

# Service Discovery

(Elasticity & Cluster Formation)

Renato Panda

[renato.panda@ipt.pt](mailto:renato.panda@ipt.pt)

# Once again, David is back...

- David's toolbox has increased recently
  - Virtualization (servers)
  - Provisioning tools such as Ansible
  - DevOps culture and automation
  - Load Balancing
  - High Availability



# Once again, David is back...

- David's toolbox has increased recently
  - Virtualization (servers)
  - Provisioning tools such as Ansible
  - DevOps culture and automation
  - Load Balancing
  - High Availability

... but David is clever!



# Once again, David is back...

- David's toolbox has increased recently
  - Virtualization (servers)
  - Provisioning tools such as Ansible
  - DevOps culture and automation
  - Load Balancing
  - High Availability

... but David is clever and curious!



# Once again, David is back...

With these solutions (LB + HA)  
and well-designed products\* I  
can **scale out** as needed...

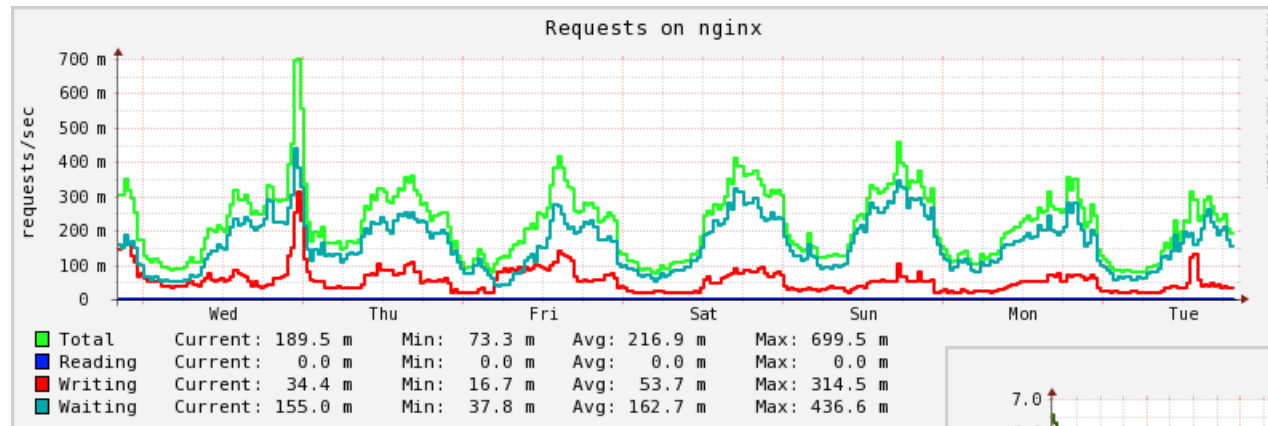
...but how do I  
define "as needed"?

Isn't that  
constantly  
changing?

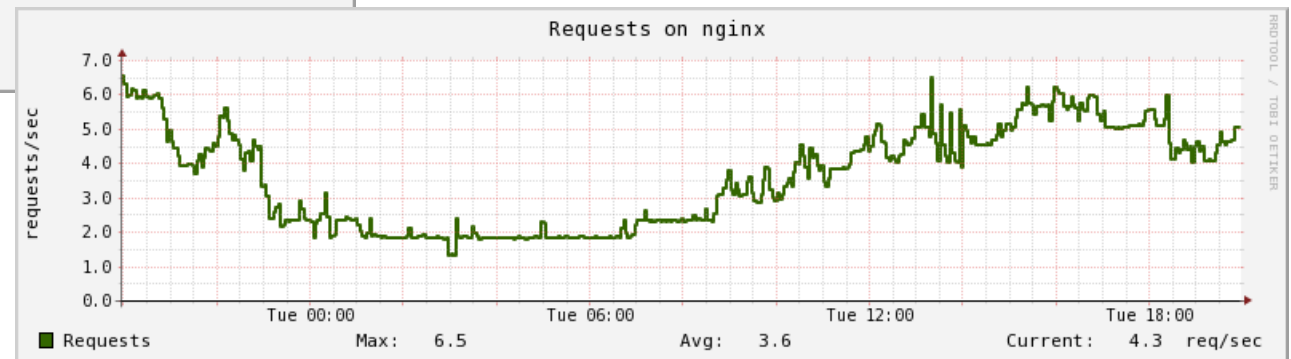


# Scale out... “as needed”?

- David is right, the number of requests is not constant over time
  - For internal applications, most requests will be during the working hours

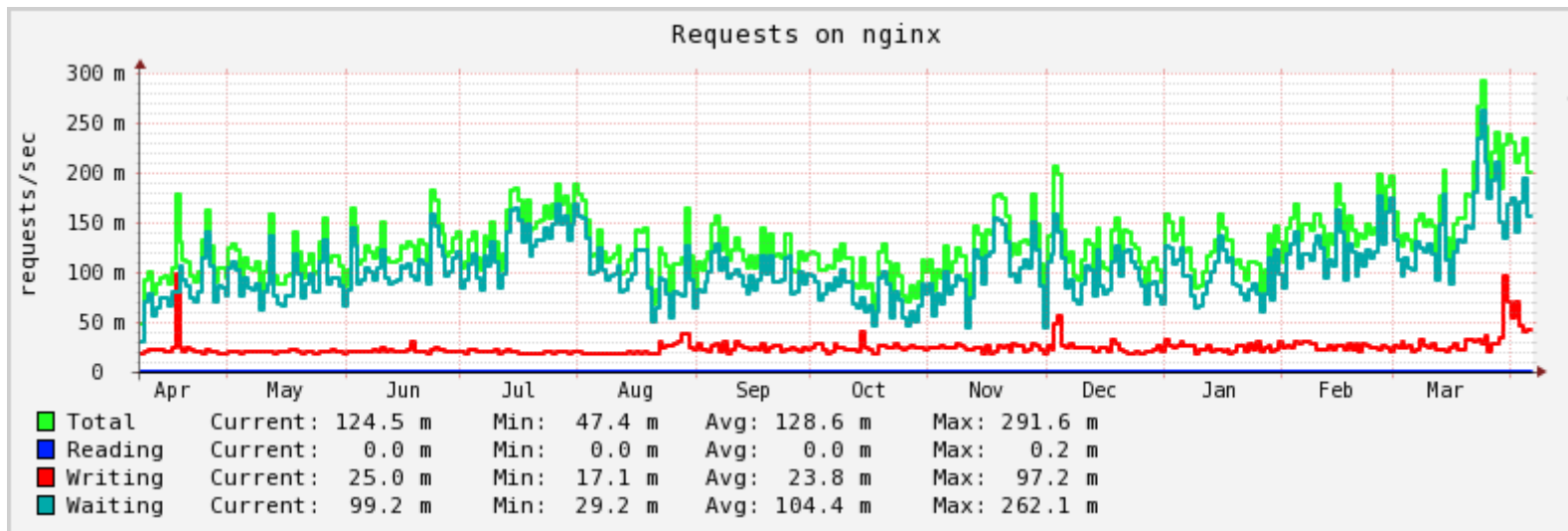


But this might change during weekends and summer...



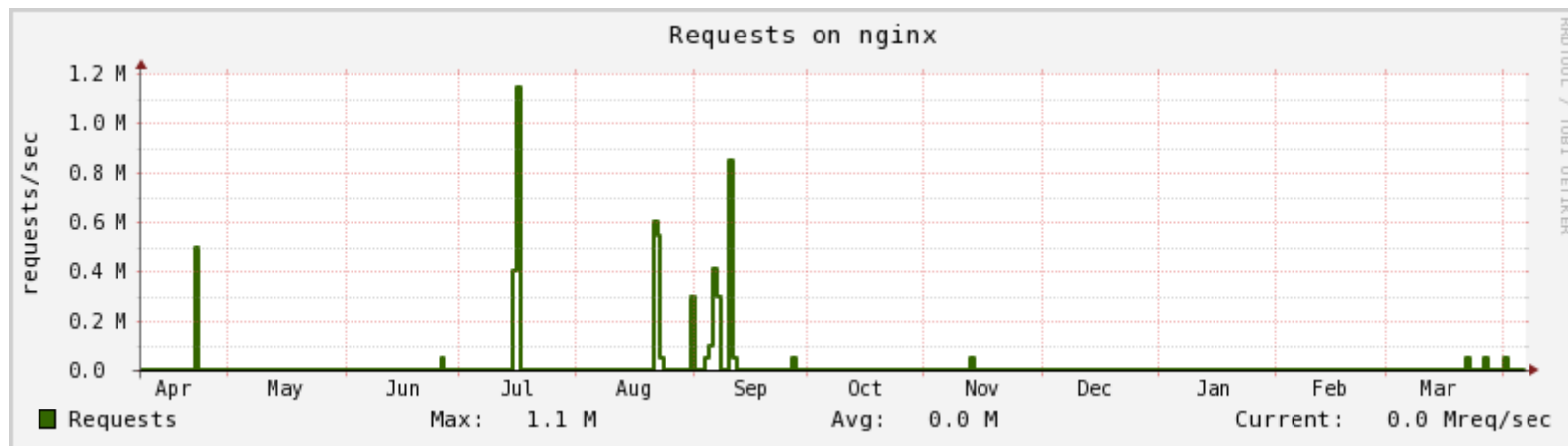
# Scale out... “as needed”?

- David is right, the number of requests is not constant over time
  - Some products might generate more traffic as they get more popular



# Scale out... “as needed”?

- David is right, the number of requests is not constant over time
  - For a myriad of reasons, we can even get huge unpredictable spikes
    - min-saude.pt during March 2020
    - A product or company getting popular on social networks
    - Netpa during class registration, BigBlueButton during mandatory quarantine





# Scale out... “as needed”?

- David is right, the number of requests is not constant over time
  - For a myriad of reasons, we can even get huge spikes
    - min-saude.pt nowadays
    - A product or company getting popular on social networks
    - Netpa during class registration, BigBlueButton during mandatory quarantine

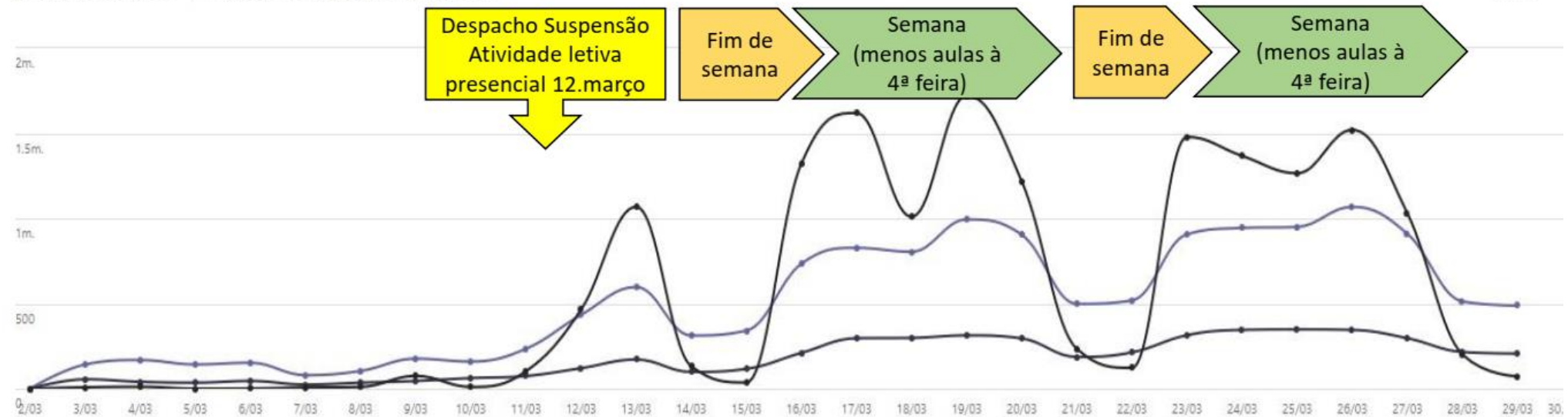


# IPT MS Teams during Covid19 Quarantine

## Dashboard de Utilização do Teams

### Relatório de utilização do Teams

01/04/2020 10:39:0 UTC | Intervalo de datas: 2/03/2020 - 30/03/2020



1,602

Total de utilizadores ativos

1,431

Utilizadores ativos de equipas e de ...

747

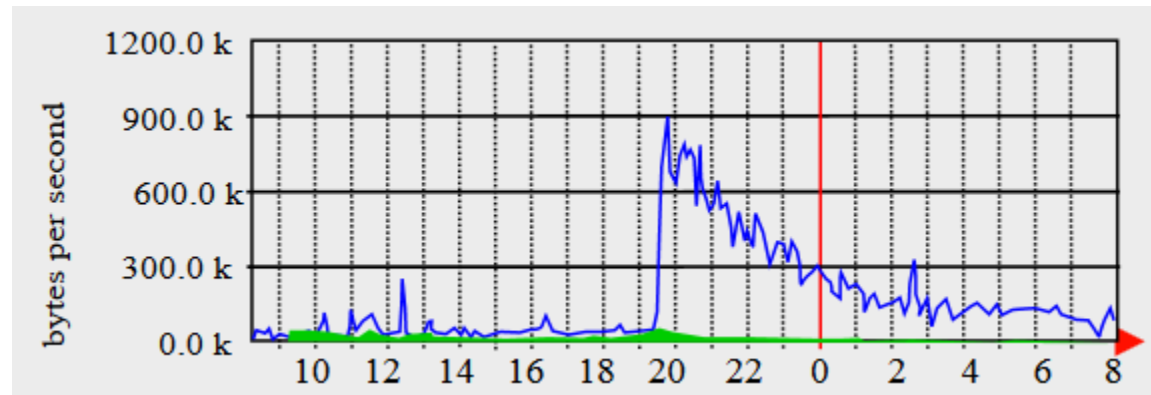
Canais ativos

17,582

Mensagens

# Slashdot effect ([Wikipedia](#))

“The **Slashdot effect**, occurs when a popular website links to a smaller website, causing a **massive increase in traffic**. This **overloads** the smaller site, causing it to **slow down** or even temporarily become **unavailable**.”



# Slashdot effect ([Wikipedia](#))

“The name stems from the **huge influx of web traffic** which would result from the technology news site Slashdot linking to websites.”

## Slashdot effect ([Wikipedia](#))

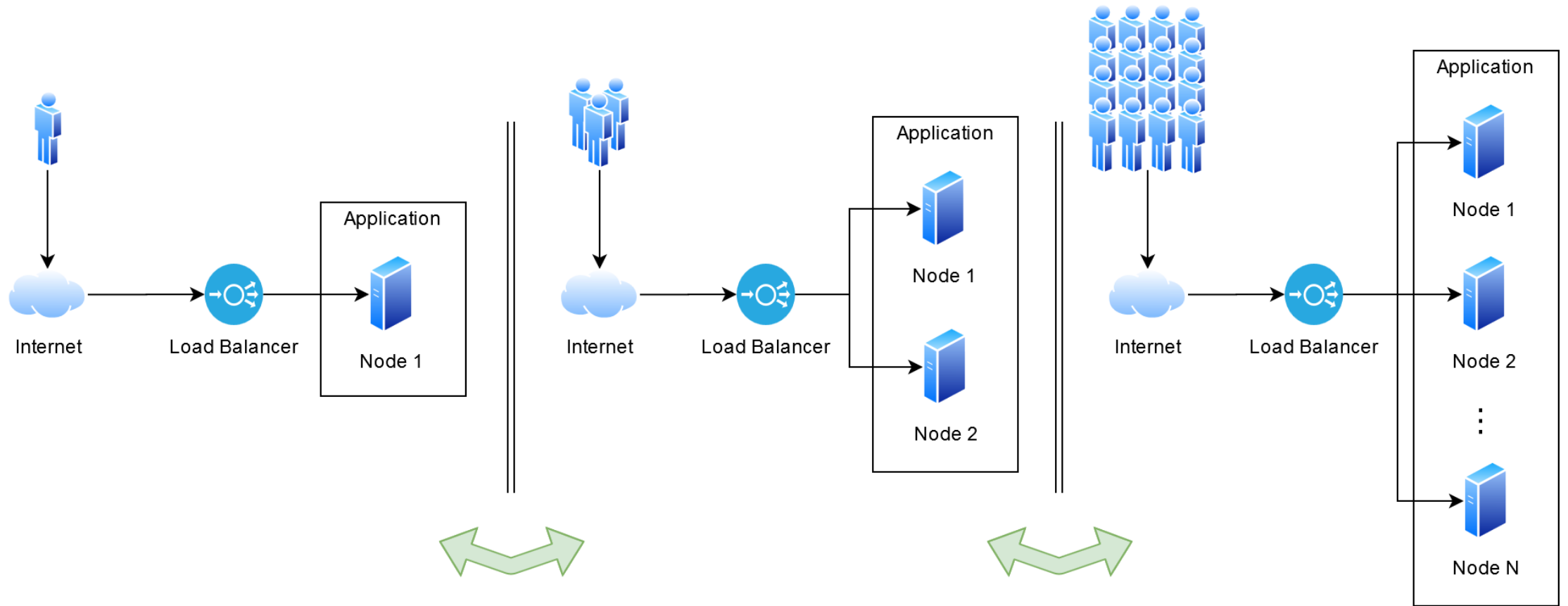
“The original circumstances have changed, as flash crowds from Slashdot were reported in 2005 to be diminishing due to competition from similar sites, and the general adoption of **elastically scalable cloud hosting platforms.**”

# Slashdot effect ([Wikipedia](#))

“The effect has been associated with other websites or metablogs such as Fark, Digg, Drudge Report, Imgur, Reddit, and Twitter, leading to terms such as being "farked" or "drudged", being under the "**Reddit effect**"—or receiving a "**hug of death**" from the site in question.”

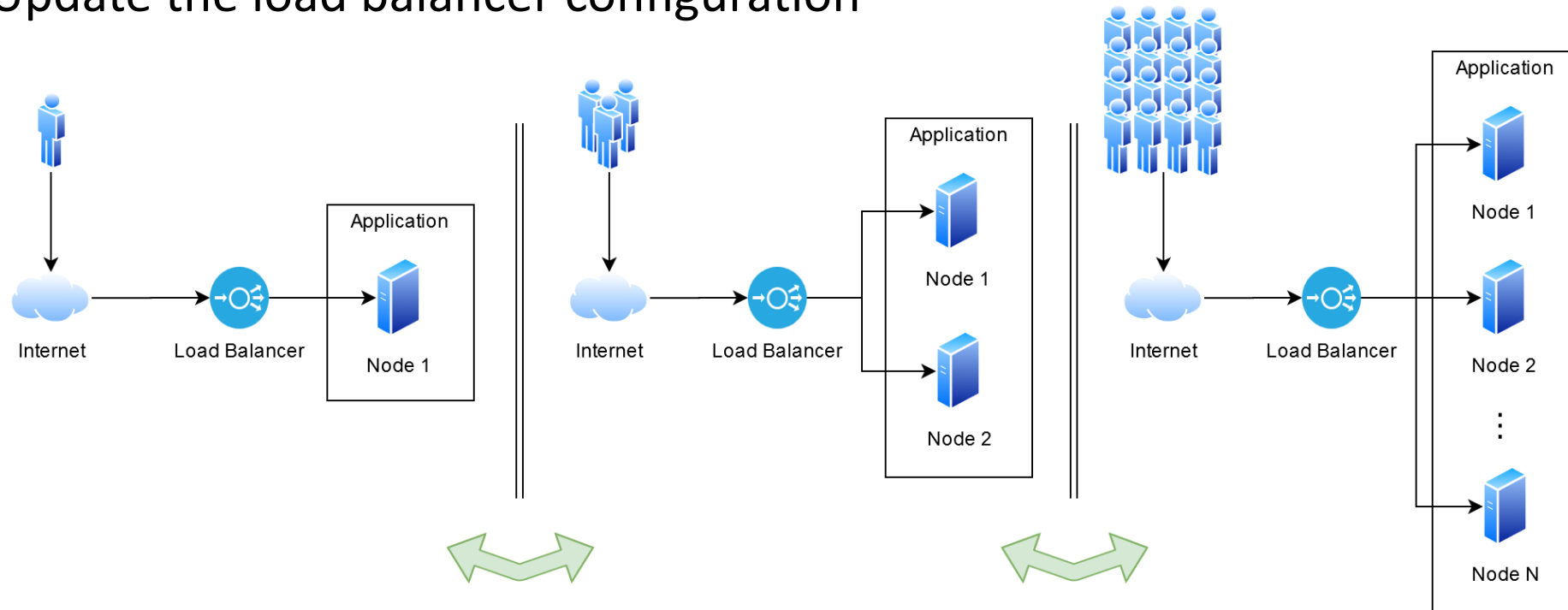
What is the price of going offline in a situation like this?

# Scaling out as needed...



# Scaling out as needed...

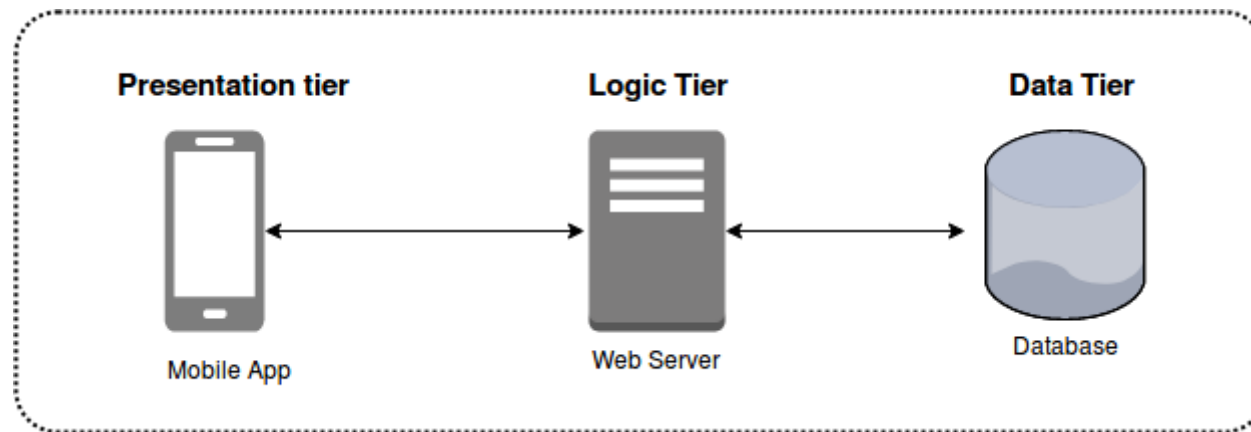
- How do we achieve this [in this simplified scenario]?
  - Start or stop one of the application instances
  - Update the load balancer configuration





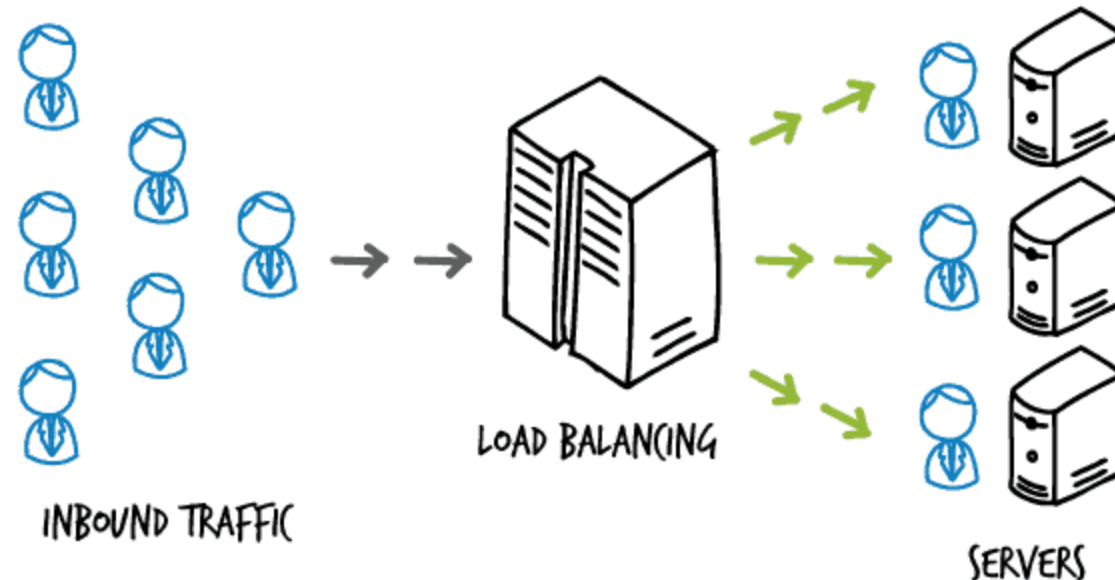
# The Problem

- To make requests, we need to know IP addresses and ports
  - In traditional applications hosted in physical hardware, this is not a problem
    - Network locations of services are relatively static



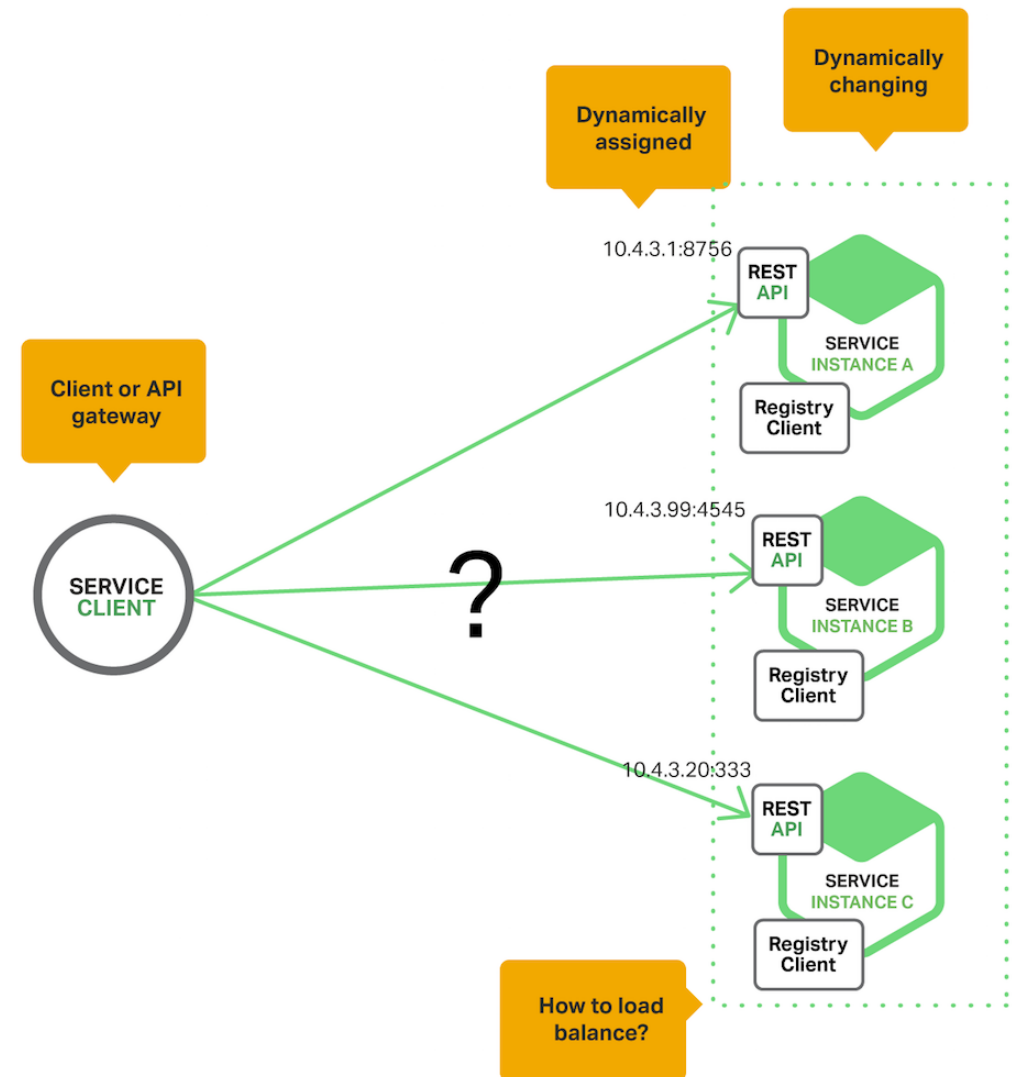
# The Problem

- To make requests we need to know IP addresses and ports
  - Even with distributed systems, this can be handled
    - As long as the scenario remains immutable
    - When changes are needed, someone needs to update the LB configuration



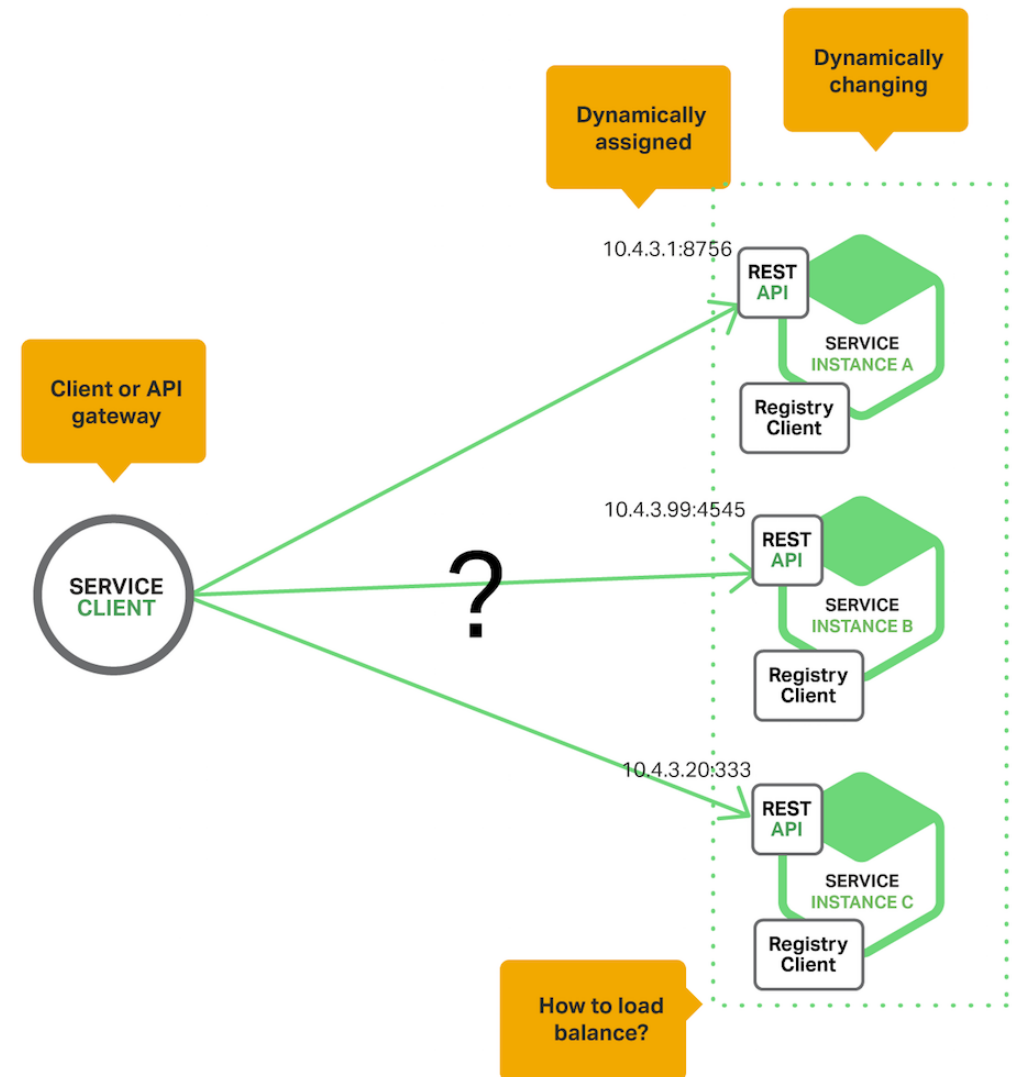
# The Problem

- In modern virtualized or cloud-based architectures, this becomes much harder
  - Instances have dynamically assigned network locations
  - Instances may change dynamically
    - (Auto) scaling
    - Failures and recoveries
    - Upgrades



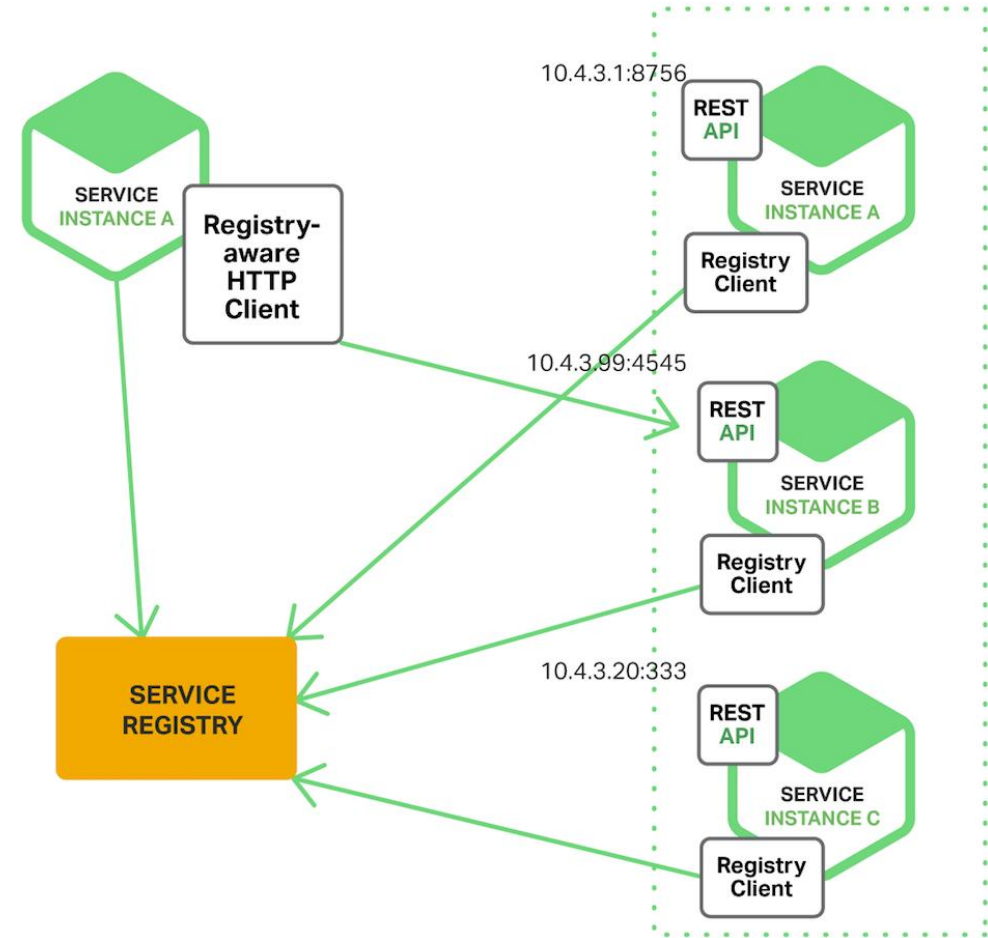
# The Problem

- How to solve this?
  - Use a service discovery mechanism
- Two main service discovery patterns
  - Client-Side Discovery Pattern
  - Server-Side Discovery Pattern



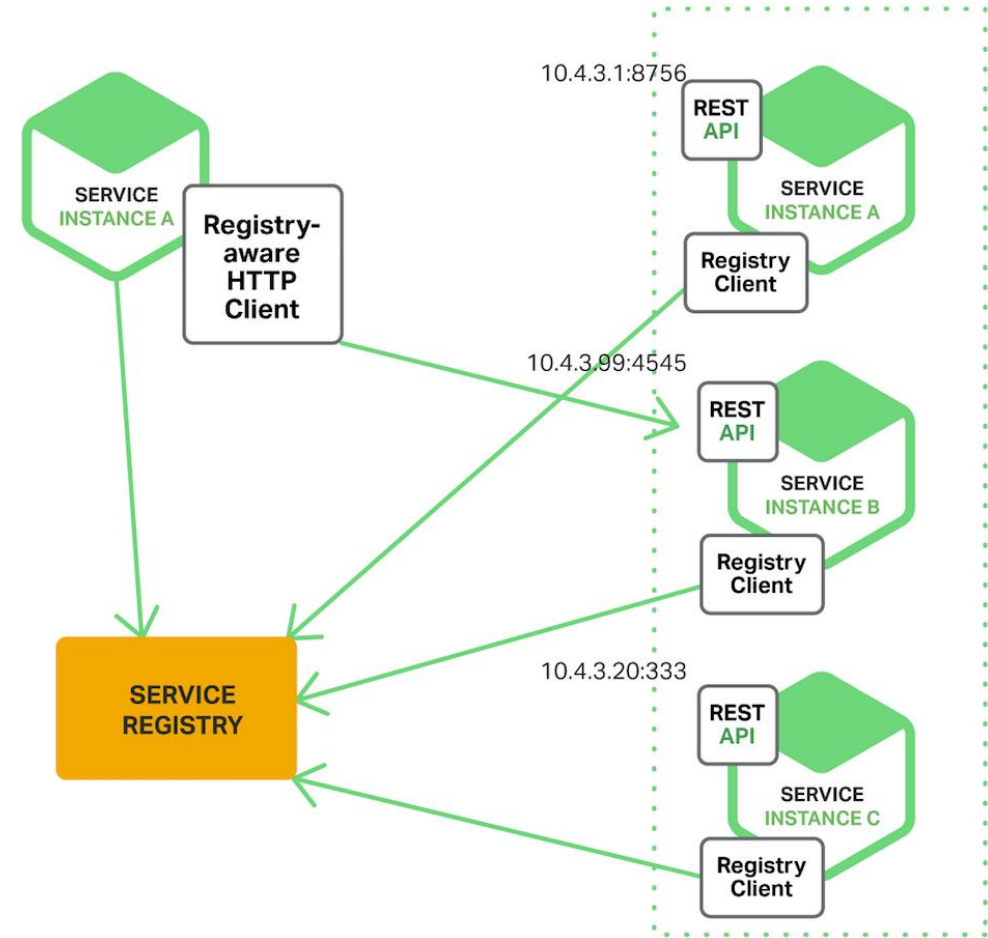
# Client-Side Discovery Pattern

- Client is responsible for discovering service instances and load balancing requests across them
- How does it work?
  - Client queries a service registry
    - DB of available service instances
  - Client selects one of the services
    - Uses a load-balancing algorithm
  - Client makes the request



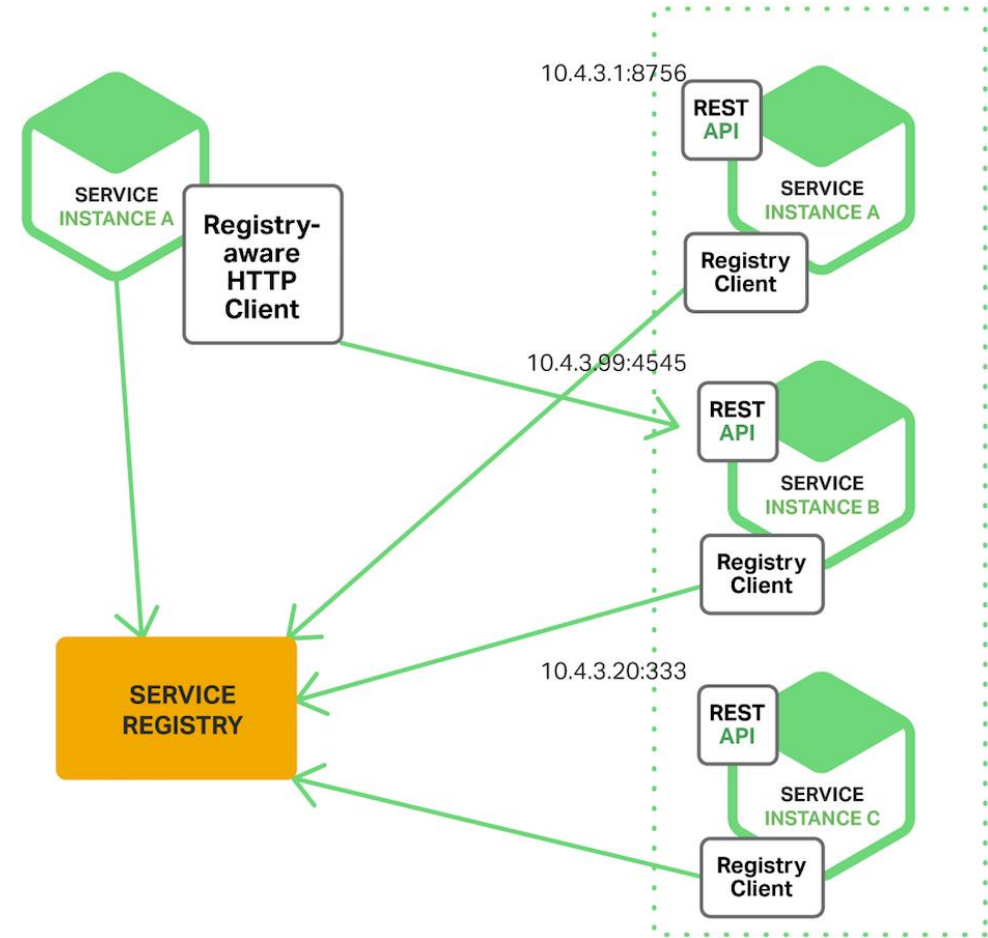
# Client-Side Discovery Pattern

- Client is responsible for discovering service instances and load balancing requests across them
- How are service instances registered?
  - Instance registers its location when starting
    - Also removed on termination
  - Heartbeat mechanisms normally used
    - To refresh instance registration



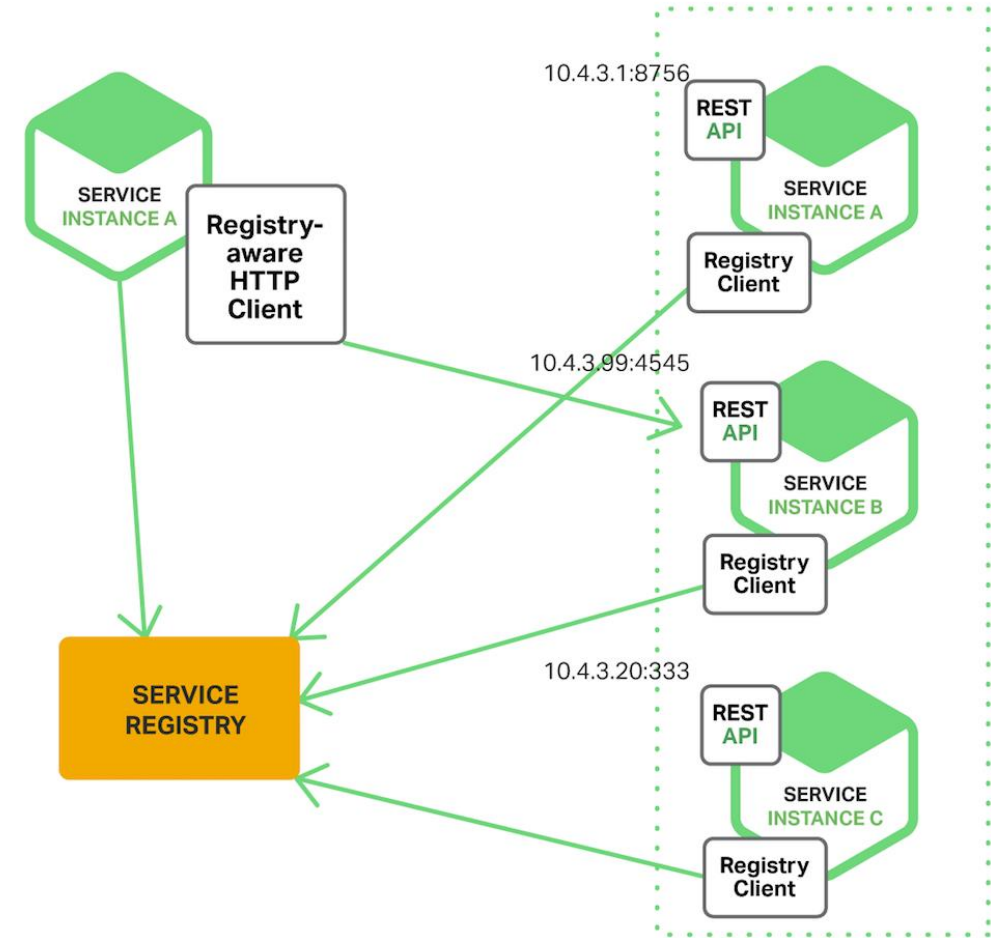
# Client-Side Discovery Pattern

- Client is responsible for discovering service instances and load balancing requests across them
- Advantages?
  - Relatively straightforward
    - Only new part is the service registry
  - Intelligent application-specific load balancing decisions can be made
    - Since client knows the available instances



# Client-Side Discovery Pattern

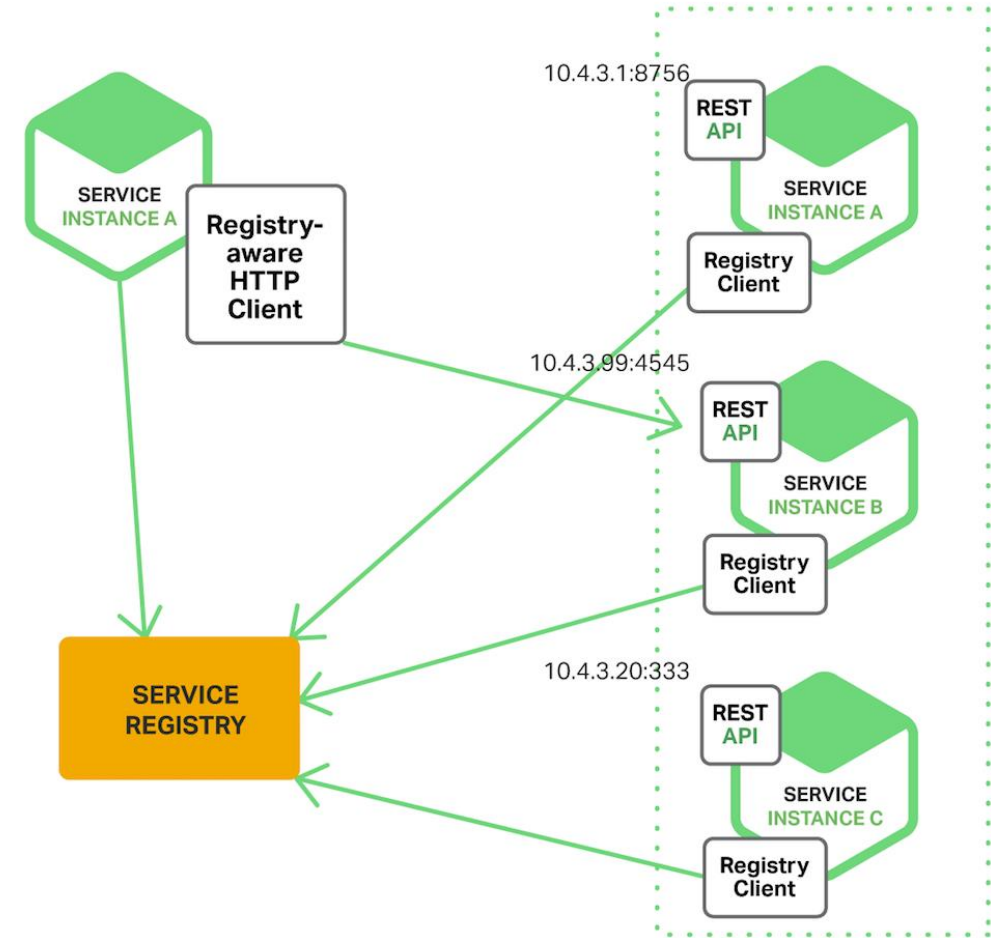
- Client is responsible for discovering service instances and load balancing requests across them
- Disadvantages?
  - Service registry logic coupled with client
    - Each *client*\* must implement this





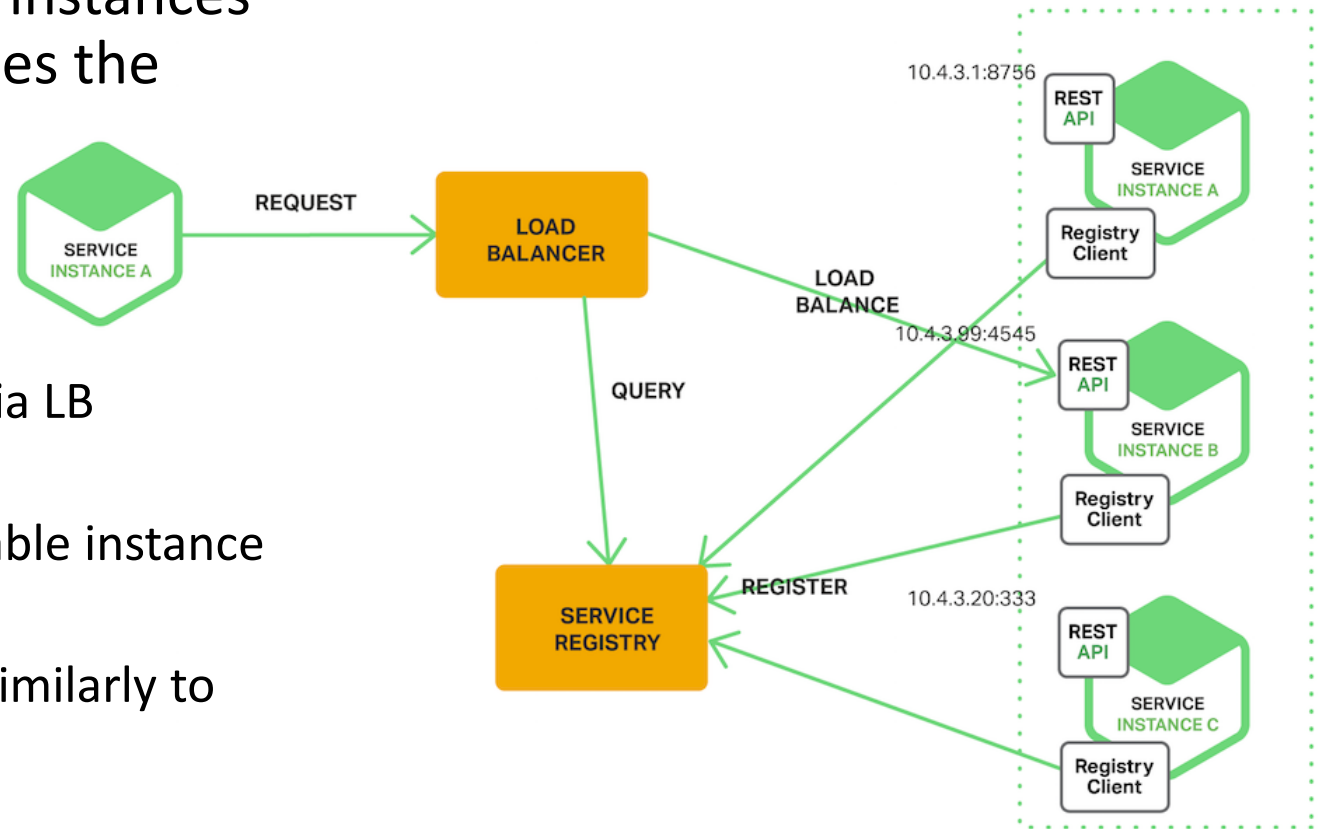
# Client-Side Discovery Pattern

- Client is responsible for discovering service instances and load balancing requests across them
- Examples
  - Netflix Eureka (+ Netflix Ribbon)



# Server-Side Discovery Pattern

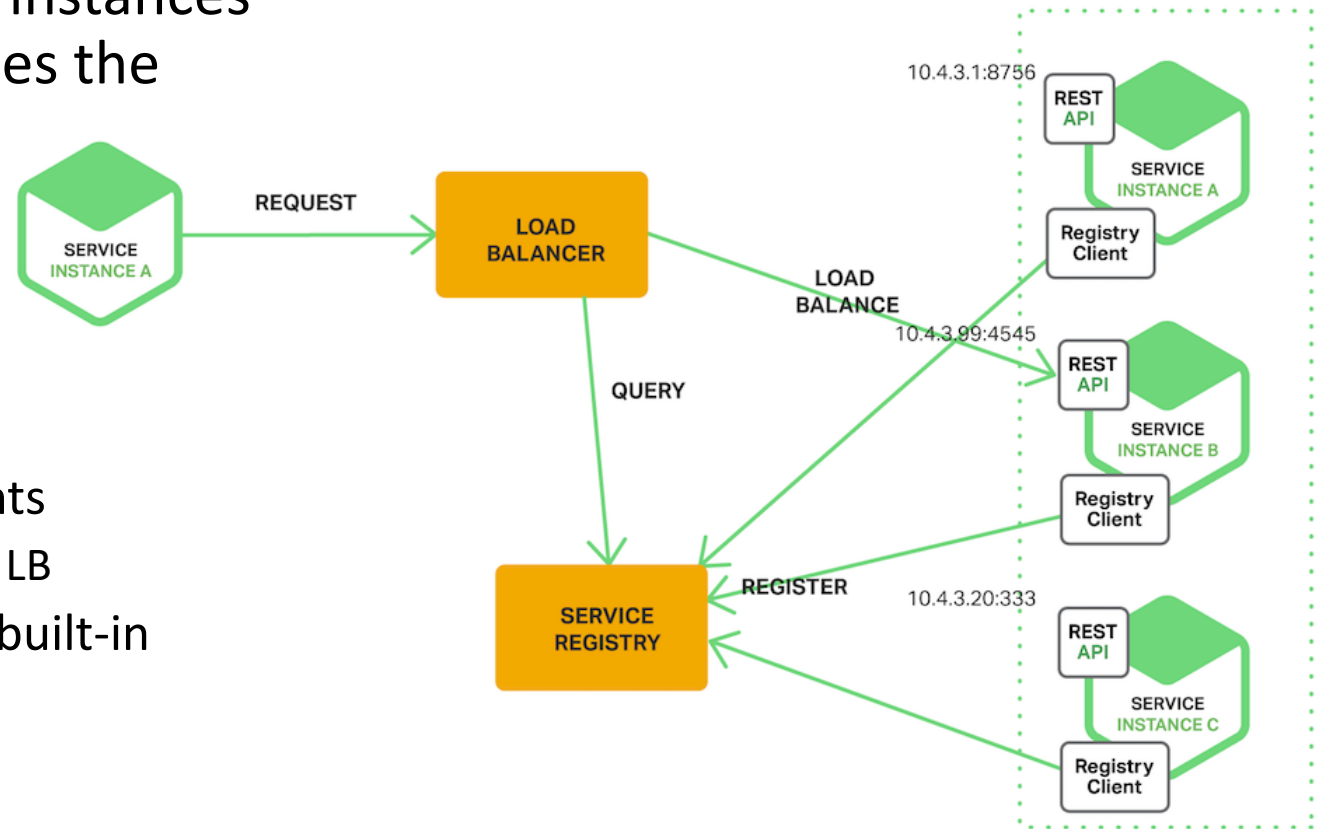
- Client makes request to service instances via a load balancer, which queries the service registry
- How does it work?
  - Client makes request to service via LB
  - LB queries service registry
  - LB routes the request to an available instance
  - Service instances are registered similarly to the client-side discovery pattern



# Server-Side Discovery Pattern

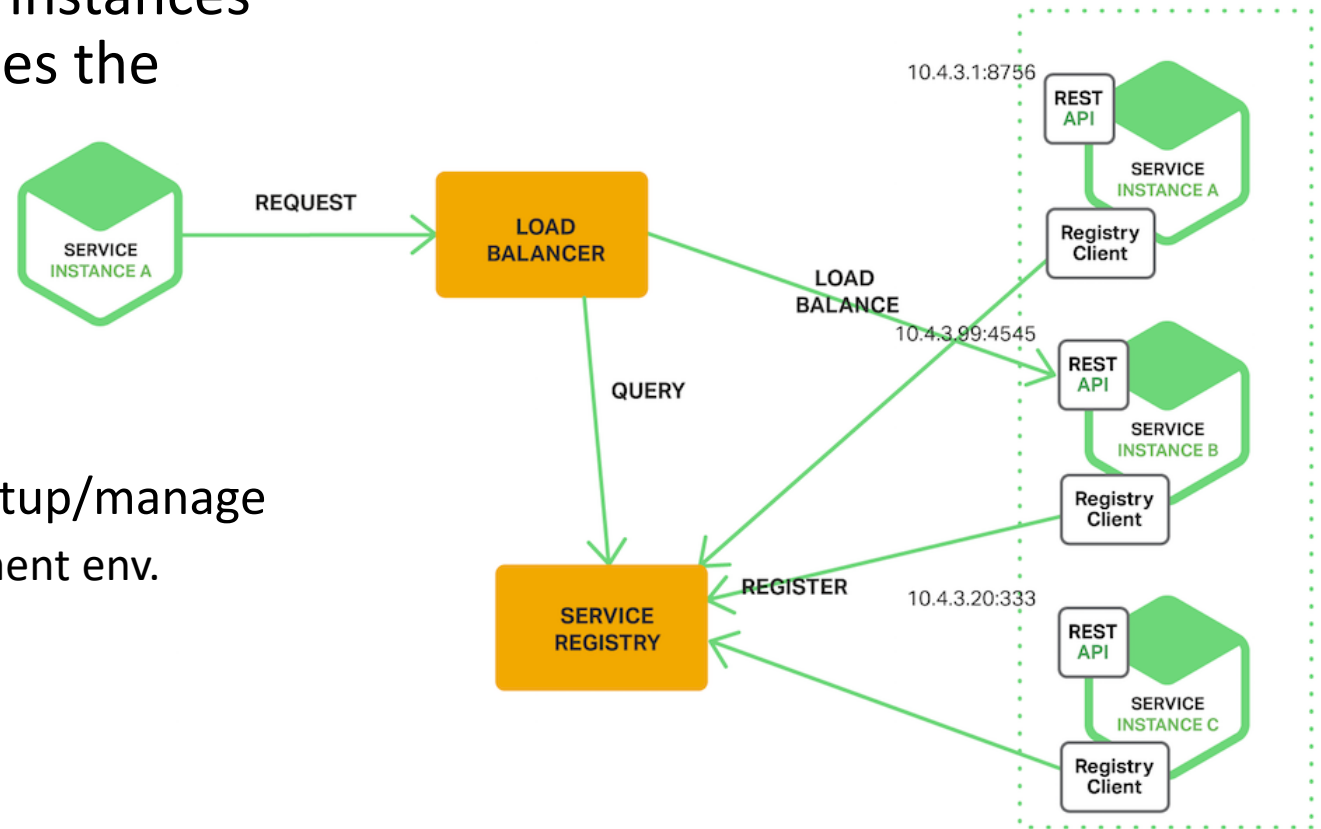
- Client makes request to service instances via a load balancer, which queries the service registry

- Advantages?
  - Discovery is abstracted from clients
    - Clients simply make requests to LB
  - Some environments provide this built-in



# Server-Side Discovery Pattern

- Client makes request to service instances via a load balancer, which queries the service registry



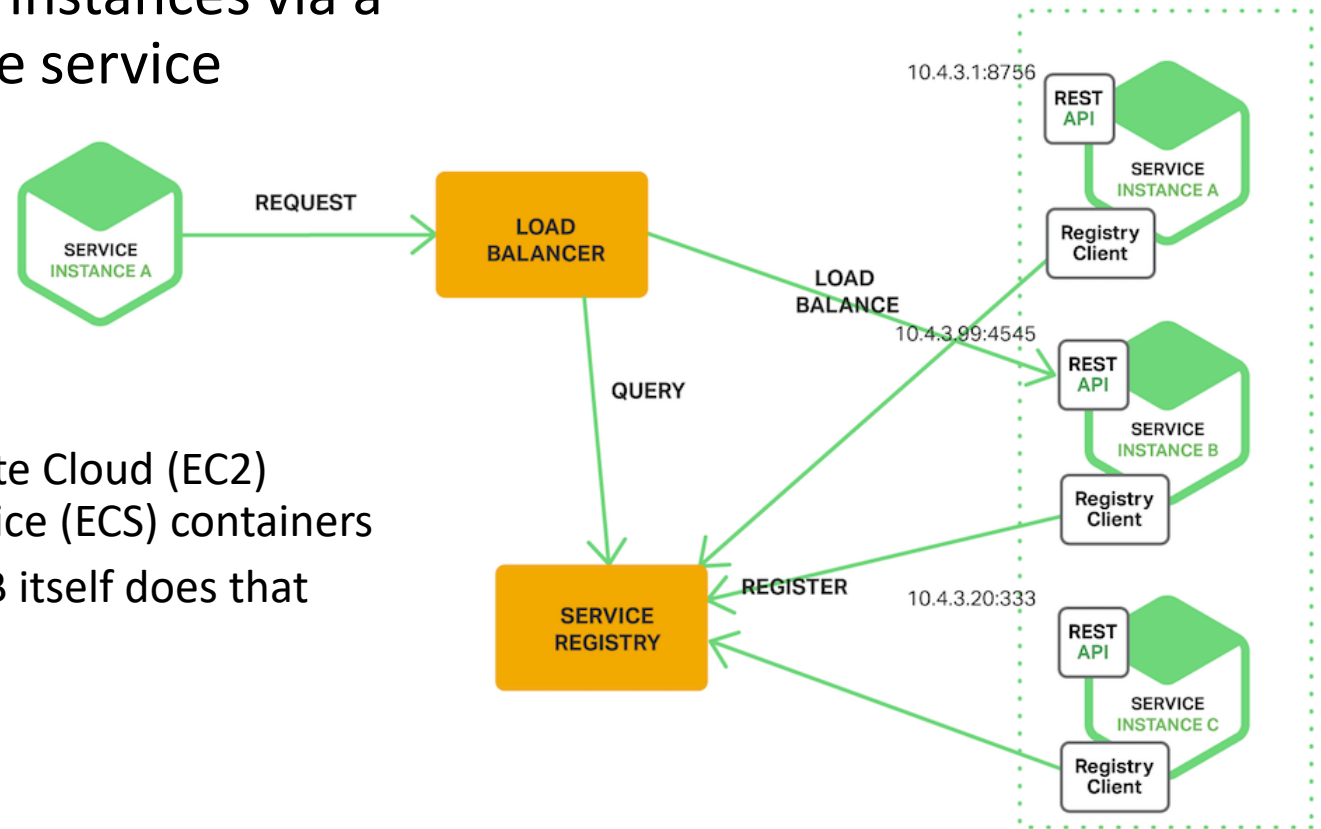
- Disadvantages?
  - LB is yet another HA system to setup/manage
    - Unless provided by the deployment env.

# Server-Side Discovery Pattern

- Client makes request to service instances via a load balancer, which queries the service registry

- Examples

- AWS Elastic Load Balancer (ELB)
  - Balance traffic to Elastic Compute Cloud (EC2) instances or EC2 Container Service (ECS) containers
  - No separate service registry, ELB itself does that

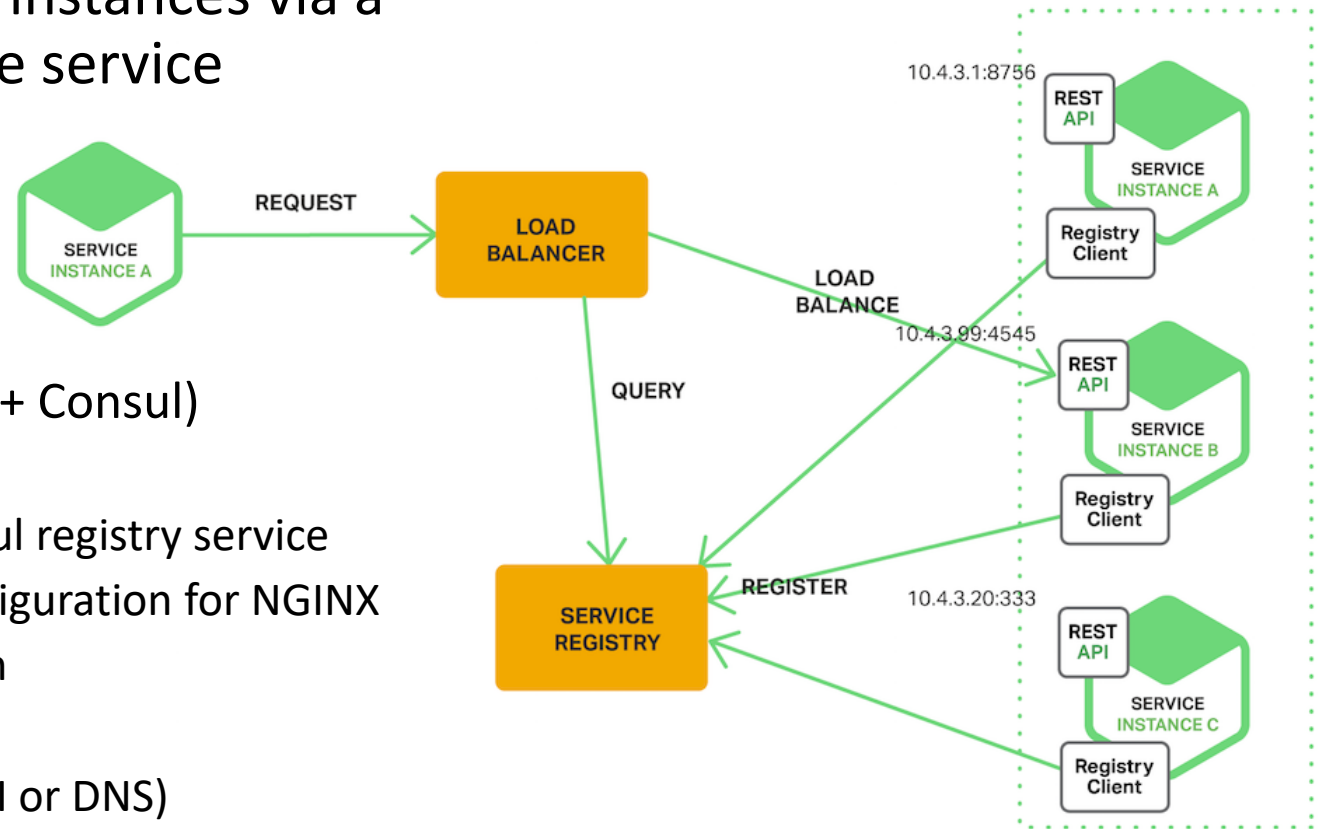


# Server-Side Discovery Pattern

- Client makes request to service instances via a load balancer, which queries the service registry

- Examples

- LB + service registry (e.g., NGINX + Consul)
  - NGINX works as load balancer
  - Instances are registered in consul registry service
  - Consul Template generates configuration for NGINX
  - NGINX reloads the configuration(... or just Consul tools)
- (or NGINX Plus using either its HTTP API or DNS)

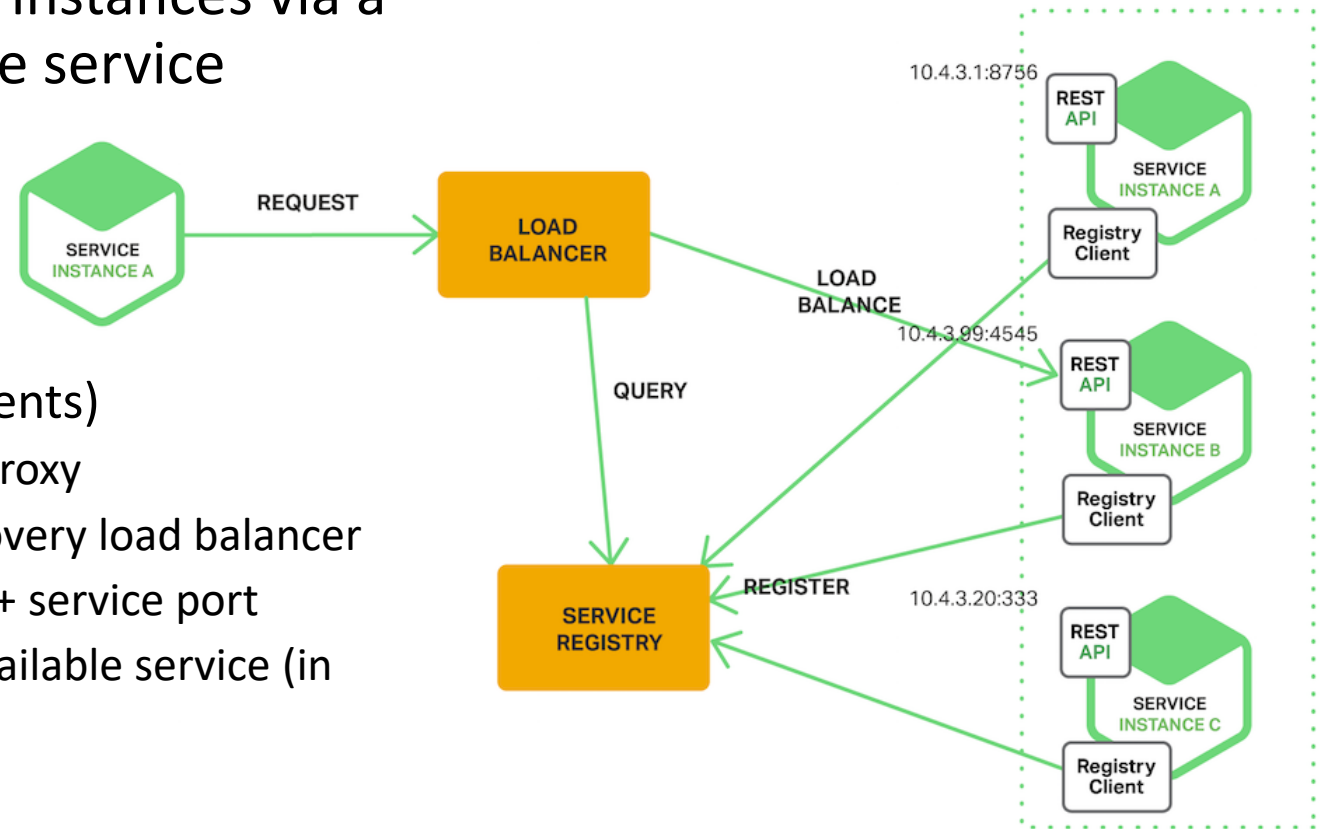


# Server-Side Discovery Pattern

- Client makes request to service instances via a load balancer, which queries the service registry

- Examples

- Kubernetes (and other environments)
  - Each host in the cluster runs a proxy
  - Proxy works as server-side discovery load balancer
  - Client makes request to host IP + service port
  - Proxy forwards request to an available service (in the cluster)



# Service Registry

- Service registry is the core block of service discovery
  - Database of the network locations of available service instances
  - Needs to be highly available and up-to-date
- How does it work?
  - Service that provides a REST API to update and query (or DNS\*) its database
    - Used to register or unregister service instances
    - To query for service instance addresses
    - Other actions (such as heartbeat checks)



# Service Registry

- Service registry is the core block of service discovery
  - Database of the network locations of available service instances
  - Needs to be highly available and up-to-date
- Popular service registries include
  - consul (HashiCorp)
  - etcd
  - Eureka (Netflix)
  - Zookeeper (Apache)
  - In some solutions (e.g., Kubernetes, Docker Swarm, AWS ELB, ...), the service registry is already built-in as part of the system

# Service Registration Options

- Service instances must be registered with and deregistered from the service registry
- How?
  - Self-Registration Pattern
  - Third-Party Registration Pattern

# Self-Registration Pattern

- A service instance is responsible for registering and deregistering itself with the service registry

- Instance may also need to send heartbeat requests to prevent its registration from expiring

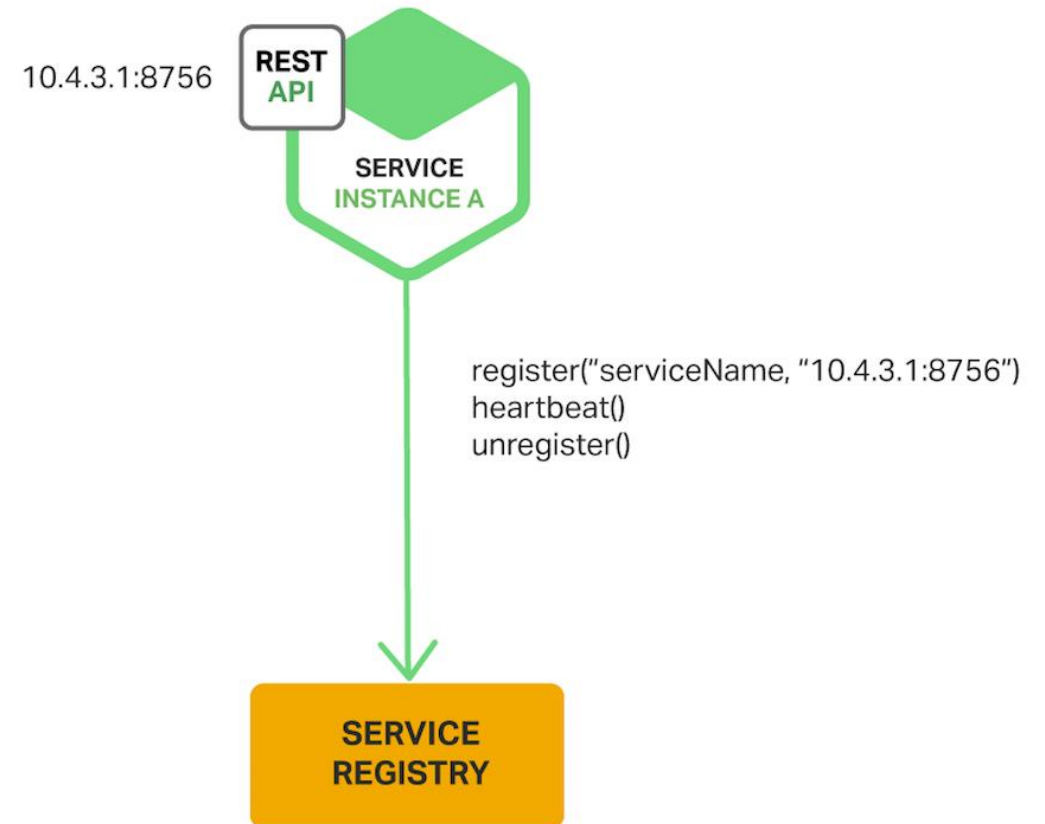
(e.g., Netflix Eureka client)

- **Advantages**

- Does not require additional systems

- **Disadvantages**

- Ties instances with service registration
    - Instances must implement registration

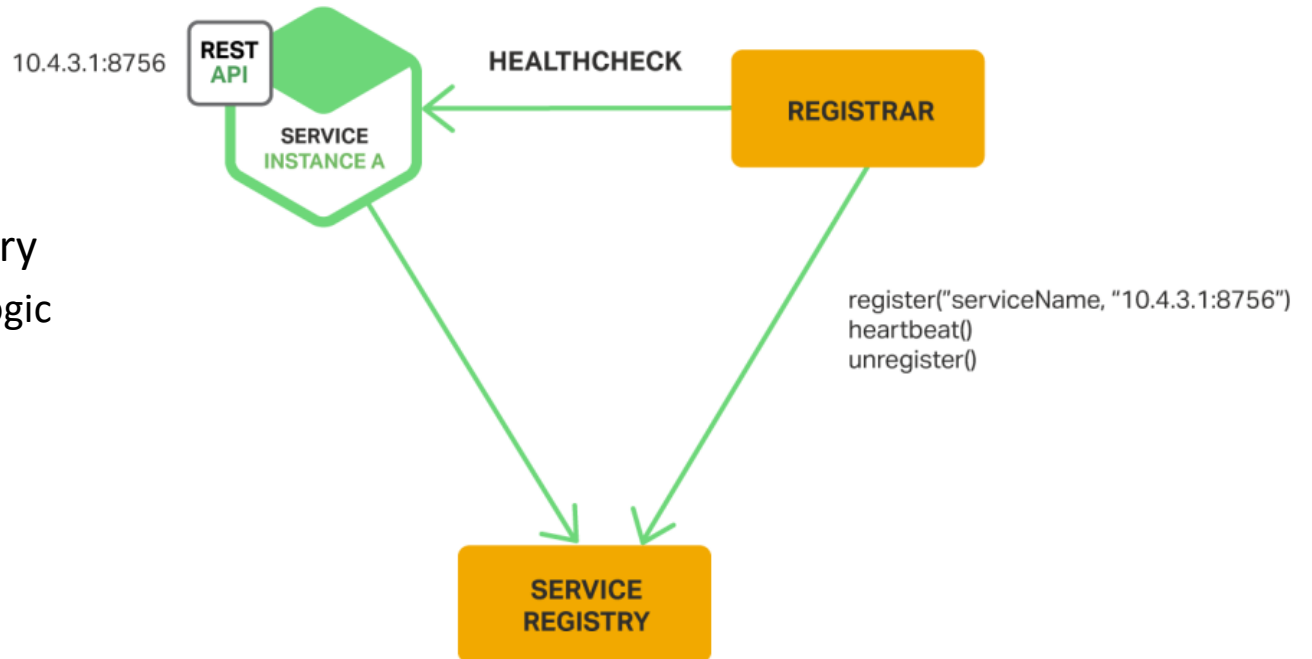


# Third-Party Registration Pattern

- Another system component known as the **service registrar** handles the registration
  - Registrar tracks changes (running instances) by either polling the deployment environment or subscribing to events

- Advantages
  - Services decoupled from service registry
    - They do not implement registration logic

- Disadvantages
  - Yet another HA component to manage

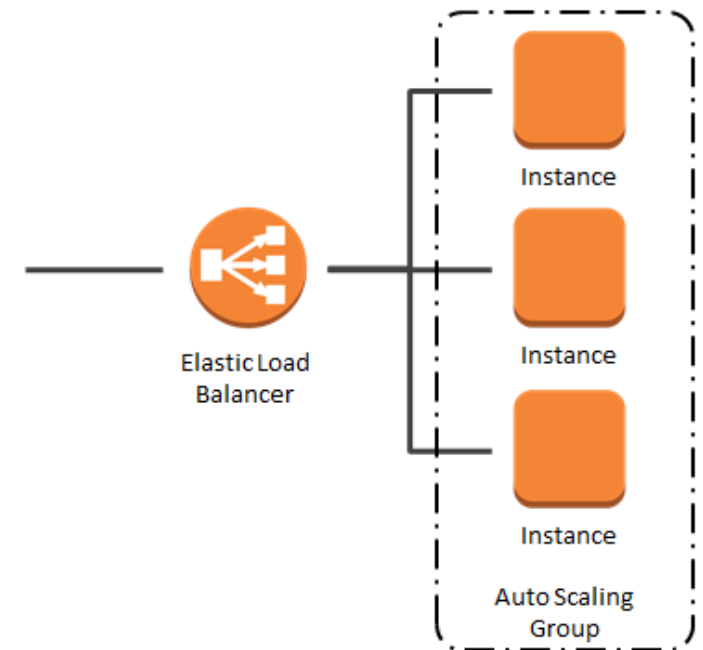


# Third-Party Registration Pattern

- Several **service registrars** exist
  - Registrator project – <https://github.com/gliderlabs/registrator>
    - Automatically registers and deregisters services for any Docker container by inspecting containers as they come online
    - Supports several service registries, including Consul and etcd
  - Netflix Prana – <https://github.com/netflix/Prana>
    - Sidecar app that runs with the service instance
    - Registers and deregisters the service instance with Netflix Eureka
    - Primarily intended for services written in Non-JVM languages

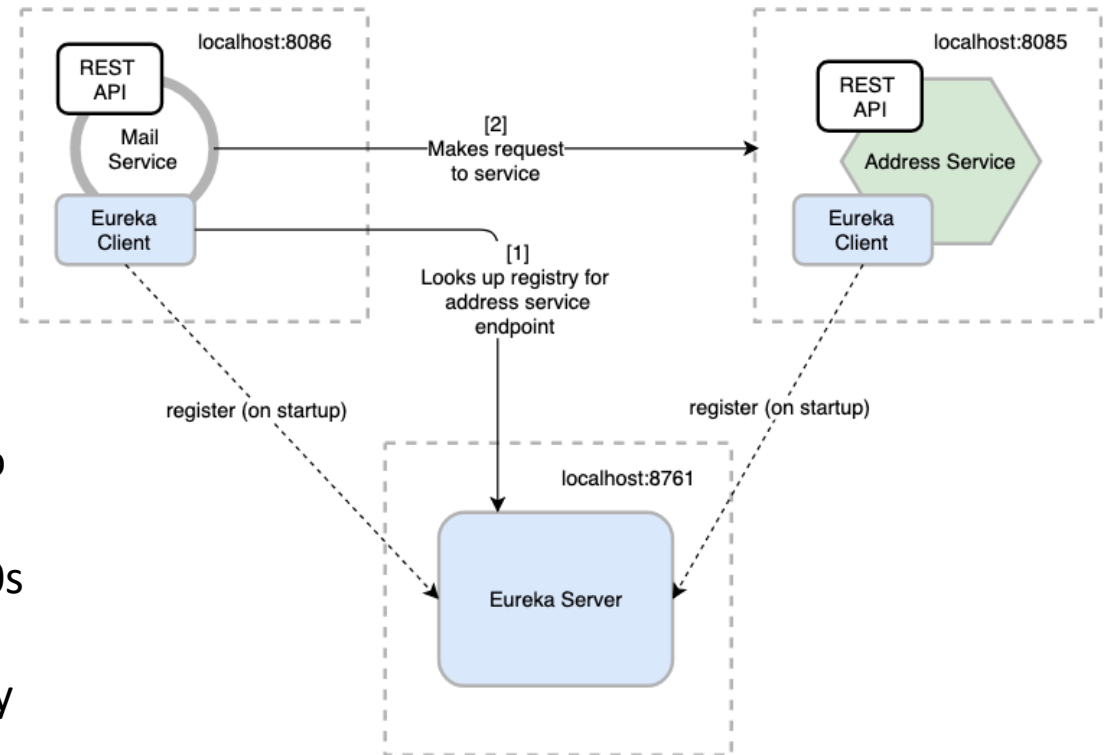
# Third-Party Registration Pattern

- Several **service registrars** exist
  - Deployment environments normally have *service registrar* built-in
    - Amazon EC2 instances created by an Autoscaling Group can be automatically registered with an ELB
    - Kubernetes services are automatically registered and made available for discovery



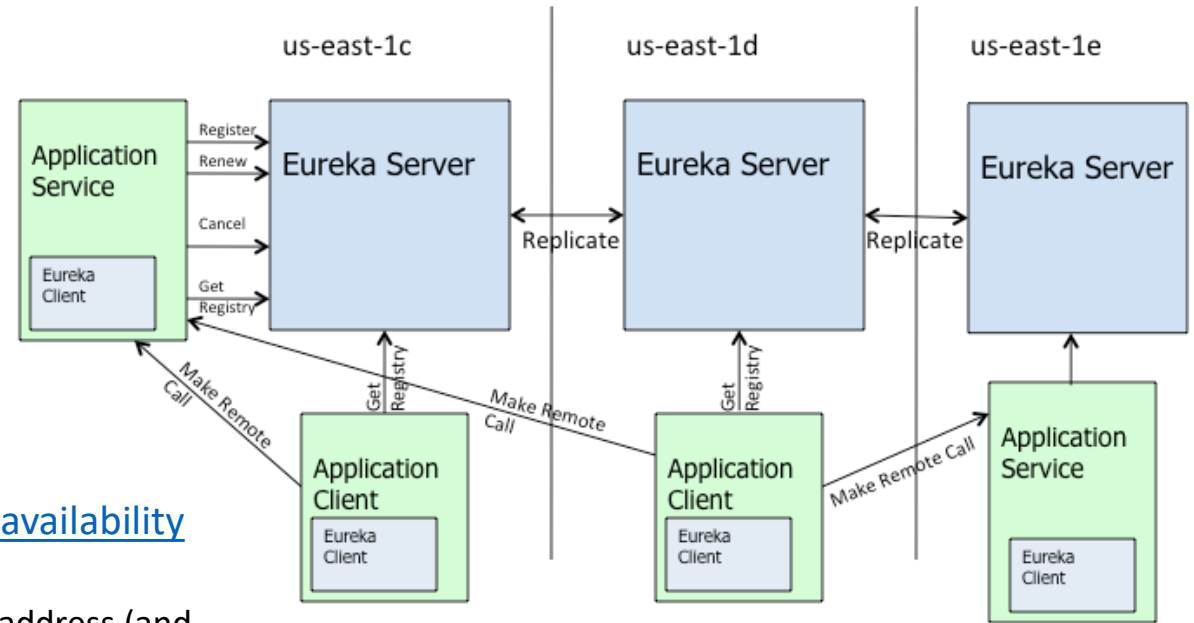
# Service Registry Examples

- Netflix Eureka
  - Service registry used by Netflix
- How does it work?
  - Provides REST API
    - POST request – used by service instances to register themselves
    - PUT request – update registration (every 30s instances must refresh their registration)
    - HTTP DELETE – to remove an instance, or by timing out
    - GET request – retrieve the list of service instances



# Service Registry

- Netflix Eureka
  - Service registry used by Netflix
- How does Netflix achieve HA?
  - One or more Eureka servers in each [Amazon AWS availability zone](#) (AZ)
    - Each runs on an EC2 instance that has an Elastic IP address (and replicate their data)
  - Cluster configuration is stored using DNS TXT records
    - A map from AZ to Eureka servers network locations
  - When a Eureka server starts up
    - Gets the Eureka cluster configuration by querying the DNS TXT records
    - Locates its peers (to sync data)
    - Assigns itself an unused Elastic IP address



```
---
spring:
  profiles: aws

server:
  port: 8761

logging.level.com.netflix.eureka: INFO

eureka:
  environment: production
  instance:
    non-secure-port: ${server.port}
  client:
    registerWithEureka: true
    fetchRegistry: true
    eurekaServerDNSName: eureka-server.us-east-1.amazonaws.com
    eurekaServerPort: 8761
    eurekaServerURLContent: http://eureka-server.us-east-1.amazonaws.com:8761
    useDnsForFetchingServerList: true
  server:
    bindingStrategy: roundRobin
    route53DomainTTL: 300
```

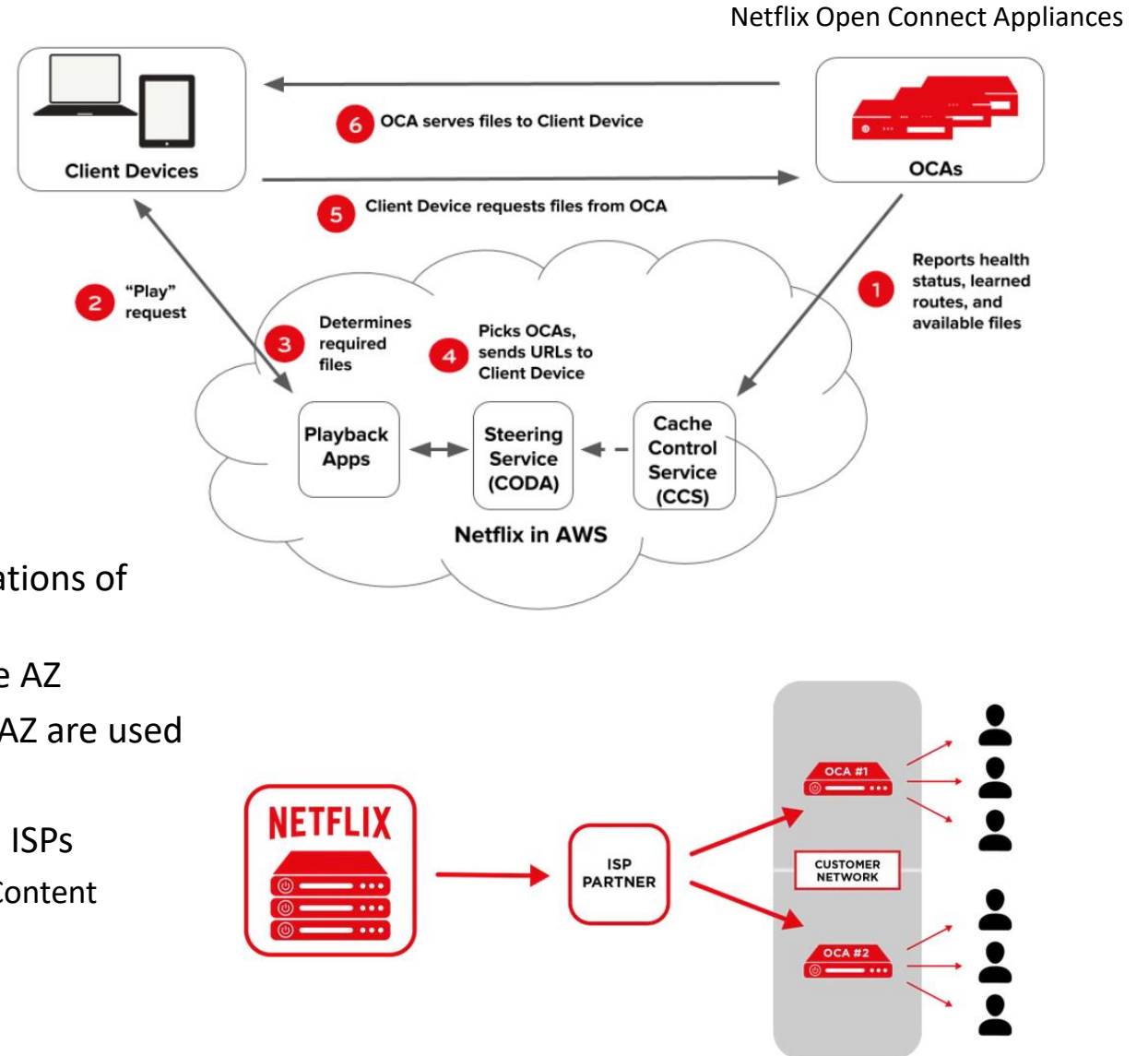
### ROUTE 53 FOR EUREKA

- ▶ TXT record per region to point to AZ in that region  
txt.us-east-1.gr8conf.vpc ->  
us-east-1d.gr8conf.vpc  
us-east-1e.gr8conf.vpc
- ▶ TXT record per AZ in region  
txt.us-east-1d.gr8conf.vpc ->  
ec2-###-###-###-###.compute-1.amazonaws.com  
ec2-###-###-###-###.compute-1.amazonaws.com



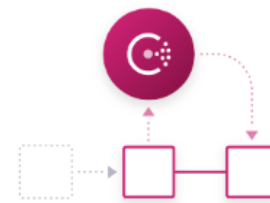
# Service Registry

- Netflix Eureka
  - Service registry used by Netflix
- How does Netflix achieve HA?
  - Eureka clients (services and service clients)
    - Query the DNS to discover the network locations of Eureka servers
    - Clients try to use Eureka servers in the same AZ
    - If none is available, Eureka servers in other AZ are used
  - Video traffic is served from OCAs located @ ISPs
    - [Netflix Open Connect](#) is our purpose-built Content Delivery Network (CDN)



# HashiCorp Consul

- Consul provides more than a simple service registry
  - Service discovery
    - Queried using a DNS interface or an HTTP interface
  - Key/Value storage
    - Enables storing dynamic configuration
  - Service Mesh
    - Enables secure service-to-service communication (TLS + identity-based authorization)
  - Health Checking
  - Multi-Datacenter



*Service Discovery*



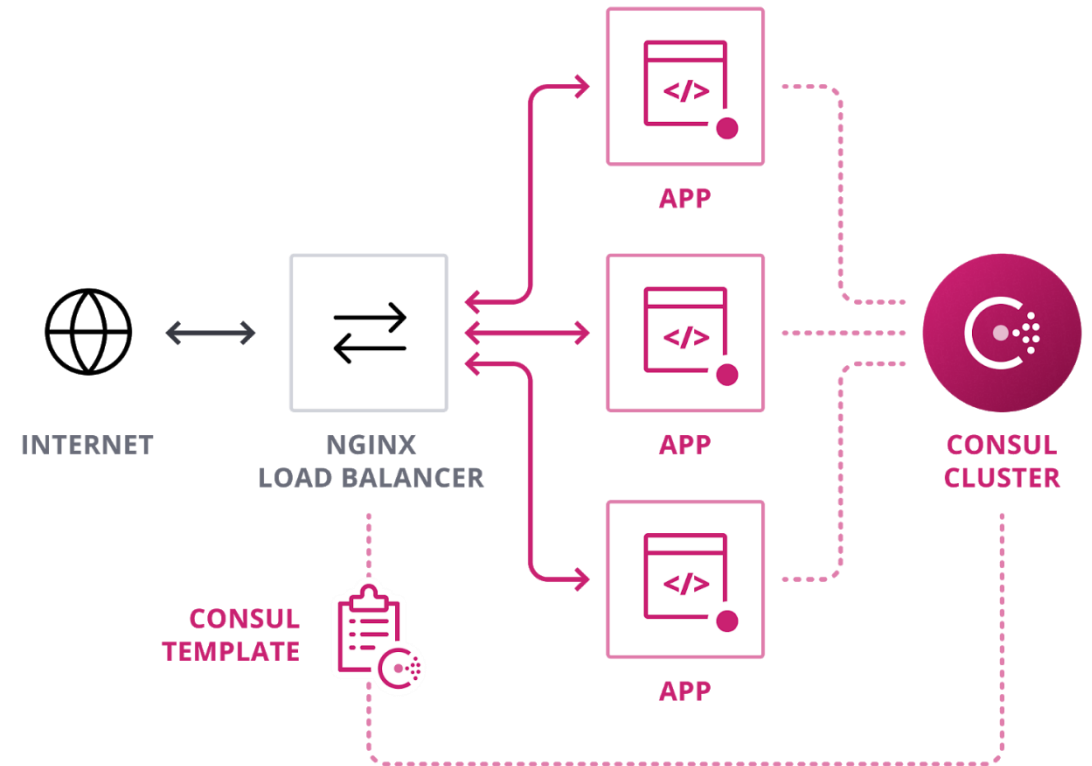
*Service Configuration*



*Service Segmentation*

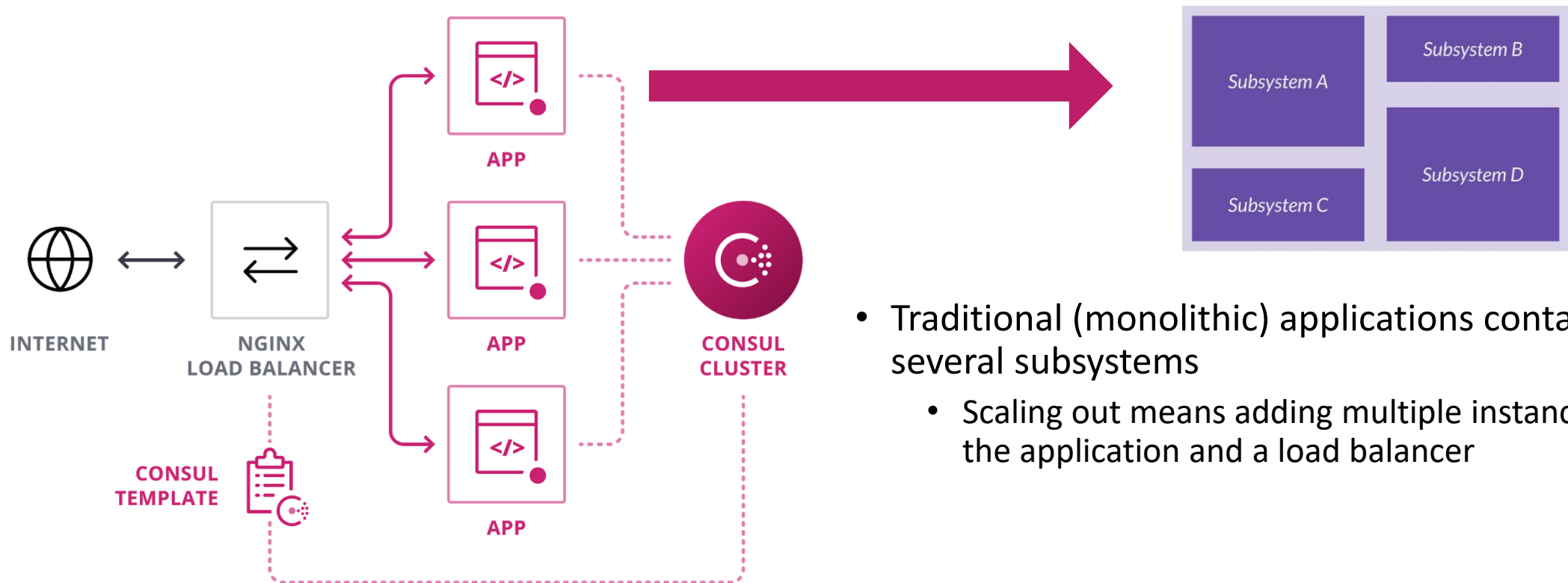
# Consul for Service Discovery

- Consul typical usage
  - Instances register themselves in Consul registry
  - Consul provides service discovery over HTTP or DNS
  - Consul template can be used to generate LB configurations dynamically



# Consul for Service Discovery

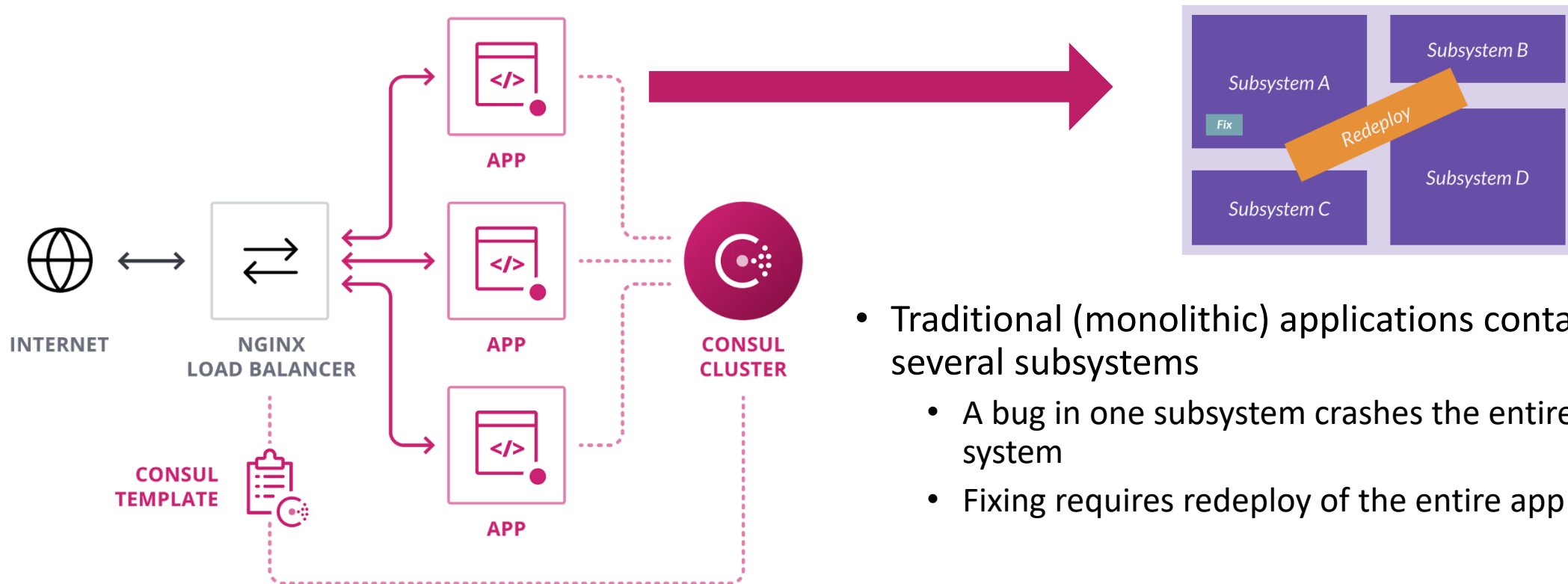
- Nowadays more advanced scenarios are common



- Traditional (monolithic) applications contain several subsystems
  - Scaling out means adding multiple instances of the application and a load balancer

# Consul for Service Discovery

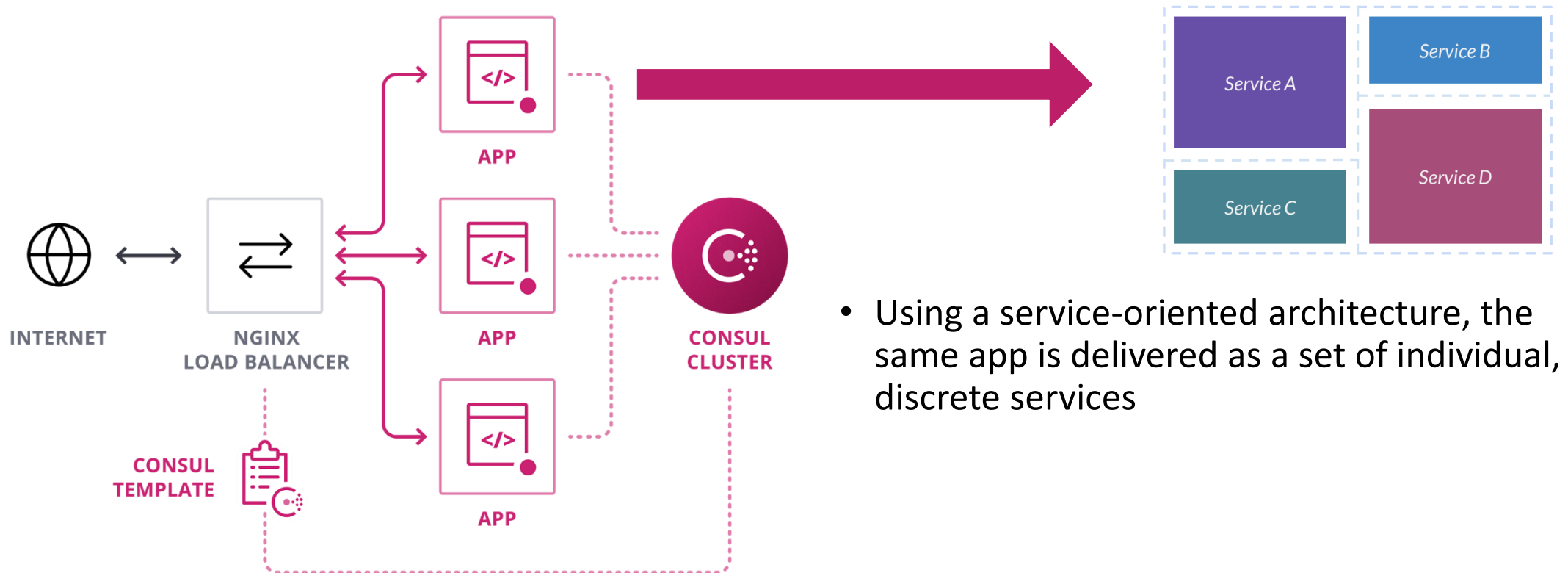
- But nowadays more advanced scenarios are common



- Traditional (monolithic) applications contain several subsystems
  - A bug in one subsystem crashes the entire system
  - Fixing requires redeploy of the entire app

# Consul for Service Discovery

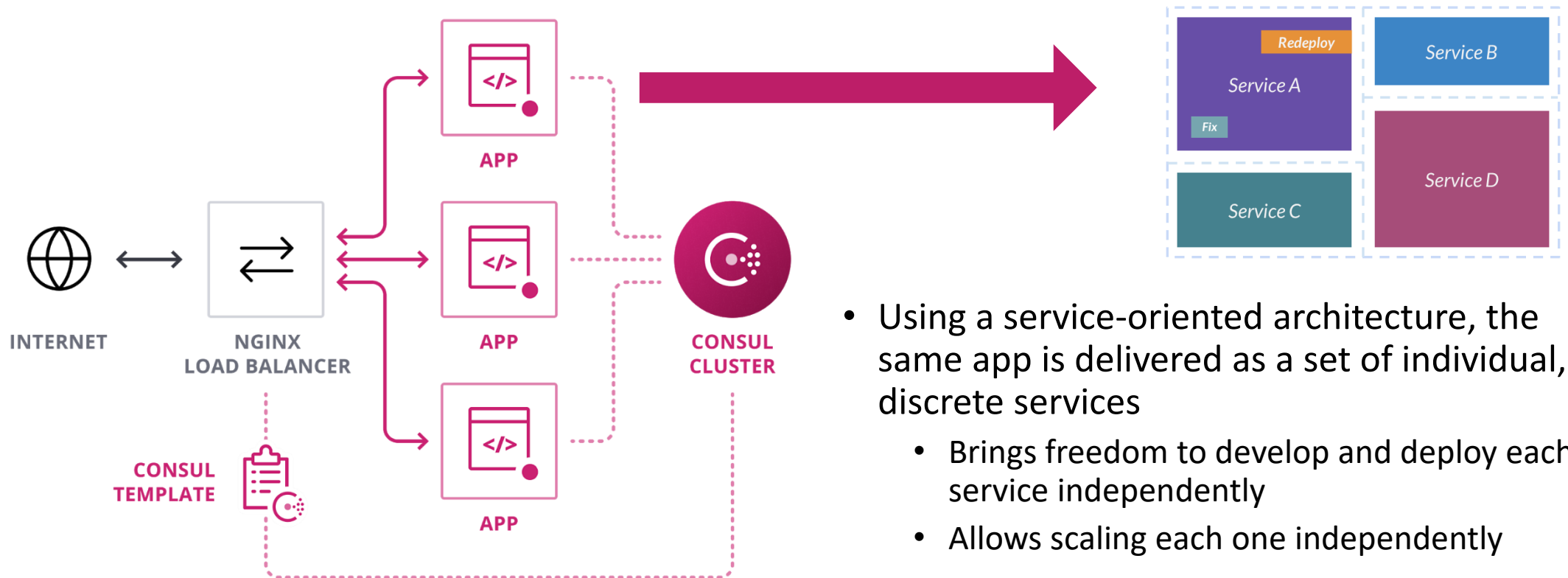
- But nowadays more advanced scenarios are common



- Using a service-oriented architecture, the same app is delivered as a set of individual, discrete services

# Consul for Service Discovery

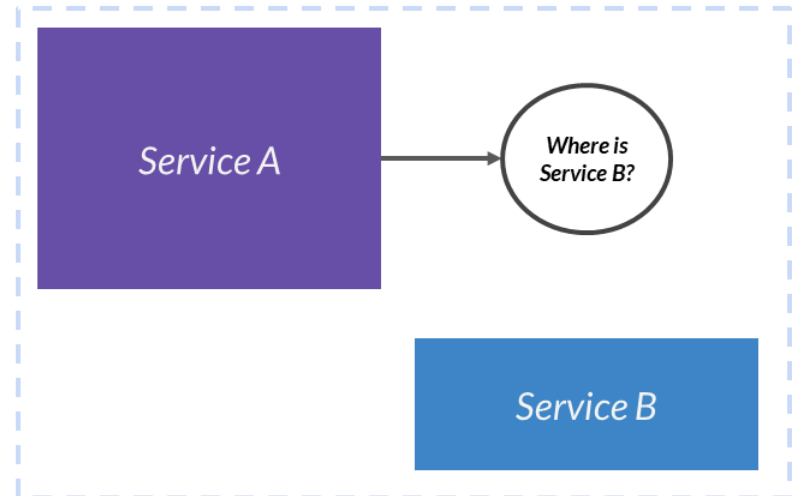
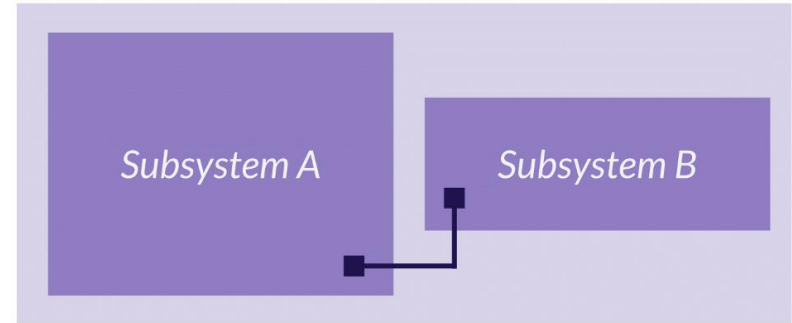
- But nowadays more advanced scenarios are common



- Using a service-oriented architecture, the same app is delivered as a set of individual, discrete services
  - Brings freedom to develop and deploy each service independently
  - Allows scaling each one independently

# Consul for Service Discovery

- Service communication in distributed systems is more complex
  - In monolith applications, simple function calls between subsystems are used
  - In distributed systems this normally means talking over the network
    - Hardcode the address of each service?
    - What about scalability?





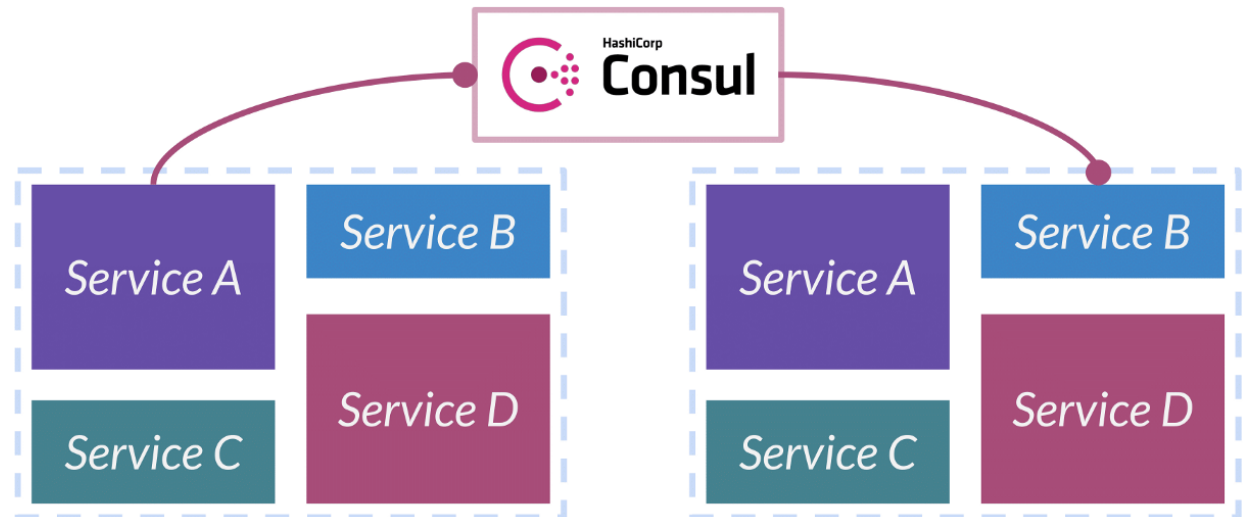
# Consul for Service Discovery

- Service communication in distributed systems
  - Using a central service registry is one possible solution to this problem
    - Contains entries for all upstream services
    - Updated each time a service instance starts or stops
    - Queried using the DNS interface

Service A wants to talk with B?

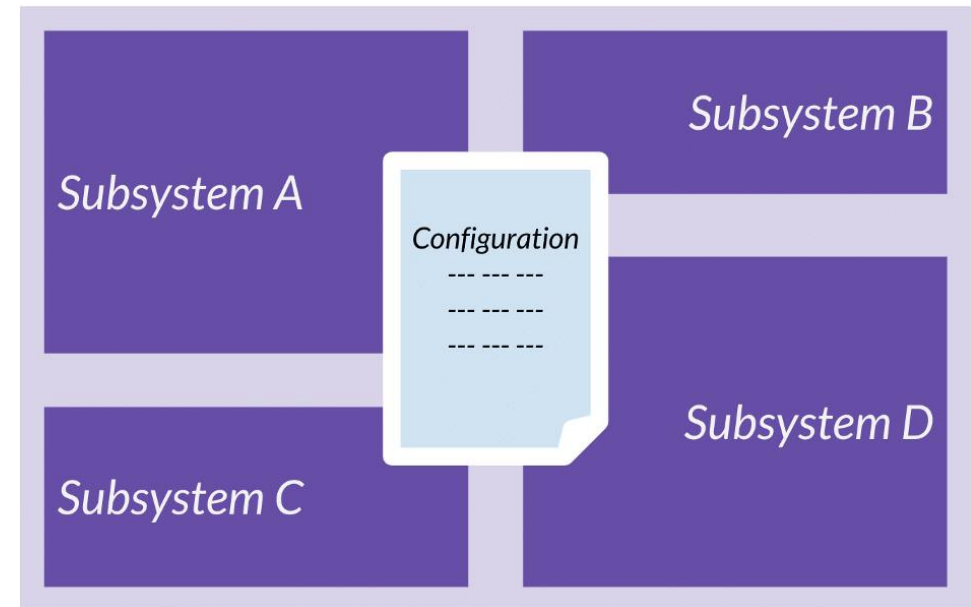
Consul acts as a DNS server

- Returns the IP address of Service B
- Also provides health checks
- Distributes load if B contains multiple instances



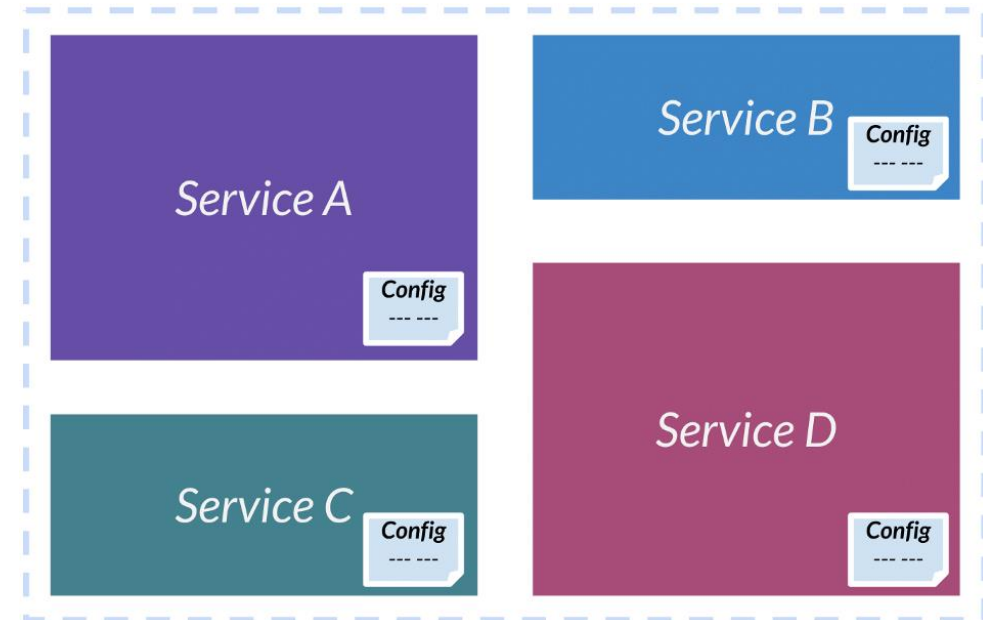
# Consul for Configuration Management

- Application settings are normally stored under YAML, JSON, XML files
  - In monolithic architectures this configures the entire app
  - All subsystems have the same source of configuration



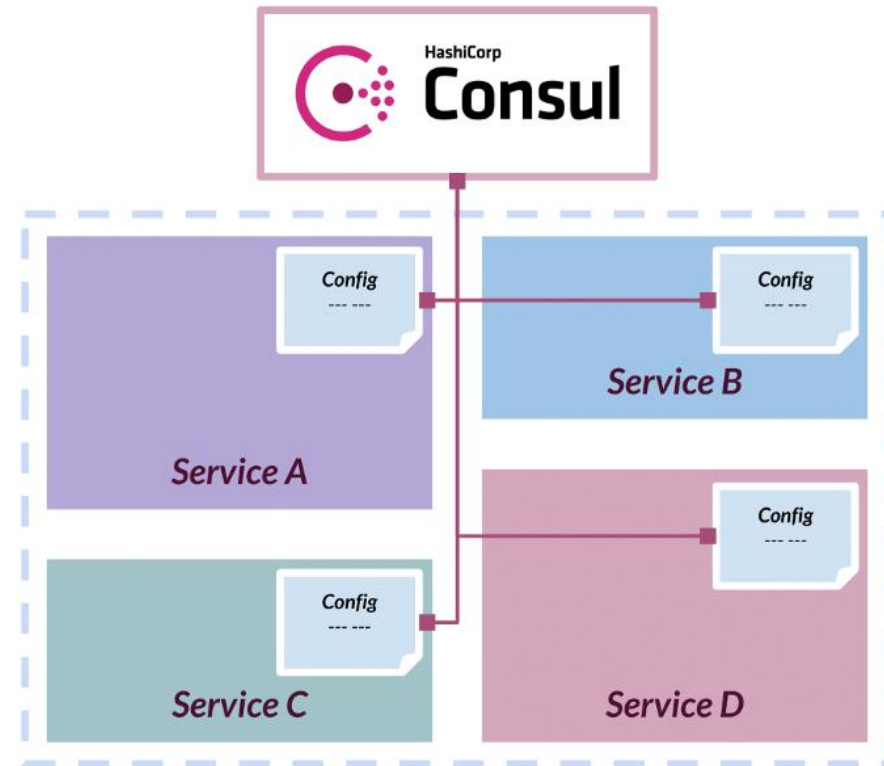
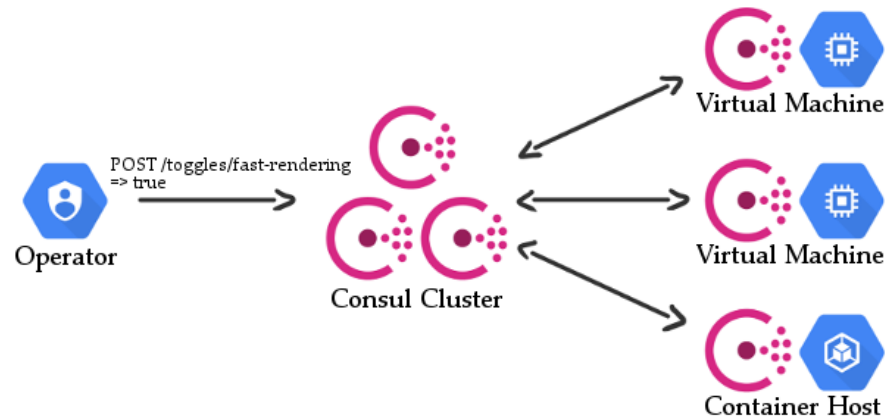
# Consul for Configuration Management

- Application settings are normally stored under YAML, JSON, XML files
  - Moving to distributed systems introduces new challenges since each service has its own configuration
    - How do we maintain consistency?
    - What if the configuration needs to be updated dynamically?



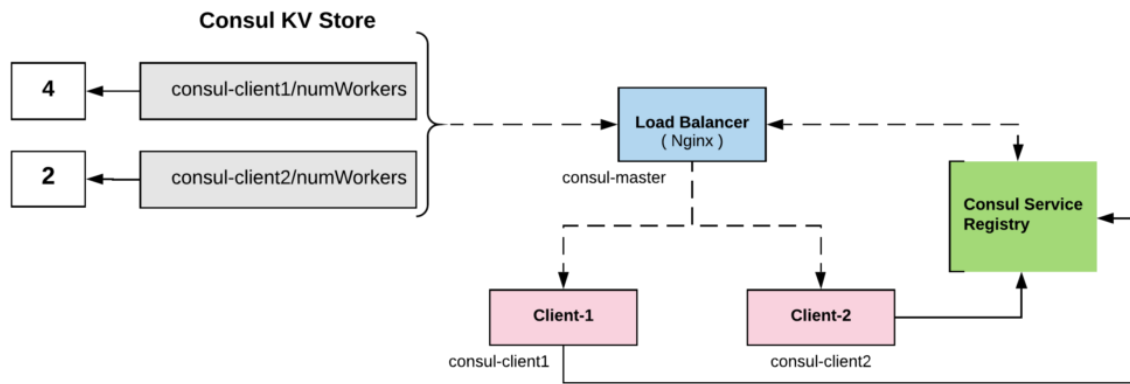
# Consul for Configuration Management

- Applications settings are normally stored under YAML, JSON, XML files
  - Consul provides a central Key-Value to address this
  - Changes are replicated to each node
  - ACL can be used to protect this

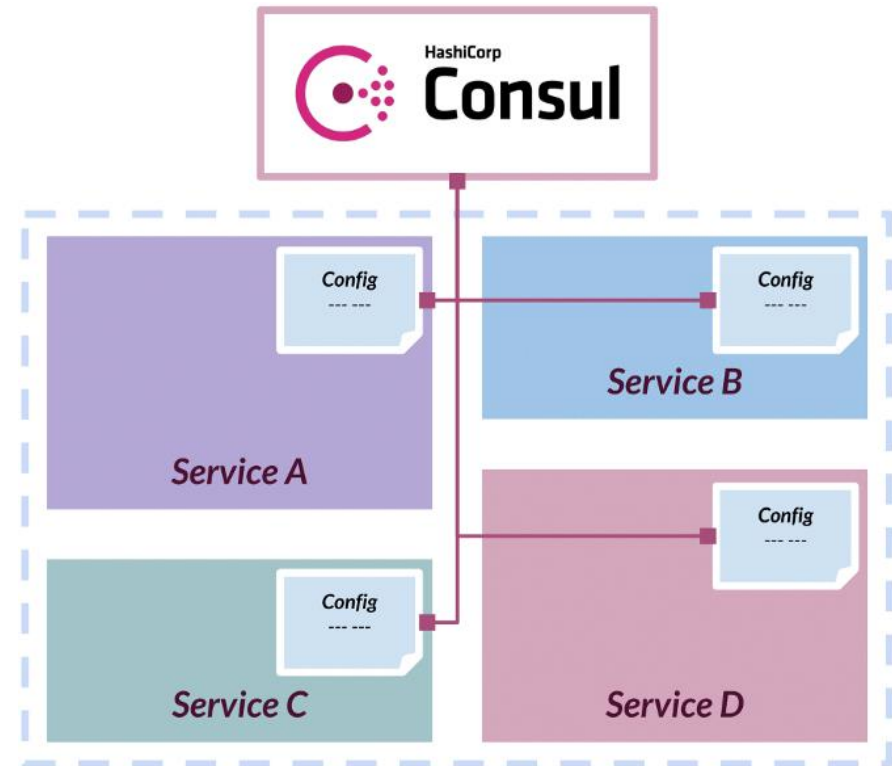


# Consul for Configuration Management

- Applications settings are normally stored under YAML, JSON, XML files
  - Consul provides a central Key-Value to address this
  - Changes are replicated to each node
  - ACL can be used to protect this



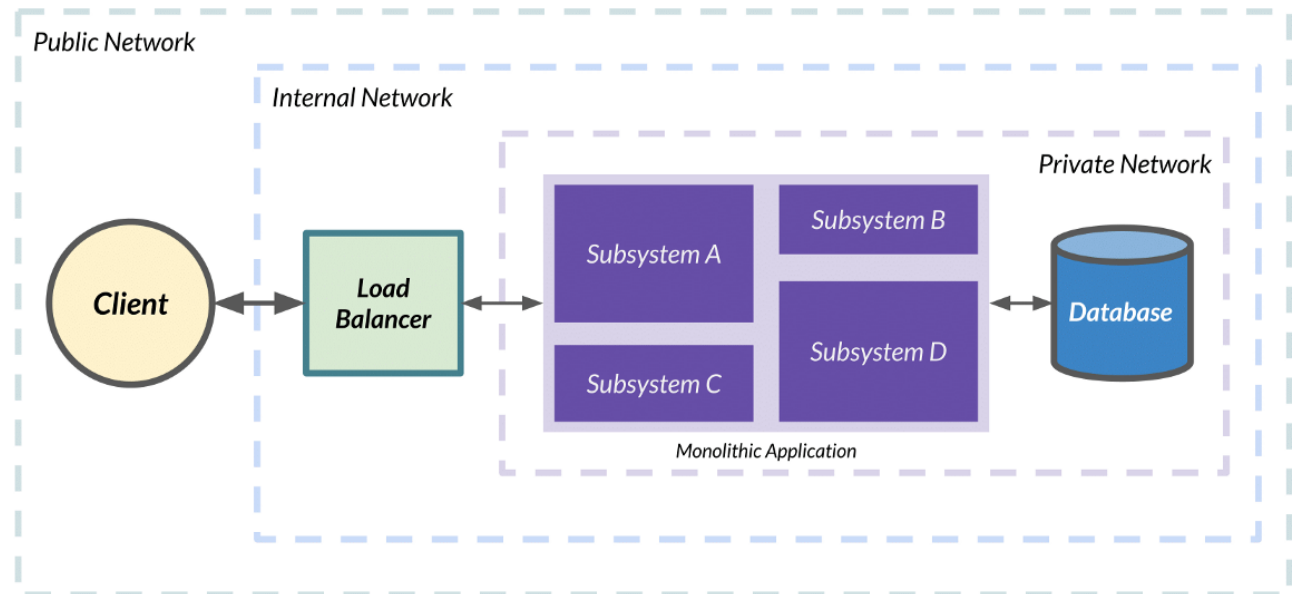
Consul KV Store



# Service Mesh and Consul Connect

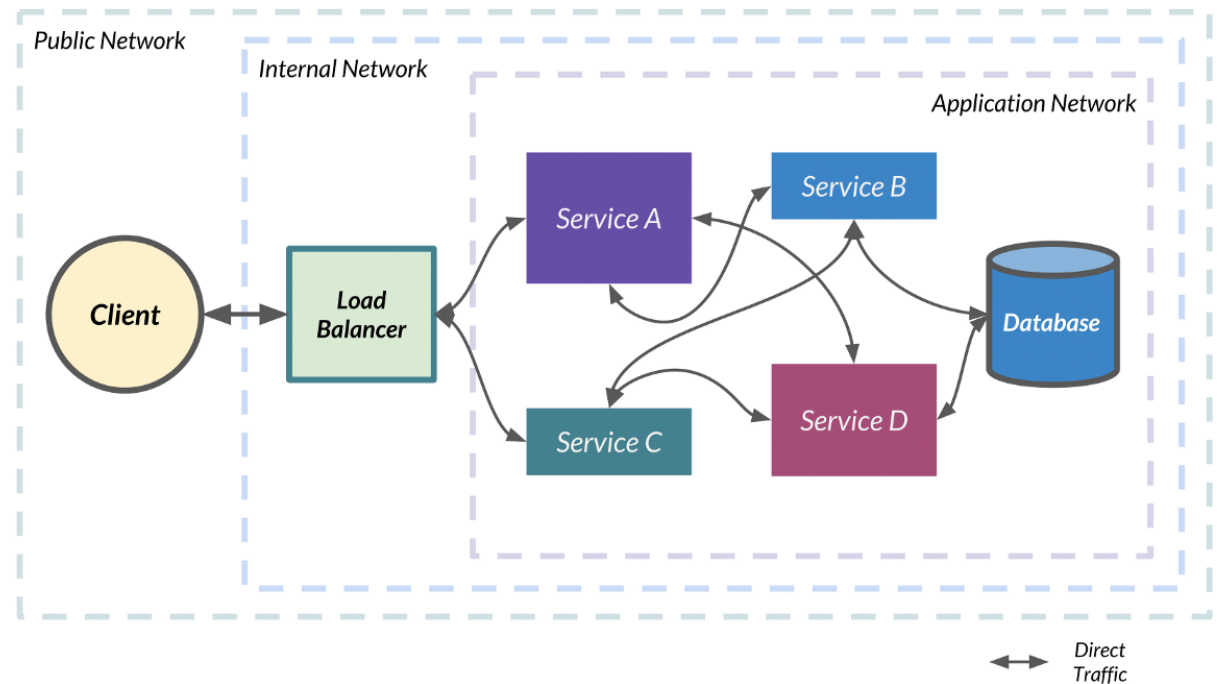
- In a monolithic architecture the network is typically divided into 3 or 4 zones
  - A DMZ zone (internal) with an LB handling internet traffic
  - An internal (private) zone where our apps sit
  - Data can be separated into its own zone

Only the LB reaches the application  
Only the app reaches the DB



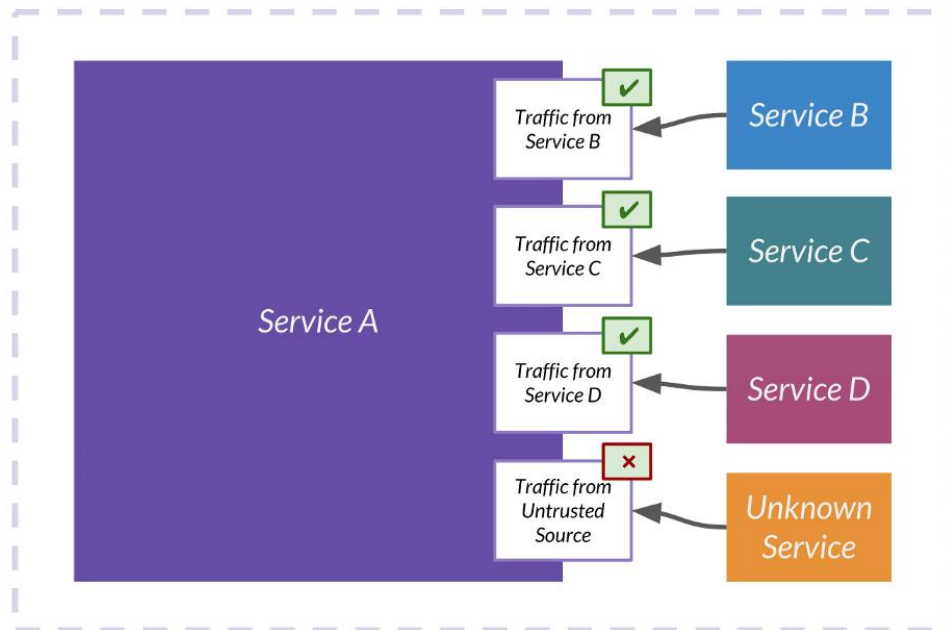
# Service Mesh and Consul Connect

- Communication changes drastically with distributed applications
  - Multiple services in the internal network communicate, posing several challenges
    - Security of communications is a major one



# Service Mesh and Consul Connect

- Communication changes drastically with distributed applications
  - Multiple services in the internal network communicate, posing several challenges
    - Security of communications is a major one



Services should be able to identify that the traffic they are receiving is from a verified and trusted entity in the network.

Ideally, the communications should also be encrypted.

- Not only in the DB but also in transit

Solving this with firewalls and network segmentation is complex.





# Service Mesh and Consul Connect




- Consul solves this with Consul Connect
  - First, a service graph is created to define allowed communications (Intentions)

## Intentions

Create

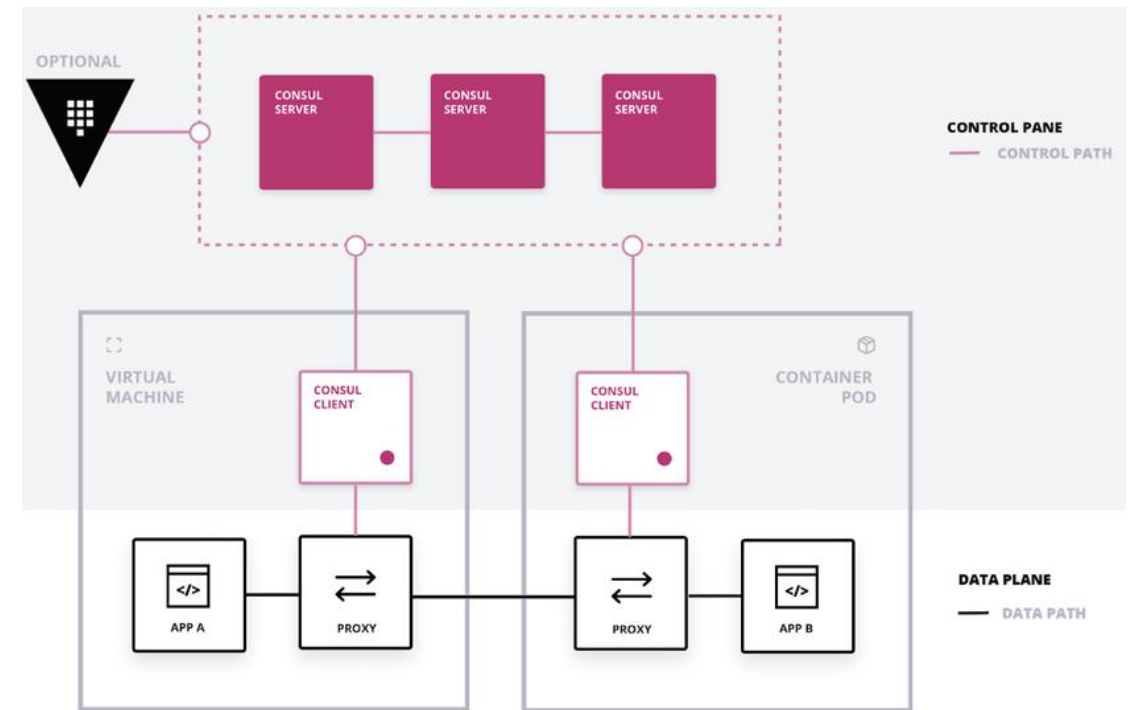
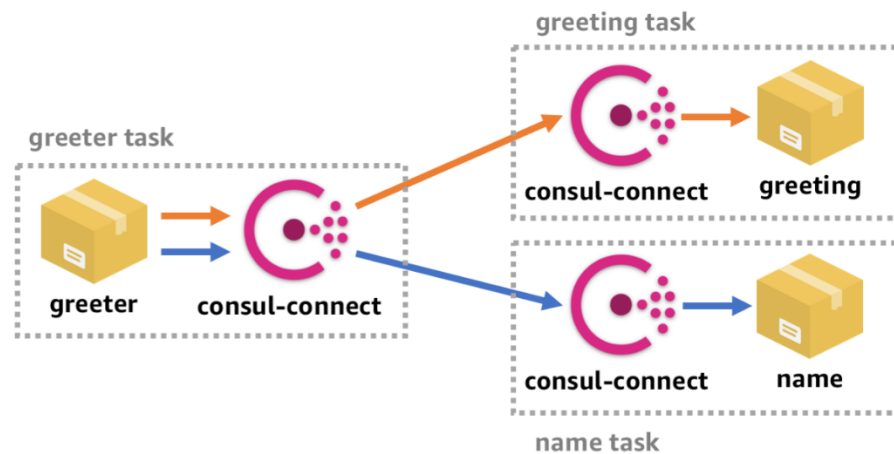
All (10) → Allow (3)  Deny (7)

Search by Source or Destir 

Source		Destination	Precedence	Actions
api		consul	9	...
api	→	postgresql	9	...
api		uuid	9	...
api	→	web	9	...
postgresql		web	9	...

# Service Mesh and Consul Connect

- Consul solves this with Consul Connect
  - Each app instance will have a sidecar proxy (consul connect)
    - Validate connections using Intentions
    - Validate services with certificates
    - Communicates securely using TLS



# Consul Demo

- Consul is just a binary (<https://www.consul.io/downloads.html>)
  - Download and unzip
    - wget [https://releases.hashicorp.com/consul/1.7.2/consul\\_1.7.2\\_linux\\_amd64.zip](https://releases.hashicorp.com/consul/1.7.2/consul_1.7.2_linux_amd64.zip)
    - unzip consul\_1.7.2\_linux\_amd64.zip
    - move consul binary to PATH
      - echo \$PATH
      - E.g., mv consul /usr/bin/ or export PATH="\$PATH:/path/to/consul"
- Check install

```
vagrant@1b01:~$ consul
Usage: consul [--version] [--help] <command> [<args>]

Available commands are:
  acl          Interact with Consul's ACLs
  agent        Runs a Consul agent
  catalog      Interact with the catalog
  config       Interact with Consul's Centralized Configuration
```

# Consul Demo

- Starting consul as an agent in development mode (not secure nor scalable)
  - Also started as a server
  - Claimed leadership

```
==> Consul agent running!
2020-04-12T19:35:45.579Z [INFO] agent: Started gRPC server: address=127.0.0.1:8502 network=tcp
2020-04-12T19:35:45.616Z [WARN] agent.server.raft: heartbeat timeout reached, starting election: last-leader=
2020-04-12T19:35:45.617Z [INFO] agent.server.raft: entering candidate state: node="Node at 127.0.0.1:8300 [Candidate]" term=2
2020-04-12T19:35:45.617Z [DEBUG] agent.server.raft: votes: needed=1
2020-04-12T19:35:45.617Z [DEBUG] agent.server.raft: vote granted: from=6d02f565-caad-1b49-b74b-a410f522f4a3 term=2 tally=1
2020-04-12T19:35:45.617Z [INFO] agent.server.raft: election won: tally=1
2020-04-12T19:35:45.617Z [INFO] agent.server.raft: entering leader state: leader="Node at 127.0.0.1:8300 [Leader]"
2020-04-12T19:35:45.618Z [INFO] agent.server: cluster leadership acquired
2020-04-12T19:35:45.618Z [INFO] agent.server: New leader elected: payload=1b01
```

```
vagrant@1b01:~$ consul agent -dev
==> Starting Consul agent...
    Version: 'v1.7.2'
    Node ID: '6d02f565-caad-1b49-b74b-a410f522f4a3'
    Node name: '1b01'
    Datacenter: 'dc1' (Segment: '<all>')
    Server: true (Bootstrap: false)
    Client Addr: [127.0.0.1] (HTTP: 8500, HTTPS: -1, gRPC: 8502, DNS: 8600)
    Cluster Addr: 127.0.0.1 (LAN: 8301, WAN: 8302)
    Encrypt: Gossip: false, TLS-Outgoing: false, TLS-Incoming: false, Auto-Encrypt-TLS: false

==> Log data will now stream in as it occurs:

2020-04-12T19:35:45.559Z [DEBUG] agent: Using random ID as node ID: id=6d02f565-caad-1b49-b74b-a410f522f4a3
2020-04-12T19:35:45.563Z [DEBUG] agent.tlsutil: Update: version=1
2020-04-12T19:35:45.564Z [DEBUG] agent.tlsutil: OutgoingRPCWrapper: version=1
2020-04-12T19:35:45.565Z [INFO] agent.server.raft: initial configuration: index=1 servers="[{"Suffrage":Voter ID:6d02f565-caad-1b49-b74b-a410f522f4a3 Address:127.0.0.1:8300}]"
2020-04-12T19:35:45.566Z [INFO] agent.server.serf.wan: serf: EventMemberJoin: 1b01.dc1 127.0.0.1
2020-04-12T19:35:45.567Z [INFO] agent.server.serf.lan: serf: EventMemberJoin: 1b01 127.0.0.1
2020-04-12T19:35:45.568Z [INFO] agent: Started DNS server: address=127.0.0.1:8600 network=udp
2020-04-12T19:35:45.568Z [INFO] agent.server.raft: entering follower state: follower="Node at 127.0.0.1:8300 [Follower]" leader=
2020-04-12T19:35:45.572Z [INFO] agent.server: Adding LAN server: server="1b01 (Addr: tcp/127.0.0.1:8300) (DC: dc1)"
2020-04-12T19:35:45.573Z [INFO] agent.server: Handled event for server in area: event=member-join server=1b01.dc1 area=wan
2020-04-12T19:35:45.575Z [INFO] agent: Started DNS server: address=127.0.0.1:8600 network=tcp
2020-04-12T19:35:45.578Z [INFO] agent: Started HTTP server: address=127.0.0.1:8500 network=tcp
2020-04-12T19:35:45.578Z [INFO] agent: started state syncer
==> Consul agent running!
```

# Consul Demo

- Check members (in a second console)

```
vagrant@lb01:~$ consul members
Node Address      Status Type   Build Protocol DC Segment
lb01 127.0.0.1:8301 alive server 1.7.2 2      dc1 <all>
vagrant@lb01:~$
```

- We can check cluster members using the HTTP API

```
vagrant@lb01:~$ curl localhost:8500/v1/catalog/nodes
[
  {
    "ID": "a8877770-9059-b4e5-c6a6-c581ec911d6c",
    "Node": "lb01",
    "Address": "127.0.0.1",
    "Datacenter": "dc1",
    "TaggedAddresses": {
      "lan": "127.0.0.1",
      "lan_ipv4": "127.0.0.1",
      "wan": "127.0.0.1",
      "wan_ipv4": "127.0.0.1"
    },
    "Meta": {
      "consul-network-segment": ""
    },
    "CreateIndex": 10,
    "ModifyIndex": 11
  }
]
```

# Consul Demo

- Check members (in a second console)

```
vagrant@lb01:~$ consul members
Node Address      Status Type   Build Protocol DC   Segment
lb01 127.0.0.1:8301 alive  server 1.7.2  2     dc1  <all>
vagrant@lb01:~$
```

- Or do the same using the DNS interface

```
vagrant@lb01:~$ dig @localhost -p 8600 lb01.node.consul

; <<>> DiG 9.10.3-P4-Ubuntu <<>> @localhost -p 8600 lb01.node.consul
; (2 servers found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 42529
;; flags: qr aa rd; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 2
;; WARNING: recursion requested but not available

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags::; udp: 4096
;; QUESTION SECTION:
;lb01.node.consul.                IN      A

;; ANSWER SECTION:
lb01.node.consul.                0       IN      A      127.0.0.1

;; ADDITIONAL SECTION:
lb01.node.consul.                0       IN      TXT     "consul-network-segment="

;; Query time: 1 msec
;; SERVER: 127.0.0.1#8600(127.0.0.1)
;; WHEN: Sun Apr 12 19:52:15 UTC 2020
;; MSG SIZE rcvd: 97
```

# Consul Demo

- Stopping the agent is achieved with “consul leave”

```
vagrant@1b01:~$ consul leave
Graceful leave complete
```

- “consul leave” != killing the agent
  - **Leave** notifies other members that the agent left the datacenter
    - Node services and checks are removed from consul catalog
  - **Killing** the agent indicates a node failure
    - Node health marked as critical
    - Node not removed from catalog
    - Consul will try to reconnect with the node

```
2020-04-12T19:59:24.455Z [ERROR] agent.server.autopilot: Error updating cluster health: error="error getting server raft protocol versions: No servers found"
2020-04-12T19:59:26.453Z [ERROR] agent.server.autopilot: Error updating cluster health: error="error getting server raft protocol versions: No servers found"
2020-04-12T19:59:27.668Z [INFO] agent: Requesting shutdown
2020-04-12T19:59:27.669Z [INFO] agent.server: shutting down server
2020-04-12T19:59:27.669Z [DEBUG] agent.leader: stopping routine: routine="CA root pruning"
2020-04-12T19:59:27.670Z [DEBUG] agent.leader: stopped routine: routine="CA root pruning"
2020-04-12T19:59:27.671Z [INFO] agent: consul server down
2020-04-12T19:59:27.673Z [INFO] agent: shutdown complete
2020-04-12T19:59:27.674Z [DEBUG] agent.http: Request finished: method=PUT url=/v1/agent/leave from=127.0.0.1:43254 latency=11.011668258s
2020-04-12T19:59:27.680Z [INFO] agent: Stopping server: protocol=DNS address=127.0.0.1:8600 network=tcp
2020-04-12T19:59:27.680Z [INFO] agent: Stopping server: protocol=DNS address=127.0.0.1:8600 network=udp
2020-04-12T19:59:27.681Z [INFO] agent: Stopping server: protocol=HTTP address=127.0.0.1:8500 network=tcp
2020-04-12T19:59:28.181Z [INFO] agent: Waiting for endpoints to shut down
2020-04-12T19:59:28.182Z [INFO] agent: Endpoints down
2020-04-12T19:59:28.182Z [INFO] agent: Exit code: code=0
vagrant@1b01:~$
```

# Consul Demo

- A service is registered by providing a service definition
  - Using the HTTP API or creating a file

```
vagrant@lb01:~$ cat /etc/consul.d/web.json
{"service":
  {
    "name": "web",
    "tags": ["rails"],
    "port": 80
  }
}
```

- How?
  - Create service file (e.g., under /etc/consul.d/)
  - Start the server
    - Service “web” is synced
      - In this case, the service is not running yet, just registered.

```
vagrant@lb01:~$ consul agent -dev -enable-local-script-checks -config-dir=/etc/c
onsul.d/
==> Starting Consul agent...
      Version: 'v1.7.2'
      Node ID: '2703315d-c2d7-589c-1236-71e978a011d5'
      Node name: 'lb01'
      Datacenter: 'dc1' (Segment: '<all>')
      Server: true (Bootstrap: false)
      Client Addr: [127.0.0.1] (HTTP: 8500, HTTPS: -1, gRPC: 8502, DNS: 8600)
```

```
Managed automatically. CHECK-SERVICES
2020-04-12T22:07:55.518Z [INFO] agent: Synced node info
2020-04-12T22:07:55.519Z [INFO] agent: Synced service: service=web
2020-04-12T22:07:55.520Z [DEBUG] agent: Node info is sync
```



# Consul Demo

- Querying services
  - Using the DNS interface
    - Querying by SRV returns the PORT

```
vagrant@lb01:~$ dig @127.0.0.1 -p 8600 web.service.consul SRV
```

```
;; ANSWER SECTION:
web.service.consul. 0      IN      SRV     1 1 80 lb01.node.dc1.consul.
;; ADDITIONAL SECTION:
lb01.node.dc1.consul. 0      IN      A       127.0.0.1
```

```
vagrant@lb01:~$ dig @127.0.0.1 -p 8600 web.service.consul

; <<>> DiG 9.10.3-P4-Ubuntu <<>> @127.0.0.1 -p 8600 web.service.consul
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 9548
;; flags: qr aa rd; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1
;; WARNING: recursion requested but not available

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;web.service.consul.          IN      A
;; ANSWER SECTION:
web.service.consul. 0      IN      A       127.0.0.1

;; Query time: 1 msec
;; SERVER: 127.0.0.1#8600(127.0.0.1)
;; WHEN: Sun Apr 12 22:15:24 UTC 2020
;; MSG SIZE rcvd: 63
```

- Currently we are advertising the loopback address
  - This can be changed with “-advertise” flag or using “address” in service definition

# Consul Demo

- Querying services
  - Using the DNS interface
    - TAGS can also be used to filter queries
    - <TAG>.<NAME>.service.consul

```
vagrant@1b01:~$ dig @127.0.0.1 -p 8600 rails.web.service.consul SRV
; <<>> DiG 9.10.3-P4-Ubuntu <<>> @127.0.0.1 -p 8600 rails.web.service.consul SRV
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 28325
;; flags: qr aa rd; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 3
;; WARNING: recursion requested but not available

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:: udp: 4096
;; QUESTION SECTION:
;rails.web.service.consul.      IN      SRV

;; ANSWER SECTION:
rails.web.service.consul. 0      IN      SRV      1 1 80 1b01.node.dc1.consul.

;; ADDITIONAL SECTION:
1b01.node.dc1.consul. 0      IN      A      127.0.0.1
1b01.node.dc1.consul. 0      IN      TXT     "consul-network-segment="

;; Query time: 1 msec
;; SERVER: 127.0.0.1#8600(127.0.0.1)
;; WHEN: Sun Apr 12 22:23:13 UTC 2020
;; MSG SIZE rcvd: 145
```

# Consul Demo

- Querying services
  - Using the HTTP API
  - HTTP API lists all nodes hosting a service
  - Filter healthy is done with
    - .../service/web?passing

```
vagrant@1b01:~$ curl http://localhost:8500/v1/catalog/service/web
[
  {
    "ID": "2703315d-c2d7-589c-1236-71e978a011d5",
    "Node": "1b01",
    "Address": "127.0.0.1",
    "Datacenter": "dc1",
    "TaggedAddresses": {
      "lan": "127.0.0.1",
      "lan_ipv4": "127.0.0.1",
      "wan": "127.0.0.1",
      "wan_ipv4": "127.0.0.1"
    },
    "NodeMeta": {
      "consul-network-segment": ""
    },
    "ServiceKind": "",
    "ServiceID": "web",
    "ServiceName": "web",
    "ServiceTags": [
      "rails"
    ],
    "ServiceAddress": "",
    "ServiceWeights": {
      "Passing": 1,
      "Warning": 1
    },
    "ServiceMeta": {},
    "ServicePort": 80,
    "ServiceEnableTagOverride": false,
    "ServiceProxy": {
      "MeshGateway": {},
      "Expose": {}
    },
    "ServiceConnect": {},
    "CreateIndex": 12,
    "ModifyIndex": 12
  }
]
```

# Consul Demo

- Updating services
  - Change service definition files
  - Run *consul reload*
  - Alternatively, HTTP API can be used

```
web.json
{
  "service": {
    "name": "web",
    "tags": ["rails"],
    "port": 80,
    "check": {
      "args": ["curl", "localhost"],
      "interval": "10s"
    }
  }
}
```

```
vagrant@lb01:~$ consul reload
Configuration reload triggered
```

```
2020-04-12T22:36:18.000Z [DEBUG] agent: Service in sync: service=web
2020-04-12T22:36:18.000Z [DEBUG] agent: Check in sync: check=service:web
2020-04-12T22:36:21.193Z [WARN] agent: Check is now critical: check=service
:web
2020-04-12T22:36:31.216Z [WARN] agent: Check is now critical: check=service
:web
2020-04-12T22:36:41.250Z [WARN] agent: Check is now critical: check=service
:web
```

# Consul Demo

- In our example, DNS queries now return no services
  - Health check is working ok since the defined service is not running.

```
vagrant@lb01:~$ dig @127.0.0.1 -p 8600 web.service.consul

; <<>> DiG 9.10.3-P4-Ubuntu <<>> @127.0.0.1 -p 8600 web.service.consul
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 53448
;; flags: qr aa rd; QUERY: 1, ANSWER: 0, AUTHORITY: 1, ADDITIONAL: 1
;; WARNING: recursion requested but not available


;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags;; udp: 4096
;; QUESTION SECTION:
;web.service.consul.                IN      A

;; AUTHORITY SECTION:
consul.                0        IN      SOA     ns.consul. hostmaster.consul. 15
86731257 3600 600 86400 0

;; Query time: 1 msec
;; SERVER: 127.0.0.1#8600(127.0.0.1)
;; WHEN: Sun Apr 12 22:40:57 UTC 2020
;; MSG SIZE rcvd: 97
```

# Consul Demo

- Consul also provides a web UI to manage the Cluster
  - In production you need to run consul with “-ui”
  - E.g., “`consul agent -dev -enable-local-script-checks -config-dir=/etc/consul.d/ -bind 192.168.33.100 -client 192.168.33.100`”
  - `http://192.168.33.100:8500/ui`

 dc1 <b>Services</b> Nodes Key/Value ACL Intentions			
<b>Services</b> 2 total			
service:name tag:name status:critical search-term			
Service	Health Checks ⓘ		Tags
consul	✓ 1		
web	✓ 1	✗ 1	rails