Doctor Consul

The Manual

Authored by:

Josh Wolfer
Sr. Technical Product Manager
HashiCorp

v. First Draft 09-08-2023

Table of Contents

Abstract

HashiCorp Consul is an enterprise-grade service discovery and service mesh tool. Doctor Consul is an environment that automagically sets up a fairly complex implementation of Consul Enterprise with many “fake” applications with the intent to provide:

- A “real-ish world” implementation of enterprise-grade service networking.
- Working configuration examples to pick and pull from as needed.
- A quickly and reliably provisioned playground to try out new Consul features.
- Lots of inline comments within the configurations that demystify some of Consul’s complexity or “quirks”.

Doctor Consul is heavily and quickly evolving over time. With rapid improvements, comes volatility in the accuracy of the documentation for it. This guide is intended to be a point of reference for the parts of Doctor Consul that are “stable” and less likely to change. There will inevitably be content added to Doctor Consul that has not had a chance to be properly documented yet.

Doctor Consul has many different configuration options and can be used in its entirety or paired down into smaller components. The architecture is discussed in detail further in this document.

Doctor Consul can be found in the following GitHub repository:

> GitHub: https://github.com/joshwolfer/doctorconsul

This guide is written with the assumption that you are already familiar with Consul and service networking in general. In many cases, care is taken to provide as many useful details as possible as they pertain to the features and applications used in Doctor Consul.

For a better understanding of the fundamentals of Consul, please refer to the official documentation and HVDs (HashiCorp Validated Designs).

> HVDs are a relatively new project within HashiCorp and may not be available until late 2023.

First Things First

Formatting Guide

When applicable, the following color scheme is used to differentiate between various infrastructure types:

```
Consul Partition
Consul Namespace
Consul Cluster (datacenter)
Consul Peer
Kubernetes Cluster
Kubernetes Context
```

Green boxes: Links to GitHub content

> GitHub:

Yellow boxes: Important noteworthy information

>

Grey boxes: Code, Commands, and Outputs

```
$
```

Architectural Overview

Consul is a multi-platform tool. Although it can run on many platforms, such as Windows, Linux, Kube-native (with a Helm chart and CRDs), ECS, and more, there are two primary platforms: VMs (Linux) and Kubernetes.

Doctor Consul provides infrastructure and applications in BOTH styles of configuration.

The VM-style environment utilizes docker-compose. While everything is launched in containers, the configuration files are provided in HCL and are a 1:1 representation of the syntax used with Consul on actual VMs.

The Kubernetes environment has a few different modular options. By default, four Kubernetes clusters are built locally in K3d. K3d is a lightweight, Kubernetes distribution based on Rancher K3s. Doctor Consul alternatively supports configuring 4 existing Kubernetes clusters in AWS EKS, and soon GCP GKE.

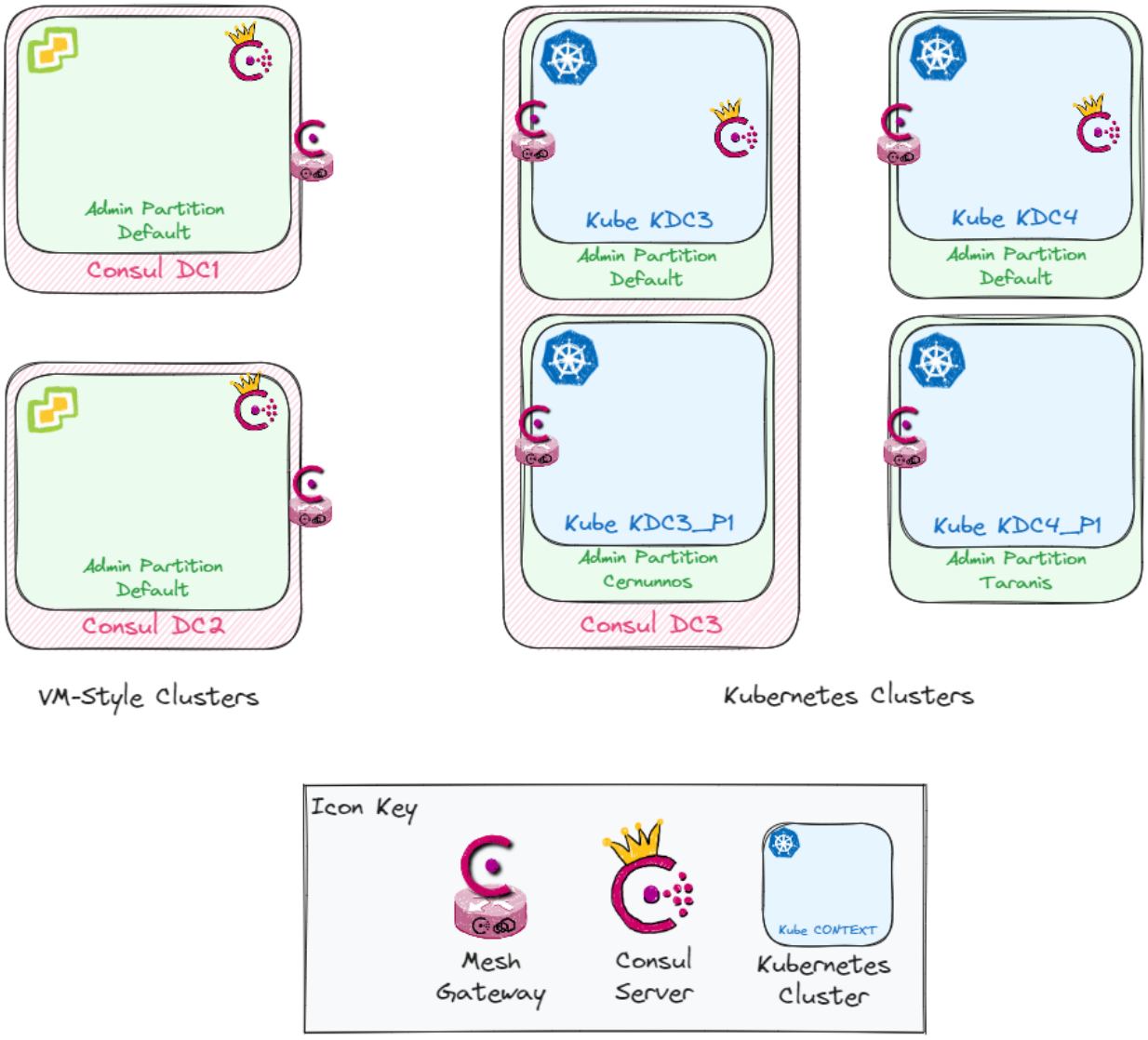
The two different styles of running Consul are NOT duplicate environments. You can choose to run one or the other or both. They are complementary to each other and part of a larger integrated architecture in which applications are spanned across all of the clusters in different ways. Since Kubernetes is by far the most popular way to use service mesh, many users only ever spin up the Kubernetes portion of Doctor Consul.

There are four Consul clusters (datacenters) in total:

- 2 Consul clusters in VM-style: DC1 and DC2
- 2 Consul clusters in Kubernetes: DC3 and DC4
    - Each Kubernetes cluster has an additional child Admin Partition cluster, making 4 total Kubernetes clusters.

Cluster peering is used to federate between select Consul clusters and partitions.

The basic worldview of Doctor Consul is below. If every detail were to be placed in a single diagram, it would be an overwhelming ball of spaghetti (believe me, I tried). This is merely a simplified view and much more detail is provided in the respective sections of this manual.



^^^ *Very simplified view of the Doctor Consul Architecture*

Doctor Consul: Pre-Reqs

Doctor Consul requires many tools to be installed prior to being run. This section provides a list of general tools needed depending on whether Doctor Consul is run in VM or Kubernetes style. Unique requirements for each of the types of Kubernetes deployments are mentioned in each respective section further below.

Generic Requirements

Doctor Consul

Clone the Doctor Consul repository:

> GitHub: https://github.com/joshwolfer/doctorconsul

The core of Doctor Consul is essentially a bunch of elaborate Bash scripts and should be run from the cloned directory.

HashiCorp Consul Enterprise

A HashiCorp Consul Enterprise license is required. The primary goal of Doctor Consul is to utilize Consul within an enterprise-grade capacity. Consul OSS lacks scalability and flexibility features and is therefore not used at all.

> The Consul Enterprise license contents must be placed in a file "`./license`" in the root of the repository.
>
> The Consul Enterprise binary must installed and in the execution `PATH`.

The Consul Enterprise binary can be attained from https://releases.hashicorp.com/consul/.

VM-Style Requirements

Docker Compose

The VM-style Environment requires Docker-Compose.

> ***!! MAC M1 USERS !!***
> The Docker images referenced in the `docker-compose.yml` are AMD64, not ARM64.
>
> M1 users will need to build their own ARM64 consul+envoy images using https://github.com/joshwolfer/convoy-build and modify the 3 Docker variable files

(`./docker-configs/docker_vars`) to point the "`CONVOY_IMAGE`" variable to these new images.

Kubernetes-Style Requirements (General)

Kubectl

- Installation instructions HERE

Helm

- Helm is used to configure and install Consul into Kubernetes.
- Installation instructions HERE

k9s (Optional)

- It's highly recommended to get k9s to make navigating Kubernetes a million times easier.
- https://github.com/derailed/k9s/releases

HashiCorp consul-k8s CLI

- Installation instructions HERE

Kubernetes-Style Requirements (K3d)

- K3d is a dockerized version of K3s, which is a simple version of Rancher Kubernetes.
- K3d can be used for the local Kubernetes portion of this environment.
- Installation instructions HERE

Kubernetes-Style Requirements (AWS EKS)

- Doctor Consul Supports using 4 existing AWS EKS clusters
    - This is an alternative to using k3d locally.
- Requires using Terraform + https://github.com/ramramhariram/EKSonly to provision the four EKS clusters
- See specific "Building Doctor Consul: Kubernetes (AWS EKS)" instructions below.
- Requires:
    - AWS CLI version 2 HERE
    - Terraform OSS HERE

Standard Linux Tools

- sed
- jq

Building Doctor Consul: VMs (Docker Compose)

<< Low priority content. This is coming last >>

Startup Instructions

<< Low priority content. This is coming last >>

Setup: Post-config Script

<< Low priority content. This is coming last >>

Building Doctor Consul: Kubernetes (Local K3d)

Launching the K3d-based local Kubernetes environment “should be” very simple. However, there are lots of nuanced things that have to happen for it to work properly. For a truly turn-key drama-free implementation of Consul in Kubernetes, it’s recommended to use the EKS option, since very little can go wrong.

> Doctor Consul’s k3d environment was developed and tuned in WSL2 on Windows. While it *should* work on Macs, I have not personally run it on a Mac. Be advised, things may go wonky and we’ll need to adjust better for Macs.

All of the heavy lifting to build the Kubernetes environments is performed by the `./kube-config.sh` script. Without any additional arguments, the script launches Kubernetes using K3d. A full breakdown of features in the `kube-config.sh` is in Appendix A.

The `kube-config.sh` on its own has no reliance on the VM-style environment, meaning you can simply run `./kube-config.sh` and build a working 4 cluster configuration of Consul in k3d locally.

As much as possible, Doctor Consul is “modularized” and launches various additional scripts to build the intended environment. The additional scripts are located in `./scripts/`.

K3d local details You should probably know

K3d Image Registry

To prevent getting denied by Docker Hub due to repeatedly pulling docker images all day, the container images are placed locally within a K3d image registry. The K3d registry requires a hostname to be used. The hostname `k3d-doctorconsul.localhost` is put into the local `hosts` file, if it doesn’t already exist.

The K3d registry listens on `doctorconsul.localhost:12345`.

Images must be explicitly downloaded and cached. The images that are cached are:

```
IMAGE_CALICO_CNI="calico/cni:v3.15.0"
IMAGE_CALICO_FLEXVOL="calico/pod2daemon-flexvol:v3.15.0"
IMAGE_CALICO_NODE="calico/node:v3.15.0"
IMAGE_CALICO_CONTROLLER="calico/kube-controllers:v3.15.0"
IMAGE_FAKESERVICE="nicholasjackson/fake-service:v0.26.0"
```

To change the version cached, simply modify the variables within the `./scripts/k3d-config.sh`.

> When modifying the images that are cached, don't forget that the deployment YAML also needs to be changed to reference the images.

The Calico CNI is installed in the K3d environment. There were issues between the default K3d CNI and the Consul CNI plugin, that were not present in Calico. And therefore, the switch was made.

K3d Clusters

Four K3d clusters are built in parallel. Each cluster has a set of local port forwards defined to expose various services. The specific ports and application details are output when the `./kube-config.sh` script completes.

> There may be an issue on the very first build of the K3d where it cannot find the docker network. This should be resolvable by running the starting up the VM style environment first and then immediately killing it. I likely need to build some sort of pre-check on the existence of the specific doctorconsul_wan network.
>
> To start the VM style environment: `./start.sh`
>
> After the containers fire up and logs start scrolling like crazy, break the shell (control+c) and run the kill script: `./kill.sh -docker`
>
> Until there is a better way, this should build the underlying docker network that K3d relies upon.

Building Doctor Consul: Kubernetes (AWS EKS)

The easiest way to use the Doctor Consul environment is with the AWS EKS integration. It requires first installing four EKS clusters using Terraform and this specific repo:

> GitHub: https://github.com/ramramhariram/EKSonly

Building EKS clusters

The EKSOnly repo builds the clusters with the exact parameters needed to integrate with Doctor Consul.

Doctor Consul and EKSOnly default to building EKS clusters in `us-east-1`, but this can be changed to any AWS region.

To set a custom AWS region:

- Modify the `EKSOnly/terraform.tfvars` file `region` variable.
- Modify the `doctorconsul/kube-config.sh` script `AWS_REGION` variable.

> The region in the helm chart uses the wildcard value `*.elb.amazonaws.com` for `tls.serverAdditionalDNSSANs`. This is too open for a production environment but works great in Doctor Consul. Any AWS region ***should*** work.
>
> Only `us-east-1` and `eu-west-2` have been tested and are successful.

Doctor Consul automagically remaps the EKS cluster contexts to the format used in Doctor Consul.

The EKSOnly configuration needs to be set to build four clusters. This is configured via `eks_total = 4` within the `terraform.tfvars` file.

> GitHub: https://github.com/ramramhariram/EKSonly/blob/main/terraform.tfvars#L8

Once this is set, simply run `terraform apply` to build the EKS clusters.

> Do not forget to copy your AWS credentials into environment variables on any shells that are used for `terraform` or Doctor Consul.

Provide the EKS Terraform state file location

Doctor Consul needs to be told where the Terraform state file from the EKSOnly build lives. This is done via the `EKSONLY_TF_STATE_FILE` environment variable. The default value is my personal location (`/home/mourne/git/EKSonly/terraform.tfstate`). It will NOT work unless you provide the correct location.

Deploy Doctor Consul

Once Terraform has finished and four EKS clusters are available, finish the installation of the Doctor Consul Kubernetes environment by running `./kube-config.sh -eks`. Once the plan is executed successfully, the environment is automagically constructed and a comprehensive list of access points to engage with Doctor Consul is provided. All of the services are exposed via public ELBs.

> Consolidating ELBs would be very handy, but today each `loadBalancer` service is unique. Perhaps someone with more AWS EKS knowledge can determine how to group services that are at least within the same Kubernetes namespace.

See the supplemental information in this manual for details on the available uses.

Local Kubernetes Port Forwards

Doctor Consul locally with K3d uses a specific set of static localhost port forwards to access applications and resources within Doctor Consul. It's advantageous to use these port forwards as opposed to the public ELB endpoints.

The primary advantage is that each time Doctor Consul is provisioned into EKS, the load balancer endpoint address change. This can be somewhat cumbersome to constantly be changing out browser tabs with the new addresses. Using `localhost:xxxx` address makes it so you can keep the same browser tabs open and simply refresh them when Doctor Consul is re-run.

The secondary benefit is that the Consul ACL token used to login to the UI is cached and re-used when you use the same localhost addresses. So you only need to login to the Consul UI once, regardless of how many times Doctor Consul is re-run.

The `./kube-config` outputs an easy block of port forwards that can copypasta'd into the shell to establish all the necessary port forwarding.

If the outputs need to be regenerated at any point in time, run:
`./kube-config -eks -outputs`.

Building Doctor Consul: Kubernetes (GCP GKE)

Deploying Doctor Consul to GKE is currently under construction. Consul does not support GKE autopilot clusters until ~Sept 2023. Until then, this project is on hold.

When this becomes available, it will be provisioned via the `./kube-config.sh -gke` command.

Killing & rebuilding Doctor Consul

To nuke Doctor Consul resources, the `./kill.sh` script is provided. There are semantical differences in what it does, depending on the arguments provided.

VM-style Docker Compose environment

To stop the docker-compose environment, you merely have to control+c the main running window. This stops all running containers. All of the containers are still configured and can be seen via `docker ps`. To completely clean out all of the containers and return to “pre-doctor consul” conditions, run: `./kill.sh -docker`.

K3d local Kubernetes environment

To destroy the local K3d clusters, but still keep the K3d Registry with cached images, run `./kill.sh` with no arguments.

AWS EKS environment

Extra care needs to be taken when deprovisioning the EKS environment. One would think that you could simply run a `terraform destroy` and it would cleanly remove the entire environment, but this is far from accurate. If this is done, terraform will hang and AWS resources will be orphaned. Specifically, AWS orphans EC2 loadbalancers.

Run `./kill.sh -eks` prior to `terraform destroy`.

This process takes a while to perform, but when it finishes, all of the EKS cluster will be cleansed of resources that can cause Terraform to fail. Once it completes, then run `terraform destroy`.

GCP GKE environment

When the GKE implementation of Doctor Consul is completed, `./kill.sh -gke` will be used to destroy the GKE clusters.

Interacting with the VM-Style Environment

<< Low priority content. This is coming last >>

Common Architecture

<< Low priority content. This is coming last >>

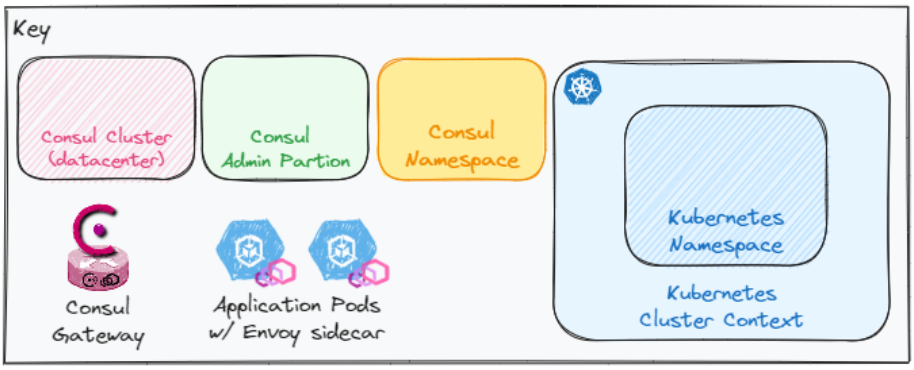
<Create Architecture Diagrams>

Interacting with the Consul Kubernetes Environment

Common Architecture

The Kubernetes environment changes quite frequently. Instead of presenting a single monolithic visual diagram that looks like spaghetti and needs to be updated regularly, specific diagrams will be presented per individual application / use-case.

The core architecture as of September 2023 is:



Kubernetes API

Interacting with the Kubernetes clusters is performed in the traditional way. The Kubernetes APIs are exposed and the contexts are written into the kube config automatically. When the `./kube-config.sh` script completes, the output provides environment variables that can be used in conjunction with the `kubectl --context` command.

The Context variables are:

Kube Context Variable	Consul Cluster / Partition Name
$KDC	**DC3** (**Default**)
$KDC_P1	**DC3** (**Cernunnos**)
$KDC4	**DC4** (**Default**)
$KDC4_P1	**DC4** (**Taranis**)

K9s

I cannot stress enough the importance of k9s for greatly simplifying the navigation of Kubernetes, and for improving the collection of troubleshooting data. I have provided a k9s plugin that makes collection of crucial troubleshooting data much easier. No longer are the days of having to keep a notes file of pages of `kubectl` commands handy. See more about the plugin in Appendix B.

For more detailed information on troubleshooting the Consul Dataplane and Envoy, see the Consul Dataplane Troubleshooting Guide:

<< It doesn't exist yet :D. It's half-written, but will be linked here when it's ready. >>

Kubernetes Configure files

All of the Kubernetes configuration YAML is located in sub directories of `./kube/`. The Helm installation uses different parameters depending on which Kubernetes environment is used (k3d, EKS, so forth).

Helm is installed via the `./scripts/helm-install.sh` and uses a combination of helm chart values file + command line arguments to install `consul-k8s`.

Applications and other Consul resources are installed via their respective scripts. More details about each component can be found within its own documentation further in this manual as well as directly in comments within each of the configs.

Consul Cluster Peering

Doctor Consul's default configuration includes cluster peering over Mesh Gateways using the following connections:

Dialer Cluster / Partition	Acceptor Cluster / Partition
`DC4/default`	`DC3/default`
`DC4/taranis`	`DC3/default`

The peering connections are established via the `kube-config.sh`

> The peering connections are established using the Consul API directly, instead of using Kubernetes CRDs. This choice was made purely because it's a lot easier to perform peering via the API / UI than it is via CRDs.

Various Applications and Use-cases

To learn more about each application and its use cases, continue on.

VM-based Applications & Use Cases

<< Low priority content. This is coming last >>

- Web
    - Web-chunky
    - Web-upstream
- Donkey
- Baphomet
- Virtual-baphomet
- Unicorn
    - Includes failover

<< Each Needs its own sections of details >>

Kubernetes Applications & Use Cases

The various applications and their use cases are detailed within this section of the manual.

The different applications and use cases are mostly launched via their own sub-scripts. This makes it convenient to see exactly which configuration files are responsible for setting up different scenarios.

Some portions of the configurations are defined in the main `kube-config.sh`. While this may seem confusing to split portions of the application between two different scripts, there is a method to the madness. Some configuration files are scoped singularly at the admin partition level, even though the configurations may affect multiple applications managed by completely separate unrelated application dev teams. Examples of these types of configs are `mesh`, `exportedServices`, and `samenessGroup`.

Instead of burying these configuration files within the script of the first application that uses them, it makes more sense to call these configs from the main `kube-config.sh` script. The application sub-scripts are used for configs unique to the application itself.

> If a second application accidentally applies its own version of one of these partition-scoped configs, it will stomp the existing config, breaking the previously configured application.
>
> This behavior must be avoided! In the real world, great care is needed to provide some form of configuration management to prevent these types of scenarios.

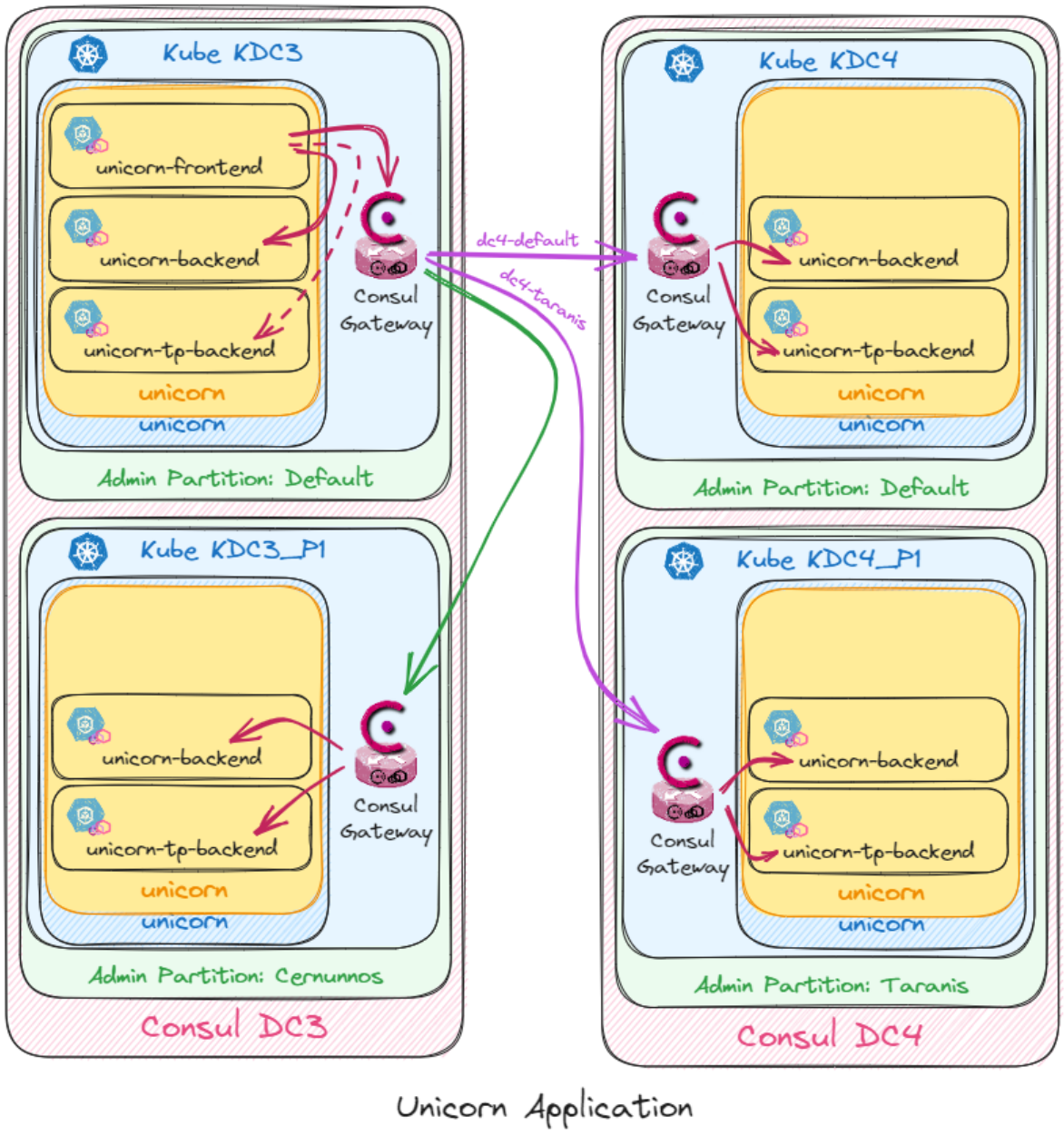
k9s:

K9s is used heavily within these application and use case guides. In addition to just the core k9s, I have provided a k9s plugin to greatly improve the experience around using these applications. Make sure this is installed or the instructions won't work as expected.

Unicorn Application

Description:
The Unicorn application is intended to show a wide variety of techniques to connect a single downstream service (`unicorn-frontend/`**`unicorn`**`/`**`default`**`/`**`dc3`**) to 8 different upstream services (`unicorn-backend/`**`unicorn`**) each in a different location.

Architecture:



Unicorn Application

Build scripts:

```
./scripts/app-unicorn.sh
```

Access: http://127.0.0.1:11000/ui/

Details:
While this application may look complicated on the surface having eight different upstream services, we can logically break them into smaller patterns. There are four different locations the upstreams live in:

- `DC3 Default` (local)
- `DC3 Cernunnos` (cross admin partition)
- `DC4 Default` (cross cluster peer)
- `DC4 Taranis` (cross cluster peer - non-default partition)

There are two upstream services within each of these 4 locations. One is accessed via explicit proxy (localhost listener ports on unicorn-frontend) and one is accessed via transparent proxy. They are named accordingly:

- `unicorn-backend` (explicit proxy)
- `unicorn-tp-backend` (transparent proxy)

While transparent proxy is the absolutely recommended pattern for feature robustness and positive UX, explicit proxy is still supported, which is why this application showcases both. If a breaking change is introduced into a new Consul version, this application works as a single place to see which combination of proxy methods and destination locations is functioning correctly.

Additionally, `unicorn-tp-backend/unicorn/default/dc3` is configured within a `SamenessGroup` and automatically fails over in the same order as listed above. Tinkering with failover and Sameness Groups is better done within the unicorn-ssg-failover application, since it also uses a `SamenessGroup` but only has a single upstream service, instead of 8.

What to do with it:
This provides a lot of different configurations within the same downstream proxy. It's a good example to use to familiarize yourself with Envoy clusters and the Envoy config dump.

It also provides a good place to run new Consul versions and quickly verify that no changes broke basic connectivity between mesh services.

Noteworthy:
The Consul API Gateway serves the unicorn-frontend service. This is covered in more detail within the API Gateway use case in this manual.

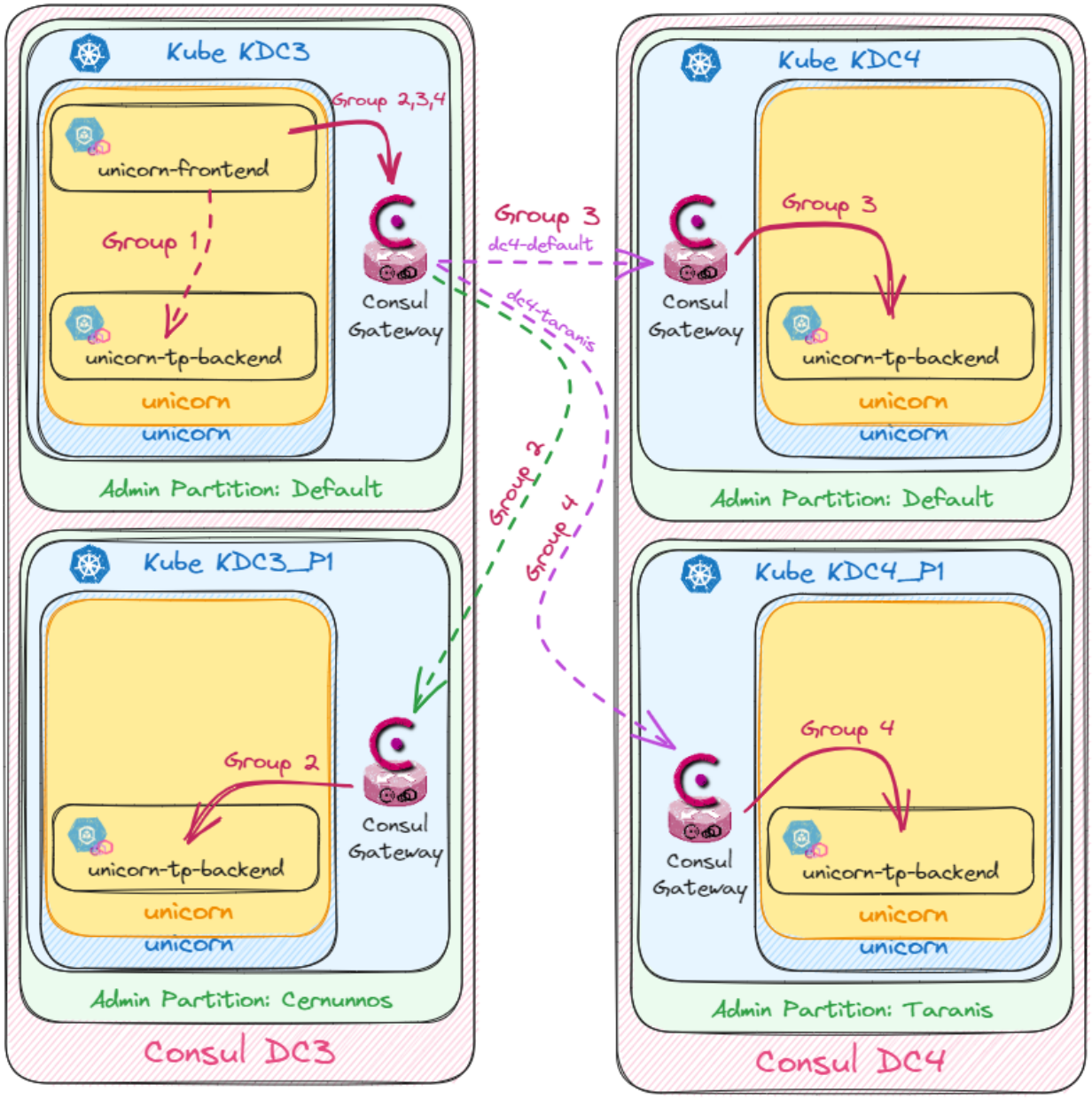
Because transparent proxy is used, the Fake Service UI port must be ignored via pod annotation otherwise the UI will be inaccessible:

```
consul.hashicorp.com/transparent-proxy-exclude-inbound-ports: "10000"
```

Unicorn SSG Application

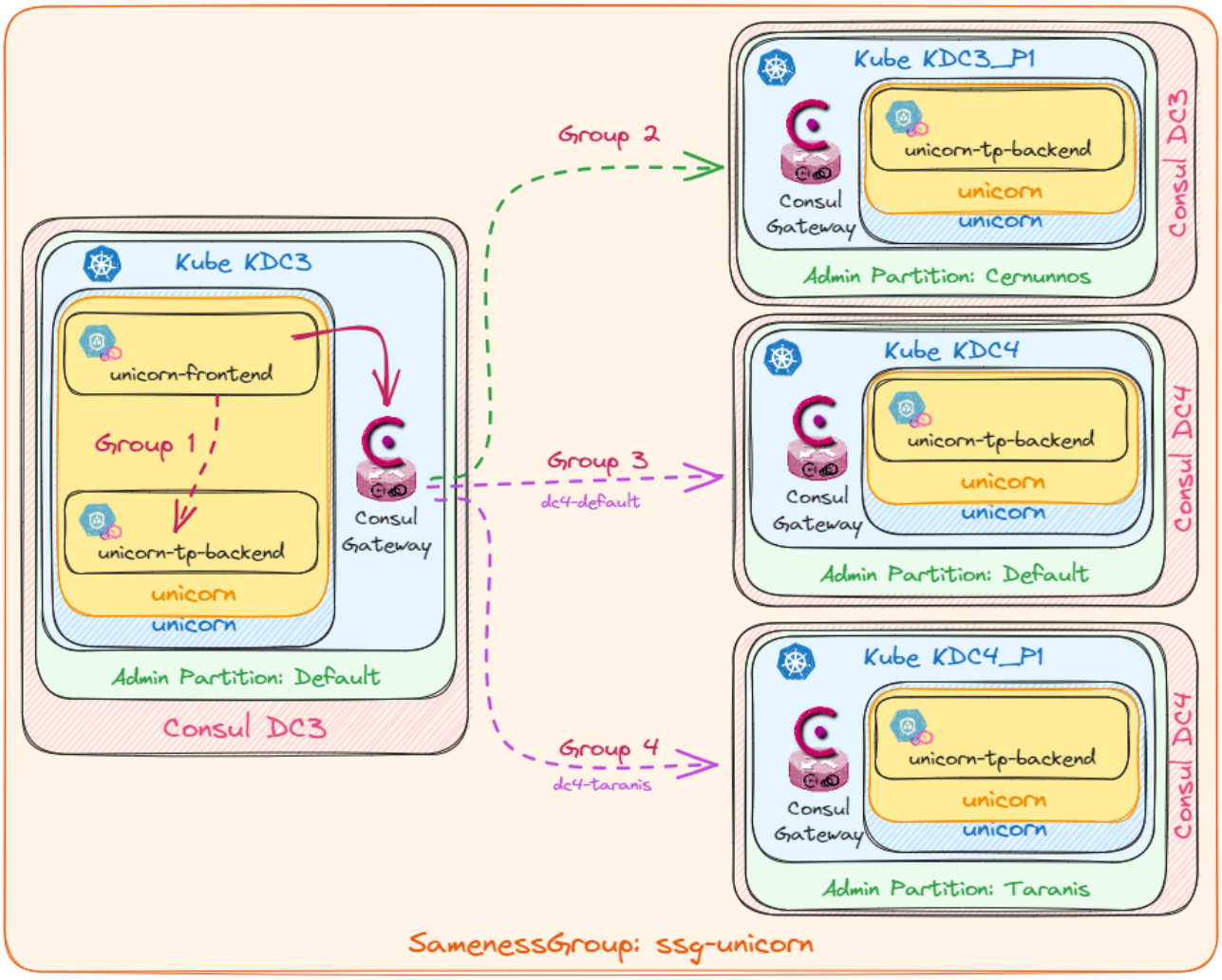
Description:
The Unicorn SSG application is very similar to the Unicorn application, except it showcases using a single downstream service (`unicorn-ssg-frontend/`**`unicorn`**`/`**`default`**`/`**`dc3`**) connecting to a single upstream service (`unicorn-backend/`**`unicorn`**) that lives in four different locations. The upstream service selection is performed via a SamenessGroup. When there are no healthy instances within one location, the next location is selected, until there are no available healthy upstream services.

Architecture:



Unicorn SSG Application

A better way to look at the architecture may be from the logical priority point of view. The SamenessGroup defines the order of upstreams as such:



Unicorn SSG Application (logical)

Build scripts:

```
./scripts/app-unicorn.sh
./kube-config.sh
```

Access: http://127.0.0.1:11001/ui/

Details:

The unicorn-ssg-frontend service connects to a single upstream `unicorn-tp-backend`, which lives in the same **`unicorn`** namespace and **`default`** partition. The SamenessGroup defines multiple sub-groups that are accessed in order (fail over) whenever there are no available healthy services within a member group.

The failover order in this application is:

`unicorn-backend/unicorn/default/dc3` (local)
`unicorn-backend/unicorn/cernunnos/dc3` (cross-partition)
`unicorn-backend/unicorn/DC4-Default/dc3` (peer)
`unicorn-backend/unicorn/DC4-Taranis/dc3` (peer)

What to do with it:
The Unicorn SSG application is great for observing the Envoy clusters and Envoy Config in relation to SamenessGroups. It's also great to observe SamenessGroup failover in real time. To see it in action follow the steps below.

- Go to the Application UI.
  - Notice the upstream being accessed is: "unicorn-backend {Transparent} (DC3)"
  - Also take note of the Request URI. See how it points to the local `unicorn-tp-backend`.
- Kill the local upstream service.
  - In k9s:
  - Set the context (:ctx) to `k3d-dc3`
  - In pods (:pod), highlight the `unicorn-tp-backend` pod and scale it to zero with `shift+0`
- Refresh the Application UI
  - Notice that the upstream is now: "unicorn-backend {Transparent} (DC3 Cernunnos)"
  - The Sameness Group has forced the upstream to change, since there are no healthy local Default instances of `unicorn-tp-backend`.
  - Also take note that the Request URI ***hasn't*** changed. The downstream application believes it's still talking to the same upstream service.
- Kill the Cernunnos upstream service:
  - Set the context to `k3d-dc3-p1`
  - In pods, highlight the `unicorn-tp-backend` pod and scale it to zero with `shift+0`
- Refresh the Application UI
  - Notice that the upstream is now: "unicorn-backend {Transparent} (DC4)"
  - The Sameness Group has forced the upstream to change again, since there are no healthy Cernunnos instances of `unicorn-tp-backend`.
  - It's now being accessed across the cluster peer dc4-default.
  - The Request URI still ***hasn't*** changed. And it won't ever (spoiler alert).
- Kill the dc4-default upstream service.
  - Set the context to `k3d-dc4`
  - In pods, highlight the `unicorn-tp-backend` pod and scale it to zero with `shift+0`
- Refresh the Application UI
  - Notice that the upstream is now: "unicorn-backend {Transparent} (DC4 Taranis)"

    - It's now being accessed across the cluster peer dc4-taranis.
- Kill the dc4-taranis upstream service.
    - Set the context to `k3d-dc4-p1`
    - In pods, highlight the `unicorn-tp-backend` pod and scale it to zero with `shift+0`
- Refresh the Application UI
    - Uh oh. The application is critically failing now.
    - If you wish, you can scale one or all of the pod deployments back to 1 and refresh the service.
        - In deployments (:deployment) highlight `unicorn-tp-backend` and press `Shift+1` to scale to 1 pod.
        - Refresh the application UI to see it connect to the healthy upstream service.

Noteworthy:
Pull the Envoy clusters on `unicorn-ssg-frontend`. Notice how the SSG failover targets are organized and named:

```
failover-target~0~unicorn-tp-backend.unicorn.dc3.internal.<spiffeID>.consul
failover-target~1~unicorn-tp-backend.unicorn.dc3.internal.<spiffeID>.consul
failover-target~2~unicorn-tp-backend.unicorn.dc3.internal.<spiffeID>.consul
failover-target~3~unicorn-tp-backend.unicorn.dc3.internal.<spiffeID>.consul
```

You can see that there are 4 separate clusters for when attempting to access `unicorn-tp-backend.unicorn.dc3`. You can't really see where the true destination is for these though, unless you dig deeper into each cluster within the Config Dump.

For example. Here we correlate the third failover target to the service instance that lives in the Taranis partition (cluster peer).

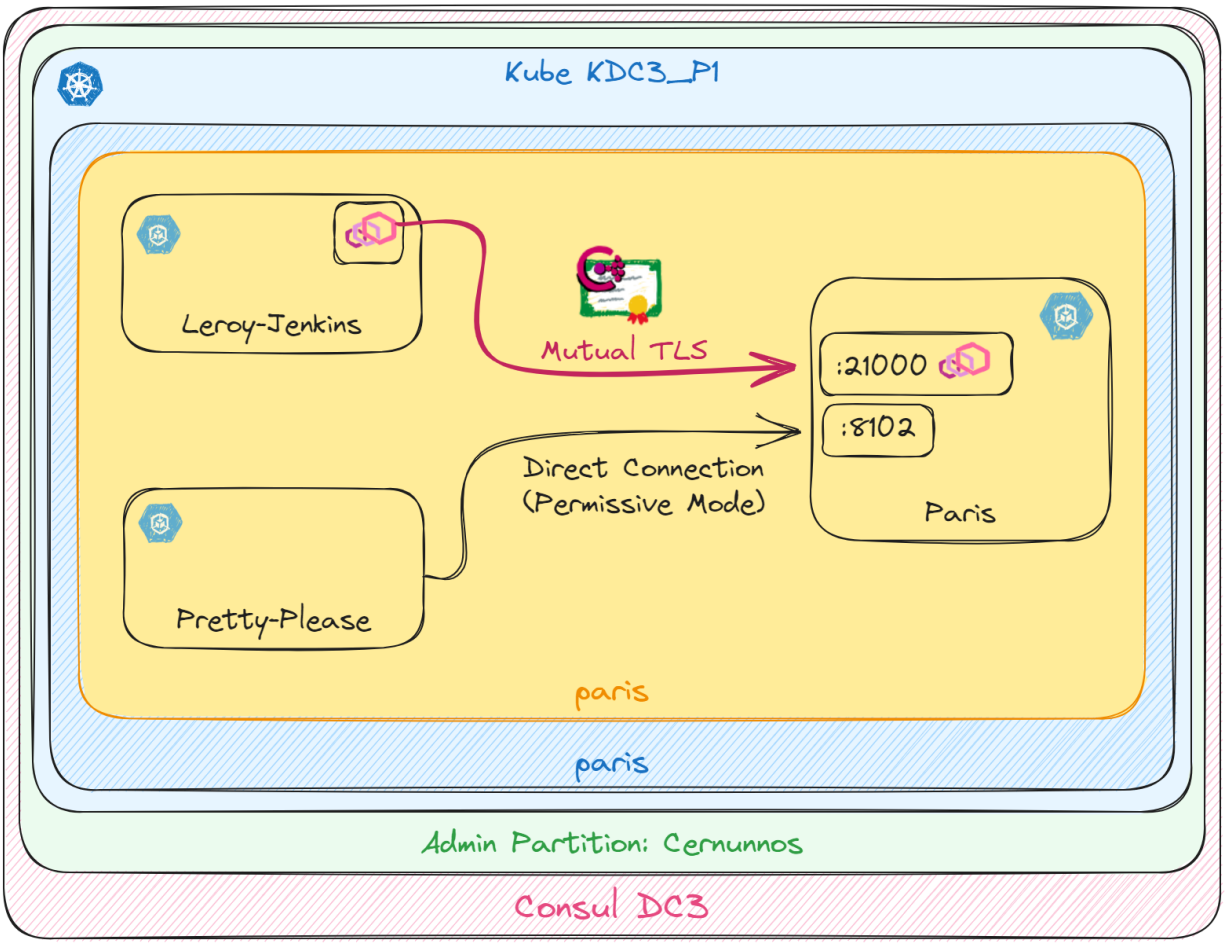
```
Cluster.name: "name":
"failover-target~3~unicorn-tp-backend.unicorn.dc3.internal.29ec06be-ba0c-ea79-763f-d90f81aa70d1.consul"
…
"sni":
"unicorn-tp-backend.unicorn.taranis.dc3-default.external.00f6b946-713c-9ca4-d964-cc0b6ac793c9.consul"
```

Paris Application (Permissive Mode)

Description:
The Paris application makes use of Consul Permissive TLS Mode.
`paris/paris/cernunnos/dc3` is a permissive mode-enabled upstream service. It is accessed by two downstream services.

> `leroy-jenkins/paris/cernunnos/dc3` connects via the Consul service mesh.
> `pretty-please/paris/cernunnos/dc3` connects directly over plaintext.

Architecture:



Paris (Permissive) Application

Build scripts:

```
./scripts/app-paris.sh
```

Access:

Leroy-Jenkins: http://127.0.0.1:8100/ui/
Pretty-Please: http://127.0.0.1:8101/ui/

Details:

There isn't much to really dig deeper on with `Leroy-Jenkins` > `Paris`, as this is a standard Consul service mesh connection. What makes the Paris application special is that it also accepts non-Consul mesh connections and forwards them into the Paris application listener on TCP/8102.

To enable Permissive TLS mode in Consul, there are two requirements:

- Permissive mode is globally allow within a `mesh` configuration:
  - `spec.allowEnablingPermissiveMutualTLS: true`
- Permissive mode is enabled via `service-defaults` for the specific service:
  - `spec.mutualTLSMode: true`

What to do with it:

Nothing particularly special beyond a working example to copypasta.

Noteworthy:

Permissive mode creates a new listener (`tcp.permissive_public_listener`) in Envoy. Stats can be fetched from the Envoy /stats endpoint.

```
tcp.permissive_public_listener.downstream_cx_no_route: 0
tcp.permissive_public_listener.downstream_cx_rx_bytes_buffered: 0
tcp.permissive_public_listener.downstream_cx_rx_bytes_total: 16688
tcp.permissive_public_listener.downstream_cx_total: 44
tcp.permissive_public_listener.downstream_cx_tx_bytes_buffered: 0
tcp.permissive_public_listener.downstream_cx_tx_bytes_total: 15069
tcp.permissive_public_listener.downstream_flow_control_paused_reading_total: 0
tcp.permissive_public_listener.downstream_flow_control_resumed_reading_total: 0
tcp.permissive_public_listener.idle_timeout: 0
tcp.permissive_public_listener.max_downstream_connection_duration: 0
tcp.permissive_public_listener.upstream_flush_active: 0
tcp.permissive_public_listener.upstream_flush_total: 0
```

What's the difference between using Permissive Mode in Consul and simply creating a transparent proxy exclusion on the port listened on by the upstream service?

Both will allow downstream applications to connect directly to the upstream service, but with Permissive mode, you get the following benefits:

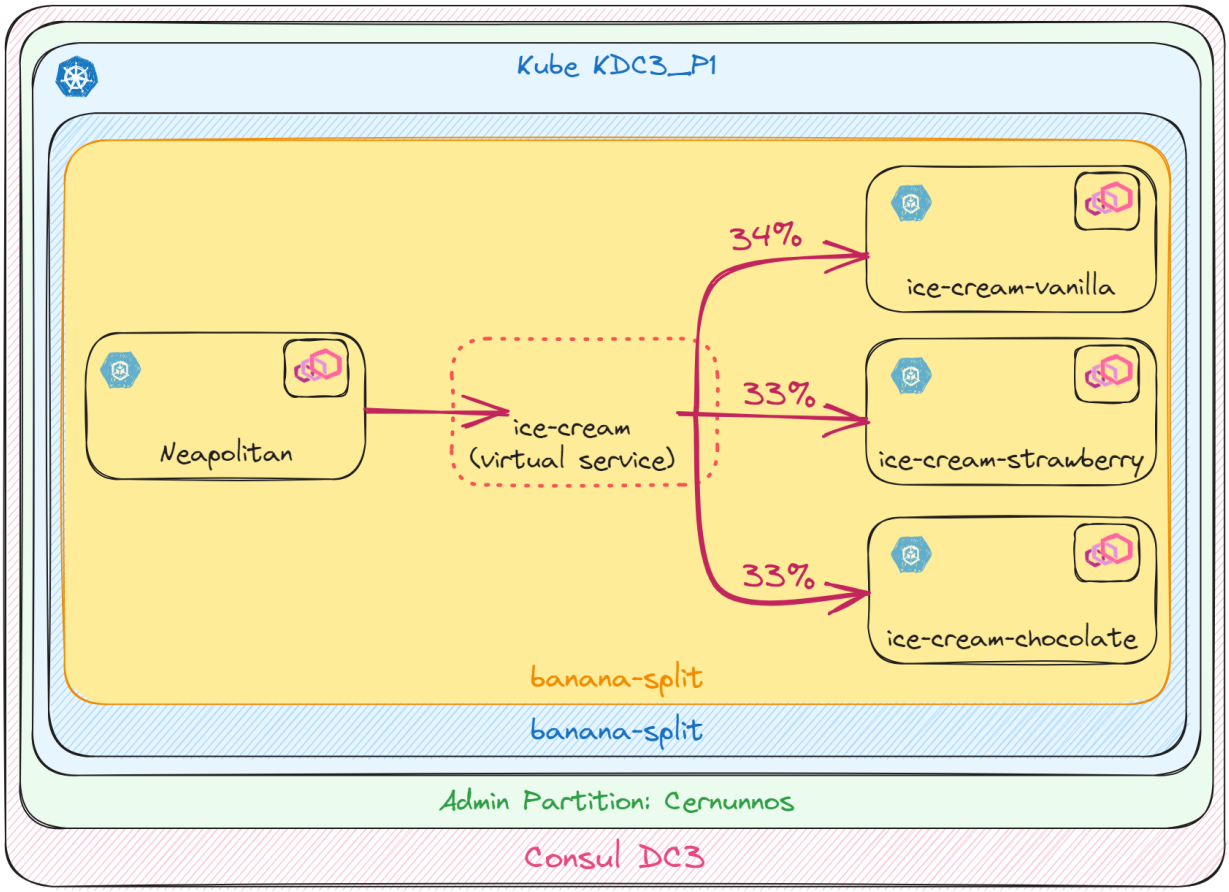
- Envoy Metrics
- Envoy extensions that apply to inbound listeners
- Envoy's circuit breakers

Banana Split (Traffic Splitting, Blue/Green, Canary)

Description:
The Banana Split application makes use of Consul weighted traffic splitting. `neapolitan/banana-split/cernunnos/dc3` is a downstream service that connects the virtual service `ice-cream`. The `ice-cream` service splits the traffic evenly across three upstream services:

```
ice-cream-vanilla/banana-split/cernunnos/dc3
ice-cream-strawberry/banana-split/cernunnos/dc3
ice-cream-chocolate/banana-split/cernunnos/dc3
```

Architecture:



Banana-Split (Route Splitting) Application

Build scripts:

```
./scripts/app-banana-split.sh
```

Access:

Neapolitan: `http://127.0.0.1:8007/ui/`

Details:

This application primarily illustrates one way that blue/green and canary deployment models can be achieved. In Consul, there are primarily two ways that traffic splitting can be achieved:

`route-splitter` splits traffic to different Consul services.
`route-splitter` split traffic between service "Subsets".

This application makes use of weighted splitting to ***different*** upstream Consul services. This is the most straightforward way of setting up splitting. Alternatively, Consul can create Service Subsets using things like meta tags of instances that belong to the ***same*** Consul service.

It appears to be fairly non-trivial to configure Kubernetes pod deployments to differentiate pods within the same Kube deployment name, but with different Consul metadata. For this reason, banana-split utilizes splitting via different upstream services, not the same service with subsets.

It appears that the other option may be possible, but difficult to implement. Need to investigate https://developer.hashicorp.com/consul/tutorials/developer-mesh/service-splitters-canary-deployment further, since this may have an example of the Service Subset method.

The default weights used to route the upstream traffic are:

34% ice-cream-vanilla
33% ice-cream-strawberry
33% ice-cream-chocolate

The weights are configured within the `service-splitter` configuration (./kube/configs/dc3/service-splitter/service-splitter-ice_cream.yaml).

What to do with it:

Go to the Neapolitin UI and repeatedly hit the refresh button. The selected upstream service switches with each request.

Try changing the weights within the `service-splitter` and re-applying the configuration file. This change will immediately change the split percentages.

The following example removes Vanilla and sets the majority of the traffic to go to Chocolate:

```
spec:
  splits:
    - weight: 0
      service: ice-cream-vanilla
    - weight: 33
      service: ice-cream-strawberry
    - weight: 67
      service: ice-cream-chocolate
```

Apply the configuration change:

```
kubectl apply --context $KDC3_P1 -f
./kube/configs/dc3/service-splitter/service-splitter-ice_cream.
yaml
```

This is how blue / green or canary deployments can be achieved. Once the deployment is successful, the `service-splitter` can be updated to remove the old version of the application.

Noteworthy:
Since the `ice-cream` service is a virtual service, there are no running pods or infrastructure for it in Kubernetes. It purely exists logically within the Consul registry as a way to implement traffic splitting.

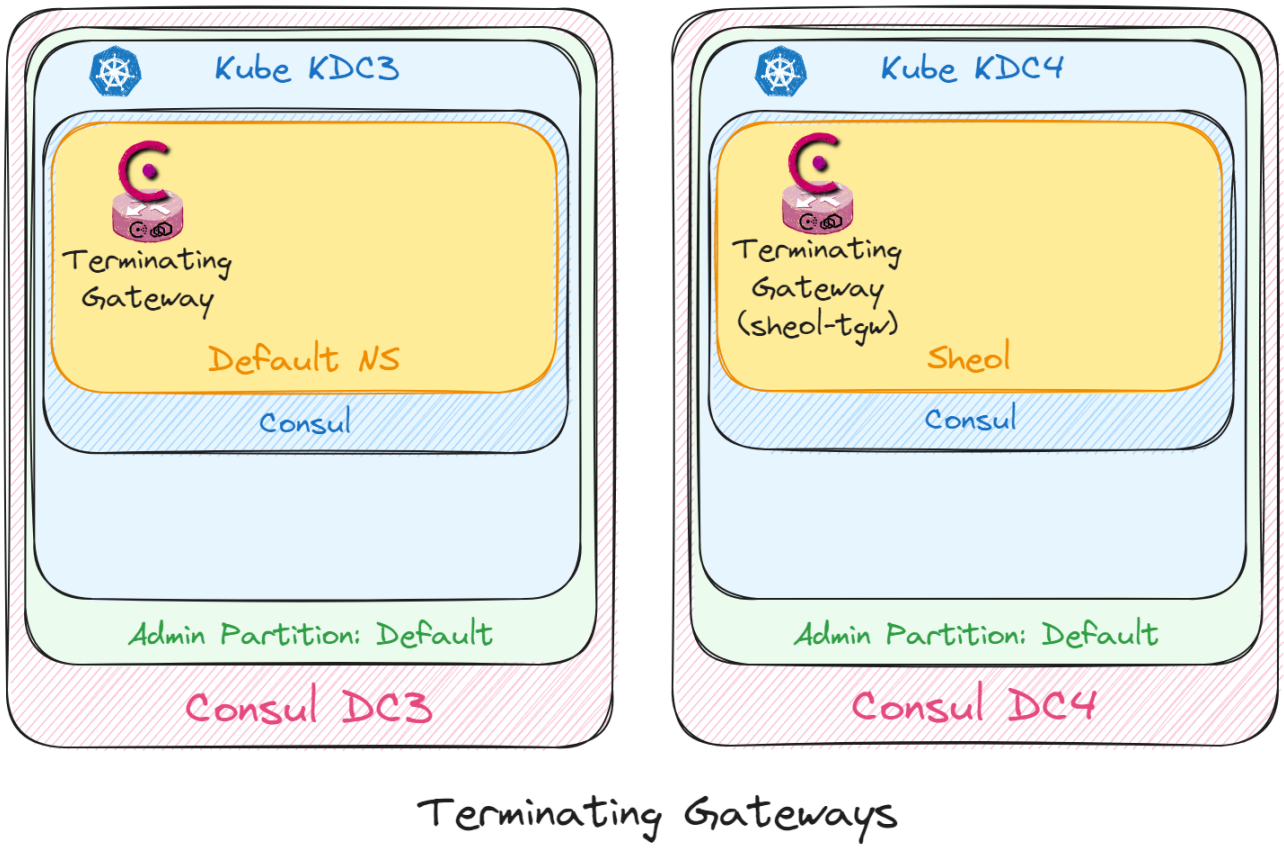
If the `service-splitter` used “Service Subsets” instead of “services”, a `service-resolver` must also be defined to assign service instances into subsets using various criteria, such as meta version tags.

Terminating Gateway

Description:
The Doctor Consul Kubernetes environment builds two Terminating Gateways, as specified in the helm chart values files. The Terminating Gateway setup script provides additional setup required to make use of the TGW in the Sheol and Externalz applications. The Terminating Gateways are:

```
terminating-gateway/default/default/dc3
sheol-tgw/sheol/default/dc3
```

Architecture:



Terminating Gateways

Build scripts:

```
./scripts/terminating-gateway.sh
```

Access:

No UI access.

Accessing a shell requires attaching a Debug container (`Shift+D`)

Details:
By itself, this setup script doesn't do anything flashy. There are two key components, that are VERY important to understand. The applications that use the Terminating Gateways are defined in their own scripts.

***ACL Permissions Wonkery*:**
There is a particularly challenging issue with running Terminating Gateways in Kubernetes. TGWs require `service:write` permission on each service that it fronts. There are no ACL Policy CRDs, so updating the permissions must be done via the Consul API directly.

An extra challenge is that in Kubernetes, the TGWs obtain their ACL tokens via the Kubernetes auth method. The easiest way to add the appropriate permissions is to create a new policy file and add (update) it to the existing role. The process Doctor Consul uses to do this can be found in the `terminating-gateway.sh` script. This is done for both Terminating Gateways.

***Define which services are attached to the TGWs*:**
The second purpose for this script is to define exactly which services are fronted by the Terminating Gateways.

What to do with it:
See a working example of how to attach additional ACL policies to a Kubernetes-based Terminating Gateway. The official Consul docs are straight-up wrong.

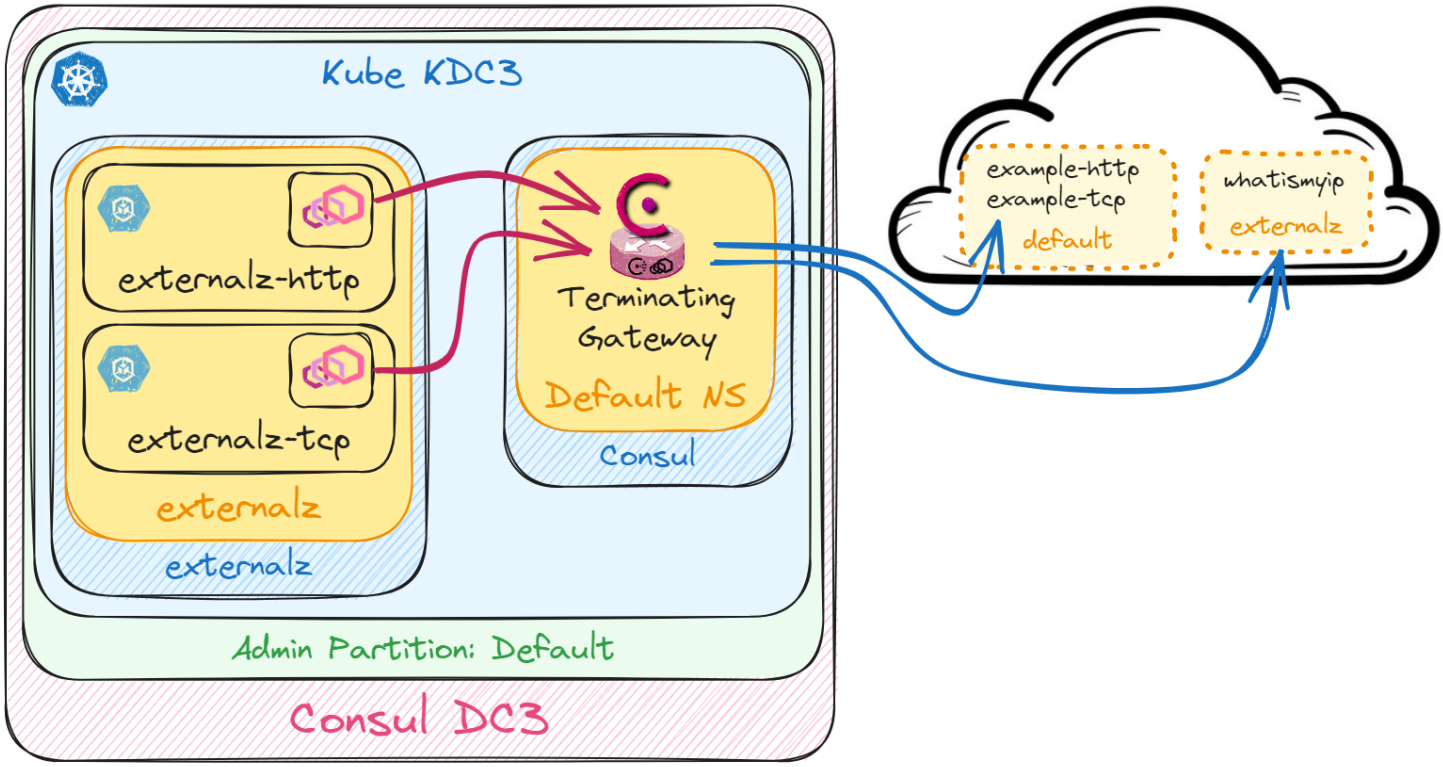
Noteworthy:
At first glance, it might seem like it'd make more sense to put the terminating gateway configs into the same scripts as the applications that actually use the TGW (`externalz` and `sheol`). The problem with that approach is that there can only be ***ONE*** `TerminatingGateway` config shared among ***ALL*** services that use the TGW. It's better to keep the configs separate, so you can find it again later. This illustrates a real-world challenge of using Consul and deciding who owns which portions of the Consul configurations.

Externalz Application (TGW + External Services)

Description:
The externalz applications use a Terminating Gateway to access services that live outside of the mesh. There are two downstream services, one defined as a Consul TCP service and the other a Consul HTTP service.

```
externalz-http/externalz/default/dc3
externalz-tcp/externalz/default/dc3
```

Architecture:



Externalz Application

Build scripts:

```
./scripts/app-externalz.sh
./scripts/terminating-gateway.sh
```

Access:

externalz-http: http://127.0.0.1:8003/ui/
externalz-tcp: http://127.0.0.1:8002/ui/

Details:
The Externalz applications dig into the nuance of how external services are defined and how the protocol configuration within the `service-defaults` drastically affects how Envoy matches traffic.

General Flow:

- Pre-Req: Terminating Gateways created (via the `terminating-gateway.sh` script)
- External services are defined via service-defaults
    - `./kube/configs/dc3/external-services/service-defaults-example.com_tcp.yaml`
    - `./kube/configs/dc3/external-services/service-defaults-example.com_http.yaml`
    - `./kube/configs/dc3/external-services/service-defaults-whatismyip.yaml`
- Service Intentions allow traffic from the `externalz` apps to the external services.
- `Externalz-tcp` and `Externalz-http` applications connect out to various external services.
    - Consul Transparent Proxy intercepts connections to these pre-defined external services and sends the traffic to the Terminating Gateway using mTLS.
- The Terminating Gateway accepts the Consul Mesh connection from the `externalz` applications and establishes a direct connection to each external service. This enables the operators to deny all other outbound access from the individual services forcing the Terminating Gateway to be used for non-mesh outbound connections.

It's important to really understand how each external service is configured.

> Ignore the "whatismyip" external service definition. It's not currently used, but may be used in a future version of the externalz application. The primary reason it originally existed was to test out an external service that belongs to a non-default namespaced `service-defaults` config. It worked and I never incorporated it into an official workflow.

Notice that the only differences between `example-http` and `example-tcp` are the names, protocols (`http` vs `tcp`), and port numbers (`80` vs `443`). The addresses are exactly the same:

```
addresses:
  - "example.com"
  - "www.wolfmansound.com"
```

These differences are extremely important to understand. It's the difference between the applications working or not.

> Unlike the other applications in Doctor Consul which default to a working state, the Externalz apps intentionally have certain upstreams that fail. This is to provide both working and non-working examples and understand why each either works or is broken.

What to do with it:

There are two primary uses for the externalz applications:

1. Provides a ready playground for testing your own external service destinations:
    a. Services for HTTP and TCP already exist. All you need to do is modify the external service definitions and push the YAML change.
    b. Shell into the appropriate externalz downstream pod and issue `openssl s_client` or `curl` commands to test accordingly.
2. Provides a configuration to illustrate common misconceptions with how to configure external services (more below).

Demo: Understanding how to correctly configure external services

Externalz-tcp:
Look at the `externalz-tcp` application UI. The default state has four external upstream connections:

- `http://example.com` **(broken)**
- `https://example.com` **(working)**
- `http://www.wolfmansound.com` **(broken)**
- `https://www.wolfmansound.com` **(working)**

Why do half work and the other fail? The `externalz-tcp` application is provisioned to use the `example-tcp` external services. The `service-defaults` definition is:

```
protocol: tcp
  destination:
    addresses:
      - "example.com"
      - "www.wolfmansound.com"
    port: 443
```

In TCP mode, the FQDNs in the addresses field only match against TLS (SNI match). HTTP Host headers do NOT match.

This is why both of the HTTPS upstreams work (matched SNI header) and both HTTP upstreams fail (Envoy in TCP mode is not protocol aware and therefore HTTP headers are not analyzed).

The port number defined is 443, the default HTTPS port. It's important to understand that even if port 80 was added to the external service config, it still wouldn't succeed because of the previously mentioned reasons.

Externalz-http:
Look at the `externalz-http` application UI. The default state has four external upstream connections:

- `http://example.com` **(working)**
- `https://example.com` **(broken)**
- `http://www.wolfmansound.com` **(broken)**
- `https://www.wolfmansound.com` **(broken)**

The `example-http` external service definition is:

```
protocol: http
  destination:
    addresses:
      - "example.com"
      - "www.wolfmansound.com"
    port: 80
```

It makes perfect sense that the two HTTPS upstreams are failing to connect, because they use HTTPS when the external service definition is set to HTTP and port 80.

But why is `http://www.wolfmansound.com` failing? Shouldn't that meet all the expected criteria? It should, but the reason it still fails is nuanced. It requires a little more digging to understand.

Go to the JSON output of the Fake Service UI, by removing `ui/` from the URL. The `.upstream_calls` block contains the following:

```
"http://www.wolfmansound.com": {
      "uri": "http://www.wolfmansound.com",
      "code": -1,
      "error": "Error communicating with upstream service: Get
\"https://www.wolfmansound.com/\": read tcp
10.66.212.90:33580->\u003e208.113.169.166:443: read: connection
reset by peer"
    },
```

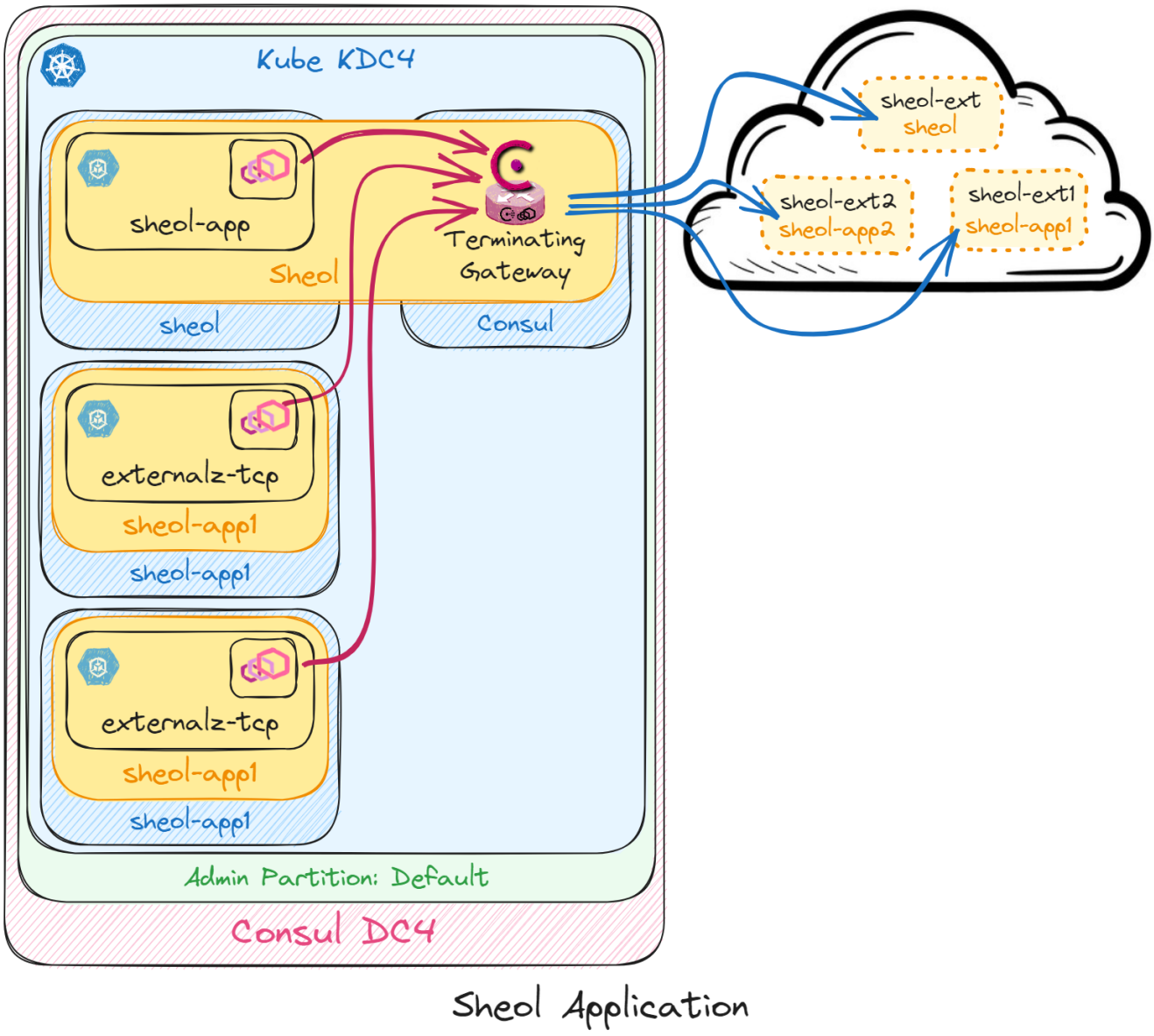
Notice that it's attempting http, but the error mentions https. That's rather odd, isn't it? The reason is that the http version of the website uses a 302 redirect to the https version of the site and we don't have the external service set to match HTTPS!

To get a website that redirects http to https working correctly, we'd need to have both `HTTP` (port 80) AND `TCP` (port 443) service-defaults configs. This is fairly tricky nuance.

Sheol Application (TGW + External Services + Multi Namespace)

Description:
The Sheol application also uses a Terminating Gateway to connect to external services. The primary architectural difference between Sheol and Externalz is that the TGW lives in its own namespace, instead of living in the `default` namespace.

Architecture:



Sheol Application

Build scripts:

```
./scripts/app-sheol.sh
```

Access:

- Sheol-app: http://127.0.0.1:8004
- Sheol-app1: http://127.0.0.1:8005
- Sheol-app2: http://127.0.0.1:8006

Details:
(Consul 1.16.1+ent) All three applications are broken. This is due to an ACL policy bug: https://hashicorp.atlassian.net/browse/NET-5355

To work around the issue, the Terminating-Gateway-Service-Write/default/default/dc4 policy needs to be updated to explicitly include the default partition (even though it should be inferred).

Fixed Config:

```
partition "default" {
namespace "sheol" {
  service "sheol-ext" {
    policy = "write"
  }
}

namespace "sheol-app1" {
  service "sheol-ext1" {
    policy = "write"
  }
}

namespace "sheol-app2" {
  service "sheol-ext2" {
    policy = "write"
  }
}
}
```

The easiest way to update the policy is to use the DC4 Consul UI. The policy can be edited directly from the web browser. If the Consul UI is removed, the policy needs to be updated via the CLI / API.

Once this policy is updated, the three applications should instantly start working.

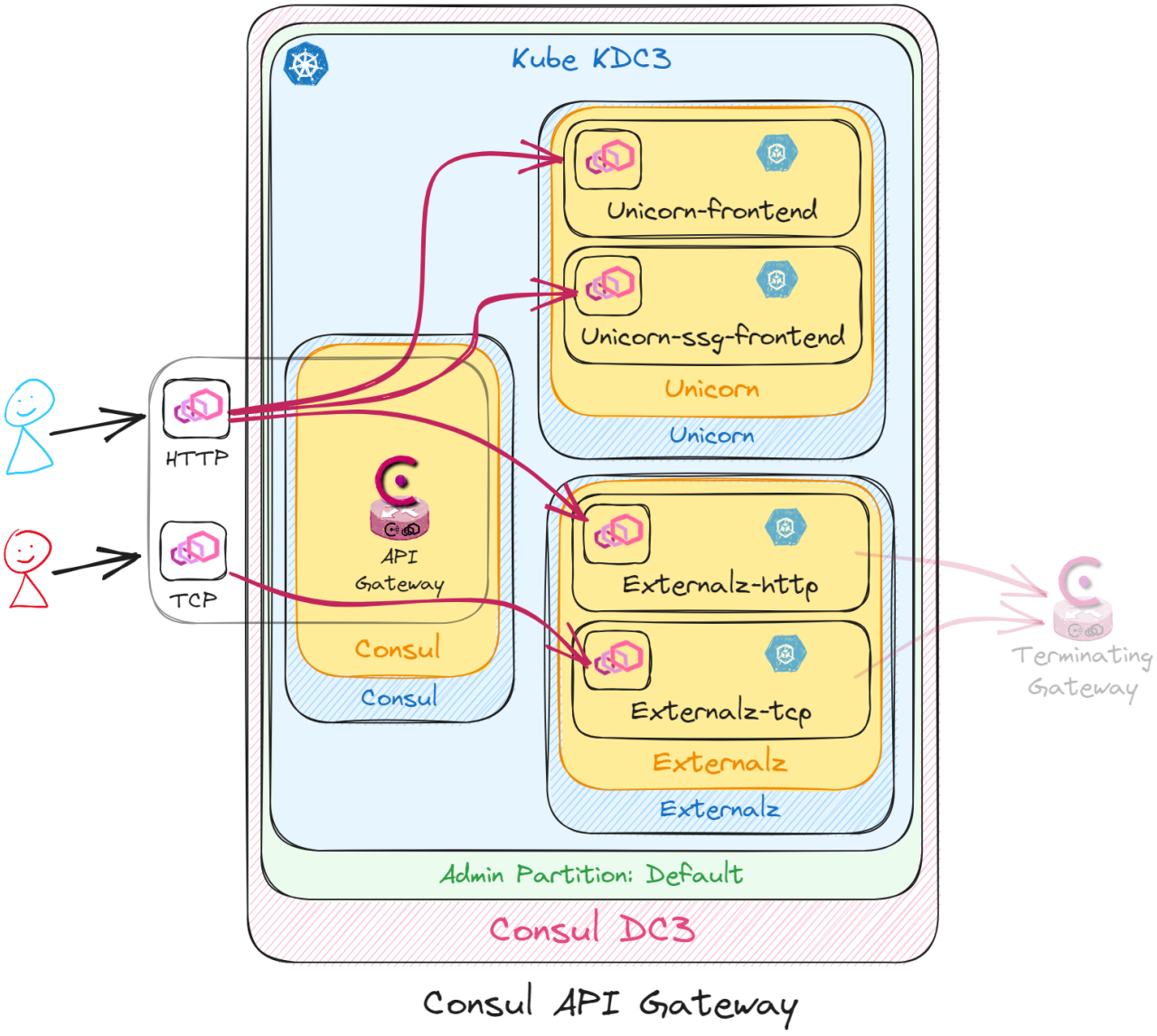
What to do with it:
There isn't a fancy demo for this application other than confirming when the ACL bug above is fixed.

Consul API Gateway Ingress

Description:
The Consul API Gateway is configured to be an ingress point for external users and applications to connect to Consul service mesh services without being directly connected to the mesh themselves.

The Consul API Gateway is configured with both TCP and HTTP public listeners and has HTTP routes using both path-based and header-based routing.

Architecture:



Consul API Gateway

Build scripts:

```
./scripts/apigw-config.sh
```

Access:

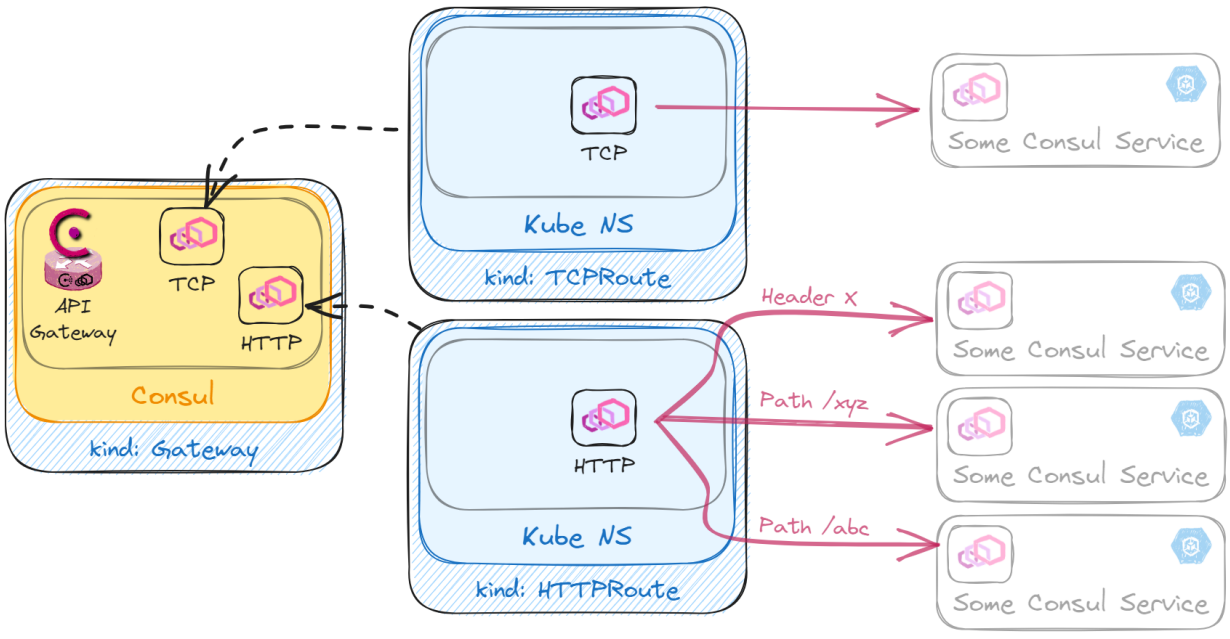
- HTTP listener: http://127.0.0.1:1666
- TCP listener: http://127.0.0.1:1667

> For some reason, the local port forwards don't work with the Consul API GW and EKS. You'll need to access them via the Elastic LB addresses presented in the kube-config output.

Details:

The Consul API Gateway uses the Kubernetes Gateway API spec to define the Gateway details and the various routes to backend services in the Consul service mesh. Users who are not already familiar with the Kubernetes Gateway API will likely struggle to understand the structure. Luckily, the most common routes are used here in Doctor Consul.

Overview of the key components for the Gateway spec + Consul



Consul + Gateway Spec

- `kind: Gateway`: Defines the Consul API and its listener interfaces
- `kind: TCPRoute`: Defines TCP routing to a backend service.
- `kind: HTTPRoute`: Defines HTTP policy-based routing matches to backend services.
- `kind: ReferenceGrant`: Gives cross namepaces permissions to routes (more below)

APIGW TCP Listener:
The Consul APIGW TCP listener maps 1:1 to the `externalz-tcp` service. Gateway listeners in TCP mode have no way of differentiating what service a client intends to reach, so it maps the listener 1:1 to a single backend service.

APIGW HTTP Listener:
The HTTP listeners support several different options to route to different backend services. The most common are path-based routing and header-based routing. The Consul API Gateway in Doctor Consul uses both. The mappings are:

`unicorn-frontend`:
- Path: `/unicorn-frontend`
- Header: `Host: unicorn-frontend`

`unicorn-ssg-frontend`:
- Path: `/unicorn-ssg-frontend`
- Header: `Host: unicorn-ssg-frontend`

`externalz-http`:
- Path: `/externalz-http`
- Header: `Host: externalz-http`

> For the Consul APIGW to have access to these applications, the `unicorn-frontend`, `unicorn-ssg-frontend`, and `externalz-http` applications must be already running.
>
> They are launched via their own application scripts, not the `apigw-config.sh` script.

What to do with it:
Play around with the various HTTP routing options configured. For the path-based routing options, simply open your browser to the address of the Consul APIGW and change the path around.

Examples:
```
http://$APIGW-ADDR/unicorn-frontend/ui/
http://$APIGW-ADDR/unicorn-ssg-frontend/ui/
http://$APIGW-ADDR/externalz-frontend/ui/
```

To use header based routing, curl can be used. Change the host headers to reach different services.

Examples:
```
curl -s http://$APIGW-ADDR:1666/ -H "Host: unicorn-frontend" | jq .name
```

```
curl -s http://$APIGW-ADDR:1666/ -H "Host: unicorn-ssg-frontend" | jq .name
curl -s http://$APIGW-ADDR:1666/ -H "Host: externalz-http" | jq .name
```

Noteworthy:
Doctor Consul doesn't currently bind TLS certificates to the API Gateway listeners. This is something that will be added in the future.

Whenever a `TCPRoute` or `HTTPRoute` refers to a backend service that is NOT in the same Kubernetes namespace as the route itself, a `ReferenceGrant` must be created to allow cross-namespace routes.

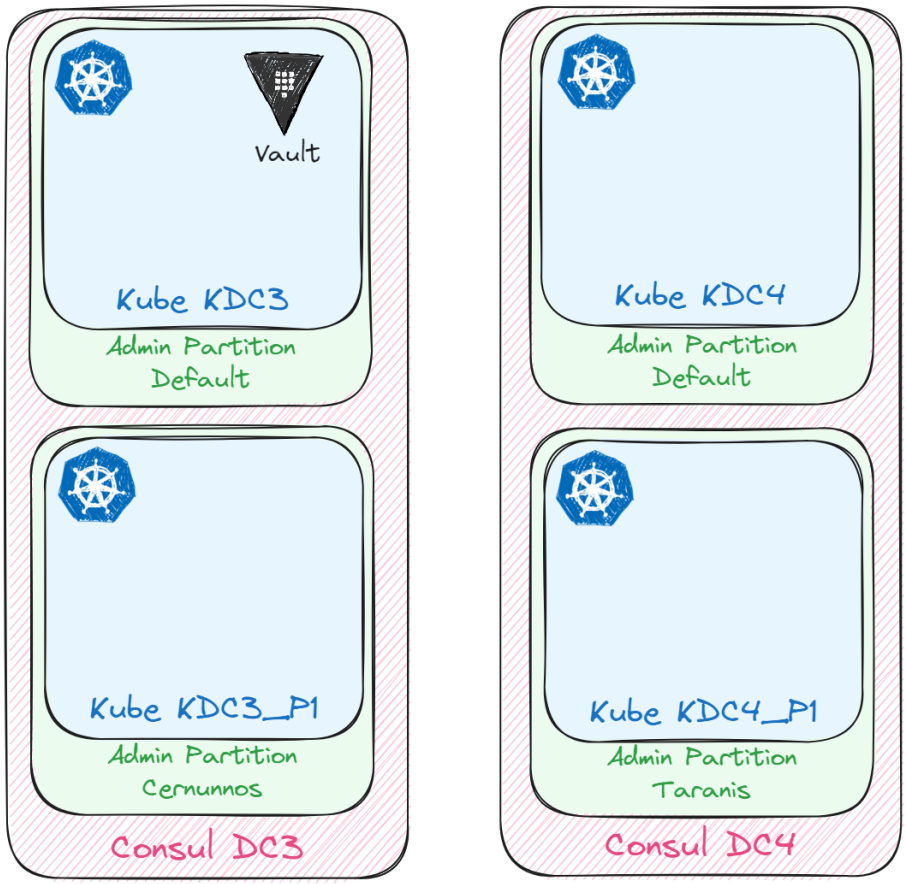
In this environment, the `HTTPRoute` mapping the `unicorn-frontend` and `unicorn-ssg-fronted` to the Gateway, exists in the same namespace "`unicorn`", so there is no issue.

Notice that the same `HTTPRoute` also links to the `externalz-http` service in the `externalz` namespace. This is not allowed by default. Within the same YAML as the `HTTPRoute` is a `ReferenceGrant` that gives the `unicorn` namespace permission to route to services in the `externalz` namespace.

HashiCorp Vault

Description:
HashiCorp Vault can be installed in any of the Kubernetes clusters. By default it is installed in the KDC3 cluster.

Architecture:



HashiCorp Vault

Build scripts:

```
./scripts/vault-config.sh $CONTEXT
```

Access: `<See the output of the script>`

Details:
Vault installs and unseals automatically.

The Vault API address, root token, and unseal keys are output to the terminal. They are also written to files:

Root token: `./tokens/vault-dcX-root.token`
Unseal Key: `./tokens/vault-dcX-unseal.key`

What to do with it:
There are no specific Vault use cases built into Doctor Consul yet. In the future, it will be used for:

- Vault as a Consul secrets backend
- Vault as a Connect CA for at least one of the clusters.
- Generation of Consul APIGW certificates.

Until then, it simply exists to be a playground as needed.

Appendix A: kube-config.sh options

The kube-config.sh script has many different options to build Doctor Consul in different environments. The `-help` argument can be used to list the options.

The important options are listed below:

`-k8s-only`
: Builds 4 local K3d clusters only without any other applications or Consul.

`-eks`
: Once 4 Kube clusters have been built using the `EKSonly` repo (https://github.com/ramramhariram/EKSonly), Doctor Consul renames the EKS contexts to fit the Doctor Consul syntax and build the Doctor Consul Kube environment in EKS.

`-eks-context`
: If you ever need to refresh the Kube Contexts from AWS EKSOnly, this will do it. IE: If you accidentally delete them.

`-nuke-eks`
: Deletes most of the Doctor Consul resources out of the EKS clusters. This is crucial to be able to perform the `terraform destroy` function, because left over ELB load balancers cause terraform to hang.

`-no-apps`
: Doctor Consul installs Consul and cluster peering, but does not install any Doctor Consul applications. This argument can be combined with others, such as:
`./kube-config.sh -no-apps -eks`

`-vars`
: All of the Doctor Consul configuration scripts rely on variables for things like cluster names and color codes. This option prints out the necessary variables so they can be copypasta'd back into the terminal before manually running the application scripts.

`-outputs`
: Reprint the outputs which include all of the available load balancer addresses for the applications. Can be combined with other arguments, such as:
`./kube-config.sh -eks -outputs`

Appendix B: Additional Doctor Consul Tools

K9s Plugin

Doctor Consul provides a k9s plugin that greatly assists with using k9s with Consul. Installing the plugin can be a little tricky, as it appears there is no well defined location to install the plugin.

The k9s plugin can be found in

- `./xtra/k9s/plugin.yml`
- `./xtra/k9s/plugin-mac.yml`

> I've included a version of the plugin which is ready to use visual studio code on Mac. You MUST rename `plugin-mac.yml` to `plugin.yml` first!

Through trial and error and collaboration, the following locations will likely work. Move copy the plugin to one of these locations:

Windows WSL2 Ubuntu

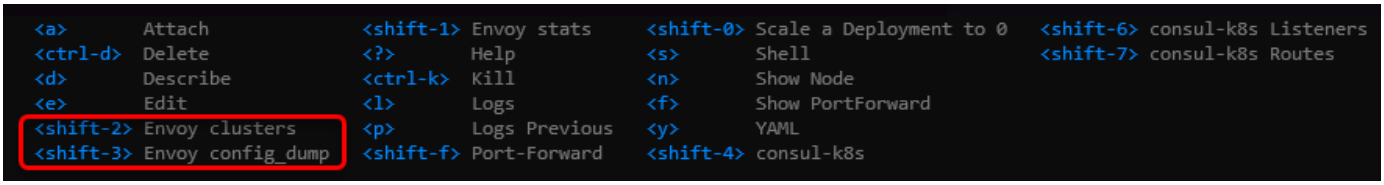
`~/.config/k9s/plugin.yml`
`~/k9s/plugin.yml` (also a location that has worked…)

MacOS

`${HOME}:`

Re-launch k9s.

New commands are now available depending on the current view. You can see them mixed in with the standard commands on the top bar:



> Some of the commands push the output into Visual Studio Code. VSC needs to be installed first, or you can modify the plugin.yml to remove the references to code

The new plugin commands are below:

:pods

`Shift+0`: Scale Pod Deployment to Zero.
`Shift+1`: Fetch Envoy `/stats` > open in VSC
`Shift+2`: Fetch Envoy `/clusters` > open in VSC
`Shift+3`: Fetch Envoy `/config_dump` > open in VSC
`Shift+4`: Run `consul-k8s proxy read` > open in VSC
`Shift+5`: Run `consul-k8s proxy read -clusters` > open in VSC
`Shift+6`: Run `consul-k8s proxy read -listeners` > open in VSC
`Shift+7`: Run `consul-k8s proxy read -routes` > open in VSC

:pods.containers

`Shift+d`: Launch a `nicolaka/netshoot` debug container for troubleshooting

> This is crucial for attaching to Consul Servers and Consul Gateways.

:deployments

`Shift+0`: Scale Pod Deployment to 0.
`Shift+1`: Scale Pod Deployment to 1.
`Shift+2`: Scale Pod Deployment to 2.
`Shift+3`: Scale Pod Deployment to 3.

Zork.sh

Zork is a catch-all script for doing many different things. It launches a menu-driven interface.

Some of the functionality is no longer necessary as Doctor Consul has evolved over time. The most useful features are below:

`Else > Change Component Versions`

: This allows you to enter a new version for Consul, FakeService, or Convoy. This modifies all of the appropriate YAML and Docker Compose variables.

  I typically keep the Doctor Consul repo on the latest supported versions, but this is a quick way to modify all of the appropriate config files. It mostly just does `sed` text replacements.

`Service Discovery > Service Discovery Template`

: Prints out example curl syntax for the `service` API and `catalog` API.

Appendix C: Fake Service Application

Nic Jackson's Fake Service is a powerful light-weight application used to demonstrate various mesh functionality. It is used over and over in Doctor Consul. It's an exceptional application to use for testing, because of its simplicity. FS can be deployed as an upstream or downstream application.

Fake Service can be found here:

- https://github.com/nicholasjackson/fake-service
- https://hub.docker.com/r/nicholasjackson/fake-service

The Fake Service configuration is defined using only ENV variables.

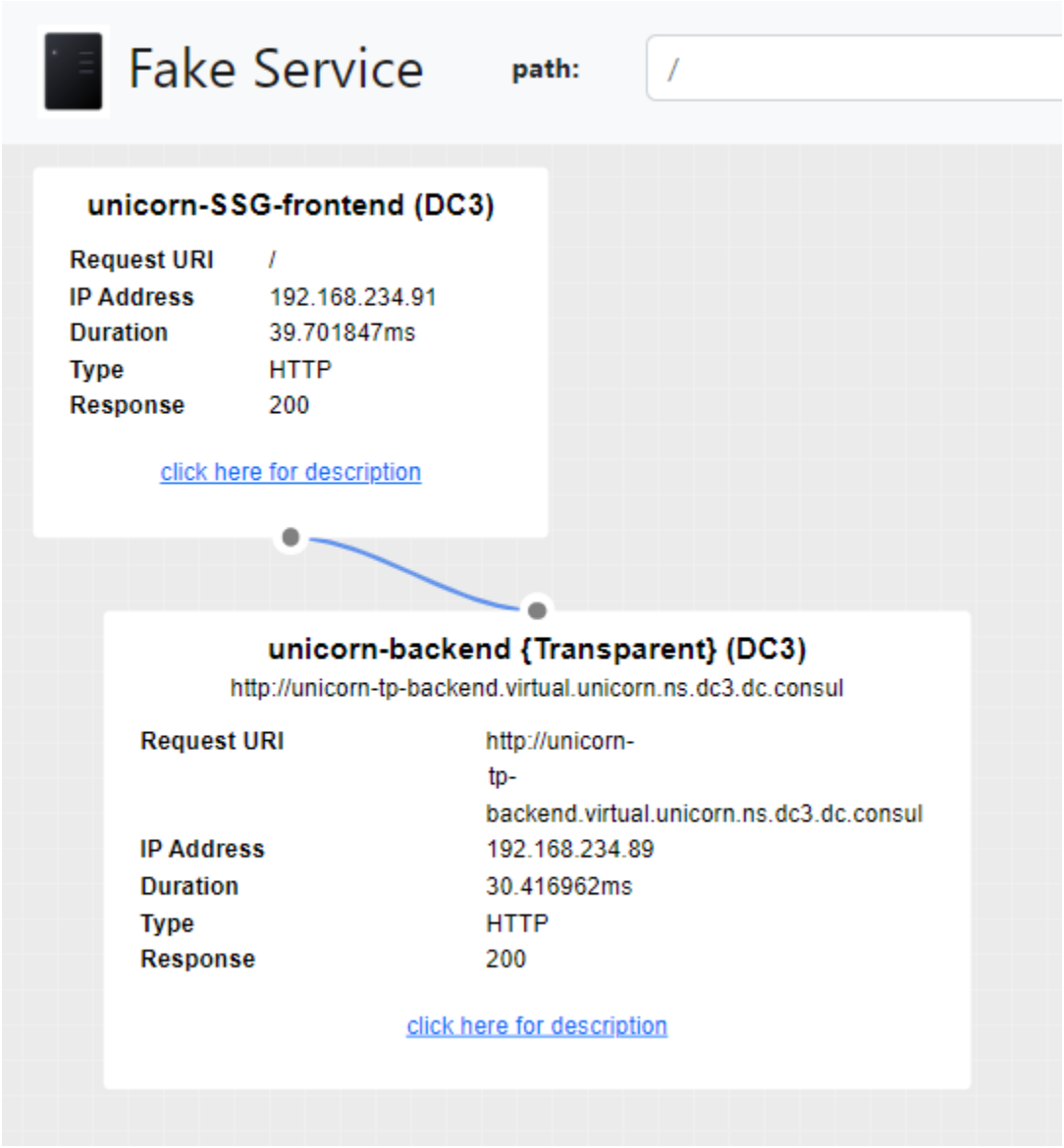
Listener Settings:

ENV Var	**Meaning**
LISTEN_ADDR: 0.0.0.0:9090	IP address and port to bind service to. Responds with a JSON output all the details of upstream services and their statuses. If accessed via a Web browser with /ui/ a UI is visualized with the more important details.
MESSAGE: "Hello World"	Message to be returned from service, can either be a string or valid JSON
SERVER_TYPE: "http"	Service type: [http or grpc], default:http. Determines the type of service HTTP or gRPC
NAME: "Service_name"	Name of the service

Upstream settings:

ENV Var	**Meaning**
UPSTREAM_URIS: http://localhost:9091	Comma separated URIs of the upstream services to call
HTTP_CLIENT_KEEP_ALIVES: "false"	Enable HTTP connection keep alives for upstream calls
HTTP_CLIENT_REQUEST_TIMEOUT: "30s"	Maximum duration for upstream service requests

Example of the Web UI (/ui/):



Every time the Listerner is accessed (via curl or browser refresh), the Fake Service application sends a request to each of its configured upstream services.

Appendix D: K9s tips

K9s is the absolute best way to interact with Kubernetes clusters for most tasks. The days of remembering 800 different `kubectl` commands is over. This section is meant to just cover the basics for new users.

Navigating K9s

K9s uses colon "`:`" menus for everything. Every "view" is accessed via `:something`.
To see context help for the current view, press "`?`"

Common areas:

- `:ctx` (contexts)
- `:pods`
- `:svc` (services)
- `:deployments`
- `:secrets`

> Any time you switch to a new view, if you see nothing, it's likely because the view is being filtered via namespaces. The number keys switch between all (`0`) and previously accessed namespaces (`1-9`).
>
> When in doubt, just hit `0` to see everything.

Forwarding Ports

In the `:pods` and `:svc` views, you can use shift+f to quickly setup local port forwards to a pod or service listener.

Tips for specific views

`:secrets`

To see an opaque secret in plain-text (base64 decoded), highlight the secret and hit `x`.

`:deployments`

To quickly scale deployments between 0 and 3 pods, use `shift+0`, `shift+1`, `shift+2`, and `shift+3` (requires the Doctor Consul k9s plugin).

`:pods`

Pods can be quickly deleted using `control+d`. The pod deployment will automagically create a new pod. This is great for forcing changes. Whenever changes are made to Terminating Gateways or service configs and you don't get the expected behavior, just start nuking pods. Devops'ing is fun!

`:pods.containers`

To see the `containers` view of a pod, first go to :pods, highlight the desired pod, and hit enter. From within this view, you can hit `shift+d` to launch a debug container. (requires the Doctor Consul k9s plugin).
This allows you to shell into any of the restricted Consul pods, such as Terminating Gateways, so you can fetch troubleshooting details and perform tests.