Fachhochschule
Nordwestschweiz

Concept of a Sovereign and Scalable Hybrid Public and Community Cloud Architecture

Benjamin Koltermann

IP7 Project – August 23, 2026.
Master of Science in Engineering (MSE)

Supervisor: Prof. Dr. Sebastian Graf

MASTER OF SCIENCE
IN ENGINEERING

Contents

1 Introduction

Cloud computing has consolidated into a small number of very large providers. The arrangement is efficient and it has a consequence. An organization running its workload on infrastructure it does not own accepts a legal and operational dependency it cannot inspect. For public bodies and regulated industries that dependency has become a problem in its own right, independent of price or availability.

At the same time, a large amount of computing capacity already exists and sits unused across many owners. This work starts from the observation that pooling that capacity is a different problem from building another data center. The hardware is already there. What is missing is a way to share it between organizations that do not administer each other.

We designed and implemented a platform for exactly that. Several organizations each contribute a Kubernetes cluster, which we call a zone. A tenant defines its workloads once and chooses which zones may run them. Containers and virtual machines run in those zones on the tenant's own private network, and that network spans the zones even though no single party owns the result. The tenant keeps control over which jurisdiction holds which workload, because the tenant names the zones and no provider's routing decides the placement.

1.1 Scope

We assume that every participating organization already runs Kubernetes. We do not install it, we do not require a particular distribution, and we do not ask a contributor to change how they operate their cluster. This assumption is what makes the approach realistic. An organization that would have to adopt a new platform in order to contribute hardware will not contribute hardware.

The platform is delivered as a Helm chart. A zone owner installs it into their existing cluster and connects the zone to the hub. Three things then work. Containers run in a tenant network. Virtual machines run on that same network and are addressed the same way. Traffic between two zones crosses the public internet on the tenant's private addresses.

Several things are explicitly outside this scope. Storage is not part of this work, and that boundary limits what we can say about moving a stateful workload. Each zone provides its own persistent storage locally; we do not federate volumes, replicate them between zones or address what happens to data at rest when a zone leaves. Monitoring is likewise not addressed, and we did not build a federated view of metrics or logs. The operating model in Chapter 7 is a proposal: we describe how such a federation could be governed and funded, and we built none of the billing, membership or settlement it describes.

This is also not a production-ready system. We built and operated it on real clusters at two providers instead of in a laboratory, and that produced the findings here, but properties a production deployment would require are missing. Traffic between zones is not encrypted. The hub runs as a single instance. One of our two zones is limited to a single node, for a reason we explain in Chapter 4. We document these as limitations, because several follow from the architecture itself.

1.2 Contribution

The largest part of this work was not the platform above the network. It was making two Kubernetes clusters at two different providers share a single tenant subnet.

Connecting the control planes is straightforward, because that traffic is an ordinary outbound connection to an interface that already exists. Connecting the data planes is not. Two zones must build a tunnel directly between their gateway nodes, each must publish an address the other can reach, and the tenant networks on both sides must be joined into one address space without either cluster becoming authoritative over the other. Every assumption that holds inside a single data center stops holding here.

That part consumed most of the effort and produced most of the findings. We document four defects diagnosed on running clusters, and they share one property: none announced itself. In every case the clusters registered, the routes were exchanged in both directions and the health signals stayed green while no application traffic moved. Two passed a lossless ping at the full packet size while transferring nothing. Chapter 6 returns to what this means for testing a federated data plane.

Our contribution is therefore fourfold. We present an architecture for a peer-to-peer cloud in which authority is central and tenant traffic is not. We extend the interconnect of our network plugin, which joins only the default private cloud across clusters, so that a tenant network spans zones as well. We report what a zone must guarantee about its network before it can join, stated as a condition of membership. Finally we record the defects, the measurements that identified them and the

reasoning behind the decisions we took, including the fixes that turned out to be wrong.

1.3 Outline

Chapter 2 describes the motivation for sovereign infrastructure, the regulatory pressure behind it and the open-source components we build on. Chapter 3 presents the architecture, the split between zones and the hub, and how a tenant network spans several clusters. Chapter 4 presents the challenges we met when we built this on real clusters, which is the largest chapter and concerns the network almost entirely. Chapter 5 records the decisions we took and the alternatives we rejected. In Chapter 6 the measurements follow, and Chapter 7 discusses the governance questions that a federation of independent organizations raises. Finally Chapter 8 draws a conclusion and describes what we would do next.

2 Background

In this chapter, we summarize why sovereign cloud infrastructure is being asked for, why a federation of independently owned clusters is worth investigating, and which components we build on.

2.1 Why Sovereignty Has Become a Problem

Data sovereignty means keeping data inside a chosen geographic or political boundary. The requirement is not new, but it has become much harder to satisfy since most computing moved into a small number of very large providers.

European regulation is the most visible driver. The General Data Protection Regulation [4] places obligations on where personal data may be processed and who may access it. Several sectors add their own rules, and public administration frequently requires that data never leaves the country.

The difficulty is that the large providers are global by design. An organization that stores data in a European region of a large American provider is still exposed to the legal framework of the provider's home country, which the CLOUD Act [17] makes explicit for data held abroad. The physical location of the disk and the legal reach over the company that owns it are two different things, and only the first is easy to verify.

There is also a motivation that has nothing to do with regulation. An organization running its workload on infrastructure it does not control accepts a dependency it cannot inspect. For critical infrastructure and for public bodies this is increasingly treated as a risk in itself, independent of any specific law.

The market consolidated for a good reason. Building a data center is expensive and the economics only work at a scale very few organizations reach. Small and medium organizations therefore cannot answer the sovereignty requirement by building their own cloud.

2.2 Related Work

Several projects address parts of this problem. None addresses the combination this work is about, where the members neither administer each other nor trust each other.

Gaia-X [5] asks the same question at the level of rules. It defines how a provider describes itself and how a customer verifies a claim about jurisdiction, so that a provider becomes accountable for a promise. This report asks the question one layer down. Can hardware owned by several parties carry one tenant network, and what must each party guarantee first? The two are complementary. A federation of the kind described here would join such a framework as a member.

Federation on the control plane is the oldest answer. KubeFed [9] and Admiralty [1] distribute Kubernetes objects across clusters and schedule work between them. Both assume a common operator. One party holds the federated control plane, decides what goes where, and reads every member. This work removes that assumption, and removing it forces the rule of one writer per field in Chapter 5. A member publishes what it observes and receives what a tenant declared, and no member reads another member's tenants.

Liqo [7] offers a peer's capacity as if it were a local node. The abstraction is elegant, because an existing workload needs no change to spread across clusters. It also asks a cluster to schedule onto machines it does not own, so the two operators must trust each other with placement. Our members are competitors, and the tenant picks its zones itself.

Submariner [16] solves the network half, and it sits close enough to this work that we carry it as an optional component. Its route agent runs in the host network namespace of a gateway node. It finds the cluster interface by matching a host address against an advertised range, then programs a tunnel device and host routes there. A tenant network here is an isolated logical router with no such interface. Traffic never reaches the routes Submariner programs, and inbound packets have no way in. The limitation is structural and not a missing feature, and it is why we extend the interconnect of the network layer instead, as Section 5.2 describes.

None of these gives a tenant the choice of jurisdiction while the providers share hardware without sharing authority. That combination is what this work builds.

2.3 Pooling Capacity Instead of Building It

A large amount of computing capacity already exists and is not used. Many organizations run their own server rooms at a fraction of their capacity, and larger ones hold spare capacity for peak load that is idle for most of the year. Added up across many owners, that unused capacity is substantial.

Pooling it is a different problem from building more. The hardware is already there and already inside the right jurisdictions. What is missing is a way for organizations that do not administer each other to share it.

Such an arrangement offers three things a single provider cannot. The first is an explicit choice of jurisdiction: a tenant selecting which member runs a workload also selects the legal framework that applies to it. The second is resilience through independence, because capacity comes from several owners and the failure of one removes part of the system and not all of it. The third is the absence of lock-in, because a member joins or leaves without the tenant rewriting anything.

The economics follow. A small organization contributes hardware it already owns and gains access to resources it could not justify buying, while a larger one with idle capacity gains a use for it. Neither has to become a cloud provider to take part, and that keeps participation realistic.

2.4 Building on the Cloud-Native Ecosystem

We build the platform on existing open-source components instead of on something new, and this is a deliberate decision.

Kubernetes [13, 2] already abstracts a set of machines into a single scheduling surface, and most organizations that would contribute hardware already run it, which lowers the barrier to joining as we argue in Chapter 1. For a system whose purpose is to avoid dependence on a single vendor, software any participant can inspect and fork is not a secondary consideration. A sovereign infrastructure built on a proprietary control plane would reproduce the problem it is meant to solve one layer down.

Inspecting the source is the weaker half of the argument. The stronger half is that no layer is fixed. Kubernetes, the storage driver and the image registry are each interchangeable, and no component ties a member to one vendor or one country.

Table 2.1 lists the components and what each contributes.

Component	**What it provides**	**Role in this work**
Kubernetes	orchestration of workloads	the interface every zone exposes
Kube-Open Virtual Network (OVN)	virtual networks and routers	tenant networks and the interconnect
KubeVirt	virtual machines as first-class objects	workloads that cannot be containerized
Container runtime	packaging and isolation	consistent execution across owners
Longhorn and Rook	distributed block storage	persistent storage inside one zone
Prometheus and Fluentd	metrics and log collection	operation of a zone by its owner
Helm	packaging and release management	how a zone owner installs the platform

Table 2.1: Components the platform is built on and what each contributes

Only the first three rows are load-bearing for this work. Kubernetes provides the surface, Kube-OVN [12] provides the tenant networks and the mechanism that joins them across clusters, and KubeVirt [14] makes virtual machines behave like any other workload on those networks. Almost every difficulty described in this report comes from the second of the three.

The remaining rows are listed for completeness and are not part of this work. Storage stays inside a zone, because moving persistent data between jurisdictions is what a tenant may need to avoid, and we neither federate volumes nor address what happens to them when a member leaves. Monitoring is left to each zone owner, who operates their own cluster with their own tooling. Ordinary monitoring of that kind does not detect the failures that matter most in a federated data plane, and we return to this point in Chapter 4.

3 Architecture

This chapter describes the architecture we designed and built: the two kinds of cluster, the rule that governs which may write what, the objects a tenant works with, and how a tenant network spans several zones.

3.1 The Architecture as Built

This section gives the whole picture before the parts. Figure 3.1 shows the system as we built it: one hub, two zones, and the two channels that run between every pair of zones.

Three properties of that picture carry the rest of the chapter. Authority is central and traffic is not, so the hub holds what a tenant declared while packets go straight between gateways. A tenant appears in a zone only when it has named that zone, which is where the sovereignty claim comes from. The platform and the tenant keep separate channels with separate address ranges, so a mistake in one cannot silently take out the other.

Everything below is the same system in more detail. Each zone runs Kubernetes with Kube-OVN for the networks and KubeVirt for the virtual machines, and the platform arrives as a single Helm chart.

3.2 Zones and the Hub

The architecture has exactly two kinds of cluster. A zone is a Kubernetes cluster contributed by an organization. It runs the workloads, it owns the hardware, and its operator keeps their own distribution and their own operational practices. Any distribution works, because we rely only on interfaces that all of them provide.

The hub is a single small cluster that holds the definitions. It is not a scheduler, runs no tenant workload and does not administer the zones. It exists because some questions have no answer inside a single zone. A zone cannot know which other zones exist, cannot authenticate a user it has never seen, and cannot decide alone which zone currently runs a given virtual machine.

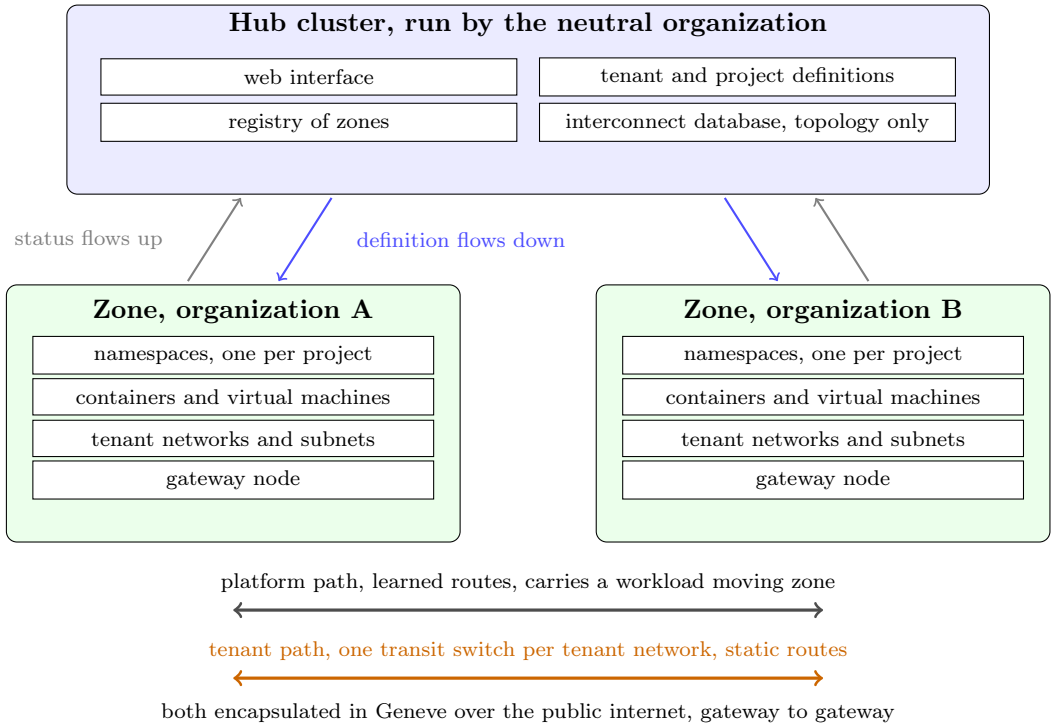


Figure 3.1: The system as built. Definitions travel through the hub, tenant traffic never does, and the two cross-zone channels are kept apart

Figure 3.2 shows the two kinds of cluster and what each one contains. Tenant traffic never passes through the hub. Two zones exchange packets directly, so the data plane is a mesh even though the authority is central. If tenant packets crossed a central point, that point would become a jurisdiction of its own and the architecture would defeat its own purpose.

One property of the hub is architectural and not organizational. It must be operated by a party that is not itself a zone owner. A registry run by one of the participating providers would give that provider sight of every other provider's tenants and a say in a record its competitors depend on. Chapter 7 describes who we propose should operate it.

The rule that keeps this workable is that every field has exactly one writer. Chapter 5 states the rule in full, together with the alternatives we rejected.

3.3 What a Tenant Works With

A tenant is an organization using the federation, which is not the same as one contributing to it. The two roles are independent: a company can contribute a

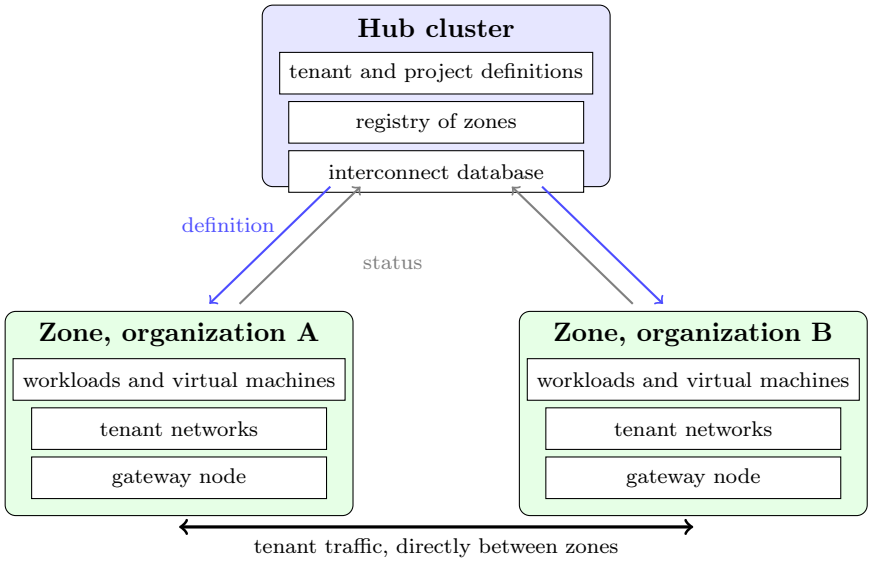


Figure 3.2: The two kinds of cluster. Definitions travel through the hub and tenant traffic does not

zone without placing any workload, and place workloads without contributing anything.

A tenant names the zones it wants to use. This choice is the mechanism behind the sovereignty claim, because a zone only ever sees the tenants that named it. A tenant that names no zone exists nowhere, which is the safe default. The opposite default would place every tenant on every partner's hardware.

Inside a tenant, work is grouped into projects, and a project holds the resources. A virtual private cloud is a private network with its own address range, divided into subnets that workloads attach to. Containers and virtual machines attach to the same subnets and are addressed the same way, so a tenant does not treat them differently.

A project maps to a Kubernetes namespace in every zone the tenant named. A tenant deploys into its own namespaces and into no others. The roles that grant this are generated from a single list in the code, and a test compares that list against the chart on every build. Two defects in exactly this layer are described in Section 6.8.

3.3.1 Moving a Workload to Another Zone

There is no migration object. Changing the zone of a workload is the migration, and the state is the pair of the requested zone and the reported zone. While the two agree, the workload is settled. While they differ, the source zone keeps the

Kind	Group	Scope	What it holds
Tenant	cloud.ch	cluster	the named zones, the tenant namespace
Cluster	cloud.ch	cluster	a zone: endpoint, gateway nodes, reachability
Project	tenant.cloud.ch	namespaced	a project and the namespace it maps to
VPC	tenant.cloud.ch	namespaced	a private network and the zones it spans
Subnet	tenant.cloud.ch	namespaced	an address range placed in one zone
CrossZone	tenant.cloud.ch	namespaced	confirms a subnet may leave its zone
Container	tenant.cloud.ch	namespaced	a container attached to a subnet
VM	tenant.cloud.ch	namespaced	a virtual machine and where it currently runs
Account	tenant.cloud.ch	namespaced	a user or a service identity
Grant	tenant.cloud.ch	namespaced	a role held inside one project
TenantGrant	tenant.cloud.ch	namespaced	a role held across the whole tenant

Table 3.1: The custom resources the platform defines, in two API groups

workload until the target reports that it holds it, and only then does the source tear down.

The source never lets go before another zone has claimed the workload, and that makes a mistyped zone name harmless. A name matching no zone leaves the workload held where it is and placed nowhere else, so the cost is a stalled move and not a deleted machine.

A workload gets a new address on arrival, from a subnet of the target zone, in the same way that changing zone changes address on a public cloud. Keeping the old address would require the same range in both zones, and two identical prefixes inside one private cloud cannot both be routable.

3.4 The Objects as a Kubernetes API

Everything above is a Kubernetes object. The platform adds two Custom Resource Definition (CRD) groups to every cluster it runs in, and a tenant works with them through `kubectl` or through the web interface without the two being different. The operator is the part that turns those objects into routers, subnets, pods and virtual machines inside a zone. Table 3.1 lists the full set.

Several of these carry a claim made earlier in this chapter. The zones a tenant named are a field on the tenant object, so the sovereignty choice is one value and not a policy spread across components. A private cloud carries the list of zones it spans, which is what lets one network exist in several places. The confirmation that a subnet may be reached from another zone is its own object with a reason field on it, which is why an approval can be audited later. A virtual machine records where it currently runs, and that field is the handover state during a move. A zone record reports whether it is reachable, and Chapter 8 returns to why that is not enough. The accounts and

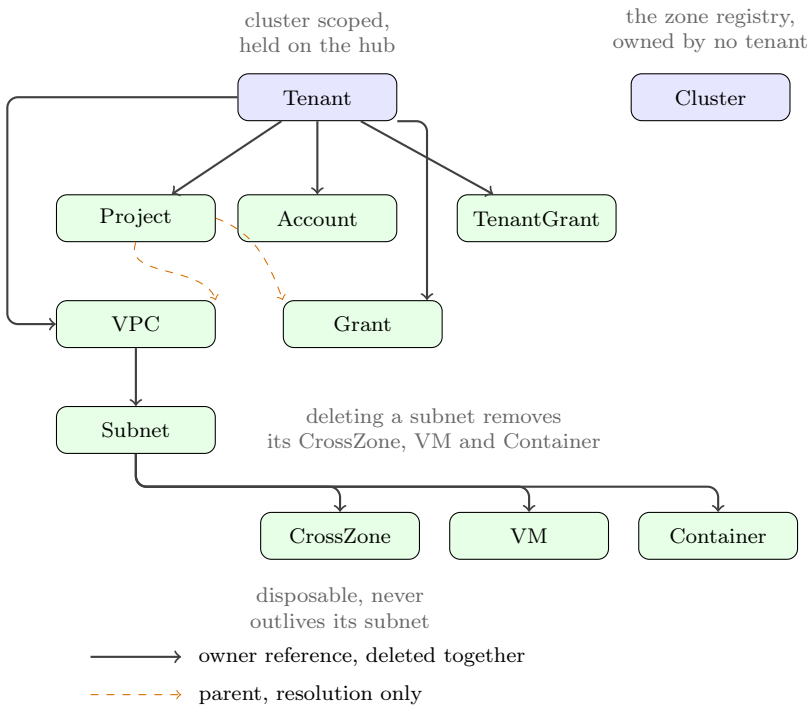


Figure 3.3: How the custom resources relate. The two dashed edges are the places where the logical parent and the owner are not the same object

the two kinds of grant are the identity layer whose generated roles produced both defects in Section 6.8.

Figure 3.3 shows how the objects relate. Two relations run through the set and they are not the same. The parent relation is the logical hierarchy the operator resolves a request through. The owner relation is the Kubernetes `ownerReference` that decides what is deleted with what. They agree in most places and part in two.

A private cloud sits under a project and belongs to the tenant. A grant does the same. Both are reached through the project, and both survive it, because a tenant that removes a project should not silently lose the network its workloads run on. Everything below a subnet belongs to that subnet, so deleting one address range takes its confirmations, its virtual machines and its containers with it. The confirmation object is marked disposable, and the operator refuses to let a disposable object be owned by the tenant, so an approval to cross a border can never outlive the subnet it was granted for.

Listing 3.1 shows the two fields that carry the most weight.

The tenant object also reports, for each kind, whether a zone has established it. The hub therefore knows not only which zones exist but which part of the API each

```
apiVersion: tenant.cloud.ch/v1alpha1
kind: VPC
metadata: {name: production, namespace: acme}
spec:
  zones: [az-switch, az-hetzner]
  defaultSubnet: web
---
apiVersion: tenant.cloud.ch/v1alpha1
kind: CrossZone
metadata: {name: web, namespace: acme}
spec:
  subnet: web
  reason: the frontend in Zurich reaches the database in Falkenstein
```

Listing 3.1: A private cloud spanning two zones, and the confirmation that lets one of its subnets be reached from the other zone

of them mirrors, and a zone that lags behind on one kind is visible without asking it.

3.5 Tenant Networks Across Zones

A virtual private cloud can exist in more than one zone. Each zone materializes the part that lives locally, a router and the subnets placed there. The halves are joined so that a workload in one zone reaches a workload in the other over the tenant's own private addresses.

The join is made by a transit switch, which is a small shared segment that both routers attach to. Figure 3.4 shows the arrangement. Only the topology is published to the interconnect database. No tenant packet is ever sent to it.

Two such paths exist between any pair of zones and are kept apart on purpose. The platform path carries traffic between the clusters themselves, and a workload uses it when it moves between zones. The tenant path carries tenant traffic. Each has its own transit switch and address range, and the two ranges may never overlap. The platform path learns its routes as zones register; the tenant path takes a static route per confirmed subnet, so nothing crosses a border without the confirmation described below.

Each zone computes the addresses of its own attachment instead of requesting them. The name of the transit switch and the address a zone takes on it are both derived from the identity of the tenant network, so two zones that have never spoken arrive at the same answer independently. There is no allocator, no lease and no round trip, so there is nothing to disagree about and nothing to fail.

The derivation works only because both zones read the same object from the hub and so agree on the identity of the network and the order of the zone list. Given that

Algorithm 3.5.1: Deriving the transit switch and the address this zone takes on it

Input: the tenant, project and network names, the declared list of zones Z, this zone z

Output: the name of the transit switch and the address to configure locally

$h \leftarrow$ sha256(tenant/project/network);
name $\leftarrow$ the prefix *ts-* followed by the first twelve hexadecimal characters of h;
$i \leftarrow h \bmod B$; `// B is the number of available address blocks`
block $\leftarrow$ the ith block of the reserved transit range;
$r \leftarrow$ the position of z in Z;
address $\leftarrow$ *block* $+ r + 1$;
return *name* and *address*;

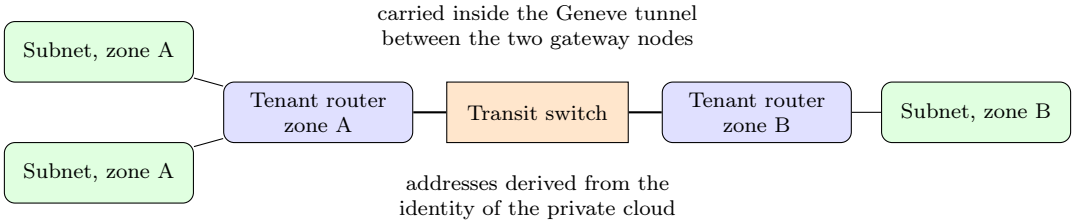


Figure 3.4: Two halves of one tenant network joined by a transit switch

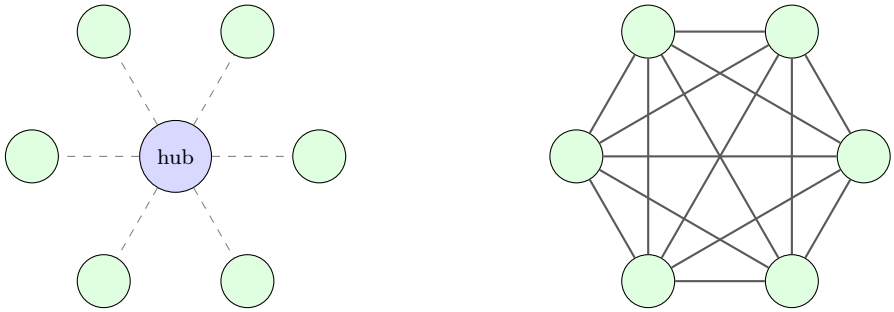
one agreed input, everything else follows locally. This is what the shared registry buys.

A subnet does not become reachable from another zone because the network spans both. A project administrator must confirm each subnet explicitly, and until then the network reports honestly that it advertises nothing. We made this a separate object instead of a field, because an object is easier to audit than a flag, and because approving traffic that leaves a jurisdiction deserves its own record.

3.6 The Federation at Scale

So far we have described two zones, because that is what we built. The architecture is meant for many more, so this section describes the shape it takes at that size and estimates how large the peering database becomes.

Figure 3.5 shows the two structures side by side. Registration is a star: every zone has a relationship with the hub and none with any other, because there is only one. Tenant traffic is a mesh: every zone sharing a tenant network with another builds a tunnel directly to it. The federation has a central component and a decentralized data path at the same time.



Registration
one relation per zone

Tenant traffic
a tunnel per sharing pair

Figure 3.5: Registration is a star through the hub and tenant traffic is a mesh between the zones

3.6.1 What the Peering Database Holds

The peering database is small by design, because it holds topology and nothing else. It records four kinds of thing.

The first is the list of zones, each registering once and staying until it leaves. The second is one transit switch per tenant network that spans more than one zone; a network confined to a single zone never appears, and most are of that kind. The third is one attachment per zone that a tenant network reaches into. The fourth is one route per confirmed subnet, advertised by the zone that owns it.

No tenant packet, no workload address and no credential is ever stored there. A reader learns which zones exist and which address ranges are reachable through which of them, the same information a routing registry on the public internet holds.

3.6.2 How Many Entries

We can estimate the size directly. Let N be the number of zones, V the number of tenant networks that span more than one zone, z_v the number of zones a given network reaches into and s_v its number of confirmed subnets. The number of entries is then

$$E = N + V + \sum_{v=1}^{V} (z_v + s_v) \tag{3.1}$$

Writing $\bar{z}$ and $\bar{s}$ for the averages, Equation 3.1 simplifies to a form that is easier to read.

	Small	**Medium**	**Large**
Zones	5	50	200
Tenant networks crossing a border	20	500	5000
Zones reached per network	2	3	3
Confirmed subnets per network	3	4	4
Entries in the peering database	**125**	**4050**	**40200**
Tunnels held by one gateway, worst case	4	49	199
Tunnels in the federation, worst case	10	1225	19900

Table 3.2: Estimated size of the peering database and the tunnel mesh at three federation sizes

$$E \approx N + V\,(1 + \bar{z} + \bar{s}) \tag{3.2}$$

What is absent from Equation 3.2 matters: there is no term in N^2. The database grows with the number of tenant networks that cross a border and how far each reaches, not with the number of possible pairs of zones. Adding the fiftieth zone costs one entry plus whatever that zone participates in.

Table 3.2 applies this to three federation sizes. The subnet and zone counts per network are estimates and not measurements, since we operated two zones and cannot claim otherwise.

40,200 rows is a small database. The peering database is an ordinary transactional store, and a table of this size is not where a federation of 200 members would run into trouble.

3.6.3 What Grows With What

The tunnel counts in Table 3.2 are the quadratic term. They look alarming and are not.

A gateway builds one tunnel to each peer gateway it shares a tenant network with, so it holds at most $N-1$ tunnels, and the federation at most $N(N-1)/2$. The number is quadratic in the zones and in nothing else. It grows with neither the number of tenants nor the number of tenant networks, because one tunnel between two zones carries every tenant they have in common and the tunnel identifier separates them. A federation of 200 zones has a gateway holding a couple of hundred tunnels, whether it serves 10 tenants or 10,000.

This is also the difference between an automatic mesh and a configured one. Nobody writes those tunnels down. Two gateways build one once both have published an address and remove it when the last shared network goes away. Chapter 5

contrasts this with an approach where each pair is configuration somebody maintains.

The quantity that does grow with the tenants is the state inside each zone, which is a local concern and not a federation one. A zone carries the networks of the tenants that named it and nothing else. A member serving few tenants carries little, no matter how large the federation around it becomes.

3.7 Networking and Virtualization Components

Each zone runs Kube-OVN as its network plugin. It supplies the virtual private clouds, the subnets and the routers that the transit switches above are built from. Where a zone already runs a network plugin of its own, Multus [8] allows the two to coexist, so a contributor does not have to replace the networking of an existing cluster in order to join.

Traffic between zones is encapsulated in Geneve and travels over the public internet between the gateway nodes of the two zones. This is the part of the architecture that proved hardest to make work in practice, and Chapter 4 is largely about it.

Virtual machines are provided by KubeVirt [14]. A tenant that cannot containerize a workload runs it as a virtual machine on the same networks, managed through the same interface.

Tenants reach the platform through a web interface the hub serves. It carries no framework, no external stylesheet, no web font and no build step, and neither repository holds a package manifest. Its response header permits the page to contact its own origin and nothing else, so a reference to an outside host fails instead of quietly widening what the platform depends on. A test asserts that header, which keeps the property from decaying into an intention.

A tenant network has no gateway to the internet. No tenant workload reaches a public address, and nothing reaches a tenant workload from outside the federation. This is why disk images come from a catalog that each zone mirrors locally. A machine starts from a copy held inside the zone, so starting it depends on no registry in another jurisdiction and on no egress path that does not exist.

4 Challenges

In this chapter, we set out the problems a federation of independently owned clusters has to solve. We pose them as questions, because most admit several answers. Chapter 5 states what we chose and why, and Chapter 6 reports what happened when we built it.

4.1 Where the Authoritative State Belongs

A tenant is defined once and must then exist in several clusters at the same time, each owned by a different organization. This is the first question the design has to answer, and it constrains everything after it.

If every cluster may write the definition, then two of them can disagree and nothing decides between them. Disagreement here is not an abstract concern. One field records which cluster currently runs a given virtual machine, and a second writer on that field can start the same disk image twice or tear it down everywhere. Deletion is ambiguous in the same way. Chapter 5 works both cases through.

If instead one place holds the definition, those problems disappear and a new one takes their place. That place is now special, in a design whose claim is that no participant is privileged. Whoever operates it sees which tenant placed what where, and every member depends on it being available.

Figure 4.1 shows the shape of the compromise we describe in Chapter 5. Neither direction is a copy. What a tenant declares travels one way and what a cluster observes travels the other, so no field ever has two authors.

4.2 Where the Peering Database Belongs

Joining the networks of several clusters needs a second shared record, and it is a different question from the first with a different failure mode.

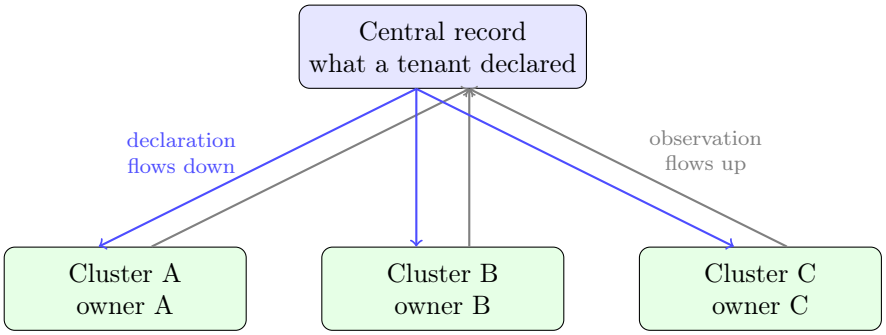


Figure 4.1: Splitting an object so that no field has two authors

This record holds topology: which clusters exist, which address ranges are reachable through which of them, and how the joining segments are named. Every cluster has to reach it and somebody has to run it, so it faces the same objection as the first record.

What makes it a separate question is what it must not hold. A registry of reachable address ranges is not a registry of traffic. No packet, no workload address and no credential belongs in it. Keep that line and the record is comparable to a routing registry on the public internet, which many independent parties share without any becoming privileged. Lose it and the central component quietly becomes a transit provider.

A second property matters here. Two clusters have to agree on the identity of a shared segment and on the addresses they take on it. They can negotiate, which needs a protocol and an allocator and adds a step that can fail, or derive the answer from something they both know. The second option exists only if there is a shared record to derive it from, which argues for the central record and not against it.

4.3 How Clusters Communicate

Two clusters that federate have two separate communication paths. Treating them as one problem is a mistake.

The first path carries definitions and status. It is an ordinary outbound connection to an endpoint that already exists and is already reachable, because a cluster nobody can reach is not usable by its own owner either. Address translation does not obstruct it, nor does a filter permitting outbound connections, and only one of the two parties needs to be reachable.

Almost none of that holds for the second path, which carries tenant traffic. Both parties must be reachable by the other, because neither is a client of the other. The

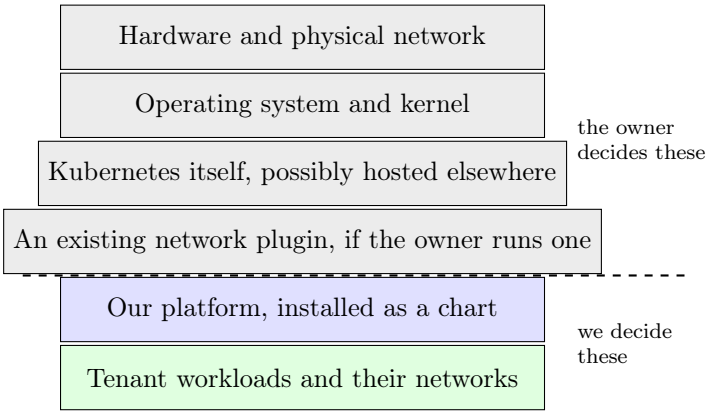


Figure 4.2: The boundary between what a zone owner controls and what the platform controls

traffic belongs to a tenant whose private addresses may collide with addresses in the receiving cluster, so it cannot be routed directly. It has to be carried inside something, so the two clusters must agree on an encapsulation, on a packet size that survives the path, and on a port both networks permit.

A cluster can therefore register, exchange topology correctly and carry no tenant traffic at all. The two paths succeed and fail independently, so any design that infers the health of the second from the first will report a working federation that moves nothing.

4.4 Operating Inside a Cluster We Do Not Control

The property that makes this architecture worth building is also the source of most of its difficulty: a zone is a cluster somebody else owns. We cannot choose the distribution, require a particular network plugin, assume a version, or ask an owner to change how they operate a cluster that serves their own business. Chapter 1 states this as an assumption about contributors; here it constrains everything the platform may require of them.

We also do not control the network the cluster sits in. Figure 4.2 shows where the boundary falls. The provider decides whether a public address appears on a machine or only on a router in front of it, which ports are permitted between machines, and what packet size survives between sites. Each of those can be a fixed property of a commercial offering and not a setting, so the platform has to work with whatever it finds.

A managed offering moves the boundary further. Where the provider hosts the control plane in their own infrastructure, it never appears among the machines of the cluster, so software that expects to place a component there has nowhere to put it. Chapter 6 reports the case that cost us an install.

Diagnosis is constrained in the same way. In a cluster we own we would read the host, capture on its interfaces and inspect its state directly. Here we see what the Kubernetes interface will tell us, and when the network is broken that interface is frequently the first thing to stop answering. The platform has to stay diagnosable through the narrow channel that is left.

4.5 Trust Between Parties That Do Not Trust Each Other

Every participant is exposed to organizations it has no relationship with. A tenant runs workloads on hardware owned by someone else, and a zone owner runs workloads belonging to someone else. Neither may see the other's data, and neither may be asked to take the other on faith.

The first consequence concerns what a member may know. A member should see the tenants that chose to place work with it and nothing else. That rules out any arrangement copying the shared record to every participant, because a copy in every cluster means every member holds every tenant's records, including those of tenants that chose not to place anything with them.

The second concerns identity. Somebody has to vouch for who a user is, and no member can do it for another, because a member issuing identities for the whole federation would hold authority over its competitors. The same problem appears for machines: two clusters that have never exchanged a credential cannot authenticate each other's traffic.

The third concerns the traffic itself. It crosses a network neither party owns. Confidentiality is the obvious requirement and it is not the only one, because a receiver that cannot tell a genuine packet from a forged one has a problem that encryption alone does not solve.

4.6 Knowing Whether It Works

The last problem follows from all the others, and we underestimated it.

A federated data plane spans organizations, so no single party observes it end to end. Each side sees its own half. The signals a cluster can produce about itself are therefore local by construction. A component reports that it is running, a database reports that

it agrees with its peer, a registration reports that it succeeded. None of those observations involves a tenant packet reaching the other side.

The usual health signals are therefore not just incomplete. They answer a different question from the one being asked of them, and they answer it correctly. A federation can register, exchange topology, report every component healthy and carry nothing. Any claim that such a system works has to rest on an observation that crosses the boundary, and choosing that observation is part of the design.

4.7 Research Questions

The problems above are what this work had to solve in order to answer the two questions the project was set. We restate them here in the terms this report uses.

RQ 1 How do different levels of virtualization abstraction affect the interregional migration capability of workloads, and what requirements do stateful instances place on the synchronicity of storage and network when moving between cloud domains?

RQ 2 To what extent does a software defined network architecture allow consistent security and isolation policies to be enforced in a multi-tenant environment, in order to guarantee technological sovereignty over distributed infrastructure resources?

5 Design Decisions

Here we record the decisions we had to take and the reasoning behind them. Some were forced by the challenges in Chapter 4, others taken before we knew a problem existed. We also record the options we rejected, because the reasons carry more than the outcome.

5.1 Overlay Networking and Its Terminology

The traffic between two zones runs over an overlay network. The term is used loosely, so we fix it here. The underlay is the network that already exists between the two sites, in our case the public internet. The overlay is the tenant network we build on top of it, where two pods in different countries share an address range and behave as if they sat on the same switch.

The mechanism that joins them is encapsulation. A tenant packet is placed unchanged inside a new packet, that new packet is routed over the underlay, and the far side removes the wrapper. Figure 5.1 shows the layout for Geneve [6], which is the format Kube-OVN uses.

Two terms from this picture recur throughout the report. The tunnel endpoint is the address in the outer header, the one a peer must be able to reach. The tunnel identifier separates one tenant network from another, so two tenants can use the same private range without meeting. The difficulty in Section 6.4 comes from the first, because a zone behind address translation cannot hold its own tunnel endpoint address on an interface.

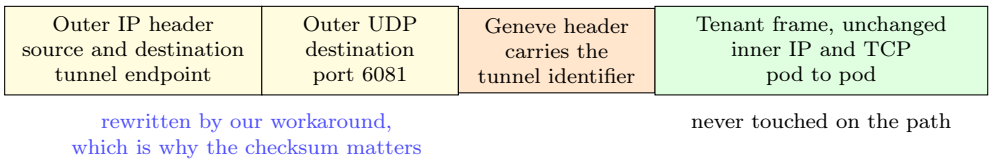


Figure 5.1: A tenant frame carried inside a geneve packet

5.2 Interconnecting a Tenant Network

The network plugin we build on does not join tenant networks across clusters, and the algorithm in algorithm 3.5.1 exists because of it.

Kube-OVN solves two neighboring problems. It peers custom private clouds inside one cluster, and interconnects the default private cloud across clusters. Neither covers a tenant network spanning several zones, the one case this work needs, because the interconnect controller names the default router in its own code and a tenant router is never offered to it.

The limit belongs to the controller and not to the layer beneath it. Open Virtual Network attaches any logical router to a transit switch, and its documentation [15] treats those switches as ordinary logical switches, so the gap is small. Our operator creates the transit switch and the router ports itself and leaves the rest of the stack alone. The commands are idempotent, so a zone that already holds the attachment does nothing on the next pass.

This is also why the addresses are derived and not assigned. The stock controller would have been the allocator, and without it there is no component that could hand out an address. Both zones therefore compute the same answer from the identity of the tenant network, and no round trip is needed to agree.

One range has to stay clear. The transit addresses of the tenant path may never overlap the range the plugin uses for the platform path, because a tenant route covering the platform range suppresses it. We hold this as a chart validation that refuses to render and a test that pins the ranges apart, not as a comment. Section 6.7 reports what happens when such a range slips through.

The zone admission contract in Section 5.5 carries the same requirement one level up. A member must use pod and join ranges that no other member uses, because the platform path learns those ranges from its peers.

5.3 Encrypting the Interconnect

The tunnel between our two zones crosses the public internet in cleartext. We confirmed this with a packet capture instead of assuming it. Anybody who can observe the path can read tenant traffic, and Geneve carries no authentication either, so a packet from a forged source is accepted as readily as a real one. For a design whose purpose is data sovereignty this is the weakest point in the system.

Open Virtual Network [15, 10] ships an answer. When enabled, each chassis obtains a certificate and the tunnels are wrapped in IPsec, with tenant traffic unchanged

and no second overlay. It is attractive because it costs one setting and not a new component.

The weakness is where the trust is anchored. Each cluster signs its own certificates with its own authority. Inside one cluster that is sound, because the cluster owns both ends. Across a federation it is not, because each partner decides for itself which chassis it trusts, and two zones that have never exchanged a root cannot authenticate each other. A partner willing to sign a certificate for a chassis that is not theirs can also join the fabric. Making this work needs a shared authority or a deliberate cross-import of roots, which is a governance question and not a configuration one. A body that every member governs and that already operates the registry is the natural place to anchor that trust, as Chapter 7 describes.

5.3.1 Why We Did Not Choose WireGuard

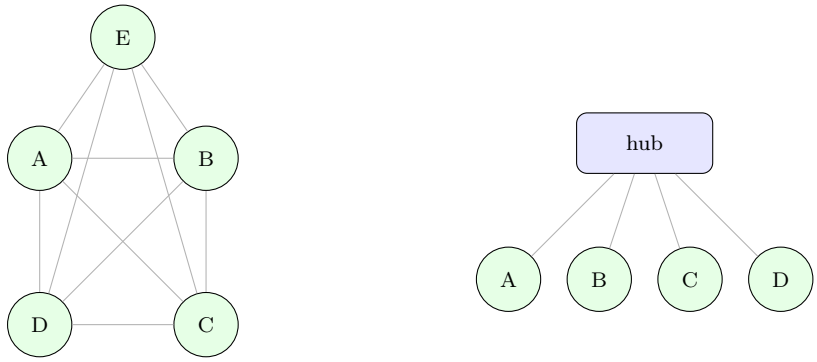
WireGuard [3] is the obvious alternative and it is a good protocol. It is also a poor fit for this particular problem, for reasons that have little to do with cryptography.

First, WireGuard is a routing construct and not only an encryption layer. Every peer carries a list of allowed addresses that decides where traffic goes. Our tenant ranges are created and destroyed while the system runs, and every such change would have to be pushed into the peer configuration of every member. The overlay already solves that, so we would maintain a second routing system beside the one we have.

Second, the peering has to be configured for every pair. Our data plane is already a full mesh, so the difference is the effort per pair. A Geneve tunnel appears on its own once both sides publish an address, while a WireGuard peer relation is configuration somebody writes and keeps current on both ends. A federation of 10 members needs 45 such relations, one of 100 needs almost 5,000, and Figure 5.2 shows how that grows. Every member also needs a public endpoint every other member can reach, the requirement that gave us the most trouble in practice, now multiplied by the members.

Third, the overhead adds up. WireGuard costs 60 bytes and Geneve another 50. On a normal 1500 byte path that leaves 1390 bytes for the tenant, and the setting must be correct on every member at once. We already carry a reduced size of 1400 bytes for the Geneve tunnel alone.

None of this makes WireGuard a bad tool. It makes it the wrong layer for this job. The traffic is already encapsulated, so encryption belongs to the encapsulation and not beside it.



a peer relation for every pair one published address per member

Figure 5.2: Configured peer relations grow with the square of the members while published addresses grow linearly

5.4 One Writer per Field

The first decision we took had the widest consequences. A federation needs a place where the definition of a tenant lives, and since every member is an independent organization, the obvious idea is that each keeps its own copy and the copies agree. We did not do that, and the reason is not the one we expected.

The rule we settled on is that every field has exactly one writer. The tenant defines the specification on the hub and no zone may change it. A zone writes only what it observes locally, and the hub never contradicts it. Nothing in the system merges two versions of the same field, because no field ever has two versions.

5.4.1 Why Not a Copy in Every Cluster

A replicated database is the natural answer to a distributed problem. Three arguments against it here are about consistency, and a fourth settles the matter.

First, two writers on one field produce a value that flaps. Before the rule was in place, a status field alternated every few seconds depending on which zone had reported last. Nothing was broken and no error appeared. The field stopped meaning anything.

Second, one field in the system decides which zone currently runs a virtual machine. If two zones can write it, both can believe they hold the machine, or neither can. The first case runs one disk image twice and the second tears it down everywhere. There is by design no separate object that records a migration, because a second

object stating where a machine belongs would be a second answer to the same question.

Third, deletion becomes ambiguous. When a record disappears from one copy, a member cannot tell whether the tenant removed it or whether it has not arrived yet. The two need opposite responses.

The fourth argument decides it, and it has nothing to do with consistency. Our design gives a zone sight of a tenant only when that tenant has named the zone. Replicating the database to every member means every partner holds the records of every tenant, including tenants that chose not to place anything with them. That is the outcome the system exists to prevent. A design whose normal operation contradicts its own purpose cannot be the right one, however well it converges.

5.4.2 The Alternatives We Considered

Table 5.1 summarizes the options against the properties we need. The comparison is not about which technology is better in general. It is about which one can hold state for organizations that do not trust each other with it.

Approach	**Survives a slow link**	**No merge needed**	**Unambiguous placement**	**Keeps data local**
One writer per field, central registry	yes	yes	yes	yes
Replicated multi-master	yes	no	no	no
Consensus stretched across the wide area	no	yes	yes	no
Conflict-free replicated types	yes	yes	no	no
Gossip-based membership	yes	no	no	no
Distributed ledger	yes	yes	yes	no

Table 5.1: Approaches to holding the authoritative state of the federation

The last column eliminates most of the table. Every approach that replicates state to every member fails it by construction, the ledger included. Conflict-free types converge without a merge conflict, but converging is not the same as being correct: two zones can agree perfectly on a value saying both hold the same virtual machine.

5.4.3 What Centralizing Authority Costs

The price is real. If the hub is unavailable, no tenant can be created, changed or deleted, and no zone can report its state upwards. We have not deployed the hub redundantly, so this limitation stands as described.

There is a second half the technical case cannot supply. Centralizing authority is defensible only if the party holding it is not one of the competing participants. A registry run by the largest hosting provider is a very different proposition from one run by a body that all members govern and none controls. We describe that body in Chapter 7, and the argument here depends on it.

It is important to note what the outage does not affect. Virtual machines keep running. Traffic inside a zone is unaffected. Traffic between zones over links that already exist keeps flowing, because those links live in the tunnels and not in the hub. Each zone continues to reconcile what it already knows. Losing the hub therefore loses changes and not service. This assessment is reasoned from the structure of the code and not measured, because we did not run that experiment.

5.5 The Zone Admission Contract

A partner has to know what their cluster must provide before they build anything, so we state the requirement as a condition for joining and not as documentation to be discovered later. It is not a new kind of obligation: every zone already exposes a Kubernetes endpoint the hub can reach, so this adds one more reachable address. Chapter 7 describes a membership organization, and satisfying this contract is what membership requires in technical terms.

A zone may join if it nominates a gateway node with a stable address that every other member can reach. That address must be usable as the tunnel endpoint, and there are two ways to satisfy this. Either the address is configured on a local interface, which works with any number of nodes. Or it is a stable one-to-one translation such as a floating IP, and the zone then runs a single chassis. The second route also needs node-local address rewriting, which Chapter 6 describes.

The second route has a consequence that reaches beyond the zone that takes it. Rewriting the outer addresses invalidates the outer checksum, so the checksum has to be disabled on every member of the federation and not only on the zone that rewrites.

Section 6.6 explains the mechanism. A zone with a perfectly routable address does not get to keep the checksum if one of its peers does not have one.

Beyond the address, a zone must open User Datagram Protocol (UDP) port 6081 in both directions towards every other gateway. It must also accept the agreed packet size and use address ranges that do not overlap with any existing member. Overlapping ranges do not produce an error. They produce routes that resolve to the wrong cluster, which is worse.

This is a requirement and not a limitation. Admitting a member with no reachable address would mean relaying its traffic through somewhere that has one, and a relay puts every tenant packet through the jurisdiction of a third party. Refusing such members is what keeps the federation free of a privileged middle.

5.6 Where a Workaround Belongs

During the work we accumulated patches applied by hand to the running clusters, and had to decide which were temporary and which were defects belonging in the source. The answer differed for each, and we got one wrong.

A compensation can live in four places, and they are not equally durable. Table 5.2 orders them by what silently undoes them. The first tier is the dangerous one: a setting applied at runtime survives until something restarts the component that writes it, and nothing records that it was ever there. Our checksum setting was exactly that, and a restart of the network agent puts it back to the value that breaks the fabric.

Where the fix lives	Example	Undone by	Checked by
Runtime state on a node	the checksum setting	a pod restart	nothing
A patch applied when vendoring	the chart indentation defect	resolving dependencies	nothing
A value in our own chart	the missing permissions	nothing	rendering the chart
The source of the operator	the generated permissions	nothing	the build

Table 5.2: Where a compensation for a defect can live, ordered by what silently undoes it

The rule we now follow is to move a compensation down this table until it reaches the lowest tier that can express it.

A workaround belongs in a patch when it compensates for one deployment: the tunnel address rewriting is specific to providers that translate the public address, so it stays optional and off by default. It belongs in the chart when the chart is wrong, as with the missing permission on namespaces. It belongs in the code when the chart is right and the software asks for something it should not, which is the case we got wrong.

Our tenant controller generated Roles that named the kinds which decide who holds which role. Those kinds never leave the hub by design. The zone therefore could not grant them, and the write failed. Our first fix widened the permissions of the zone until the write succeeded. That fix works and it is the wrong direction. It hands a cluster owned by a partner authority over the objects that control access for every tenant, in a system whose entire purpose is to keep such authority in one place.

The correct fix is smaller: the zone stops asking for kinds it does not host. The set of kinds a zone mirrors is already defined in one place, and the generator consults that list, so the chart and the software stay equal by construction. A test compares the two and fails the build if they drift in either direction, because our own wrong fix would otherwise have passed.

5.7 Verifying a Federated Data Plane

The last decision concerns what counts as evidence that the system works.

Two separate defects in this project passed a clean test with ping. Both times the routes were exchanged and the clusters reported themselves healthy. Both times 20 ping packets at the full packet size crossed between the two countries without a single loss. Both times no application traffic moved. In the second case the connection was even established successfully before the transfer stopped.

Our conclusion is that ping is not a test of a tunnel. It proves that a route exists and that the path can carry a packet of a given size. It proves nothing about whether the fabric can carry a conversation. An acceptance test for a zone must therefore transfer a payload over Transmission Control Protocol (TCP) and check the result, and it must do so in both directions. This is now the check we run, and the ping is kept only because it separates a routing problem from a transport problem when something does fail.

The wider lesson applies to the control plane as well. Both access control defects in Chapter 6 produced a cluster that reported itself healthy while doing nothing. A status field that is written by the component whose failure it is meant to report is worth

very little. What we would add if we continued this work is a condition on the zone record. It should reflect whether the peer is reachable at the level of the data plane, and not only whether its Kubernetes endpoint answers.

6 Experiments

This chapter reports what happened when we installed the platform on real clusters at two providers. Every defect described here was found on a running deployment that is still live. The order follows the order in which the problems appeared, because each one became visible only once the previous one was solved.

6.1 Test Environment

The deployment runs on three real Kubernetes clusters at two providers in two countries. We deliberately avoided a laboratory setup on one machine, because every defect in this report is invisible when all nodes share one network. The two providers differ in one specific way. One offers a public address only as a translation on a router, the other places it directly on the interface. That asymmetry let us test the same fabric with and without address translation on one side.

The hub runs at SWITCH Engines[1] with Cilium as its Container Network Interface (CNI). It holds the tenant definitions and the interconnect database. It carries no tenant workload of its own.

The first zone, az-switch, also runs at SWITCH Engines. It is an OpenStack-based private cloud with a hosted control plane, and the instance holds a private address and reaches the internet through a floating IP. This is the zone with the tunnel endpoint problem below, and it is reduced to a single node for the reason given there.

The second zone, az-hetzner, runs k3s at Hetzner[2] in Germany with two nodes. Its gateway node holds a routable public address directly on its interface, so it needs none of the workarounds of the first zone.

Both zones run Kube-OVN in version 1.16.2 with KubeVirt for virtual machines. The packet size on the tenant network is set to 1400 bytes on both sides. The default of 1500 does not survive a Geneve tunnel over the public internet, and a mismatch between the two zones would produce a fabric that works in one direction only.

[1] `https://www.switch.ch/engines/`
[2] `https://www.hetzner.com/`

6.2 Installing the Stack

Installation is a single Helm release per zone and completes without intervention on a cluster that behaves as the documentation expects. Neither of ours did. Both problems share a cause: each piece of software assumes a property that a cluster we do not own may not have.

The first problem concerns the virtualization layer. KubeVirt installs an operator that requires a control-plane node, expressed as a hard node affinity, so the scheduler will not place the pod anywhere else. A managed Kubernetes offering frequently hosts the control plane in the provider's own infrastructure, where it never appears among the nodes of the cluster. Every node we can see reports no role. The operator stayed pending forever and KubeVirt never finished installing. The same chart installs unchanged on the Hetzner zone, which runs its own control plane, so only one of our zones showed the problem.

The second problem is caused by the isolation our own design provides. Cloning a disk image has two paths. The efficient one asks the storage driver to copy the volume. The fallback starts a helper pod inside the tenant network and copies the data through it. A tenant network has no route to anything outside itself, so the fallback cannot work by construction. We pin the storage profile to the driver-based path, and we verified afterwards that this is load-bearing and not a precaution. Only the class we pinned reports the efficient strategy, while three sibling classes on the identical driver still report the fallback.

6.3 Establishing Cross-Zone Connectivity

The first experiment asked whether a pod in Switzerland can reach a pod in Germany over the tenant network. We started from a fabric that reported itself as fully connected and moved no packets.

After placing the floating IP on a local interface and adding the address rewriting, traffic crossed in both directions. A round trip between the two countries takes between 11 and 17 milliseconds, consistent with the geographic distance. We verified the packet size separately, because it is the one property the fabric will not report. A ping with a payload of 1372 bytes and the do-not-fragment bit set crosses, and one with 1400 bytes fails as it should. The second result is not a fault. A payload of 1400 bytes with the headers is 1428 bytes on the wire, larger than the tunnel can carry.

At this point every signal said the system worked: both zones reachable, routes exchanged in both directions, heartbeats arriving, ping lossless at the full packet size. This was wrong in two separate ways, which the next two sections describe.

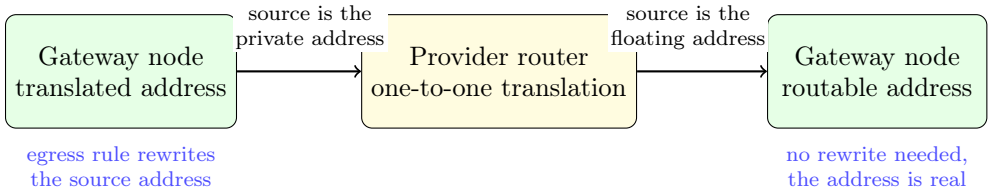


Figure 6.1: The outer source address on each hop between the two gateway nodes

6.4 Reaching a Tunnel Endpoint Behind Address Translation

Two zones exchange tenant traffic through a Geneve tunnel, so each side must publish an address the other can send to. Kube-OVN takes that address from a local interface and writes it into the interconnect database as `ovn-encap-ip`. This is the assumption that broke first.

This assumption does not hold on every provider. Several offer a public address only as a one-to-one translation on a router, which the instance never sees. A floating address in a managed private cloud is the common case, and it was ours. The node advertised its private address and the peer had no route to it. Nothing reported a problem. The clusters registered, the routes were exchanged, and no packet ever arrived.

The obvious idea is to put the floating IP on a dummy interface and point Kube-OVN at it. That part works. The second node does not. We measured this instead of assuming it. A control ping to the private address of the node was visible on the wire, and a test ping to the floating address produced nothing. The tenant network does not deliver a frame addressed to a floating IP.

A chassis has one encap address and uses it for both paths. The same address carries traffic to the peer zone across the internet and to the neighboring node inside the cluster. If it is reachable only from outside, the second path breaks. We reduced that zone to a single node, which removes the conflict without solving it.

Figure 6.1 shows the packet on each hop once the workaround is in place. The node sends from the floating address, an egress rule rewrites the source to the address the instance really holds, and the provider router translates it back on the way out. Inbound the same happens in reverse.

It is important to note that the provider's packet filter must also allow inbound UDP on port 6081, since the default group permits Internet Control Message Protocol (ICMP) and not arbitrary UDP. Without that rule the fabric registers, exchanges routes and forwards in one direction only. The rule lives outside Kubernetes and travels with none of our manifests.

Condition	Time to establish a connection
With stateful address translation	35.8 s
Immediately after flushing the tracking table	0.030 s
With stateless rewriting in place	0.023 s

Table 6.1: Connection setup time across zones before and after removing Geneve from connection tracking

6.5 Stateful Address Translation and the Geneve Protocol

The first version of the rewriting used the `nat` table of iptables. Packets arrived, but every new connection between the zones took about half a minute to establish.

The reason is a property of Geneve that is easy to overlook. A Geneve tunnel does not use one flow for both directions. The source port of the outer UDP header is derived from a hash of the inner packet, so A to B and B to A are independent flows with different port numbers. Connection tracking cannot pair them.

iptables address translation is stateful by construction, so every Geneve packet entered the connection tracking table. Every entry stayed in the unreplied state, because the reply never matched. The default timeout for unreplied UDP entries is 30 seconds, and that number appeared directly in our measurements.

We inspected the tracking table on the gateway and found every Geneve flow unreplied. To confirm that the table was the cause and not a symptom, we flushed the relevant entries and repeated the measurement immediately. Table 6.1 shows the result.

The difference is more than three orders of magnitude. The measured 35.8 seconds also matches the default timeout of 30 seconds for unreplied entries, and that match is what identified the mechanism.

The fix is to rewrite the addresses without tracking anything, which nftables can do and iptables cannot. The two hooks below run at raw priority, before connection tracking, and `notrack` keeps Geneve out of the table completely.

After replacing the rules with the stateless version, the number of Geneve entries in the tracking table is zero and stays zero.

This was the first defect that a ping did not reveal. The tunnel carried ICMP perfectly the whole time, because a ping is a short exchange that does not care about half a minute of setup cost.

```
table ip ovnfip {
  chain prerouting {
    type filter hook prerouting priority raw;
    udp dport 6081 ip daddr $FIXED ip daddr set $FIP notrack
  }
  chain output {
    type filter hook output priority raw;
    udp dport 6081 notrack
  }
  chain postrouting {
    type filter hook postrouting priority mangle;
    udp dport 6081 ip saddr $FIP ip saddr set $FIXED
  }
}
```

Listing 6.1: Stateless address rewriting for the Geneve tunnel

6.6 A Defect That Ping Cannot See

Removing Geneve from connection tracking took away the delay. It did not take away the real fault underneath it.

Cross-zone ICMP was flawless: 20 packets at the full path Maximum Transmission Unit (MTU) of 1400 bytes with the do-not-fragment bit set, no loss. A cross-zone TCP connection established in about 10 milliseconds, then an Hypertext Transfer Protocol (HTTP) request over it timed out with zero bytes transferred. Intra-zone traffic was never affected.

This shape is the classic signature of a path MTU problem, and that is where we spent the first hours. The evidence against it was already in our hands. A ping at the full MTU with the do-not-fragment bit set cannot succeed if the path cannot carry a packet of that size.

One practical detail cost us two false conclusions. The capture tool buffers its output when it does not write to a terminal, and a remote run stopped by a timeout is killed before the buffer is written. Twice we read the empty result as proof that no packet had arrived, when the packets had arrived and the evidence died with the process. Every capture in this work therefore forces line buffering.

The turning point was a capture taken at both ends of the same connection at the same time, matched on the inner sequence numbers. Single-sided captures had produced two contradictory readings. Figure 6.2 shows what we saw. The first packet of the connection arrived. The second, sent 23 milliseconds later on an identical outer five-tuple, never did.

The two packets were identical in every field a device on the path could act on: same source, same destination, same outer source port, same time to live, same do-not-fragment bit, same Geneve options, same tunnel identifier, and a valid inner

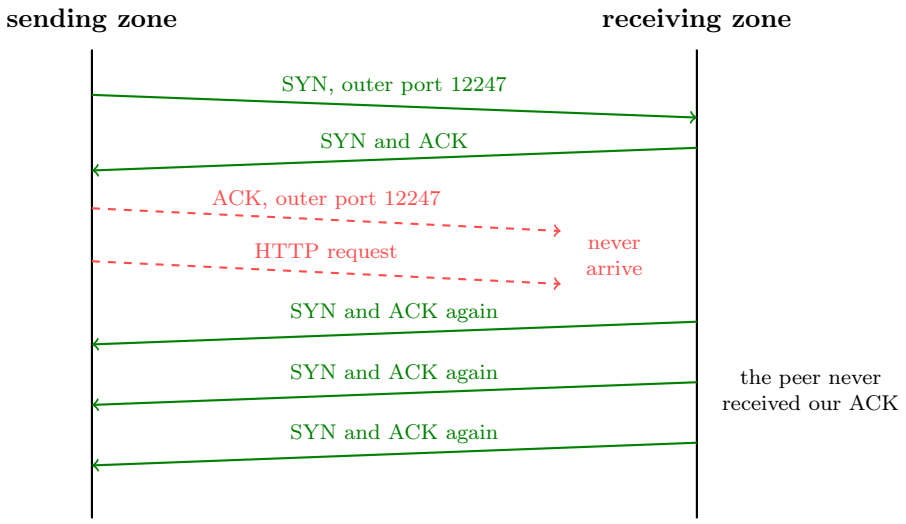


Figure 6.2: The same connection captured at both ends. The SYN arrives and the ACK on the identical outer five-tuple does not

checksum. We then eliminated the causes one at a time. Table 6.2 lists the 11 hypotheses with the measurement that ruled each one out.

One entry deserves comment. We built a syntactically valid Geneve packet in software, with the correct version, the correct tunnel identifier and a real option, wrapping a synthetic inner connection. It crossed the path without loss, which ruled out any device that understands the protocol. A packet we built by hand survived where one built by Open vSwitch did not, and the difference is that ours carried a correct checksum.

The cause was the outer UDP checksum. Kube-OVN enables it by default, so each tunnel port carries `csum=true` and every Geneve packet leaves with a computed

Hypothesis	**Test**	**Result**
Packet size on the path	Ping at 1372 bytes, do-not-fragment	no loss
The underlay drops packets	20 probes on a fixed flow	20 of 20 arrived
New flows are dropped	30 probes on 30 different ports	30 of 30 arrived
The rewriting drops packets	15 probes through it, 15 beside it	both complete
Something inspects Geneve	Hand-built valid Geneve with inner TCP	10 of 10 arrived
Local send errors	Queue and interface counters	zero drops
Node placement	Target moved to the gateway node	still failed
Specific to one tenant	Same test in the second tenant network	failed the same way
Overlapping address ranges	Route tables on every logical router	correct and separate
One direction only	Connection started from each side	failed from both
The outer checksum	Disabled on both zones	**everything works**

Table 6.2: Hypotheses eliminated during the transport investigation, with the measurement that ruled each one out

```
ovs-vsctl set open_vswitch . external_ids:ovn-encap-csum=false
```

Listing 6.2: Disabling the outer checksum on a chassis

checksum. Our workaround rewrites the outer source address outbound and the outer destination address inbound. Both are part of the UDP pseudo-header, so the checksum has to be recomputed, and the result was not correct for every packet. The provider router translates this flow, and connection tracking discards UDP with a bad checksum as invalid instead of forwarding it.

Whether a packet survived depended on the inner protocol. The outer checksum is either handled by the network card or computed in software, and which one happens depends on the state of the packet when it reaches the tunnel. That state differs for an ICMP echo from busybox and for TCP from a pod interface. It is important to note that we did not instrument this path directly. What we measured is that turning the checksum off repairs every case immediately and repeatably.

The fix is a single setting. A zero UDP checksum is allowed for IPv4 and it is what Open vSwitch uses for Geneve by default. With no checksum present there is nothing for the address rewriting to invalidate.

The setting has to be applied on both zones, and the reason is not obvious. An Open vSwitch tunnel port takes its checksum option from the record of the remote chassis and not from its own. The setting on the German zone governs what the Swiss zone sends, and the other way round. Our workaround rewrites addresses in both directions, so both need it.

6.7 A Tenant Subnet That Cuts the Federation

This failure has the widest blast radius of any we met, and it needs no privilege to trigger.

A project editor creates subnets, an ordinary namespaced write that every project editor may do. If the range covers the pod network of another zone and nobody confirms the subnet for cross-zone use, the platform blocks it and the range lands on the route blacklist of that zone.

The blacklist suppresses advertising, which is its purpose. It also suppresses learning, which is not obvious from its name. The zone stops learning that range from its peer, the route on the platform path disappears, and every workload move in the federation stops. One tenant, acting inside their own namespace and within their rights, takes out a service every other tenant depends on.

Each link in the chain is correct on its own. The subnet is legal, blocking an unconfirmed subnet is the behavior we designed, and suppressing a prefix is what a blacklist does. Nothing reports an error, which puts this defect in the same class as the ones above.

The fix rejects the range before anything is written, with a reason that names the conflict, so the tenant sees the problem where they caused it.

The direction of the overlap is what matters. Checking whether the tenant range sits inside a reserved range is not enough, because a larger tenant subnet covers a reserved one just as completely and the blacklist matches on prefix. We test the overlap in both directions, and the larger case is the one that would have gone through.

6.8 Silent Failures in the Control Plane

The network was not the only place where a failure hid behind a healthy status. Two defects in our own Helm chart stopped a zone from doing anything, and neither produced an error a normal look at the cluster would find.

The first concerns permissions on namespaces. Our resource controller reads the namespace of a project for every project scoped object, to check the labels recording which tenant and project it belongs to. The chart never granted read access on namespaces. Because the controller uses a cached client, the missing permission never surfaces as a rejected request. The informer never finishes its initial synchronization and the worker waits forever. There is no error, no event and no status condition. The zone reports itself healthy and materializes nothing.

The second concerns the way Kubernetes prevents privilege escalation. Our tenant controller creates a Role for every tenant role, generated from a single place in the code, and every generated rule set begins with a rule on events at read verbs. The chart granted the controller only create and patch on events. Kubernetes refuses to let anybody create a Role granting a permission the creator does not hold, so the first write failed. The tenant never reached the ready state, and the syncer waited for resources that were already present.

Both are invisible in a local test cluster, because there the controllers run with full privileges and the escalation check has nothing to compare against. They appear the first time the software runs somewhere real. Section 5.6 returns to where such a fix belongs.

Taken together with the network defects, a pattern emerges that we consider the central finding of this work. Every failure we met reported success. Table 6.3 collects them with the check that distinguishes a working system from a broken one. Each

health signal answers a narrower question than the one asked of it. A heartbeat proves that a process runs. A learned route proves that two databases agree. Neither proves that a packet arrives.

What is reported	**What is actually true**	**Check that reveals it**
The zone is reachable	only its Kubernetes endpoint answers	transfer a payload between zones
Routes learned in both directions	no packet crosses	transfer a payload between zones
Ping crosses with no loss	no application traffic moves	transfer a payload between zones
The tenant object exists in the zone	no permission to act on it	look for the objects it should create
The controller is running	its cache never synchronized	look for the objects it should create
The operator is installed	its pod was never scheduled	read the pod state, not the object
The subnet was created	a platform route was suppressed	move a workload between zones

Table 6.3: Signals that report success while the system carries nothing

Tenant network	Direction	Before	After
First tenant	Switzerland to Germany	timeout	200
Second tenant	Switzerland to Germany	timeout	200
First tenant	Germany to Switzerland	7.0 s or timeout	200 in 0.023 s
Second tenant	Germany to Switzerland	timeout	200 in 0.026 s
Both	Ping at 1372 bytes	no loss	no loss

Table 6.4: Cross zone transfers before and after disabling the outer checksum

6.9 Results After the Fix

Disabling the outer checksum on both zones repaired every case at once. Table 6.4 shows the same tests before and after the change.

The transferred document is 896 bytes, small but a real payload and not a handshake. We repeated the request five times in sequence in both tenant networks without a failure. The number of tracked Geneve flows remains zero and the ping results are unchanged, so the fix did not trade one problem for another.

A tenant workload in Switzerland now reaches a workload in Germany over the tenant's own private addresses, in both directions, across two providers and two jurisdictions, with a round trip of about 11 milliseconds and a transfer completing in 23 milliseconds. The traffic is not encrypted, for the reasons in Chapter 5.

7 Governance

This chapter describes how such a federation could be governed. The architecture avoids a privileged party in the data path, but a federation of independent organizations still needs a body that keeps a registry and publishes the software. We propose an organization modeled on the regional internet registries. How money moves through the federation is the part we leave open.

7.1 A Neutral Organization at the Center

The internet already solved a similar problem. Address space and routing policy are coordinated by regional registries such as the RIPE Network Coordination Centre [11], which are membership organizations and not commercial providers. They carry no traffic and operate no member's network. They keep a registry that everybody agrees to use, and their members govern them. The problem here has the same shape.

The organization operates the hub and the peering database, which records which zones exist, which tenants named which zone, and how the tenant networks are joined. No tenant traffic passes through it, as Chapter 3 describes, so it is a registry and not a transit provider. A member that leaves takes its hardware and its customers with it.

It also publishes and maintains the software. A federation in which every member ran a slightly different version of the interconnect would spend its time debugging version skew, and our experience in Chapter 4 suggests those failures would be silent ones. A single release process removes that class of problem.

It is also the natural place to anchor trust between members. Chapter 5 describes why encrypting the interconnect is blocked on a question no single cluster can answer. Each cluster signs its own certificates, so two members that have never exchanged a root cannot authenticate each other. A body that all members govern can hold that root, which turns an unsolved technical problem into a membership procedure. Whether it also handles the money is a separate question, set out in Section 7.4.

7.2 Membership and Funding

The software is free for end customers. A tenant pays only for the capacity it uses.

Hosting providers pay a fixed annual fee, scaled by company size and not by revenue or contributed capacity, up to 100,000 euros per year for the largest members. A fee proportional to capacity would penalize the members whose hardware makes the federation useful, and one proportional to revenue would require the organization to audit its own members.

The fee buys the software and a seat, which means a vote in how the organization is run and in what the stack does next. A provider that pays is not a customer but an owner.

The ceiling matters as much as the scale. Without it the largest member would pay the most and expect influence to match, which is how a neutral registry becomes an instrument of its biggest participant.

7.3 Obligations of a Zone Operator

Membership carries one obligation. A hosting provider must open its zones to tenants that wish to deploy there. A provider may not take the software and the seat and then run a zone that serves only itself, because that takes the benefits of the federation without contributing what creates them.

Every operator may still choose which tenants it accepts, for legal reasons, for commercial reasons or because the workload does not suit its hardware. The obligation is that the zone is open, and not that it is unconditional.

One question we do not answer is how to tell the two kinds of participant apart. An end customer that contributes its own hardware may legitimately declare itself the only tenant on its own servers, because it joined to keep its workloads under its own control. A hosting provider may not. Nothing prevents a company from presenting itself as an end customer to avoid the obligation while benefiting from a federation that others keep open. Where that line falls is for the members to decide.

7.4 Pricing and Settlement

Pricing is decentralized and each provider sets its own. The organization publishes no rate, approves none and operates no marketplace that clears at a single price. Competition between members is intended, because a federation whose prices were

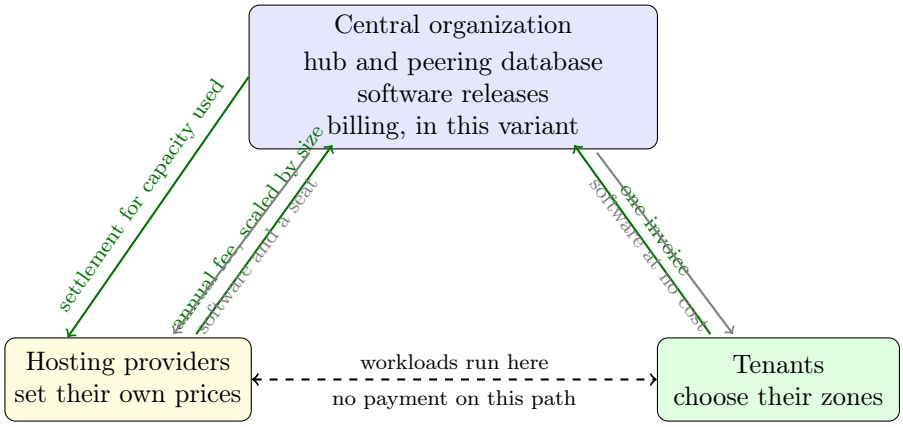


Figure 7.1: The centralized billing variant. Prices are still set by each provider, and only the invoice is aggregated

set centrally would be a cartel with a registry attached. How the money is collected is a question we leave open, and two arrangements work with decentralized prices.

7.4.1 Billing Through the Organization

The tenant receives one bill. The organization aggregates what the tenant consumed across every zone it used, issues a single invoice and settles with each provider afterwards. Figure 7.1 shows it.

The argument is friction. A tenant using five zones in four countries should not have to become the customer of five companies, and handling five contracts and five currencies would remove much of the benefit for the smaller organizations the federation is meant to serve.

It costs two things. The organization carries the payment risk, because it pays members for capacity whether or not the tenant pays, and a real deployment would need reserves and credit limits that we have not designed. It also holds a record of how much each tenant consumed in every zone, which is a detailed picture of the federation concentrated in one place.

7.4.2 Billing by Each Zone Operator

Each provider bills its own customers directly and no money passes through the organization. This is closer to the model we took the organization from, because a regional internet registry does not invoice anybody for traffic, and that is part of why it is straightforward to trust.

It removes both costs at once. There is no payment risk, and the center never learns what a tenant consumed, which weakens the concern we raise in Section 7.5. The registry still records which tenant named which zone, because the platform cannot function otherwise, but the commercial detail stays between the two parties that are trading.

What it costs is the friction the first arrangement removes. A tenant spanning several providers deals with several contracts and possibly several currencies, and each provider assesses its creditworthiness separately. For a large tenant that is normal business. For the small organization this federation is meant to serve, it may be the difference between using several zones and using one.

7.4.3 Which One to Choose

We do not decide, because the answer depends on facts a report cannot supply. A federation of five providers serving large customers would probably not need a clearing house. One of 50 serving small ones almost certainly would. A third route is billing as an opt-in service, which keeps the friction argument and limits what the center learns. We have not worked out whether the added machinery is worth it.

7.5 Open Questions

Several governance questions remain, and we list them without answering them.

The first is the one this report invites most directly. Chapter 3 argues that no tenant packet may cross a central point, because that point would become a jurisdiction of its own. The arrangement here keeps that promise for traffic and not for everything else. The organization holds the registry of which tenant placed what in which zone, and if it also issues the invoices it gains a record of consumption more complete than any single member has. Where the organization is incorporated therefore becomes a question about the sovereignty claim and not only about company law. This is one reason we left the billing arrangement open in Section 7.4. Direct billing removes the commercial half of that record but not the registry itself, and distributing the registry is what Chapter 5 argues against on other grounds.

Liability is the second. When a workload fails on hardware owned by another member, it is not obvious who owes the tenant what, nor whether the organization is a party to that claim. A registry that also issues the invoice looks much more like a contractual counterparty.

Service levels are the third. Members contribute hardware of different quality, and a tenant that chose a zone for its jurisdiction and not for its reliability still has an

expectation. Whether the organization certifies members, publishes measurements or leaves the question to the bilateral relationship is unresolved.

The last is exit. A member that leaves takes its hardware with it, and the tenants running there need somewhere to go. The technical mechanism exists, because a tenant can name a different zone and the workload materializes there. The commercial and legal mechanism does not, and a federation that is easy to join and difficult to leave would not deserve much trust.

8 Conclusion

This work set out to build a federation of Kubernetes clusters owned by different organizations, and to find out what it can guarantee about placement, movement and isolation. We built it and ran it on real clusters at two providers in two countries, and it is still running. This chapter answers the two questions the project was set and describes what we would do next.

8.1 Results

8.1.1 Virtualization Abstraction and Interregional Migration

The level of virtualization abstraction does not affect whether a workload can be reached across zones. Containers and virtual machines attach to the same subnets and take addresses from the same range, so a workload in Switzerland reaches one in Germany the same way in either form. The tenant sees one network and need not know which form is on the other side.

The abstraction level decides what has to travel with the workload. A container that moves to another zone is started again there. A virtual machine holds state that would have to arrive with it, and this is where the two forms stop being equivalent.

Moving a workload without its state works in both forms. A tenant names a different zone and the workload appears there, in the order described in Chapter 3: the holding zone keeps it until the target reports that it has taken over, and only then does the source tear down. A zone name matching nothing leaves the workload where it is. We reason this from the structure of the code and did not run the experiment, because the deployment was occupied by the network work in Chapter 6.

Live migration is a different matter, and the boundary is storage. A machine that migrates while it runs needs its disk to follow it, and every zone in this federation keeps its persistent storage local. We do not federate volumes and we did not test live migration between zones. That is a scope decision taken at the start and stated in Chapter 1, not a result of the work.

The second half of the question asks what a stateful instance requires of the synchronicity of storage and network. We can answer the network half with measurements. The storage half we cannot answer. Across zones a tenant network is one address space, a round trip takes between 11 and 17 milliseconds, the usable packet size is 1400 bytes, and the traffic is unencrypted. Any storage replication that followed a running machine across this federation would have to work inside those numbers. Whether it can is the question we leave open.

8.1.2 Software Defined Networking and Multi-Tenant Isolation

Separation between tenants is structural. A tenant network is its own logical router with its own address range, so reaching another tenant is not a filter decision that could be set wrongly. There is no route. The same holds across zones, where the tunnel identifier keeps two tenants apart even though one tunnel between two gateways carries both.

Placement is always named. A zone sees only the tenants that chose it, a tenant naming no zone exists nowhere, and a subnet becomes reachable from another zone only after an administrator confirms it as a separate object. A tenant network reaches no public address, because it has no gateway to one.

The Kubernetes layer separates the same way. A project maps to a namespace, a tenant deploys into its own namespaces and into no others, and the roles that allow this are generated from one list that a test compares against the chart on every build.

Several of these are enforced and not documented. The zone admission contract in Section 5.5 is a condition of joining. The role test fails the build in both directions and caught our own wrong fix in Section 5.6. The chart refuses to render when a zone picks a transit range colliding with the platform, and a test asserts the response header of the web interface.

The limit is the more useful result. The isolation above is work we had to do, and not a property the network layer handed us. Section 6.7 reports a tenant action that needed no privilege and stopped workload movement for every other tenant in the federation. We found it in operation and added a guard. A software defined network makes isolation of this kind expressible, not automatic, and the difference is where the effort of this project went.

What the architecture does not deliver is confidentiality. Traffic between zones crosses the public internet in cleartext and carries no authentication, so a forged packet is accepted like a real one. The database that holds the topology is reachable in the same condition. Write access to the cross-zone routing of every member sits behind a firewall rule instead of a credential. Encryption covers one of the two paths

between zones, and not the one tenants use. The mechanism that would fix this ships with the network layer and is blocked on a trust root that no single cluster can hold, which Chapter 7 treats as a question of membership.

The answer to the question as asked is therefore split. Separation and placement can be enforced across organizations that do not administer each other, and we enforce them. Confidentiality and authenticity of the traffic cannot be, in the state we leave the system. Sovereignty over where a workload runs is achieved. Sovereignty over the traffic while it is in transit is not.

8.1.3 Additional Findings

Every failure we met reported success. Table 6.3 collects them. Four defects in the network and two in the control plane produced a federation that registered, exchanged routes in both directions, reported every component healthy and carried nothing.

Ping is not a test of a tunnel. Two separate defects passed 20 packets at the full packet size without a single loss while no application traffic moved. A ping proves that a route exists and that the path carries a packet of that size. It proves nothing about a conversation.

The common cause is that each signal answers a narrower question than the one being asked of it. A heartbeat proves that a process runs. A learned route proves that two databases agree. An acceptance test for a zone has to transfer a payload over TCP in both directions and check the result, and that is the check we run now.

8.2 Future Work

Encrypting the interconnect is the first thing we would do. The mechanism exists in the network layer, and the missing piece is a trust root held by the body in Chapter 7, which turns an unsolved technical problem into a membership procedure. The topology database needs the same treatment, so that a credential and not a firewall rule protects it.

Federating storage would close the half of the first question we left open. It is also what makes live migration between zones testable, so the two follow each other.

The hub runs as a single instance. Losing it loses changes and not service, but a federation members depend on needs it deployed redundantly, and we would measure the outage behavior instead of reasoning about it.

One of our zones is limited to a single node, because a translated address cannot serve as a tunnel endpoint inside the cluster and towards a peer at once. Lifting that would let a zone behind address translation contribute more than one machine.

The zone record should carry a condition reflecting the data plane and not only the Kubernetes endpoint. This report argues throughout that the two are independent, and the record currently reports the one that is easier to observe.

Finally, the federation we operated had two zones. Table 3.2 estimates what happens at 50 and at 200, but those numbers are arithmetic and not measurement. A larger deployment would show whether the peering database and the tunnel mesh behave as estimated.

A Acronyms

CNI Container Network Interface

CRD Custom Resource Definition

HTTP Hypertext Transfer Protocol

ICMP Internet Control Message Protocol

MTU Maximum Transmission Unit

OVN Open Virtual Network

TCP Transmission Control Protocol

UDP User Datagram Protocol

B Appendix

B.1 Permissions Required by a Zone

The two rules below are the ones that were missing from our chart and that caused the silent failures described in Section 6.8. The first one is read access on namespaces for the resource controller. The second one widens the verbs on events for the tenant controller, so that it can create the Roles it generates for each tenant.

B.2 Reconciling the Tunnel Endpoint

The workaround from Section 6.4 runs on the gateway node and repeats every 30 seconds, so that it survives a replacement of the node. The listing below is the core of that loop with the error handling removed. The guard in the first branch is the important part. If the zone ever holds more than one node, the workaround removes itself instead of leaving the traffic inside the cluster broken.

B.3 Zone Configuration for a Translated Tunnel Endpoint

A zone that reaches the interconnect through address translation needs the two settings below. Both have to match on every member of the federation, and not only on the zone that performs the rewriting, for the reason given in Section 6.6.

```
- apiGroups: [""]
  resources: ["namespaces"]
  verbs: ["get", "list", "watch"]

- apiGroups: [""]
  resources: ["events"]
  verbs: ["get", "list", "watch", "create", "patch"]
```

Listing B.1: The two rules a zone controller was missing

```
while true; do
  FIP="$(node external address)"
  COUNT="$(number of nodes in the cluster)"

  if [ "$COUNT" != "1" ]; then
    # A translated address is not deliverable inside the tenant network,
    # so a second chassis cannot use it. Revert rather than break the
    # traffic inside the cluster.
    teardown
  elif [ -z "$FIP" ]; then
    log "node has no external address yet, waiting"
  else
    ip link add ovnfip type dummy
    ip link set ovnfip up
    ip addr add "$FIP/32" dev ovnfip

    nft -f - <<NFT
    ... the stateless ruleset from \autoref{lst:nft} ...
NFT

    # The annotation comes last. The rewriting rules are only correct
    # once the tunnel already uses the translated address. Applied in
    # the other order they rewrite every incoming packet, including
    # the ones from the neighboring node.
    annotate node ovn.kubernetes.io/tunnel_interface=ovnfip
  fi
  sleep 30
done
```

Listing B.2: The reconcile loop for the tunnel endpoint

```
kubeovn:
  networking:
    pods:
      mtu: 1400
    encapChecksum: false

natTunnelEndpoint:
  enabled: true
```

Listing B.3: Zone settings for a translated tunnel endpoint

List of Figures

List of Tables

List of Algorithms

List of Listings

Bibliography

[1] Admiralty Technologies. Admiralty: Multi-cluster scheduling for kubernetes. https://admiralty.io/.

[2] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *ACM Queue*, 14(1):70–93, 2016.

[3] Jason A. Donenfeld. WireGuard: Next generation kernel network tunnel, 2017. https://www.wireguard.com/papers/wireguard.pdf.

[4] European Parliament and Council of the European Union. Regulation (eu) 2016/679 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data (general data protection regulation). Official Journal of the European Union, L 119, 2016. https://eur-lex.europa.eu/eli/reg/2016/679/oj.

[5] Gaia-X European Association for Data and Cloud AISBL. Gaia-x architecture document, 2024. https://gaia-x.eu/.

[6] Jesse Gross, Ilango Ganga, and T. Sridhar. Geneve: Generic network virtualization encapsulation. RFC 8926, Internet Engineering Task Force, 2020. https://www.rfc-editor.org/rfc/rfc8926.

[7] Marco Iorio, Fulvio Risso, Alex Palesandro, Leonardo Camiciotti, and Antonio Manzalini. Computing without borders: The way towards liquid computing. *IEEE Transactions on Cloud Computing*, 11(3):2820–2838, 2023.

[8] Kubernetes Network Plumbing Working Group. Multus CNI: A CNI meta-plugin for multiple network interfaces. https://github.com/k8snetworkplumbingwg/multus-cni.

[9] Kubernetes SIG Multicluster. KubeFed: Kubernetes cluster federation. https://github.com/kubernetes-retired/kubefed.

[10] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J. Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martín Casado. The design and implementation of Open vSwitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 117–130, 2015.

[11] RIPE Network Coordination Centre. RIPE NCC: Regional internet registry for europe, the middle east and parts of central asia. https://www.ripe.net/.

[12] The Kube-OVN Authors. Kube-OVN: A Kubernetes network fabric based on OVN. https://www.kube-ovn.io/.

[13] The Kubernetes Authors. Kubernetes: Production-grade container orchestration. https://kubernetes.io/.

[14] The KubeVirt Authors. KubeVirt: Virtual machine management on Kubernetes. https://kubevirt.io/.

[15] The Open vSwitch Project. OVN: Open virtual network. https://docs.ovn.org/.

[16] The Submariner Project. Submariner: Cross-cluster L3 connectivity for kubernetes. https://submariner.io/.

[17] United States Congress. Clarifying lawful overseas use of data act (cloud act), h.r. 4943, 115th congress, 2018. https://www.congress.gov/bill/115th-congress/house-bill/4943.