

Fine-tune on the HPC cluster, serve on Kubernetes

Hi, and thanks for stopping by. Meta-scheduling is one of my favourite parts of the Klique platform. Amongst many things, Klique is a meta scheduler: one control plane that schedules work across every environment a company runs. In this scenario we have a Slurm cluster and a Kubernetes cluster.

Klique can check every submitted task against the team's resource policy, whoever submitted it, from wherever, and through whichever surface: the UI, the SDK and CLI, the REST API, a CI/CD pipeline or an AI agent over MCP. It can then pick the pool that runs it from the compute resource's profile and pool order, and count the GPUs against reservations and limits. A Klique worker daemon in that pool then schedules the work as the cluster's own unit, a batch job on Slurm or a pod on Kubernetes. So one piece of Python code goes to either cluster just by picking a compute resource, under the same governance, and I get one place to send work, follow it and pick up the results.

Train where the GPUs are

The HPC cluster keeps its scheduler, queues and accounting

Serve where services live

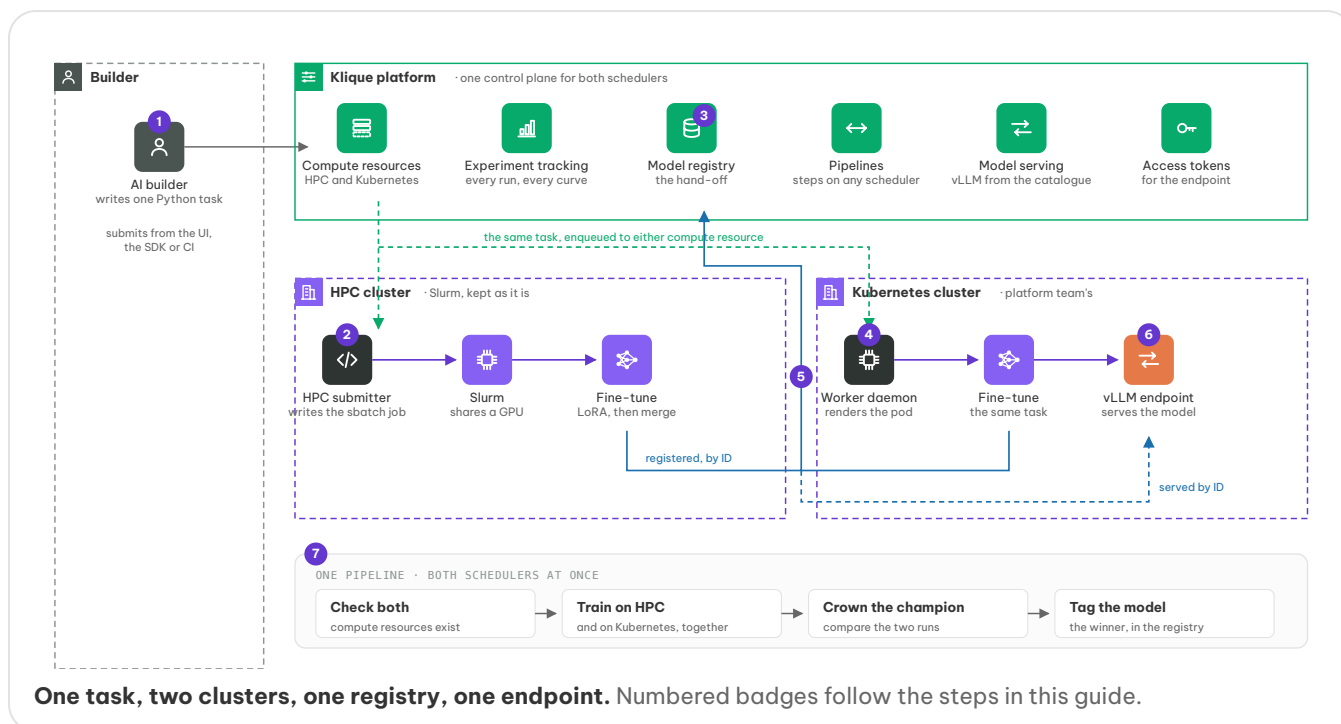
Kubernetes runs the endpoint behind an authenticated route

One task, either cluster

Choosing where it runs is choosing a compute resource

Hand over by ID

The model registry carries the model from one to the other



01

Why train on HPC and serve on Kubernetes

Here is what I wanted to do with it: fine-tune an open-weight model on an HPC cluster, then serve it on Kubernetes, without writing a batch script, a container manifest or a copy step in between. In this guide I explain why that split makes sense, step through the training code, and then walk through my own run in my Kliques environment: my clicks, the mechanics underneath and the result on screen. Your environment will name its compute resources differently, so at each step I point out which one to pick in yours.

Along the way I show a few ways of getting to the same end state: the Kliques UI, a script, a task that calls the endpoint and, at the end, one pipeline for the whole round. That is by way of example, to show the platform's capabilities; pick whichever suits the way you work.

In the real world I am often talking to customers with mixed estates: an HPC cluster that research has run for years, Kubernetes for services, sometimes a public cloud account on top. This is where Kliques excels, as one control plane across all of it.

Another conversation I have a lot is with customers who already have HPC and Kubernetes clusters, running on a mix of bare metal, some VMs and a cloud commit, and are looking at unification and management options, so that builders and IT managers no longer work with a separate submission surface for each. They want one place to submit work, one set of policies and one view of usage.

The list goes on, but the example below always seems to resonate: an AI team wants a model fine-tuned on its own data, and then answering requests from an internal application. The company already runs both kinds of infrastructure, and, as it happens, each one suits one half of that work.

Training belongs on the HPC cluster

The HPC cluster has the densest GPU nodes, fast interconnect between them and storage close to the data. Its batch queues, fair share and accounting are tuned for long, heavy runs, and the capacity is already paid for. A fine-tune is that kind of run, and it can grow to many nodes later without moving.

Serving belongs on Kubernetes

An endpoint is a long-running service, not a batch run. Kubernetes keeps it up, restarts it, rolls out new versions and adds replicas, and it sits next to the applications that call it, behind the platform team's access control.

The hand-off is where it usually breaks

Moving a model between the two normally means a batch script on one side, a container and a manifest on the other, a login on each cluster and a copy of the weights in between. Each step is small; together they slow every model down.

Where meta-scheduling comes in

With Klique, one Python task runs on either cluster by choosing a compute resource. Klique worker daemons schedule the work on each side, the model registry carries the result across, and a serving app on Kubernetes pulls the model by ID. Same code, same tracking, whichever cluster runs it.

My goal: use meta-scheduling to fine-tune an open-weight chat model on the HPC cluster, run the same task on Kubernetes to compare, serve the model on Kubernetes, call it, and then make the whole round repeatable from one pipeline.

My setup

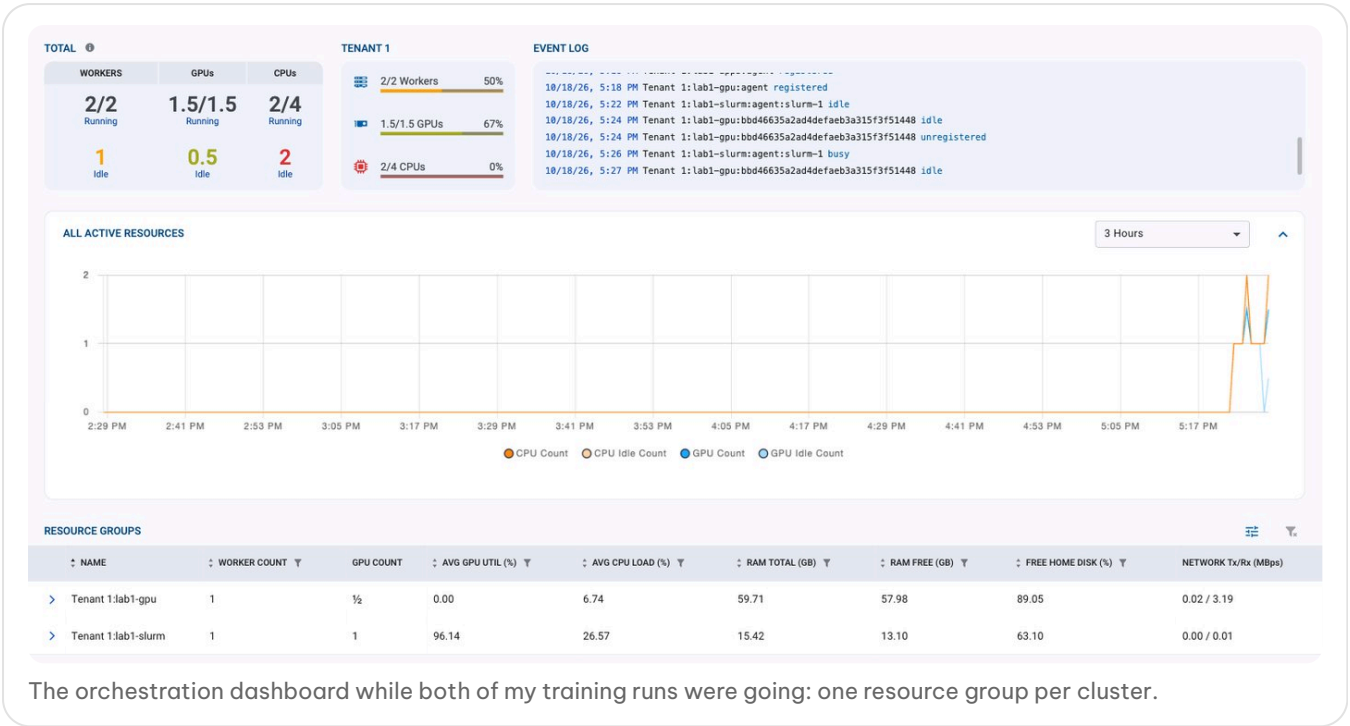
My Klique environment is connected to two clusters: a Slurm cluster and a Kubernetes cluster with GPU nodes. Each cluster has a Klique worker daemon registered against it. On the Slurm cluster that is the HPC submitter, running on the login node; on Kubernetes it is a worker daemon inside the cluster. Each worker daemon serves a set of compute resources, and my platform team has granted my team access to them through resource policies: one for training on Slurm, one for training on Kubernetes and one for serving.

That gives me four compute resources to work with, and each one lets me do one part of this build:

COMPUTE RESOURCE IN MY SETUP	WHERE THE WORK RUNS	WHAT I USE IT FOR
SLURM Half GPU (train)	The Slurm cluster, half a GPU through Slurm GPU sharding	The fine-tune
Half GPU (train, K8s)	The Kubernetes cluster, half a GPU	The same fine-tune, to compare the two clusters
Half GPU (serve)	The Kubernetes cluster, half a GPU	The vLLM endpoint that serves the model
1 CPU Slot	The Kubernetes cluster, CPU only	The pipeline controller, its comparison step and calling the endpoint

This is meta-scheduling in practice. I choose the compute resource; my platform team decides, through the policies and the pools behind each compute resource, which cluster runs the work and how much of it my team can use. If the platform team later adds a third cluster, or a cloud pool for busy days, my code stays the same.

So here is the plan. I fine-tune on the HPC cluster by sending a task to SLURM Half GPU (train). I send a copy of the same task to Half GPU (train, K8s) to compare. I serve the better model through Half GPU (serve), call it, and finally put both training runs into one pipeline that runs on 1 CPU Slot. Both clusters show up in the orchestration dashboard as their own resource group, so I can see GPU use on each side by side:



In your setup: the names will differ. Look for a compute resource that sends work to your HPC cluster, one with GPUs on Kubernetes for training, one your team uses for serving and a CPU one. If one of them is missing from your list, ask your platform team for access; a compute resource appears once a worker daemon serves it and a policy grants it to your team.

Setting it up in your environment

If you are the one connecting the clusters, or want to know what your platform team did, these pages in the Klisque documentation cover each part of my setup:

T0	KLIQUE DOCUMENTATION
Register a Klisque worker daemon in a Kubernetes cluster	Klique for Kubernetes
Run the HPC submitter on a Slurm or PBS login node	Klique for HPC
Turn a task into a batch job from a compute resource's template	HPC in the Orchestration Engine
Understand what a worker daemon does with queued tasks	The Orchestrator
Grant a team access to compute resources, with reservations and limits	Resource policies
Offer half or quarter GPUs as compute resources	Fractional GPUs

Tasks? What is a task?

Everything I send to a compute resource is a **task**. A task is one recorded run of some code: where the code lives in git and at which commit, the Python packages it needs, its parameters, and, once it runs, its console output, metrics, plots and the models it produces. I can create a task from a git repository in the Klisque UI, or from my laptop by running a script that calls the Klisque SDK. Once a

task exists, I can clone it, change a parameter and send the clone to any compute resource I have access to. That is how the same training code ends up on the HPC cluster and on Kubernetes without a change.

Models and serving

When a task produces a model, it registers it in the Klique model registry, with its weights in shared storage that both clusters can read. To serve it, I use the vLLM Model Deployment app from the Klique app catalogue: it takes the model's ID, pulls the weights and publishes an OpenAI-compatible endpoint behind an authenticated route. The [Meta-Scheduler Reference Architecture](#) covers the platform side in more depth.

Stepping through train.py

So, here's the code. For this build I created some example training code and put it on GitHub, in the [hands-on-labs repository](#), so you can follow along with exactly what I ran. It is general purpose on purpose: an open-weight base model (Qwen2.5-0.5B-Instruct by default), a public instruction dataset and settings that can all be changed from the command line or in the Klique UI. The places to change for your own environment are marked ADAPT.

FILE	WHAT IT DOES
train.py	Fine-tunes the base model with LoRA, sends the loss curves to Klique, measures loss on a held-out split, merges the adapter and registers the model
pipeline.py	Runs train.py on two compute resources at the same time and tags the model with the lower held-out loss as the champion
call_endpoint.py	Calls the served model through its OpenAI-compatible endpoint, from a laptop or as a Klique task
requirements.txt	The packages, with the versions I pinned to match the serving image

To grab it:

```
git clone https://github.com/damianerangey-cml/hands-on-labs.git
cd hands-on-labs/train-hpc-serve-k8s
```

Most of train.py is ordinary Hugging Face code; the Klique parts are a handful of lines. Before running anything, let's step through them in the order they run. The same file runs on a laptop, on the HPC cluster and on Kubernetes; only where it is sent changes.

1. Create the task

```
from clearml import OutputModel, Task

PROJECT = "Builders - Train on Slurm, Serve on Kubernetes" # ADAPT: your Klique project

task = Task.init(project_name=PROJECT, task_name="Fine-tune a chat model",
                 task_type=Task.TaskTypes.training)
```

Task.init creates the task in my project. From this line on, Klique records the code execution information, the git repository, commit and any uncommitted changes, and the Python packages installed, and it captures the output: the console, scalars, plots and models.

2. Arguments become hyperparameters

```
DEFAULTS = {"base_model": "Qwen/Qwen2.5-0.5B-Instruct", "dataset": "yahma/alpaca-cleaned",
            "max_samples": 1000, "epochs": 1, "learning_rate": 2e-4, "lora_r": 16, ...}

def parse_args():
    p = argparse.ArgumentParser()
    for key, val in DEFAULTS.items():
        p.add_argument(f"--{key}", type=type(val), default=val)
    p.add_argument("--queue", default="")
    return p.parse_args()

args = parse_args()          # parsed after Task.init
```

Klique connects to argparse automatically, so every argument appears on the task under Configuration > Args. Parsing after Task.init matters: when I clone the task in the UI and edit, say, the number of epochs, the run picks up my edited value instead of the default.

3. Decide where the outputs go

```
task.output_uri = True      # ADAPT: or "s3://bucket/path", "gs://...", "azure://..."
```

Models the task registers are uploaded here. True means the platform's default storage. Shared storage is what lets a model trained on the HPC cluster be served from Kubernetes.

4. Send the task to a compute resource

```
if args.queue:
    task.execute_remotely(queue_name=args.queue)
```

When I run the file on my laptop with `--queue`, this line stops the local run and sends the task to that compute resource. When the file is already running under a worker daemon, the call has no effect, so the script carries straight on into training.

5. Load the data and the model

```
rows = load_rows(args)          # ADAPT: (prompt, response) pairs from your data
tok = AutoTokenizer.from_pretrained(args.base_model)
model = AutoModelForCausalLM.from_pretrained(args.base_model, dtype=dtype)
model = get_peft_model(model, LoraConfig(r=args.lora_r, lora_alpha=args.lora_alpha,
                                         task_type="CAUSAL_LM", target_modules="all-linear"))
```

Plain Hugging Face code. `load_rows` reads a dataset from the Hub, or a local JSON Lines file, and turns each row into a prompt and a response, formatted with the model's own chat template. LoRA trains a small adapter instead of every weight, which is why a fine-tune of a small model fits on part of one GPU. `dtype` is `bf16` where the GPU supports it and `fp32` otherwise.

6. Report while it trains

```
logger = task.get_logger()

class KliqueScalars(TrainerCallback):
    SKIP = ("epoch", "total_flos") # bookkeeping, not curves
    def on_log(self, a, state, control, logs=None, **kw):
        for key, val in (logs or {}).items():
            if isinstance(val, (int, float)) and key not in self.SKIP and ...:
                logger.report_scalar(title=key, series="train", value=float(val),
                                    iteration=int(state.global_step))
```

Each value the Trainer logs, the loss, the learning rate and the gradient norm, goes to its own chart on the task's Scalars tab as the run goes. I can follow the curves in the Klique UI while the task is on the cluster's GPUs, with no login to the cluster.

7. Measure, merge and register the model

```
eval_loss = float(trainer.evaluate()["eval_loss"])
logger.report_single_value("eval_loss", round(eval_loss, 4))

merged = model.merge_and_unload()
merged.save_pretrained("merged_model"); tok.save_pretrained("merged_model")

out = OutputModel(task=task, name=args.model_name, framework="PyTorch")
out.update_weights_package(weights_path="merged_model", auto_delete_file=False)
print("Registered model ID:", out.id)
```

The held-out loss becomes one summary number on the task, the number my pipeline later compares runs on. Merging folds the adapter back into the base model, so the serving side loads ordinary weights. `OutputModel` registers the folder as a model in the Klique model registry, uploads it to the storage from block 3 and links it to this task. Its ID is all the serving side needs.

One way of doing it. I drove this build from the Klique UI and a script, which makes every step visible. You can take the same steps from a pipeline or a CI pipeline, or hand them to an agent: a coding agent launched from the app catalogue can clone the task, send it to each compute resource, compare the runs and launch the serving app through the Klique MCP server, with your own permissions.

What I did, step by step

Each step has three parts: my action, the mechanics underneath and the result I saw. From my first click to an answer from the endpoint took me about 25 minutes, most of it package installs and the first start of the serving image.

Step 1. Create the task from git

WHAT I DID

I opened my project and chose **New Task**. On the Code tab I gave it the repository `https://github.com/damianerangey-cml/hands-on-labs.git`, branch `main`, working directory `train-hpc-serve-k8s` and script `train.py`, and cleared **Add Task.init call**, since the script makes its own. On the Environment tab I chose **Specify packages** and pasted the training lines from `requirements.txt`. Then I saved it as a draft.

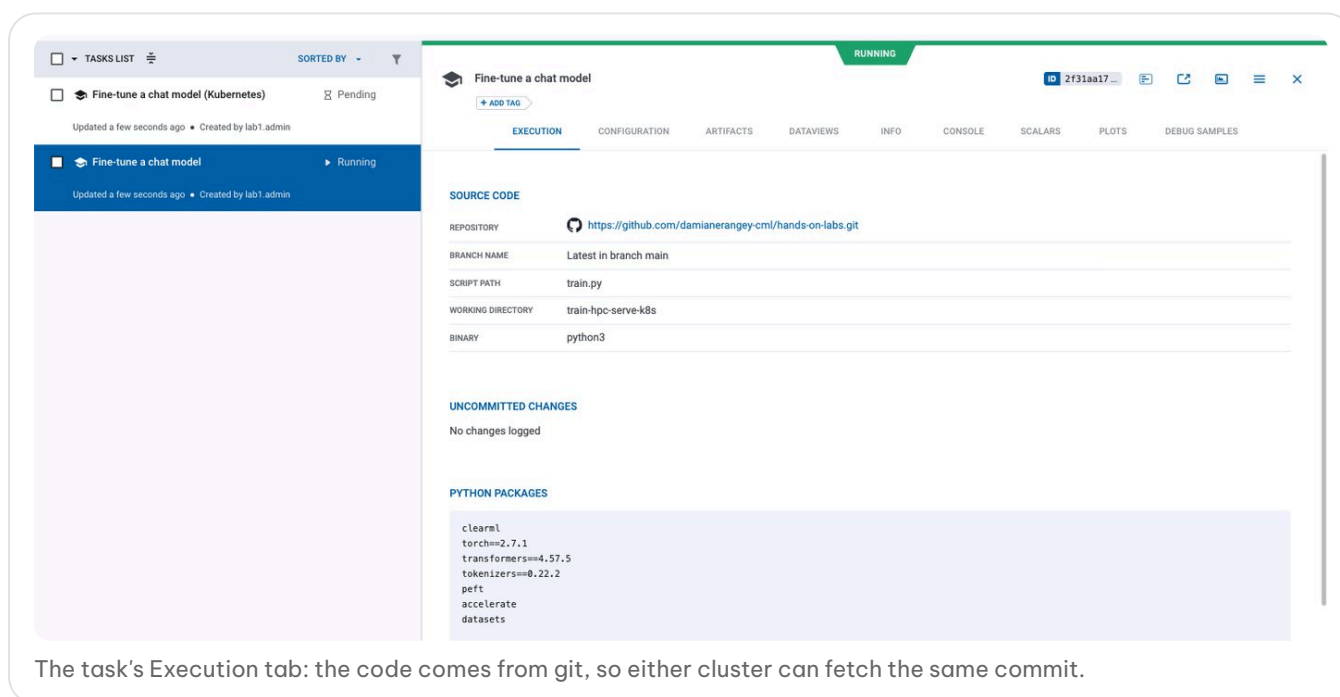
In your setup: use the same repository, or your own fork of it. From a laptop, `python train.py --queue "<your HPC compute resource>"` creates the task and sends it in one go.

UNDERNEATH

A draft task records where the code is and what it needs. No compute is used yet.

WHAT I SAW

A task in Draft status. Its Execution tab lists the repository, the working directory, the script and the packages:



The task's Execution tab: the code comes from git, so either cluster can fetch the same commit.

Step 2. Send it to the HPC cluster

WHAT I DID

I right-clicked the task and chose **Enqueue**. In my setup the compute resource for the HPC cluster is called SLURM Half GPU (train), so that is the one I picked: it makes sure the task runs on the Slurm cluster, on half a GPU.

In your setup: pick whichever compute resource sends work to your HPC cluster.

UNDERNEATH

Klique's HPC submitter on the cluster's login node pulled the task, filled in the compute resource's batch template and submitted it with `sbatch`. The template belongs to the platform team; in my setup it asks the gpu partition for half a GPU through Slurm GPU sharding. The batch job then started a Klique worker daemon on the compute node for this one task, which cloned the repository, built a Python environment with the packages and ran `train.py`. The task's console shows the batch directives the HPC submitter wrote:

```
launching with sbatch:
#SBATCH --job-name=clearml_2f31aa17e0da4c8a939922191f9842d1
#SBATCH --partition=gpu
#SBATCH --gres=shard:2
#SBATCH --mem=6G
#SBATCH --cpus-per-task=2
#SBATCH --time=01:00:00
#SBATCH --output=/var/log/clearml-agent/sbatch-2f31aa17e0da4c8a939922191f9842d1-%j.log
...
sbatch command rc=0: sbatch --parsable /tmp/clearml_slurmtmpl_ncv8buu0.txt
```

WHAT I SAW

Status Running within seconds. The console showed the batch directives, about three minutes of package installs on this first run, then a progress bar to 238 steps. The Slurm resource group on the orchestration dashboard showed its GPU busy.

Step 3. Read the outputs in Klique

WHAT I DID

While it ran, I opened the task's tabs: Configuration > Args, Scalars, Console and, at the end, Models.

UNDERNEATH

The SDK calls from the walkthrough put each output there: `argparse` for the arguments, the logger for the curves and the held-out loss, `OutputModel` for the model, uploaded to shared storage.

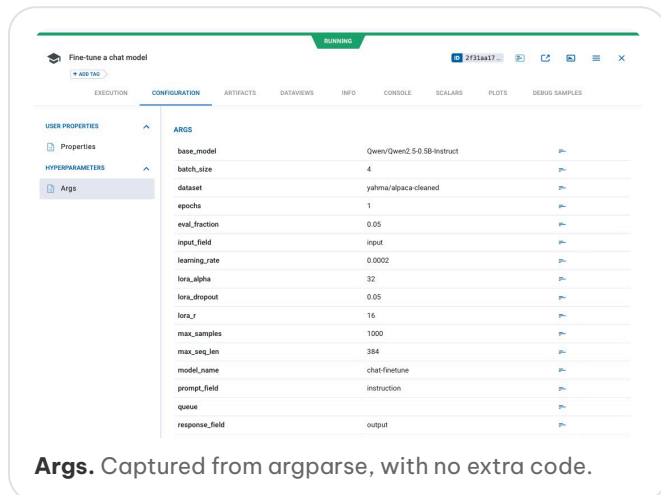
WHAT I SAW

Every argument of train.py under Args, with its default, ready to edit on a clone. A chart per training metric, and eval_loss under Summary. At the end the console printed the model's ID, and the task recorded the model as its output, with the weights in shared storage:

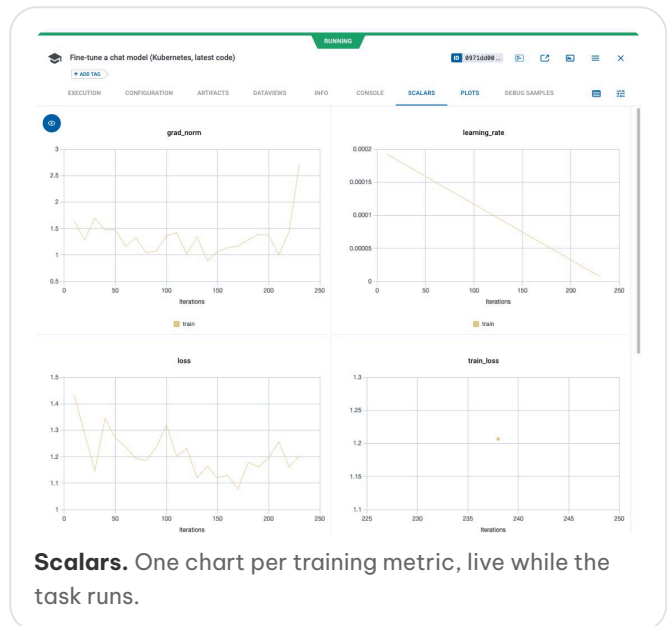
Registered model ID: 0aa00dfd8cd8438fa66fc04782dd1d17

Output model: chat-finetune (PyTorch)

Location: [https://<file server>/Builders - Train on Slurm, Serve on Kubernetes/
Fine-tune a chat
model.2f31aa17e0da4c8a939922191f9842d1/models/model_package.zip](https://<file server>/Builders - Train on Slurm, Serve on Kubernetes/Fine-tune a chat model.2f31aa17e0da4c8a939922191f9842d1/models/model_package.zip)

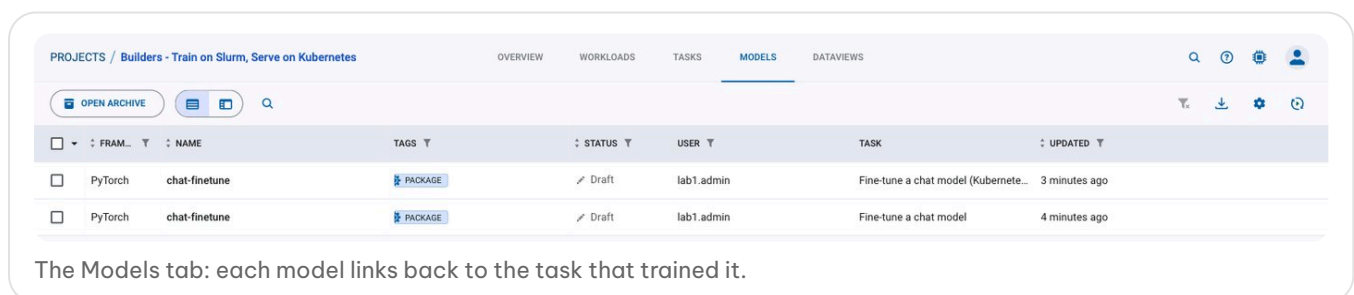


Args. Captured from argparse, with no extra code.



Scalars. One chart per training metric, live while the task runs.

Then I opened the project's Models tab, and there they are! One model per training run (the second one comes from step 4), each linked to the task that made it. I made a note of the model ID from the HPC run, 0aa00dfd8cd8438fa66fc04782dd1d17, because we'll need it shortly to serve the model.



The Models tab: each model links back to the task that trained it.

Step 4. Run the same task on Kubernetes

WHAT I DID

I right-clicked the task and chose **Clone**. On the clone's Execution tab I set the container image to `pytorch/pytorch:2.7.1-cuda12.6-cudnn9-runtime`, which already ships torch. Then I enqueued the clone to Half GPU (train, K8s), the compute resource in my setup that sends training work to the Kubernetes cluster.

In your setup: pick the Kubernetes GPU compute resource your team trains on, and a container image with Python 3.9 or later.

UNDERNEATH

The Klique worker daemon in the cluster pulled the task, rendered a pod for it with the container image and the GPU share the compute resource provides, and the cluster's scheduler placed it. Inside the pod the same repository was cloned and the same file ran. On the HPC cluster the task runs on the node itself, so the container image matters only on Kubernetes.

WHAT I SAW

The task showed Pending while the pod pulled its image and started, then Running. Both runs landed in the same project, side by side, and both registered a model.

Step 5. Serve the model on Kubernetes

WHAT I DID

I opened **vLLM Model Deployment** in the app catalogue and added an instance. I gave it a name, my project and the serving compute resource, which in my setup is Half GPU (serve). Under Model Configuration I pasted the model ID I noted in step 3 into **Model** and gave the endpoint a name in **Model Endpoint Name**. Then I launched it.

In your setup: pick the compute resource your team serves models on.

UNDERNEATH

An apps worker daemon started the app's controller, which launched the vLLM service as its own task on the serving compute resource. The worker daemon rendered that as a pod with a GPU share. vLLM downloaded the weights from shared storage by model ID and loaded them, and the app published an OpenAI-compatible endpoint behind an authenticated route.

WHAT I SAW

The instance showed a queued icon, then LOADING, then ACTIVE: ten minutes and 23 seconds for me, most of it the serving image download on first use, then about a minute for vLLM to fetch the 957 MB model by ID and load it. Then the model and its endpoint appeared on the instance page.

Import Configuration

Instance Name ⓘ

chat-finetune

Service Project - Access Control ⓘ*

Builders - Train on Slurm, Serve on Kubernetes

Queue ⓘ*

Half GPU (serve) → serve-0.5xgpu

AI Gateway Route ⓘ

Choose a route, otherwise a dynamic route will be used

Model Configuration

0aa00dfd8cd8438fa66fc04782dd1d17

+ Add item

LoRA Configuration

The launch form, with my serving compute resource and the model ID from the registry.

Step 6. Call the endpoint

**vLLM Model Deployment**
By ClearML
Create secure, authenticated LLM endpoints using the vLLM serving engine
v1.17.2.3666

**ACTIVE**
Idle for 0.0 hours
Automatically stopped after ∞ hours

GENERATE TOKEN: </settings/workspace-configuration>
Session: d57a6f2d4f914f6c8f36673c01d1a444 | [Console Log](#)

Model Name	Endpoint
chat-finetune	<a href="https://<router address>/service/<instance>/chat-finetune/v1">https://<router address>/service/<instance>/chat-finetune/v1

My running instance: status ACTIVE, the served model and its endpoint. I have hidden the host part of the address.

Once the instance was ACTIVE, its page gave me what I needed to call the model. Across the top: the status, the session, which is the ID of the vLLM service task running on Kubernetes, and a link to generate a Klique access token. Underneath sits the served model and its endpoint URL. The URL is built from the router address, then /service/ and that session ID, then the endpoint name I gave the model in the launch form (chat-finetune), then /v1. That /v1 is the base of an OpenAI-compatible API, so an OpenAI client adds /chat/completions to it. Every request goes through the authenticated route, so it needs a Klique access token.

Now let's call it. I made this a bit easier by using a simple script, [call_endpoint.py](#) from the example repo. The part that does the call is plain OpenAI client code:

```
client = OpenAI(base_url=args.endpoint, api_key=token) # the endpoint URL ending in /v1
reply = client.chat.completions.create(
    model=args.model, max_tokens=args.max_tokens, # the endpoint name: chat-finetune
    messages=[{"role": "user", "content": args.prompt}])
print("Answer:", reply.choices[0].message.content)
```

WHAT I DID

I copied the endpoint URL from the instance page and ran `call_endpoint.py` from inside Klique: a task from the same repository with `call_endpoint.py` as its script and the endpoint address under Args, sent to 1 CPU Slot. With no token set, the script records itself as a task and uses that task's own access token, so I never had to copy a token anywhere. From a laptop, it takes a Klique access token generated under Settings > Workspace configuration:

```
export ENDPOINT=<endpoint address from the instance page>
export TOKEN=<Klique access token>
python call_endpoint.py --prompt "Explain LoRA in two sentences."
```

Any OpenAI-compatible client works the same way.

UNDERNEATH

The request carried the token to the authenticated route, which checked it and forwarded the call to the vLLM pod. vLLM applied the model's chat template and generated the answer.

WHAT I SAW

An answer within a second or two, printed in the console. Here are my two answers, word for word:

```
Prompt: Write a two-sentence reply to a customer whose order arrived late.
Answer: I'm sorry to hear that your order was delivered late. Could you please let me know
when it
should have been shipped so I can make sure it's on time for the next time you place an
order?

Prompt: Explain LoRA in two sentences.
Answer: LoRA stands for Long Range Authentication and is an authentication method that allows
users
to authenticate with their phone's SIM card ...
```

The first is on topic; the second is fluent and wrong. That is what a half-billion-parameter model tuned on a thousand general instruction examples does: this build proves the path from the HPC cluster to an endpoint, and the quality of the answers comes from the base model you choose, your own data and an evaluation of your own.

Step 7. Another way: the whole round as one pipeline

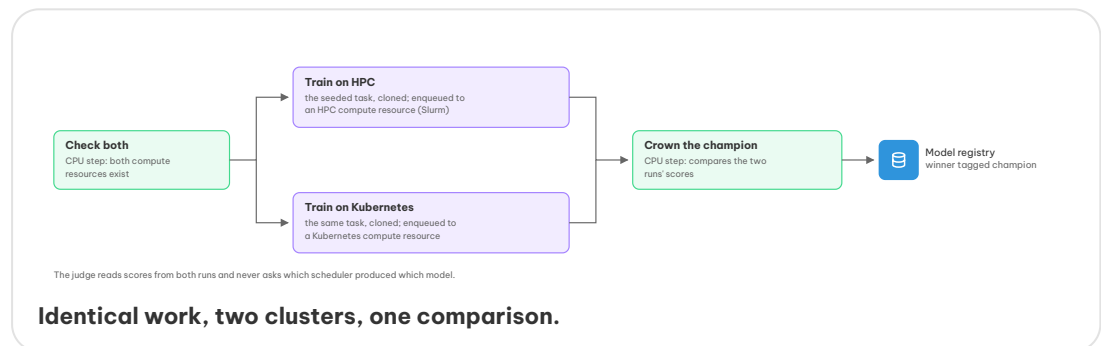
WHAT I DID

Steps 1 to 6 got me to the end state by hand. This is the same round as code, so it can run again with no clicks, from CI on every push for example. I ran [pipeline.py](#) with the ID of a draft copy of my training task and my three compute resources: SLURM Half GPU (train), Half GPU (train, K8s) and 1 CPU Slot. I created the controller as a task from the same repository, with `pipeline.py` as its script and the arguments under `Args`, and sent it to 1 CPU Slot.

In your setup: pass your own compute resource names to `--hpc`, `--k8s` and `--cpu`.

UNDERNEATH

The controller cloned the base task twice and sent one clone to each training compute resource, with the container image set on the Kubernetes step. When both finished, a function step read `eval_loss` from each run and tagged the better model champion.

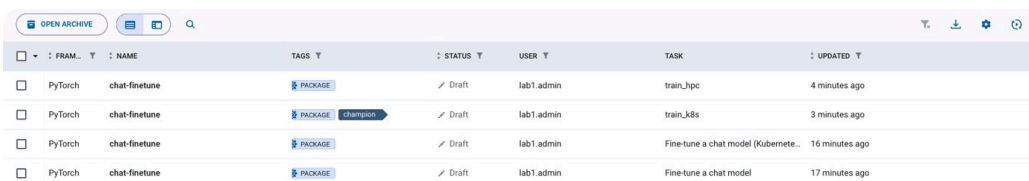


```
pipe.add_step(name="train_hpc", base_task_id=a.base_task_id, execution_queue=a.hpc)
pipe.add_step(name="train_k8s", base_task_id=a.base_task_id, execution_queue=a.k8s,
              task_overrides={"container.image": a.k8s_image})
pipe.add_function_step(name="pick_champion", function=pick_champion,
                      function_kwargs={"hpc_task_id": "${train_hpc.id}", "k8s_task_id": "${train_k8s.id}"},
                      function_return=["champion_model_id"], parents=["train_hpc", "train_k8s"])
pipe.start(queue=a.cpu)
```

WHAT I SAW

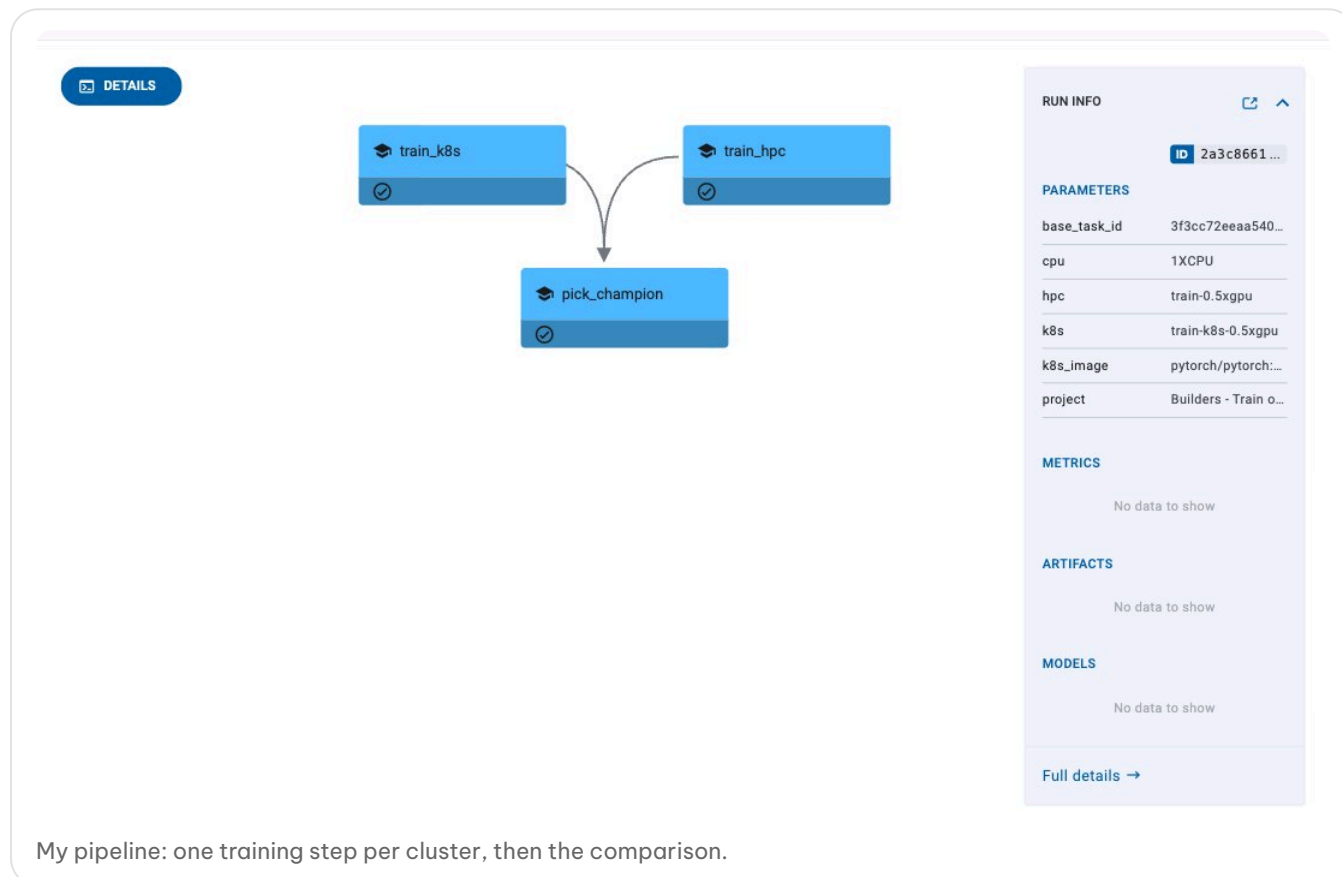
The pipeline page showed the two training steps running at once and the comparison step after them. The comparison step's console printed the scores and the winner, and the project's Models tab showed that model tagged champion:

```
eval_loss per run: {'b9d0542f8b1f438f9b162c10552c46b5': 1.1987,  
'fe35012016ca4f368b8e1bed300d29a5': 1.197} | champion model: 8271e3d0db8a4e069b2f1511d8449443
```



	FRAMEWORK	NAME	TAGS	STATUS	USER	TASK	UPDATED
☐	PyTorch	chat-finetune	PACKAGE	Draft	lab1.admin	train_hpc	4 minutes ago
☐	PyTorch	chat-finetune	PACKAGE champion	Draft	lab1.admin	train_k8s	3 minutes ago
☐	PyTorch	chat-finetune	PACKAGE	Draft	lab1.admin	Fine-tune a chat model (Kubernet...	16 minutes ago
☐	PyTorch	chat-finetune	PACKAGE	Draft	lab1.admin	Fine-tune a chat model	17 minutes ago

The Models tab after the pipeline: every model links to its task, and the comparison step tagged one champion.



The screenshot shows the 'DETAILS' view of a pipeline. On the left, a diagram illustrates the workflow: two parallel training steps, 'train_k8s' and 'train_hpc', both marked with a checkmark, feed into a final 'pick_champion' step, also marked with a checkmark. On the right, a 'RUN INFO' sidebar provides details for the selected run (ID: 2a3c8661...). The sidebar includes sections for 'PARAMETERS' (listing base_task_id, cpu, hpc, k8s, k8s_image, and project), 'METRICS' (showing 'No data to show'), 'ARTIFACTS' (showing 'No data to show'), and 'MODELS' (showing 'No data to show'). A 'Full details' link is at the bottom of the sidebar.

My pipeline: one training step per cluster, then the comparison.

What I got

Here are my first two training runs, from steps 2 and 4, with the default settings: 1,000 examples, one epoch, 238 steps.

	HPC CLUSTER (SLURM)	KUBERNETES
Compute resource	Half a GPU through Slurm GPU sharding	Half a GPU through a Kubernetes compute resource
GPU and precision	An older GPU class without bf16, so fp32	A newer GPU class with bf16
Environment	A Python 3.12 virtual environment on the node	The PyTorch container image, Python 3.11
Training time	296 seconds	48 seconds
Held-out loss	1.198	1.1973
Model registered	Yes, 957 MB of merged weights	Yes, 957 MB of merged weights
In the pipeline (step 7)	Held-out loss 1.1987	Held-out loss 1.197, tagged champion

Both clusters produced the same model quality from the same code and data. The difference in time comes from the GPUs and the precision they support, not from the schedulers: the Slurm cluster in my setup has an older GPU class. On an HPC cluster with current GPUs, the same task trains in the same way at that hardware's speed. For me the main result is not in the table: one task ran on two clusters with two different schedulers, with the same code and the same tracking, and the only choice I made was the compute resource.

Try it yourself

Thanks for taking the time to read this. I hope it shows how little stands between a fine-tune on an HPC cluster and a model answering requests on Kubernetes once both clusters sit behind the same platform. The best way to see it is to run a version of it yourself.

Try it on Klique

The Klique hands-on lab for the meta scheduler gives you a Slurm cluster and a Kubernetes cluster side by side, with compute resources for both. Or take the example code and run it with the open-source ClearML SDK on your own setup.

[Launch a hands-on lab](#)

[Get the code on GitHub](#)

RESOURCE	WHAT IT IS	LINK
Example code	train.py, pipeline.py, call_endpoint.py and requirements.txt	damianerangey-cml/hands-on-labs
Klique hands-on labs	A hosted Klique environment with an HPC cluster and Kubernetes	clearml-hol.com
ClearML SDK	The open-source SDK behind tasks, models and pipelines	clearml/clearml
vLLM	The open-source serving engine	vllm-project/vllm
Meta-Scheduler Reference Architecture	How Klique schedules across every environment	klique.ai
LLM Gateway and Tokenomics Reference Architecture	Putting the endpoint behind an alias, with budgets and guardrails	klique.ai
Klique HPC documentation	Slurm and PBS through the Klique Orchestrator	klique.ai/docs

Klique AI · Builders series · Fine-tune on the HPC cluster, serve on Kubernetes. Screens, logs and results are from one run in my Klique environment; times and losses depend on the model, data and hardware used. Slurm is a trademark of SchedMD LLC; Kubernetes is a trademark of The Linux Foundation.