Rethinking Networking Abstractions for Cloud Tenants

Sarah McClure
sarah@eecs.berkeley.edu
UC Berkeley

Sylvia Ratnasamy
sylvia@eecs.berkeley.edu
UC Berkeley

Deepak Bansal
dbansal@microsoft.com
Microsoft

Jitendra Padhye
padhye@microsoft.com
Microsoft

ABSTRACT

We argue that network virtualization as experienced by many cloud tenants is overly complex and needs to be rethought. We propose that the goal for a new design should be to free cloud tenants entirely from having to build and operate virtual networks. Building on this philosophy, we propose that instead of low-level building blocks (virtual links, routers, firewalls), cloud networking should be exposed to tenants in a *declarative* and *endpoint-centric* manner.

ACM Reference Format:
Sarah McClure, Sylvia Ratnasamy, Deepak Bansal, and Jitendra Padhye. 2021. Rethinking Networking Abstractions for Cloud Tenants. In *Workshop on Hot Topics in Operating Systems (HotOS '21), May 31-June 2, 2021, Ann Arbor, MI, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3458336.3465303

1 INTRODUCTION

A growing number of enterprises have workloads that span multiple regions within a cloud, multiple clouds, and private on-premises datacenters. For example, a recent survey reports that over 88% of surveyed enterprises use two or more cloud providers and over 92% have both public and private cloud deployments [24]. In such scenarios, the tenant must network their workloads – *e.g.,* connecting their Spark cluster on Azure to their database on AWS, and their on-prem alert management system. Today, tenants achieve this by building one or more virtual networks using the abstractions provided by the cloud providers (e.g. VPCs in AWS, VNets in Azure) and interconnecting these virtual networks to their on-premises network and to their networks in other clouds.

For tenants such as the above, building a virtual network is ad hoc, complex, and ultimately expensive. The underlying problem is that the networking abstractions available to tenants are essentially virtualized versions of the low-level building blocks used to build physical networks: virtual gateways, routers, appliances, links, *etc.* A simple virtual network may be sufficient for some tenant use cases. However, when networking larger deployments, tenants have no choice but to construct a complex virtual network from these low-level building blocks. With additional performance or security requirements, a range of appliances may be necessary. Further, as their infrastructure spans multiple clouds and on-premises datacenters, tenants face inter-domain technologies such as BGP. Compounding all of the above, each cloud exposes slightly different versions of these low-level abstractions, provisioned and configured uniquely.

The number of virtual components and their complex configuration amounts to an undue burden on the tenant. Construction and management of a virtual network requires detailed knowledge of low-level networking concepts and sophisticated reasoning about network scaling, availability, and security. Overall, this means that despite the simplicity the cloud promises, tenants must have the skill set of a seasoned network operator.

We believe that it is time to rethink tenant networking with the goal of simplification. We conjecture that the goals currently met by a tenant's virtual network can instead be achieved with the right support from the tenant's application layer and the cloud provider's networking layer. Further, we argue that this cloud support should not be provided through a bloated set of low-level abstractions as it is today and, in fact, that tenants should not be required to construct and manage a virtual network at all.

Instead, cloud providers should support a more *declarative* approach in which a tenant specifies a set of "connectivity parameters" associated with each compute *endpoint*. E.g., specifying the desired level of QoS, isolation, and availability associated with compute instances.

HotOS '21, May 31-June 2, 2021, Ann Arbor, MI, USA
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8438-4/21/05.
https://doi.org/10.1145/3458336.3465303

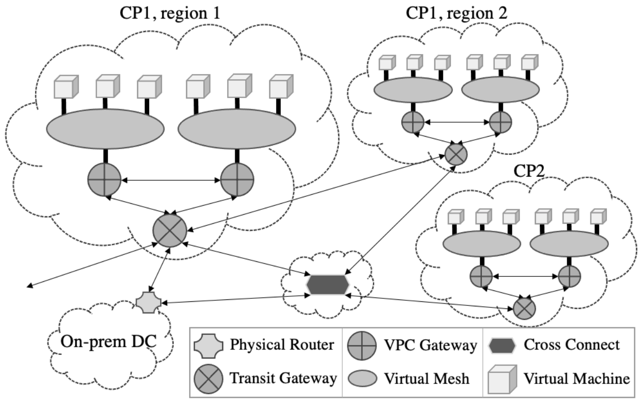

Figure 1: A tenant's virtual network spanning multiple cloud providers, regions, and an on-prem datacenter.

This declarative form benefits both providers and tenants. Tenants specify their desired level of service without worrying about *how* it is achieved while cloud providers are free to engineer their networks as they choose to meet these goals. In addition to being streamlined, this approach provides a more uniform interface to multiple clouds while still allowing providers to differentiate through rich performance, availability, and security tiers.

Notably, this architecture change is forward-looking, but not clean-slate. In most cases, the cloud providers would only need to change the tenant's *interface* to their network. The underlying virtualization techniques may remain the same and evolve independently. Not all cloud tenants may initially embrace this approach. However, just as with initial cloud adoption, we believe that a few low-risk tenants/workloads may be willing to confront the initial uncertainty for the promise of greater simplicity.

In the rest of the paper, we motivate the need to revisit tenant networking (§2-3) and present an outline of our proposal (§4-6) which we present as a starting point for a broader discussion on how to mitigate complexity in tenant networking.

2 TENANT NETWORKING TODAY

We elaborate on the complexity that tenants face in networking their workloads. For simplicity, we use the terminology of a specific cloud provider (AWS) though most of the abstractions we discuss have equivalents in other clouds. Therefore, we believe our discussion is sufficiently representative of a tenant's general experience.

We consider an enterprise tenant whose workloads span multiple regions within a cloud, multiple clouds, and an on-prem deployment as shown in Fig. 1. These workloads are backend applications such as data analytics, storage, *etc.* At a high level, constructing these networks involves five steps.

(1) Creating individual virtual networks (VPCs). The basic construct in a tenant's network is a *Virtual Private Cloud* (VPC) 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 VPCs, involves assigning IP prefixes to each VPC 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 VPCs becomes challenging, prompting AWS to recommend special address planner tools [35]. Beyond address management, defining a VPC involves configuring a range of additional parameters such as security groups, ACLs, route tables, and individual VM interfaces (Table 1). 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 a VPC. Next, the tenant must define how instances within a VPC access resources outside the VPC. Options include setting up an "Internet Gateway (IGW)" (connecting the VPC to the public Internet via IPv4), an "Egress-only IGW" (for connectivity via IPv6), or a "Virtual Private Gateway (VPG)" (for private VPN connectivity to an on-prem datacenter). Finally, a tenant might also need a "NAT Gateway" for address translation. Each of the above gateways must be configured with the appropriate routing and access policies.

(3) Networking multiple VPCs. The tenant's next task is to connect multiple VPCs both within and across multiple cloud providers as well as to any on-prem or branch locations. Once again, the tenant must navigate multiple building blocks to achieve this. A tenant may create a private "VPC peering connection" between two VPCs within a single provider. However, to interconnect VPCs across clouds and on-prem, the tenant uses a "Transit Gateway (TGW)". The TGW interconnects different networks much like a BGP or border router in physical networks. Because TGWs are limited to a particular region, the tenant may need to deploy and network multiple TGWs (Fig 1). Additional options exist for tenants with more advanced peering goals [30]. Again, each option comes with an associated set of configuration knobs; *e.g.,* Table 1 lists a subset of configuration options for the TGW.

(4) Specialized connections. An increasingly common component in large tenant networks are dedicated connections that promise high availability and QoS. E.g., Direct Connect provides a dedicated connection between a port on an AWS gateway and a port on a router outside AWS, typically at an exchange location such as Equinix [28]. This helps to construct a predictable and secure link between AWS and deployments in other clouds or on-prem. E.g., the exchange router at Equinix might connect to a dedicated connection (called ExpressRoute) to an Azure deployment and/or an MPLS connection

Abstraction	Cloud	Options	Features	Configuration Parameters (incomplete list)
Load Balancer	AWS	Application Load Balancer	L7 load balancing	Path, host, and header conditions; health checks; target groups
		Network Load Balancer	L4 load balancing	Addressing; availability zones; health checks; target groups
		Classic Load Balancer	L4 & L7 load balancing	Rules; availability zones; health checks; target groups
		Gateway Load Balancer	L3 load balancing	Rules; appliances; VPCs
Virtual Network	AWS	VPC	Isolated virtual network	Addresses; route tables; security groups; appliances; peerings
Gateway	AWS	Transit Gateway	VPC to on-prem connection	Route tables; attachments; route propagation; MTU

Table 1: Sample of virtual network components. [27, 32, 33]

to an on-prem location (Fig. 1). 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 virtualized appliances such as load balancers and firewalls. And again, 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.*). We see many issues with this long and involved provisioning process, as we elaborate in the following section.

3 COMPLEXITY IN TENANT NETWORKING

We briefly summarize the problems we see in today's tenant networking solutions.

(1) Abstractions that are too low-level. Given low-level abstractions (links, subnets, routers), tenants have no choice but to take on the various tasks associated with assembling these components (*e.g.,* topology planning, route selection, per-device configuration) and have little support in reasoning about how their overall network design meets end-to-end application goals.

(2) Complex planning. Not only does the tenant have many components to select, each component comes with diverse options in terms of features and pricing. For example, Azure offers four load balancer options and the documentation [19] that guides tenants on which load balancer to leverage involves a decision tree that is five levels deep!

A further complication is that, for many appliances, a tenant may choose between the cloud provider's native implementation and that offered by third-party vendors, and/or whether to consume it as an appliance vs. a managed service (*e.g.,* [34]). Vendor appliances promise consistency across clouds but lack first-party support.

(3) Complex configuration. Once selected, each component requires configuration which is a complex and error-prone process and, despite many years trying, we still lack a uniform configuration management solution [4, 7, 22]. Further, network configuration is decoupled from the applications the network serves, requiring coordination between network operators and app developers.

(4) Fragmented across clouds. With multiple cloud deployments, complexity worsens since each cloud has slightly different abstractions for similar functions. Hence as enterprises onboard to more than one cloud, they effectively maintain a different tenant network layer for each cloud. This siloed structure may be manageable with minimal or high-level abstractions, but this is not the case today.

(5) Complex to maintain and evolve. Given the diversity of options, different tenants' networks often look vastly different [18]. This high degree of customization complicates maintenance as each tenant must independently develop diagnostic and upgrade procedures suited to their design. Moreover, as cloud APIs are constantly evolving, a tenant must independently assess how these changes impact their current design. The lack of a uniform approach is also undesirable for cloud providers as supporting tenants becomes more difficult, particularly on issues that involve deployments outside their own cloud.

How do tenants handle this complexity today? 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, VPNs, firewall configuration, *etc.* Even worse, these teams must understand multiple cloud environments, which change constantly and outside of their control.

In addition, some tenants are turning to a new generation of multi-cloud management solutions [1, 3, 36, 37]. 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 [3, 36] but do not fundamentally change the level of abstraction. Yet others essentially run a tenant network as a service [1, 37], allowing tenants to completely outsource the problem. This shifts the burden but does not fundamentally solve it.

How did we get here and how do we move forward? Network virtualization technologies were originally designed to allow cloud providers to virtualize their *physical* network infrastructure [9, 12]. 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 believe this approach is neither desirable nor necessary as tenants embrace the cloud more fully in terms of 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 their own virtual networks for reasons that range from satisfying compliance requirements (*e.g.,* with data protection laws [21]), to familiarity with existing tools, and the perception of greater security (see §4). 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. 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. Further, by offering unmanageable complexity, they give up some portion of the market to a cottage industry of new vendors that build on top on their infrastructure and consequently lose the direct relationship with their customers. Finally, the cloud provider could likely achieve higher resource efficiency by taking control of networking and orchestration from tenants.

4 TOWARDS A BETTER APPROACH

We propose an alternate architecture that meets tenant goals without requiring them to construct and operate complex virtual networks. We have found that customers' network goals broadly fall into 4 categories: (1) connectivity, (2) availability, (3) security, and (4) QoS. Overall, our proposal is to eliminate the tenant networking layer and replace it with an API which allows customers to declaratively specify the above goals for their deployment. All of the functionality previously provided by the tenant networking layer will be provided by a combination of the tenant's application layer and the cloud provider's networking layer. Further, we seek to design an API which makes intra and inter-cloud communication simple without requiring cooperation between the different administrative domains. To achieve this goal, we make two assumptions:

(1) Service-centric application designs. We assume that tenants' applications follow a *service-centric* design in which clients access application functionality via well-defined APIs. All accesses (including management related) are first routed to an *API gateway* which verifies the client's access credentials and that the API call is well-formed [2, 26]. This is the well-known microservices design paradigm used by Kubernetes-based applications, among others [8]. We recognize that not all applications follow this design pattern and hence our approach will initially be limited to workloads that do.

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
`set_qos(region, bandwidth)`	Sets region BW allowance

Table 2: Proposed cloud tenant network API.

(2) All tenant instances have "public but default-off" IP addresses. We assume that each tenant VM/container has a globally routable IP address. However, once the cloud provider receives the packet for such an address, it will enforce a default-off policy in which only sources that are explicitly enumerated in a tenant-provided permit-list (described below) are allowed to proceed through the provider's network to the destination VM/container. We recognize that this raises security concerns and discuss this later in the paper. However, this design choice is key to making communication across clouds and on-prem datacenters trivial.

We now describe our proposed API, by considering each tenant goal. Our initial aim is to develop an API that enables functionality that is equivalent to what tenants achieve today via their virtual networks. More advanced functionality can be achieved through future extensions.

Connectivity. Connectivity between the tenant's VMs in the same cloud, across clouds, and to their on-prem network 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 (see Table 2). 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 cloud providers the maximum flexibility in assigning addresses from their overall pool (*e.g.,* to maximize the ability to aggregate for routing, *etc.*) and should not affect tenants in any way (since tenants are no longer configuring routing with mechanisms such as BGP).

By giving all endpoints publicly routable IP addresses, we can abandon the VPC and network appliance abstractions altogether. Accordingly, tenants are not obligated to construct the networks to facilitate communication outside of a given VPC, as is required by the inherent isolation of the VPC abstraction. One VPC feature not addressed here is its function as a grouping mechanism, though this could easily be built into our API as an extension or as a part of the compute interface.

Availability. 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. For this, we allow tenants to request a SIP and introduce a *bind* API that allows tenants to associate EIPs with a SIP (Table 2). 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).

Security. Today, tenant services are protected by a combination of application and network-layer defenses such as authorization checks in the application code, private address spaces, router ACLs, and complex DPI firewalls. We conjecture that an equivalent level of security can be achieved via: (i) mandatory API-level authentication and access control as implemented by service-centric applications (our assumption above), and (ii) in-network access control implemented by cloud providers using permit-lists provided by tenants. API-level checks enforce application semantics while the permit-list guards against network resource-exhaustion attacks such as DDoS. We do not support custom middlebox appliances – e.g. deep-packet inspection firewalls.

Technologies such as Kubernetes and service meshes have already made it commonplace to construct and enforce these API-level checks (*i.e.,* at the API gateway) [8]. To achieve (ii), we extend our API to allow the tenant to communicate a permit-list to the cloud provider. Mechanisms for in-network access control are already seen in cloud DDoS prevention options [20, 31]. We're adding the explicit communication of an access list from the tenant to the cloud provider using the API in Table 2; we conjecture this is feasible for the backend workloads we target since tenants already limit access to these via higher-layer credentials.

One might view our proposal as violating the common adherence to defense in depth, however, we believe this is open to debate. In particular, we argue that authentication and access control is better done at the layer which understands application semantics (rather than through packet inspection in the network) and that the widespread adoption of technologies like Kubernetes have given us a de-facto architectural template for how to implement API-level security. Further, our approach does not strictly preclude defense in depth as the cloud provider may implement the customer-specified access control list in a distributed and redundant manner.

QoS. We seek to meet the QoS capabilities of today's abstractions with the possibility of further extensions not currently supported. QoS guarantees offered to tenants today are rather limited. Most cloud providers (and most ISPs) do not currently offer any guarantees on latency or jitter, nor do they offer end-to-end guarantees on bandwidth. For individual VMs, they typically offer an egress bandwidth guarantee allowing a VM to send up to specified limit before it will be throttled. We adopt this with no changes.

For traffic exiting the cloud, a tenant may choose how the traffic is transported based on a so-called hot versus cold potato model [29]. In the hot potato case, the traffic leaves the cloud provider's WAN backbone as soon as possible. For customers seeking better performance, the cold potato model keeps traffic on the cloud backbone for as long as possible before exiting close to the destination. This higher-tier service model generally results in better performance and availability, though no hard guarantees are offered. Once again, we adopt this option unchanged.

We do not support the dedicated links mentioned in §2 in our model, as we do not want the tenants to deal with low-level networking artifacts required to provision and steer traffic onto such links. Instead, we provide an API that the customer can use to request total egress bandwidth from the cloud provider at some pre-defined granularity (*e.g.,* regional).[1] This guarantee comes into play after hot or cold potato routing has been applied, as per the customer's profile.

While this model does not include the stronger guarantees of dedicated connections, we conjecture that we can approximate them. With egress bandwidth guarantees from different cloud providers overlapping at internet exchange points with an MPLS link to on-prem, tenants may receive comparable performance to dedicated connections terminating at the same exchange point. Of course, evaluation is necessary to determine if this simplified version is a reasonable approximation. We leave this to future work.

Summary: Table 2 summarizes our proposed API. Ultimately, this API allows tenants to specify their high-level goals without having to design the virtual network to realize them. The complex details of network management are shifted from a burden on the tenant to an obligation of the cloud provider and application design. In provisioning resources through specification of connectivity, security, availability, and QoS goals, the API is essentially associating SLOs with endpoints. We see this as a natural extension to what cloud providers already offer in compute and storage, and cloud providers can innovate below this interface without developing tenant-layer abstractions for every possible feature.

5 DISCUSSION

In our proposed API, we adopt an endpoint-centric view of tenant deployments. Our "globally-routable but default-off" addressing design was chosen to free tenants from having to

[1] Extensions might allow the tenant to indicate what portions of their traffic should consume this reserved bandwidth; we leave developing such extensions to future work.

construct and maintain virtual networks. Importantly, it does so in a manner that works within and across multiple cloud deployments and does so without requiring new protocols or cross-provider cooperation.

Our notion of *declarative* is different from prior work on declarative networking [6, 10, 15–17]: broadly, their aim is to enable declarative specifications of the protocols running within a network (*e.g.,* routing algorithms) while our goal is to abstract the network internals away entirely and instead allow users to directly express their end-to-end goals for performance, availability, and so forth.

There is a significant amount of prior work in distributed rate limiting within a datacenter [5, 13] and traffic engineering across cloud WANs [11, 12, 14] that suggest that our QoS proposal is feasible. However, our QoS API depends on the combination of these two areas: enforcing bandwidth guarantees per tenant beyond the scope of a singular datacenter. Prior work in this area [23, 25] does not explicitly assume that each tenant will be guaranteed a minimum egress bandwidth as our model does nor does it claim to approximate dedicated links. Accordingly, our QoS proposal requires further research as explained in §6.

We have created an initial prototype of our API on top of existing cloud APIs for both Azure and AWS. For now, the QoS API has been excluded since we introduce new functionality not currently offered (bandwidth guarantees). We have found the API can construct our target class of applications (service-based architectures) easily, requiring minimal changes to move deployments across clouds. Further, the tenant sees many fewer network "boxes" and does not have to deal with the complexity of constructing their network. We leave a full implementation and evaluation of the API to future work.

Applying our API to the example deployment in Fig. 1, the tenant will no longer have to consider any of the 6 VPCs or 9 gateways in the original topology, only the endpoints themselves. A more developed version of this example would likely include firewalls and other network appliances which would also be excluded using our proposed API. The tenant must only interact with each instance, requesting addresses, setting permit lists, *etc.* as needed. Further, any migration between clouds will become incredibly simple as the basic interface will be constant between clouds (though extension parameters for each implementation may vary).

6 OPEN QUESTIONS

The reader may be wondering whether the architecture we propose is *simple* or *simplistic*? There are two aspects to this: (1) whether our architecture is technically feasible, and (2) whether enterprise tenants will adopt this approach.
Questions regarding the feasibility include:

(i) Scalability: Does our assumption that all endpoints are given a publicly routable address scale in terms of the size of routing tables within a cloud provider? Does a (dynamic) shared permit-list between tenants and cloud providers scale? Can egress bandwidth quotas be scalably enforced? These questions can be quantitatively answered given the appropriate data traces; e.g., with traces that include launch/teardown times for tenant instances, per-instance communication patterns, *etc.*

(ii) QoS: We proposed an approach based on the combination of potato parameters and per-tenant quotas applied on a per-region basis. Does this approach offer application performance that is comparable to that achieved with the use of dedicated links? With cooperation from cloud providers, this is a question that we can experimentally evaluate: *e.g.,* setting up deployments and measuring application performance with each approach.

(iii) Security: We proposed an approach based on the combination of network-layer enforcement of L3/L4 permit-lists and API-level access control. Does this provide sufficient security? Or, at least, security on par with today's approach to building private networks? This is perhaps the hardest answer to quantify. Possible approaches include a formal security analysis of the surface area of attacks for each approach, the feasibility of applying verification techniques to kubernetes API servers, and/or evaluating each approach through the lens of past attacks.

We are not claiming that the answers to the above are easy or obvious. However, with the right data and access, we believe we can evaluate the feasibility of our proposal.

The second part of the 'simple or simplistic' question is whether it is plausible that enterprises will adopt our approach? Specifically, will enterprises be comfortable dialing back the defense-in-depth mindset that leads them to isolate their workloads within (virtual) private networks? We do not claim to know the answer and only note that our proposed solution can co-exist with existing architectures allowing tenants to incrementally opt-in for whatever workloads they deem appropriate at that time. This is, in fact, not unlike how enterprises embraced cloud computing, under very similar concerns. Hence, the question to answer is: what tenants and workloads are the most likely early adopters for a new architecture such as we propose?

Finally, one should question whether our proposed approach is in fact the right one and what alternative solutions we should be considering. Further, one may ask if our solution is sufficiently high-level. E.g., could we abstract above details such as IP addresses entirely? We hope to discuss these and other questions with the broader community.

REFERENCES

[1] Alkira. 2020. Alkira. Retrieved May 13, 2021 from https://www.alkira.com/

[2] The Kubernetes Authors. 2021. Ingress Controllers. Retrieved May 13, 2021 from https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/

[3] Aviatrix. 2021. Aviatrix. Retrieved May 13, 2021 from https://aviatrix.com/

[4] M. Bjorklund. 2010. *YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)*. RFC 6020. RFC Editor. 1–173 pages. http://www.rfc-editor.org/rfc/rfc6020.txt

[5] Mosharaf Chowdhury, Zhenhua Liu, Ali Ghodsi, and Ion Stoica. 2016. HUG: Multi-Resource Fairness for Correlated and Elastic Demands. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA, 407–424. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/chowdhury

[6] David Chu, Lucian Popa, Arsalan Tavakoli, Joseph M. Hellerstein, Philip Levis, Scott Shenker, and Ion Stoica. 2007. The Design and Implementation of a Declarative Sensor Network System *(SenSys '07)*. Association for Computing Machinery, New York, NY, USA, 175–188. https://doi.org/10.1145/1322263.1322281

[7] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman. 2011. *Network Configuration Protocol (NETCONF)*. RFC 6241. RFC Editor. 1–113 pages. http://www.rfc-editor.org/rfc/rfc6241.txt

[8] Jennifer Lin Eric Brewer. 2019. Application modernization and the decoupling of infrastructure services and teams. https://services.google.com/fh/files/blogs/anthos_white_paper.pdf

[9] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. 2011. VL2: A Scalable and Flexible Data Center Network. *Commun. ACM* 54, 3 (March 2011), 95–104. https://doi.org/10.1145/1897852.1897877

[10] Timothy L. Hinrichs, Natasha S. Gude, Martin Casado, John C. Mitchell, and Scott Shenker. 2009. Practical Declarative Network Management. In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking* (Barcelona, Spain) *(WREN '09)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/1592681.1592683

[11] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. 2013. Achieving High Utilization with Software-Driven WAN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (Hong Kong, China) *(SIGCOMM '13)*. Association for Computing Machinery, New York, NY, USA, 15–26. https://doi.org/10.1145/2486001.2486012

[12] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2013. B4: Experience with a Globally-Deployed Software Defined Wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (Hong Kong, China) *(SIGCOMM '13)*. Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/2486001.2486019

[13] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Albert Greenberg, and Changhoon Kim. 2013. EyeQ: Practical Network Performance Isolation at the Edge. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX Association, Lombard, IL, 297–311. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/jeyakumar

[14] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C. Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, Mathieu Robin, Aspi Siganporia, Stephen Stuart, and Amin Vahdat. 2015. BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (London, United Kingdom) *(SIGCOMM '15)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/2785956.2787478

[15] Changbin Liu, Yun Mao, Mihai Oprea, Prithwish Basu, and Boon Thau Loo. 2008. A Declarative Perspective on Adaptive Manet Routing. In *Proceedings of the ACM Workshop on Programmable Routers for Extensible Services of Tomorrow* (Seattle, WA, USA) *(PRESTO '08)*. Association for Computing Machinery, New York, NY, USA, 63–68. https://doi.org/10.1145/1397718.1397733

[16] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. 2009. Declarative Networking. *Commun. ACM* 52, 11 (Nov. 2009), 87–95. https://doi.org/10.1145/1592761.1592785

[17] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. 2005. Implementing Declarative Overlays. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom) *(SOSP '05)*. Association for Computing Machinery, New York, NY, USA, 75–90. https://doi.org/10.1145/1095810.1095818

[18] Microsoft. 2018. Cloud Design Patterns. Retrieved May 13, 2021 from https://docs.microsoft.com/en-us/azure/architecture/patterns/

[19] Microsoft. 2019. Overview of load-balancing options in Azure. Retrieved May 13, 2021 from https://docs.microsoft.com/en-us/azure/architecture/guide/technology-choices/load-balancing-overview

[20] Microsoft. 2020. Azure DDoS Protection Standard overview. Retrieved May 13, 2021 from https://docs.microsoft.com/en-us/azure/ddos-protection/ddos-protection-overview

[21] U.S. Department of Health and Human Services Office for Civil Rights. 2013. HIPAA Administrative Simplification.

[22] OpenConfig. 2016. OpenConfig. Retrieved May 13, 2021 from https://www.openconfig.net/

[23] Barath Raghavan, Kashi Vishwanath, Sriram Ramabhadran, Kenneth Yocum, and Alex C. Snoeren. 2007. Cloud Control with Distributed Rate Limiting. *SIGCOMM Comput. Commun. Rev.* 37, 4 (Aug. 2007), 337–348. https://doi.org/10.1145/1282427.1282419

[24] David Ramel. 2019. Research Brief Summarizes Trends in Multi-Cloud Deployments. https://virtualizationreview.com/articles/2019/10/21/cloud-trends.aspx

[25] Ahmed Saeed, Nandita Dukkipati, Valas Valancius, Terry Lam, Carlo Contavalli, and Amin Vahdat. 2017. Carousel: Scalable Traffic Shaping at End-Hosts. In *ACM SIGCOMM 2017*.

[26] Amazon Web Services. 2021. Amazon API Gateway. Retrieved May 13, 2021 from https://aws.amazon.com/api-gateway/

[27] Amazon Web Services. 2021. Amazon Virtual Private Cloud. Retrieved May 13, 2021 from https://aws.amazon.com/vpc/

[28] Amazon Web Services. 2021. AWS Direct Connect. Retrieved May 13, 2021 from https://aws.amazon.com/directconnect/

[29] Amazon Web Services. 2021. AWS Global Accelerator. Retrieved May 13, 2021 from https://aws.amazon.com/global-accelerator/

[30] Amazon Web Services. 2021. AWS Global Transit Network. Retrieved May 13, 2021 from https://aws.amazon.com/solutions/implementations/aws-global-transit-network/

[31] Amazon Web Services. 2021. AWS Shield. Retrieved May 13, 2021 from https://aws.amazon.com/shield/

[32] Amazon Web Services. 2021. AWS Transit Gateway. Retrieved May 13, 2021 from https://aws.amazon.com/transit-gateway/

[33] Amazon Web Services. 2021. Load balancer types. Retrieved May 13, 2021 from https://docs.aws.amazon.com/AmazonECS/latest/

developerguide/load-balancer-types.html

[34] Amazon Web Services. 2021. NAT. Retrieved May 13, 2021 from https://docs.aws.amazon.com/vpc/latest/userguide/vpc-nat.html

[35] Amazon Web Services. 2021. VPCs and Subnets. Retrieved May 13, 2021 from https://docs.aws.amazon.com/vpc/latest/userguide/VPC_Subnets.html

[36] VMWare. 2021. Multi Cloud Operations: Visibility & Control. Retrieved May 13, 2021 from https://www.vmware.com/cloud-solutions/multi-cloud-ops.html

[37] Volterra. 2021. Volterra. Retrieved May 13, 2021 from https://www.volterra.io/