloud Storage for checkpoints and TensorBoard data, set checkpointStorage.type in values.yaml to gcs and set checkpointStorage.bucket to the name of the bucket. The pods launched by the Determined master must have read, write, and delete access to the bucket. For example, when launching GKE nodes you need to specify --scopes=storage-full to configure proper GCS access.

Default Pod Specs (Optional)

As described in the Deploy on Kubernetes guide, when tasks (e.g., experiments, notebooks) are started in a Determined cluster running on Kubernetes, the Determined master launches pods to execute these tasks. The Determined helm chart makes it possible to set default pod specs for all CPU and GPU tasks. The defaults can be defined in values.yaml under taskContainerDefaults.cpuPodSpec and taskContainerDefaults.gpuPodSpec. For examples of how to do this and a description of permissible fields, see the specifying custom pod specs guide.

Default Password (Optional)

Unless otherwise specified, the pre-existing users, admin and determined, do not have passwords associated with their accounts. You can set a default password for the determined and admin accounts if preferred or needed. This password will not affect any other user account. For additional information on managing users in MLDE, see the topic guide on users.

Database (Optional)

By default, the Helm chart deploys an instance of Postgres on the same Kubernetes cluster where MLDE is deployed. If this is not what you want, you can configure the Helm chart to use an external Postgres database by setting db.hostAddress to the IP address of their database. If db.hostAddress is configured, the Determined Helm chart will not deploy a database.

TLS (Optional)

By default, the Helm chart will deploy a load-balancer which makes the Determined master accessible over HTTP. To secure your cluster, MLDE supports configuring TLS encryption which can be configured to terminate inside a load-balancer or inside the Determined master itself. To configure TLS, set useNodePortForMaster to true. This will instruct MLDE to deploy a NodePort service for the master. You can then configure an Ingress that performs TLS termination in the load balancer and forwards plain text to the NodePort service, or forwards TLS encrypted data. Please note when configuring an Ingress that you need to have an Ingress controller running your cluster.

1. **TLS termination in a load-balancer (e.g., nginx).** This option will provide TLS encryption between the client and the load-balancer, with all communication inside the cluster performed via HTTP. To configure this option set useNodePortForMaster to true and then configure an Ingress service to perform TLS termination and forward the plain text traffic to the Determined master.
2. **TLS termination in the Determined master.** This option will provide TLS encryption inside the Kubernetes cluster. All communication with the master will be encrypted. Communication between task containers (distributed training) will not be encrypted. To configure this option create a Kubernetes TLS secret within the namespace where MLDE is being installed and set tlsSecret to be the name of this secret. You also need to set useNodePortForMaster to true. After the NodePort service is created, you can configure an Ingress to forward TLS encrypted data to the NodePort service.

An example of how to configure an Ingress, which will perform TLS termination in the load-balancer by default:

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: determined-ingress
  annotations:
    kubernetes.io/ingress.class: "nginx"

    # Uncommenting this option instructs the created load-balancer
    # to forward TLS encrypted data to the NodePort service and
    # perform TLS termination in the Determined master. In order
    # to configure ssl-passthrough, your nginx ingress controller
    # must be running with the --enable-ssl-passthrough option enabled.
    #
    # nginx.ingress.kubernetes.io/ssl-passthrough: "true"
spec:
  tls:
  - hosts:
    - your-hostname-for-determined.ai
    secretName: your-tls-secret-name
```

```
rules:
- host: your-hostname-for-determined.ai
  http:
    paths:
      - path: /
        backend:
          serviceName: determined-master-service-<name for your deployment>
          servicePort: masterPort configured in values.yaml
```

To see information about using AWS Load Balancer instead of nginx visit Using AWS Load Balancer.

Default Scheduler (Optional)

MLDE includes support for the lightweight coscheduling plugin, which extends the default Kubernetes scheduler to provide gang scheduling. This feature is currently in beta and is not enabled by default. To activate the plugin, set the defaultScheduler field to coscheduler. If the field is empty or doesn't exist, MLDE will use the default Kubernetes scheduler to schedule all experiments and tasks.

defaultScheduler: coscheduler

MLDE also includes support for priority-based scheduling with preemption. This feature allows experiments to be preempted if higher priority ones are submitted. This feature is also in beta and is not enabled by default. To activate priority-based preemption scheduling, set defaultScheduler to preemption.

defaultScheduler: preemption

Node Taints

Tainting nodes is optional, but you might want to taint nodes to restrict which nodes a pod may be scheduled onto. A taint consists of a taint type, tag, and effect.

When using a managed kubernetes cluster (e.g. a GKE, AKS, or EKS cluster), it is possible to specify taints at cluster or nodepool creation using the specified CLIs. Please refer to the set up pages for each managed cluster service for instructions on how to do so. To add taints to an existing resource, it is necessary to use kubectl. Tolerations can be added to Pods by including the tolerations field in the Pod specification.

kubectl Taints

To taint a node with kubectl, use kubectl taint nodes.

```
kubectl taint nodes ${NODE_NAME} ${TAINT_TYPE}=${TAINT_TAG}:${TAINT_EFFECT}
```

As an example, the following snippet taints nodes named node-1 to not be schedulable if the accelerator taint type has the gpu taint value.

```
kubectl taint nodes node-1 accelerator=gpu:NoSchedule
```

kubectl Tolerations

To specify a toleration, use the toleration field in the PodSpec.

```
tolerations:
  - key: "${TAINT_TYPE}"
    operator: "Equal"
    value: "${TAINT_TAG}"
    effect: "${TAINT_EFFECT}"
```

The following example is a toleration for when a node has the accelerator taint type equal to the gpu taint value.

```
tolerations:
  - key: "accelerator"
    operator: "Equal"
    value: "gpu"
    effect: "NoSchedule"
```

The next example is a toleration for when a node has the gpu taint type.

```
tolerations:
  - key: "gpu"
    operator: "Exists"
    effect: "NoSchedule"
```

Setting Up Multiple Resource Pools

To set up multiple resource pools for MLDE on your Kubernetes cluster:

1. Create a namespace for each resource pool. The default namespace can also be mapped to a resource pool.
2. As MLDE ensures that tasks in a given resource pool get launched in its linked namespace, the cluster admin needs to ensure that pods in a given namespace have the right nodeSelector or toleration automatically added to their pod spec so that they can be forced to be scheduled on the nodes that we want to be part of a given resource pool. This can be done using an admissions controller like a PodNodeSelector or PodTolerationRestriction. Alternatively, the cluster admin can also add a resource pool (and hence namespace) specific pod spec to the task_container_defaults sub-section of the resourcePools section of the Helm values.yaml:

```
resourcePools:
        pool_name: prod_pool
kubernetes_namespace: default
task_container_defaults:
gpu_pod_spec:
apiVersion: v1
kind: Pod
spec:
tolerations:
        key: "pool_taint"
operator: "Equal"
value: "prod"
effect: "NoSchedule"
```

3. Label/taint the appropriate nodes you want to include as part of each resource pool. For instance you may add a taint like kubectl taint nodes prod_node_name pool_taint=prod:NoSchedule and the appropriate toleration to the PodTolerationRestriction admissions controller or in resourcePools.pool_name.task_container_defaults.gpu_pod_spec as above so it is automatically added to the pod spec based on which namespace (and hence resource pool) a task runs in.
4. Add the appropriate resource pool name to namespace mappings in the resourcePools section of the values.yaml file in the Helm chart.

Install NVIDIA GPU Operator via Helm

To add the NVIDIA GPU Operator, run the following commands:

helm repo add nvidia && helm repo update

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

To validate the NVIDIA GPU Operator:

Create a configuration file to run a test cuda-vectoradd.yaml:

```
apiVersion: v1
kind: Pod
metadata:
  name: cuda-vectoradd
spec:
  restartPolicy: OnFailure
  containers:
  - name: cuda-vectoradd
    image: "nvcr.io/nvidia/k8s/cuda-sample:vectoradd-cuda11.7.1-ubuntu20.04"
    resources:
      limits:
        nvidia.com/gpu: 1
```

Run the pod

```
kubectl apply -f cuda-vectoradd.yaml
pod/cuda-vectoradd created
```

Check the logs

```
kubectl logs pod/cuda-vectoradd
[Vector addition of 50000 elements]
Copy input data from the host memory to the CUDA device
CUDA kernel launch with 196 blocks of 256 threads
Copy output data from the CUDA device to the host memory
Test PASSED
Done
```

Remove pod

```
kubectl delete -f cuda-vectoradd.yaml
pod "cuda-vectoradd" deleted
```

Useful Resources

- NVIDIA container toolkit installation guide
- Kubernetes: Scheduling GPUs
- Kubernetes: Scheduling GPUs with NVIDIA specifications

Options for RKE2

You'll first need to have a symbolic link for /sbin/ldconfig.real to /sbin/ldconfig and wait before the gpu-validator completes. This command line will connect to the RKE2 config and socket:

```
helm install --generate-name -n gpu-operator --create-namespace   nvidia/gpu-operator $HELM_OPTIONS --set
toolkit.env[0].name=CONTAINERD_CONFIG     --set toolkit.env[0].value=/var/lib/rancher/rke2/agent/etc/containerd/config.toml.tmpl     --set
toolkit.env[1].name=CONTAINERD_SOCKET     --set toolkit.env[1].value=/run/k3s/containerd/containerd.sock     --set
```

```
toolkit.env[2].name=CONTAINERD_RUNTIME_CLASS  --set toolkit.env[2].value=nvidia  --set
toolkit.env[3].name=CONTAINERD_SET_AS_DEFAULT  --set-string toolkit.env[3].value=true
```

Install MLDE

Once finished making configuration changes in values.yaml and Chart.yaml, MLDE is ready to be installed. To install MLDE, run:

```
helm install <name for your deployment> determined-helm-chart
```

determined-helm-chart is a relative path to where the `Determined Helm Chart` is located. It may take a few minutes for all resources to come up. If you encounter issues during installation, refer to the list of useful kubectl commands. Helm will install MLDE within the default namespace. If you wish to install MLDE into a non-default namespace, add -n <namespace name> to the command shown above.

Once the installation has completed, instructions will be displayed for discovering the IP address assigned to the Determined master. The IP address can also be discovered by running kubectl get services.

When installing MLDE on Kubernetes, I get an ImagePullBackOff error

You may be trying to install a non-released version of MLDE or a version in a private registry without the right secret. See the documentation on how to configure which version of Determined to install on Kubernetes.

Upgrade MLDE

To upgrade MLDE or to change a configuration setting, first make the appropriate changes in values.yaml and Chart.yaml, and then run:

```
helm upgrade <name for your deployment> --wait determined-helm-chart
```

Before upgrading MLDE, consider pausing all active experiments. Any experiments that are active when the Determined master restarts will resume training after the upgrade, but will be rolled back to their most recent checkpoint.

Uninstall MLDE

To uninstall MLDE run:

```
# Please note that if the Postgres Database was deployed by Determined, it will
# be deleted by this command, permanently removing all records of your experiments.
helm delete <name for your deployment>

# If there were any active tasks when uninstalling, this command will
# delete all of the leftover Kubernetes resources. It is recommended to
# pause all experiments prior to upgrading or uninstalling Determined.
kubectl get pods --no-headers=true -l=determined | awk '{print $1}' | xargs kubectl delete pod
```

Next Steps

Customize a Pod Development Guide Set up and Manage an Azure Kubernetes Service (AKS) Cluster Set up and Manage an AWS Kubernetes (EKS) Cluster Set up and Manage a Google Kubernetes Engine (GKE) Cluster

Customize a Pod

In a Determined cluster running on Kubernetes, tasks (e.g., experiments, notebooks) are executed by launching one or more Kubernetes pods. You can customize these pods by providing custom pod specs. Common use cases include assigning pods to specific nodes, specifying additional volume mounts, and attaching permissions. Configuring pod specs is not required to use MLDE on Kubernetes.

In this topic guide, we will cover:

1. How MLDE uses pod specs.

2. The different ways to configure custom pod specs.
3. Supported pod spec fields.
4. How to configuring default pod specs.
5. How to configuring per-task pods specs.

How MLDE Uses Pod Specs

All MLDE tasks are launched as pods. Determined pods consists of an initContainer named determined-init-container and a container named determined-container which executes the workload. When users provide a pod spec, MLDE inserts the determined-init-container and determined-container into the provided pod spec. As described below, users may also configure some of the fields for the determined-container.

Ways to Configure Pod Specs

MLDE provides two ways to configure pod specs. When MLDE is installed, the system administrator can configure pod specs that are used by default for all GPU and CPU tasks. In addition, you can specify a custom pod spec for individual tasks (e.g., for an experiment by specifying environment.pod_spec in the experiment configuration). If a custom pod spec is specified for a task, it overrides the default pod spec (if any).

Supported Pod Spec Fields

This section describes which fields users can and cannot configure when specifying custom pod specs.

MLDE does not currently support configuring:

- Pod Name - MLDE automatically assigns a name for every pod that is created.
- Pod Namespace - MLDE automatically sets the pod namespace based on the resource pool the task belongs to. The mapping between resource pools and namespaces can be configured in the resourcePools section of the Helm values.yaml.
- Host Networking - This must be configured via the master configuration.
- Restart Policy - This is always set to Never.

As part of their pod spec, users can specify initContainers and containers. Additionally users can configure the determined-container that executes the task (e.g., training), by setting the container name in the pod spec to determined-container. For the determined-container, MLDE currently supports configuring:

- Resource requests and limits (except GPU resources).
- Volume mounts and volumes.

Default Pod Specs

Default pod specs must be configured when installing or upgrading Determined. The default pod specs are configured in values.yaml of the Helm Chart Configuration Reference under taskContainerDefaults.cpuPodSpec and taskContainerDefaults.gpuPodSpec. The gpuPodSpec is applied to all tasks that use GPUs (e.g., experiments, notebooks). cpuPodSpec is applied to all tasks that only use CPUs (e.g., TensorBoards, CPU-only notebooks). Fields that are not specified will remain at their default Determined values.

Example of configuring default pod specs in values.yaml:

```
taskContainerDefaults:
  cpuPodSpec:
    apiVersion: v1
    kind: Pod
    metadata:
      labels:
        customLabel: cpu-label
```

```
  spec:
    containers:
      # Will be applied to the container executing the task.
      - name: determined-container
        volumeMounts:
          - name: example-volume
            mountPath: /example-data
      # Custom sidecar container.
      - name: sidecar-container
        image: alpine:latest
    volumes:
      - name: example-volume
        hostPath:
          path: /data
gpuPodSpec:
  apiVersion: v1
  kind: Pod
  metadata:
    labels:
      customLabel: gpu-label
  spec:
    containers:
      - name: determined-container
        volumeMounts:
          - name: example-volume
            mountPath: /example-data
    volumes:
      - name: example-volume
        hostPath:
          path: /data
```

The default pod specs can also be configured on a resource pool level. GPU jobs submitted in the resource pool will have the task spec applied. If a job is submitted in a resource pool with a matching CPU / GPU pod spec then the top level taskContainerDefaults.gpuPodSpec or taskContainerDefaults.cpuPodSpec will not be applied.

Example of configuring resource pool default pod spec in values.yaml.

```
resourcePools:
  - pool_name: prod_pool
    kubernetes_namespace: default
    task_container_defaults:
      gpu_pod_spec:
        apiVersion: v1
        kind: Pod
        spec:
          affinity:
            nodeAffinity:
              requiredDuringSchedulingIgnoredDuringExecution:
                nodeSelectorTerms:
                  - matchExpressions:
                      - key: topology.kubernetes.io/zone
                        operator: In
                        values:
                          - antarctica-west1
```

Per-task Pod Specs

In addition to default pod specs, it is also possible to configure custom pod specs for individual tasks. Pod specs for individual tasks can be configured under the environment field in the experiment config (for experiments) or the task configuration (for other tasks).

Example of configuring a pod spec for an individual task:

```
environment:
  pod_spec:
    apiVersion: v1
    kind: Pod
    metadata:
      labels:
        customLabel: task-specific-label
    spec:
      # Specify a pull secret for task container image.
      imagePullSecrets:
        - name: regcred
      # Specify a service account that allows writing checkpoints to S3 (for EKS).
      serviceAccountName: <checkpoint-storage-s3-bucket>
      # Specify tolerations for scheduling on tainted nodes.
      tolerations:
        - key: "tained-nodegroup-name"
          operator: "Equal"
          value: "true"
          effect: "NoSchedule"
```

When a custom pod spec is provided for a task, it will merge with the default pod spec (either resourcePools.task_container_defaults or top level task_container_defaults if resourcePools.task_container_defaults is not specified) according to Kubernetes strategic merge patch. MLDE does not support setting the strategic merge patch strategy, so the section titled “Use strategic merge patch to update a Deployment using the retainKeys strategy” in the linked Kubernetes docs will not work.

Some fields in pod specs are merged by values of items in lists. Volumes for example are merged by volume name. If for some reason you would want to remove a volume mount specific in the default task container you would need to override it with an empty volume of the same path.

Example values.yaml

```
resourcePools:
  - pool_name: prod_pool
    kubernetes_namespace: default
    task_container_defaults:
      gpu_pod_spec:
        apiVersion: v1
        kind: Pod
        spec:
          volumes:
            - name: secret-volume
              secret:
                secretName: prod-test-secret
          containers:
            - name: determined-container
              volumeMounts:
                - name: secret-volume
                  mountPath: /etc/secret-volume
```

Example expconf.yaml

```
environment:
  pod_spec:
    apiVersion: v1
    kind: Pod
    spec:
      volumes:
        - name: empty-dir-override
          emptyDir:
```

```
        sizeLimit: 100Mi
    containers:
      - name: determined-container
        volumeMounts:
          - name: empty-dir-override
            mountPath: /etc/secret-volume
resources:
  resource_pool: prod_pool
```

NVIDIA AI Enterprise Containers in MLDE

Update Registry Secrets

To allow MLDE to pull NVIDIA AI Enterprise container images from NGC, the NGC service account information needs to be added to the Kubernetes secret.

Create the secret object in the correct namespace for the compute node pool:

```
kubectl create secret --namespace=default docker-registry ngccred --docker-server=nvcr.io --docker-username="\$oauthtoken" --docker-password="<apikey>" --docker-email="."
```

Using det shell to run NVIDIA AI Enterprise Containers

Create configuration file

Create a configuration yaml file with the required NGC container and command line, similar to a standard Kubernetes pod file, but no pod name as MLDE will create the pod around the chosen container:

```
environment:
  environment_variables:
    - NCCL_DEBUG=TRACE
  pod_spec:
    apiVersion: v1
    kind: Pod
    spec:
      containers:
      - args:
        - cd /test/tf; LIGHTRUN=0 ./start.sh
        command:
        - /bin/bash
        - -c
        - --
        image: nvcr.io/nvaie/tensorflow-3-1:23.03-tf2-nvaie-3.1-py3
        name: nvcertjob
        resources:
          limits:
            nvidia.com/gpu: 0
        volumeMounts:
        - mountPath: /test
          name: nvaie-volume
        - mountPath: /dev/shm
          name: dshm
      imagePullSecrets:
      - name: ngccred
      restartPolicy: Never
      volumes:
      - name: nvaie-volume
        persistentVolumeClaim:
```

```
      claimName: a100-pvc
    - emptyDir:
      medium: Memory
    name: dshm
resources:
  resource_pool: A100
```

Start configured shell

```
det shell start --detach --config-file tf.yml
```

The above command will output the shell ID, this can be used to get information about the Kubernetes pod, as well as monitoring the logs. First need to find the full name of the Kubernetes podusing the shell ID from above.

```
kubectl get pods -A | grep <shellID>
```

Which will provide information about the pod, which can be used to monitor the pod logs with:

```
kubectl logs -n <namespace> <podname>
```

Or if there is need to interact with the container with the exec command:

```
kubectl exec -ti -n <namespace> <podname> --container <containername> -- /bin/sh
```

Custom Images

While the official images contain all the dependencies needed for basic deep learning workloads, many workloads have additional dependencies. If the extra dependencies are quick to install, you might consider using a startup hook. Where installing dependencies using startup-hook.sh takes too long, it is recommended that you build your own Docker image and publish to a Docker registry, such as Docker Hub.

Warning

Do NOT install TensorFlow, PyTorch, Horovod, or Apex packages, which conflict with Determined-installed packages.

It is recommended that custom images use one of the official Determined images as a base image, using the FROM instruction.

Example Dockerfile that installs custom conda-, pip-, and apt-based dependencies:

```
# Determined Image
FROM determinedai/environments:cuda-11.3-pytorch-1.12-tf-2.11-gpu-0.24.0

# Custom Configuration
RUN apt-get update && \
  DEBIAN_FRONTEND="noninteractive" apt-get -y install tzdata && \
  apt-get install -y unzip python-opencv graphviz
COPY environment.yml /tmp/environment.yml
COPY pip_requirements.txt /tmp/pip_requirements.txt
RUN conda env update --name base --file /tmp/environment.yml
RUN conda clean --all --force-pkgs-dirs --yes
RUN eval "$(conda shell.bash hook)" && \
  conda activate base && \
  pip install --requirement /tmp/pip_requirements.txt
```

Assuming that this image is published to a public repository on Docker Hub, use the following declaration format to configure an experiment, command, or notebook:

```
environment:
  image: "my-user-name/my-repo-name:my-tag"
```

where my-user-name is your Docker Hub user, my-repo-name is the name of the Docker Hub repository, and my-tag is the image tag to use, such as latest.

If you publish your image to a private Docker Hub repository, you can specify the credentials needed to access the repository:

```
environment:
  image: "my-user-name/my-repo-name:my-tag"
  registry_auth:
    username: my-user-name
    password: my-password
```

If you publish the image to a private Docker Registry, specify the registry path as part of the image field:

```
environment:
  image: "myregistry.local:5000/my-user-name/my-repo-name:my-tag"
```

Images are fetched using HTTPS by default. An HTTPS proxy can be configured using the https_proxy field in the agent configuration.

The custom image and credentials can be set as the defaults for all tasks launched in MLDE, using the image and registry_auth fields in the master configuration. Make sure to restart the master for this to take effect.