

Multi-tenant reference architecture

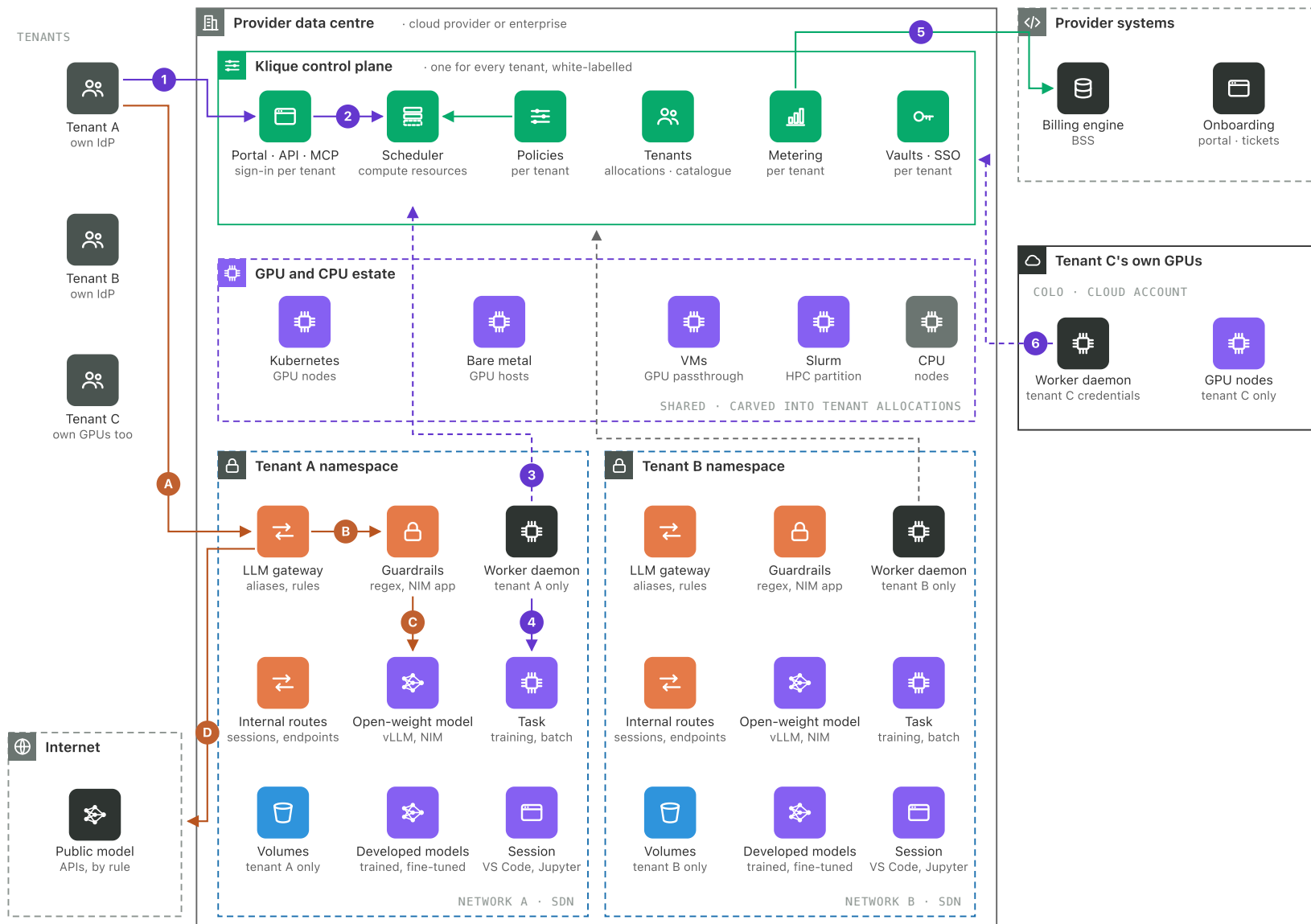
This reference architecture covers the following scenario: a sovereign cloud provider, or an enterprise serving its own business units, runs the Clique control plane in its own data centre, under its own brand, and offers GPUs, tokens and AI tooling to many tenants from a shared estate. The design is the same either way; only the billing relationship changes. It shows how the components fit together, how tenants are kept apart, and how every GPU-hour and every token is governed, metered and billed.

CONTROL PLANE One, in the provider's data centre, white-labelled	GPU CAPACITY A shared estate carved into tenant allocations, plus tenants' own GPUs
TENANTS Enterprise customers, or internal business units, each isolated, each billed	AUDIENCE IT architects and sales engineers

Assumption: the provider runs its own Kubernetes and/or bare-metal environment, on a software-defined network (SDN), or a CNI that enforces network policies, that can give each tenant its own network. Clique places each tenant's workloads on that network; the environment and the network layer are the provider's.

Terms: the *provider* is whoever runs the platform, a cloud provider or an enterprise's central platform team. A *tenant* is its customer or its business unit. Where the provider invoices a customer, the enterprise charges a cost centre.

How to read the diagram: numbered badges follow a compute provisioning task for tenant A, lettered badges follow one of tenant A's model calls.



STEP	DETAIL	LOCATION	CONNECTION
A COMPUTE PROVISIONING TASK, TENANT A			
1 Sign in & submit	A tenant A user signs in through tenant A's own identity provider and provisions work on a compute resource: from the portal, a session, the SDK or an agent over MCP.	Control plane	User → portal, HTTPS
2 Check the policy	The scheduler checks tenant A's team policies, inside the allocation the provider granted tenant A.	Control plane	None: a scheduling decision
3 Pull	Tenant A's Klisque worker daemon, in tenant A's namespace and registered with tenant A's credentials, pulls the task. No other tenant's worker daemon can see it.	Tenant A namespace	Worker daemon → control plane, outbound
4 Run on the allocation	The worker daemon creates the pod from the compute resource's template. Node selectors place it on tenant A's allocated GPU nodes, on tenant A's network.	Shared GPU estate	Stays on network A
5 Meter & bill	GPU-hours are metered per tenant and per GPU class, priced from the provider's price book, and exported as billing events to the provider's billing engine.	Control plane → provider systems	API
6 Own GPUs	Tenant C's own GPUs run a worker daemon registered with tenant C's credentials. They serve tenant C only and dial out to the same control plane.	Tenant C's site	Worker daemon → control plane, outbound HTTPS
A MODEL CALL, TENANT A			
A One endpoint	Tenant A's apps, agents and IDEs call tenant A's own LLM gateway, with the user's identity and an alias such as chat.	Tenant A namespace	Caller → gateway, HTTPS
B Guardrails	Input checks run on the call: deny or allow regex patterns at the gateway's edge, and model-based checks through the NIM Guardrails app, a NeMo Guardrails proxy with a safety NIM, on tenant A's allocation. A PII check is one example, written as a pattern or run by the app.	Tenant A namespace	Stays on network A
C Self-hosted model	By default the gateway routes to an open-weight or developed model served on tenant A's own allocation.	Tenant A namespace	Stays on network A

STEP	DETAIL	LOCATION	CONNECTION
D Frontier API	Only when one of tenant A's rules allows it. The provider key stays in the gateway.	Internet	Gateway → internet

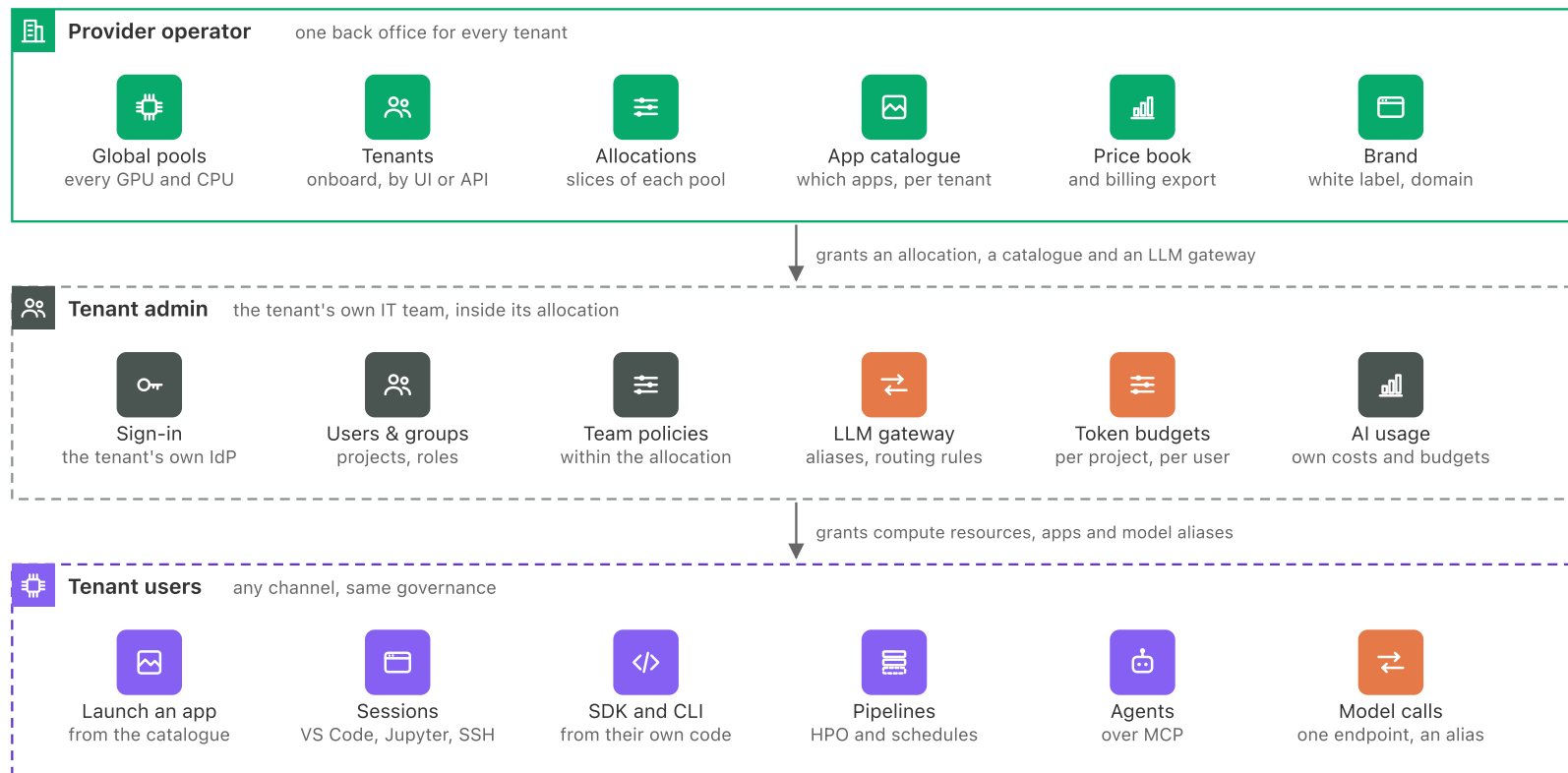
Every GPU-hour and every token lands on tenant A's own ledger, and on the provider's bill for tenant A.

Business value

- **For the provider: one estate, many tenants.** Tenants are onboarded from the management center or the API. The provider carves pools into allocations, chooses each tenant's apps and sets the price book, running one control plane under its own brand.
- **For each tenant's IT: a governed AI factory, without building one.** Each tenant gets its own sign-in, network, LLM gateway and budgets. The tenant admin runs its teams inside the allocation, with no view of any other tenant.
- **For finance: every GPU-hour and token on a bill.** Usage is metered per tenant, priced from the provider's own price book, and exported to the billing system the provider already runs.

Three layers of control

Multi-tenancy has three layers. The **provider operator** runs one back office for the whole estate, the Platform Management Center. Each **tenant** is a separate workspace with its own users, projects, compute resources and vaults, its own SSO (OIDC, SAML or LDAP) and login page, and its own **tenant admin**: the admins group inside the tenant, which works inside what the provider granted. **Tenant users** only ever see their own compute resources, apps and model aliases, through whichever channel suits them.



The provider grants an allocation, the tenant admin divides it between teams, and users consume it. Every layer can be driven from the UI or the API, so onboarding and allocation changes can come from the provider's own portal or ticketing system.

WHO	WORKS IN	DECIDES
Provider operator	Platform Management Center, each tenant's worker daemon configuration, or the API	Pools, tenants, allocations, app catalogue per tenant, price book, brand
Tenant admin	The tenant's own workspace	Users and groups from the tenant's IdP, team policies, LLM gateway rules, token budgets
Tenant user	Portal, sessions, SDK and CLI, MCP	Which compute resource, app or model alias to use

Carving the estate into tenant allocations

The provider owns the estate: Kubernetes, bare metal, VMs with GPU passthrough and Slurm partitions, from NVIDIA, AMD and other vendors. Worker daemons bring every pool under management, whatever the hardware. The provider carves each pool into **tenant allocations** through each tenant's worker daemon configuration: the node pools its pod templates may target, and how much each compute resource may run at once. Dedicated allocations carry only that tenant's pods. Burst is not a default: it is an allowance the provider decides to grant, tenant by tenant, to use free capacity above the reservation. Klique's Platform Management Center shows orchestration and usage per tenant. The provider then leaves the tenant admin to divide its allocation between teams with the same policy engine a single enterprise uses: a **reservation**, a **limit** and a **priority** per team.

SHARED GPU AND CPU POOLS · SET BY THE PROVIDER

Kubernetes GPU pool
shared nodes, fractional GPUs

Tenant A · reserved
held for tenant A

Tenant B · reserved
held for tenant B

Burst space
over-quota allowance for A and B, when free

Bare-metal GPU pool
whole hosts

Tenant A · dedicated hosts
no other tenant's pods

Tenant B · reserved
held for tenant B

Unallocated
room for the next tenant

VMs · GPU passthrough
one VM per tenant

Tenant B · dedicated VMs
billed monthly

Burst space
any tenant with an allowance, per GPU-hour

CPU pool
worker daemons, controllers, CPU tasks

Tenant A · reserved
worker daemons, app controllers

Tenant B · reserved
agentic sessions, data prep

Burst space
CPU tasks for any tenant, when free

INSIDE TENANT A'S ALLOCATION · SET BY TENANT A'S ADMIN

Tenant A's policies
reservation and limit per team

AI engineering team
reserved

Data science team
reserved

Borrowing
up to each team's limit, preempted if needed

Two levels of the same idea. The provider decides each tenant's share of each pool. Inside that share, the tenant admin decides each team's. Colour follows the tenant; hatching marks capacity used only when it is free.

ALLOCATION	THE TENANT GETS	HOW IT IS BILLED
Dedicated	Whole hosts or VMs that carry no other tenant's pods	Flat monthly fee per GPU
Reserved	A share of a shared pool, held for the tenant	Monthly commitment
Burst	Free capacity above the reservation, when available, for tenants the provider grants a burst allowance	Per GPU-hour, lower price, may wait
Fractional	Slices of a GPU: NVIDIA MIG or memory-capped sharing, AMD partitions	Per slice-hour

Name compute resources by what they give (a whole GPU, a slice, a node), never by where they run. The provider can then move a tenant between pools, or add a new one, with no change to anyone's code or habits.

Tenant isolation

Tenants share hardware and the control plane software. They never share identity, records, secrets, networks or model keys. On Kubernetes each tenant gets its own **namespace**, with its own **Klique worker daemon** and **internal routes**, registered with that tenant's credentials, on that tenant's own **network**.

	TENANT A · ITS OWN	SHARED · RUN BY THE PROVIDER	TENANT B · ITS OWN
IDENTITY	Own IdP, own users	One portal, a login per tenant	Own IdP, own users
RECORDS	Projects, tasks, models, datasets	One control plane, data per tenant	Projects, tasks, models, datasets
SECRETS	Vaults, storage credentials	The vault service	Vaults, storage credentials
COMPUTE	Own namespace and worker daemon	GPU and CPU pools, and their nodes	Own namespace and worker daemon
NETWORK	Own SDN network and policies	Physical network and fabric	Own SDN network and policies
ACCESS	Own internal routes	Ingress	Own internal routes
STORAGE	Own volumes and file-server space	Storage platform	Own volumes and file-server space
TOKENS	Own LLM gateway, keys, budgets	Provider-hosted models, if offered	Own LLM gateway, keys, budgets

Stricter tenants: dedicated hosts that carry only that tenant's pods, on the same estate.

Shared hardware, separate everything else. Tenants that need more get dedicated hosts that carry only their pods.

How isolation is enforced

The tenant is the isolation boundary. Each tenant is a company workspace with its own users, projects, records, compute resources, role-based access control (RBAC), vaults, storage credentials and volumes. A user belongs to one tenant, projects sit inside a tenant, and the queues under a tenant's compute resources are that tenant's own.

Worker daemons pull only their tenant's work

Each tenant's worker daemon is registered with that tenant's credentials and pulls that tenant's tasks only. Its RBAC is namespace-scoped by default, so it creates pods in the tenant's own namespace.

Access checked on every route

Sessions and endpoints are reached through the tenant's own internal routes, which check access on every route with access tokens.

Its own sign-in

Each tenant has its own SSO, through OIDC, SAML or LDAP, and its own login page. The tenant admin is the admins group inside the tenant.

The provider's stack

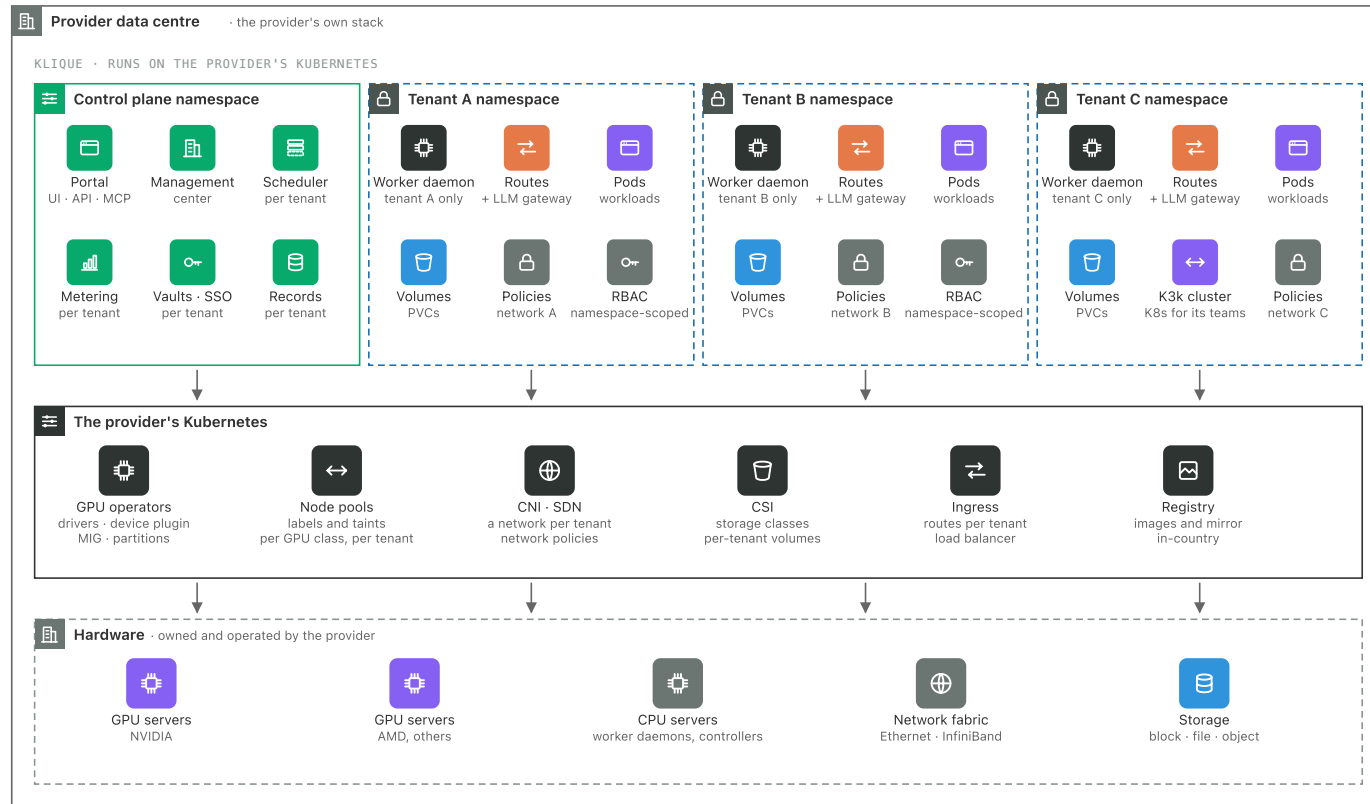
The provider brings the stack; Klique runs on top of it. At the bottom sit the provider's GPU and CPU servers, network fabric and storage. On them runs the provider's own Kubernetes, with the pieces that make multi-tenancy work: GPU operators, node pools labelled and tainted per GPU class and tenant, a CNI or SDN that gives each tenant its own network, CSI storage classes for per-tenant volumes, ingress and a registry. Klique adds a control plane namespace and one namespace per tenant, and uses each of those layers rather than replacing them.

The provider works from the Management Center

The provider operator works in the Platform Management Center, a separate application with its own credentials.

Where tenant data lives

Per-tenant volumes come from the Management Center's volume templates, and each tenant has its own file-server space. Storage credentials sit in the tenant's vault, and the output destination is set per project to the tenant's own bucket or storage.

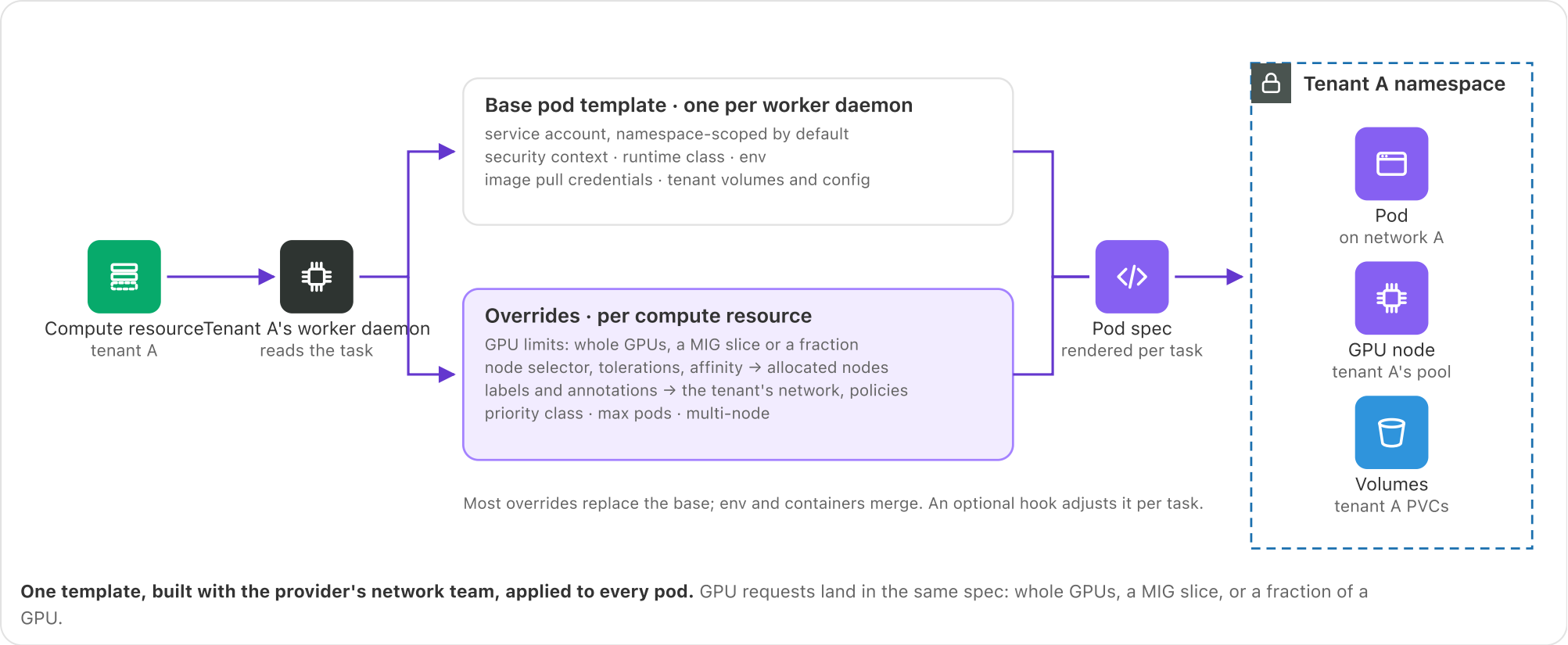


Hardware, the provider's Kubernetes, then Klisque. The worker daemon is namespace-scoped by default, so each tenant's worker daemon can only create pods in its own namespace.

Network isolation

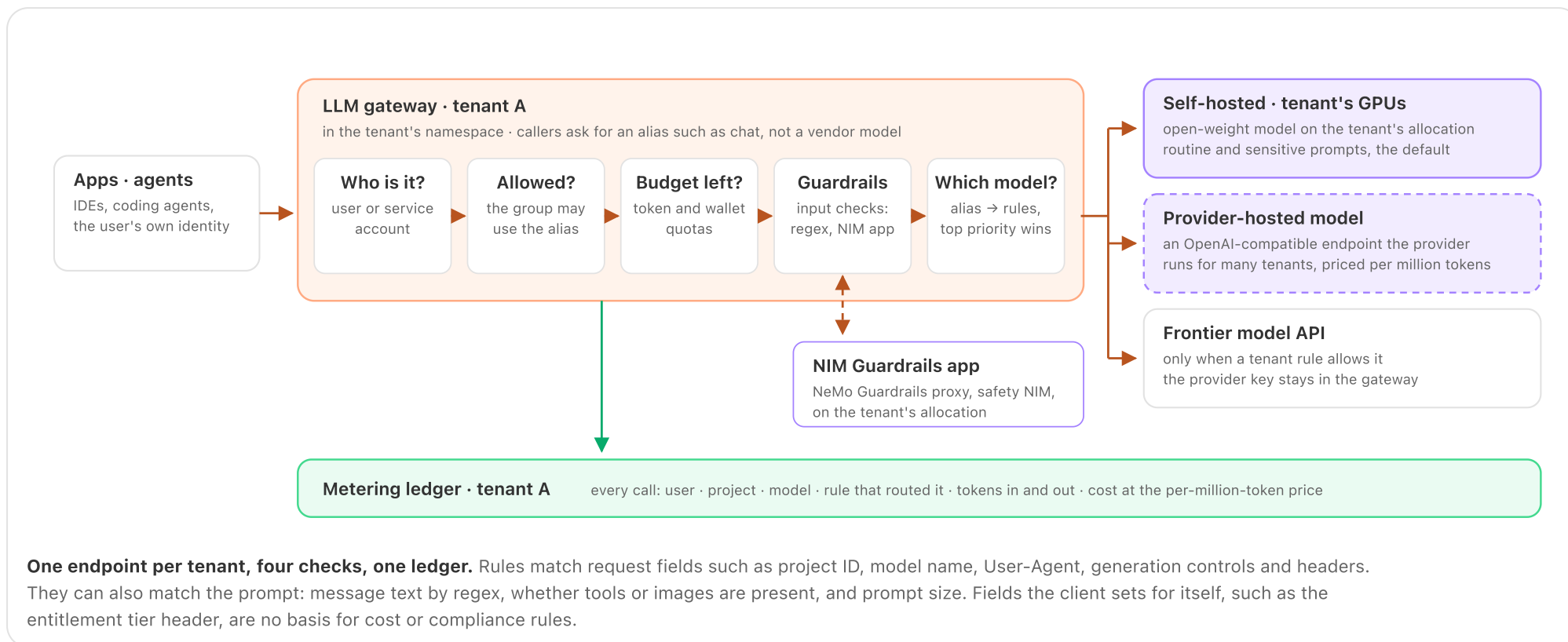
This reference architecture assumes an SDN, or a CNI that enforces network policies, run by the provider. The worker daemon generates every pod spec itself, from two parts: a **base pod template** for the tenant's worker daemon, and **overrides per compute resource**. Node selectors, tolerations and affinity choose the tenant's allocated nodes; labels and annotations carry what the provider's network and policies match on; the namespace sits on the tenant's network, with network policies around it. Every task,

session or endpoint the tenant launches, from any channel, gets the same treatment. Sessions and endpoints are reached only through the tenant's own internal routes, with access checked on every route.



Tokens as a service

Every tenant gets its own **LLM gateway** as part of onboarding, not as an add-on: it runs in the tenant's own namespace. The tenant admin sets its aliases, routing rules, guardrails and budgets, and the provider has no need to see its traffic. Callers use one OpenAI-style endpoint and an alias; the rules behind the alias decide which model answers. The gateway itself is covered in depth in the [LLM Gateway and Tokenomics Reference Architecture](#).

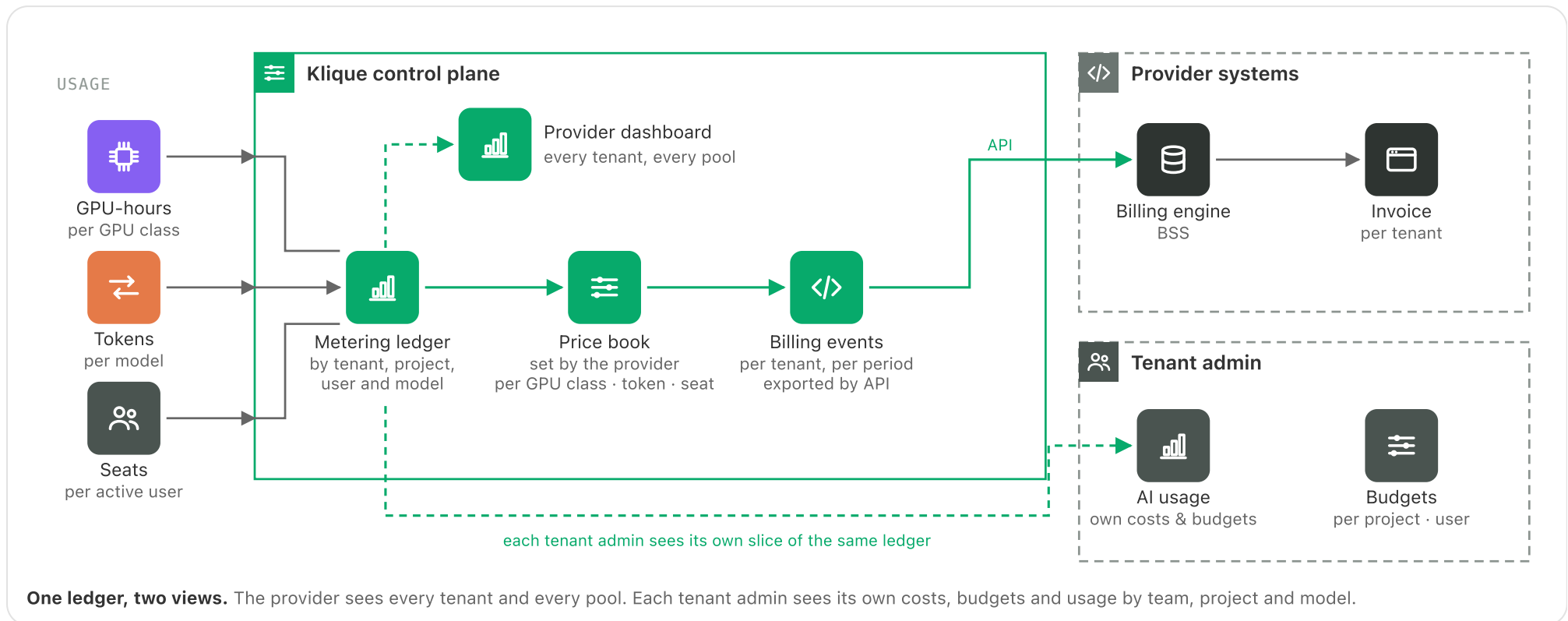


RULE A TENANT ADMIN MIGHT SET	GOES TO	WHY
Every request	Input checks: deny or allow regex patterns, and the NIM Guardrails app for model-based checks	A PII check, as a pattern or in the app, stops confidential data before it reaches the chosen model
Requests with a small max-tokens value	An open-weight model on the tenant's allocation	Lowest cost per token, stays in-country
Hard reasoning, when allowed	A frontier model API	Quality where it matters, on the provider's key
A project with a budget	A wallet quota on the project	Spend tied to an outcome with an owner

The provider can sell tokens two ways: tenants run models on their own allocation and pay for the GPUs, or the provider hosts popular models once, as an OpenAI-compatible endpoint, and prices them per million tokens in, out and cached. Both land on the same ledger.

Metering and billing

Every GPU-hour, token, seat and gigabyte is recorded per tenant, project and user. The provider sets a **price book**: a price per GPU class per hour, per million tokens and per seat. The prices are the provider's own numbers, built from its costs for power, cooling, space and hardware, plus its margin. Klisque meters and prices; the provider's billing engine invoices. An enterprise runs the same ledger as chargeback or showback, with the price book set as an internal rate per business unit.

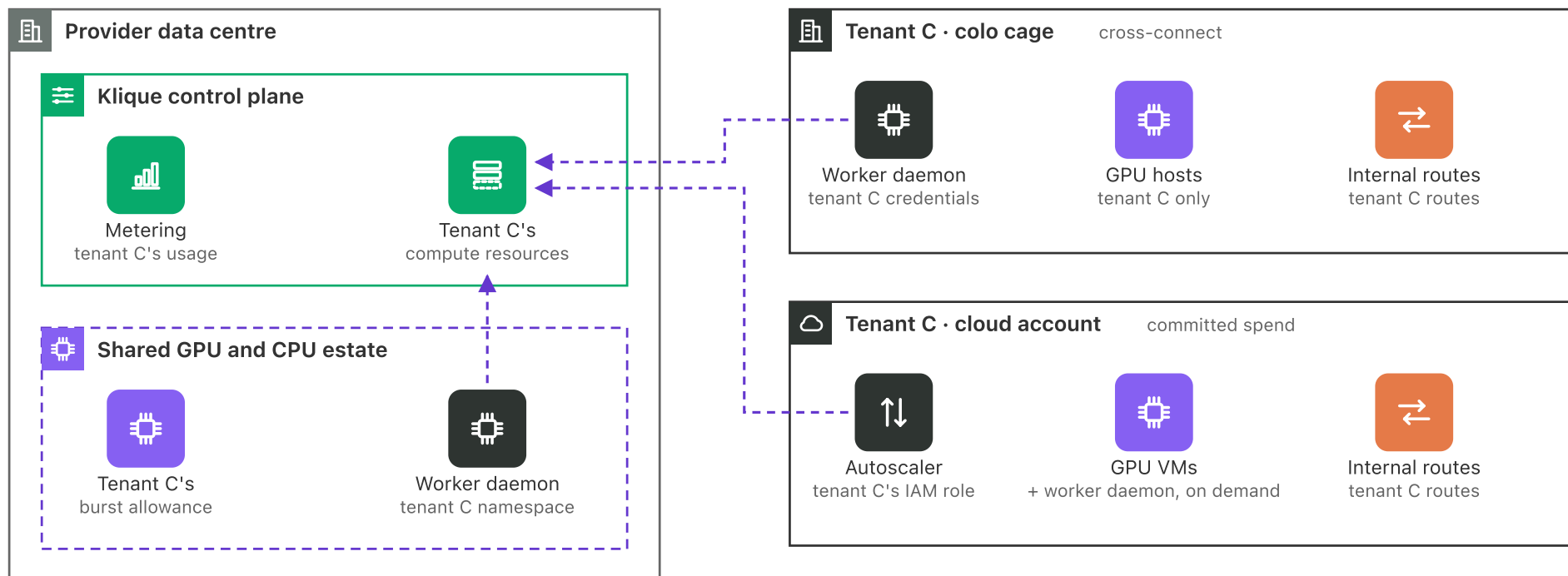


OFFER	UNIT	TYPICAL TENANT
Dedicated GPUs	Per GPU, per month	Steady training and serving, predictable spend
On-demand GPUs	Per GPU-hour, per GPU class	Bursty work, experiments, peaks
Tokens	Per million tokens in, out and cached	Teams that want models, not machines
Seats	Per user, per month	Platform access, and tenants that bring their own GPUs
Storage	From the storage meter	Datasets, models and artifacts kept on the platform

Export, not re-entry. Usage leaves by API, per tenant and date range, or as a stream of events, straight into the billing system the provider already runs.

Tenants' own GPUs

Some tenants already own GPUs: in a colo cage beside the provider, reached over a cross-connect, or in a cloud account with committed spend to burn down. A worker daemon installed there, registered with the tenant's credentials, brings those GPUs under the same control plane. They serve that tenant only. The tenant's IT team runs one estate instead of two, and the provider earns on the platform even where it doesn't own the hardware.



Worker daemons dial out over HTTPS. Tenant C's compute resources: own colo GPUs first → own cloud account next → the provider's burst space last.

Their GPUs, the provider's platform. Policy decides the order: the tenant's own GPUs first, then its cloud account, then the provider's burst space.

Owned GPUs suit a **seat** or **per-GPU-under-management** price, since the provider supplies the platform, not the hardware. Burst into the provider's estate is billed per GPU-hour like any other tenant's.

Service portfolio

The platform supports three service lines, each aimed at a different buyer. Tenants typically adopt one and expand into the others as their AI programmes mature, with no change of platform.

Ready-to-run AI applications

IT-led, a managed application catalogue: model endpoints, development workspaces, sandboxed coding agents, training and automation, clusters on demand, vector databases, app builders and any container the provider adds. Launched on demand, with the provider controlling which applications each tenant can use.

Governed access to models

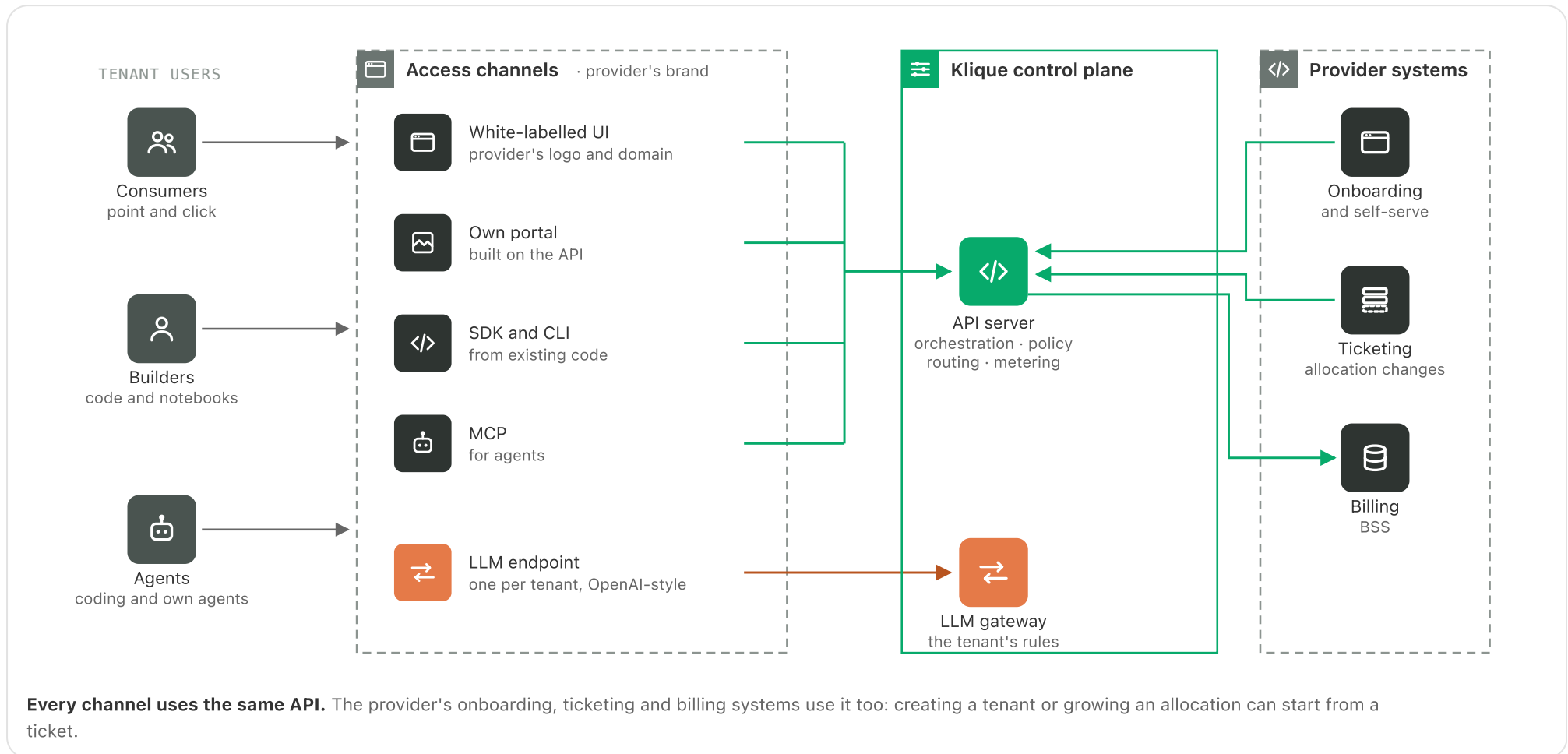
IT operations, models and tokens as a service: a private LLM gateway for each tenant, with routing between self-hosted and frontier models, guardrails, token budgets and usage reporting by team. Billed per token or per GPU.

End-to-end model development

Engineering-led, an AI development platform: data versioning, experiment tracking, pipelines and fine-tuning, driven from code or from an agent over MCP. All work runs on, and is billed to, the tenant's allocation.

Branding and access channels

Tenants see the provider's brand. The Klique UI takes the provider's logo, titles and links, set once for the whole platform. Providers that want their own experience, a self-serve catalogue or a serverless-style console, build it on the same API that the UI uses: the **API server** is the orchestration, policy, routing and metering engine, and it has no opinion about the front end.



The app catalogue

Behind these channels sits a catalogue of ready-to-launch applications. Each one is a form in the UI and a call in the API or MCP; each runs on the tenant's own compute resources, under the tenant's policies. The provider decides, per tenant, which apps are deployed, and can publish a tenant-specific version with its own launch form.

Endpoints in minutes

Model serving: vLLM, SGLang and Llama.cpp model deployment, embedding model deployment, NVIDIA NIM, NVIDIA Dynamo and NIM RAG, and an MCP server.

Workspaces on platform GPUs

Development: JupyterLab, VS Code, SSH sessions and a remote desktop, on a whole GPU or a slice of one.

Sandboxed coding agents

Agentic: Claude Code, Codex and OpenCode, each in its own container on the tenant's network, with model calls through the tenant's LLM gateway. Claude Desktop for the desktop.

From one GPU to many nodes

Training and automation: the multi-node trainer, hyperparameter optimisation, the task scheduler and trigger manager, and project dashboards.

A cluster, on demand

Clustering services: Slurm on Kubernetes for HPC-style teams, and K3k, Kubernetes in Kubernetes, for tenants that want their own Kubernetes API.

Vector databases

Data: Milvus and Qdrant sessions for retrieval and RAG, beside the tenant's models and data.

Tenant-facing apps

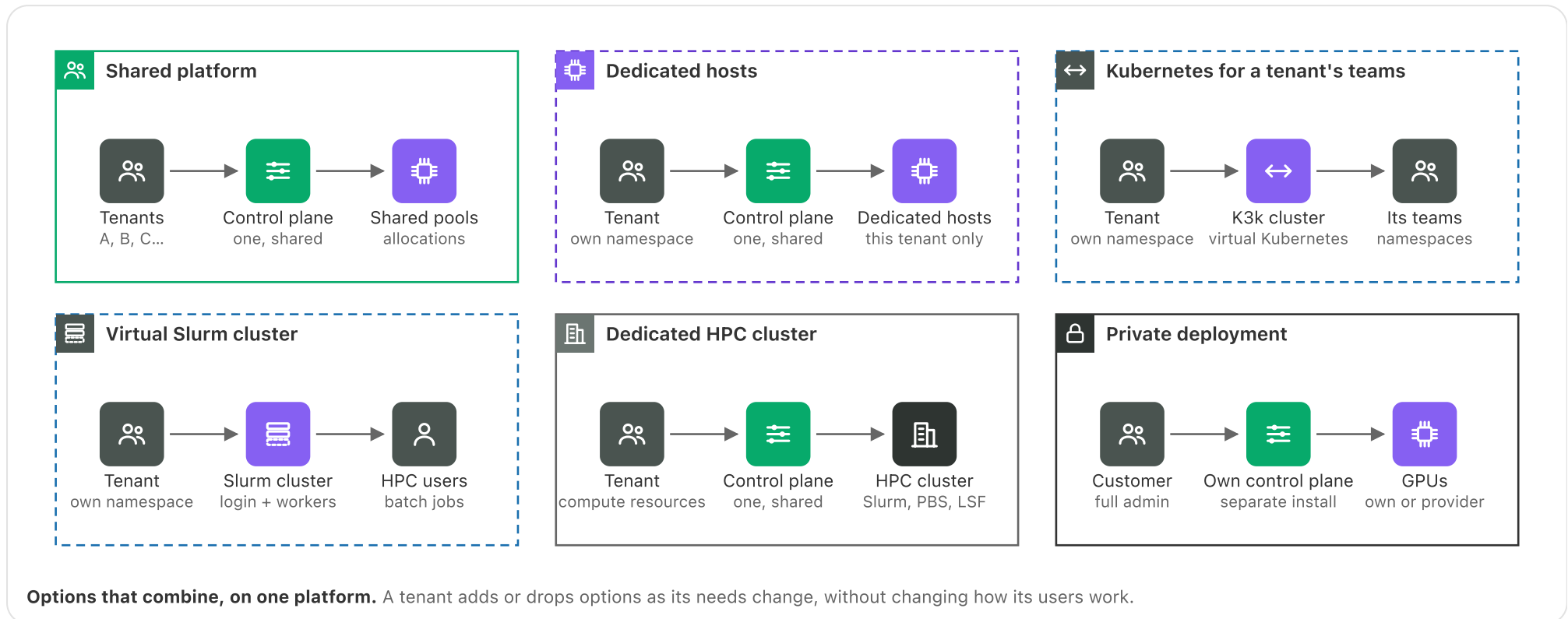
App builders: Gradio and Streamlit launchers turn a model and a script into an app the tenant's users can open.

Any container, as a catalogue app

Custom apps: the containerized application launcher, or a custom app uploaded in the Platform Management Center, such as AMD Inference Microservices (AIM), a database service, or the provider's own offer.

Service options

The shared platform is the foundation for every tenant. On top of it, the provider can offer further service options, and a tenant can combine them: shared GPUs for everyday work, dedicated hosts for a regulated project, a virtual Slurm cluster for one team and a dedicated HPC cluster for large training runs, all under one control plane, one set of policies and one bill. Only a private deployment stands apart, as a separate installation. Scheduling one control plane's work across Kubernetes, HPC, bare-metal and cloud estates is covered in the [Meta-Scheduler Reference Architecture](#).



OPTION	ISOLATION	OPERATED BY	BEST FOR
Shared platform	Own workspace, namespace, network and gateways	Provider	Every tenant: the foundation the other options add to
Dedicated hosts	As shared, plus hosts that carry only this tenant's pods	Provider	Regulated tenants that must not share a node
Kubernetes for a tenant's teams	As shared; the K3k clustering service gives the tenant its own Kubernetes API	Tenant, inside its allocation	Tenants running a platform for their own teams
Virtual Slurm cluster	As shared; a Slurm cluster inside the tenant's allocation	Tenant, inside its allocation	HPC-style teams that work with batch jobs
Dedicated HPC cluster	A whole Slurm, PBS or LSF cluster; batch jobs run as the tenant's users	Provider or tenant	Sustained HPC and multi-node training
Private deployment	A separate control plane and estate	Provider as managed service, or the customer	Customers that need full platform administration

Design considerations

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

Build the tenant template once

SDN or CNI mapping, node selectors, storage classes and per-tenant volumes. Every later tenant reuses it, so onboarding is configuration, not a project.

Decide the allocation types

Dedicated, reserved, burst and fractional each carry a price and a commitment to the tenant. Offer few, and keep the sum of reservations within each pool's real capacity.

Keep declared capacity true

The scheduler works from the capacity each pool declares, not by probing hardware. Out-of-date numbers mean waiting work or oversold pools.

Plan the CPU pool

Worker daemons, app controllers, agentic sessions and data preparation run on CPUs. A GPU-only estate leaves them nowhere to go.

Price from real costs

Set the price book from the provider's own cost per GPU-hour (power, cooling, space, hardware) plus margin, per GPU class. Klique applies it; it doesn't set it.

Keep provider keys in the gateway

Frontier API keys live in each tenant's LLM gateway and never reach tenant users. Rotate them without touching a single client.

Brand once

White-labelling applies to the whole platform. Tenant names appear inside it; the portal, domain and logo are the provider's.

Automate onboarding

Tie tenant creation, IdP set-up and allocation changes to the provider's self-serve portal or ticketing through the API, so growth doesn't wait on a person.

Plan offboarding

Removing a tenant removes its workspace and records. Buckets outside the platform are the tenant's and are cleaned up separately.

Accelerators beyond NVIDIA and AMD

Other accelerators, NPUs included, are supported case by case where the vendor's drivers are available. Fractional support comes later than whole-device support.

Glossary

Tenant	One customer's workspace: its users, projects, compute resources, vaults, gateways and ledger, kept apart from every other tenant's.
Provider operator	The provider's platform team. Runs the estate, onboards tenants and sets allocations, catalogues and prices.
Platform Management Center	The provider's back office: tenants, pools, allocations, app catalogues per tenant, usage and cost across the platform.
Tenant admin	The tenant's own IT team: the admins group inside the tenant. Manages users and groups from the tenant's own SSO (OIDC, SAML or LDAP), team policies, LLM gateway rules and budgets, inside the allocation.
Allocation	A tenant's share of a pool: dedicated, reserved, burst or fractional.
Compute resource	What a user sends work to, from any channel. It names an allocation, never a location. Under the hood it is a queue.
Namespace	The tenant's slice of the Kubernetes estate, on its own network, holding its worker daemon, internal routes and workloads.
Klique worker daemon	The worker that pulls a tenant's work and starts it as a pod or container. Registered with one tenant's credentials.
Internal routes	The tenant's routes to its sessions and endpoints, with access checked on every route.
LLM gateway	One endpoint per tenant for every model call: identity, access, budgets, guardrails, routing rules and metering.
Alias	A model name for a kind of work, such as chat, that the tenant admin can point at any model with no client change.

Price book	The provider's prices per GPU class per hour, per million tokens and per seat.
Billing event	Priced usage per tenant and period, exported by API or as events to the provider's billing engine.
K3k	A clustering service a tenant launches inside its own allocation: a virtual Kubernetes cluster, for handing Kubernetes namespaces to the tenant's own consumers.

Klique AI · Multi-tenant reference architecture · one control plane, a shared GPU and CPU estate, many tenants. Tenant names, pools, allocations and prices are illustrative. Every provider's estate looks different, which is the point.