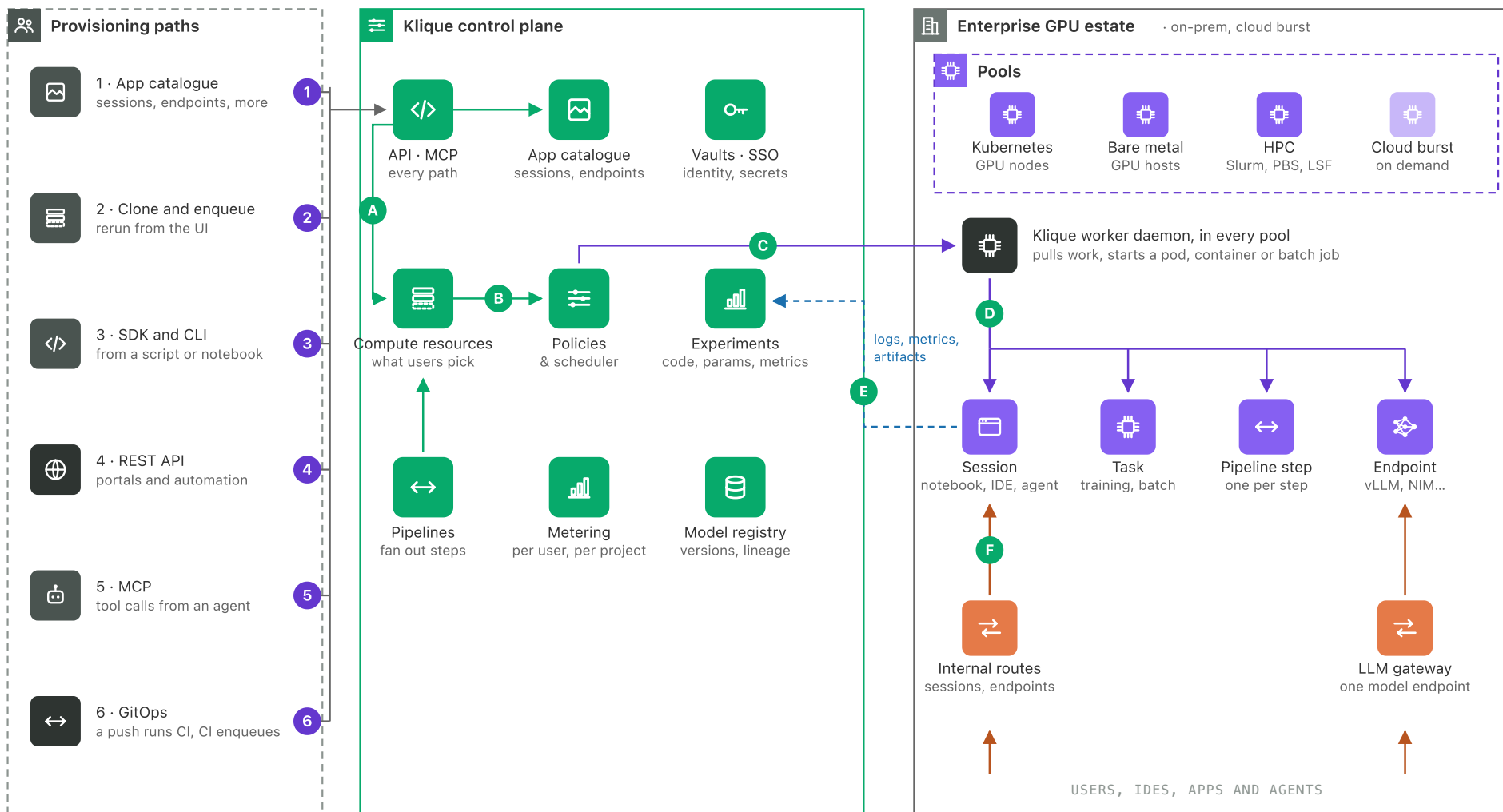


GPU as a service reference architecture

This reference architecture covers the following scenario: an enterprise platform team offers GPUs as a service internally. Data scientists, engineers, application teams, portals and agents have several ways to provision compute resources, all through one Klique control plane, on the company's own estate with burst to the cloud. It shows the two kinds of GPU workload, how each is stood up on Kubernetes, bare-metal and HPC estates, the paths people and agents use to ask for them, and how every GPU-hour is governed, metered and attributed.

CONTROL PLANE One, on-prem, run by the platform team	GPU CAPACITY Kubernetes, bare-metal and HPC pools, cloud burst on demand
PROVISIONING App catalogue, clone and enqueue, SDK, API, MCP, GitOps	AUDIENCE IT architects and sales engineers

How to read the diagram: numbered badges are the provisioning paths; green lettered badges follow what happens underneath, which is the same for every path.



PATH	DETAIL	LOCATION	CONNECTION
1 App catalogue	Launch any catalogue app: a JupyterLab, VS Code or coding-agent session, a vLLM, SGLang or NIM endpoint, a Slurm or K3k cluster, a vector database, and more.	UI → catalogue	Browser → API, HTTPS
2 Clone and enqueue	Clone a past run, edit its parameters, enqueue it. Or let the task scheduler and trigger manager do it.	UI	Browser → API, HTTPS
3 SDK and CLI	Two lines track a script; one more sends it to a GPU. clearnl-task runs any repo without code changes.	Laptop, notebook, CI	SDK → API, HTTPS
4 REST API	Every other path is built on it. Internal portals, platforms and automation call it directly, with a service user's keys.	Portals, scripts	JSON over HTTPS
5 MCP	An agent such as Claude Code calls the same platform as tools: list compute resources, enqueue, read metrics, launch apps.	Agent → MCP server	MCP over HTTPS, OAuth
6 GitOps	A push or pull request starts a CI job; the job runs the work on a compute resource and posts results back.	CI runner	CI → API, keys in CI secrets
UNDERNEATH, FOR EVERY PATH			
A One API	Every path lands on the same API, under the caller's own identity, so the same policy applies.	Control plane	HTTPS
B Admit	The request resolves to a compute resource; policy and the scheduler admit it to a pool.	Control plane	A scheduling decision
C Pull	The Klique worker daemon in that pool pulls the work.	GPU pool	Worker daemon → control plane, outbound
D Run	The worker daemon starts the work: a pod on Kubernetes, a container on bare metal, a batch job on HPC (through the HPC submitter on the login node). The work is a session, a task, a pipeline step or an endpoint.	GPU node	Stays on the node
E Record	Logs, metrics, artifacts and models flow back to experiment tracking and the model registry.	Control plane	Outbound HTTPS

PATH	DETAIL	LOCATION	CONNECTION
F Reach	Sessions and endpoints are reached through internal routes; model calls go through the LLM gateway.	Gateways	HTTPS

Business value

- **For the platform team: one way to hand out GPUs.** Compute resources, policies and a catalogue are set once. Every path lands on the same scheduler, so the compute policy engine applies the same reservations and limits however people come in.
- **For builders: the path that fits the work.** A notebook for exploring, the SDK for real runs, a pipeline for production, an agent for the repetitive steps. The same GPUs and the same tracking throughout.
- **For finance: every GPU-hour attributed.** Every path creates the same kind of record, with an owner and a project, so usage is attributed the same way whichever path created it.

Interactive and non-interactive GPU requests

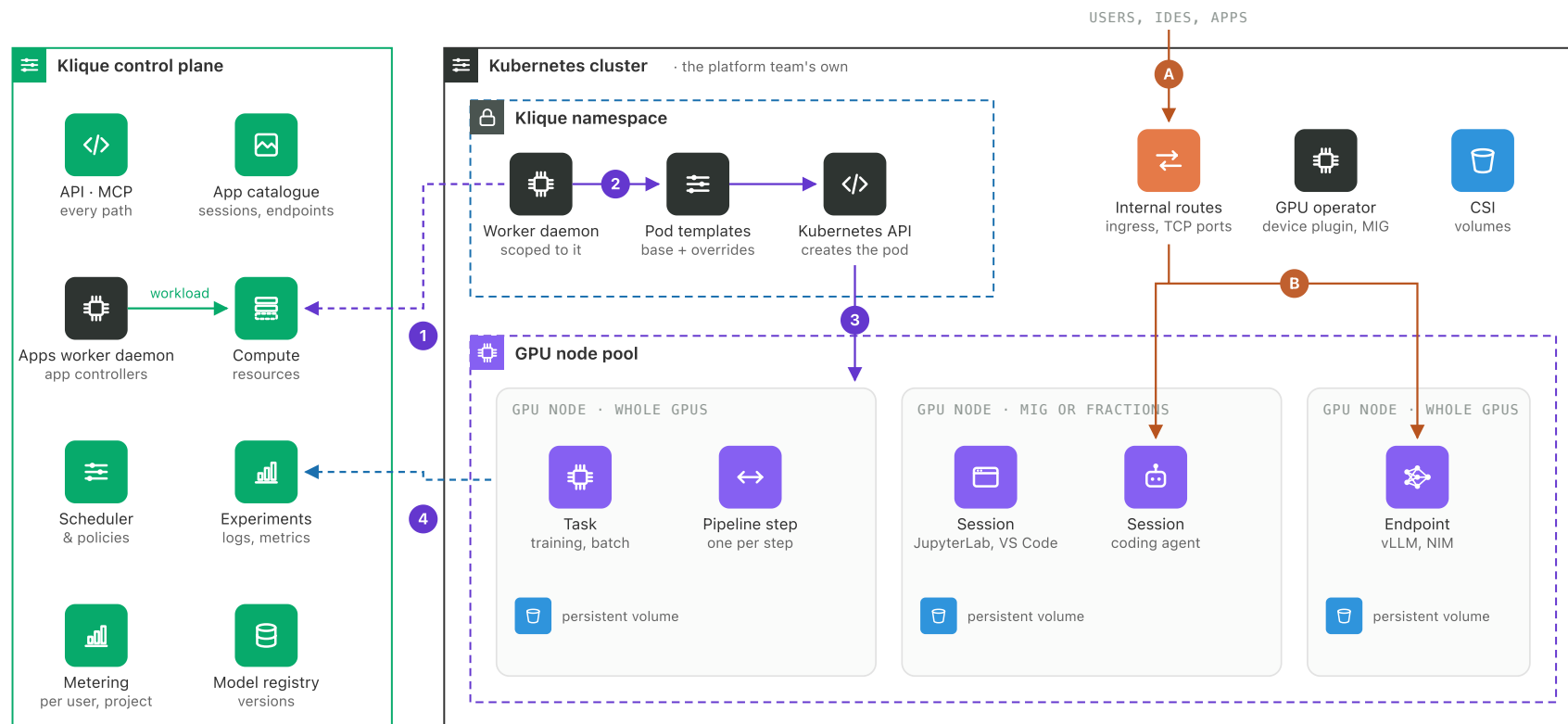
Everything a GPU does here is one of two kinds. **Non-interactive** work runs to completion and gives the GPU back: training, fine-tuning, batch inference, optimisation trials, pipeline steps. **Interactive** work holds a GPU while someone, or something, uses it: notebooks, IDEs, coding agents, SSH sessions and model endpoints. Every provisioning path creates one or the other; the estate underneath decides how it is stood up.

	NON-INTERACTIVE	INTERACTIVE
Examples	Training, fine-tuning, batch inference, HPO trials, pipeline steps	JupyterLab, VS Code, SSH, coding agents, model endpoints
Created as	A task on a compute resource	An app instance, whose controller creates the session on a compute resource
Lifecycle	Ends when the run ends	Until stopped, or idle past its limit
Reached through	Results in the UI, API or MCP	Internal routes (HTTPS or a TCP port), or an SSH tunnel from a laptop
On Kubernetes	A pod rendered from the compute resource's template	A session pod, with an internal ingress route
On bare metal	A container on the GPUs the worker daemon picks	A host-network container with an internal route
On HPC	A batch job built from the compute resource's template (Slurm, PBS, LSF)	Usually placed on a Kubernetes or bare-metal pool beside the cluster

Every estate, one control plane. The next sections show how each kind is stood up on Kubernetes, on bare metal and on an HPC cluster (Slurm, PBS or LSF), then how one estate mixes all three.

Kubernetes architecture

On Kubernetes the worker daemon runs as a controller in its own namespace, scoped to it by default. It never runs the work itself: for every task it **renders a pod spec** from a base template plus the compute resource's overrides, and the Kubernetes API does the rest. Whole GPUs, MIG slices and memory-capped fractions are all just different specs.

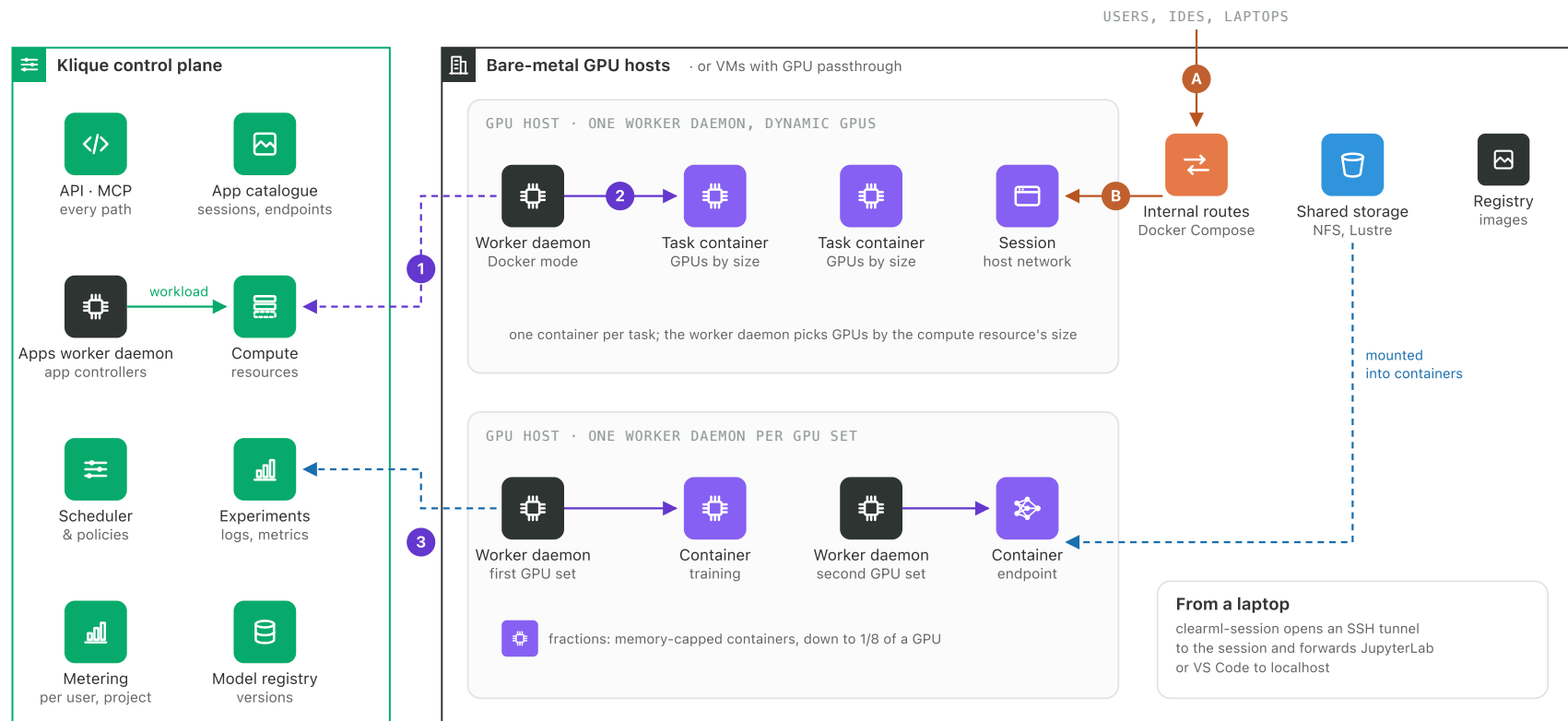


The worker daemon generates the pod; Kubernetes runs it. Numbered: how any workload is stood up. Lettered: how people reach the interactive ones.

STEP	DETAIL
1 Pull	The worker daemon pulls work from its compute resources over an outbound connection.
2 Render	Base pod template plus the compute resource's overrides: GPU limits, a MIG profile or a fraction label, node selectors, tolerations, volumes, priority.
3 Create	The Kubernetes API creates the pod on a GPU node; the GPU operator's device plugin hands it the GPUs. Persistent volumes come through CSI.
4 Report	Logs, metrics and artifacts stream back to the control plane. A task pod exits when the run ends.
A Reach	Users, IDEs and apps reach sessions and endpoints through internal routes: HTTPS through ingress, TCP through stream ports.
B Route	The route reaches the session or endpoint pod, with access checked on every route. Idle sessions stop and free their GPU.

Bare-metal architecture

On bare metal, or VMs with GPU passthrough, the worker daemon runs **on each GPU host**, and how a task runs depends on the mode it was started in. In **Docker mode** it runs the task's container and executes the task inside it. In **virtual environment mode** it builds a Python environment on the host, installs the task's packages, clones the code, applies any uncommitted changes and runs the task natively, reusing cached environments between tasks. Either way the work lands on the GPUs the worker daemon assigns. One worker daemon can own all of a host's GPUs and hand each task the number its compute resource asks for, or one worker daemon can own each GPU set. Fractions need Docker mode: memory-capped containers, down to an eighth of a GPU.

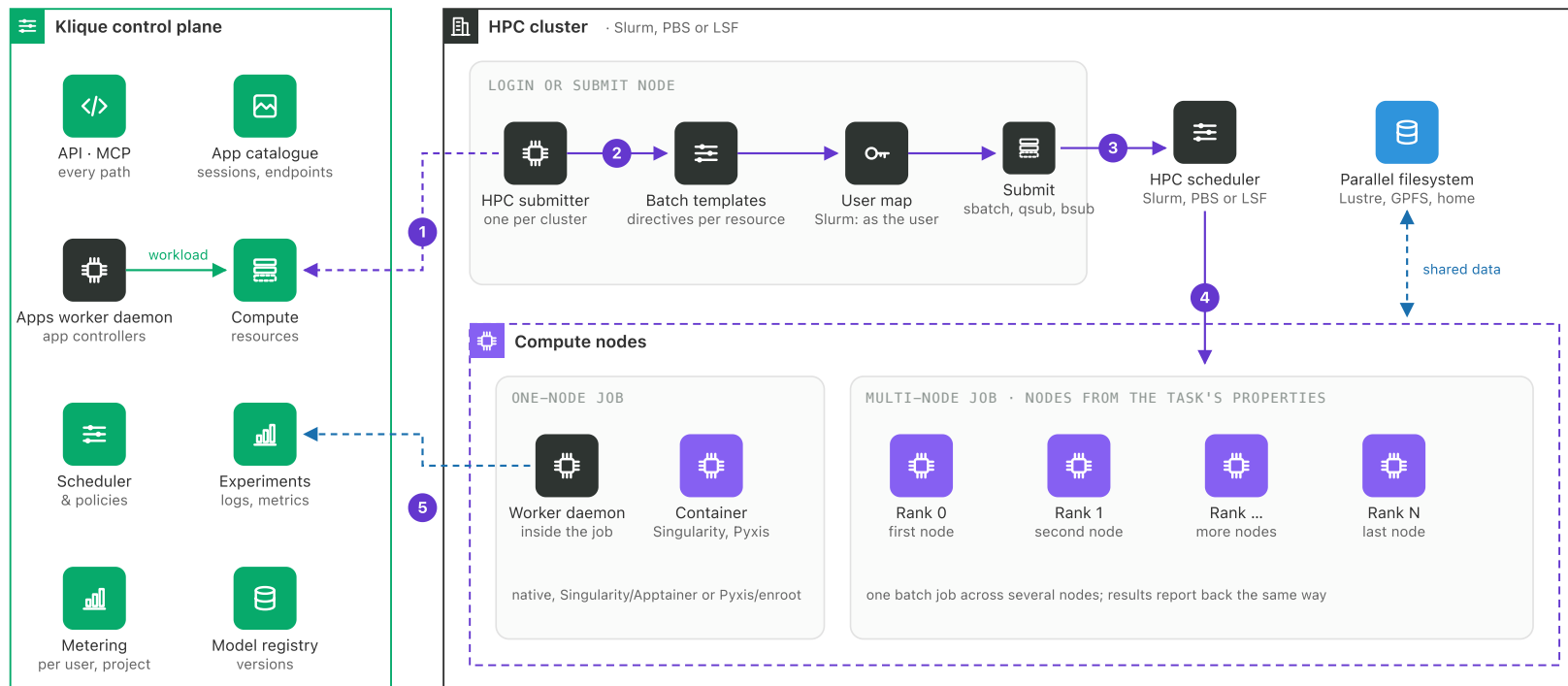


Docker mode here: a container per task, on the GPUs the worker daemon picks. Sessions run on the host network and get internal routes from a router run on the same network with Docker Compose.

STEP	DETAIL
1 Pull	The worker daemon pulls work from the compute resources it serves, in priority order.
2 Start	In Docker mode it starts a container with the task's image, packages and code; in virtual environment mode it builds the environment on the host and runs the task there. Either way on the assigned GPUs, with shared storage mounted.
3 Report	Logs, metrics and artifacts stream back; the container or the process ends with the run.
A Reach	Users reach sessions through internal routes, over HTTPS or a TCP port. One router serves each isolated network.
B Route	The route reaches the session's address on the host. From a laptop, clearml-session tunnels over SSH instead.

HPC architecture: Slurm, PBS and LSF

On an HPC cluster, Klique works through the cluster's own scheduler rather than around it. A **submitter** on the cluster's login node turns each task into a **batch job**, using a batch template tied to the compute resource, and submits it with sbatch on Slurm. On Slurm, a user map (`agent.slurm.user_map`) submits each batch job as the user's own Linux account. PBS and LSF follow the same submitter pattern: a batch template, submitted with qsub or bsub. The scheduler places it; inside the job, a worker daemon builds the environment and runs the task, natively or in a Singularity, Apptainer or Pyxis/enroot container.

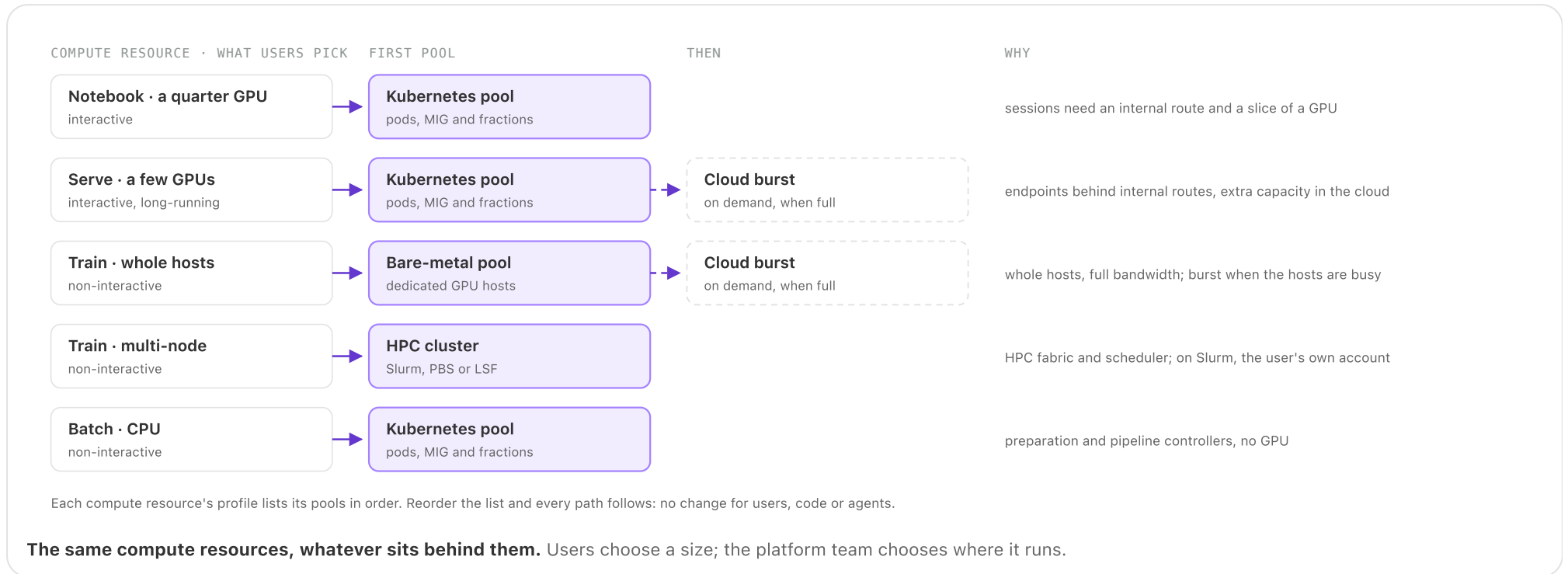


The cluster's scheduler; on Slurm, the user's own account. Task properties drive the template's directives, such as the node count. Interactive sessions and endpoints usually run on a Kubernetes or bare-metal pool beside the cluster.

STEP	DETAIL
1 Pull	The submitter on the login node pulls work from the compute resources mapped to it; one instance serves the cluster.
2 Render	It renders the batch script from the compute resource's template. On Slurm, the user map (<code>agent.slurm.user_map</code>) maps the Klique user to a Linux user.
3 Submit	It submits the batch job to the cluster's scheduler: <code>sbatch</code> on Slurm, as the mapped user; <code>qsub</code> on PBS; <code>bsub</code> on LSF.
4 Run	The scheduler places the job on compute nodes; the worker daemon inside it runs the task on the parallel filesystem.
5 Report	Logs, metrics and artifacts stream back over outbound connections, the same as on any other estate.

A mixed estate

Most estates are a mix: Kubernetes for sessions and endpoints, dedicated hosts for heavy training, an HPC cluster (Slurm, PBS or LSF) for multi-node runs, and the cloud for peaks. One control plane serves them all. Each compute resource lists its pools in order, so placement is a decision the platform team makes once. The [Meta-Scheduler reference architecture](#) shows how the scheduler places work across estates.



The app catalogue, by how it uses a GPU

The catalogue is the fastest path: a form in the UI, and the same call in the API or over MCP. Grouped by how each app consumes a GPU, which is what matters for planning pools and policies:

Development and agents

Interactive sessions, until stopped or idle: JupyterLab, VS Code, SSH sessions, a remote desktop; Claude Code, Codex and OpenCode as coding agents, each in its own container. Often a slice of a GPU is plenty.

Model serving

Long-running endpoints: vLLM, SGLang and Llama.cpp model deployment, embedding models, NVIDIA NIM, NVIDIA Dynamo and NIM RAG, and an MCP server deployment.

Training and automation

Batch work, until done: the multi-node trainer, hyperparameter optimisation (one trial per task), the task scheduler and the trigger manager.

Every launch form exports and imports as JSON, so a team's standard session or endpoint can be kept in git and relaunched the same way.

Platforms for teams

Clustering services: Slurm on Kubernetes for HPC-style teams, and K3k, Kubernetes in Kubernetes, for a virtual cluster with its own API.

Vector databases

Data alongside the models: Milvus and Qdrant sessions for retrieval and RAG, next to the models and the data.

App builders and custom apps

Apps for end users: Gradio and Streamlit launchers, the containerized application launcher, and any app the platform team uploads, such as AMD Inference Microservices.

GPU provisioning paths

The catalogue suits point and click. The other paths suit code, portals, agents and pipelines. All of them create the same objects, so each lands on the same compute resources, policies and ledger.

PATH	WHO	CREATES	LIFECYCLE	REACHED THROUGH	TYPICAL GPU SIZE
1 · App catalogue	Anyone, point and click	An app instance: a session, endpoint, cluster...	Until stopped or idle	Internal routes, browser or laptop	A slice or a whole GPU
2 · Clone and enqueue	Anyone in the UI	A cloned task, or scheduled runs	Until the run completes	Progress in the UI	Whole GPUs
3 · SDK and CLI	Developers	A tracked task, pipeline steps	Until the run completes	Progress in the UI	A slice to many nodes
4 · REST API	Portals, platforms, automation	Any of these objects	As the call made	The calling system	As requested
5 · MCP	Agents, on a user's behalf	The same tasks and app instances	As the path it mirrors	The agent's answer	As requested
6 · GitOps	CI pipelines	A task per push or pull request	Until the run completes	Results on the pull request	Whole GPUs

SDK and CLI

Two lines track a script; one more hands it to a GPU. `clearml-task` wraps any repository without changing its code.

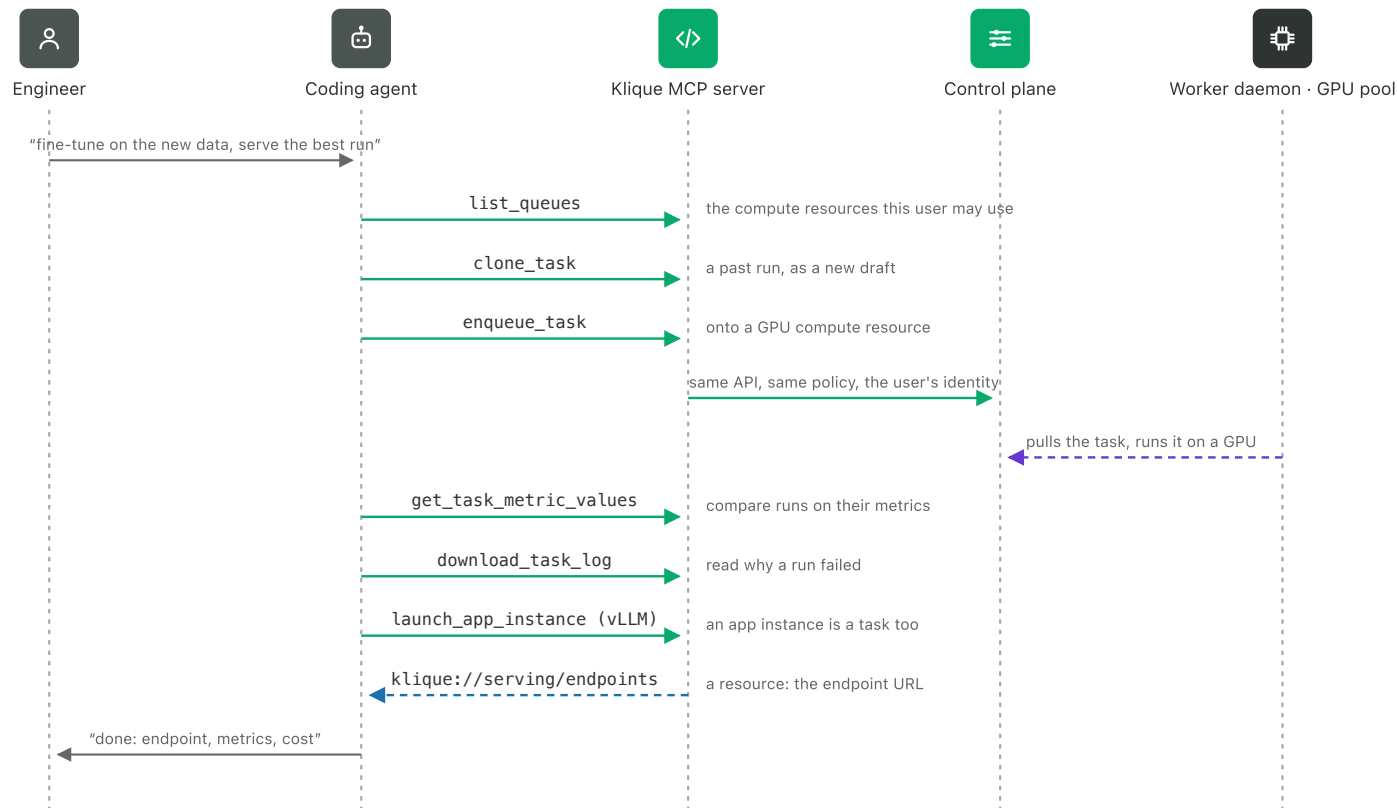
```
from clearml import Task
task = Task.init(project_name='AI Engineering', task_name='train')
task.execute_remotely(queue_name='compute resource')
```

REST API

Every other path is built on the API. Internal portals, platforms and automation call it directly with a service user's keys: to create and enqueue tasks, launch catalogue apps, and read usage.

MCP

The platform ships an **MCP server** that fronts the API. Agents such as Claude Code connect to it over HTTPS with OAuth, and every tool call runs under the user's own identity and permissions: the same compute resources, policy and ledger as the UI. Registering it takes one command, `claude mcp add --transport http klique https://<host>/mcp`.



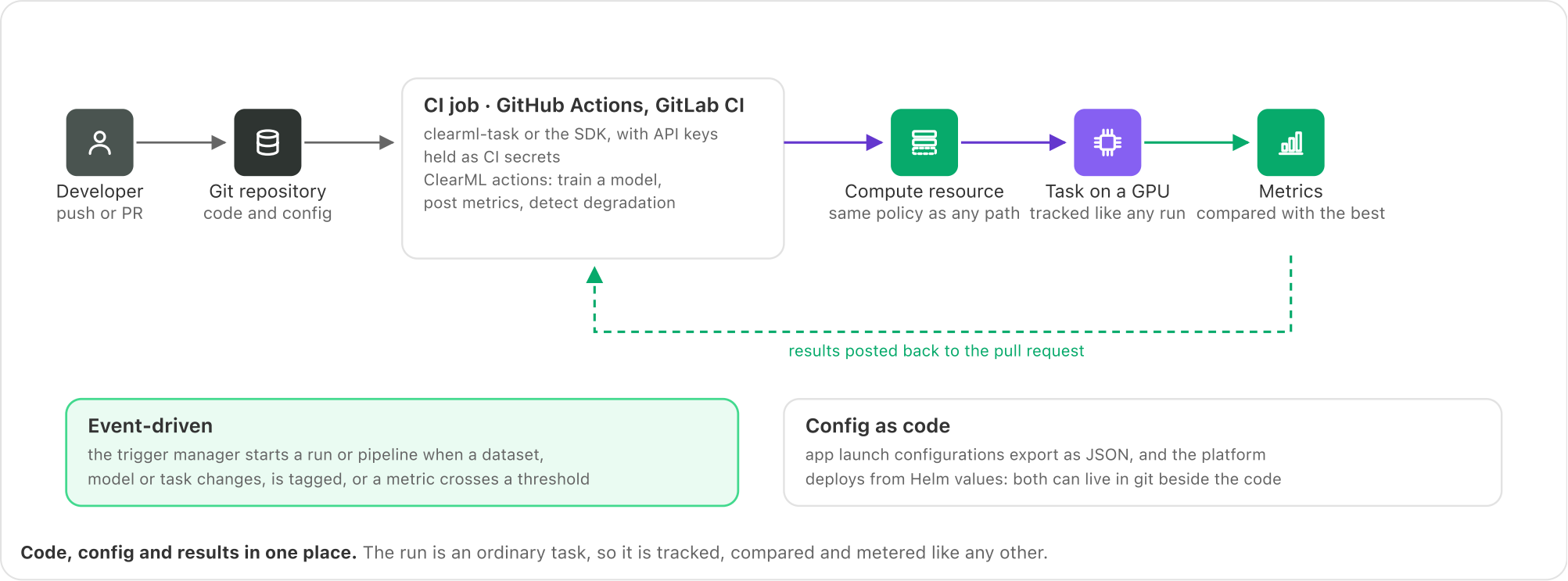
An agent drives the whole loop. Every call is an ordinary platform action, tracked and metered like any other.

TOOL GROUP	EXAMPLES	WHAT AN AGENT DOES WITH THEM
Compute	<code>list_queues</code>	Find the compute resources the user may use
Tasks	<code>create_task</code> , <code>clone_task</code> , <code>enqueue_task</code>	Start and rerun work on a compute resource
Results	<code>get_task_metric_values</code> , <code>download_task_log</code>	Compare runs on their metrics, read why one failed
Apps	<code>launch_app_instance</code> , the <code>klique://apps</code> resource	Open a session or deploy an endpoint from the catalogue
Models and serving	<code>list_models</code> , <code>get_serving_endpoint_metrics</code>	Find a registered model, track an endpoint's latency and GPU use

App launches can be refused, by an admin lock or the company's running-apps quota, and the agent gets a readable reason rather than a raw failure.

GitOps

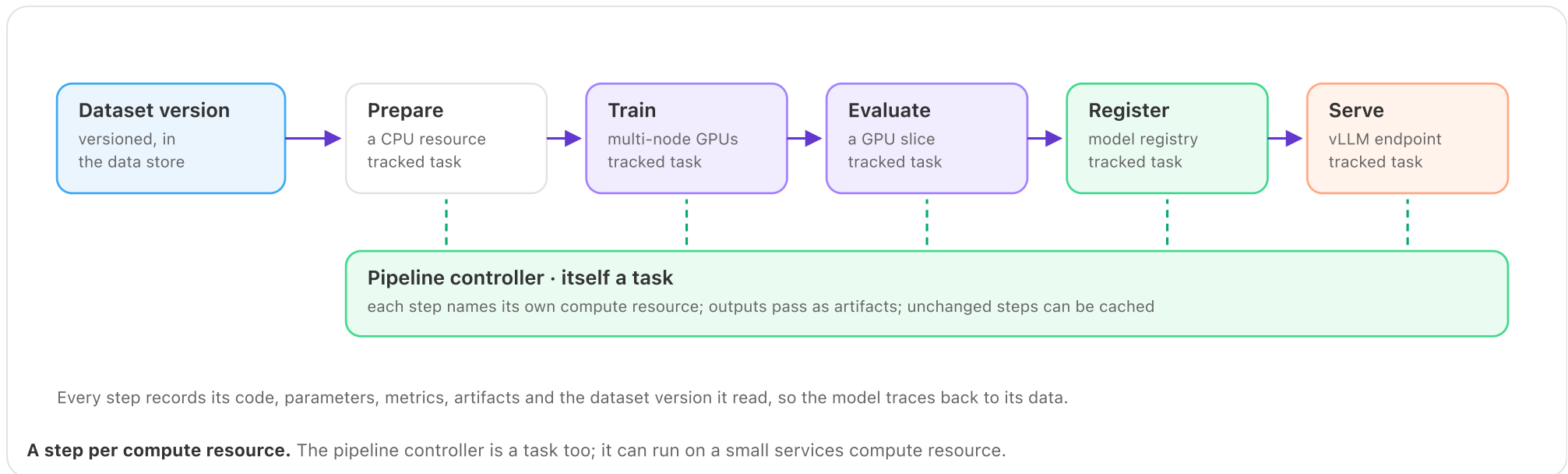
Training belongs in the same loop as the code. A push or a pull request starts a CI job in GitHub Actions or GitLab CI; the job uses `clearml-task` or the SDK, with API keys held as CI secrets, to run the work on a compute resource. ClearML's own GitHub actions cover the common steps: train a model on a comment, post a run's metrics to the pull request, and flag a model that got worse.



Experiments and pipelines

Whichever path starts a run, the platform records it the same way: the code and uncommitted changes, packages, parameters, console output, metrics, models and artifacts. That record makes any run reproducible and comparable, and it is the record an agent reads when it compares runs over MCP.

A **pipeline** chains runs. Each step names its own compute resource, so preparation lands on CPUs, training on many GPUs and evaluation on a slice, and outputs pass between steps as artifacts.



```

@PipelineDecorator.component(execution_queue='CPU')
def prepare(url): ...

@PipelineDecorator.component(execution_queue='multi-node GPUs')
def train(data): ...

@PipelineDecorator.component(execution_queue='GPU slice')
def evaluate(model): ...
  
```

Where the work lives

Task record

The repository, commit, uncommitted changes, parameters, packages and owner sit on the control plane.

Code

Cloned on the node from git, which needs a route and credentials, or baked into the image.

Secrets

Held in vaults on the control plane and injected when the work starts. Never in images or launch parameters.

Output destination

Defaults to the control plane's file server. Each project can point it at other storage, such as an in-region bucket or on-prem storage.

Datasets

Metadata on the control plane, files in storage.

Logs and metrics

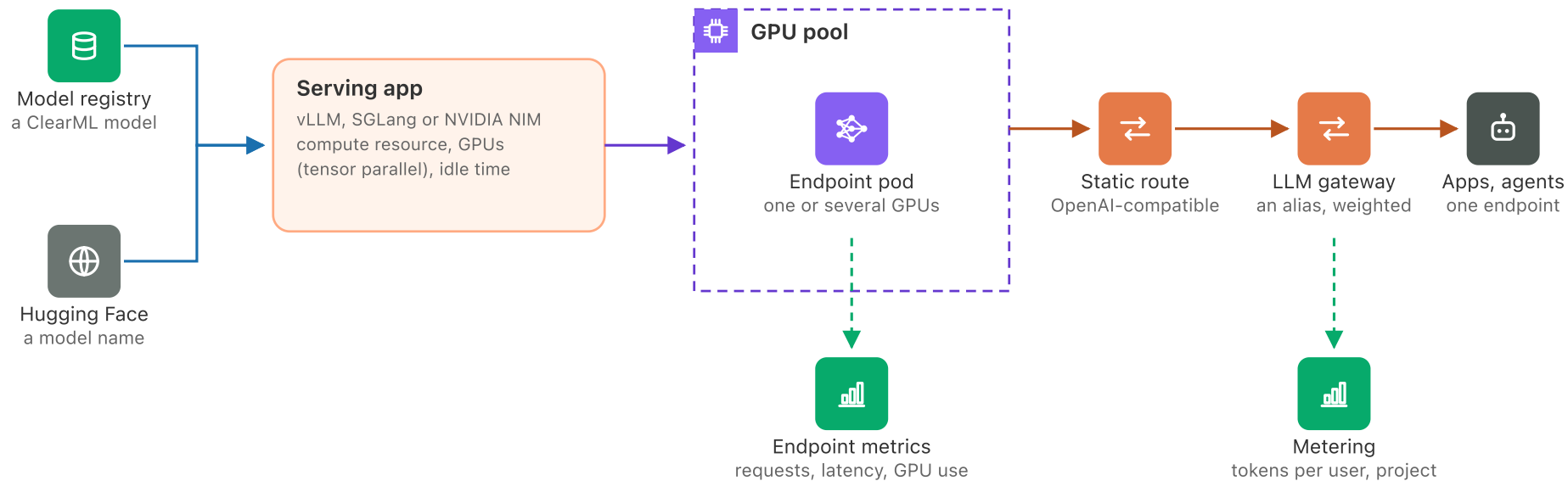
Stream home to the control plane as the work runs. Secrets stay out of printed output.

Model serving

For most application teams, GPU as a service means a model endpoint. The serving apps (vLLM, SGLang and NVIDIA NIM) take a model from the registry or from Hugging Face by name, request the compute resource, and set the number of GPUs for tensor parallelism. The endpoint is OpenAI-compatible and sits behind an authenticated internal static route.

Putting that endpoint in front of callers follows the same order each time. The platform team creates a **static route**, a fixed address that outlives any one endpoint. The serving app is launched against that route, so the endpoint it stands up answers there. The route is then registered with the **LLM gateway** under an alias, and the alias is what applications call. The model behind the route can be swapped, and a second instance added for capacity, with client code left as it is.

Routing rules, guardrails, budgets and token metering sit behind that alias; the [LLM Gateway and Tokenomics reference architecture](#) covers them.



Endpoints stop when idle. More capacity means another instance behind the LLM gateway, which balances across weighted endpoints.

One endpoint for callers, many behind it. Put the LLM gateway in front: callers use an alias, and the platform team can add capacity or swap models without touching a client.

One governed path underneath

Whatever the path, the same controls apply, because every path ends at a compute resource:

- **Compute resources** are what people pick. Name them by what they give (a whole GPU, half a GPU, a node), never by where they run.
- **Resource policies** decide which teams may use which compute resources, and give each team a reservation, a limit and a priority. The [Meta-Scheduler reference architecture](#) covers how the scheduler applies them.
- **Pools** behind each compute resource are tried in order: on-prem first, cloud burst when full. The [Cloud Autoscaler reference architecture](#) shows how cloud capacity scales up from zero and back.
- **Metering** records every run with an owner and a project. Sessions and endpoints also report their runtime, and model calls report tokens.

Identity comes from the company's single sign-on: OIDC, SAML or LDAP. Agents over MCP act under their user's identity, and CI acts as its own service user, with the same policy and attribution as anyone else. Every task carries its owner.

One ledger, every path. Usage is attributed to the user and project that created the work, whether it came from a browser, a script, a CI job or an agent.

Design considerations

Decide these before opening up GPU access, rather than finding them out in production.

Offer sizes, not hardware

Publish a few compute resources per size (a whole GPU, half, a quarter, a node) and let each one's pool order decide where it runs.

Set idle limits

Sessions and endpoints hold GPUs until stopped. Idle limits on time, GPU and network activity hand them back.

One router per isolated network

Internal routes must reach the sessions they serve. Kubernetes gets one per namespace; each bare-metal network gets one run with Docker Compose.

Choose the bare-metal worker daemon model

One worker daemon per host with dynamic GPUs packs work by size; one worker daemon per GPU set is simpler and more predictable.

Map HPC users

On Slurm, batch jobs run as the user's own Linux account through the user map (`agent.slurm.user_map`). Unmapped users are refused unless a default account is set. PBS and LSF follow the same submitter pattern.

Keep sessions beside the cluster

Interactive work holds its allocation until idle. Run it on Kubernetes or bare metal and keep the HPC cluster for batch and multi-node.

Size the apps worker daemon

It runs every app's controller. It needs CPU, not GPUs, and should never be the bottleneck for a launch.

Pre-bake images

Cold starts are image pulls. Keep standard images in a local registry, the same on every estate.

Give CI its own identity

A service user with its own keys, held as CI secrets, scoped to the compute resources CI should use.

Agents act as the user

MCP calls use the user's identity and permissions. An agent can only do what its user can.

Plan endpoint capacity

Serving apps don't autoscale. Add instances behind the LLM gateway, which balances across weighted endpoints.

Multi-node needs wider rights

On Kubernetes, multi-node training needs cluster-wide worker daemon permissions. Grant them only where multi-node runs.

Glossary

Compute resource	What a user sends work to, from any path. It names a size, never a location. Under the hood it is a queue.
Profile	The allocation behind a compute resource (a fraction, a GPU or several GPUs, which can be specified by GPU memory) and its ordered pool links. Reordering the links changes placement for every path at once.
Resource pool	A set of nodes running worker daemons in one place: an on-prem Kubernetes cluster, bare-metal hosts, a Slurm partition or an autoscaled cloud account.
Resource policy	Which teams may use which compute resources, with a reservation, a limit and an execution priority.
Task	One tracked run: its code, environment, parameters, outputs and owner. Sessions and endpoints are tasks too.
App instance	A launched catalogue app, stored as a task of type application.
Apps worker daemon	Runs each app's controller, which creates the app's workload on the chosen compute resource.
Klique worker daemon	The worker in each pool that pulls work and stands it up: a pod on Kubernetes, a container on bare metal, a batch job on HPC.
Pod template	On Kubernetes, the base pod spec plus each compute resource's overrides, from which the worker daemon renders every pod.
Worker daemon on bare metal	The worker daemon process on a GPU host, running each task as a container in Docker mode or in a Python environment it builds on the host, on the GPUs it assigns.

HPC submitter	On an HPC cluster, the process on the login node that turns tasks into batch jobs for Slurm, PBS or LSF, one batch template per compute resource.
Internal routes	Authenticated routes to sessions and endpoints.
LLM gateway	One endpoint for model calls: aliases, routing, budgets and metering.
MCP server	The platform's Model Context Protocol server: the API as tools for agents, under the user's identity.
Pipeline	A chain of tasks, each on its own compute resource, run by a controller task.
Model registry	Registered models with their versions and the runs that produced them.

Klique AI · GPU as a service reference architecture · one control plane, every provisioning path, one governed platform underneath. Compute resource names, sizes and code are illustrative.