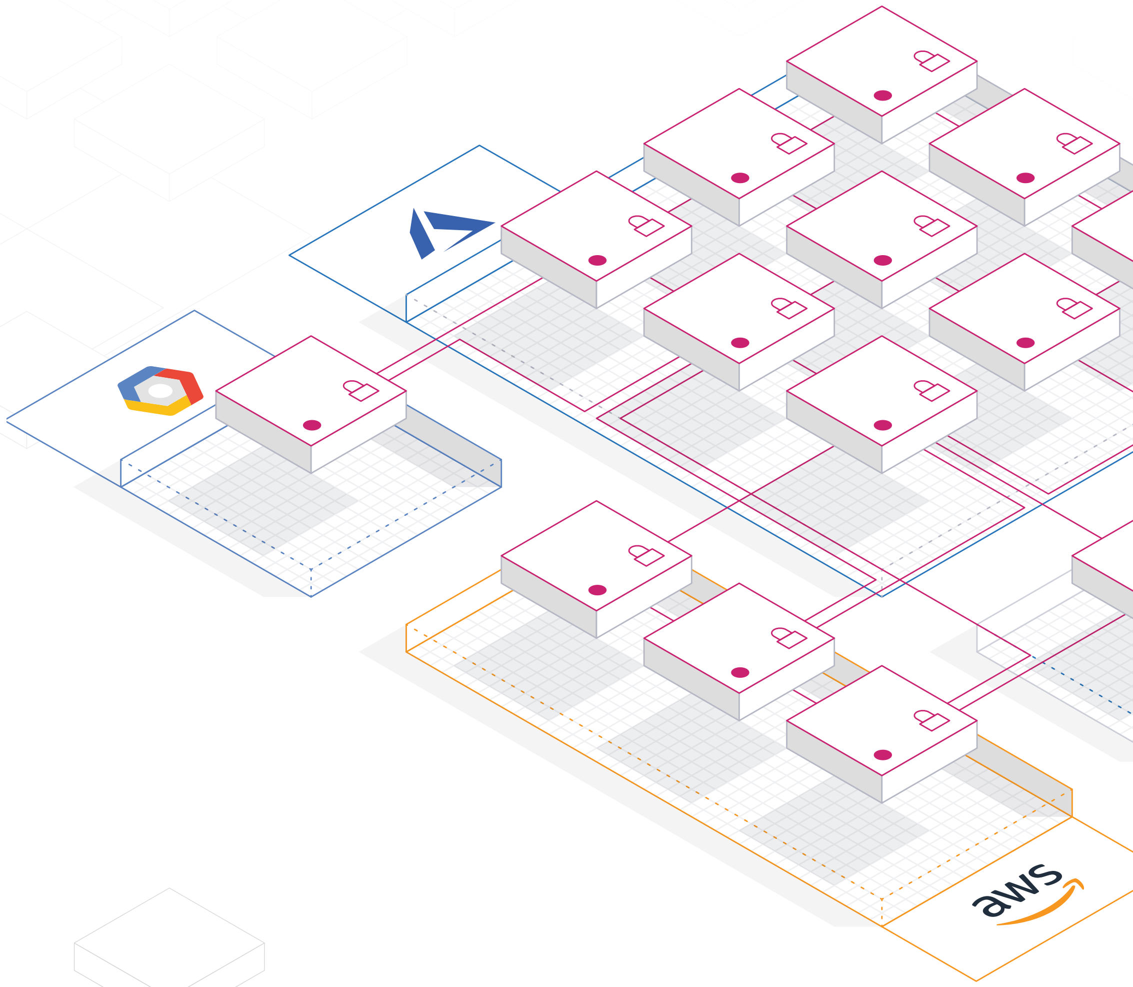
HashiCorp
Consul

Modern Service Networking for Cloud and Microservices



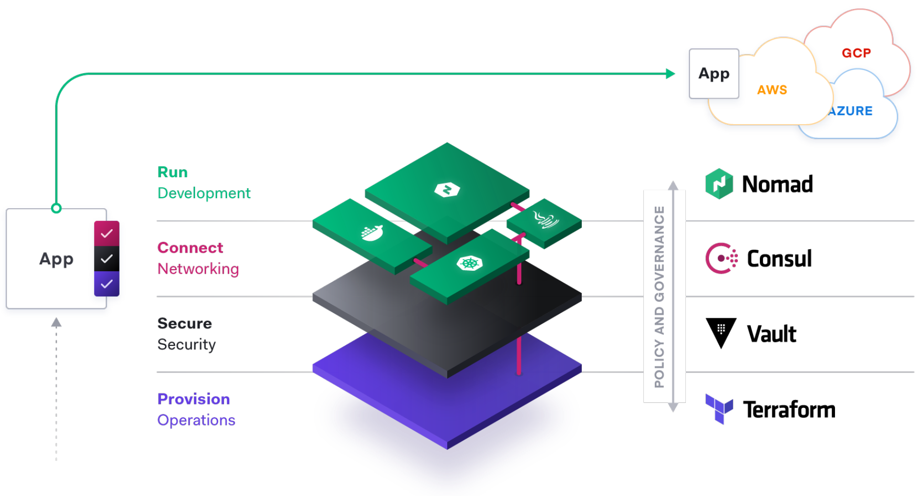
HashiCorp

Contents

Service Discovery for the Cloud Operating Model

Enterprises thriving in the multi-cloud era compete on speed to market. The development team is taking the cloud-native approach to launch new features faster. Companies are inspired by successful early adopters, like Netflix and eBay, but they often forget about the new challenges that come with modern application development and cloud adoption.

The cloud operating model is composed of four distinct layers, mapping to the personas and functional areas in IT. The challenges for supporting distributed applications at the networking layer are often one of the most difficult aspects. This white paper will examine the shortcomings of traditional networking approaches and introduce service discovery, a modern approach that we consider an essential part of multi-cloud service networking.



Monolithic applications in a static network

For many years, nearly all applications were built with a monolithic architecture. A single monolithic application contains all the sub-components for all the business functionalities that an application delivers. A banking app, for example, would include multiple modules for login, displaying your balance, wire transfers, and deposits. Although these are discrete functions, the application is packaged and deployed as a single unit.

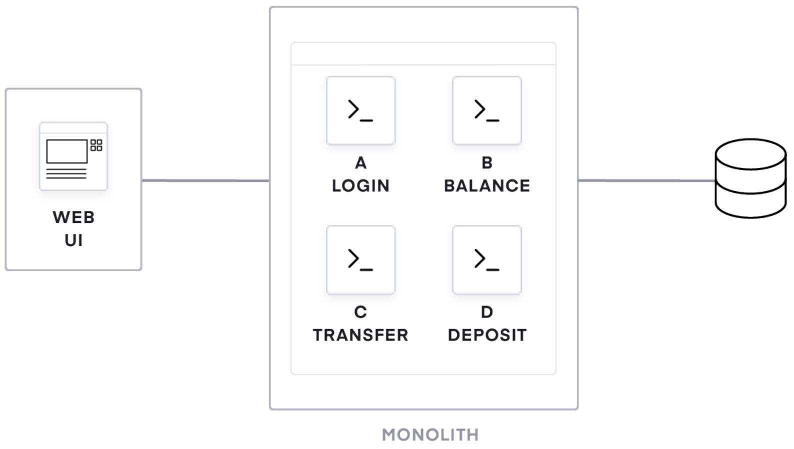


Figure 1: A monolithic banking application.

The underlying networking architecture to support monolithic applications has a set of clean traffic paths. These paths are typically north-south, which means the traffic is going in and out of the network. A common example would be the traffic between internet users and front-end web servers. To handle heavy load and improve application availability, incoming requests from users are routed through a load balancer, which splits the traffic across multiple web UI instances.

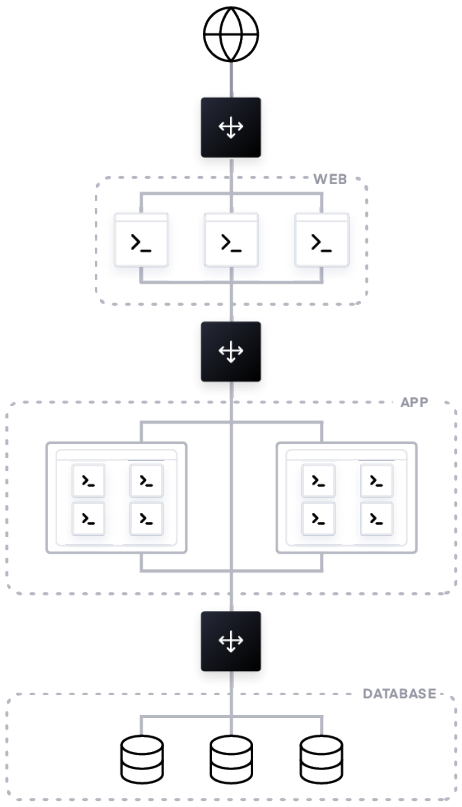


Figure 2: A static network topology with load balancers fronting the database, monolithic applications, and the web front-end.

Traffic is also routed within the network—this is called east-west traffic. An example would be, traffic between application servers and the back-end database. Legacy monolithic applications typically connected with these data center components by hardcoding the IP address of the database or connecting with another load balancer fronting the database.

Overall, these traffic paths are well-defined and relatively static. The communication between sub-components of a monolithic application is via an in-memory function call.

Moving to microservices

Monolithic applications are easy to develop, deploy and scale. However, over time an application gains new features and functionality and requires a larger team to develop. As this team grows larger, it needs to split into many independent teams who own different pieces of functionality. At scale, there can be hundreds of developers working on a single code base. This creates immense challenges around coordinating a release and doing proper testing and Q/A. In many cases, releases slow down to a quarterly or slower cadence.

As a solution to this problem, many forward-thinking companies have adopted a new application architecture called "microservices." Transforming a monolithic application to this architecture means breaking out its sub-components into loosely-coupled, independent modular services. Each of these services is owned by independent development teams. Microservices offer a number of benefits over monolithic software architectures, including:

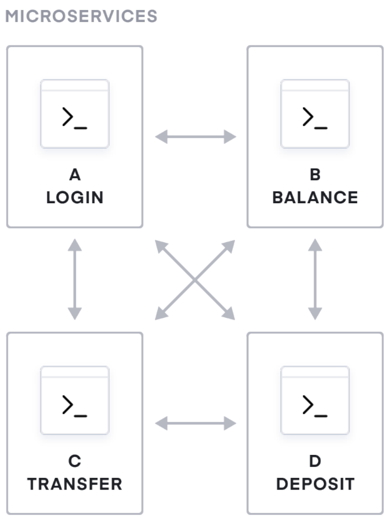


Figure 3: A microservices banking application.

- **Fast deployment:** Teams can build, test, and deploy each service independently without being constrained by a single release cycle. If there is a bug in service A, we can patch A and redeploy A without having to wait for development teams B, C, and D to be ready.

- **Technology flexibility:** Each service's development team has the freedom to choose different frameworks, languages, and tools that are best suited for their application. One team could use Python to develop a new machine learning service while other teams could keep their existing services in Java.

- **Efficient scaling:** Properly decoupled services can be scaled independently. This creates more cost-efficiencies since you don't need to scale the whole system, only certain services.

- **Failure Isolation:** Each service is responsible for a small piece of overall functionality. The whole application is composed of many individual services. This means if there is a bug or failure impacting a particular service, the entire application is not impacted. This allows failures to be more gracefully tolerated.

Containers and cloud: Dynamic IPs

While we are decomposing applications, these individual services are packaged and deployed as containers. Containers are generally ephemeral and have a dynamic IP address. They can start and terminate quickly to scale services up and down. This presents a new challenge: the IP address of microservices are constantly changing.

Cloud migration further complicates this issue. An organization might move many of their existing services to a public cloud, but then build new services on a second cloud, and leave their systems of record on-premises for data sovereignty. The cloud services normally receive dynamically assigned IP addresses. Once a service instance stops, the cloud IP address can be recycled and used for a different application. As a result, tracking cloud services and resources as well as keeping them up to date in a multi-cloud environment becomes much more difficult and complex.

Networking challenges in a dynamic network

Microservices, containers, and cloud computing have brought about a revolution in development agility, but it doesn't come for free. As we are gaining developer efficiency from the distributed architecture, we are inheriting new operational challenges due to the same distributed nature.

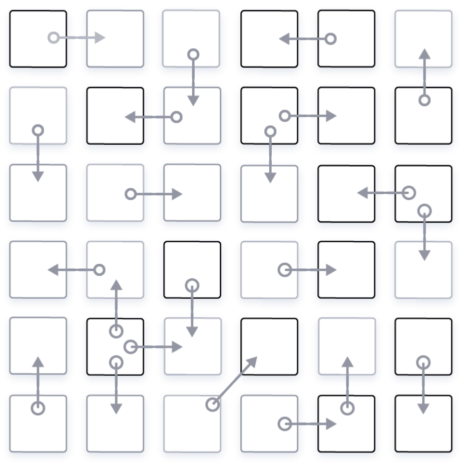


Figure 4: Communication patterns in a simple microservices application.

In a microservices environment, operators have to work with hundreds of small services each with independent deployments and scaling challenges. These services are no longer compiled into a single application, and they are not even running on the same machine. They communicate using APIs over the network on-premise or in the cloud. This introduces a challenge of service discovery, namely how downstream services discovery and route traffic to the upstream services they depend on.

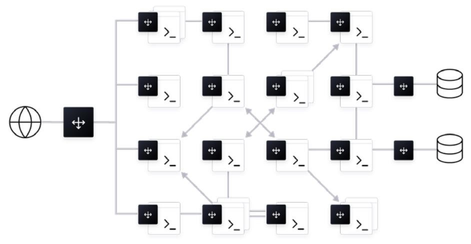


Figure 5: A theoretical dynamic network topology with each service fronted by load balancers.

In a traditional approach, every service would be fronted with a load balancer (physical, virtual, or cloud depending on the deployment environment). Downstream services would hard code the address of the load balancer when routing to the upstream service. In this way, a load balancer would provide a naming abstraction to represent a service and be responsible for routing traffic to available instances of that service. This approach seems like a logical extension of standard monolithic networking architecture, but it introduces a few problems.

Firstly, adding (or removing) a new service or a new instance of an existing service comes with a lot of overhead. The traditional ticketing system or manual operation process to update the load balancer configuration cannot keep up with the speed at which microservices are deployed. Container platforms can deploy containers in several seconds, while many ticket based processes take weeks to process changes. Additionally, since each new service type needs its own load balancer, there would be an expensive proliferation of load balancers that need to be managed and maintained.

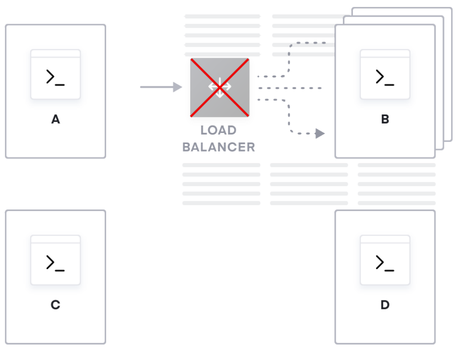


Figure 6: Despite service B's redundancy, if the only load balancer goes down, all instances of service B are unavailable.

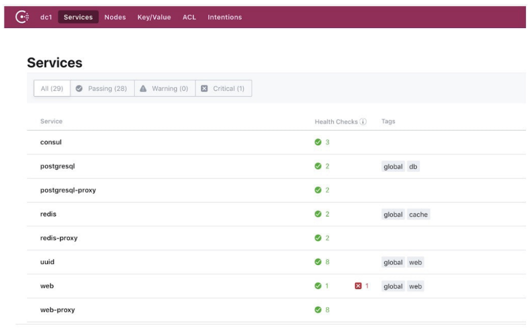
The second challenge is the introduction of single points of failure all over the system. Even though we're running multiple instances of service B for availability, service A connects to service B through its load balancer. If we lose that load balancer, all instances of service B will become unavailable. This could be mitigated to some extent by adding redundant load balancers, but this adds additional cost and more operational complexity.

The third challenge is the network latency introduced by load balancers. Instead of service A directly connecting to service B, A connects to a load balancer which then connects to B, and then any response takes the same path on the way back. This doubles the network latency for every service-to-service communication. As we adopt microservices, there are many more network calls required to service a user request and optimization of latency becomes important to preserve user experience.

Starting with a service registry

Instead of using distributed load balancers, microservice networking requires a central service discovery mechanism to dynamically discover and connect services running on the ephemeral infrastructure.

Service discovery starts with a centralized service registry, which provides a “directory” of what services are running, where they are, and their current health status. A light-weight service discovery client is running along with a service instance, allowing this service instance to programmatically register and deregister with the central registry. The system will monitor a service’s health status so operators can triage the availability of each instance.



Services

All (29) | ✓ Passing (28) | ⚠ Warning (0) | ☒ Critical (1)

Service	Health Checks		Tags
consul	✓ 3		
postgresql	✓ 2		global db
postgresql-proxy	✓ 2		
redis	✓ 2		global cache
redis-proxy	✓ 2		
uuid	✓ 8		global web
web	✓ 1	☒ 1	global web
web-proxy	✓ 8		

Figure 7: The service health status dashboard for HashiCorp Consul.

For organizations at the beginning of their cloud journey or microservice adoption, finding a way to track and manage the explosion of services across multiple subnets, data centers, and cloud regions is their first hurdle. The traditional process, which often takes the form of spreadsheets, load balancer dashboards, or configuration files, is disjointed and static. Choosing a technology that provides a central service registry is the first step in clearing that hurdle, providing a foundation for automation and more advanced microservice networking solutions to build upon.

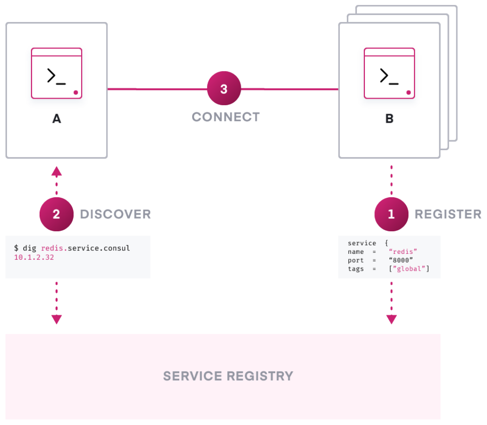
Service discovery for east-west traffic

As organizations scale their microservices, the traffic between services grows exponentially and the traffic path becomes more dynamic. Load balancers are not a scalable approach, as we illustrated prior to this section. However, service discovery provides a more elegant approach to managing east-west traffic.

With a service discovery solution in place, all service instances get registered as part of the central service registry the moment they are deployed. If service A needs to communicate with service B, it queries the service registry which returns the network location for all available instances of service B. Typically, this is done without code modification using DNS. More advanced use cases are enabled by using the rich REST API.

This solution can also perform load balancing by randomly sending traffic to different instances. In this approach, the routing and the load balancing is fully distributed which allows the system to scale into a very large environment.

Within a single data center or cloud region, if one of the service instances dies, the registry will avoid returning its address to trigger an automatic failover to other healthy instances. Across data centers, organizations can define centralized failover policies with service discovery to automatically route traffic to service instances that are running in different geo-locations or cloud regions, meaning they no longer have to hardcode this logic into applications or manage other failover appliances.



```
$ dig redis.service.consul
10.1.2.32
```

```
service  {
name  =   "redis"
port  =   "8000"
tags  =   ["global"]
```

Figure 8: Once a service or resource is automatically registered in the service registry, it can discover other services that want to use that resource or service and connect them.

Additionally, service discovery coupled with sidecar proxies or embedded client libraries can enable more advanced application-specific traffic management schemes.

Network automation for north-south traffic

Despite the explosion of east-west traffic, it is still desirable to have a dedicated load balancer or API gateway at the edge where all the north-south traffic flows into the system.

In microservice architectures, a web frontend service could change dynamically because of autoscaling, failures, and updates. Every time this happens, the load balancer needs to be reconfigured to update its back-end address pool, which is often a painful, time-consuming, and error-prone process.

With service discovery, the service registry provides a "subscription" service to automate these network operations. Developers can now auto-scale services, and the newly added or removed service instances will automatically "publish" their location information with the service registry. The load balancer can subscribe to service changes from the service registry (by using tools like Consul Template or native integration). This enables a Publish/Subscribe style of automation that can handle highly dynamic infrastructure.

Any change to a service will trigger a new configuration getting generated and reloaded to the load balancer dynamically. The traffic can be routed to new service instances instantly without manual intervention. The same approach applies to firewalls and other critical network middleware as well. This publish/subscribe model accelerates the productivity of the network and IT operation teams by decoupling their workflow from IP addresses.

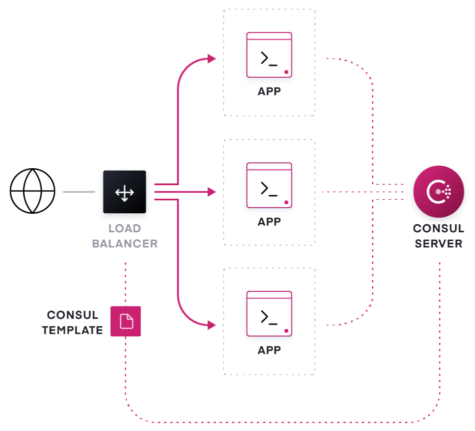


Figure 9: A dynamic network topology with HashiCorp Consul handling east-west traffic and service-to-service communication, while a load balancer balances north-south traffic.

As we scale out the microservice architecture, companies may deploy API gateways at the edge. An API gateway provides a single, unified API entry point across one or more internal APIs. It hides the complexity of internal microservices composition from external users. To send the API requests to internal services dynamically, it will need to integrate with service discovery to locate and route to the proper end services.

Bridging multiple platforms and clouds

Microservice architectures and cloud infrastructure gives development teams the freedom to choose different runtime platforms and cloud services that are best suited to their applications. However, this could lead to multiple islands of resources where the dynamic service networking is constrained within a single platform or a particular cloud. For example, a service deployed in a Kubernetes cluster cannot dynamically discover the database running on virtual machines outside the Kubernetes environment.

Another example could be that a cloud service has to hardcode the IP addresses of services running in a private datacenter. These challenges arise when companies need to support cross-cluster/platform/cloud communications. To enable services to discover and interact with each other regardless of where they reside, we need a service discovery mechanism that can be universally deployed. It needs to provide a unified service registry that is not platform or cloud-specific and can automatically sync and aggregate service catalogs across a heterogeneous environment.

Summary

Service discovery has become an established core component of microservice and multi-cloud service networking. It acts as the brain of your networking and monitoring operations, simplifying them as your service portfolio expands. There are many service discovery tools to solve this problem in different ways. HashiCorp Consul provides comprehensive service discovery capabilities, which offer not only a service registry, but also distributed health checks, security, high availability, scalability, multi-datacenter/cloud, and universal support on bare metal, virtualized, and containerized environments, including all major operating systems.

Consul has been successful in accelerating companies' cloud and microservices adoption journeys. It provides a stable, flexible, and battle-tested solution for service discovery with a central registry of services provides application visibility, dynamic routing and network automation. Instead of having applications adapt to the network infrastructure, service discovery makes the network serve the needs of the application, eliminating slow manual processes that were required to keep networks and critical middleware up to date, and replacing those processes with self-service automation.specific and can automatically sync and aggregate service catalogs across a heterogeneous environment.

HashiCorp