Invisinets: Removing Networking from Cloud Networks

Sarah McClure[⋆]*, Zeke Medley*[⋆]*, Deepak Bansal*[∗]*, Karthick Jayaraman*[∗]*, Ashok Narayanan*[†]*,*
Jitendra Padhye[∗]*, Sylvia Ratnasamy*[⋆†]*, Anees Shaikh*[†]*, and Rishabh Tewari*[∗]
[⋆]*UC Berkeley* [†]*Google* [∗]*Microsoft*

Abstract

Cloud tenant networks are complex to provision, configure, and manage. Tenants must figure out how to assemble, configure, test, *etc.* a large set of low-level building blocks in order to achieve their high-level goals. As these networks are increasingly spanning multiple clouds and on-premises infrastructure, the complexity scales poorly. We argue that the current cloud abstractions place an unnecessary burden on the tenant to become a seasoned network operator. We thus propose an alternative interface to the cloud provider's network resources in which a tenant's connectivity needs are reduced to a set of parameters associated with compute endpoints. Our API removes the tenant networking layer of cloud deployments altogether, placing its former duties primarily upon the cloud provider. We demonstrate that this API reduces the complexity experienced by tenants by 80-90% while maintaining a scalable and secure architecture. We provide a prototype of the underlying infrastructure changes necessary to support new functionality introduced by our interface and implement our API on top of current cloud APIs.

1 Introduction

Almost all large cloud customers use multiple cloud providers to improve reliability and avoid provider lock-in [57]. Unfortunately, splitting a workload among large cloud providers is not as seamless as it should be. One major problem is that today's tenant networking abstractions are essentially virtualized versions of the low-level building blocks used to build physical networks and hence customers are required to craft complex topologies using subnets, virtual links, gateways, a myriad of virtual appliances, *etc.*

While an individual virtual network or simple deployment may not be overly complex, for larger tenants, reasoning about the scalability, availability, security, *etc.* of their virtual networks requires detailed knowledge and configuration of a range of networking technologies, and the problem is particularly acute when considering transit between clouds.

In response to this underlying complexity, our goal is to make multicloud architectures simple by making this inter-virtual-network transit trivial. We achieve this primarily through leveraging existing "publicly-routable but default-off" addresses for all endpoints. These addresses are publicly-routable, but the cloud provider will deny all traffic not specifically permitted by the tenant. With this choice, connectivity becomes trivial with only minor infrastructure changes.

With this connectivity assumption, we develop a new API for cloud tenants to reserve networking resources based on high-level abstractions. Ultimately, this results in a declarative approach that allows tenants to essentially associate SLOs to endpoints, without concern for *how* to achieve their networking goals. In other words, *we believe that the best way for tenants to think about networking is to not think about networking at all.*

Today's tenant networking abstractions are largely relics of the early cloud era, with a 1:1 mapping between datacenter physical network devices and cloud networking abstractions. This appealed to early cloud customers who wanted the same management experience as they shifted infrastructure from private datacenters to the cloud, but these abstractions are not fundamentally necessary nor especially accurate to what occurs in the cloud provider's network underneath. A key contribution of our paper is to show that even a modest refactoring of the existing functionality and modification of the underlying network virtualization platform can result in a far simpler higher-level API.

Such an API is beneficial to cloud providers as well as it allows them to offer their customers a seamless multicloud experience, and yet retain the ability to differentiate (*e.g.,* through performance, options, *etc.*). Furthermore, and perhaps more importantly, the API is far less complex than the low-level APIs offered today. A simpler API means fewer mistakes and thus fewer resources dedicated to customer support.

We recognize that our proposed cloud interface may not be immediately appropriate for all cloud tenants. Fortunately, the abstractions we propose can coexist with those available today so that tenants may choose the tradeoff they desire.

This paper presents the rationale (§2), design (§3-5), implementation (§6), and evaluation (§7) of a new API – which we refer to as *Invisinets* – for tenant networking. This work builds on our earlier workshop paper [41], refining and expanding the API as well as adding a complete implementation and evaluation. Our evaluation applies this API to several deployment case studies and measures several metrics to capture complexity. Accordingly, we show that Invisinets can reduce the number of network components that a tenant must interact with by ∼80%-90% in these scenarios.

2 Motivation

In this section, we discuss the status quo to motivate the need for a simpler tenant networking API. Throughout this paper, we default to Azure terminology for cloud network

components, though equivalent ones are typically available in other clouds.

2.1 Deployment Walkthrough

Consider an enterprise tenant whose workloads span multiple regions within a cloud, multiple clouds, and an on-prem deployment. To highlight the complexity that enterprise network engineers encounter, we will walk through the steps to construct such a virtual network today. At a high level, tenants must complete the following 5 steps to create a deployment.

(1) Create virtual networks. The basic construct in a tenant's network is a *virtual network* (or VPC in the terminology of AWS and GCP) which builds on the traditional concept of a subnet – a portion of the network whose hosts share a common IP address prefix. The tenant's first step, creating their virtual networks (or vnets), involves assigning IP prefixes to each vnet based on the anticipated number of instances, whether they should be IPv4 or IPv6, public or private, *etc.* These are important decisions as a particular choice (*e.g.,* IPv4 vs. IPv6) leads to a separate path down the decision tree of subsequent connectivity options. As a tenant's deployment scales, managing non-overlapping subnets across 100s of vnets becomes challenging, prompting special address planner tools [9]. Beyond address management, defining a vnet involves configuring a range of additional parameters such as security groups, ACLs, route tables, and individual VM interfaces. For simple deployments, this complexity can be hidden via default configurations but this approach rapidly breaks down with larger deployments.

(2) Connectivity in/out of the virtual network. Next, the tenant must define how instances within a vnet access resources outside the vnet. Options include setting up a "NAT Gateway" (for address translation), a "VPN Gateway" (for private VPN connectivity), or a "Virtual Network Gateway" (which can be configured for VPN or to terminate a direct link). AWS has even more gateway options, some simply to allow the virtual network to access the Internet [8]. Each gateway must be configured with the appropriate routing and access policies.

(3) Networking multiple virtual networks. Next, the tenant must interconnect these virtual network. Within a cloud, this generally requires a virtual network peering and installing the necessary routes, though these links often have some regional limitations. To connect virtual networks across clouds, VPN gateways or Internet access (via necessary gateways or security rules) will be necessary. Each of these connections come with their own specific configuration parameters.

(4) Specialized Connections. An increasingly common component in large tenant networks are dedicated connections that promise higher bandwidth, lower latency, and/or more consistent performance than seen on the public Internet (*e.g.,* ExpressRoute [44], Dedicated Interconnect [28], or Direct Connect [7]). These allow tenants to reserve a physical, dedicated link between their virtual network and a colocation facility. From there, the enterprise may complete the circuit to their on-prem resources or stitch the connection together with one from another cloud. Provisioning and managing these links requires low-level networking knowledge such as BGP configuration and coordination between the enterprise, cloud provider, and the colocation point. Since these dedicated connections are expensive, tenants might configure their routers to schedule higher priority or sensitive traffic over these links, while routing other traffic over the public Internet.

(5) Appliances. The above steps establish a basic topology and connectivity between the tenant's instances, but tenants also deploy a range of virtual appliances such as load balancers and firewalls. Each cloud offers both first-party and third-party appliances for many of these purposes. Even within the first-party selection, there are often multiple options for a single appliance. The tenant must select appliances, place them in their virtual topology, configure routing to steer traffic through the right appliances, and finally configure each appliance (*e.g.,* DPI rules, load-balancing rules, *etc.*).

Once the tenant has completed all of these steps, the job is not done. The tenant must continue to respond to changing requirements, application and network migrations, inevitable configuration mistakes, and outages caused by any other issue. When determining their procedure for confronting these issues, tenant network operators must keep all details from the above steps in mind, acting as a seasoned network expert.

2.2 Problems

We briefly highlight the main sources of complexity that we observe from the above walkthrough.

(1) Abstractions are too low-level. Provisioning and managing a virtual network involves many of the same steps as in a physical datacenter. Tenants are essentially given virtual versions of the low-level abstractions found in a physical network (*e.g.,* links, gateways, subnets) and must assemble these (which involves topology planning, routing policies, *etc.*) to achieve their higher-level intents for the overall deployment. Many of these abstractions require addressing configuration in particular to achieve basic connectivity between tenant applications/endpoints.

(2) Complex planning. Beyond determining the topology of the deployment, cloud marketplace options can make it difficult to determine the correct virtual appliance. For example, Azure offers four load balancing options and the flow chart to guide the decision is five layers deep [43] and this does not consider other third-party (*i.e.,* non-Microsoft) options. Cloud appliance marketplaces also feature third-party options (*e.g.,* firewalls from Palo Alto Networks) that vary in features and price points. A cottage industry of businesses offering answers on how to minimize cloud costs has appeared to help tenants with this problem in recent years [10, 17–19, 66].

(3) Fragmentation across clouds. As each cloud has its own similar yet different abstractions and appliances, tenants with multicloud deployments end up with siloed stacks for each cloud. This often results in teams dedicated to each cloud

with their own expertise, scripts, and approaches.

(4) Complex to maintain and evolve. As the requirements for their applications change, tenants will evolve their own deployments. Likewise, tenants must adapt as cloud providers evolve their offerings. This adds complexity and management overhead to the existing challenge of keeping applications modern and performant for tenants. Misconfigurations are common causes of network incidents [36] and outages ([35, 62] provide recent high-profile examples). With 1:1 abstractions, virtual networks suffer many of the same issues. The cloud providers also suffer from this as well, as they are obligated to provide support to a wide array of clients with unique deployments. Further, management complexity increases as the deployment size increases, so large enterprise tenants suffer significant management burdens.

In summary, tenant networks today are constructed from low-level building blocks that are unique to a given cloud. With the growing popularity of large, multicloud deployments, this complexity can compound and become even more difficult.

2.3 Current Solutions

In our experience, many enterprises undertake this complexity themselves using an array of per-vendor controllers and DIY scripts. This often requires a team that understands networking in all its gore: BGP, address management, VPN config, *etc.* These teams must understand multiple cloud environments, which change frequently and outside of their control.

Other tenants are turning to a new generation of 3rd-party multi-cloud management solutions [5, 12, 67, 68]. Some of these solutions are essentially a shim on top of the various cloud networking abstractions. They provide a unified "pane of glass" via which the tenant manages individual devices across clouds [12, 67] but do not fundamentally change the level of abstraction; *e.g.,* a key component in Aviatrix deployments is a transit router abstraction that interconnects virtual networks. Yet other 3rd-party solutions essentially run a virtual network as a service for tenants [5, 68], which allows tenants to completely outsource the problem. This shifts the burden but does not fundamentally solve it.

Anthos [22] integrates k8s and service meshes in a manner that frees app developers from having to reimplement common networking related tasks on a per-app basis. With Anthos, every service container is integrated with a "sidecar" container that implements common network-related tasks such as TLS termination, HTTP load balancing, tracing, and so forth. This clean separation between app and networking concerns gives app developers a cloud-agnostic (and hence multicloud friendly) approach to implementing network-related features. However, it is important to note that Anthos does not address the problem of network virtualization that we address here: in Anthos, sidecars are L7 proxies and (like all k8s services) they assume L3 addressing and connectivity has already been established. Implementing that L3 connectivity still requires a network engineer to set up the virtual networks, links, VPN gateways, *etc.* that we have been discussing [30, 33]. Thus we view the goals of Anthos and Invisinets as complementary: Anthos simplifies the construction of multicloud deployments for app developers, while Invisinets does the same for infrastructure operators.

2.4 It's Time for Simplification

Network virtualization technologies were originally designed to allow cloud providers to virtualize their *physical* network infrastructure [34, 53]. In this context, providing the user (in this case, the datacenter operator) with virtualized equivalents of their physical network is appropriate, and we do not question the approach.

Extending the same approach to cloud *tenants* also made sense in the early days of cloud adoption when enterprises with well-established on-prem datacenters often used the so-called "lift-and-shift" strategy: creating a networking structure that mimics the on-premises network that previously served the workload. This strategy was justifiably appealing as it allowed tenants to use familiar tools and tested configurations in their new cloud deployments. However, we see this approach as neither desirable nor necessary as tenants embrace the cloud more fully in both the scope of their deployments and in (re)designing applications for cloud environments.

Nonetheless, we recognize that certain enterprises may choose to continue with building virtual networks for reasons that range from satisfying compliance requirements (*e.g.,* with data protection laws [55,65]), to familiarity with existing tools, and the perception of greater security. Fortunately, this need not be an either-or decision: the architecture we propose can be deployed alongside existing solutions allowing tenants to choose whether and when to migrate their workloads.

Our approach requires new support from cloud providers, but we believe this is reasonable since the current situation is non-ideal even for cloud providers. The current complexity imposes a steep learning curve for onboarding new customers, and plenty of room for configuration errors that will, regardless of fault, result in unhappy customers. Simplification can decrease the number of tenant errors and therefore decreases the support burden on the cloud provider. Further, the cloud provider could likely achieve higher resource efficiency by taking control of networking and orchestration from tenants.

3 Approach

Our guiding philosophy in designing a simplified networking API is that *the right way for a tenant to think about the network is to not think about it at all*. I.e., ideally, the network should be invisible. When diagnosing the root cause of today's complexity, we arrive at the observation that the problem starts with the fact that tenant endpoints live in a private IP address space. Given private addresses, tenants must then establish (virtual) connectivity between them which necessarily implies managing subnets, constructing a virtual topology

with links, routers, and appliances, running routing protocols that must be configured, and so on.

Seeking to avoid this leads us to an alternate proposal: *can we give every endpoint a public IP address?* This makes connectivity trivial, notably even across clouds. Essentially, this allows virtual networks to reuse the connectivity of the underlying (physical) network rather than recreating it. Importantly, we must assume IPv6 so that address scarcity is not an issue.

At first glance, our proposal might seem concerning from the viewpoint of security: if any host on the Internet can reach any tenant endpoint, then surely we're exposing a tenant's endpoints to attack, including DDoS. Yet, our intuition was that security should not be a concern for public endpoints *that are hosted in the cloud*. This is because cloud providers (or CPs for brevity) have already addressed this problem with their own address management architectures. As we elaborate on in the next section, within a CP's infrastructure, when an endpoint is assigned a public IP address P[1], this address is not actually routable *within* the CP's regional datacenters. I.e., the endpoint associated with P is not actually hosted on a server with the address P. Instead, the server at which the endpoint runs has an internal/private address D and the CP uses solutions to translate P to D with appropriate security and access control checks at the point of translation.

Thus, instead of the typical public *vs.* private address tradeoff, P represents a new form of address that is publicly routable but default off (PRDO), with the important property that a packet destined to a PRDO address is delivered to the CP 's domain but will not be delivered to an endpoint until the CP has *explicitly taken action to associate P to a physical endpoint's D*. Thus our intuition was that we can leverage the PRDO addressing architecture to spare tenants from having to solve the connectivity problem in their virtual networks.

Given this, our next step was to verify our intuition and understand whether CP address management infrastructure could indeed be leveraged and extended to serve all tenant endpoints. To answer these questions, we engaged with two major CPs (Azure and GCP) and found that, perhaps surprisingly, our proposal could be supported with little modification to their existing infrastructure and raises no new scaling or security challenges. We elaborate on existing CP address management infrastructure and the implications of PRDO addressing in §4. Thus our contribution lies not in devising novel techniques or radical clean-slate designs but rather in proposing a radically simplified tenant abstraction and showing how we can repurpose existing infrastructure to implement this abstraction.

With PRDO addressing as our starting point, what API can we offer tenants? We observe that, for the vast majority of cloud tenants, the network is a means to an end: *i.e.,* tenants care that their application endpoints (*i.e.,* VMs or containers)

[1]By which we mean an address from the address space the CP advertises into the Internet's routing infrastructure - *e.g.,* [27] for GCP

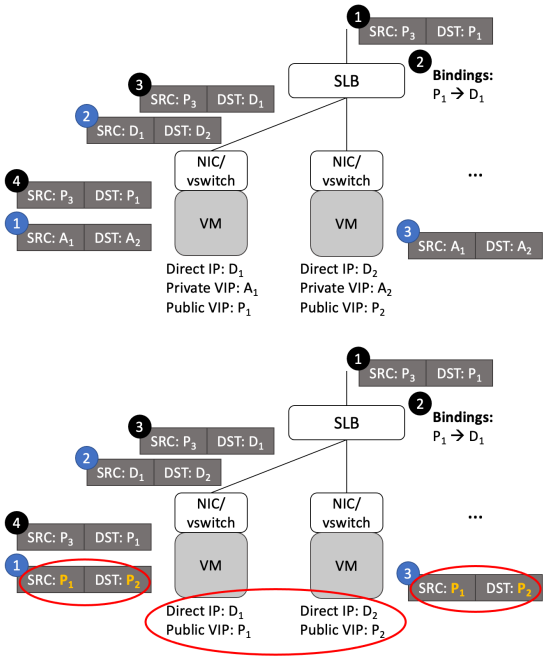


Figure 1: Cloud addressing as it is done today (top) and our proposed changes (circled, bottom). Black packets show the translation and path of packets between VMs in the same virtual network while the blue path shows an external connection using the VM's public IP.

can communicate with each other with high availability, a certain level of performance (*e.g.,* latency, throughput), security against unwanted access, and scalable mechanisms for management. These are the goals that a tenant is trying to achieve when they set up and manage a virtual network topology with firewalls, links, and routers. Hence, we design an API that allows tenants to express *what* they want the network to accomplish, rather than *how* it does so. Moreover, we note that the parameters that specify a tenant's goals are associated with how *endpoints* – VMs/containers – experience the network. Hence, our general approach is to assign PRDO addresses to all tenant endpoints and then provide tenants a high-level API that associates SLOs (for availability, security, and performance intents) with these endpoints (or groups of endpoints), thus completely eliminating today's low-level "links and boxes" abstractions.

We do not require that CPs cooperate to implement the API, nor even that all CPs adopt the API. Likewise, CPs don't have to implement identical versions of the API. Converging on a single API across all CPs would certainly be ideal but, even if each CP adopts their own flavor of the API we propose, tenants will benefit from the simplification in configuring that CP, and the high-level nature of the API will make it easier to port deployments across clouds.

4 PRDO Addressing

In this section, we first review the addressing architecture commonly implemented by CPs today (§4.1). This discussion reiterates information published before [20, 21, 54] but distills

the aspects relevant to Invisinets. We then discuss how this addressing infrastructure can be used to support Invisinets (§4.2) and the security implications of the same (§4.3).

4.1 Addressing in Today's Clouds

A cloud provider's infrastructure involves a few different address spaces. Public IP addresses (PIPs) are drawn from public prefixes that the CP advertises into the Internet at large. CPs vary in the details but, generally speaking, a packet destined to a PIP will be delivered to a specific datacenter or region in the CP's global infrastructure.

In addition, each cloud datacenter has a private address space of "direct" IP addresses (DIP) – a DIP is the actual IP address of the physical server and the basis for routing within a datacenter fabric. It is used internally to the CP and is never exposed to the tenant, providing a layer of indirection and allowing the CP to place/move VMs as needed.

Finally, at the virtualization layer, tenants see a virtual IP (VIP) for their VM or endpoint. A VIP may be public or private (a VM may have both) but crucially is not actually assigned to the server and therefore is not routable in the datacenter fabric. Instead, it is used at the tenant-layer for virtual network-level routing, *etc.* For packets sent between two VMs with private VIPs in the same virtual subnet (A_i addresses), the source and destination fields are translated in the vswitch/NIC at the VMs; the vswitch/NIC translates from the private VIP address space to the corresponding DIPs ($A_i \leftrightarrow D_i$) [25, 54] as shown in Figure 1 with the blue series of packets. The fabric only understands how to forward traffic with DIPs and is unaware of the tenant address space (giving the virtualized layer full flexibility in assigning addresses).

Endpoints with public VIPs are assigned an IP address drawn from the CP's public address space (P_i). However, this address is not directly assigned to the VM. Instead, any incoming traffic destined to P_i is first routed through a software load balancer (SLB) in the datacenter (see Figure 1 black series of packets) that maintains a binding from P_i to D_i.[2] The mapping between public VIPs and DIPs are installed when the public IP is provisioned and associated with an endpoint. Thus the SLB advertises all public VIPs under its control and translates incoming traffic to the endpoints DIP to route to the endpoint. For traffic exiting the VM, no translation is necessary as the source is the public VIP and default routes are used to exit the datacenter.

Thus in the terminology introduced earlier, all PIP addresses (including ones assigned as public VIPs) act as PRDO addresses, requiring explicit SLB and vswitch/NIC configuration before packets can be delivered to an actual endpoint.

4.2 Applied to Invisinets

The core proposal in Invisinets is that we will no longer use private VIPs at all and instead give all endpoints a public VIP (Figure 1). Thus, tenant addresses will be drawn from a CP's public IP address space and we will leverage the CP's PRDO addressing infrastructure to ensure that a tenant's endpoints are not exposed to unwanted access.

To leverage existing CP solutions, we propose a division of labor in which tenants specify *permit lists*: for each endpoint P_i, this is the list of other endpoints that are allowed to communicate with P_i (with the possibility for extension parameters for more specificity) .[3] The CP is responsible for enforcing this permit list: ensuring that only traffic explicitly permitted to access the endpoint may do so. Any packet not cleared by the permit list should be dropped.

Ensuring these semantics to the tenant requires minimal changes to the CP's infrastructure. The CP simply programs SLB bindings for external connections only as necessary based on the tenant's permit list. We use "external" to mean outside any boundary necessary to meet the CP's addressing constraints; *e.g.,* DIPs may only be unique within a region, therefore "external" connections are those spanning beyond the region. We call the boundary within which a DIP is unique the *DIP scope*. The DIP scope determines when an address must be added to the SLB bindings as any connections to endpoints outside the DIP scope may be in overlapping address space and will not be reachable without address translation.

To support this in the datacenter fabric, we first modify the address translation for traffic which typically uses private VIPs (the blue process in Figure 1). Now, packets are translated from the public VIP space to the corresponding DIPs in the vswitch/NIC. Second, the SLB translation process for packets incoming from external endpoints must be slightly modified. The translation between public VIPs and DIPs in the SLB remains the same, but the time at which the SLB binding is installed is changed. Instead of installing when the IP is associated with the VM, the binding is only instantiated when the VM's permit list allows traffic external to the datacenter. When no mapping exists for an incoming packet, the SLB simply drops the traffic (as it does today). This provides an initial coarse layer of protection for PRDO endpoints. Importantly, each endpoint's permit list is enforced at the host as well, so dropping at the SLB is not strictly necessary but offers some defense-in-depth for endpoints.

We assume that in order to support this model, the CP can allocate addresses arbitrarily as is convenient (*e.g.,* the CP isn't required to assign addresses from predetermined prefixes of the address space). The tenant should not expect any particular addressing scheme for the IPs they are assigned. Thus, our API takes reachability (L3) between endpoints as a given from the CP and builds directly on the underlay routing and address translation rather than requiring every tenant to reconstruct their own L3 network.

[2] The SLB is often implemented as a distributed scale-out software system capable of high-speed address translation [21, 54].

[3] For convenience, we will introduce the ability to specify *groups* of endpoints in §5.

4.3 Security Implications

We argue that Invisinets does not fundamentally alter the security posture of either the tenant or CP.

Consider first the public exposure of a tenant's endpoints. Today, the tenant has 3 types of endpoints: (1) public (*e.g.,* a VM associated with a public IP), (2) "semi-private " (*e.g.,* a VM with only a private VIP, but behind a VPN/NAT that restricts access to the VM), and (3) private (*e.g.,* a VM with a private VIP). Under our proposal, nothing changes for endpoints in (1). For endpoints in (3), they are equally unavailable to outside connections in Invisinets (since no SLB bindings will be programmed for them). [4] For endpoints in (2), access is still limited however Invisinets does change *how* this access control is implemented. In Invisinets, access to a semi-private endpoint is restricted by the permit list and SLB bindings; effectively, we're replacing the tenant's VPN/NAT with the CP's SLB/NAT infrastructure.[5]

This change does however, modify the trust boundary between the tenant and the CP. Today, the tenant trusts the cloud provider to implement the network architecture they specify faithfully. In Invisinets, the tenant trusts the cloud provider to construct the necessary network architecture to achieve their connectivity requirements and accordingly trusts that any connectivity not explicitly permitted will not be enabled.

From a CP viewpoint, a potential concern is that the larger number of allocated public IP addresses increases the risk or impact of a DDoS attack on their infrastructure. We found that this was not a concern for CPs because they currently advertise entire IP prefixes, independent of what subset of those addresses are actually allocated: *e.g.,* Azure advertises $>1.4 \cdot 10^{20}$ [46] while GCP advertises $> 7.1 \cdot 10^{26}$ IPv6 addresses [27]. Any of these addresses can be used as the target of a DDoS attack and this remains unchanged with Invisinets.

Furthermore, CPs today deploy cloud DDoS mitigation systems [45] to thwart high-volume traffic attacks targeting a specific endpoint (since these might overwhelm specific links in their WAN). Notably, only one public IP is sufficient for an attacker to attempt such an attack and hence Invisinets does not materially increase the likelihood of such attacks. A final concern might be attacks that spread traffic over multiple endpoints. However, such attacks are already possible today (given the large advertised address space) and handled through filtering at the destination datacenter where SLB mappings reveal what traffic is valid. In such cases, attack traffic is carried over the CP's wide-area links, however the extensive use of load-balancing in these networks is effective in this case (since the attack comprises many smaller flows) and hence the increase in backbone traffic volume is not problematic, akin to a general increase in valid traffic volume.

[4] However, if a private endpoint today is in a virtual network which spans across more than one DIP scope, it would be installed in the SLB.

[5] In some cloud SLB implementations [21], tenant-level NATs are implemented in the SLB. In this case, the only change is that each endpoint has its own binding in the cloud SLB.

API	Description
`request_eip(vm_id)`	Grants endpoint IP
`request_sip()`	Grants service IP
`bind(eip, sip)`	Binds EIP to SIP
`set_permit_list(eip, permit_list)`	Sets access list for EIP
`annotate(eip, middlebox)`	Adds middlebox to EIP's traffic
`set_qos(region, dest, bandwidth)`	Sets (region, dst) BW allowance
`set_qos_class(class, five_tuple)`	Defines tenant QoS class
`tag(eip, tag)`	Associates endpoint with tag

Table 1: Proposed cloud tenant network API.

5 API Design

In this section, we outline the Invisinets API and and describe how it achieves each of its goals. The design implications for each piece of the API are discussed as necessary. We expect the listed arguments to be the minimum required parameters and deliberately leave room for cloud providers to differentiate their services with additional extension parameters. A complete list of our proposed API is shown in Table 1.

5.1 Connectivity

Rationale. As explained in §3, we modify the CP infrastructure to support PRDO addressing for all endpoints and enable trivial connectivity. By giving all endpoints publicly-routable IP addresses, we can abandon the virtual network altogether. Accordingly, tenants are not obligated to construct the networks to facilitate communication outside of a given virtual network, as is required by the inherent isolation of the virtual network abstraction.

API. Connectivity between the tenant's VMs/storage endpoints/*etc.* in the same cloud, across clouds, and to their on-prem network (provided they expose public endpoints) is trivially achieved given that tenant instances have public IP addresses. Thus our basic `request_eip` API allows the tenant to request and receive an *endpoint IP address (EIP)* for each of its instances. A tenant must be prepared to treat its EIP as a flat address with no assumptions about whether its addresses can be aggregated, drawn from certain prefixes, *etc.* This gives providers flexibility in assigning addresses from their overall pool (*e.g.,* to maximize the ability to aggregate for routing, *etc.*) and with effective tagging mechanisms should not affect tenants in any way (since tenants are no longer configuring routing with mechanisms such as BGP).

Design. As discussed in §3, the infrastructure necessary to support PRDO addresses requires only minor modifications to the SLB in the datacenter fabric and otherwise the existing vswitch/NIC functionality is sufficient. When an external endpoint is added to a permit list, only then is the EIP of the local endpoint installed in the SLB.

IPv6 will be necessary to support PRDO-only addressing as the IPv4 address space is too small to feasibly give all cloud endpoints an address. Since these addresses will be allocated arbitrarily from the tenant's perspective, grouping mechanisms will be critical to managing a flat address space. We address this need directly in §5.5.

5.2 Availability

Rationale. Tenants often build highly-available services using multiple backend instances. The service is associated with a service IP address (SIP) and an in-network load balancer maps traffic destined for SIP to an available backend instance. We'd like to support this use-case without requiring that tenants engage with the lower-level details of load balancers.

API. We allow tenants to request a SIP and introduce a *bind* API that allows tenants to associate EIPs with a SIP (Table 1). This SIP address is globally routable, however, traffic destined for the SIP is routed / load-balanced across the EIPs bound to it and we place the responsibility of load balancing on the cloud provider. Hence, the `bind` call allows the tenant to inform the cloud provider of how load-balancing should be implemented, with optional parameters that guide load-balancing policy (*e.g.,* associating a weight with each EIP, akin to weighted fair queuing policies).

Design. Requesting SIPs can be supported today and is in fact somewhat similar to service mesh integrations with cloud load balancers today [29]. The `bind` call only requires changes to the interface tenants use request load balancing services.

5.3 Security

Rationale. In §4.3, we addressed the security implications for the network fabric. We now discuss how tenant-level security concerns are addressed in the Invisinets API. We assume the tenant is primarily concerned about service-level attacks and targeted resource exhaustion. To address potential attacks, our architecture will implement tenant-level security in two main pieces: middlebox annotations on endpoints and network-level permit lists. Both of these are specified by the tenant but implemented/enforced by the CP.

Today, tenants protect their services through a combination of per-VM/virtual network permit lists and by deploying various security appliances such as first-party firewalls and third-party DPI systems (*e.g.,* [52]). Network permit lists are specified by the tenant but implemented by the cloud provider (typically through filtering in the vswitch/NIC at each endpoint) while security appliances may be deployed and managed by the tenant. Together, these mechanisms protect the tenant from both service-level attacks (*e.g.,* intrusion or exfiltration attacks caught by DPI firewalls or proxies) as well as resource exhaustion attacks that specifically target the tenant (*e.g.,* overwhelming the tenant's service).

Ideally, we would remove middleboxes from the cloud offerings altogether and instead implement security measures in an API gateway in the service itself [6, 63]. However, we acknowledge that some applications may have strict security requirements that demand middleboxes and therefore we expand our API to include those available today.

API. The per-host tenant-level permit lists can be naively implemented at the endpoint with the access lists used today such as Network Security Groups in Azure. To the tenant, our API will look similar to these access lists. To permit from another host, the tenant simply uses the `set_permit_list` function to update the given endpoint's allowed hosts.

The tenant may use the `annotate` API to apply the desired middlebox to an endpoint's traffic. The cloud provider is responsible for the instantiation and placement of the middlebox and the routing necessary to direct the relevant traffic. The tenant will provide the type of middlebox (could be their own VM as seen with some third-party network appliances today) and configure it as done today. Additional parameters to the `annotate` API could specify a subset of the traffic to be sent through the middlebox (*i.e.,* by destination) or specify the ordering of a series of middleboxes.

Design. We propose a two-pronged approach to protect tenants from attacks. First, we allow tenants to continue their use of cloud middlebox offerings by annotating endpoints. Therefore, they may continue to use their DPI firewalls, IDS/IPS appliances, *etc.* as they do today. However, the tenant does not have to manage the placement of these appliances in their networks and route relevant traffic. The cloud provider will install the necessary routing in the vswitch/NIC. By including security-focused middlebox functionality in our API, tenants can continue their defense-in-depth best practices as they do today. Secondly, we propose that the cloud provider protect the tenant from network-level resource exhaustion attacks by reusing the same infrastructure it already implements to protect itself. In addition to the above, we assume the cloud provider will continue to enforce the tenant's permit list through filtering at the endpoint's vswitch/NIC. These permit lists are essentially available today as NSGs (Azure) and Security Groups (AWS), so we adopt the underlying architecture unchanged.

5.4 Performance

Rationale. Today, cloud providers offer rather limited network performance/QoS guarantees. Tenants are generally not promised any minimum bandwidth and are instead throttled above a certain threshold. However, some tenants seeking high availability and reliability may reserve a dedicated link [7, 28, 44] which the tenant must then provision, configure and operate as discussed in §2.

The abstraction of a dedicated link is fundamentally at odds with our goal of a high-level endpoint-centric API since a link implies a topology that incorporates it and routing that steers select traffic over the link. Our goal is to avoid this complexity and hence we instead ask: *can we approximate the benefits of these dedicated links without obligating the tenant to worry about the many details they do today?* [6]

The point-to-point link abstraction offered today requires coordination between the entities on either end of the link and

[6] One might ask whether the performance of dedicated links justify their cost and complexity in the first place. In Appendix 11, we present early results showing that these links may not always offer a performance benefit but leave a full evaluation to future work.

is not offered directly between clouds. To avoid the coordination and low-level configuration of direct links, our proposal is to approximate the *effect* of dedicated links by guaranteeing some amount of dedicated egress bandwidth to another domain to tenants who purchase it at a predefined granularity (in this paper, we will assume regional granularity). We hope that since clouds already have an incentive to be highly connected with one another, this guaranteed bandwidth when leaving a region is enough to roughly estimate the effect of dedicated links stitched together at a colocation facility. With this overall goal, we then ask how to provide such an abstraction to the tenant as a service.

The service model we assume is in terms of bandwidth reservations, rather than point-to-point links with a specified bandwidth as is available today. Tenants will specify to the cloud provider their desired amount of dedicated egress bandwidth to another domain (*e.g.,* another cloud) for some region. Then, we seek to make the traffic management necessary to use this link as simple as possible by allowing tenants to define traffic classes and map them to strictly ordered priorities. These priorities will define which traffic gets to use the dedicated bandwidth if/when the aggregate traffic for the tenant in that region is greater than the allowed dedicated bandwidth.

API. Ultimately, the tenant will define their own traffic classes (in terms of five-tuples). The CP will then label traffic as necessary and map these classes to their own high ("dedicated") and low ("best-effort") priority classes in the fabric of the datacenter. To the tenant, reserving the regional aggregate egress bandwidth will simply require calling `set_qos` and setting the priority of traffic via `set_qos_class`.

Design. Our `set_qos` API is based on the assumption that clouds are reasonably well-connected with one another so that dedicated egress bandwidth between a cloud region and another domain can approximate a direct link between the two clouds. If not, congestion between the clouds could impact the available bandwidth beyond the control of either CP. Further, we require that the CP can classify egress traffic into reserved bandwidth packets and best effort-packets. Reserved-class packets are guaranteed to not experience congestion on egress (up to their reserved bandwidth) while best-effort may.

In offering the `set_qos` API, the CP has two primary goals: (1) enforce that the tenant does not consume more than its aggregate egress guarantee and (2) make it easy for the tenant to use all of their promised bandwidth without requiring low-level traffic engineering (as is required today to utilize dedicated links). Since the bandwidths are offered at a per-tenant, per-destination-domain, per-region level, the CP must monitor usage across multiple endpoints in a region and enforce the cumulative bandwidth limit at each endpoint. In doing so, there will be a tradeoff between reactivity and cost as enforcing the limit strictly will impose higher overheads. Scalability is also a challenge as performing distributed rate limiting across all tenant endpoints (in the 10s of millions) must be done with minimal resource consumption.

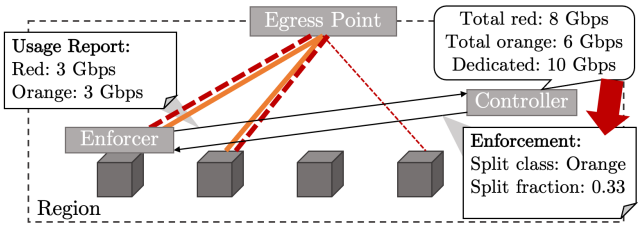


Figure 2: QoS enforcer example. The tenant sends traffic in two classes: red (dashed) and orange (solid), prioritized in that order. The reserved egress bandwidth is 10Gbps.

Our approach to enforce the `set_qos` API is as follows: we assume that tenants define traffic classes each with a different priority level and the cloud provider marks their traffic accordingly. The cloud provider then determines whether a particular traffic class (or what fraction of it) should be assigned reserved vs. best-effort bandwidth based on the tenant's current traffic demand aggregated across all its endpoints (collected at the host and reported to the controller). This mapping between tenant traffic classes and reserved vs. best-effort bandwidth is computed by a per-tenant *QoS controller*. The QoS controller communicates the appropriate mappings to a *QoS enforcer module* at each endpoint (implemented in the NIC or vswitch) which accordingly marks egress packets and rate limits reserved priority flows according to their current bandwidth allocation. Finally, the egress router classifies packets based on these markings and (priority) schedules them accordingly. Hence, the additional infrastructure that our API imposes on the cloud provider is the per-tenant QoS controllers and the QoS enforcer modules (Figure 2).

Using per-tenant controllers mitigates potential scaling issues since each controller need only scale to the number of endpoints per tenant (*vs.* all endpoints) per region, thus dividing the regional rate limiting into reasonably-sized problems.

Figure 2 shows an example of this design for a single tenant. The tenant has two traffic classes, red and orange, which are prioritized in that order. The communication between only one enforcer and the controller is shown for clarity, though each VM would participate. The controller determines that the red class does not use the full egress reservation and allows 1/3 of the orange traffic into the dedicated class. For a more formal discussion of the QoS controller, see Appendix 12.

This model requires cloud providers to engage with an underlying capacity provisioning problem (*i.e.,* how does one ensure that the total reserved bandwidth across all customers is actually there?). For the purpose of this API, we assume that the CP has some policy for installing and allocating bandwidth, but we do not address the specifics of the policy.

5.5 Grouping

Rationale. One noticeable advantage of today's abstractions over a purely endpoint-centric view is that virtual networks can serve as a helpful and simplifying grouping mechanism (*e.g.,* to apply an identical policy or configuration to all VMs

running a Spark job). Without a method to group hosts, reasoning at an endpoint-level may be difficult, especially since tenants are not given EIPs from a continuous address space.

We support the convenience of grouping via tags that can be associated with EIPs such that tenants may use tags in place of a list of EIPs in permit lists. This ensures cleaner semantics for the tenant while the cloud provider can "compile" these to IP addresses for filtering on packets at the end host associated with the permit list. Similar features, called service tags, are available today for Network Security Groups [48] while AWS offers general purpose tagging across resources [59]. Our use of tags is limited to EIPs as they are known across clouds while specific resources are not.

API. To provide the benefits of grouping, we adopt a method for tagging as a means of convenience for tenants in our API with `tag`. Here, tenants may associate an EIP with a given tag as shown in Table 1. Tags can be used in other APIs such as `set_permit_list` in place of addresses.

6 Implementation

To demonstrate the feasibility of our proposed API itself, we have developed an implementation using existing APIs for AWS and Azure to expose our simplified API ; *i.e.,* it effectively builds a shim layer on top of existing APIs. As discussed in earlier sections, most of our API requirements map onto existing abstractions and hence can be implemented in a straightforward manner. The key new addition is the QoS API for which we provide a partial implementation as described below. This approach – *i.e.,* building on top of existing APIs – is unfortunately deeply tied to specific cloud implementations. In the long term, we hope/expect the shim layer to thin over time as clouds provide the Invisinets API as a first-party implementation.

Implementing the QoS API requires three infrastructure components: (i) per-host enforcement modules, (ii) QoS controllers, and (iii) modifications to egress routers to appropriately classify and prioritize traffic. We built a prototype of the first two components but lack the access to realize (iii); in this sense, our implementation of the QoS API is only a partial one. At the same time, such classification and priority scheduling is standard on high-end routers and hence we anticipate no problem in realizing the API more fully in production.

For the enforcer module, the rate limiting is done through Linux `tc` operations and a bpf filter implemented as a bcc program [3]. This approach was chosen to fit easily into any host-based approach, though in a production deployment we expect this logic would reside in the hypervisor or virtual switch [25]. The bcc programs monitor outgoing traffic volumes on a per-traffic-class basis and reports to a reporting process on the host. The host sends usage to a per-tenant controller via RPCs [31]. The controller calculates per-host mappings to each traffic class and reports back to all hosts which reported in the last time interval. The host process then inserts this data into bpf tables for the rate limiter to observe. The rate limiter classifies traffic based on the tenant to cloud provider class mapping. The cloud provider classes correspond to classes in a `tc` hierarchical token bucket which rate limits the egress traffic from the node.

7 Evaluation

In our evaluation, we focus on answering two questions: (i) to what degree does Invisinets simplify a tenant's experience when networking their workloads? and, (ii) is implementing the Invisinets API technically feasible for cloud providers?

We evaluate simplicity using a two-pronged approach. First, we consider three sample tenant deployments and compare the complexity of implementing each deployment using Invisinets versus doing so with existing solutions (§7.1). This approach allows us to do a deep-dive evaluation for specific deployment scenarios and solutions. For a broader view on tenant deployments, we also analyzed 677 publicly available deployment scenarios as captured by their Terraform files on GitHub and quantify the extent to which existing network abstractions contribute to the overall setup and configuration complexity that a tenant faces, and the extent to which Invisinets can remedy this complexity (§7.2).

When considering the feasibility of implementing Invisinets, we focus primarily on scalability. This is because, as discussed earlier, cloud providers already implement (close approximations for) the individual building blocks needed to implement our Invisinets API – *e.g.,* address translators, load balancers, rate limiters. Hence, the main question is whether extending their use of these components to a larger fraction of their tenant pool will create new and problematic scaling bottlenecks. Thus, in §7.3, we evaluate scalability using a combination of system microbenchmarks and deployment statistics from cloud providers.

7.1 Evaluating Simplicity via Case Studies

Methodology. We compare the complexity of tenant networking using Invisinets versus existing solutions. For the latter we consider: (i) a DIY tenant that writes scripts directly atop existing "first-party" cloud APIs, and (ii) Aviatrix [12] as representative of third-party multi-cloud solutions.[7]

Given that there is no best practice for measuring simplicity, we propose a metric which we believe is reasonable, though we do not claim that it perfectly captures all notions of complexity. We measure simplicity in terms of the number of network components that the tenant must consider within three main categories: network boxes, links, and configuration points. Network *boxes* refers to device abstractions such as transit gateways, VPN devices, firewalls, and so forth. *Links* refers to various forms of virtual link abstractions including dedicated egress links (*e.g.,* Azure ExpressRoute [44]), vnet

[7] As mentioned earlier, Aviatrix offers tenants a management layer built atop per-cloud abstractions and optimized cloud appliances. Tenants view their multicloud deployments through a "single pane of glass", but must still be fluent in per-cloud building blocks as well as Aviatrix-specific constructs.

peerings [42], and private internal links [47]. Finally, *configuration points* counts any abstraction that exposes network configuration parameters for the tenant to consider. We count the number of configuration points (*e.g.,* a firewall) rather than lines of code or configuration (*e.g.,* number of firewall rules) since the latter can often be arbitrarily scaled in our scenarios and hence may be misleading.

We further clarify certain aspects of how we account for configuration points. First, every box is also counted as a configuration point: *e.g.,* a gateway is both a box (must be placed in a topology) and a configuration point (must be configured with routes, tunnels, *etc.*). Second, while boxes are always configuration points, the inverse is not true: *e.g.,* abstractions such as virtual networks or subnets are not boxes but do require configuration and hence are counted as configuration points. Invisinets in particular has multiple configuration points but few boxes so both measures are needed for a fair comparison. Finally, we ignore security groups/permit lists and endpoint IP addresses when counting configuration points as these are present in all solutions and depend on the number of endpoints which can be scaled arbitrarily.

We compare solutions using three case studies: one is a validated design published by Aviatrix [11] and the other two were defined by us to represent extremes of simple *vs.* sophisticated deployments.

Case Study 1: A Simple Tenant Network. We start with a rather contrived, simple deployment in which a single VM in one cloud (Cloud A) must communicate with a service in another cloud (Cloud B) in a shared address space as shown in Figure 3a. The service in Cloud B uses two VMs which are load balanced. Table 2 shows the complexity of implementing this scenario with each of the three solutions we consider.

Metric	First-Party	Aviatrix	Invisinets
Boxes	3 (VPN, LB)	3 (GW, LB)	0
Config. Point	7 (vnet, subnet, VPN)	7 (vnet, subnet, VPN)	1 (SIP)

Table 2: Complexity analysis for a Simple Tenant Network. In parentheses, we list the "top three" abstractions in terms of their contribution to complexity (see Fig. 3 for abbreviations).

We see that even this simple scenario incurs non-trivial complexity with existing solutions. With the DIY approach, tenants must still consider virtual network gateways, a load balancer, backend pools, local network gateways, route propagation parameters, and more, leading to a total complexity of 10 network components. Aviatrix is similar, though it uses 2 Aviatrix gateways in lieu of the first-party VPN gateways.

Invisinets, however, completely eliminates the virtual network layer and instead requires just one configuration point (the SIP in Cloud B), allowing the deployment to be expressed in just 9 lines of code as shown below:

```
eip1 = cloud_a.request_eip(vm1_id, name="vm1")
eip2 = cloud_b.request_eip(vm2_id, name="vm2")
eip3 = cloud_b.request_eip(vm3_id, name="vm3")
sip = cloud_b.request_sip(name="service")
bind(eip2, sip)
bind(eip3, sip)
set_permit_list(eip1, [eip2, eip3])
set_permit_list(eip2, [eip1])
set_permit_list(eip3, [eip1])
```

By contrast, our DIY script using the AWS and Azure Python APIs requires over 45 lines of code (snippets shown in Appendix 13) even assuming IPs and the underlying virtual network have already been provisioned.

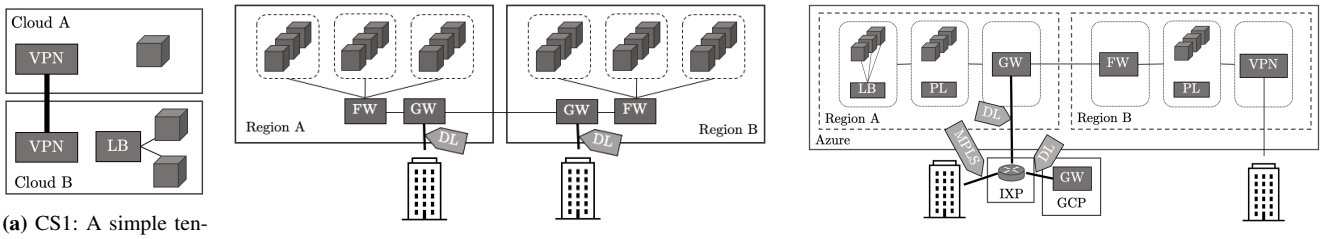
Case Study 2: (Aviatrix) Multi-Region Design. We consider a design from Aviatrix [11], as seen in Figure 3b. This deployment involves 3 virtual networks running different services in two different cloud regions. They are connected to one another via a transit virtual network which contains firewalls for security. These transit virtual networks also contain direct links to on-prem datacenters. Table 3 summarizes the complexity costs in implementing this design.

Metric	First-Party	Aviatrix	Invisinets
Boxes	4 (GW, FW)	18 (GW, FW)	2 (FW)
Links	9 (peering, DL)	2 (DL)	0
Config. Point	29 (peering, subnet, vnet)	36 (GW)	2 (egress BW)

Table 3: Complexity analysis for the Aviatrix design. See Fig. 3 for abbreviations.

We see that, compared to our first case study, complexity rises significantly with both the DIY and Aviatrix solutions. The majority of this complexity comes from the gateway and virtual network peering abstractions. Interestingly, Aviatrix incurs greater complexity than a DIY implementation due to its recommended redundant gateways, though first-party peerings cannot be redundant. In contrast, Invisinets requires only 4 network components (egress reservations to approximate the direct links and middlebox annotations for firewalls), which represents a more than 90% reduction in complexity relative to the DIY and Aviatrix solutions. Creating this deployment with the available first-party APIs would take over 45 lines of code even assuming all instances and their IPs have been provisioned and the out-of-band coordination to set up the direct links has been performed. In addition, we do not count lines only defining configuration, otherwise the script is well over 200 lines. A significant portion of this code sets up the necessary routes to get traffic from each virtual network to the appropriate firewall and/or gateway.

Case Study 3: A Heterogeneous Tenant Network. Our third scenario showcases a network deployment that involves a range of connectivity requirements, as shown in Figure 3c. This scenario is representative of virtual networks constructed by larger cloud tenants, though we chose to scale it down for understandability. For the sake of demonstration, in Figure 3c, the Azure network is depicted in some detail while the GCP deployment is unrealistically simple. In this scenario, the tenant's cloud deployments in GCP and Azure are each connected to the tenant's on-prem datacenter via direct links to an Internet exchange point where the tenant has reserved a virtual router and an MPLS link to their datacenter. In the Azure deployment, the ExpressRoute terminates at a VPN gateway which must reside in its own subnet [49]. From there, user-defined routes send traffic to the appropriate subnet or



(a) CS1: A simple tenant network.

(b) CS2: Aviatrix multi-region design [11].

(c) CS3: A heterogeneous tenant network.

Figure 3: Generalized (not taking the form of any particular current approach) network topologies for each case study. (LB = load balancer, GW = gateway, FW = firewall, PL = private link, DL = Direct Link)

onto the ExpressRoute for egress traffic to GCP or On-Prem. The other subnets in the virtual network contain VMs and a private link to an Azure service such as storage on the left and load-balanced VMs on the right. In another virtual network in Azure, a VPN gateway tunnels to a branch office. Ingress traffic is directed to a subnet containing a firewall which then allows traffic to reach VMs or another private link to a Microsoft service. Table 4 summarizes the complexity costs in this design. A script to create this deployment (again, ignoring instances, IPs, and some ExpressRoute setup) would require over 80 lines of code. While coming in the form of many different components, achieving basic connectivity via gateways, links, *etc.* continues to be the main source of complexity.

Metric	First-Party	Aviatrix	Invisinets
Boxes	6 (GW, FW, LB)	6 (GW, FW, LB)	2 (FW)
Links	5 (DL, PL, MPLS)	5 (DL, PL, MPLS)	0
Config. Point	22 (vnet, subnet, GW)	22 (vnet, subnet, GW)	4 (egress BW, FW)

Table 4: Complexity analysis for the heterogenous tenant network. See Fig. 3 for abbreviations.

7.2 Terraform Complexity

For a broader perspective, we now consider the impact of Invisinets on a larger pool of deployment scenarios. Specifically, we evaluate the scope for simplification in the Terraform files that a network administrator maintains. Terraform [32] is a popular language for specifying cloud infrastructure as declarative code directly using the first-party abstractions.

Methodology. We scraped GitHub for Terraform files creating virtual networks, selecting files that mention an AWS VPC and omitting files that only defined Terraform variables or outputs. For a scrape conducted on 9/15/2022, our filtering yielded 677 files for analysis.[8] We use these configurations since they are publicly available but recognize that they may not fully represent the tenant use-cases that Invisinets targets (*e.g.,* mid/large scale enterprises). We believe our analysis could be easily repeated on large production deployments.

Results. We parse each Terraform file, identifying whether a code block corresponds to a virtual network component. Table 5 lists the components we identified and how often they occurred in our files.[9] We also calculate the lines of code within each of these code blocks and show the total per-component line count in the last column of Table 5. Unsurprisingly, VPCs are the most common component. Beyond this, we see that abstractions used to establish basic connectivity – *e.g.,* Internet Gateway, Subnet, Route Table, Route – constitute a significant fraction of the networking abstractions that an administrator must deal with. Moreover, even complex routing abstractions such as Transit Gateways are not uncommon. These findings thus support our thesis that an approach such as Invisinets, which altogether eliminates the need for virtual topologies and their routing configurations, can substantially simplify networking configurations.

Virtual Network Component	Occurrence Count	Line Count
VPC	2,493	26,731
Route Table	1,839	13,317
Subnet	1,514	14,677
Security Group	456	9,704
Internet Gateway	445	2,802
Route	339	2,184
NAT Gateway	209	1,661
Network ACL	141	2,841
Transit Gateway	82	587
VPN Gateway	40	316
Load Balancer	22	207
Network Interface	16	123
VPN Peering Connections	5	27
Customer Gateway	4	58
VPN Route	2	10
VPN Connections	1	13

Table 5: Breakdown of Terraform lines removed (across all 677 scraped files) by virtual network component.

The lines of code identified in Table 5 can be viewed as an upper bound on the lines of code that can be omitted by using Invisinets. To evaluate whether this is a significant portion of the overall Terraform configuration, Figure 4 shows a histogram of the percentage of lines-of-code that can be omitted from the Terraform files we consider.

While the fraction of omitted configuration lines may be surprising, these resource definitions rarely carry information critical to the four tenant goals around which the Invisinets API is designed. Instead, many of these lines specify details of unnecessary abstractions such as virtual networks and the gateways required to reach external endpoints. (We show an

[8] Github API search results are limited to the first 1000 results. Additional filtering on returned files reduced the number further.

[9] We believe this estimate is approximately equivalent to the set of network components (boxes, links, and configuration points) that we measure in §7.1.

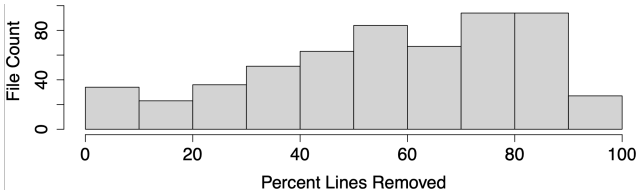


Figure 4: Histogram of percent lines of Terraform resource files that can be omitted with Invisinets API.

example of such a configuration in Appendix 14). Accordingly, the streamlined Invisinets API does not require much of this configuration.

We note that our estimates for omitted code in this section are an upper bound since we do not account for the lines of configuration that would be *added* by using Invisinets's API. Nonetheless, our evaluation in the previous section suggests that Invisinets requires far fewer (90+% fewer) components to configure and hence we expect to be left with significant savings. Moreover, the permit lists that Invisinets requires contain information that is necessary even without Invisinets; *e.g.,* in the form of VPN access lists, firewall rules, NSG configurations, and so forth.

7.3 Scalability

As mentioned earlier, we focus on the scalability of the Invisinets API. Implementing the Invisinets API implies change along two main fronts: address management to support PRDO-only addressing, and enforcing per-tenant egress bandwidth reservations. We consider each in turn.

PRDO addressing. PRDO-only addressing changes the number of endpoints that are assigned public IP addresses and hence a natural question is whether cloud providers have a large enough public IP addresses to give one to every VM. From discussions with Azure operators, we learnt that Azure advertises over $1.4 \cdot 10^{20}$ addresses in total [46] and hosts O(10M) VMs. Thus we can safely conclude that Azure has sufficient public addresses to support PRDO addressing. Similarly, GCP has over $7.1 \cdot 10^{26}$ advertised addresses in total [27], presumably plenty to allocate to all endpoints.

The next scalability concern may be the impact on the address translation infrastructure – specifically, the vswitch/NIC and SLBs used to translate PRDO addresses to internal private addresses and to enforce permit lists. Fortunately, the PRDO-only infrastructure does not increase the number or complexity of vswitch/NIC lookup operations, as every packet is translated to DIPs even today. Similarly, in considering the impact on the SLB, we note that Invisinets does not change the number of internal endpoints that need to be reachable from endpoints outside the cloud provider (defined as semi-private addresses in §3) since this depends on the tenant's workload requirements rather than the addressing architecture. Today, the bindings and permit-list rules for these endpoints are programmed in per-tenant VPNs/NATs while Invisinets implements the same in the cloud provider's SLB. Since cloud SLBs [21, 54] are already designed for elastic horizontal scaling, we do not anticipate any scaling challenges due to semi-private addresses.

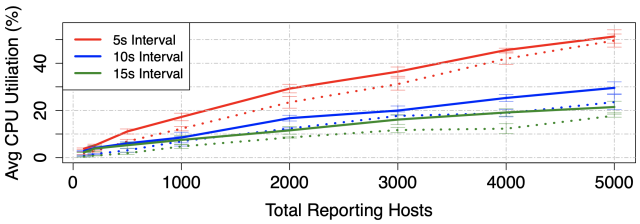


Figure 5: CPU utilization of QoS controller for different reporting intervals over 5 experiments. The report collection (solid) is the bottleneck over aggregation and reporting (dotted).

QoS Enforcer. Invisinets adds per-tenant, per-region QoS controllers to a cloud provider's infrastructure. We now benchmark the QoS components of our implementation to demonstrate that these QoS controllers could reasonably scale in a production setting. Our implementation consumes additional network bandwidth for communication between the QoS controller and enforcer modules. (The volume of tenant traffic remains unchanged with only ToS fields modified to reflect the class of traffic.) Using 32 traffic classes (chosen to be expressive), we record a worst-case overhead of only 800 bytes per host in one reporting interval for one reservation. For a VM with a 5 Gbps link, communication with the QoS controller consumes less than 0.00001% of its link bandwidth when reporting at 10 second intervals. At the QoS controller, this overhead is $n \times 800$ bytes per interval where n is the number of VMs per tenant per region. Discussions with a major cloud provider $CP-X$ revealed that, for their deployments, n is under 50,000 per region in the worst case, and $O(10)$ on average. Hence, for a 10 second reporting interval, the total bandwidth consumed at the QoS controller is around 32Mbps in the worst-case and far lower for average tenant sizes.

Next, we measure the CPU utilization at our QoS controller for increasing numbers of reporting hosts, shown in Figure 5. Our QoS controller is comprised of two main processes: a gRPC [31] server to collect bandwidth reports and an aggregation process to calculate the mapping and report back to hosts. We run our QoS controller on a t3.small instance in AWS with 2 vcpus, 2 GB of memory, and 5 Gbps bandwidth, chosen to ensure that network capacity is not a bottleneck. One hundred t2.small instances were used as hosts, with many host processes per instance. As shown in the figure, stressing our 2vCPU instance requires 1000s of reporting hosts. We believe this is a reasonable expectation for region-scale deployments since the average number of VMs per tenant is $O(10)$. Depending on the interval, with enough hosts, the collection process cannot keep up with incoming reports at which point the controller size (*i.e.,* vcpus) can be scaled as necessary. Scaling up to support a tenant with potentially millions of hosts in one region would pose an additional challenge in aggregating bandwidth measurements; we leave that challenge to future work.

Finally, we consider the total resource consumption due

to QoS controllers (across all tenants). In Azure, there are roughly an average of 16,000 tenants in each region. Accordingly, we expect the QoS enforcer to consume fewer than 1,600 cores in each region on average (since the total utilization for 100 hosts and 5 s intervals is <10%).

Despite the additional packet-processing logic due to the enforcer, no significant overheads are seen in processing host traffic. We measured the latency and throughput of outbound traffic from a host with the QoS enforcer enabled and disabled. Neither metric demonstrated a noticeable overhead. Throughput measurements using iperf [1] between two t2.medium instances for 10 second flows produced an average of 64.9 Mbps across 100 flows with the rate limiting (though set to allow many Gbps to not interfere) and 64.8 Mbps without. Latency was tested with 200 pings to a remote host. With the rate limiter enabled, the average was 0.46 ms ($\sigma = 0.12$) while the average was 0.48 ms ($\sigma = 0.258$) without it.

Thus we conclude that Invisinets's QoS infrastructure introduces little overhead for cloud providers.

8 Limitations

We acknowledge that Invisinets may not satisfy infrastructure requirements for all cloud tenants. Despite the layers of protection from the cloud provider, the tenant may deem public IPs to be an unacceptable risk for their endpoints. While addresses in Invisinets are default-off, this choice removes the layer of isolation provided by virtual network private address spaces. We note, however, that default-off semantics are present in both Invisinets and today's virtual network abstractions. Only the placement of these parameters differ (*e.g.,* endpoint permit list vs. gateway rules). In addition, the workload may have security requirements defined in terms of today's networking abstractions. Accordingly, we do not expect this interface to appeal to all tenants and target specifically cloud-native applications.

As mentioned in 5.4, our QoS API's ability to simulate the effects of a direct link is dependent on the level of connectivity between the clouds. This assumption may be particularly precarious in areas with smaller cloud footprints.

Since our primary goal with the Invisinets API was to simplify the tenant interface, our evaluation is inherently limited. Complexity is a multifaceted issue and cannot be completely captured in any one metric. We proposed a variety of metrics to evaluate simplicity as fairly as we could, though the values are not necessarily exact.

9 Related Work

We build on the extensive literature on cloud and network virtualization [20, 24, 53, 54, 60]. As mentioned in §5, most of our API leaves this underlying architecture unchanged or extends it in relatively straightforward ways.

Distributed rate limiting has also been extensively studied [37, 39, 56, 58, 64]; *e.g.,* with work on optimizing end host traffic shaping [58, 64]. Systems such as [39, 56] make network-wide rate limiting decisions similarly to the model we adopt, though [56] primarily seeks to limit traffic aggregates rather than provide a minimum guarantee. Such guarantees are provided by [40] with strategic resource placement while [37] modifies the end host networking stack.

We take the large body of work in network verification [14, 23, 26, 36, 38] as evidence of the overwhelming management complexity in networking. While verification has focused more on datacenter environments, virtual networks suffer similar complexity since they use similar abstractions. Generally, compiling high-level network intents into low-level device configs remains difficult and error prone [15].

This paper joins others in advocating a forward-looking view of cloud networking though prior proposals focused on performance guarantees [13, 50, 51] or declarative control [16] rather than the simplification we target. In [4], a platform to unify cloud services is developed by creating a substrate infrastructure across clouds. The vision in [61] takes the extreme stance of proposing a unified interface for all clouds. Invisinets can be viewed as an instantiation of their vision for networking, though we focus less on unification and instead hope that simpler APIs across clouds will result in some homogenization over today's highly-siloed APIs.

10 Conclusion

To simplify networking for cloud tenants, we proposed a declarative and endpoint-centric API which takes L3 connectivity as a given and removes the burden of deep networking knowledge from tenant operators. We acknowledge that our model may not initially meet the requirements of all tenants. However, our proposed API can coexist with existing abstractions and, in fact, can provide a spectrum of simplicity where deployments may include both today's building blocks as well as our proposed architecture. Our API requires consideration of fewer network components and obviates the need for network topologies to be constructed at all. Supporting this simple API requires minimal infrastructural changes to cloud datacenters and the new systems should be easily scaled. We believe the Invisinets API for virtual networking follows the evolution seen in cloud compute and storage from virtual replicas of physical components to higher-level services.

References

[1] iperf. https://iperf.fr/.

[2] tcpping. http://www.vdberg.org/~richard/tcpping.html.

[3] BPF Compiler Collection (BCC), 2022. https://github.com/iovisor/bcc.

[4] M. Alaluna, E. Vial, N. Neves, and F. M. V. Ramos. Secure and dependable multi-cloud network virtualization. In *Proceedings of the 1st International Workshop on*

Security and Dependability of Multi-Domain Infrastructures, XDOMO'17, New York, NY, USA, 2017. Association for Computing Machinery.

[5] Alkira. Alkira, 2022. https://www.alkira.com/.

[6] Amazon Web Services. Amazon api gateway, 2022. https://aws.amazon.com/api-gateway/.

[7] Amazon Web Services. Aws direct connect, 2022. https://aws.amazon.com/directconnect/.

[8] Amazon Web Services. Connect your vpc to other networks, 2022. https://docs.aws.amazon.com/vpc/latest/userguide/extend-intro.html.

[9] Amazon Web Services. Vpcs and subnets, 2022. https://docs.aws.amazon.com/vpc/latest/userguide/VPC_Subnets.html.

[10] Aurea. Cloudfix, 2022. https://cloudfix.com/.

[11] Aviatrix. Aviatrix multi-region high-availability cloud network design. https://aviatrix.com/aviatrix-multi-region-high-availability-cloud-network-design/.

[12] Aviatrix. Aviatrix, 2022. https://aviatrix.com/.

[13] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, page 242–253, New York, NY, USA, 2011. Association for Computing Machinery.

[14] R. Beckett, A. Gupta, R. Mahajan, and D. Walker. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 155–168, New York, NY, USA, 2017. Association for Computing Machinery.

[15] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations: Brief reflections on abstractions for network programming. *SIGCOMM Comput. Commun. Rev.*, 49(5):104–106, nov 2019.

[16] T. Benson, A. Akella, A. Shaikh, and S. Sahu. Cloudnaas: A cloud networking platform for enterprise applications. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, New York, NY, USA, 2011. Association for Computing Machinery.

[17] Cerba. Densify, 2022. https://www.densify.com/.

[18] CloudBolt Software. Cloudbolt cost security management platform, 2021. https://www.cloudbolt.io/kumolus/.

[19] CloudZero. Cloudzero, 2022. https://www.cloudzero.com/.

[20] M. Dalton, D. Schultz, J. Adriaens, A. Arefin, A. Gupta, B. Fahs, D. Rubinstein, E. C. Zermeno, E. Rubow, J. A. Docauer, J. Alpert, J. Ai, J. Olson, K. DeCabooter, M. de Kruijf, N. Hua, N. Lewis, N. Kasinadhuni, R. Crepaldi, S. Krishnan, S. Venkata, Y. Richter, U. Naik, and A. Vahdat. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 373–387, Renton, WA, Apr. 2018. USENIX Association.

[21] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein. Maglev: A fast and reliable software network load balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 523–535, Santa Clara, CA, Mar. 2016. USENIX Association.

[22] J. L. Eric Brewer. Application modernization and the decoupling of infrastructure services and teams, 2019. https://services.google.com/fh/files/blogs/anthos_white_paper.pdf.

[23] S. K. Fayaz, T. Sharma, A. Fogel, R. Mahajan, T. Millstein, V. Sekar, and G. Varghese. Efficient network reachability analysis using a succinct control plane representation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 217–232, Savannah, GA, Nov. 2016. USENIX Association.

[24] A. D. Ferguson, S. Gribble, C.-Y. Hong, C. Killian, W. Mohsin, H. Muehe, J. Ong, L. Poutievski, A. Singh, L. Vicisano, R. Alimi, S. S. Chen, M. Conley, S. Mandal, K. Nagaraj, K. N. Bollineni, A. Sabaa, S. Zhang, M. Zhu, and A. Vahdat. Orion: Google's Software-Defined networking control plane. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 83–98. USENIX Association, Apr. 2021.

[25] D. Firestone. Vfp: A virtual switch platform for host sdn in the public cloud. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI) 2017*, March 2017.

[26] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 469–483, Oakland, CA, May 2015. USENIX Association.

[27] Google. cloud.json, 2022. https://www.gstatic.com/ipranges/cloud.json.

[28] Google. Dedicated interconnect overview, 2022. https://cloud.google.com/network-connectivity/docs/interconnect/concepts/dedicated-overview.

[29] Google. Load balancer overview, 2022. https://cloud.google.com/anthos/clusters/docs/multi-cloud/aws/how-to/load-balancers.

[30] Google. Secure and encrypted communication between anthos clusters using anthos service mesh, 2022. https://cloud.google.com/architecture/encrypt-secure-communication-between-multiple-anthos-clusters-concept.

[31] gRPC Authors. gRPC, 2022. https://grpc.io/.

[32] Hashicorp. Terraform, 2022. https://www.terraform.io/.

[33] Istio Authors. Vpn connectivity, 2019. https://istio.io/v1.1/docs/setup/kubernetes/install/multicluster/vpn/.

[34] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 3–14, New York, NY, USA, 2013. Association for Computing Machinery.

[35] S. Janardhan. Update about the october 4th outage, oct 2021. https://engineering.fb.com/2021/10/04/networking-traffic/outage/.

[36] K. Jayaraman, N. Bjørner, J. Padhye, A. Agrawal, A. Bhargava, P.-A. C. Bissonnette, S. Foster, A. Helwer, M. Kasten, I. Lee, A. Namdhari, H. Niaz, A. Parkhi, H. Pinnamraju, A. Power, N. M. Raje, and P. Sharma. Validating datacenters at scale. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 200–213, New York, NY, USA, 2019. Association for Computing Machinery.

[37] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, A. Greenberg, and C. Kim. EyeQ: Practical network performance isolation at the edge. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 297–311, Lombard, IL, Apr. 2013. USENIX Association.

[38] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 113–126, San Jose, CA, Apr. 2012. USENIX Association.

[39] A. Kumar, S. Jain, U. Naik, A. Raghuraman, N. Kasinadhuni, E. C. Zermeno, C. S. Gunn, J. Ai, B. Carlin, M. Amarandei-Stavila, M. Robin, A. Siganporia, S. Stuart, and A. Vahdat. Bwe: Flexible, hierarchical bandwidth allocation for wan distributed computing. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 1–14, New York, NY, USA, 2015. Association for Computing Machinery.

[40] J. Lee, Y. Turner, M. Lee, L. Popa, S. Banerjee, J.-M. Kang, and P. Sharma. Application-driven bandwidth guarantees in datacenters. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 467–478, New York, NY, USA, 2014. Association for Computing Machinery.

[41] S. McClure, S. Ratnasamy, D. Bansal, and J. Padhye. Rethinking networking abstractions for cloud tenants. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 41–48, New York, NY, USA, 2021. Association for Computing Machinery.

[42] Microsoft. Virtual network peering. https://learn.microsoft.com/en-us/azure/virtual-network/virtual-network-peering-overview, year = 2022, lastaccessed = March 12, 2023.

[43] Microsoft. Overview of load-balancing options in azure, 2021. https://docs.microsoft.com/en-us/azure/architecture/guide/technology-choices/load-balancing-overview.

[44] Microsoft. What is azure expressroute?, 2021. https://docs.microsoft.com/en-us/azure/expressroute/expressroute-introduction.

[45] Microsoft. Azure ddos protection standard overview, 2022. https://docs.microsoft.com/en-us/azure/ddos-protection/ddos-protection-overview.

[46] Microsoft. Azure ip ranges and service tags – public cloud, 2022. https://www.microsoft.com/en-us/download/details.aspx?id=56519.

[47] Microsoft. Private link, 2022. https://azure.microsoft.com/en-us/products/private-link/.

[48] Microsoft. Virtual network service tags, 2022. https://docs.microsoft.com/en-us/azure/virtual-network/service-tags-overview.

[49] Microsoft. What is vpn gateway?, 2022. https://docs.microsoft.com/en-us/azure/vpn-gateway/vpn-gateway-about-vpngateways.

[50] J. C. Mogul and L. Popa. What we talk about when we talk about cloud network performance. *SIGCOMM Comput. Commun. Rev.*, 42(5):44–48, sep 2012.

[51] J. C. Mogul and J. Wilkes. Nines are not enough: Meaningful metrics for clouds. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 136–141, New York, NY, USA, 2019. Association for Computing Machinery.

[52] Palo Alto Networks. Palo alto networks, 2022. https://www.paloaltonetworks.com/.

[53] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri. Ananta: Cloud scale load balancing. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 207–218, New York, NY, USA, 2013. Association for Computing Machinery.

[54] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri. Ananta: Cloud scale load balancing. 43(4):207–218, aug 2013.

[55] PCI Security Standards Council. Pci security standards council, 2022. https://www.pcisecuritystandards.org/.

[56] B. Raghavan, K. Vishwanath, S. Ramabhadran, K. Yocum, and A. C. Snoeren. Cloud control with distributed rate limiting. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '07, page 337–348, New York, NY, USA, 2007. Association for Computing Machinery.

[57] D. Ramel. Research brief summarizes trends in multicloud deployments, October 2019. https://virtualizationreview.com/articles/2019/10/21/cloud-trends.aspx.

[58] A. Saeed, N. Dukkipati, V. Valancius, T. Lam, C. Contavalli, and A. Vahdat. Carousel: Scalable traffic shaping at end-hosts. In *ACM SIGCOMM 2017*, 2017.

[59] A. W. Services. Tagging aws resources, 2022. https://docs.aws.amazon.com/general/latest/gr/aws_tagging.html.

[60] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, H. Liu, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. *Commun. ACM*, 59(9):88–97, aug 2016.

[61] I. Stoica and S. Shenker. From cloud computing to sky computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 26–32, New York, NY, USA, 2021. Association for Computing Machinery.

[62] Strickx, Tom and Hartman, Jeremy. Cloudflare outage on june 21, 2022, jun 2022. https://blog.cloudflare.com/cloudflare-outage-on-june-21-2022/.

[63] The Kubernetes Authors. Ingress controllers, 2021. https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/.

[64] K. To, D. Firestone, G. Varghese, and J. Padhye. Measurement based fair queuing for allocating bandwidth to virtual machines. In *Proceedings of the 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, HotMIddlebox '16, page 14–19, New York, NY, USA, 2016. Association for Computing Machinery.

[65] U.S. Department of Health and Human Services Office for Civil Rights. Hipaa administrative simplification, March 2013.

[66] Virtana. Virtana, 2022. https://www.virtana.com/.

[67] VMWare. Multi cloud operations: Visibility & control, 2022. https://www.vmware.com/cloud-solutions/multi-cloud-ops.html.

[68] Volterra. Volterra, 2022. https://www.volterra.io/.

Appendix

11 Benefits of dedicated links.

Further, we conducted a small experiment to determine the benefit of these dedicated links. We provisioned an ExpressRoute from Azure in Northern California to a colocation facility in Silicon Valley and connected it to Direct Connect to AWS in Northern California. A diagram is shown in Figure 6. Both dedicated links were 50Mpbs and the virtual router in the colocation facility can handle up to 500Mbps. In parallel to this dedicated connection between clouds, we connected two hosts in each deployment via the public Internet. We collected throughput measurements using iperf [1] every 5 minutes for a week and performed tcppings [2] every minute. We found that at these low bandwidths, the primary performance benefit is consistent throughput (see summary in Table 6). Notably, the latency can even be worse across these dedicated links, though not significantly so considering the variability.

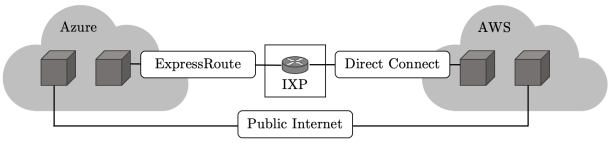


Figure 6: Direct link measurement setup.

Client Cloud	Direct Link?	Measurement	Mean	Std. Dev.
AWS	Yes	Throughput	51 Mbps	0.23 Mbps
AWS	No	Throughput	169 Mbps	85 Mbps
Azure	Yes	Throughput	50 Mbps	1.4 Mbps
Azure	No	Throughput	117 Mbps	80 Mbps
AWS	Yes	Latency	4.89 ms	1.74 ms
AWS	No	Latency	3.38 ms	1.73 ms
Azure	Yes	Latency	12.53 ms	9.36 ms
Azure	No	Latency	7.45 ms	4.39 ms

Table 6: Summary of direct link measurements.

12 QoS Controller

More formally, the job of a QoS controller is as follows. Each tenant, t, can have n classes of traffic, decreasing in priority: $C_1, C_2 \ldots$ Traffic in these classes must be mapped to the cloud provider best-effort and dedicated classes B and D. The tenant reserves some dedicated egress bandwidth, r_t. To map the classes, each host x reports x_i^t, the bandwidth tenant t consumed in class C_i every interval of k seconds to the per-tenant controller. Every interval, the controller calculates the total bandwidth for each class, C_i. Starting with the highest priority class, C_1, the controller adds x_i^t to the running total dedicated bandwidth, x_D, and maps the class to the dedicated cloud provider class, D. When $x_D > r_t$, the controller maps the current class C_s to a split class, S. All subsequent classes $C_{j>s}$ are mapped to B. The fraction of the bandwidth used by C_i which would fit into the reserved bandwidth $f_s = (x_D - r_t)/C_s$ is calculated as well. The controller sends f_s and C_s to all hosts that reported in the previous interval.

The (enforcement module at) hosts then install the calculated mapping between tenant and cloud provider classes. Traffic belonging to $C_i < C_s$ is simply marked as reserved (D) while traffic in $C_i > C_s$ is marked as best-effort (B). For traffic belonging to C_s, the host calculates the maximum allowed dedicated bandwidth in the split class $b_d = f_s \cdot x_i^t$. This value is used for per-flow admission to D for traffic in C_s. When a new flow in C_s arrives, a timestamp is recorded. After m time has passed, the flow is eligible for promotion to D. Its bandwidth over the last m seconds, b_{flow}, is compared against b_d. If $b_{flow} < b_d$ the flow is mapped to D and b_d is updated ($b_d = b_d - b_{flow}$), otherwise the flow is mapped to B. This evaluation is performed every m seconds for each flow in C_s which has not been admitted to D. Accordingly, this process requires per-flow state, though only proportional to the number of flows in C_s during a single reporting interval.

13 Case Study 1 Code

In Figure 7 is an excerpt of the code required to implement Case Study 1 using first-party cloud APIs.

```
def setup():
    vnet = network_client.virtual_networks.get(RESOURCE_GROUP, vnet_name)
    eip = network_client.public_ip_addresses.get(RESOURCE_GROUP, EIP1_NAME)
    eip1_config = network_client.\
        network_interface_ip_configurations.get(RESOURCE_GROUP,
                                                get_nic_name_from_ipconf_id(
                                                    eip.ip_configuration.id),
                                                get_resource_name_from_id(
                                                    eip.ip_configuration.id))

    eip = network_client.public_ip_addresses.get(RESOURCE_GROUP, EIP2_NAME)
    eip2_config = network_client.\
        network_interface_ip_configurations.get(RESOURCE_GROUP,
                                                get_nic_name_from_ipconf_id(
                                                    eip.ip_configuration.id),
                                                get_resource_name_from_id(
                                                    eip.ip_configuration.id))

    backend_address1 = \
        LoadBalancerBackendAddress(name="endpoint1-backend",
                                   virtual_network=SubResource(id=vnet.id),
                                   ip_address=eip1_config.private_ip_address)
    backend_address2 = \
        LoadBalancerBackendAddress(name="endpoint2-backend",
                                   virtual_network=SubResource(id=vnet.id),
                                   ip_address=eip2_config.private_ip_address)

    params = PublicIPAddress(location=LOCATION,
                             sku=PublicIPAddressSku(name="Standard"),
                             public_ip_allocation_method="Static",
                             public_ip_address_version="IPv4")
    poller = network_client.\
        public_ip_addresses.begin_create_or_update(RESOURCE_GROUP,
                                                   "lb-ip",
                                                   params)
    wait_for_complete(poller)
    lb_ip = poller.result()
    ip_config = FrontendIPConfiguration(name="lb-ipfrontend",
                                        public_ip_address=lb_ip)

    # Set up probes and backend pools
    probe = Probe(name="lb_probe", protocol="Tcp", port=80,
                  interval_in_seconds=5, number_of_probes=2)
    backend_pool = BackendAddressPool(name="lb-pool-1",
        load_balancer_backend_addresses=\
        [backend_address1, backend_address2])

    # Create a new LB
    # ...
```

Figure 7: Excerpt of the code necessary to setup the Azure side of Case Study 1.

14 Terraform Example

Below is a code snippet from one of our scraped Terraform files.

```
resource "aws_subnet" "mgmt_subnet2" {
  vpc_id            = aws_vpc.vpc_sec.id
  cidr_block        =
  var.security_vpc_mgmt_subnet_cidr2
  availability_zone = var.availability_zone2
  tags = {
    Name = "$mgmt-subnet2"
  }
}

# Routes
resource "aws_route_table" "data_rt" {
  vpc_id = aws_vpc.vpc_sec.id
  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.igw_sec.id
  }
  route {
    cidr_block         = var.spoke_vpc1_cidr
    transit_gateway_id =
    aws_ec2_transit_gateway.TGW-XAZ.id
  }
  route {
    cidr_block         = var.spoke_vpc2_cidr
    transit_gateway_id =
    aws_ec2_transit_gateway.TGW-XAZ.id
  }
  tags = {
    Name = "$data-and-mgmt-rt"
  }
}
```

Figure 8: Code snippet from a scraped Terraform file.