

ClusterLink: Redefining Application Connectivity for the Multi-cloud Era

Kfir Toledo, Pravein Govindan Kannan, Michal Malka,
Etai Lev-Ran, Or Ozeri, Vita Bortnikov, Ziv Nevo, Kathy Barabash
IBM Research
{kfir.toledo, pravein.govindan.kannan, michal.malka}@ibm.com,
{etai, oro, vita, nevo, kathy}@il.ibm.com

Abstract—Modern software development abstracts applications from the underlying infrastructure, enabling global-scale deployment with minimal concern about low-level networking details. However, when these infrastructure-agnostic software components need to communicate, they encounter significant networking limitations. This forces developers to either navigate complex, low-level networking constructs to achieve the desired connectivity or give up on truly flexible connectivity and limit their software to static connectivity patterns.

In this paper, we focus on the evolving challenges of application connectivity in today’s hyper-distributed reality. We propose to model connectivity around the notion of application services and have realized this proposal as ClusterLink, which exposes the app-level APIs for specifying communication policies at a very granular level and implements them efficiently. This paper shares details on ClusterLink design principles, APIs, architecture, and implementation, and shows that ClusterLink outperforms its closest competitor by 2.5x in throughput in a cloud-based experimental setting.

I. INTRODUCTION

The modern world is inconceivable without access to digital services. Organizations of any size, structure, and purpose, individuals of almost any age, and even devices today depend on data and services available over the network. This became possible due to a series of significant technological advancements over the past two decades. We have witnessed the shift from monolithic single-purpose applications running on dedicated servers in isolated data centers to elastic groups of interconnected micro-services deployed in virtualized environments. Virtualization technologies such as VMs, containers, and ephemeral functions allow separating the software from the hardware it runs on. Moreover, they allowed hardware to be shared among several independent applications. Cloud providers harnessed this opportunity and created consolidated infrastructures and advanced orchestration software to enable today’s seemingly effortless way of creating, consuming, and operating digital services.

At the same time, the evolution of networking services has not been as rapid as that of computing services, and there are many good reasons for that. Networking has been advancing piece-wise. In some cases, the networking community has dared to innovate and achieved network virtualization comparable to that of the compute, e.g., in Cloud Networking. In some other cases, evolution came as a reaction to acute needs

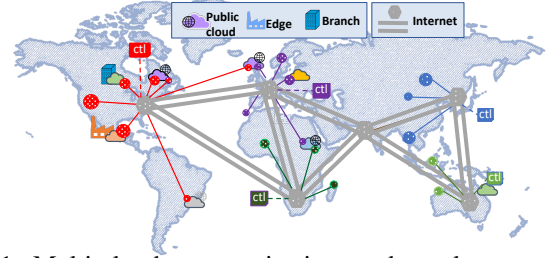


Fig. 1: Multi-cloud communications today rely on multiple disjoint network islands, both virtualized and legacy.

as happened, for example, with the software-defined wide area network (SD-WAN) that came to free the multi-branch enterprises from their dependency on rigid and costly leased-lines services. Unfortunately, even if each individual network virtualization solution, deployed as a self-contained network domain, is solid and supports sufficient levels of abstraction and control over intra-domain communications, cross-domain communications inevitably must traverse the not yet virtualized networking areas, be it legacy organizational backbones, last-mile access networks, or the ubiquitous public Internet. This situation hinders the benefits of compute virtualization from being harvested across deployment areas with different networking technologies. Even in rare cases where individual environments are supported by proper network virtualization and are interconnected with virtualized WAN technology, such as SD-WAN[1], [2], [3], [4] or multi-cloud connectivity solutions[5], [6], the problem of stitching different virtualized networks into coherent connectivity environments persists.

Fig. 1 illustrates how the inevitable use of different network virtualization islands increases the overall complexity and administrative overheads of the large-scale deployments that encompass areas with different network virtualization technologies as well as areas where only legacy physical networking is present. In practice, this results in a tendency to install static and overly limiting controls over application connectivity, contradicting the need for intensive and dynamic communications between individual components. Most prevalent although trivial examples of cross-deployment communications are: client access to software services hosted in

the cloud, public or private; streaming data, either business or monitoring data, produced at individual deployment areas for further processing or storage; DevOps teams accessing their software deployment for performing management and maintenance tasks, etc. More advanced examples involve a direct exchange of data and commands between applications and services deployed in different clouds, private or public, as well as in different on-prem locations, be it data centers, branch offices, or Edge.

Solutions like VPNs [7], [8], [9] and multi-cloud backbones [5], [6] that exist to take care of some cross-deployment communication use cases operate at low levels of abstraction, such as IP subnets, routing rules, etc., and are not easily extensible to support the end-to-end app-level connectivity needs. On the other side of the spectrum, solutions like Cilium [10], Istio [11], Skupper [12], etc., while solving the immediate pain of app-level connectivity, tend to be coupled to their application environments, are not always transparent, are not inter-operable, and would not scale to support generic workloads deployed across different domains.

To overcome these challenges, we introduce ClusterLink, a lightweight and efficient multi-cloud connectivity solution. ClusterLink enables high-level, network-independent connectivity specifications while efficiently leveraging underlying network services, whether physical or virtual. We share our experience realizing ClusterLink and show how it helps to simplify multi-cloud application deployments by decoupling the app-level connectivity from the underlying networking, without imposing restrictions on how applications are deployed or on how the infrastructure networks are operated.

Our main contributions in this paper are as follows:

- 1) First, we share a comprehensive analysis of the multi-cloud application connectivity problem and articulate a new approach to address it by designing a high-level connectivity abstraction applicable irrespective of the deployment type, application type, location, etc.
- 2) Second, we contribute a set of generic design principles the solution must possess.
- 3) Third, we present a specific realization of the proposed abstraction, called ClusterLink, that follows the stated design principles for a very prominent case of cloud-native workloads [13] communicating across Kubernetes (K8s)[14] clusters.
- 4) We show how we designed and implemented ClusterLink, a programmable building block for dynamic provisioning of application connectivity across multiple different remote clusters while enforcing high-level policy controls.
- 5) We demonstrate how ClusterLink has been validated and adopted in a broad range of real-world use cases, including an application management platform, an LLM serving platform, and a data protection management platform.
- 6) We evaluate ClusterLink in a cloud-based environment, measuring latency, throughput, and scalability, and show that it outperforms alternative solutions.
- 7) Finally, ClusterLink is open-source and is available online [15].

The rest of this paper is organized as follows: §II defines the multi-cloud application connectivity problem, discusses the collected requirements, and outlines the design space and the choices we have made; §III summarizes the related work; §IV presents the solution architecture, focusing on crucial componentization and major interfaces; §V presents ClusterLink implementation and deployment for Kubernetes clusters; §VI shows how ClusterLink was validated for the identified real-life use cases; §VII presents the evaluation results; §VIII discusses future directions and plans; and finally §IX concludes the paper.

II. THE APPLICATION CONNECTIVITY PROBLEM

The main focus of this work is the need for end-to-end, secure, and controlled connectivity between application components and services deployed in different administrative, technology, and geographical domains.

In this section, we focus on analyzing the application connectivity problem from a top-down perspective, discuss the requirements, and present a feature set that any adequate solution must possess. For each of the above, we start from generic considerations followed by more specific aspects related to cloud-native applications communicating across Kubernetes clusters.

A. Existing Challenges

The main problem with cross-platform application connectivity is the drastically different perspectives of the involved stakeholders, their roles, responsibilities, processes, etc. On the one hand, people and teams responsible for the networking infrastructure, e.g., network administrators, CloudOps teams, data center operators, etc., need their systems to be as stable and reliable as possible, work under strict controls, and must follow slow, rigid processes. As a result, infrastructure networking tends to remain static and change on relatively long cycles. On the other hand, teams responsible for software services such as Site Reliability Engineering (SRE), DevOps, application architects, developers, and end users who access digital services, strive for flexibility and versatility. Unfortunately for the latter group, provisioning application connectivity involves multiple teams and processes such as compliance, identity, asset protection, etc., while undergoing scrutiny with regard to the organization's policies and costs.

Having analyzed multiple cases featuring such problems, we have identified the following major gaps:

Lack of adequate uniform abstraction Today, for an enterprise to connect applications running in a private cloud to services deployed in a public cloud like Amazon Web Services (AWS), Google Cloud Platform (GCP), or Azure, one must deploy and configure multiple network services, e.g., VPN gateways [8], [9], [7]. There are multiple problems with this approach. First, the resulting deployment is very tightly coupled with network provisioning and configuration. Second, the abstractions exposed by cloud providers and other vendors are too low-level when it comes to connectivity and often enforce users to express their needs in terms of IP addresses,

subnet ranges, routing tables, firewall rules, etc. Third, these abstractions are inconsistent across solutions and require users to acquire new skills when working with new providers. When there was a need to encourage organizations to migrate from their private environments to clouds, the decision to expose abstractions familiar to network administrators was very reasonable. However, now it is time to revisit this decision to allow for further evolution[16].

Lack of interoperability Although many solutions have been created to address the problem of inconsistent networking abstractions, it turned out that the problem is deeper than just an interface level. As far as wide-area connectivity is concerned, there's a need to consider not only cloud deployments but also legacy non-virtualized deployments. As a result, all the solutions in this space [5], Alkira [6], Submariner [17] are realized as private L3 overlays, not interoperable with each other. The same is true for application-level connectivity solutions like Cilium Mesh [10], Istio gateway [11], Skupper [12] that solve connectivity between environments of the same type and not across the types. Moreover, the application connectivity and the WAN connectivity solutions operate on different levels of abstraction, and sometimes we can see application-level overlays implemented over the L3 global network overlays, increasing the overall complexity and affecting performance.

B. Requirements

To present the requirements for the solution that aims to overcome the gaps and address the need for application-level global connectivity, we introduce the following terminology:

Workloads are business applications packaged and delivered as a collection of communicating software entities, e.g., VMs, containers, processes, appliances, functions, etc. *Workload components* are entities comprising the workload. Here, we are only interested in *communicating workload components*, i.e., workload components that exchange messages and data between them, establishing communication flows. Such workload components can either receive or initiate communication flows or both. From the networking perspective, this translates to workload components being either servers or clients of the underlying connectivity service.

Locations are environments, either infrastructure or platform, that host workload components. For example, a cloud VPC is a location where workload components are deployed as VMs, while a Kubernetes cluster is a location where workload components are deployed as Pods. Other types of locations can be a group of servers in a private data center or edge point, or private clouds, such as VMware or OpenStack. Note that in some cases, locations can be nested, e.g. Kubernetes cluster deployed in a VPC or on-prem VMware cluster. We only consider the top-most environments because they are the immediate hosts of the workload components.

Administrative domain is an independent organization or person that controls the location or the workload or both. In the most general case, the solution should support connectivity across different administrative domains that can be fully autonomous with respect to their local operations, including

connectivity, access permission assignment, naming, etc. In particular, networks in different administrative domains can use different technologies and must not expect the ability to access each other's management APIs, for e.g. Kubernetes APIs or routing configurations.

Remote service is a 'server' workload component deployed in one or more participating locations and made available for access from other participating locations. *Remote client*, similar to Remote service, is a 'client' workload component deployed in one participating location and making access to 'server' workloads in other participating locations.

With the above terminology, here is the list of generic requirements, not necessarily in the order of importance:

- Allow controlled sharing of information about workload components (both the remote services and the remote clients) across locations while respecting the information ownership rules imposed by all the participating domains. Each participating domain must have full control over the scope and the granularity of information sharing and be able to revoke any shared information at all times.
- Define policy on cross-domain communication flows between the workload components, based on shared workload attributes. Policies can include communication security, access control, load balancing, observability, etc.
- Allow establishing the connections without requiring special privileges, e.g., root or administrator, in participating locations and without breaking administrative boundaries between the administrative domains.
- Ensure security so that components and their communications are protected with the specified controls. This includes enforcing a Zero Trust approach [18], which denies all communication by default and permits it only when explicitly authorized. Zero Trust assumes no implicit trust between entities, regardless of their location in the network, and requires strict identity verification and policy-based access control. It also includes implementing micro-segmentation, whereby each party in the communication independently enforces its own set of policy rules.
- Avoid dependency on specific location's networking functionality, such as a specific Container Networking Interface (CNI) plugin or service mesh in Kubernetes case (Flannel [19], Istio [11], Cilium [10], etc) or VPC in a cloud, etc.
- Minimize dependency on infrastructural connectivity between the locations, requiring as less configuration as possible, e.g., routing setups, firewall rules, etc.
- Ensure that each workload connection, i.e., each separate conversation between a pair of workload components, is distinguishable in the data plane. This is required to support sufficient levels of observability as well as fine-grained control over individual conversation, e.g., paths and devices it is forwarded through, QoS applied to it, etc.

III. RELATED WORK

As more practical use cases for multi-cloud computing are identified, including advanced research focusing on opti-

mizing distributed computations deployment across multiple locations [20], [21], [22], [23], the networking aspects of multi-cloud computing draw more attention. Although it is possible to network multi-cloud applications with the existing solutions, some serious gaps exist. The most relevant existing solutions can be categorized into three buckets.

L3 Overlays. Numerous SD-WAN and multi-cloud networking solutions created for multi-cloud infrastructure connectivity [5], [1], [4], [3], [24] as well as for platform level connectivity [17], employ overlay network virtualization and expose L3 abstractions to their users. Our approach is fundamentally different and avoids replicating the work done by the L3 devices existing in the infrastructure. Instead, we choose to operate at the session, or connection, level and to expose high-level, application-meaningful abstractions.

Service mesh solutions. There are several service mesh solutions [11], [25], [26] that try to solve the multi-cloud networking problem. In contrast to ClusterLink, these solutions attempt to create a full-service mesh between all the clusters, which can lead to scalability issues. Additionally, creating a full mesh between all the clusters is not always possible, as described in §VI-B.

Application Overlays. These solutions [12], [27] are better suited for creating self-service application connectivity but do not interoperate with the underlying networking infrastructure. As a result, they are less performant, not scalable, and unable to support all the services usually expected from the networks, such as quality of service. Skupper [12] is the most similar to ClusterLink. The abstraction it exposes does not allow for dynamic fine-grained control over connectivity policies. This capability is crucial for meeting the requirements of a Zero Trust approach and enforcing policies based on workload attributes. In addition, implementation-wise, Skupper lacks a clear separation between the control and the data planes and overly depends on Advanced Message Queuing Protocol (AMQP), both as the data and the control plane are concerned, while our solution features clean separation and supports data plane plugins. In addition, there are multi-cloud projects, e.g., Skyplane [23] and CloudPilot [28], that created connectivity aids to support effective bulk data transfer across clouds. We believe that with ClusterLink, such applications can be enhanced to support dynamic multi-cloud policy-protected connectivity as we have illustrated with the Fybrik use-case (§VI-D).

IV. CLUSTERLINK DESIGN

ClusterLink is a building block that facilitates multi-cloud connectivity for cloud applications deployed across clusters. While ClusterLink is envisioned as a general-purpose utility to connect applications running across remote clusters, we use Kubernetes as the primary deployment target for ClusterLink. ClusterLink is realized as an in-cluster gateway responsible for handling application traffic according to definitions and policies specified through declarative APIs. In this section, we present important guiding principles we followed in creating ClusterLink, describe the essential data types that underlay

TABLE I: ClusterLink Management APIs

API	Description
Peer	Entity representing the remote ClusterLink gateways.
Exported Service	Local cluster service to be shared among peers.
Imported Service	Service consumed by the local cluster from other peers.
Policy	Policy information that will be applied per connection.

the internal state, the operations, and the declarative APIs of ClusterLink, and highlight the most important aspects of ClusterLink’s design.

A. Design Principles

We design ClusterLink such that it possesses the following important properties:

Programmable. ClusterLink exposes APIs that allow software architects to model the desired connectivity for their applications and services in familiar terms they use to articulate software designs. Additionally, people responsible for business-level compliance and infrastructure networking are allowed to annotate these specifications with rules and policies at their levels of responsibility.

Secure. ClusterLink ensures secure connectivity between all services and clusters, preventing use data from being exposed. Secure connectivity is crucial, as part of the multi-cluster connections may utilize the public internet.

Open & Extensible. ClusterLink is created to be an extensible building block allowing to create solutions for different use cases with different connectivity needs, using powerful connectivity and policy abstractions. Not an end-to-end solution in itself, ClusterLink also features a modular design to enable 3rd parties to replace and augment default functionality with specialized extensions. This is especially important for supporting use-case-specific policies and their implementations.

Connection Oriented. To better control connectivity at the application level while facilitating efficient collaboration with the underlying infrastructure networks, ClusterLink operates at application connection, L4 and above, and delegates the packet-level processing to where it belongs, the heavy-duty network providers.

B. ClusterLink APIs

Following the requirements outlined in §II, we identified four types of entities essential for defining communication at the application/service level (see Table I). In each ClusterLink gateway, these entities represent the system state as a whole. ClusterLink gateway sits close to the application deployed in the cluster.

We will now describe each of these entities in more detail: **Peer.** The `Peer` entities represent remote ClusterLink gateways and contain the metadata necessary for creating protected connections to these remote peers. The `Peer` API contains the public host IP or DNS name of the remote gateways and the port number. By default, it uses port 443, which allows secure connectivity through most firewalls. The remote

gateway can have more than one host address, which can be used for failover in case one of the host addresses becomes inaccessible. The ClusterLink gateway also validates that each peer is reachable by sending a heartbeat signal at regular intervals. The status of the peer, including whether it is reachable and the time of the last heartbeat, is updated in the peer's Custom Resource (CR) status.

In addition, the remote peer host address must be publicly accessible to allow other ClusterLink gateways to establish a connection. For cases where the Kubernetes cluster is behind Network Address Translation (NAT) or a firewall that does not allow incoming connections, ClusterLink can be deployed with Fast Reverse Proxy (FRP [29]), which creates a reverse tunnel to overcome this issue. This deployment is described in more detail in §VI-F.

Exported Service. The `Exported Service` entities represent application services hosted in the local cluster and exposed to remote ClusterLink gateways as `Imported Service` entities in those peers. The exported service API includes (1) the namespace where the service is located, (2) the host address of the service, and (3) the service port. ClusterLink validates the reachability of the service and maintains the status of the exported service.

The host service can exist either as an internal service within the cluster or as an external service, as long as it remains reachable from within the cluster.

Imported Service. The `Imported Service` entities represent remote application services that the gateway makes available locally to clients inside its cluster. For each imported service, the gateway creates a local listener that represents the remote service in the cluster.

The imported service API contains: (1) the name and port of the local service that represents the remote service, (2) the namespace where it will be created, and (3) details of the exported service. The exported service details include the exported service name, the exported service namespace (allowing the export of the same service from different namespaces), and the exported peer name. Each service can be imported from multiple different peers or namespaces. Additionally, there is a load-balancing scheme field that determines which load-balancing policy will be applied by the policy engine for this service (more details are provided in §IV-C).

Policy. The `Policy` entities represent communication rules that must be enforced for all cross-cluster communications at each ClusterLink gateway. While traditionally, network policies use priority (defined as an integer) to define which rules take precedence, ClusterLink proposes a simplified and intuitive policy management using just two tiers of access policies.

- The first tier is the high-priority policy. This tier is intended for cluster/peer administrators to set access rules that cannot be overridden by cluster users. High-priority policies are controlled by the Privileged Access Policy APIs and are cluster-scoped (i.e., they have no namespace).
- The second tier is the regular access policy. This tier is intended for cluster users, such as application developers

and owners. Regular policies are namespaced and have an effect only within their namespace.

For ClusterLink deployments that are not in privileged mode (within a namespace scope), only the second-tier policy API is exposed.

For a connection to be established, both the ClusterLink gateway on the client side and the gateway on the server side must allow the connection according to their policy entities.

The policy API contains the actions of the policy (allow or deny) and the attributes of the connection source and destination. To allow fine-grained policy control, we define policy control attributes at three levels:

Peer level attributes. These define attributes that describe the clusters, such as cluster name, cloud provider, and cluster location. This allows enforcing communication rules at a higher level, for example, ensuring all traffic runs on the same cloud provider or does not leave the continent, or setting failover based on regional location.

Service level attributes. These define attributes per service, such as the exported name, namespace of the service, application, or role. This allows control over connections at the service level, for instance, restricting connections to services belonging to the production environment.

Workload level attributes. These define attributes that describe the workloads, such as pod labels or service accounts, enabling pod-level controls between the application resources.

In addition, each level can have user-defined attributes to allow the policy to be extensible.

Each ClusterLink gateway maintains a set of these entities in correspondence with the declarative intent provided by the ClusterLink users, be it operators, e.g., application or infrastructure admins, or tools, e.g., application or infrastructure orchestrators. Manipulation of the state can be performed using ClusterLink management APIs. In particular, ClusterLink provides Kubernetes CRDs (Custom Resource Definition) for each type of entity. This allows users to declaratively define and configure these entities in YAML files, then apply them to the cluster using best practices for managing configuration as code. ClusterLink monitors the resources it defined and updates its internal states as they are added, modified or deleted.

We further demonstrate in §VI, how these four essential entity types allow ClusterLink to streamline the connectivity of applications deployed across different remote environments.

C. ClusterLink Gateway

ClusterLink is designed following the widely accepted Software Defined Networking (SDN) principles, where the network Control Plane is logically separated from the network Data, or forwarding, Plane, and each of these components is programmable. In principle, such separation allows the control and the data plane aspects to be realized as separate deployment units, bringing lots of benefits. However, we have identified the need for a third architectural component to encompass everything related to communication policies. We note that the policy considerations tend to be the main factor

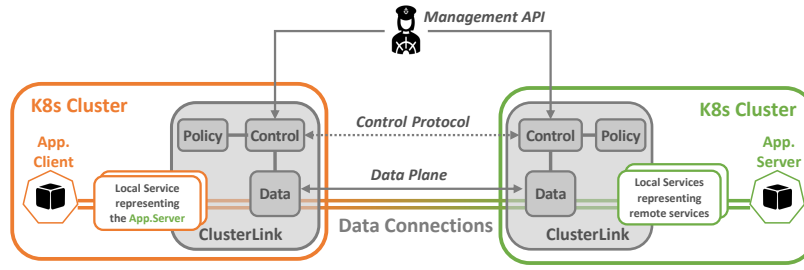


Fig. 2: ClusterLink Gateway Architecture and Interfaces

that prevents clean separation between the control and the data planes, because of the duality between the policy resolution that naturally belongs to the control plane and the policy enforcement that must often reside, at least partially, in the data plane. ClusterLink gateway architecture is thus created according to this extended SDN approach and consists of three major components, the Control Plane, the Data Plane, and the Policy Engine as presented in Fig. 2 and described at a high level below.

Control Plane is responsible for maintaining the internal state of the gateway, for all communications with the remote peer gateways (except for forwarding user traffic) by means of the ClusterLink control plane protocol (extended Kubernetes APIs), and for commanding the local data plane to forward user traffic according to policies. Internal state is manipulated externally through the APIs and internally in response to data plane events such as connection establishment, tear-down, etc. The control plane protocol plays a crucial role in negotiating connection establishment, which includes authorization of keys, creation of connection tokens, and end-to-end policy checks and enforcement. Moreover, the control plane ensures the reachability of remote peers and maintains the state within the ClusterLink fabric. In case of a peer failure, the control plane detects the unreachability through heartbeat timeouts (with detection typically occurring within a second) and orchestrates failover by updating routing decisions in the data plane. This allows new connections to be re-established with alternate peer addresses, while existing connections may briefly disconnect and retry. The control plane is programmed using the management APIs described in §IV-B, and it exports metrics to the management regarding the connections for observability.

Data Plane responds to user connection requests, both local and remote, initiates policy resolution in the control plane, and maintains the established connections. In addition to user traffic, ClusterLink data plane is also used by the control plane for its own connectivity needs, e.g., for communicating between the two remote peers upon user connection establishment and other control plane protocol communications. ClusterLink data plane relies upon standard protocols and avoids redundant encapsulations, presenting itself as a cluster-level service (in Kubernetes) inside the cluster and as a regular HTTP endpoint from outside the cluster, requiring only a single open port (HTTP/443) and leveraging HTTP endpoints for connection

multiplexing. These data plane choices allow ClusterLink data plane to transparently support common protocols such as TCP and UDP, ensure connection security with mTLS, and have good scalability properties. The control plane publishes the states regarding peer and service to the data plane, which opens up a listener for imported service and forwards connection for authorization. This state separation allows the data plane engine to be pluggable, enabling effortless integration with various standard data plane implementations.

Policy Engine is responsible for handling policy resolution requests from the control plane. The goal is to support multiple different types of policies. Today, the policy engine supports access control and load balancing. The ClusterLink policy engine uses the policy API to decide the access control for each connection as described in §IV-B. The access control policy allows users and administrators fine-grained control over which client workloads may access which services. ClusterLink is designed to be a Zero Trust system, meaning that all connections are denied by default unless an explicit policy allows them. Another mechanism applied in the policy engine is micro-segmentation, which is a basic requirement of Zero Trust systems. This approach enables security rules and boundaries for each cluster. With the use of access policies, users and organizations can enforce corporate security rules and segment the peers in the fabric into trust zones. The second policy type is the load-balancing policy. This policy is used when the exported remote service is located in multiple destinations, allowing the application to choose from which destination to consume the service. The following load-balancing schemes are implemented:

Round-Robin Load Balancing. Distributes the connections in a cyclic order between the different servers of the exported service.

Equal-Cost Multi-Path Routing (ECMP) Load Balancing. Creates uniform randomization to distribute the connections equally between the different servers of the exported service.

Static Load Balancing. Chooses a specific destination to consume the remote service from.

In addition, the policy engine is responsible for the failover mechanism. This mechanism allows applications to use a remote service only from reachable destinations. In case a remote service is not reachable, it can direct the application to use an alternative destination service as defined by the user. The policy engine is designed to be extensible, allowing users

or third-party developers to add more policies, such as rate limiting, or define their own custom policy types and plug in their implementations into the ClusterLink policy engine.

D. ClusterLink Security and Administrative Domains

ClusterLink is designed to meet enterprise management and strict security requirements as described in §II-B. With these requirements, it can allow central management, i.e., corporate IT/security engineers, to enforce corporate policy rules while still enabling users to run applications on multiple clouds and clusters seamlessly with their own control.

There are three types of administrative domains in the ClusterLink system:

Fabric Administrator - A fabric is a set of collaborating clusters, all sharing the same root of trust. Only clusters within the same fabric can enable cross-cluster connectivity. The fabric administrator is responsible for managing which clusters can join the fabric. They are also responsible for creating the fabric certificates used as the certificate authority (CA) and the peer certificates (private and public keys) for each peer.

Peer Administrator - The peer administrator is responsible for deploying ClusterLink to the cluster, selecting which peers from the fabric users can connect to, and defining privileged access policies to enforce connectivity rules at the cluster level.

Users - Users are responsible for creating cross-cluster communication. They can decide which services to export and import from the namespaces they have permissions. Additionally, users can enforce regular access rules on the application services.

In addition to the above, all ClusterLink communications follow several security principles to further strengthen our Zero Trust model.

- All cross-cluster communications, as well as internal communication between ClusterLink components (e.g., control plane and data plane), are authenticated using certificates and mTLS.
- Services need to be explicitly exported and imported to be accessible across clusters.
- All communication attempts are governed by ClusterLink's ingress and egress access control policies on each cluster (§IV-C), ensuring that connections are denied by default unless explicitly permitted.

To support secure connectivity, ClusterLink peers operate across two distinct trust domains, each managed by separate certificate authorities:

- **Inter-cluster trust domain** – used for securing communication between remote peers, typically managed by the fabric administrator.
- **Intra-cluster trust domain** – used for local communication between the control plane, data plane, and user-facing interfaces within a single cluster, typically managed by the peer administrator.

ClusterLink supports flexible certificate management: users can either bring their own certificates or allow ClusterLink to generate and manage certificates locally within the cluster's file system.

V. CLUSTERLINK IMPLEMENTATION & DEPLOYMENT

ClusterLink is implemented using Golang [30] and supports deployment across various Kubernetes flavors, including AWS, GCP, Azure, IBM Cloud, enterprise orchestrators like OpenShift, and lightweight orchestrators such as K3s. ClusterLink supports two types of data planes:

Envoy Data Plane. Envoy [31] is an open-source edge and service proxy designed for cloud-native applications. It is a highly performant and extensible proxy widely used by other projects in the industry, such as Istio [11].

Lightweight Go Data Plane. This is a lightweight Go-based proxy, allowing quick development and support for new features and requirements that are not always supported in Envoy. It doesn't contain all the features of Envoy but is more lightweight and useful for testing, development, and lightweight deployment scenarios. The control plane and data plane exchange stateful information using xDS protocol [32]. The control plane publishes cluster and listener information, and the data plane listens to these messages regarding a new peer or an imported service. Additionally, the Policy engine manages authorization requests from the control plane for outgoing and incoming connections.

To simplify deployment, ClusterLink provides a Command Line Interface (CLI) tool to automate tasks such as certificate generation for the ClusterLink gateway, ClusterLink deployment, and resource removal. In addition, we created a Kubernetes Operator, the ClusterLink Operator, which can be invoked by the CLI and is responsible for creating and managing all ClusterLink components.

A. ClusterLink Connection Setting

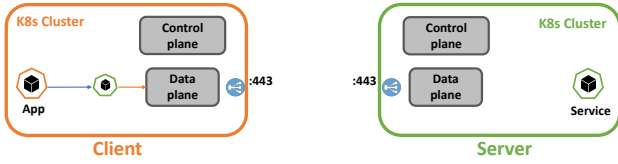
ClusterLink is designed to allow connectivity between different clusters with just a few easy steps. After deploying ClusterLink to all peer clusters using the CLI and the operator, the connectivity setup can be done using ClusterLink CRDs. In this example, we will demonstrate how to configure a client application in four steps to create connectivity to a remote service in the server cluster:

- First, set up the Peer CRDs to allow connectivity between the ClusterLink gateways.
 - Export the service in the server cluster using the Exported Service CRD.
 - Import the remote service in the client cluster using the Imported Service CRD.
 - Allow connectivity by creating Policy CRDs in both clusters.
- Now, a connection can be established between the client application and the remote service. In the next section, we describe the stages of connectivity establishment.

B. ClusterLink Connection Establishment

After setting up all the configurations (peer, export, import, and policy) to establish connectivity between the application and the remote service, the connectivity creation workflow is divided into four steps:

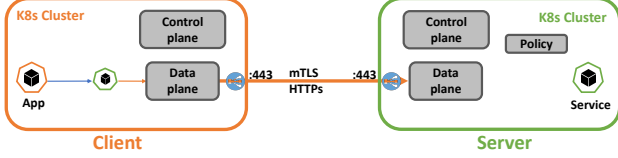
- As the first step (Fig. 3a), the application creates a connection to the local service that represents the remote service.



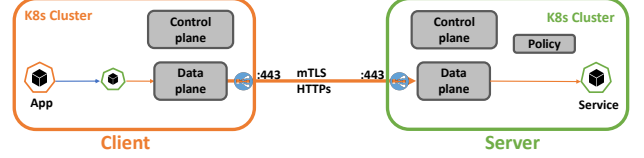
(a) First Step: The application accesses the local service, which forwards the request to the ClusterLink data plane.



(b) Second Step: Control plane authorization and policy validation.



(c) Third Step: Creating secure tunnel connectivity between the client and the server for the connection.



(d) Fourth Step: Creating app-to-service connectivity between ClusterLink data planes.

Fig. 3: Connectivity establishment between the application and remote service.

This service forwards the request to the ClusterLink data plane.

- Next (Fig. 3b), the ClusterLink data plane forwards the request to the control plane that checks if there is any policy that denies the connection, considering the application attributes and the remote service attributes. The client control plane creates an authorization request to the server control plane (through the server data plane, reusing a peer-to-peer secure connection). The server control plane retrieves the application attributes that initiated the connection and validates them. If there isn't any policy denying the connection, it sends back an authorization response that allows the connection establishment.
- Next (Fig. 3c), after the connection is authorized by the control plane, the client data plane establishes an mTLS-secured tunneling connection to the server data plane.
- Finally (Fig. 3d), with secure connectivity established between the ClusterLink gateways, the server's data plane creates a connection to the remote service, allowing the application to establish a connection through the ClusterLink data planes.

VI. CLUSTERLINK VALIDATION WITH USE-CASES

We have validated ClusterLink through integration with a broad range of multi-cloud use cases: a cloud-native web application (§VI-A), an extension to a cross-cluster service-mesh solution (§VI-B) an enterprise-grade multi-cloud management platform that establishes protected connectivity for its tenants (§VI-C), a protected data access and governance platform deployed across clusters in several cloud platforms (§VI-D), an LLM-serving platform that schedules serving jobs across multiple Kubernetes clusters (§VI-E), and a solution to securely connect IoT networks behind NATs or firewalls (§VI-F).

A. Simple Cloud Native Web App

For the initial validation, we experimented with several well-known web applications, including Istio's BookInfo [33] and IBM's Quote-Of-The-day(QoTD) [34], each consisting of multiple micro-services. Fig. 4 illustrates the QoTD deployment across three distinct Kubernetes clusters distributed across different locations of the GCP and the IBM clouds. Each cluster has ClusterLink gateway installed and configured through the ClusterLink APIs to connect between the gateways and to distribute the information about the deployed microservices based on the policies. ClusterLink could dynamically create the connectivity patterns required for the correct QoTD operations, supporting access control, load balancing, and failover policies. Notably, one of the microservices is responsible for serving images stored in a database that is positioned outside of the cluster. With ClusterLink, seamless connectivity could be easily achieved by adding an external endpoint, demonstrating the ability to support non-Kubernetes services as well. By using ClusterLink, external services that are accessible only from one cloud provider or a specific cluster in a specific region can be exposed to all the peers. In addition, the setup consisted of clusters powered by two different CNI plugins, Calico [35] and Flannel [19], demonstrating the ability to co-exist with different local Kubernetes networks.

B. Service Mesh Extension

We explored how ClusterLink extends traditional service mesh capabilities to efficiently manage services across disparate clusters and cloud environments. Today, service meshes have become the standard way to manage cloud-native applications within a single Kubernetes cluster. However, applications often run on multiple clusters with independent administrative domains and consume services from a multi-cloud environment.

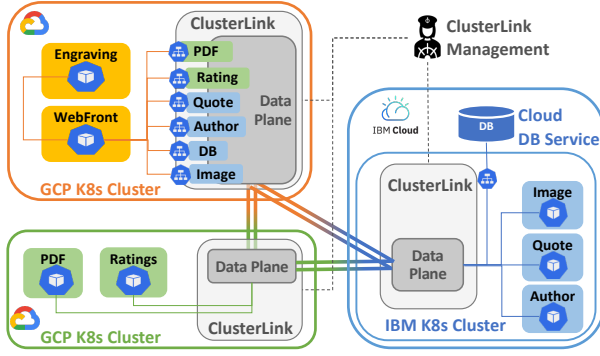


Fig. 4: ClusterLink interconnects micro-services of the “Quote of the day” web application across three clusters on different clouds. Services that are not explicitly exported can not be accessed.

Popular service mesh solutions like Istio [11], Linkerd [25], and NGINX Service Mesh [26] typically require creating a full mesh across all clusters, resulting in a large, interconnected service mesh. This approach may not be efficient or scalable. Additionally, in scenarios such as edge computing, running a service mesh might not be feasible, necessitating a more flexible solution. Another challenge arises in hybrid environments where remote clusters are behind NATs or firewalls, or where clusters have different administrative domains, making it infeasible to create a single service mesh to control all clusters.

To address the limitations of existing service mesh solutions, we set up applications with the Istio service mesh and enable them to consume remote services using ClusterLink. By utilizing ClusterLink, we can allow applications to continue using service meshes for managing services while also providing a scalable solution for connecting different remote services. Additionally, with ClusterLink’s policy engine, we can control access policies not just for cluster services but also for remote services. Furthermore, because ClusterLink creates a local service that represents the remote service in Istio, various features such as Istio Virtual Service, which manages and applies routing rules, can operate on the local service created by ClusterLink.

C. Multi-Cloud Management Platform

In the next step, we have integrated ClusterLink with a proprietary enterprise-grade multi-cloud management platform that orchestrates multi-tenant services to create private connectivity fabric for each tenant using ClusterLink gateways. When the fabric is established, it can be further modified through ClusterLink APIs. In addition, workload components and services, as well as the policies, can be defined through ClusterLink APIs and deployed by a less privileged role in the tenant’s organization. This integration validates the applicability and the sufficiency of the ClusterLink APIs for an enterprise-grade Multi-cloud management platform while being lightweight, secure, and performant.

D. Protected Data Access Platform

Additionally, we have integrated ClusterLink with an open-source data orchestration solution called Fyrik [36], a cloud-native platform to streamline data access, governance, and orchestration. With ClusterLink, we successfully established multi-cloud and multi-cluster connectivity for the Fyrik Solution that is deployed across multiple cloud platforms and enables secure client access to a remote database across multiple clouds. Additionally, ClusterLink enabled interconnecting Fyrik modules deployed in different remote clusters. ClusterLink provided an easier way for the Fyrik modules providing the data access to be deployed locally in a specific cloud, while other Fyrik components that need to access, e.g., the common data reduction and orchestration modules, could remain in a central remote cluster on a different cloud.

E. LLM Serving Platform

We have integrated ClusterLink with SkyShift [37], an open-source LLM serving platform serving jobs across multiple Kubernetes clusters based on GPU availability, while hosting the front ends for the multi-model serving service from a single primary cluster. This platform leverages ClusterLink to connect the front-end service to the LLM serving job replicas and uses ClusterLink’s load-balancing policies to balance the load across the replicas running on various Kubernetes clusters. ClusterLink creates EndpointSlices of the vLLM [38] pods in the primary cluster, and can be used to dynamically scale LLM serving based on the demand. ClusterLink, with its underlying Envoy data plane, can naturally evolve to support LLM routing and load-balancing strategies such as inference-time key-value computation reuse (KV-Cache) and session awareness [39] for latency optimization of serving.

F. Clusters Behind a NAT or Firewall

The ClusterLink gateway requires a public IP to export services and establish cross-cluster communication. However, many Kubernetes clusters are behind a NAT or a corporate firewall that only allows outgoing connections, as shown in Fig. 5. To address this connectivity challenge, we integrated ClusterLink with Fast Reverse Proxy (FRP) [29]. FRP creates reverse tunnels between its clients, providing access to clusters that only have outgoing connections behind a NAT. In this mode, each ClusterLink gateway creates an FRP client and connects through it to other clusters. Similar to a VPN, it is sufficient to have only one public IP for all the peers in the fabric. This deployment facilitates connectivity between diverse authority environments with varying security rules. We integrated ClusterLink into the IoT to Cloud Operating System (ICOS) open-source project [40]. This project allows the connectivity of multiple IoT devices to the cloud for data processing. With ClusterLink, we can enable connectivity between all the IoT devices, even those behind a NAT, to the central data center.

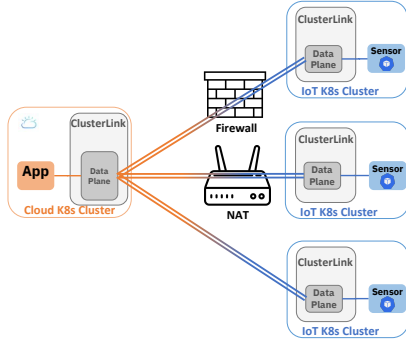


Fig. 5: ClusterLink interconnects Kubernetes clusters across different network boundaries. The cluster can be behind a NAT or firewall.

VII. EVALUATION

In this section, we present the results obtained from the evaluation of ClusterLink, focusing on throughput, latency, and scalability. We analyze its performance in a cloud-based environment, demonstrating its efficiency in handling network traffic across Kubernetes clusters.

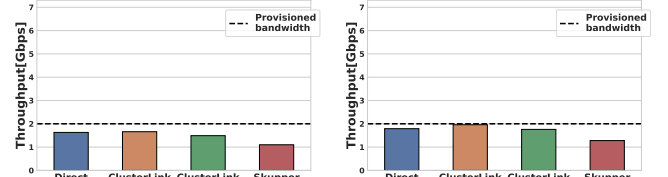
TABLE II: Latency between different GCP regions.

Type	Source Region	Destination Region	Latency
Intra-zone	us-central1-a	us-central1-a	0.3 ms
Inter-zone	us-central1-a	us-central1-b	0.8 ms
Inter-region (same continent)	eu-west1-b	eu-west3-a	8.5 ms
Inter-region (long distance, same continent)	us-central1-a	us-west1-a	37 ms
Inter-continental (near)	us-east5-a	eu-west2-a	81 ms
Inter-continental (medium distance)	us-east1-b	eu-north1-a	113 ms
Inter-continental (long distance)	us-south1-a	au-southeast1-a	168 ms

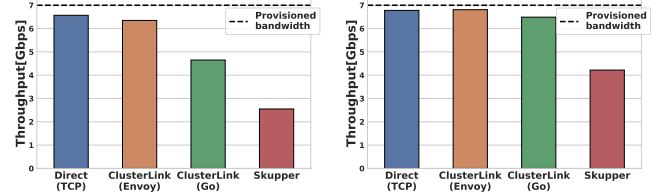
A. Throughput Evaluation

We evaluate ClusterLink’s throughput performance against Skupper [12], which is an L7-based application connectivity solution using AMQP. Skupper performs coarse-grained sharing of services between namespaces across clusters and does not support policies. While Skupper is fundamentally different from ClusterLink in design, as discussed in §III, we include it in the throughput comparison to highlight the efficiency of ClusterLink in handling network traffic across Kubernetes clusters.

We deploy two Kubernetes clusters in the same GCP availability zone. Throughput measurements are conducted using iPerf3 traffic measurement tool [41], averaging results over 40-second test intervals across 20 repetitions. We compare direct connectivity (manual configuration), ClusterLink with both Go-based and Envoy-based data planes, and Skupper, which relies on an AMQP-based overlay.



(a) Low bandwidth servers with a single connection. (b) Low bandwidth servers with two connections in parallel.



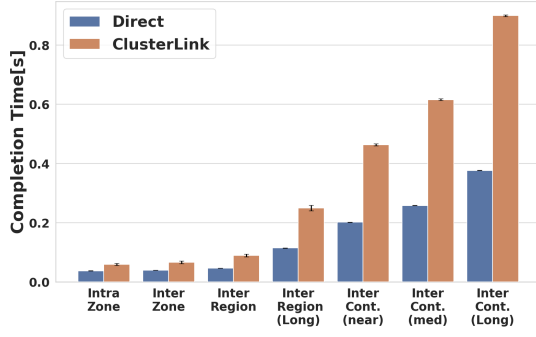
(c) High bandwidth servers with a single connection. (d) High bandwidth servers with two connections in parallel.

Fig. 6: Throughput comparison of ClusterLink versus Skupper.

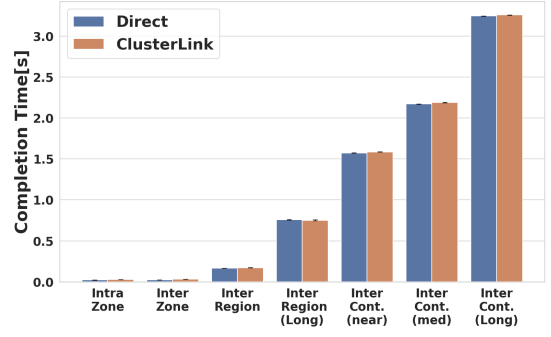
We repeat each test twice: once with a single connection and once with two connections in parallel to maximize the network bandwidth capacity.

Performance with Low Bandwidth Capacity. First, using the GCP’s e2-medium instances with a provisioned bandwidth of 2 Gbps (Fig. 6a), we achieve 1.63 Gbps with a direct connection. While Skupper achieves only 1.1 Gbps, ClusterLink Envoy and Go data plane achieve 1.66 Gbps and 1.49 Gbps respectively. We observe that gateway-induced overhead is lower for ClusterLink compared to Skupper due to simpler connection-based multiplexing achieved by the HTTP/ mTLS-based data plane (based on HTTP Connect) compared to the complex AMQP data plane of Skupper. Also, we see that ClusterLink has minimal overhead and can achieve the same performance as a direct connection. Second, we measure the total performance of two connections in parallel to maximize the capacity bandwidth. As shown in Fig. 6b, similar to before, we can see ClusterLink outperforms Skupper and reaches the maximum provisioned bandwidth by achieving 1.96 Gbps, while Skupper achieves only 1.28 Gbps. ClusterLink handles multiplexing connections better by creating different connections per flow, in contrast to Skupper, which uses AMQP to multiplex all flows in a single connection.

Performance with high Bandwidth Capacity. We repeat the evaluation with GCP’s more powerful n2-standard instances, provisioned with 7 Gbps of bandwidth. As shown in Fig. 6c, ClusterLink demonstrated superior performance again, surpassing Skupper by a factor of 2.5× with a throughput of 6.35 Gbps for a single iPerf3 connection, compared to Skupper’s 2.55 Gbps. Also, for the total throughput of two connections in parallel (Fig. 6d), we can see that both ClusterLink data planes (Envoy and lightweight-Go) achieve similar throughput as a direct connection: 6.81 Gbps, 6.49 Gbps, and 6.78 Gbps respectively, while Skupper achieves only

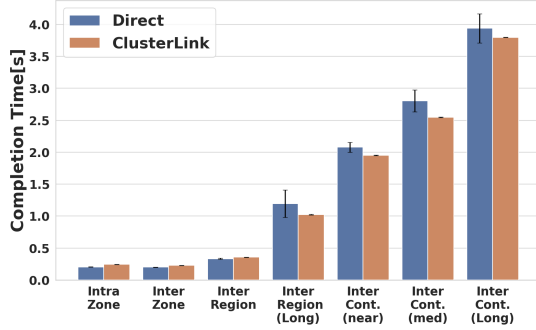


(a) First 1B file (includes connection setup).

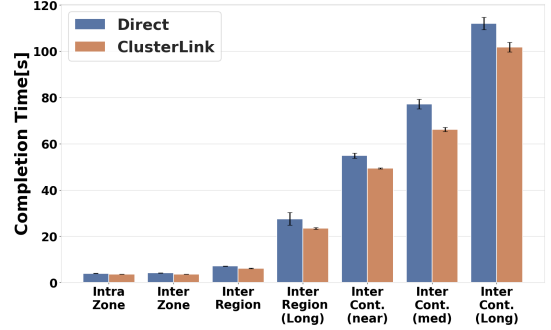


(b) Following 1B files (after connection setup).

Fig. 7: Completion time for downloading the initial 1B file and the following 1B files after connection setup.



(a) 2.5MB files



(b) 100MB files

Fig. 8: Completion time for downloading files of varying sizes: 2.5MB and 100MB.

4.22 Gbps. In terms of CPU utilization, ClusterLink uses additional CPUs to host the envoy/go proxy data plane, achieving similar levels of utilization as observed with iPerf3 CPU processing. We also found that memory usage to support specific bandwidth was negligible.

B. Latency Overhead

We examine the latency overhead imposed by ClusterLink by establishing connectivity between clusters in different regions within the GCP cloud, as shown in Table II. Each configuration introduces varying delay setups. We use Apache JMeter [42] to evaluate ClusterLink’s performance compared to direct connections when downloading files of different sizes: large files of 100MB, medium-sized files of 2.5MB (typical web page size [43]), and 1 byte. Each test measures the total time to download 20 files of the same size. We repeat each test 20 times and report the average. We conduct tests for both ClusterLink data plane types, but since they achieve similar results, we present only the results for the Envoy data plane.

Initially, we evaluate the performance of transferring a single byte, as shown in Fig. 7a. The results indicate that ClusterLink’s setup time is higher than that of a direct connection due to key authorization, end-to-end policy checks, and data encryption. We analyze the latency incurred at each cluster (Cluster1 & Cluster2) during the connection stage.

Our analysis shows that ClusterLink adds three additional round-trip times (RTTs) compared to an unsecured direct

connection: one for authorization between the ClusterLink control planes, one for establishing a secure mTLS connection, and one for HTTP CONNECT request and response between the ClusterLink data planes to create a secure tunnel. In total, we note that ClusterLink’s internal components add about 8.7 ms latency within the components. Subsequent data transfers show ClusterLink achieving similar performance to direct connections, as shown in Fig. 7b. Therefore, in subsequent experiments, we do not consider the initial connection establishment time for the first file.

Next, we measure the performance for requests involving 20 files, each with a medium size of 2.5MB, as shown in Fig. 8a. Under low-latency conditions, ClusterLink achieves results equivalent to direct connections without overhead in data transfer. As latency increases, ClusterLink outperforms direct connections. For larger 100MB files, as shown in Fig. 8b, ClusterLink maintains its advantage, consistently outperforming direct connections as latency increases. This improvement, in both cases, is attributed to the ClusterLink data plane, which is optimized for network connectivity compared to traditional applications. By leveraging features such as larger TCP buffers, ClusterLink significantly enhances connectivity performance under heavy workloads.

C. Scalability

We evaluate ClusterLink service scalability and observe that the cluster can support more than three thousand imported

services per peer, enabling applications to connect to a vast number of remote service resources. In the future, we plan to add support for a multi-dataplane, further enhancing scalability and improving the efficiency of remote service connectivity.

VIII. DISCUSSION

There's active work in progress in several directions:

Supporting other orchestrators. Currently, ClusterLink supports Kubernetes-based orchestrators. However, we are working on supporting deployment in orchestrators such as Ray [21] and SLURM [44]. We plan to support a plugin model to integrate with the corresponding workload orchestrator.

Workload Identity. Currently, ClusterLink's policies support expression in terms of attributes such as pod labels, peer labels, service names, and more. We have begun integrating ClusterLink with an open-source workload identity framework to support identity-based policies across cloud environments, such as SPIFFE/SPIRE [45], which can be validated using attested certificates in Zero Trust environments.

Enrich the Policy Engine. We plan to extend the policy engine to incorporate advanced policy-related aspects such as pluggable policies, policy validation and verification, and interactions with intra-cluster policies, such as generating and installing Kubernetes network policies.

IX. CONCLUSION AND FUTURE WORK

In this paper, we motivate the necessity of rethinking application connectivity in multi-cloud environments and propose abstractions that allow both high-level specification and efficient implementation of connectivity. We introduce ClusterLink, a foundational building block toward realizing our vision for multi-cloud connectivity. We describe ClusterLink's architecture and implementation, demonstrate its applicability to realistic multi-cloud use cases, and present performance evaluations highlighting the benefits of ClusterLink.

REFERENCES

- [1] Cisco, "Cisco SD-WAN: white papers," <https://www.cisco.com/c/en/us/solutions/enterprise-networks/sd-wan/white-paper-listing.html>, 2025.
- [2] VMware, "VMware SD-WAN: Data-sheet," <https://wan.velocloud.com/rs/098-RBR-178/images/sdwan-712-edge-platform-spec-ds-1020.pdf>, 2025.
- [3] Palo Alto Networks, "Prisma SD-WAN: Data-sheet," <https://www.paloaltonetworks.com/resources/datasheets/prisma-sd-wan-instant-on-network-ion-device-specifications>, 2025.
- [4] Fortinet, "Fortinet secure SD-WAN: white paper," <https://www.fortinet.com/content/dam/fortinet/assets/white-papers/sd-wan-in-the-age-of-digital-transformation.pdf>, 2025.
- [5] Aviatrix, "Aviatrix documentation," <https://docs.aviatrix.com>, 2025.
- [6] Alkira, "Alkira documentation," <https://www.alkira.com>, 2025.
- [7] GCP, "Cloud VPN Documentation — Google Cloud," <https://cloud.google.com/network-connectivity/docs/vpn>, 2025.
- [8] AWS, "Connect your VPC to remote networks using AWS VPN," <https://docs.aws.amazon.com/vpc/latest/userguide/vpn-connections.html>, 2025.
- [9] Azure, "Microsoft Azure VPN Gateway," <https://azure.microsoft.com/en-us/products/vpn-gateway/>, 2025.
- [10] Cilium, "eBPF-based Networking, Observability, Security," <https://cilium.io/>, 2025.
- [11] Istio, "Service mesh solution," <https://istio.io/latest>, 2025.
- [12] Skupper, "Multicloud communication for Kubernetes," <https://skupper.io>, 2025.
- [13] C. N. C. Foundation, "CNCF Cloud Native Interactive Landscape," <https://landscape.cncf.io/>, 2025.
- [14] Kubernetes, "Production-Grade Container Orchestration," <https://kubernetes.io/>, 2025.
- [15] ClusterLink Team, "ClusterLink: An Open-Source Project for Multi-Cloud Connectivity (GitHub Repository)," <https://github.com/clusterlink-net/clusterlink>, 2025.
- [16] S. McClure, Z. Medley, D. Bansal, K. Jayaraman, A. Narayanan, J. Padhye, S. Ratnasamy, A. Shaikh, and R. Tewari, "Invisinets: Removing networking from cloud networks," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Apr. 2023, pp. 479–496. [Online]. Available: <https://www.usenix.org/conference/nsdi23/presentation/mcclure>
- [17] Submariner, "Submariner: Networking Solution for Kubernetes Clusters," <https://submariner.io/>, 2025.
- [18] "What is Zero Trust Architecture (ZTA)?" <https://www.paloaltonetworks.com/cyberpedia/what-is-a-zero-trust-architecture>, 2025.
- [19] Flannel-IO, "Flannel CNI," <https://github.com/flannel-io/flannel>, 2025.
- [20] I. Stoica and S. Shenker, "From cloud computing to sky computing," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 26–32.
- [21] Anyscale, "Ray: A Framework for Scaling and Distributing ML Applications," <https://docs.ray.io/en/latest/cluster/vms/index.html>, 2025.
- [22] Z. Yang, Z. Wu, M. Luo, W.-L. Chiang, R. Bhardwaj, W. Kwon, S. Zhuang, F. S. Luan, G. Mittal, S. Shenker, and I. Stoica, "SkyPilot: An intercloud broker for sky computing," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Apr. 2023, pp. 437–455. [Online]. Available: <https://www.usenix.org/conference/nsdi23/presentation/yang-zongheng>
- [23] P. Jain, S. Kumar, S. Wooders, S. G. Patil, J. E. Gonzalez, and I. Stoica, "Skyplane: Optimizing transfer cost and throughput using Cloud-Aware overlays," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Apr. 2023, pp. 1375–1389. [Online]. Available: <https://www.usenix.org/conference/nsdi23/presentation/jain>
- [24] Flexiwan, "The World's First Open Source SD-WAN & SASE," <https://flexiwan.com>, 2025.
- [25] Linkerd, "Light and fast service mesh," <https://linkerd.io/>, 2025.
- [26] NGINX, "NGINX Service Mesh: Light Service Mesh," <https://docs.nginx.com/nginx-service-mesh/>, 2025.
- [27] OpenZiti, "Secure by design networking, anywhere, as software," <https://openziti.io>, 2025.
- [28] K. Toledo, D. Breitgand, D. Lorenz, and I. Keslassy, "Cloudpilot: Flow acceleration in the cloud," *Computer Networks*, vol. 224, p. 109610, 2023.
- [29] FRP, "Fast reverse proxy open source," <https://github.com/fatedier/frp>, 2025.
- [30] Google, "The Go Programming Language," <https://golang.org/>, 2025.
- [31] Lyft, "Envoy service proxy for cloud-native applications," <https://www.envoyproxy.io/>, 2025.
- [32] xDS-WG, "xDS protocol," <https://github.com/cncf/xds>, 2025.
- [33] Istio, "Istio BookInfo application," <https://istio.io/latest/docs/examples/bookinfo/>, 2025.
- [34] IBM-Watson, "Quote Of The Day," <https://gitlab.com/quote-of-the-day/quote-of-the-day>, 2025.
- [35] Tigera, "Calico CNI," <https://www.tigera.io/project-calico/>, 2025.
- [36] Fybrik, "A cloud-native platform to control data usage," <https://fybrik.io/v1.3/>, 2025.
- [37] SkyShift, "LLM serving platform," <https://github.com/sky-shift/skyshift>, 2025.
- [38] "vLLM," <https://github.com/vllm-project/vllm>, 2025.
- [39] Envoy, "Envoy AI Gateway," <https://github.com/envoyproxy/ai-gateway>, 2025.
- [40] ICOS, "IoT to cloud operation system project," <https://www.icos-project.eu/>, 2025.
- [41] iPerf3 project, "Measuring network performance tool," <https://iperf.fr/iperf-download.php>, 2025.
- [42] JMeter, "Measurement performance tool," <https://jmeter.apache.org/>, 2025.
- [43] HTTP Archive, "Average Web Page Weight," https://httparchive.org/reports/page-weight?start=2024_04_01&view=list, 2025.

- [44] SchedMD, “Slurm Workload Manger,” <https://slurm.schedmd.com/>, 2025.
- [45] SPIFFE and SPIRE Working Group, “SPIFFE and SPIRE: Providing Strongly Attested, Cryptographic Identities to Workloads Across a Wide Variety of Platforms,” <https://spiffe.io/>, 2025.