hedgehog

Secure Multitenancy in AI Networks

Executive Summary

Cloud providers — AWS in particular — have defined what users expect from multitenancy: VPC-level isolation, self-service provisioning, and connectivity that just works. As AI economics push GPU infrastructure on-premises and into neoclouds, enterprises now need to deliver those same guarantees on hardware they own and operate.

This is harder than it sounds. The standard isolation mechanisms do not work for AI clusters. Virtual machines require the IOMMU, which must be disabled for GPU performance. Containers cannot accommodate tenants that need different kernels, GPU drivers, or Kubernetes configurations. That leaves the physical network — BGP/EVPN, VXLAN, VRFs, route maps, and ACLs configured across dozens of switches — as the only viable isolation boundary. But configuring this correctly is error-prone, difficult to audit, and slow, exactly the opposite of the self-service experience users expect.

Using LLMs to generate switch configuration directly ("AI Ops") does not solve the problem. Adding a single tenant can change hundreds of lines of config across 20–30 devices, producing diffs too complex for meaningful human review. LLMs hallucinate, and in network configuration a single wrong route-map entry or missing ACL leaks tenant data.

Hedgehog takes a different approach. Operators declare intent — VPCs, peerings, gateways — as Kubernetes CRDs. A deterministic Fabric Controller translates that intent into tested, validated switch configuration. The output is the same every time: no hallucination, no statistical variation. The Kubernetes API integrates with standard tooling (e.g. kubectl, RBAC, Helm, ArgoCD, Terraform) and fits naturally behind customer consoles or into GitOps workflows where AI can generate small, readable manifests that humans can actually review before a deterministic system enforces them.

The result is secure multitenancy with reliable operations: tenant isolation enforced by tested code, continuous reconciliation on every switch, and a full audit trail in git.

Cloud VPCs Set the Multitenancy Standard

Multitenancy has a long history in computing, going all the way back to the early IBM 360 mainframes. But today, cloud sets the standard for what multitenancy is and what users expect from it, and as the most popular cloud platform, Amazon Web Services (AWS) is the gold standard. When users think about multitenancy, they think about the guarantees offered by AWS via its Virtual Private Cloud (VPC) service [1].

What a VPC Provides

The core concept in multitenancy is tenant isolation, and AWS's VPCs provide it. Within each VPC, users allocate many services, but the main ones relevant to this whitepaper are the host, either as a virtual machine, or a bare metal server. The network provides isolation between the tenants, whether at the physical or virtual network layers. Hosts within a VPC can talk to each other via the network. Hosts in other VPCs are generally unreachable unless some type of gateway service is used to interconnect them. Even the internet is unreachable without a Gateway.

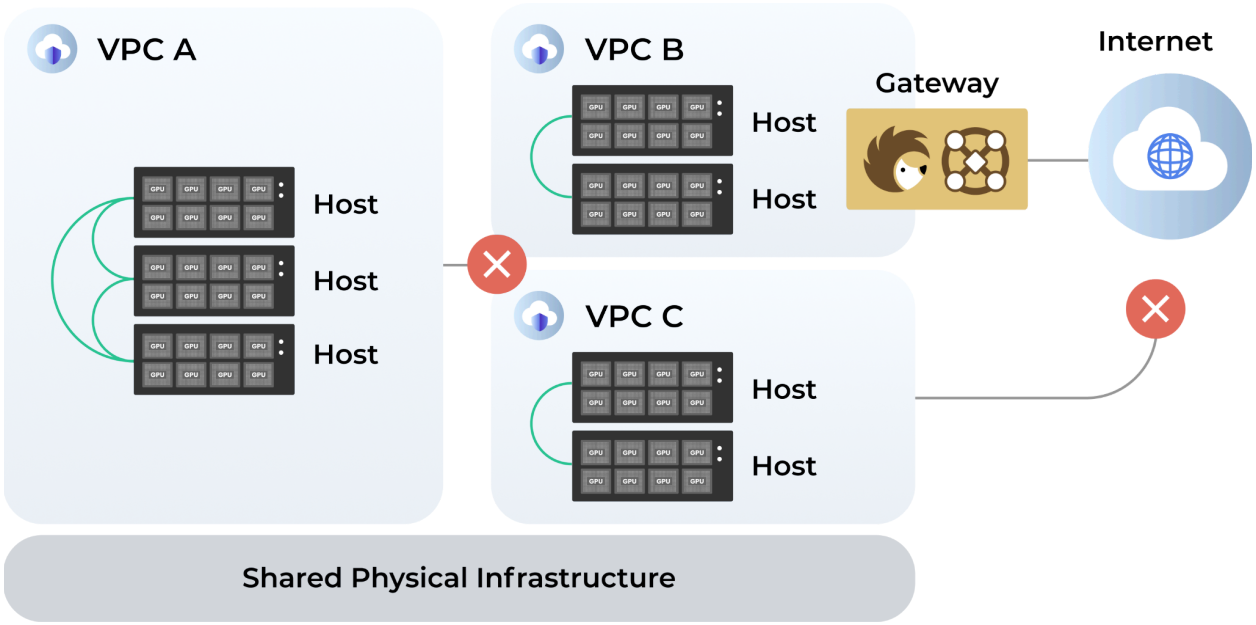

Figure 1: VPC isolation model. Each VPC is an isolated network domain. Hosts within a VPC communicate freely; hosts in different VPCs are unreachable unless explicitly connected through a peering or gateway service. Even internet access requires an explicit Gateway.

Within a VPC, a user is free to configure whatever they want as long as it uses unicast IPv4 or IPv6. They can assign whatever IP addresses they want, use whichever protocols they want above the IP layer, and can expect things to work. There are minor exceptions — a few addresses in each subnet are reserved, the broadcast address doesn't actually do broadcast — but these are small concessions that users discover as they use the service.

The Ideal On-Premises VPC

Stepping away from AWS's specific limitations, let's consider an ideal VPC. We'll stick with the idea that the unit of compute is a host or a virtual machine, though the concept can be extended to more general function-as-a-service or container-as-a-service environments.

For any on-premises or colocated infrastructure, it would be ideal to have a VPC where everything just works without any special restrictions. IP Multicast, no problem. Ethernet broadcast, also no problem. In practice, however, real network hardware imposes constraints that break the abstraction of a single, fully transparent L2 network per subnet. For example, some high-performance switch chipsets do not support full L2 overlay bridging at scale, or impose limits on broadcast and multicast handling in order to preserve performance. These constraints are driven by the underlying hardware design — itself shaped by cost, performance, and vendor tradeoffs. Instead of manually tracking these platform idiosyncrasies, it is far easier to let an automated system handle them correctly.

Connectivity and Peering

Even Internet connectivity must be configured via some type of service. AWS calls this either an Internet Gateway [2] or a NAT Gateway [3] depending on whether the hosts themselves have public IP addresses or private addresses that need to use IP NAT in order to reach the public internet. In either case, the connectivity to the internet must be explicitly configured.

Since VPCs exist on the same Fabric, it should be possible to peer VPCs together, either with or without NAT, so that they can talk to each other. Care should be taken to make sure that these VPC peerings cannot be used transitively to allow unintended connectivity, and expose VPCs to unwanted, potentially malicious, traffic.

All these services have extra requirements in an AI cluster, mainly due to performance considerations, which makes the problem more complicated. Moreover, the economics of AI clusters mean that these problems are going to shift from something a few hyperscaler cloud providers have to worry about to a problem

that every enterprise that deploys self-managed on-prem and colocated clusters needs to deal with.

The AI Cloud Moves On-Prem

The economics of AI are reversing a decade of cloud migration.

The Cloud Bargain

Over the last decade and a half, enterprises have shifted the operational burden and complexity of multitenant cloud infrastructure (especially for new applications) to cloud providers, most often hyperscalers like AWS, Microsoft, or Google. They pay a 3x, 4x, or larger premium for compute and storage to offload the complexity of multitenancy [4]. But this shift to cloud was based on certain economic assumptions - namely that servers were generally underutilized. The core operational challenge for cloud providers was maximizing VM density on a limited host pool to minimize physical and power footprints. Performance was secondary to efficient, large-scale operations. Enterprises paid a premium for compute and storage, but gained a dramatic improvement in operational efficiency, reducing the manpower and software systems needed to run their own infrastructure.

AI Changes the Calculus

AI workloads have fundamentally changed this calculus. Unlike typical cloud workloads, AI clusters are simply too expensive to sit idle. The underlying GPUs are costly assets that depreciate rapidly, a critical concern for both nascent "neoclouds" [5] and established enterprises. AI workloads are also engineered to use 100% of GPU capacity whenever possible. While the operational challenges described above still exist, the scale of the premium is now unacceptable. Paying a 2x or 4x markup is no longer a marginal cost; it is a significant portion of the total expenditure, often representing a far greater proportion of cost than typical cloud operations. As AI adoption expands, the total cost of the AI cluster becomes significant, making such a high premium untenable.

This economic reality is driving a return to **on-premises** or **neocloud** infrastructure ownership. Data privacy and sovereignty concerns reinforce this trend — many enterprises are unwilling to send proprietary training data and models to a third-party cloud, and some are bound by regulation to keep them on infrastructure they control. The enterprise or neocloud now owns the infrastructure and must actively partition the cluster for different users—internal teams for the enterprise, or external customers for the neocloud. Critically, to maximize the return on this

massive infrastructure spend (a CapEx cost that rivals even stock buyback programs for some big tech firms [6]), the expensive resources cannot be allowed to sit idle. As soon as users are done, these tenants must be reprovisioned to new users to keep the cluster continuously occupied. For a neocloud, this utilization directly translates to profit margin; for an enterprise, it is essential for maximizing ROI on a massive, specialized investment.

This requirement for continuous, rapid reprovisioning and high utilization means that the central challenge shifts away from simple operational management and firmly toward **scale and performance** under a secure multitenant model.

Why Traditional Isolation Fails for AI

The standard approaches to tenant isolation all rely on running the isolation logic on the host itself. This works well for traditional workloads, but each approach breaks down when applied to modern GPU clusters.

The Host-Based Overlay Model

In the conventional approach, the physical network provides basic L3 IP connectivity between hosts, and tenant isolation is implemented in software above it. The hosts themselves run a BGP/EVPN Control Plane with VXLAN Data Plane encapsulation [7], where each host acts as a VXLAN Tunnel Endpoint (VTEP). Each tenant gets a private overlay network — a virtual L2 or L3 segment — that is invisible to the underlying physical switches. The physical network just moves VXLAN-encapsulated packets between hosts without needing to know anything about tenants.

On a hypervisor, this overlay is managed by the virtualization platform. VMware NSX [8], for example, runs the VTEP and routing functions inside the hypervisor, giving each VM its own isolated virtual network. In a Kubernetes environment, CNI plugins [9] like Calico [10] or Cilium [11] serve the same role: they configure the overlay networking on each node, assign pods to the correct virtual network, and enforce network policy. In both cases, the orchestration software — the hypervisor or the CNI — handles the complexity of setting up and tearing down these overlays as tenants come and go.

VPC peering, internet connectivity, and other services are then provided either by software running on the hosts themselves (virtual routers, NAT Gateways) or by purpose-built appliances that sit at the edge of the overlay network and bridge between tenant virtual networks and external systems.

With DPUs, this same model is moved off the host CPU and into dedicated hardware on the network card [12]. The DPU runs the VTEP, the routing, and the encapsulation, freeing the host CPU entirely for tenant workloads. This also enables bare metal multitenancy — since the DPU hardware is not configurable by the host operating system, even a bare metal tenant with full OS access cannot break out of its assigned network. This approach, however, is not common outside of hyperscalers — for a traditional server that is already under-utilized, the cost of a DPU is hard to justify for network bandwidth the applications don't need.

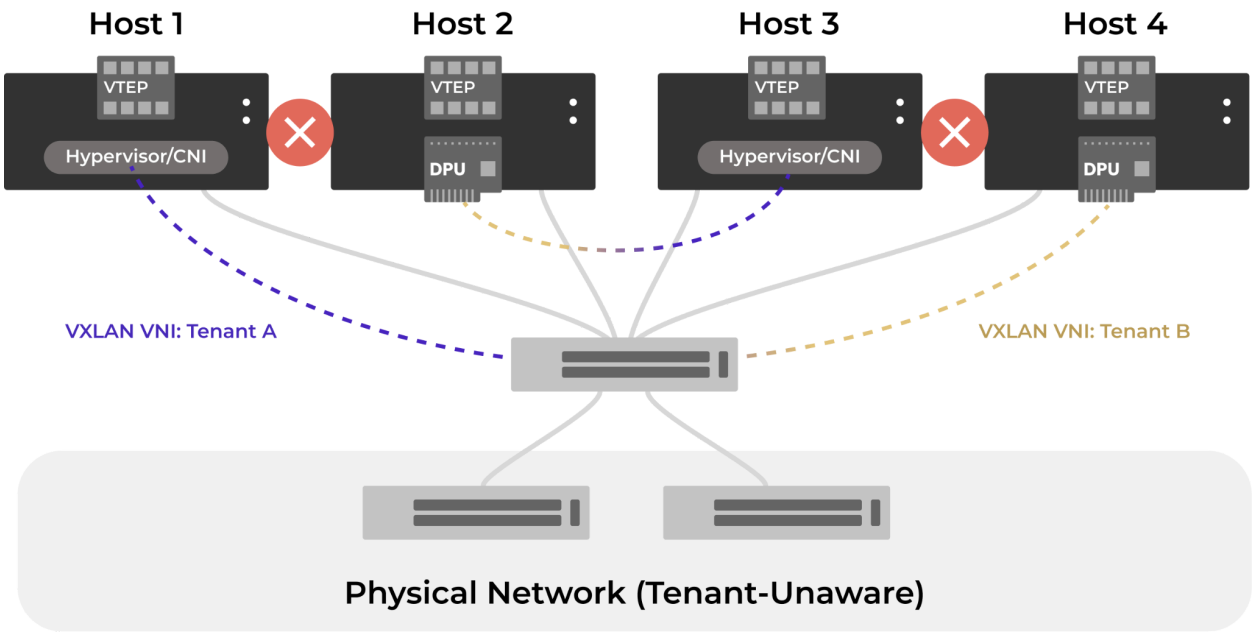

Figure 2: Host-based overlay model. Each host runs a VXLAN Tunnel Endpoint (VTEP) — either in software on the host CPU, inside a hypervisor, as a CNI plugin, or offloaded to a DPU. The physical switches carry encapsulated VXLAN packets without any knowledge of tenants. Tenant isolation is enforced entirely at the host level.

Unfortunately, neither the host-software nor the DPU approach works well on modern GPU clusters with a high-performance, RoCE-optimized backend network. There are two main constraints that make these approaches impractical as of this writing.

Host Network Performance

In a modern AI cluster, GPUs are communicating with each other at extremely high rates. For GPUs within a single host, they use either the PCI bus or other proprietary

interconnect like NVLink [13]. But between hosts (or between racks if the proprietary interconnect has been extended to the rack), the speeds are staggering. As of this writing, an NVIDIA B300 node has 800 Gb/s of connectivity *per GPU (with a typical 8 GPUs per host)* connected to the RoCE-optimized backend network [14]. That same host has 400 Gb/s of front-end connectivity storage systems and other services, usually in a 2x200 Gb/s configuration.

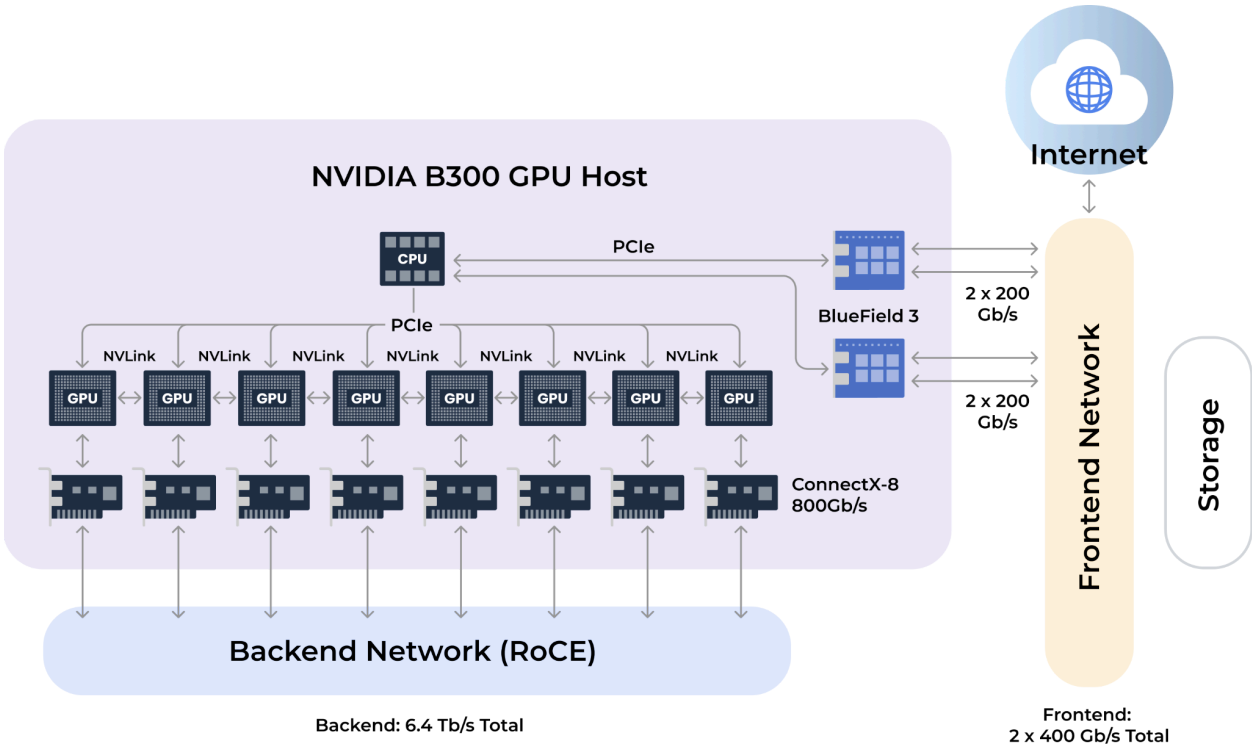

Figure 3: Network connectivity of a modern GPU host (NVIDIA B300). Eight GPUs each connect at 800 Gb/s to the RoCE-optimized backend network. A single DPU provides 400 Gb/s (2x200 Gb/s) of front-end connectivity for storage and management traffic. The backend network carries the bulk of the bandwidth and cannot accommodate a DPU or host-based overlay without unacceptable power and performance costs.

For the AI workloads to run efficiently, connectivity between GPUs across hosts, and connectivity between the GPU host and the storage system should operate at the full cut-through bandwidth of the network. Running a modern DPU for the front-end network is feasible, and is an approach that some hyperscalers use. Software for these DPUs is improving and with the proper management and automation, multitenancy on the front-end can be offloaded to a DPU solution. Over the next one to two years, this may become the de facto standard for the front-end network.

The problem is on the backend. At the speeds and number of ports, a modern DPU does not make good use of the power budget of the datacenter. A single DPU in a GPU host is one thing — but using 8-10 DPUs in the same host would consume an inordinate amount of power, and that power can instead be used to power bigger more powerful GPUs, GPUs with more memory, or to pack more GPU hosts in a datacenter with a fixed power budget.

As a result, the network cards that run the backend network are under the control of the host operating system, and this means that the only reliable approach for security are virtual machines, or, if somewhat lesser isolation guarantees are sufficient, separate containers. But this too is not a solution for today's AI networks.

IOMMU, Performance, and VMs

In order to properly isolate virtual machines while still preserving line rate performance for the network, modern environments expose the network PCI devices directly to the virtual machine, allowing the virtual machine guest OS device drivers to directly control the hardware. A similar approach can be used to map the GPU into the virtual machine as well.

However, it is still possible for the guest OS to configure data transfers between CPU memory and the network or GPU devices such that the data would be written to or read from the address space of other guest machines, a gross violation of tenant isolation. Fortunately, modern CPUs have a solution - the IOMMU.

On the CPU the MMU, or memory management unit, is used to partition physical memory into different regions that can be used to isolate processes within the OS, and to provide virtual memory to guest operating systems in such a way that they cannot overwrite or read each other's memory.

In a similar fashion, the IOMMU allows the hypervisor to provide private addresses to both the guest OS *and* PCI devices. This means that a PCI device (or part of a PCI device) can be exposed to the guest OS, and no configuration of the hardware can cause a read or write of memory for another guest because the IOMMU will not allow such address translations to happen, just like the MMU does for non I/O operations.

Unfortunately, the IOMMU is typically disabled (or set to passthrough mode) on GPU clusters to enable adequate performance [15]. With the IOMMU fully enabled, PCIe peer-to-peer transfers — such as GPUDirect RDMA between a NIC and a GPU — are forced through the CPU root complex instead of taking a direct device-to-device path. This root complex routing adds latency and limits throughput for the

high-bandwidth RDMA transfers that AI workloads depend on. The related PCIe Access Control Services (ACS) feature has the same effect and is also disabled. While intra-node GPU-to-GPU communication uses NVLink and bypasses PCIe entirely, the NIC-to-GPU path is critical for inter-node traffic on the backend network, and the IOMMU overhead on that path is unacceptable at 800 Gb/s per GPU.

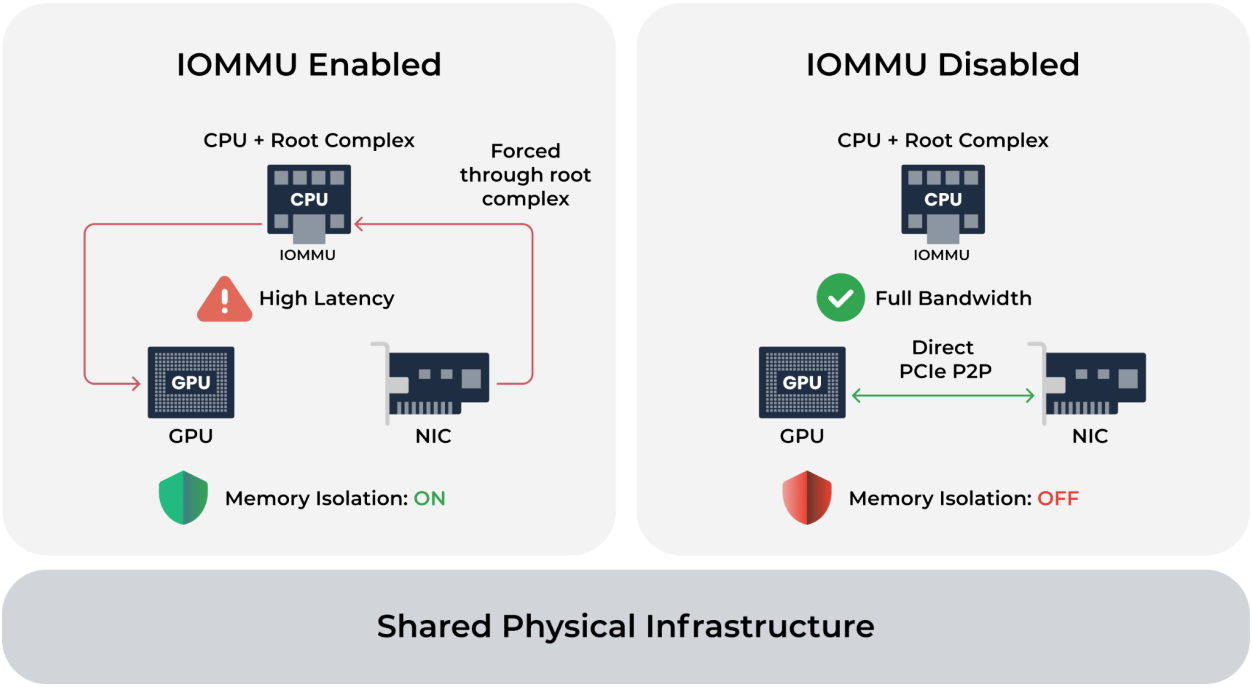

Figure 4: PCIe peer-to-peer data paths with and without IOMMU. Left: with IOMMU enabled, GPUDirect RDMA transfers between the NIC and GPU are forced through the CPU root complex, adding latency and limiting throughput. Right: with IOMMU disabled, NIC-to-GPU transfers take a direct PCIe P2P path at full bandwidth, but the ***CPU can no longer enforce memory isolation between tenants****.*

With the IOMMU disabled, direct assignment of NICs or GPUs to a guest OS is unsafe, as the devices could access memory belonging to other guests on the hypervisor, violating tenant isolation. These are not theoretical risks. Even with the IOMMU enabled the Thunderclap project demonstrated that flaws in operating system integration can allow malicious PCIe devices to perform DMA-based attacks and compromise system memory across multiple major operating systems [16]. With the IOMMU disabled or relaxed — as is required in high-performance GPU environments — these attack surfaces are further expanded. In this model, strong tenant isolation is best established at the physical network, which becomes the primary and most reliable enforcement boundary.

Containers Are Not Enough

As a result, the best multitenancy available on a properly configured GPU host is a container-based approach. Again, with the right software, it is possible to offload the network encapsulation (namely VXLAN) required for tenant isolation as long as each GPU and NIC is allocated to only a single container. But this configuration has to be managed by software on the host OS, and this software is not widely available, nor easy to configure correctly.

Moreover, one has to live with the reduced guarantees offered by containers. Unfortunately, container escapes are not uncommon — a critical vulnerability in runc (CVE-2019-5736) allowed a malicious container to overwrite the host runc binary and gain root-level code execution on the host, affecting Docker, containerd, and CRI-O [17]. In 2024, the "Leaky Vessels" vulnerability (CVE-2024-21626) again allowed container escape via a file descriptor leak in runc [18], and in 2025, yet another runc escape (CVE-2025-31133) exploited a TOCTOU race condition on masked proc paths [19]. These are not just CVEs that get patched and forgotten: in 2021, researchers demonstrated "Azurescape," the first known cross-account container escape in a public cloud, where an attacker escaped their Azure Container Instance and gained access to other tenants' data and credentials on the same Kubernetes node [20].

Beyond these security concerns, state of the art AI workloads may want to load their own drivers for GPUs, or have a kernel compiled and optimized for the workload. This is not possible in a container-only tenant isolation strategy. Worse still, different AI workloads may have conflicting requirements in this regard.

The final complication with containerized approaches is that different teams and workloads may need different configurations of the container orchestration runtime, usually Kubernetes. Kubernetes, for example, does not allow different sets of CRDs to coexist on the system as they are global, non-namespaced resources. As a result, it is desirable in a multitenant cluster to run multiple instances of Kubernetes, and this will require virtualization, which is not an option, or the ability to partition the cluster by host with multitenancy provided by something that lives outside the GPU host - namely, the network.

Tenant Isolation at the Network Level

Given that virtual machines are not a tenant isolation option because of the IOMMU, and because container-based isolation cannot accommodate tenants that need different kernels, GPU drivers, or Kubernetes configurations, the network is the

remaining option — and the switch hardware can do it, but the configuration complexity is daunting.

The Network Can Do It

The good news is that modern switch hardware has the capability to provide multitenancy at the GPU-host level. The network can be configured to isolate a collection of GPU hosts into their own virtual network using BGP/EVPN and VXLAN. Each VPC can be given a VRF on the switch, a unique VXLAN VNI (or set of VNIs in practice). As a result, each tenant gets its own set of routes, can use overlapping IPs, and otherwise have no interference with GPU hosts in other VPCs.

Chipset Constraints

There is, however, a complication at the switch chipset level. AI backend networks demand both high port density and high per-port bandwidth, and some of the most popular chipsets that deliver this — Broadcom's Tomahawk 5 [21] and the upcoming Tomahawk 6 — do not support L2VNI mode for VXLAN. This means that the conventional approach of bridging within a VXLAN segment (which provides a full L2 Ethernet abstraction per subnet) is not available on these switches. Instead, a pure L3VNI routing approach is required, where all traffic is routed at the switch rather than bridged. This is actually preferable for AI workloads — even on switches that do support L2VNI at these densities and bandwidths, L3VNI is the better choice. L2VNI segments carry broadcast traffic (ARP, DHCP discovery, and other L2 protocols), and on a RoCE-optimized backend network, broadcast packets competing with latency-sensitive GPU traffic is exactly the kind of disruption you want to avoid. But it is yet another chipset-specific constraint that the network configuration must account for correctly.

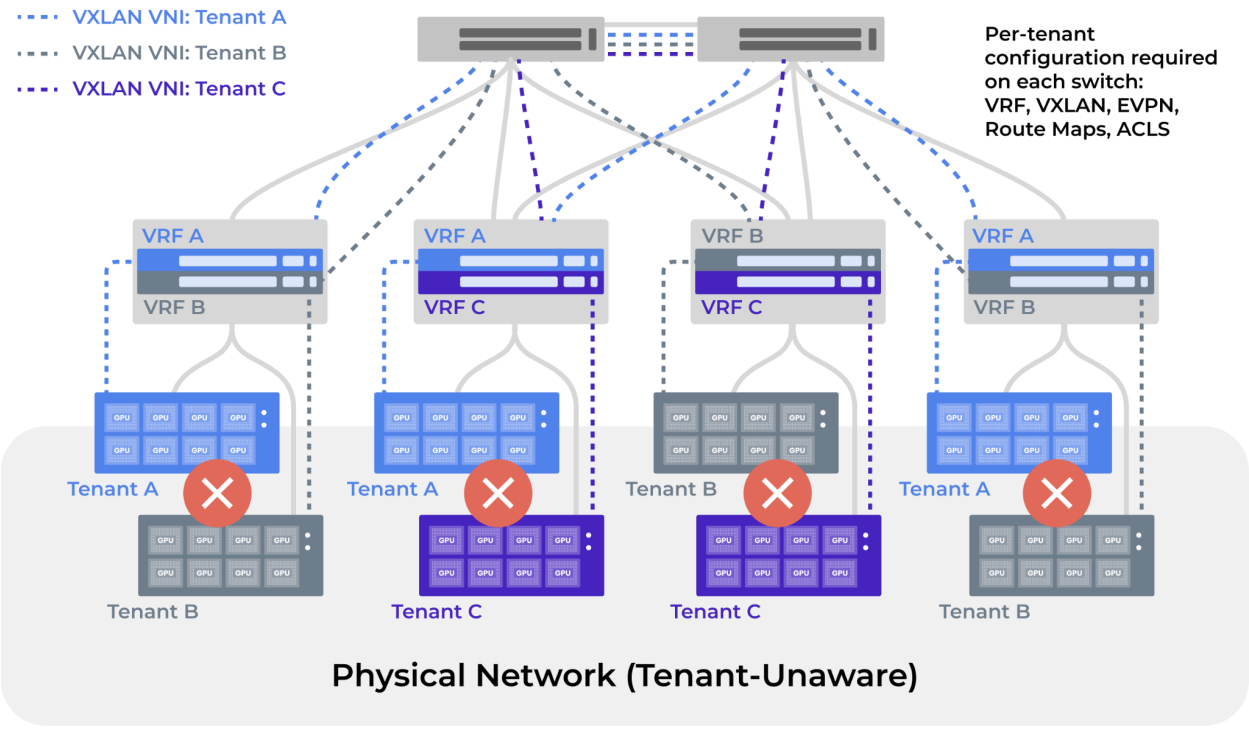

Figure 5: Switch-based multitenancy with BGP/EVPN and VXLAN. Each tenant is assigned a VRF and a unique set of VXLAN VNIs on the leaf switches. Hosts in the same VPC are connected through the Fabric; hosts in different VPCs are isolated at the switch level. This approach requires no software or hardware on the host, but the switch configuration is complex.

The Configuration Burden

The challenge is that this is very difficult to configure correctly. There are books on the subject [22]. And, for each new tenant, switch configuration has to be modified to accommodate that tenant. Any errors in this configuration means that a GPU host is either disconnected or connected to the wrong environment, violating tenant isolation. Each switch change risks disrupting other tenants.

Since an isolated VPC is not useful, VPCs need to be peered to each other (e.g., the storage VPC needs to peer with most tenants), and this peering has to be carefully configured. Route leaking between VRFs can achieve this peering, but care must be taken to have the correct ACLs and route maps so that peerings between VPC A and VPC B does not allow transit to other VPCs connected to VPC A.

To make matters even more complex, VPCs need to be able to connect to other services on the internet and inside the enterprise network. Each of these connections have to be carefully configured on border leaf switches, and the

configuration of firewalls and routers have to be coordinated so that VPCs get the appropriate firewall, NAT, and other services when connected outside the cluster.

Any error or oversight in this configuration means that data will leak between tenants, or tenants won't work. And, since each of these devices is typically shared between VPCs, misconfiguration to add or remove a tenant can disrupt other tenants if not done properly.

Moreover, cloud users have been conditioned to expect that provisioning is self-service and completes in minutes, not hours or days of manual configuration using ad hoc playbooks assembled by hand. This combination of high change velocity and complex, shared infrastructure significantly increases the likelihood of configuration errors.

The consequences of network misconfiguration are well documented outside the datacenter. In 2019, a small company's BGP optimizer leaked a massive number of routes through Verizon, which accepted and propagated them globally rather than filtering them, rerouting traffic for Cloudflare, Amazon, and others through unintended paths for hours [23]. In 2008, Pakistan Telecom's attempt to block YouTube domestically propagated globally due to missing route filtering, taking YouTube offline worldwide for approximately two hours [24]. Inside the datacenter, the Capital One breach (2019) was enabled by a misconfigured firewall rule that allowed an attacker to reach the EC2 metadata service and exfiltrate over 100 million customer records [25]. And in Azure's Cosmos DB service, a misconfigured default-enabled feature allowed one tenant to obtain the primary read-write keys of other tenants' databases [26]. These incidents underscore a consistent pattern: when network isolation depends on correct manual configuration across many devices and rules, humans inevitably make mistakes, and the blast radius is catastrophic. This is precisely the problem that an automated, intent-driven Fabric Controller eliminates — operators declare the desired connectivity, and tested code produces correct configuration every time.

The Hedgehog Approach

The previous section described the problem in stark terms: network-level tenant isolation is the only viable option for AI clusters, but configuring BGP/EVPN, VXLAN VNIs, VRFs, route maps, and ACLs across dozens of switches by hand is error-prone, slow, and dangerous. Hedgehog solves this by raising the level of abstraction. Instead of configuring switches, operators describe what they want — VPCs, peerings, gateways — and a deterministic Controller translates that intent into correct, tested, and validated switch configuration.

Architecture: Kubernetes as a Network Controller

At its core, Hedgehog runs a Kubernetes cluster on a dedicated Control Node that serves as the management plane for the entire Fabric [27]. This Control Node is a purpose-built appliance — it does not run end-user workloads. Importantly, this is not a proprietary system with a Kubernetes-like API; it is actual Kubernetes. This matters because Kubernetes comes with a mature, battle-tested set of capabilities that Hedgehog inherits for free: role-based access control (RBAC) for fine-grained permissions, admission webhooks for policy enforcement, a well-defined API server with watch semantics for real-time updates, and a rich ecosystem of tooling that operators already know.

Kubernetes was designed to be extensible through Custom Resource Definitions (CRDs) [28] — a mechanism that allows new object types to be added to the Kubernetes API alongside built-in types like Pods and Services. Hedgehog uses CRDs to teach Kubernetes about network concepts: VPCs, VPC peerings, Gateways, switches, and physical connections all become first-class Kubernetes objects that can be created, queried, updated, and deleted with standard tools like kubectl, Helm, Terraform, or any Kubernetes client library.

The physical infrastructure is declared in what Hedgehog calls the **Wiring Diagram** [29]: a set of CRDs that describe switches and their roles (spine, leaf, border leaf), servers and their connections, external systems, and the physical links between them. On top of this, operators define VPCs, peerings, and Gateways as a separate layer of CRDs that describe the desired tenant configuration. The separation is deliberate — the wiring diagram is typically set up once by the network team when the Fabric is deployed, while the VPC layer changes frequently as tenants are provisioned, reprovisioned, and decommissioned.

Each switch in the Fabric runs a **Fabric Agent** — a lightweight process that watches its corresponding CRD on the Control Node's Kubernetes API. When the Fabric Controller updates the desired state for a switch, the Agent applies the configuration locally and reports status back. This is the same reconciliation model that Kubernetes uses for application workloads: the Control Node is analogous to the API server and controller manager, and the Fabric Agent on each switch is analogous to the kubelet on each node. The result is that every switch is continuously reconciled against its declared state, and drift is corrected automatically.

The Fabric also includes **Gateway nodes** — commodity servers connected to the Fabric and managed by the Control Node that perform advanced network functions like stateful NAT and firewall that switch hardware cannot. The Controller and Gateway use BGP route advertisements to seamlessly steer traffic that needs

advanced processing to the Gateway, requiring no special-case configuration on the switches themselves. We describe these in more detail below.

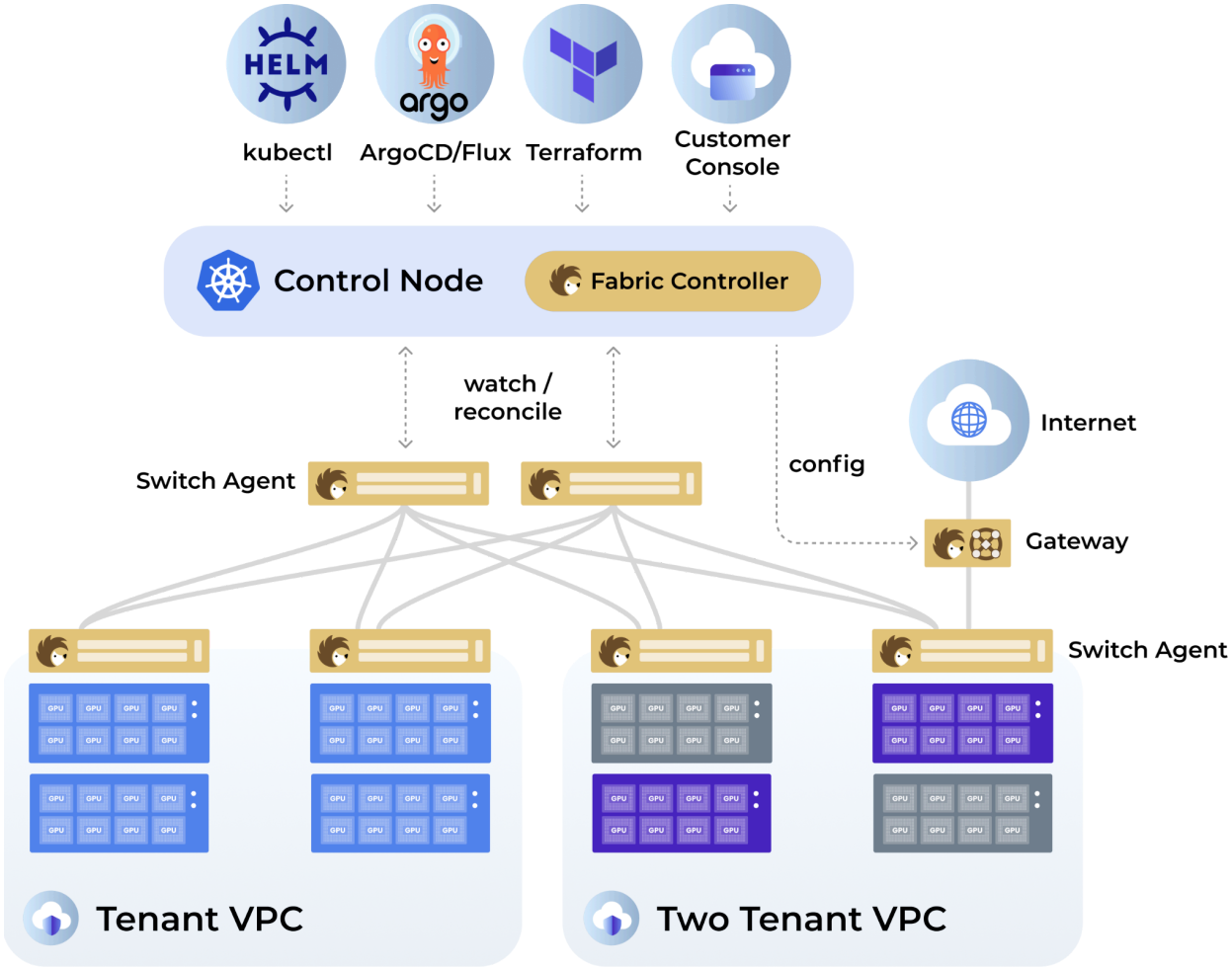

Figure 6: Hedgehog Fabric architecture. A dedicated Control Node runs Kubernetes with the Fabric Controller. Operators interact with the Kubernetes API to declare VPCs, peerings, and Gateways as CRDs. The Fabric Controller translates intent into per-switch configuration. A Fabric Agent on each switch reconciles the declared state continuously. Gateway nodes provide stateful NAT, firewall, and external connectivity.

A VPC Abstraction for the Physical Network

Hedgehog exposes the familiar VPC model as Kubernetes Custom Resource Definitions (CRDs) under the vpc.githedgehog.com/v1beta1 API group [30]. The primary objects are:

- **VPC** — a tenant's isolated virtual network, with one or more subnets, DHCP configuration, and optional static routes.
- **VPCPeering** — a declared connection between two VPCs, with control over which subnets are exposed to each other.
- **External** and **ExternalAttachment** [31] — represent upstream networks (BGP-speaking routers, static routes to the internet or enterprise backbone) and their physical connections to border switches.
- **ExternalPeering** and **Gateway Peering** — connect VPCs to external networks, with support for masquerade NAT, stateful NAT, and 1:1 NAT as needed.

The key insight is that operators work at the VPC level. They never touch VRFs, VNIs, route targets, route maps, or ACLs directly. They declare "VPC A peers with VPC B" or "VPC C has NAT access to the internet," and the system handles the rest.

Because everything is a Kubernetes CRD, all standard Kubernetes tooling works out of the box: kubectl for ad hoc operations, RBAC for access control, admission webhooks for policy enforcement, Helm charts for packaging, ArgoCD or Flux for GitOps, and the Terraform Kubernetes provider for infrastructure-as-code workflows. For neoclouds building customer-facing panels, the Kubernetes API is the integration point — no proprietary SDK or CLI required.

Deterministic, Tested Translation

Behind the VPC abstraction sits the Hedgehog Fabric Controller, a Kubernetes operator that watches for CRD changes and translates VPC intent into SONiC or Cumulus switch configuration [32]. The translation is mechanical and well-defined:

- Each VPC becomes a VRF on every switch where that VPC has attached hosts.
- VXLAN VNIs, route targets, and VLANs are auto-assigned from configured namespaces — no manual numbering.
- Routes, route maps, and ACLs are installed automatically, including the anti-transit protections that prevent VPC peering from being abused to leak traffic between unrelated tenants.
- An Agent on each switch continuously reconciles the declared state against the actual running configuration, correcting drift.
- Pre-flight validation — including connectivity and route checks — runs before changes are deployed to production switches.
- Chipset-specific idiosyncrasies (for example, certain switch chipsets that require L3VNI-only operation instead of the default L2VNI mode) are handled in tested code paths, not left to operator tribal knowledge.

This is the critical point: **the translation is deterministic software, not probabilistic AI.** Given the same VPC intent, the Controller produces the same switch configuration every time. The code paths are tested, the edge cases are enumerated, and the output is proven correct. There is no hallucination, no statistical variation, no "usually works."

The Hedgehog Gateway

VPC peering through the switches gives tenants full cut-through bandwidth, but switches are optimized for forwarding — they lack the CPU and memory needed for stateful operations. Functions like connection tracking, stateful NAT, and firewall enforcement all require resources that high-bandwidth switch chipsets simply do not have. In a traditional environment, these functions are provided by expensive, purpose-built appliances that must be separately purchased, configured, and integrated into the network.

Hedgehog solves this with the Gateway [33] — a commodity x86 server connected to the Fabric that performs these advanced network functions in software. The Gateway participates in the Fabric's BGP/EVPN Control Plane as a router that understands the L3VNI encapsulation. When a Gateway peering is configured, the Fabric Controller arranges for the Gateway to advertise the appropriate routes into the relevant VRFs on the Fabric switches. Traffic that needs advanced processing — NAT, firewall, or connectivity to an external network — is attracted to the Gateway through standard BGP route advertisements, processed, and forwarded to its destination. No special traffic steering or hairpinning is required; to the switches, the Gateway is just another BGP peer with routes to advertise.

The Gateway supports several modes of operation. Simple VPC-to-VPC peering through the Gateway enables communication between VPCs with overlapping IP address spaces by performing 1:1 stateless NAT. Stateful source NAT (masquerade) allows tenants to reach external networks through a shared IP, and the connection tracking inherent in stateful NAT doubles as a basic firewall — only return traffic for established flows is permitted, and unsolicited inbound traffic is dropped. For internet access, the Gateway peers with external networks and provides tenants with outbound connectivity behind NAT, similar to an AWS NAT Gateway.

Because the Gateway runs on commodity hardware, it can take advantage of modern NIC offloads for VXLAN encapsulation and NAT to deliver high throughput without consuming excessive CPU. And because it is managed by the same Fabric Controller and configured through the same Kubernetes API, operators provision Gateway services the same way they provision everything else — with a declarative YAML manifest.

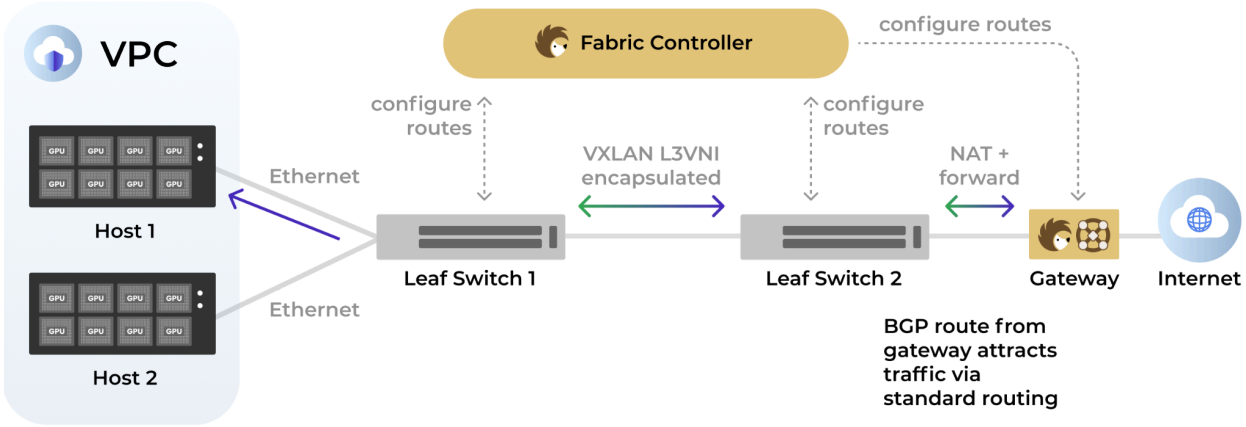

Figure 7: Gateway traffic flow. The Fabric Controller configures BGP route advertisements so that traffic requiring advanced processing (NAT, firewall, external connectivity) is attracted to the Gateway through standard routing. The Gateway processes the traffic and forwards it to its destination. To the switches, the Gateway is just another BGP peer.

AI Ops: Right Tool, Right Layer

The industry enthusiasm for applying large language models to network operations — "AI Ops" — deserves careful scrutiny when it comes to tenant isolation.

Consider what it means to use an LLM to directly generate switch configuration for a multitenant AI cluster. Adding a single tenant might change hundreds of lines of configuration across 20 to 30 devices: VRF definitions, VXLAN VNI mappings, BGP neighbor statements, route maps with community filters, ACLs for anti-transit protection, DHCP relay configuration, and more. The generated configuration is too complex for a human to meaningfully review. Can a network engineer stare at a diff of 1,000 lines of switch config across two dozen devices and confidently assert that no route map is missing a deny clause, no ACL has a gap, and no VNI is accidentally shared? In practice, the answer is no.

LLMs also hallucinate. They produce output that looks plausible but is subtly wrong. In a creative writing task, a hallucination is a minor annoyance. In a network configuration, a hallucinated route-map entry or a missing ACL rule means tenant data leaks across VPC boundaries. There is no formal guarantee of correctness, no way to prove that the generated config enforces the intended isolation.

But AI Ops is genuinely useful when applied at the right layer of abstraction. Since Hedgehog exposes its entire configuration as a Kubernetes API, it can sit behind a customer-facing panel that provides a polished UI for tenant provisioning, or it can be consumed directly via kubectl and standard automation. In either case, the Hedgehog API is also a natural target for AI Ops. An LLM can generate VPC manifests — small, readable YAML objects that say "create VPC tenant-42 with subnet 10.42.0.0/24, peer it with the storage VPC, and give it NAT access to the internet." The output is a handful of lines, and a human reviewer can verify it at a glance: does this VPC peer with the right networks? Does the NAT configuration match the customer's request? This is auditable in a way that 1,000 lines of switch config simply is not.

In a GitOps workflow, this becomes a practical pipeline: an LLM generates a pull request containing VPC manifests, a human reviews and approves the PR, and on merge the deterministic Hedgehog Controller translates the intent into correct switch config. **AI generates intent → a human validates that intent → a deterministic system enforces it.** The LLM operates where it is strong (translating natural-language requests into structured, high-level declarations), humans operate where judgment matters (reviewing whether the declared intent matches the actual request), and the deterministic Controller operates where correctness is non-negotiable (translating those declarations into switch config that enforces tenant isolation).

Example: Provisioning a Tenant

To make this concrete, here is what provisioning a tenant looks like with Hedgehog. First, create a VPC with a subnet and DHCP enabled:

```
None
apiVersion: vpc.githedgehog.com/v1beta1
kind: VPC
metadata:
  name: tenant-42
  namespace: default
spec:
  ipv4Namespace: default
  vlanNamespace: default
  subnets:
    default:
      subnet: 10.42.0.0/24
      gateway: 10.42.0.1
      dhcp:
        enable: true
```

That is the entire VPC definition. No VRFs, no VNIs, no route targets. Next, peer this tenant's VPC with a shared storage VPC so the tenant's GPU hosts can reach the storage system:

```
None
apiVersion: vpc.githedgehog.com/v1beta1
kind: VPCPeering
metadata:
  name: tenant-42--storage
  namespace: default
spec:
  permit:
    - tenant-42: {}
      storage: {}
```

Two VPC names and a permit block — that is the entire peering. The Controller will handle the route leaking, install the correct route maps and ACLs, and ensure that traffic cannot transit through the storage VPC to reach other tenants.

Finally, give the tenant NAT access to the internet via a Gateway:

```
None
apiVersion: gateway.githedgehog.com/v1alpha1
kind: Peering
metadata:
  name: tenant-42--internet
spec:
  peering:
    ext.internet-edge:
      expose:
        - default: true
    tenant-42:
      expose:
        - ips:
            - cidr: 10.42.0.0/24
          nat:
            masquerade: {}
```

The Gateway peering exposes the tenant's subnet behind masquerade NAT to the external internet connection. The tenant's hosts get outbound internet access; the Gateway handles the NAT; the Controller configures the border switches accordingly.

Compare these three manifests — roughly 40 lines of readable YAML — to the equivalent manual configuration: VRF definitions on every switch, VXLAN VNI allocation, BGP EVPN route target configuration, route maps with community strings, ACLs for anti-transit protection, DHCP relay setup, NAT rules, and firewall policies, replicated and coordinated across every switch in the path. The Hedgehog manifests are something a human can read, review, and approve in seconds. The equivalent switch config is something a human can only hope they got right.

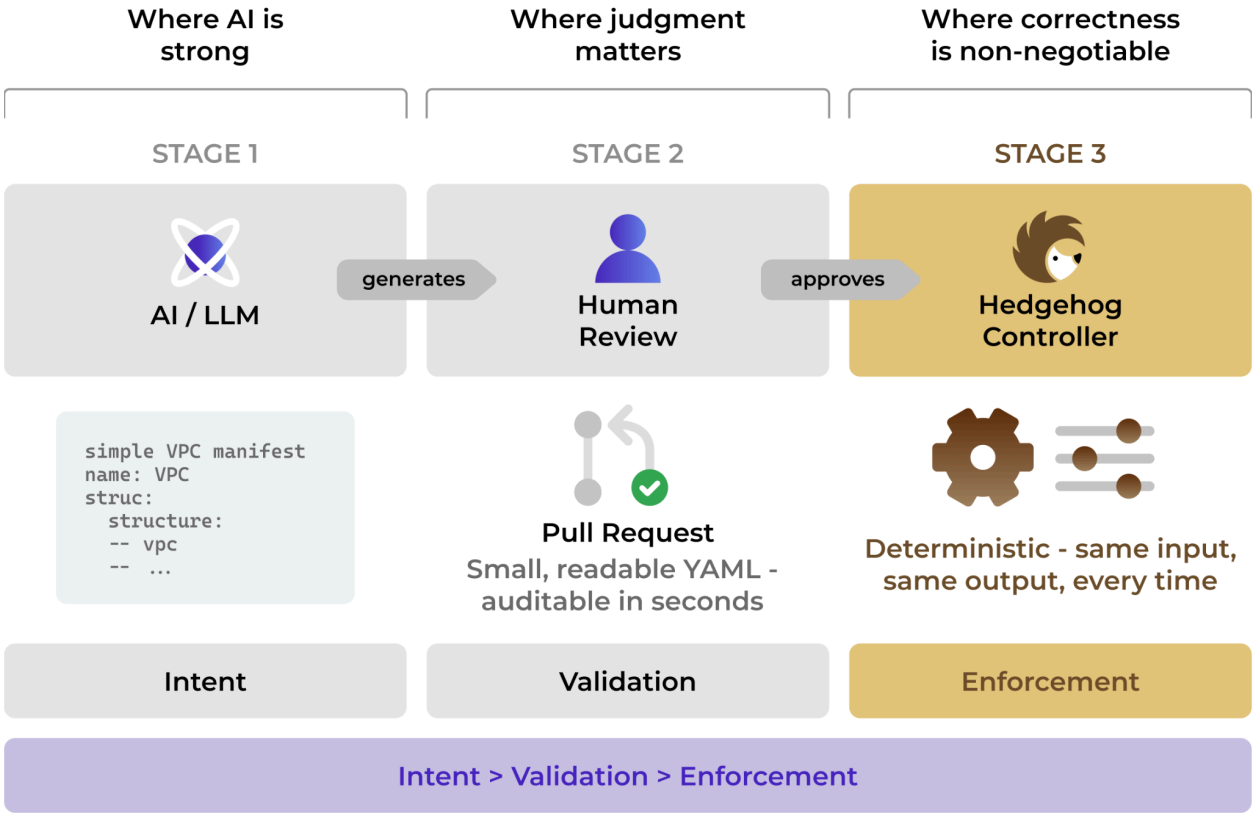

Figure 8: The right architecture for AI Ops in network configuration. AI generates high-level intent (VPC manifests) that humans can review and approve. The deterministic Hedgehog Controller then translates approved intent into correct switch configuration. Each layer operates where it is strongest.

GitOps and the Kubernetes Workflow

Because Hedgehog's entire configuration is declarative YAML, it fits naturally into a GitOps workflow [34]. The VPC manifests live in a Git repository alongside the rest of the cluster's infrastructure definitions. Argo CD [35] or Flux [36] watches the

repository and applies changes to the Kubernetes API automatically when commits land on the target branch.

This gives operators several things for free:

- **Full audit trail.** Every tenant change is a Git commit with an author, a timestamp, and a diff. When something goes wrong, git log shows exactly what changed and who changed it.
- **Rollback via revert.** If a new VPC peering causes a problem, git revert undoes the change and the GitOps controller reconciles the cluster back to the previous state.
- **Practical code review.** Reviewing a pull request that adds a VPC and two peerings is straightforward. Reviewing a pull request that modifies 1,000 lines of switch configuration across 25 devices is not.

This is also where the AI Ops pipeline described above fits naturally. An LLM generates a pull request containing VPC manifests, a human reviews the diff in a format they can actually understand, and on merge the GitOps controller and Hedgehog Fabric Controller handle the rest. The entire workflow is auditable, reversible, and safe.

Conclusion

Multitenancy is table stakes for any shared compute infrastructure, and cloud — AWS in particular — has set the standard for what users expect: VPC-level isolation, self-service provisioning in minutes, and connectivity that just works.

AI has changed the infrastructure game. The economics of GPU clusters — expensive hardware that depreciates fast and cannot sit idle — are driving enterprises and neoclouds to own their infrastructure rather than pay a cloud premium that is no longer a rounding error. Data privacy and sovereignty concerns accelerate this shift further. The result is that organizations that have never had to think about multitenant networking now need to solve it themselves, on-prem, at scale. Organizations that have built multitenant clusters usually do so within the context of a single Kubernetes cluster using a CNI plugin, but that approach breaks down when tenants need different clusters with different configurations, different kernels, or different versions of cluster-wide resources like CRDs. In practice, AI workloads drive exactly this heterogeneity, and the result is multiple Kubernetes clusters that must be isolated from each other at the physical network level.

This creates a genuine operational challenge. Host-level isolation via VMs is off the table because the IOMMU must be disabled for GPU performance. That leaves the network — and configuring BGP/EVPN, VXLAN, VRFs, route maps, and ACLs correctly across dozens of switches is exactly the kind of detailed, high-stakes work where manual processes and AI-generated config both fall short. The configuration is too voluminous and too intricate for a human to review meaningfully, and LLMs offer no guarantee that a generated config actually enforces the intended isolation. A single hallucinated route-map entry or missing ACL can leak tenant data.

Hedgehog addresses this with a layered approach. Operators express intent through a small set of Kubernetes CRDs — VPCs, peerings, Gateways — that are readable, auditable, and manageable with standard tooling. A deterministic Fabric Controller translates that intent into tested, correct switch configuration. The translation is proven software: same input, same output, every time.

The Hedgehog API can then be consumed by a "panel" solution that provides a nice user interface for tenant configuration, peering, and internet access, or the API can be used directly. It is in this context where AI Ops finds its proper role. Rather than asking an LLM to generate thousands of lines of switch config that no one can verify, the LLM generates a handful of VPC manifests and peerings that a human can review in seconds. The proven, deterministic, Hedgehog Controller then does the heavy lifting safely. Intent generated by AI, validated by humans, enforced by software — each layer doing what it does best.

The practical result is:

- **Secure multitenancy.** Tenant isolation is enforced by tested code, not by hoping that a manual or AI-generated config is correct. Anti-transit protections, ACLs, and route maps are installed automatically and consistently.
- **Reliable operations.** Continuous reconciliation on each switch corrects drift. Pre-flight validation catches errors before they reach production. Reprovisioning a tenant is a YAML change, not a multi-device manual procedure.
- **Ease of use and auditability.** The entire network state is expressed in declarative manifests that live in Git. Every change has a commit, a diff, and an author. Code review is practical. Rollback is a revert.

For enterprises and neoclouds building AI infrastructure, the path forward is not to make switch configuration easier to generate — it is to make switch configuration something that operators never have to see at all.

Visit Hedgehog.cloud to learn more!

References

[1] Amazon Web Services, "How Amazon VPC Works," *Amazon Virtual Private Cloud User Guide*. https://docs.aws.amazon.com/vpc/latest/userguide/how-it-works.html

[2] Amazon Web Services, "Enable Internet Access Using an Internet Gateway," *Amazon Virtual Private Cloud User Guide*. https://docs.aws.amazon.com/vpc/latest/userguide/VPC_Internet_Gateway.html

[3] Amazon Web Services, "NAT Gateways," *Amazon Virtual Private Cloud User Guide*. https://docs.aws.amazon.com/vpc/latest/userguide/vpc-nat-gateway.html

[4] S. Wang and M. Casado, "The Cost of Cloud, a Trillion Dollar Paradox," Andreessen Horowitz, May 2021. https://a16z.com/the-cost-of-cloud-a-trillion-dollar-paradox/

[5] SemiAnalysis, "AI Neocloud Playbook and Anatomy," October 2024. https://semianalysis.com/2024/10/03/ai-neocloud-playbook-and-anatomy/

[6] W. Richter, "AMZN, GOOG, MSFT, META, ORCL Plan $700 Billion in Largely AI-Related Capex in 2026," *Wolf Street*, February 2026. https://wolfstreet.com/2026/02/07/amzn-goog-msft-meta-orcl-plan-700-billion-in-largely-ai-related-capex-in-2026-heres-where-the-cash-comes-from/

[7] M. Mahalingam, D. Dutt, et al., "Virtual eXtensible Local Area Network (VXLAN)," RFC 7348, IETF, August 2014. https://datatracker.ietf.org/doc/html/rfc7348

[8] Broadcom/VMware, "VMware NSX 4.2 Installation Guide," *VMware NSX Documentation* (PDF). https://techdocs.broadcom.com/content/dam/broadcom/techdocs/us/en/pdf/vmware/nsx/nsx/vmware-nsx-4-2.pdf

[9] CNCF, "Container Network Interface (CNI) Specification." https://www.cni.dev/docs/spec/

[10] Tigera, "About Calico," *Calico Documentation*. https://docs.tigera.io/calico/latest/about/

[11] Cilium Project, "Introduction to Cilium & Hubble," *Cilium Documentation*. https://docs.cilium.io/en/stable/overview/intro/

[12] NVIDIA Corporation, "NVIDIA BlueField-3 DPU Datasheet." https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf

[13] NVIDIA Corporation, "NVLink and NVSwitch for Advanced Multi-GPU Communication." https://www.nvidia.com/en-us/data-center/nvlink/

[14] NVIDIA Corporation, "NVIDIA DGX B300." https://www.nvidia.com/en-us/data-center/dgx-b300/

[15] NVIDIA Corporation, "NCCL User Guide — Troubleshooting." https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/troubleshooting.html

[16] A. Sherwood et al., "Thunderclap: Exploring the Security of Thunderbolt Peripherals," NDSS 2019. https://thunderclap.io/

[17] NIST, "CVE-2019-5736 — runc Container Escape," *National Vulnerability Database*. https://nvd.nist.gov/vuln/detail/CVE-2019-5736

[18] NIST, "CVE-2024-21626 — runc 'Leaky Vessels' Container Escape," *National Vulnerability Database*. https://nvd.nist.gov/vuln/detail/CVE-2024-21626

[19] OpenContainers, "GHSA-9493-h29p-rfm2 — runc Container Escape via Masked Path Race (CVE-2025-31133)," *GitHub Security Advisory*. https://github.com/opencontainers/runc/security/advisories/GHSA-9493-h29p-rfm2

[20] Y. Shulman, "Finding Azurescape — Cross-Account Container Takeover in Azure Container Instances," Unit 42 (Palo Alto Networks), September 2021. https://unit42.paloaltonetworks.com/azure-container-instances/

[21] Broadcom Inc., "BCM78900 Series — Tomahawk 5 Ethernet Switch." https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm78900-series

[22] D. Dutt, *Cloud Native Data Center Networking*, O'Reilly Media, 2019.

[23] Cloudflare, "How Verizon and a BGP Optimizer Knocked Large Parts of the Internet Offline Today," June 2019.

https://blog.cloudflare.com/how-verizon-and-a-bgp-optimizer-knocked-large-parts-of-the-internet-offline-today/

[24] RIPE NCC, "YouTube Hijacking: A RIPE NCC RIS Case Study," February 2008. https://www.ripe.net/publications/news/industry-developments/youtube-hijacking-a-ripe-ncc-ris-case-study

[25] U.S. Department of Justice, "Seattle Tech Worker Arrested for Data Theft Involving Large Financial Institution," August 2019. https://www.justice.gov/usao-wdwa/pr/seattle-tech-worker-arrested-data-theft-involving-large-financial-institution

[26] Wiz Research, "ChaosDB: How We Hacked Thousands of Azure Cosmos DB Accounts," August 2021. https://www.wiz.io/blog/chaosdb-how-we-hacked-thousands-of-azure-cosmos-db-accounts

[27] Hedgehog, "Architecture Overview," *Open Network Fabric Documentation*. https://docs.hedgehog.cloud/dev/architecture/overview/

[28] The Kubernetes Authors, "Custom Resources," *Kubernetes Documentation*. https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/

[29] Hedgehog, "Build Wiring Diagram," *Open Network Fabric Documentation*. https://docs.hedgehog.cloud/dev/install-upgrade/build-wiring/

[30] Hedgehog, "VPCs and Namespaces," *Open Network Fabric Documentation*. https://docs.hedgehog.cloud/dev/user-guide/vpcs/

[31] Hedgehog, "External Peering," *Open Network Fabric Documentation*. https://docs.hedgehog.cloud/dev/user-guide/external/

[32] Hedgehog, "Fabric," *Open Network Fabric Architecture Documentation*. https://docs.hedgehog.cloud/dev/architecture/fabric/

[33] Hedgehog, "Gateway," *Open Network Fabric Documentation*. https://docs.hedgehog.cloud/dev/user-guide/gateway/

[34] CNCF GitOps Working Group, "OpenGitOps Principles v1.0.0." https://opengitops.dev/

[35] Argo Project, "Argo CD — Declarative GitOps CD for Kubernetes." https://argo-cd.readthedocs.io/en/stable/

[36] Flux Project, "Flux — the GitOps family of projects." https://fluxcd.io/flux/