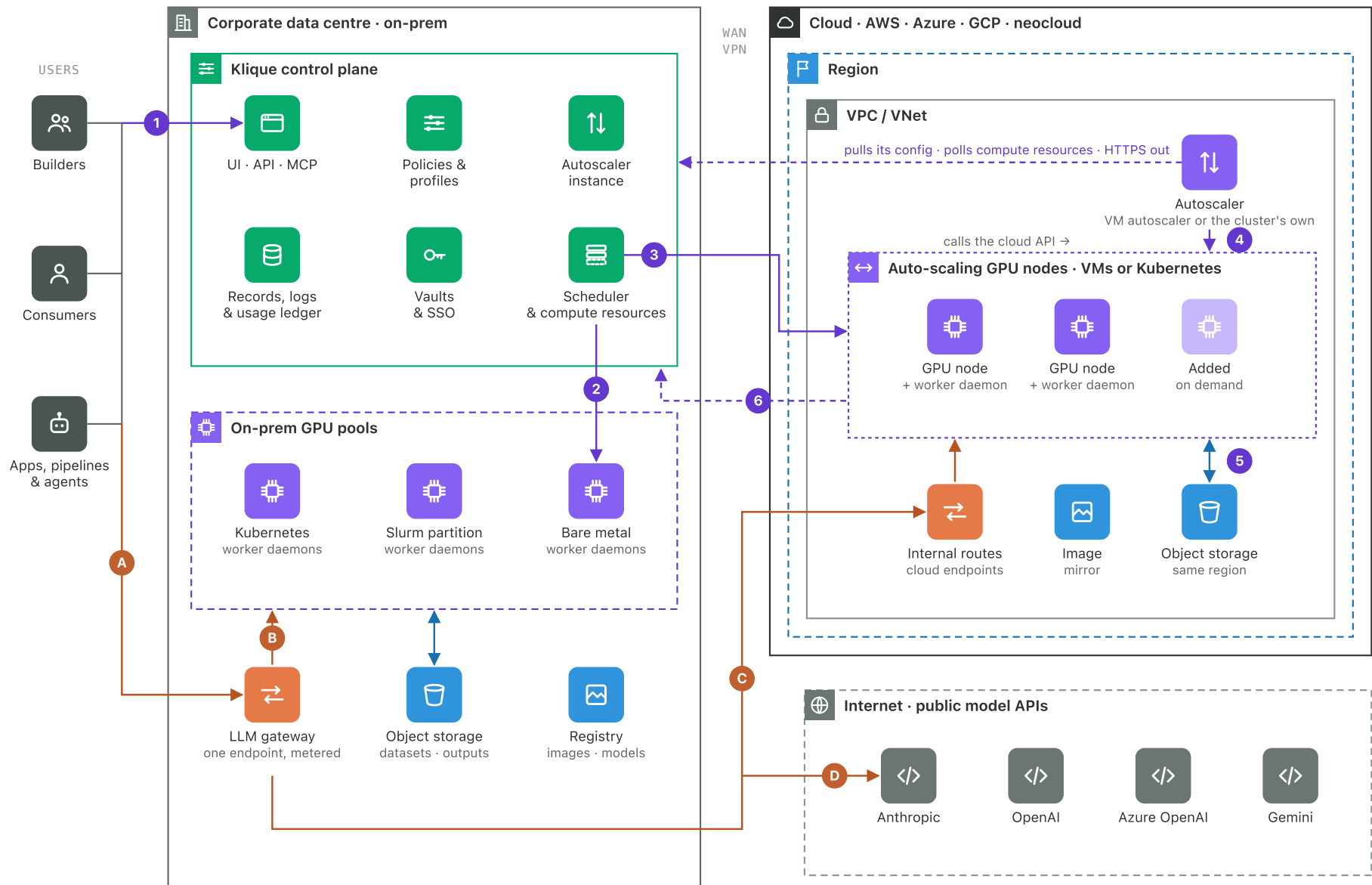


Cloud autoscaler reference architecture

This reference architecture covers the following scenario: the Kliques control plane runs on-prem, in the customer's data centre, and GPU work can burst to a cloud region when on-prem capacity is full. It shows how the components fit together, where each workload runs, and how model calls are routed and metered.

CONTROL PLANE On-prem, in the customer's data centre	GPU CAPACITY On-prem pools first, then a cloud region on demand
MODEL CALLS One LLM gateway, on-prem, for self-hosted and public models	WRITTEN FOR IT architects and sales engineers

How to read the diagram: numbered badges follow a compute provisioning task, lettered badges follow a model call.



STEP	DETAIL	LOCATION	CONNECTION
A COMPUTE PROVISIONING TASK, FROM SUBMISSION TO SCALE-DOWN			
1 Submit	A user, app, pipeline, schedule or agent sends work to a compute resource, from the UI, a session, the SDK or MCP.	Control plane	Caller → control plane, HTTPS
2 Run on-prem	The scheduler checks the team's policy and admits the task to the first pool in IT's order, on-prem. A worker daemon there pulls it and starts the container.	On-prem GPU pool	Worker daemon → control plane, outbound
3 Overflow	Once on-prem's declared capacity is used up, the next task is admitted to the cloud pool instead.	Control plane	None: a scheduling decision
4 Scale up	A GPU node is added: by Klisque's VM autoscaler, or on managed Kubernetes (AWS, Azure, Google Cloud or a neocloud) by the cluster's own node autoscaler (section 03). The task starts there with its image and code.	Cloud VPC	Autoscaler → cloud API; worker daemon → control plane, outbound HTTPS
5 Run near the data	The task reads its data and writes checkpoints and models to storage in the same region.	Cloud region	Stays in the region
6 Report & scale down	Logs, metrics, status and output pointers stream home. Once the idle timeout passes, the node is removed.	Cloud → control plane	Outbound HTTPS, small payloads
A MODEL CALL			
A One endpoint	Apps, agents and IDEs call the LLM gateway with their own identity. They ask for an alias such as chat, not a vendor model.	On-prem LLM gateway	Caller → LLM gateway, HTTPS
B Self-hosted, on-prem	Routine and sensitive traffic goes to self-hosted models on on-prem GPUs.	On-prem GPU pool	Never leaves the building
C Self-hosted, cloud	Extra capacity for the same model, served in the cloud region on a static route, behind that site's internal routes.	Cloud region	LLM gateway → the cloud site's internal routes, over the WAN
D Public API	Used only when a routing rule allows it. The provider key stays in the gateway.	Model provider	LLM gateway → internet

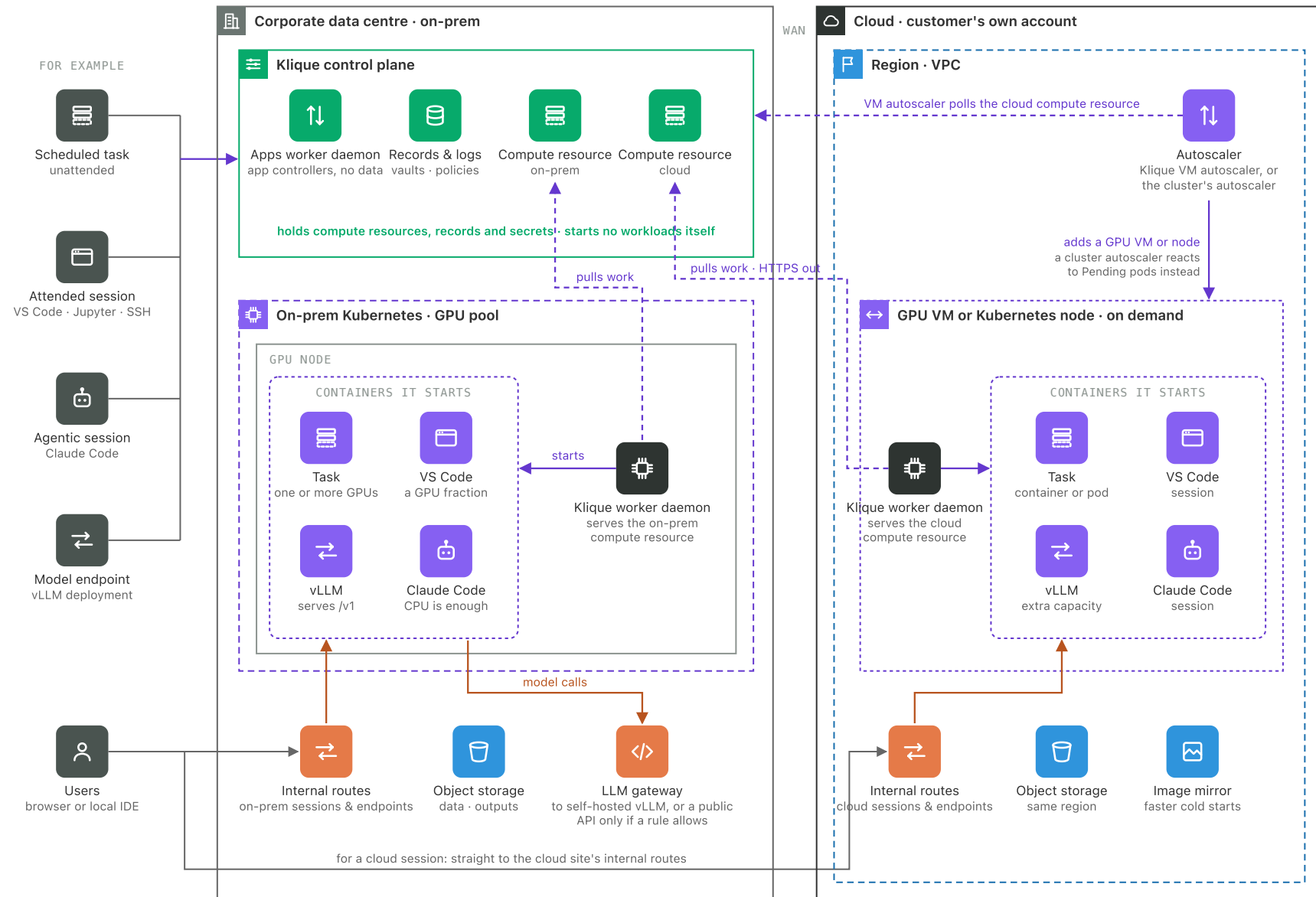
Every model call, B, C or D, is metered on the same ledger as GPU-hours: by user, project, model and the rule that routed it.

Business value

- **For IT and platform teams: placement is a policy, not a project.** The estate is described once. IT decides where work lands and in which order, on-prem first with the cloud for overflow or the reverse, with no re-platforming each time.
- **For builders: self-service GPUs, wherever they run.** Data scientists and developers request compute from the tools they already use: the UI, notebooks, the SDK, pipelines or agents. The experience is the same on-prem or in the cloud, with no tickets and no infrastructure knowledge needed.
- **For finance and leadership: one ledger for GPU-hours and tokens.** Every GPU-hour and every token is attributed to a user and a project, so owned capacity is used first and cloud spend is deliberate.

Where compute runs

For any sovereignty review, the first question is where compute runs. Every **compute provisioning task** works the same way: scheduling a task, launching an attended or unattended session, launching an agentic session, or deploying a model endpoint. In this scenario the container can start in only two places: an on-prem GPU node, or a GPU VM or Kubernetes node in the customer's own cloud account. **Policy decides which**, never the workload itself: a team can only send work to the compute resources it has been granted, and each of those is served in one place. The control plane never runs the workload. A small worker daemon at each site pulls the task and starts the container there.



Where a workload runs is set by policy, not by the workload. Whatever is provisioned, from any app in the catalogue, policy decides which compute resources a team may use. Each compute resource is served in one place: the customer's own GPU nodes, or GPU VMs or Kubernetes nodes created on demand in the

customer's cloud account. Either way, users connect at the site where the workload runs, not through the control plane.

Policy decides the location

A task, session or endpoint is the same on-prem or in the cloud. Policy decides which compute resources a team may use. Each one is served by worker daemons in one place, and only those worker daemons can start a container.

The control plane orchestrates, the sites compute

It keeps the compute resources, records, policies and secrets. For apps like VS Code or vLLM, a small controller process runs beside it to manage the instance. That controller holds no data and does no compute.

Access goes to where it runs

Sessions and endpoints are reached through the internal routes at their own site. An on-prem session never touches the cloud. An agentic session's model calls go through the LLM gateway, which keeps prompts on self-hosted models unless a rule allows a public API.

COMPUTE PROVISIONING TASK	SPINS UP AS	SENT TO AN ON-PREM COMPUTE RESOURCE	SENT TO A CLOUD COMPUTE RESOURCE	HOW USERS REACH IT	LEAVES THE SITE
Scheduled task unattended: training, batch, pipelines	One container for the task	A pod on an on-prem Kubernetes node, a Docker container on a bare-metal host, or a Slurm job	A container on a GPU VM, or a pod on a GPU node in the customer's managed Kubernetes cluster, added on demand	Not reached directly: progress shows in the UI	Logs, metrics, pointers to outputs
Attended session VS Code · Jupyter · SSH · desktop	A controller on the apps worker daemon, plus the session container on a GPU node	On-prem node, often a fraction of a GPU	Cloud VM	Browser or local IDE → that site's internal routes → the container	Session status. Files and keystrokes stay at the site.
Agentic session Claude Code and other coding agents	A controller, plus the agent's container. CPU is often enough.	On-prem node	Cloud VM	Browser terminal through that site's internal routes	Model calls go to the LLM gateway: a self-hosted model, or a public API only if allowed
Model endpoint vLLM, SGLang, NIM deployments	A controller, plus the model-server container. Weights are pulled onto that node.	On-prem GPU node	Cloud GPU VM, for extra capacity	An OpenAI-style URL on a static route at that site, usually listed behind the LLM gateway	Token usage. Prompts stay where the model and its callers are.

These rows are examples. Every workload in the app catalogue is provisioned the same way.

Sovereignty is enforced by the compute resource: if a team's work must never run in the cloud, don't give that team a cloud compute resource. Policies decide which compute resources each group can even see. A compute resource with no cloud pool behind it can only start containers on-prem.

The building blocks: one control plane, many pools

In this scenario the control plane lives in the customer's data centre. On-prem GPUs are grouped into **resource pools**: Kubernetes, bare metal or a Slurm partition. The cloud region is simply **another pool**, with an autoscaler in front of it. Every pool is served the same way. A small **Klique worker daemon** on each node polls the control plane for work, runs it and reports back.

A cloud account doesn't "connect to" Klique the way a cluster joins a network. It holds nodes that run a worker daemon, and each worker daemon asks the control plane for work. Adding a region, a second cloud or a neocloud is the same move each time: install the worker daemon, name the pool, link it to a profile. The network only has to work in one direction: the cloud side must be able to reach the control plane.

Autoscaler architecture

Klique bursts into a cloud account through VM autoscaling or managed Kubernetes. With **VM autoscaling**, a Klisque autoscaler instance creates GPU VMs directly. With **managed Kubernetes** (EKS on AWS, AKS on Azure, GKE on Google Cloud, or a neocloud's managed Kubernetes), the Klisque worker daemon runs inside the customer's cluster and the cluster's own node autoscaler adds GPU nodes. Both serve the same cloud compute resource, so users see no difference.

VM autoscaling

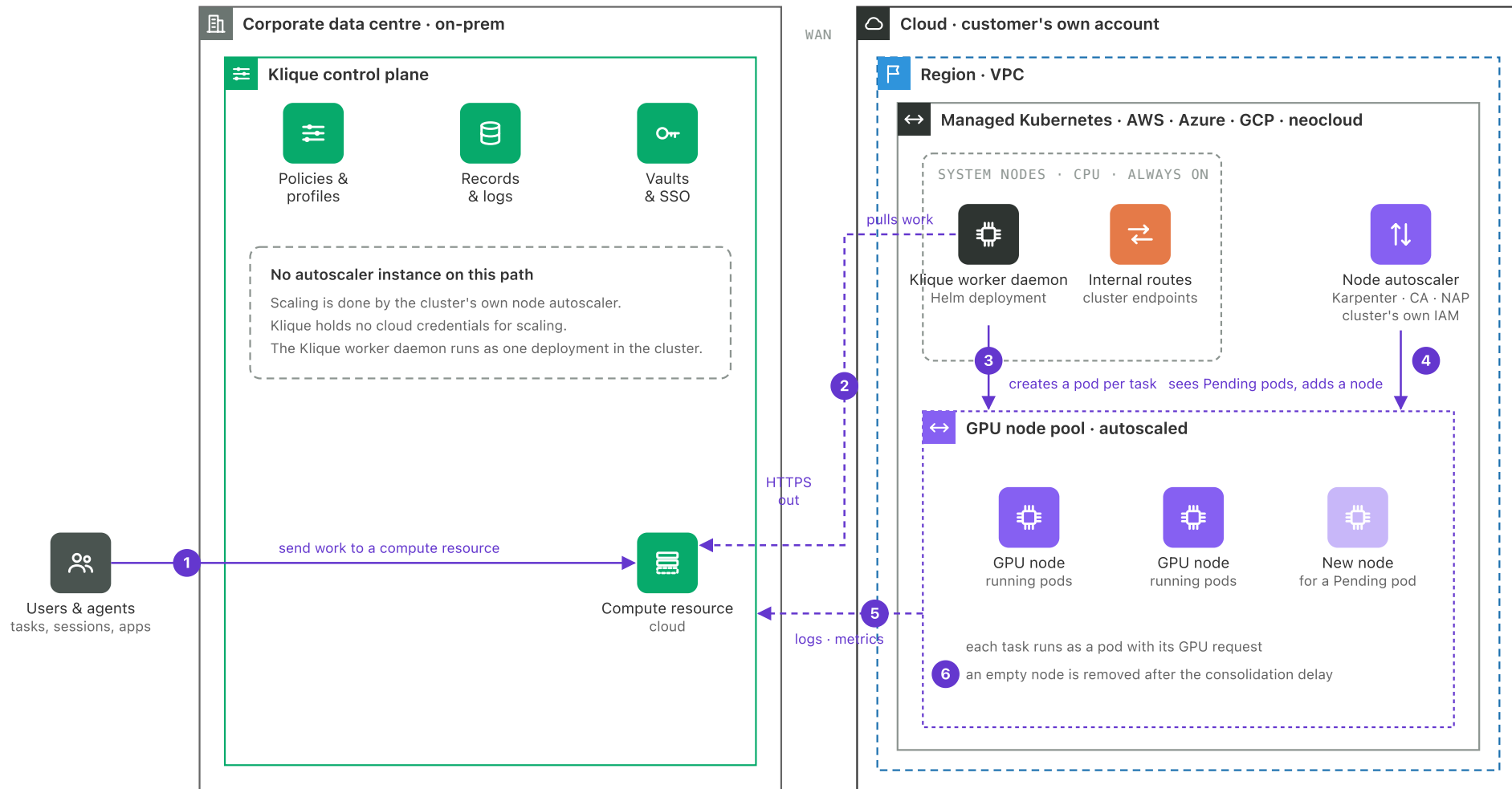
VM autoscaling has **three moving parts**. The **autoscaler instance** is defined in the control plane. An **apps worker daemon** runs it. The **GPU VMs** it creates are the only components that take work from the cloud compute resource, and they are removed again when idle.

STEP	DETAIL	LOCATION	TIMING
2 Launch	Launching it places it on the internal apps queue. At that point it is a record and a request.	Control plane	Once
3 Start	An apps worker daemon pulls it and runs the autoscaler as a small container. With the worker daemon on a VPC instance with an IAM role, no cloud keys are stored.	Always-on CPU instance in the VPC, or on-prem	Always running
4 Poll	The autoscaler checks the cloud compute resource about once a minute and registers as its listener, so work can be admitted to a pool with no VMs yet.	Autoscaler → control plane	About every minute
5 Send work	Users send tasks, sessions or apps to the cloud compute resource, or on-prem overflow sends them there.	Control plane	On demand
6 Launch a VM	Work is waiting and no VM is free, so the autoscaler calls the cloud API. It tries spot first if configured, then falls back to on-demand.	Autoscaler → cloud API	Within about a minute
7 Boot	The VM starts from a prepared image, and its start-up script launches a Klique worker daemon that serves the cloud compute resource.	GPU VM	A few minutes, image pull included
8 Run & retire	The worker daemon pulls the work over outbound HTTPS and starts the container. A VM idle past the timeout is terminated by the autoscaler.	GPU VM → control plane	Until idle, 15 min by default

WHERE THE AUTOSCALER RUNS	ON-PREM, ON THE PLATFORM'S APPS WORKER DAEMON	IN THE VPC, ON AN APPS WORKER DAEMON WITH AN IAM ROLE
Cloud credentials	Access keys, read from a vault at start	None stored. The instance's IAM role grants run, describe and terminate instances, tags and spot requests.
Network path	Outbound from the data centre to the cloud API	API calls stay inside the cloud provider. Only outbound HTTPS to the control plane crosses the WAN.
Extra always-on cost	None: it uses the control plane's apps worker daemon	One small CPU instance, running all the time
Also runs there	The apps worker daemon runs the controller for every app launched through it, not only the autoscaler. Choose its location on purpose. Controllers hold no data, but a security review will still want to know where they are.	

Managed Kubernetes

When the customer already runs Kubernetes in the cloud, Klique uses it as the cloud pool. The **Klique worker daemon** runs as one deployment in the cluster and turns each task into a pod. The **cluster's own node autoscaler** (Karpenter or Cluster Autoscaler on EKS, the AKS cluster autoscaler, node auto-provisioning on GKE, or a neocloud provider's equivalent) adds GPU nodes for pods that can't be placed. No autoscaler instance is involved, and Klique holds no cloud credentials for scaling.



STEP	DETAIL	LOCATION	TIMING
1 Send work	Users send tasks, sessions or apps to the cloud compute resource, or on-prem overflow sends them there.	Control plane	On demand
2 Pull	The Klique worker daemon in the cluster pulls the task over outbound HTTPS.	Cluster, system nodes	Seconds
3 Create a pod	It creates a pod with the GPU request from the compute resource's pod template. With no free GPU node, the pod waits as Pending.	Cluster	Seconds
4 Add a node	The cluster's node autoscaler sees the Pending pod and adds a GPU node from the node pool, using the cluster's own IAM.	GPU node pool	A minute or two, plus the image pull
5 Run & report	The pod runs. Logs, metrics, status and output pointers stream home.	GPU node → control plane	Until done
6 Scale down	When a node has no pods left, the cluster removes it after its consolidation delay.	GPU node pool	Minutes after idle

Fractional GPUs on managed Kubernetes: when every fractional slot on the running nodes is taken, the fractional-GPU admission check can reject a new pod instead of leaving it Pending. The node autoscaler then never sees the demand. Keep a small warm GPU node pool for fractional work, or run fractional work on-prem.

Choosing between them

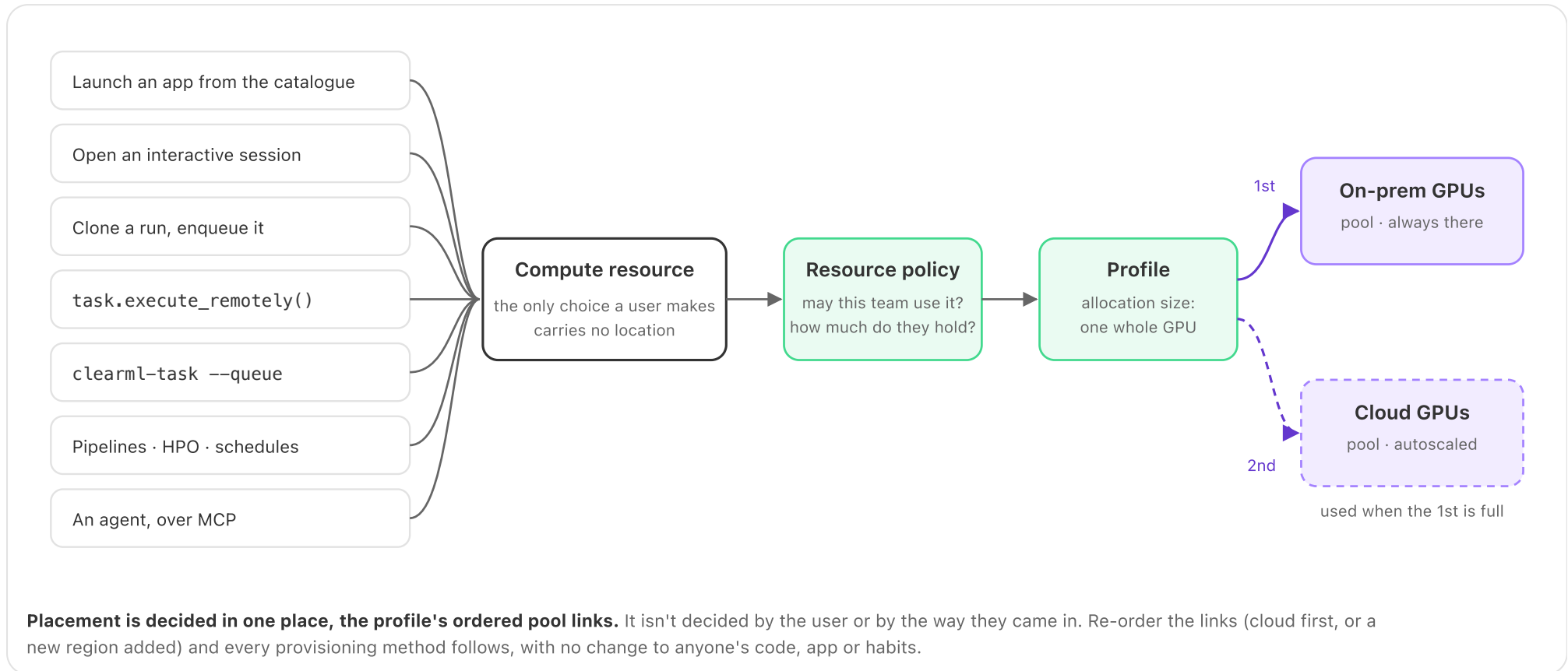
Both variants serve the same cloud compute resource, so users see no difference. The choice is about what the customer already runs and who should hold the scaling permissions.

	VM AUTOSCALING	MANAGED KUBERNETES
Klique in the cloud	An autoscaler instance run by an apps worker daemon (on-prem with keys from a vault, or in the VPC with an IAM role), plus a worker daemon on every VM	One Klique worker daemon deployment on the cluster's system nodes, installed with Helm
Scales	GPU VMs	GPU nodes in a node pool. Each task runs as a pod

	VM AUTOSCALING	MANAGED KUBERNETES
Trigger	Work waiting on the cloud compute resource, polled about once a minute	A Pending pod that no GPU node can fit
Cloud credentials	Run, describe and terminate instances; tag them; request spot	None held by Klique. The cluster's autoscaler scales with its own IAM
Removes	A VM idle past the timeout (15 minutes by default)	An empty node, after the cluster's consolidation delay
Packing	Typically one task per VM	Several pods per node. Fractional GPUs possible
Serving endpoints	Internal routes in the VPC	Internal routes inside the cluster
Clouds	AWS, GCP, OCI, Azure, neoclouds	EKS, AKS, GKE and neocloud managed Kubernetes with a node autoscaler.
Best for	The simplest burst into a cloud account, with no cluster to run	Teams already running Kubernetes in the cloud, dense packing

Many ways to provision, one compute resource

Compute is provisioned in many ways: from the app catalogue, in an attended session, from a script through the SDK or CLI, by a pipeline or schedule, or by an agent over MCP. Each one ends at the same place, a **compute resource**. It's the one choice made at provisioning time, and it carries no information about where the work will run. The [GPU as a Service Reference Architecture](#) covers each way to provision in more detail.



Launch an app

From the UI, for consumers and builders: pick an approved app (vLLM, SGLang or NVIDIA NIM for serving, JupyterLab or VS Code for development, Claude Code for agentic work) and choose a compute resource from a dropdown. The form never mentions a cluster, a node or a container.

Interactive sessions

Interactive, for builders: VS Code, Jupyter, SSH or a remote desktop, opened on platform GPUs rather than a laptop. A fraction of a GPU is often enough, and the session stays on it until the user closes it.

Clone & enqueue

Non-interactive, for builders: take any past run, change a parameter in the UI and send it to a compute resource. It's the same code with a different config. It runs unattended and reports back as it goes.

From an existing script

From the SDK: two lines to track a script, one more to hand it to the pool. The run stops locally and resumes on the GPU the compute resource provides.

```
from clearml import Task
task = Task.init(project_name="fraud-models", task_name="train")
task.execute_remotely(queue_name="compute resource") # continues on the pool
```

Any script, no code change

From the CLI: wrap an existing repo or script and send it to a compute resource straight from the terminal or a CI pipeline.

```
clearml-task --project fraud-models --name nightly \
--repo git@git.corp:ml/fraud.git --script train.py \
--queue "compute resource"
```

Pipelines, HPO & schedules

Automation: each pipeline step, each hyperparameter trial and each scheduled or triggered run names its own compute resource. Prep on CPU, train on several GPUs, evaluate on a slice.

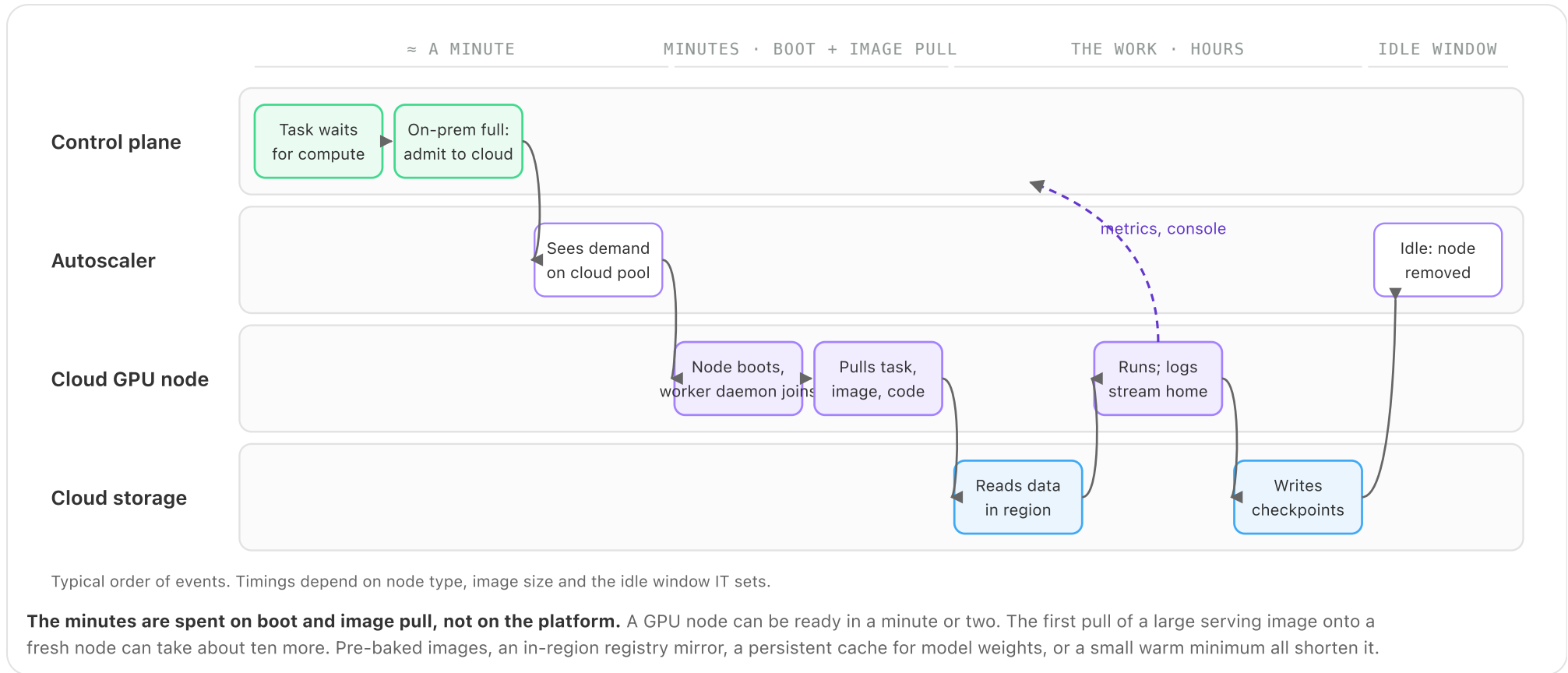
```
@PipelineDecorator.component(execution_queue="compute resource")
def train(dataset_id): ...
```

An agent, over MCP

Agentic: the platform is also a tool an agent can call. From Claude Code, a desktop assistant or a team's own agent, it can launch an app, enqueue a run, or read back a run's results. It works under the same identity, policy and attribution as the person who owns it. The agent never gets its own route to the hardware.

A cloud burst, step by step

Follow one training task. The on-prem GPUs are all busy, so the task overflows to the cloud region. The timeline shows each component's part and its typical duration.



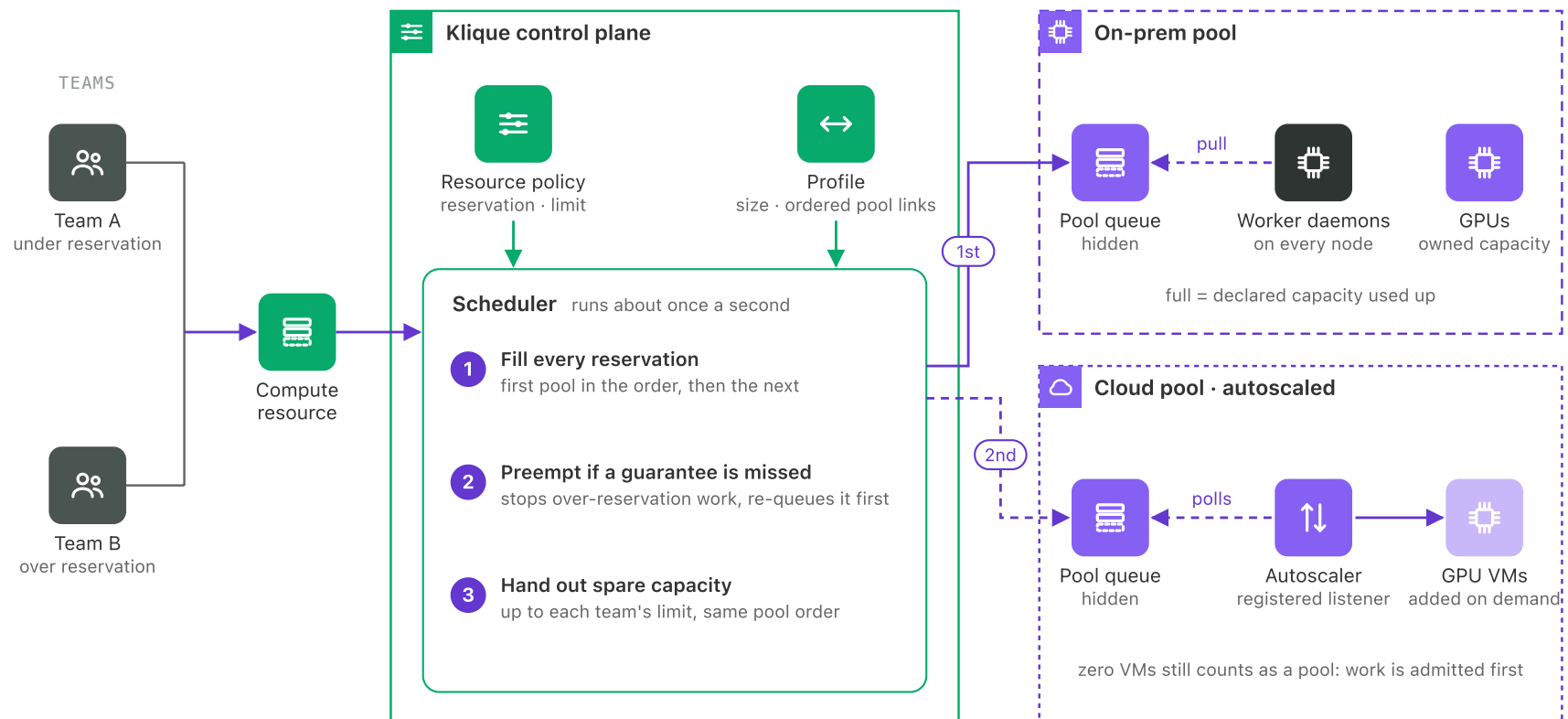
Scheduler

Behind every compute resource sits a small scheduler that runs about once a second. It works in three passes, and the order drives hybrid placement:

- 1. **Fill every team's reservation first.** Try each profile's first pool, then its second, then the rest.
- 2. **Make room if a guarantee is being missed.** If a team under its reservation is waiting and the pools are full, work from teams running over theirs is stopped and put back at the front of its queue.
- 3. **Then hand out the spare,** up to each team's limit, again first pool before second.

"Full" means the pool's *declared* capacity, the number IT enters, is used up. When the scheduler picks a pool, it moves the task into a hidden queue that only that pool's worker daemons listen on. A cloud pool with no nodes yet still counts, because its autoscaler registers as the listener on that queue. This lets a pool **scale from zero**: the work is admitted first, and the node that runs it arrives next.

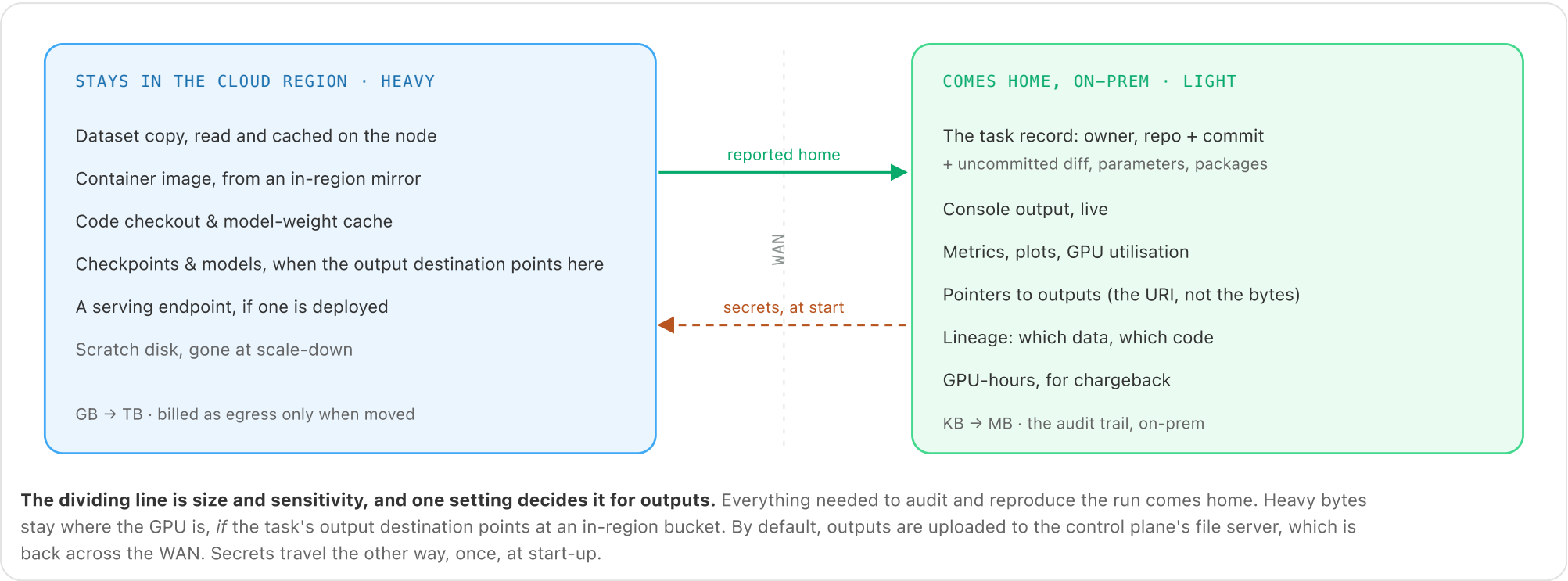
The same scheduler places work across Kubernetes, Slurm, bare-metal and cloud pools, covered in the [Meta-Scheduler Reference Architecture](#).



The scheduler decides, the pools execute. Policy and profile feed one scheduler in the control plane. Each pool has its own hidden queue, served only by that pool's worker daemons, or in the cloud by the autoscaler that registers as its listener.

When it runs in the cloud, where does the work live?

Every security and data team asks this question. **The control plane holds the record of the work, not the work itself.** Heavy bytes (data, checkpoints, weights) stay beside the GPUs that use them. The data that comes home is small: parameters, logs, metrics, lineage, and *pointers* to where the outputs were written.



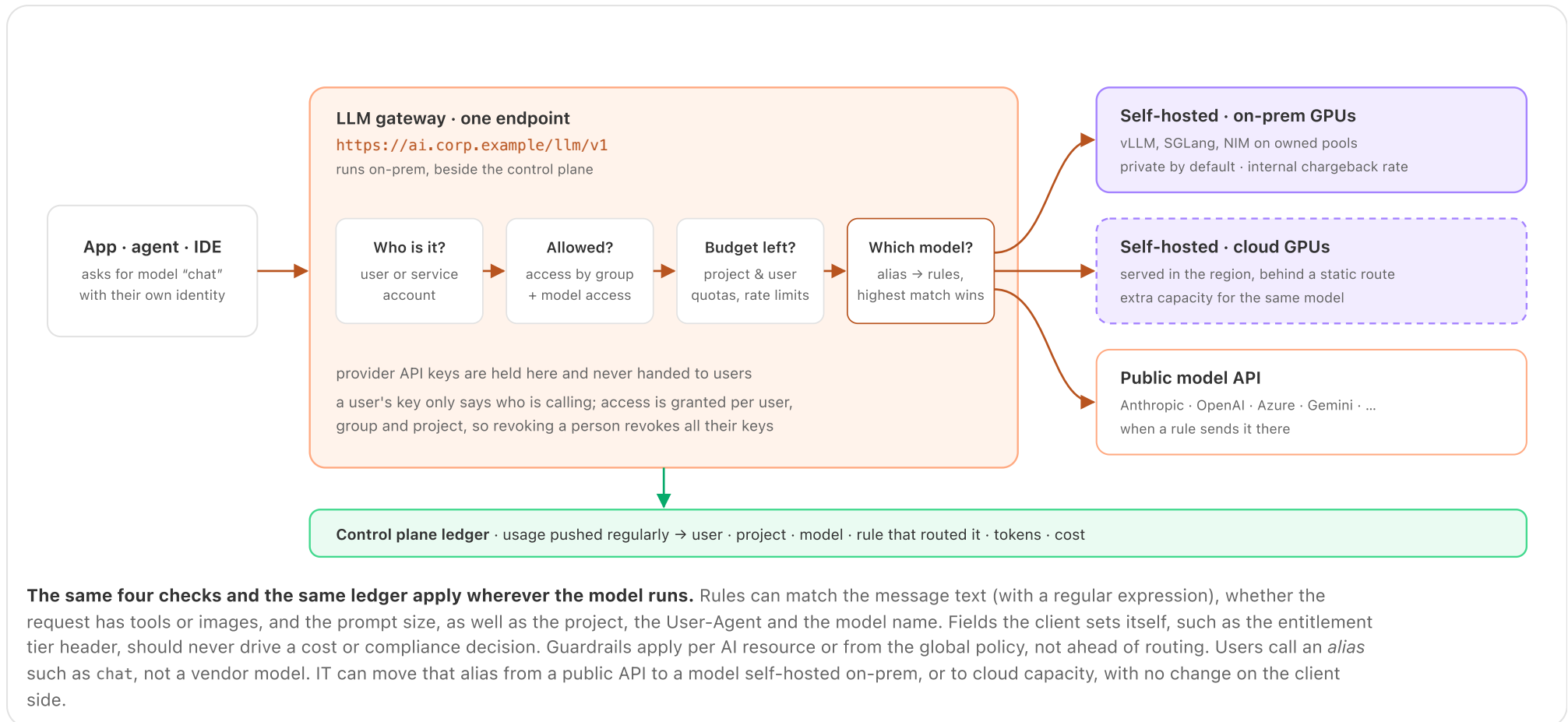
ASSET	LOCATION DURING A CLOUD RUN	CROSSES BACK?
Task record	Created on the control plane when the task is defined. It holds the repo, commit and uncommitted diff, parameters, packages, and who ran it.	Lives on-prem
Code	Cloned onto the node from the git server at start, and any uncommitted changes recorded on the task are applied. Or it's already baked into the container image. The node needs a route and credentials to the git server.	No, stays in the region

ASSET	LOCATION DURING A CLOUD RUN	CROSSES BACK?
Environment	Rebuilt on the node from the recorded packages, or taken from the image, then cached for the next task on that node.	No, stays in the region
Container image	Pulled from a registry. Point cloud pools at an in-region mirror to cut cold starts and egress.	No, stays in the region
Input data	A versioned dataset: metadata on the control plane, files in storage. It's fetched and cached on the node. For heavy data, store the dataset in an in-region bucket, or tell cloud workers to read the registered location from a local mirror.	No, stays in the region
Secrets	Held in vaults on the control plane and injected into the task at start. They're never baked into images or code.	No, flows out only
Outputs	Checkpoints, artifacts and models go to the task's <i>output destination</i> . The default is the control plane's file server, back across the WAN. Set it per project or per pool to an in-region bucket to keep bytes in the cloud, or to on-prem storage for anything that must come home. The control plane records where they went.	Depends on the output destination
Model weights	For serving, downloaded at start from the model registry or Hugging Face. Put the cache on a persistent volume, or every cold node downloads them again.	No, stays in the region
Logs & metrics	Streamed to the control plane while the task runs, so a cloud task can be followed exactly like an on-prem one. Anything a task prints comes home too, so a task should never print secrets or raw records.	Yes, streamed home
A model endpoint	Runs on the cloud node, on a static route behind that site's internal routes. The token traffic it serves is metered centrally (section 07).	Usage records only
Scratch disk	Local to the node. Anything not written to an output destination is lost when the node is removed.	No, removed at scale-down

For regulated data: the same mechanics run in reverse. Link a sensitive project's profile only to in-country or on-prem pools and keep its output destination on-prem. Its work then never lands in a cloud region, whichever way a user submits it.

Tokens: one gateway for every model

GPUs are one way to consume AI. Tokens are the other. Every model call, to a self-hosted model or to a public API, goes through **one endpoint**, the LLM gateway. The gateway checks who is calling, whether they're allowed, and whether their team has budget left. It then chooses a model from rules IT has set, and records the cost against the user and project. In this scenario the gateway runs on-prem, so every call enters through the customer's data centre, including calls to extra model capacity in the cloud region or to a public API.



Where a token request goes, case by case

THE MODEL RUNS...	PATH OF THE REQUEST	METERED	GOOD FOR
On-prem GPUs	Client → gateway → a self-hosted model server on owned pools. The request stays in the building.	On-prem, at the internal rate	Routine and sensitive traffic, the default
Cloud GPUs	Client → LLM gateway → WAN → the cloud site's internal routes → the model server's static route in-region	On-prem, same ledger	Extra serving capacity, users near that region
Public API	Client → gateway → provider. The gateway swaps the user's identity for the provider key it holds.	On-prem, at the provider's price	Frontier quality when a rule or a team's entitlement calls for it

Putting a self-hosted model in front of callers follows the same order on-prem or in the cloud region. 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.

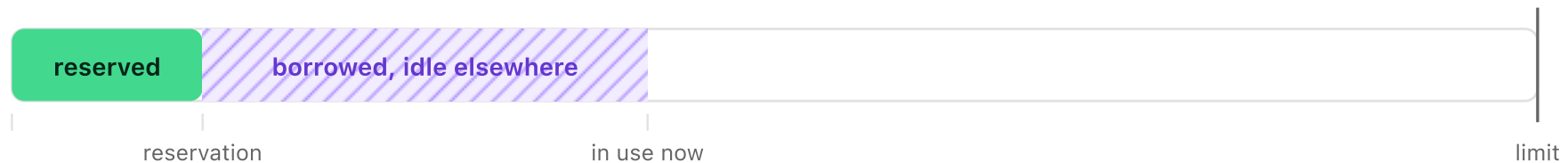
Coding agents and desktop tools that talk to a provider directly can be pointed at the same endpoint, so their tokens land on the same ledger instead of on someone's personal provider account.

Routing rules, guardrails and budgets sit behind that alias. The [LLM Gateway and Tokenomics Reference Architecture](#) covers them.

Compute policy engine

The compute policy engine keeps owned GPUs busy first and spends cloud capacity on purpose. It has two sets of controls, one for GPUs and one for tokens, and both write to the same ledger.

AI Engineering team · resource policy on a compute resource



Reserved is a guarantee, not a fence: when other teams' GPUs sit idle, this team can run past it, up to its limit.

If an owner needs a borrowed GPU back, the task holding it is stopped and re-queued at the front. Its checkpoint lets it resume.

Guarantee plus borrowing keeps utilisation high. On-prem GPUs rarely sit idle while work waits. The cloud pool only grows once owned capacity is genuinely full.

GPU controls

Policies give each team a reserved amount and a limit, over the compute resources they're allowed to use. When several teams want the same GPUs, **execution priority** decides who goes first. **Profiles** decide the size of the allocation: a GPU fraction, a whole GPU, or several GPUs on one node. Give each compute resource a usage weight, so a fractional GPU counts as that fraction of a GPU-hour.

Token controls

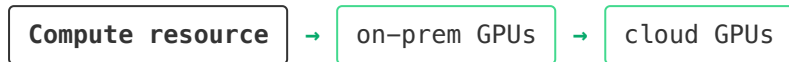
A **budget per project**, daily, weekly or monthly, ties spend to an outcome with an owner. **Per-user limits** stop one person running away with it. **Rate limits** protect shared models. **Guardrails** are set in the global policy or per AI resource: deny and allow patterns at the gateway's edge, and the NIM Guardrails app for a model-based check of input and output.

Why per project matters: a project is a named piece of work with an owner. Put its GPU-hours and tokens next to its results, and the conversation turns to return on investment rather than whose usage went up.

Placement patterns

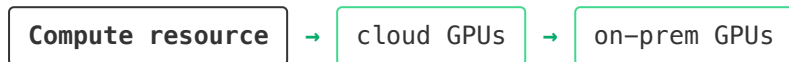
The scenario here is the first pattern below: on-prem first, cloud for overflow. The same building blocks support other strategies. Each one is a profile's ordered pool links, so changing strategy means reordering links, not moving workloads.

On-prem first, cloud for overflow



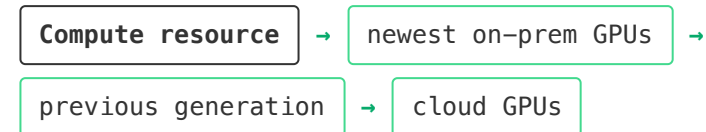
The default for most customers. Owned GPUs are always used first, and the cloud absorbs peaks such as quarter-end retrains or a new model launch.

Burn down a cloud commitment



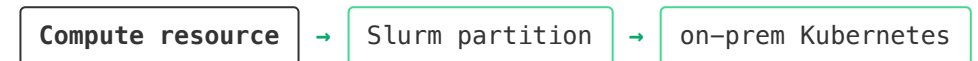
For a committed-spend agreement that has to be used up, or for moving off the cloud gradually as on-prem capacity arrives. Flip the order back later.

Specify capability, not a GPU model



A profile can specify GPU memory rather than a GPU model. When the newest GPUs are busy, an older one with enough memory still qualifies, and cloud is the last resort.

Train on HPC, serve on Kubernetes



Training runs that belong on the cluster don't have to wait for it. The endpoint that serves the result launches from the same catalogue onto Kubernetes.

Sovereign by policy



For EU projects only. A regulated project can only reach in-country pools, and its outputs are written to in-country storage. Only metadata reaches the control plane. A second site in another jurisdiction is just another pool with its own worker daemons: its models and data stay there, and only reports come home.

Design considerations for cloud burst

Decide these on purpose before the first burst, rather than finding them out in production.

Cold start is minutes, not seconds

Node boot plus image pull dominates, then model weights. For latency-sensitive work, keep a small warm minimum, pre-bake images, mirror the registry in each region, and put weight and package caches on persistent volumes.

Move compute to data, not data to compute

Egress is billed per gigabyte and large datasets are slow to move. Keep a versioned copy in each burst region, and let the dataset version pin exactly which bytes were used.

Set the output destination per pool

By default every checkpoint travels home to the file server, and each one is billed as egress. Cloud scratch disk disappears at scale-down. Point cloud pools at an in-region bucket, and checkpoint long runs so preemption or spot reclaim costs minutes, not hours.

Tune the idle window

Short windows save money but cause more cold starts. Long windows do the reverse. Set it per pool: short for bursty training, longer for serving and interactive pools.

Reachability runs one way

Cloud worker daemons need outbound HTTPS to the control plane's API and file server (over VPN, private link or a published endpoint), and to the git server, registry and package mirrors. Batch work needs no inbound connection into the cloud. A site that serves endpoints or sessions needs its own internal routes and ingress. Those routes are the only inbound path to a site.

Mind the WAN hop for tokens

Calls to a cloud-hosted model go through the one LLM gateway. Put that gateway near the heaviest callers, and keep chatty, latency-sensitive models close to them.

Keep location out of the name

Users only ever see a compute resource. Its name describes the allocation, never the location. That way IT can reorder pools, add a region or change clouds without anyone changing their code or habits. Offer users only these compute resources, so every task goes through policy.

Keep declared capacity accurate

The scheduler overflows once a pool's declared capacity is used, not by probing the hardware. Keep those numbers true, and retire any compute resource no node serves, or work will wait on it.

Secrets come from vaults, not images

Inject credentials at run time from group vaults. Never bake keys into an image or pass them as a launch parameter, where they would be recorded on the task for anyone who can read it.

Connections

This table gathers the connections a cloud burst needs. Users sign in through SSO, and agents over MCP through OAuth. Worker daemons authenticate with the credentials they were registered with.

Budget at two levels

GPU policies (reserved, limit, priority) and token budgets (project, user, rate) cover different risks. Cloud spend also has hard ceilings: the autoscaler instance's maximum instances, and the resource policy's limit on the cloud compute resource. Set both levels, and review them against the same per-project ledger.

When something fails

When spot capacity is unavailable, the autoscaler falls back to on-demand. When the cloud pool has no VM free, work waits on the compute resource until the autoscaler adds one, up to its maximum instances. If the apps worker daemon that runs the autoscaler is down, no new VMs are added until it is back. Long runs on spot should save checkpoints, so they can resume.

FROM	TO	DIRECTION	PURPOSE
Users, apps and agents	Control plane: UI, API, MCP	Into the control plane, HTTPS	Send work to compute resources, manage it, read results
Worker daemons, on-prem and cloud	Control plane API and file server	Outbound HTTPS from the site	Pull work, stream logs and metrics, upload outputs when the output destination is the file server
Autoscaler	Cloud API	Outbound, from the VPC or from on-prem	Run, describe and terminate instances, tag them, request spot
Users	The site's internal routes	Into the site: the only inbound path to a site	Reach sessions and endpoints running there
LLM gateway	Cloud-site models and public model APIs	Outbound from on-prem	Route model calls. Provider keys stay in the gateway
GPU nodes	Git server, registry and package mirrors	Outbound from the node	Clone code, pull images and packages

Glossary

Control plane	The Klique server: UI, API and MCP, compute resources and scheduler, policies, identity, vaults, and the metadata and usage ledger. It holds intent and records; the data stays where the work runs.
Klique worker daemon	The small worker process on every GPU node, whether VM, bare metal, Kubernetes or Slurm. It polls the control plane for work over outbound HTTPS, runs it, and reports back.
Compute resource	The target a user sends work to, from any provisioning method. Under the hood it's a queue. It's the only piece of this architecture a user ever sees.
Resource pool	A set of nodes running worker daemons in one place: an on-prem cluster, a Slurm partition, or an autoscaled cloud account.
Profile	The allocation a task receives (a GPU fraction, a whole GPU or several GPUs), linked to one or more pools in priority order.
Resource policy	Which teams may use which compute resources, with a reserved amount, a limit, and an execution priority.
Autoscaler instance	The autoscaler's definition, stored in the control plane: the compute resources to poll, instance types, limits, idle timeout and credentials.
Autoscaler	The running process that creates GPU VMs when work waits on the compute resource it polls, and removes them when they go idle. On Kubernetes, the cloud's own node autoscaler takes this role.
Apps worker daemon	The component that runs autoscalers and app controllers. By default it sits beside the control plane. It can also run on a small instance in the customer's VPC with an IAM role.

Internal routes	The routes to sessions and endpoints at each serving site, with access control. A self-hosted model's fixed address on them is a static route.
LLM gateway	The single endpoint for every model call. It handles identity, access, guardrails, budgets, routing rules, and metering.
Alias	A model name that callers use, such as chat or embed, that IT can point at any model without clients changing.
Output destination	Where a task writes checkpoints, artifacts and models: an in-region bucket or on-prem storage. The control plane stores the pointer.

Klique AI · Cloud autoscaler reference architecture · control plane on-prem, GPU work bursting to a cloud region. Pool names, compute resource names, instance types and numbers are illustrative; every estate differs.