Learn Kubernetes + Microservices

A very long time ago, applications used to combine all functionalities into big machines, often called mainframes. Although this approach made sense then, soon cheaper commodity hardware, distributed processing started making way for more scalable ways of developing and maintaining software systems. With many internet scale applications taking over daily routines, came different approaches to deploy and scale, slowly converging towards fundamentals of scale. One of the most popular ones being Virtual Machines. Although a generic concept in many operating systems, virtualization has seen many shapes and sizes.

Companies started creating specialized softwares called “Hypervisors” to host different OSs in machines. But why do all this? With the growing need for organizations to run different software ranging from the business logic systems, reporting systems, internal portals and what not; for efficient utilization of hardware, it made sense to pack more things into machines and utilize them efficiently; after all these were big machines. But managing such systems again took a lot of effort, often having to run hundreds of servers, and things were often slow (comparatively). The main requirement was to have applications running in machines, to be isolated and can be controlled.

Jump to 2008, linux kernel started to build the same capabilities, without needing special tools -- this was a game changer! Now running many such applications seems similar to opening a tab in the browser or running the music player in your phone, things just start to work with the same kind of isolation hypervisors used to provide without the baggage of having to “boot” into a different OS. Oh and did I mention, linux can run any *nix OS as a base for the application. Linux all along was it’s way was going towards this journey, with different approaches along the way like chroots and jails, it’s always been in the works. Afterall, one of the main functions of an OS was to have process isolation - be it processor, memory, disk or network. With the addition of cgroups and namespaces, it was possible to run a different OS itself as a process, and for that process it would look like an isolated compute unit. Each process was becoming another abstract machine.

Thus containers were born. Docker is one the most popular tools for creating such “containers”, them being a package which contains the OS, the application and the supporting libraries or softwares it requires, all put into one standalone, self-sustaining box. These can run anywhere, and run the same way it runs in your laptop, on the servers too.

Now, with organizations creating more complex applications, they started having teams managing parts of such big projects, and since this would have it's own development workflows, deployment cycles and processes, breaking down big applications into smaller such applications which work together started becoming popular. Also, with that the focus on systems which can manage such complex applications efficiently. One such framework is Kubernetes (a.k.a K8s). Simply think of this as a system that makes it easy to deploy, scale, and manage a bunch of containers. One can say, it's almost like running your own cloud (indeed).

This guide contains a lot of links to help you navigate through the activities, do take time and read through.

Now let's start hacking!

Concepts

You will find many resources which will help you to navigate through the exercise.

Docker

Start by installing docker. Get Docker

Docker containers or images are created by writing a file called Dockerfile. These files are used to give commands to the docker daemon to create such images. These images then can be deployed to different environments.

> Get started by going through this link: Orientation and setup

> Try creating your own images and running them by following this simple tutorial: Sample application

> To see some of the commands to navigate around docker: 🐳 ELI5: Docker | docker-explained

Now that you have a way to create docker containers, let's hop to some kubernetes (K8s) concepts.

K8s Control Plane and Nodes

Kubernetes has a few components, put together, called the control plane, which orchestrates containers deployed on K8s. Now, the control plane itself requires some resources or servers to run, which is not usually used to run applications. Servers which run applications are called Nodes. These also have some softwares installed which works along with the control plane to provide a nice interface to manage your applications. More about this here: Kubernetes Components

Pod

Pods are the smallest deployable unit in K8s, can have one or more containers, which can share things, like network.

Deployment

Deployment is more like a template to create replicas of such pods. Say you want 5 copies of your application, you can use deployments rather than manually creating 5 pods. Specify the template, configurations and number of pods required, voila!

Service

Service is a way to address a set of pods. This is done using tags in pods which are used to select the pods under a service. Using service, K8s provides a way to call them using K8s's internal DNS system: Kubernetes DNS for Services and Pods

Confusing much? Let's try some of these things out.

Even though K8s is deployed mostly in clouds or on-premise servers, you don't necessarily need those to start playing around with kubernetes. There are many projects like - kind or minikube which helps to run K8s on your laptop.

Follow this guide and create a kubernetes cluster using *kind*: kind – Quick Start

Or if you prefer *minikube*, minikube start (usually this needs more resources)

We will be basing this exercise on kind, but once you have kubernetes up and running using kind or minikube, things would be similar.

Once you're comfortable with these topics, move on to the problem statement. The following section will be divided into groups of problems, each having reference guides and hints to accomplish the tasks. Go through the guides and try to accomplish the goals mentioned in each section.

The Setup

Here, we are going to start building a simple microservice application, and deploy and monitor, using K8s.

Start by unzipping the code-base: k8s-learnathon

Follow the readme to start setting up the application using the *kind* cluster created.
k8s-learnathon/README.md

The README covers:

1. Creating a cluster using kind
2. Walkthrough of the application (microservice)
3. Sample deployment configurations

Task 0:

Follow the readme, setup a cluster and deploy the sample microservice application.

Now that you have the application running, let's improve this stack.

Activity 1: **Monitoring**

When there are many servers and even more applications running inside each, monitoring resources and applications becomes important. Systems break, applications crash. To build a robust product, monitoring becomes key. With kubernetes and other cloud native solutions, a Time Series Database called Prometheus is often used. Prometheus collectes metrics from agents and applications which are running in the cluster and provides a query language to create meaningful visualizations.

Another open source dashboard called Grafana is used to create graphs and metrics with the use of prometheus.

Let's start by installing Prometheus into the cluster and collect metrics like Node and Pod cpu, memory, etc.

Guides:

1. How to monitor your Kubernetes cluster with Prometheus and Grafana
2. prometheus-community/helm-charts: Prometheus community Helm charts
3. grafana/helm-charts

Note: The guides follow the older version, checkout links 2 and 3 for installing prometheus and grafana respectively.

Use the above guides as a starting point to set up prometheus and grafana. Most of these would be using predefined templates for creating all the components. The popular way is to use Helm. Prometheus and Grafana helm charts can be found in the links above.

If you want to customize your graphs, take a look at this introduction to prometheus query language or PromQL: PromQL tutorial for beginners and humans | by Aliaksandr Valialkin | Medium

Look through the collections of publicly available dashboards here: Grafana Dashboards - discover and share dashboards for Grafana.

Task 1.1:

Setup Prometheus and Grafana and have some dashboards working, monitor the resource utilization for the application we created.

- Prometheus Node Exporter Full dashboard for Grafana
- Kubernetes / Compute Resources / Cluster dashboard for Grafana
- Kubernetes / Compute Resources / Node (Pods) dashboard for Grafana

- Kubernetes / Compute Resources / Pod dashboard for Grafana

Guides: If you want to start collecting more metrics for a particular pod, it has to expose more metrics, often using a prometheus client which is integrated with the application. These integrations can be done in the applications using supported clients or using exporters.

Annotations can be used to tell prometheus to also scrape the pods when clients or exporters are configured. Annotations are key-value pairs that can be added to pod/deployment specs. For scraping using prometheus these annotations can be used:

- `prometheus.io/scrape`: The default configuration will scrape all pods and, if set to false, this annotation will exclude the pod from the scraping process.
- `prometheus.io/path`: If the metrics path is not /metrics, define it with this annotation.
- `prometheus.io/port`: Scrape the pod on the indicated port instead of the pod's declared ports.

More information: Kubernetes & Prometheus Scraping Configuration

Task 1.2:

Sending more traffic to the UI and see if this reflects in the metrics. More traffic should use more pod resources. You can use something like Hey to generate load.

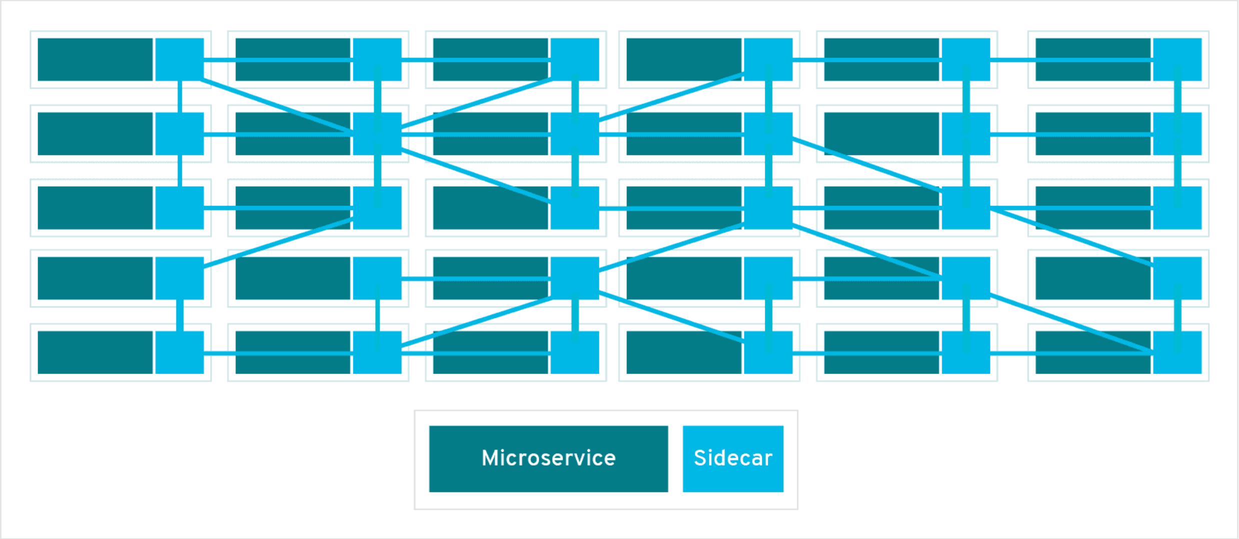
When you're done with this activity, take a break and move on to the next one.

Activity 2: **Mesh**

When things were running in big machines, it was easier to manage fewer such servers than having to deal with thousands of things when applications started to be broken down into microservices. This new found complexity can be tamed by using the right practices. Having good monitoring to see the health of systems was being important. But this alone wasn't enough.

With teams building different parts using different languages and libraries, teams who deploy, and monitor these workloads need to be also aware about how each of these pieces behave under load, how to fix something when errors happen. With kubernetes, some of these challenges were resolved by having unified ways of deploying. But when it comes to monitoring and unified management, it was still becoming more complex. Imagine having to deploy hundreds of services and making sure everything is running correctly, debugging what is not working, would involve nightmares.

A new pattern of handling microservices emerged, called service mesh. Usually a lightweight proxy running along with the application. Often called a sidecar proxy, it abstracts certain features, such as inter-service communication, security and monitoring, away from the main application to ease the tracking and maintenance of the application as a whole. Just as a sidecar is attached to a motorcycle, a sidecar proxy is attached to a parent application to extend or add functionality.



This pattern also makes it easy for people to now only work on the service mesh level, and not worry about application complexity. The monitoring, logging and security policies configured at the service mesh now applies to any application without having to do anything special as the incoming and outgoing traffic to the application is always going through the sidecar proxy.

Alright! During the next set of tasks, we will be converting the microservices application to create our own service mesh (-ish). A reverse proxy accepts a request from a client, forwards it to a server that can fulfill it, and returns the server's response to the client. This also enables collection of metrics and logs while the requests are being processed.

Envoy

We will be using Envoy Proxy. It is a modern, high performance, small footprint edge and service proxy. Envoy is most comparable to software load balancers such as Nginx and HAProxy. We prefer using envoy, because of its performance and all of it is API driven.

Envoy is used in many of the popular cloud native service mesh implementations. It’s dynamic, scalable design helps with its adoption.

Guides:

- Envoy 101: Configuring Envoy as a Gateway
- Get Started with Envoy on your Laptop | by Mark McBride
- Service Mesh with Envoy 101. In this article we will briefly discuss… | by Arvind Thangamani | HackerNoon.com
- Example Configs

A sample configuration can be found in the code-base:
k8s-learnathon/k8s/configs/envoy-example.yaml

This configuration proxies the request from frontend-service from the microservice, this can be used as a reference to build more of the mesh. Here, we used a couple of concepts from kubernetes to make things easy.

Config Maps

It's a way to inject configuration files to a set of pods dynamically without having to hardcode it in the docker image. It makes it possible to use the same container images with different configurations at runtime. These configs can be mounted as files into pods or even used as environment variables. Reference: Kubernetes Docs - ConfigMaps

Here, a Volume Mount was used to inject this configuration file into the pod.

- Kubernetes Docs - ConfigMaps
- Volumes

Sidecar

The concept of multiple containers, or called sidecars, is a way to add envoy proxy along with the application. Sample configuration can be found here: Example Kubernetes pod for the multi-container sidecar design pattern Use this as a reference to create pods with sidecars. Read more about creating multiple containers in pods: Pods – Kubernetes Documentation

Now that these primitives are available, let’s start by creating the mesh.

Task 2.1:

Add a second container in all the microservice applications. For frontend, pinger and details services, add a sidecar envoy container which intercepts inbound and outbound traffic. Wire up the services correctly by adjusting the environment variables for the frontend service deployment. Remap the service configs in k8s to reflect this new change.

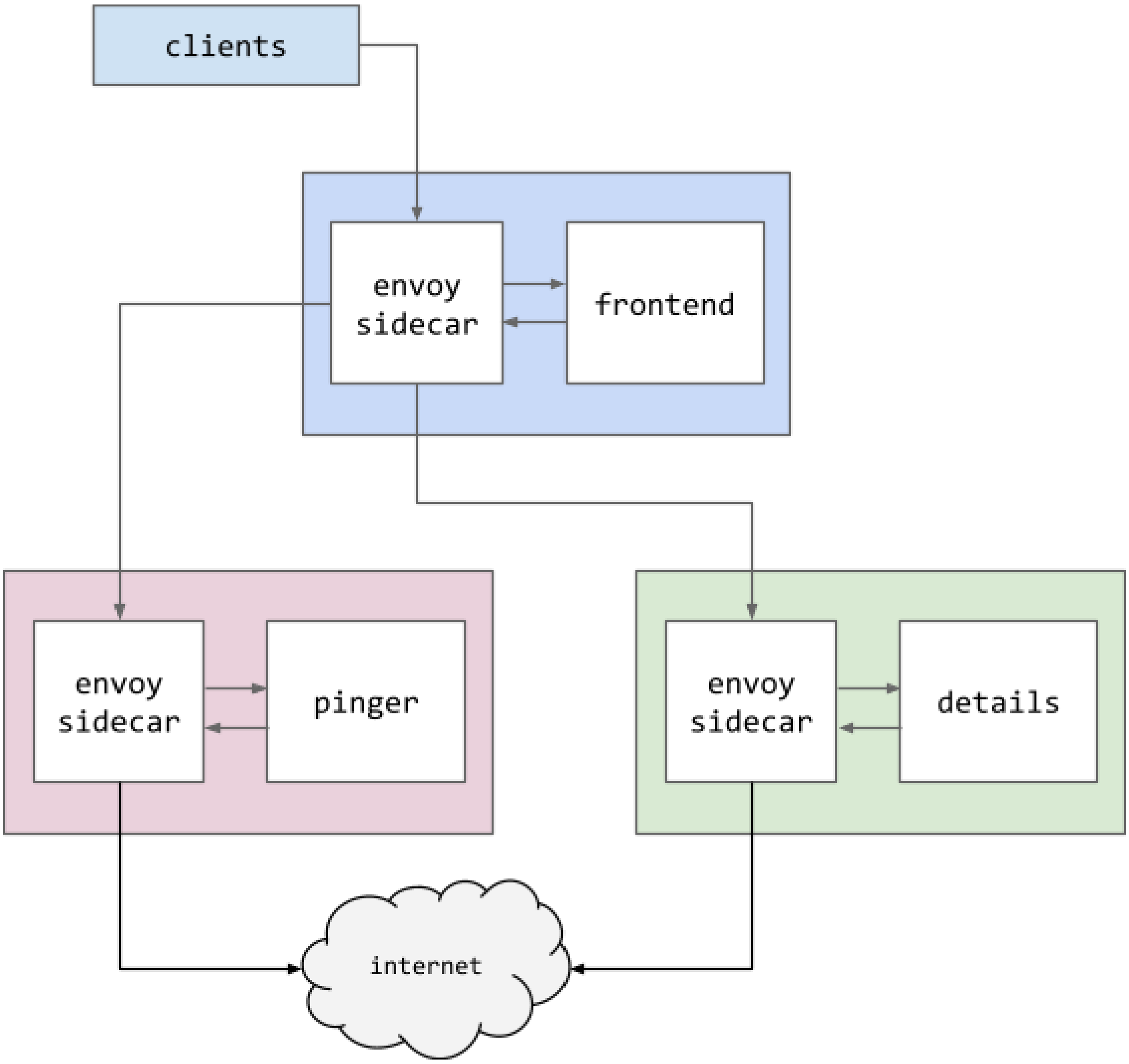
Once this is ready, we can start setting up metrics to see how we can monitor the traffic using the prometheus and grafana setup.

Envoy offers a lot of metrics exposed by the proxy itself. Checkout these guides to see how they work.

- Microservices monitoring with Envoy service mesh, Prometheus & Grafana
- transferwise/prometheus-envoy-dashboards

Task 2.2:

Use annotations to enable prometheus scrapping for all the microservices. Build grafana dashboards to visualize the flow of traffic. With this, you should be able to see the requests coming in and going out of each service, latency and HTTP status codes.



Rough schematics of the microservices

Activity 3: **Stagger**

Now that we have the basic mesh setup, let's start by deploying a different version of the pinger service. The sample deployment config uses the v1 version of the pinger service. The v2 version of pinger service has more details, which will show up in the UI.

Canary Deployments

When dealing with large systems, it is usually risky to push new versions of applications directly into production. Even when applications are tested, it could still behave differently when deployed into production. To de-risk this, deployments are done by sending some percentage of the traffic to the new versions. Say, 1% of traffic goes to the new version and 99% goes to the old version. If things go well, the traffic is rolled up in parts, staggering more traffic to the new version, like 5%, 10%, 50% and eventually the whole traffic, gradually reducing the old version to 0%. This helps in making sure there are no regressions. In case of new bugs, the new version can be quickly rolled back to 0%, making the old version serve the full traffic. This strategy is called Canary Deployment.

Weighted Cluster

Using this primitive, multiple envoy clusters can be mapped, and used to stagger traffic based on percentages. Check below references on how to achieve this:

- Traffic Shifting/Splitting — envoy 1.19.0-dev-63307f documentation
- Implementing Blue / Green rollouts | envoyproxy
- Set of Envoy Proxy features demos | unofficialism

Task 3.1:

Create a weighted cluster for the outbound traffic for the frontend service. Start by cutting over 5% traffic to pinger-v2 service. Use the grafana dashboards to show the shift in traffic. Continue the cutover by gradually increasing the traffic to 50%. See the changes in the grafana dashboard. Finally cutover pinger-v2 to 100%.

Hot Restarting

When a configmap is updated, the file which was injected in the container also gets updated. But this necessarily doesn't mean that the application, in this case envoy, would automatically pick this up. One way to get the updated configuration is to restart or recycle the pod. This isn't ideal as it would affect the traffic, and cause downtimes. A better approach is to use envoy's hot-restart feature.

When used, it updates and reloads the configuration seamlessly without affecting the live traffic.

Now, this is a bit tricky to get it right, here are some guides:

- Hot restart — envoy 1.19.0-dev-73ade9 documentation
- Hot restart Python wrapper — envoy 1.19.0-dev-3e9678 documentation
- Apr 21, 2019: Envoy Hot Restart Mystery
- https://github.com/infinitewhile/hot-restarting-envoy

Task 3.2:

Create a script to automate the cutover, by adding configs dynamically, using envoy's hot restart feature. The script can take the new and old service names, and also the % weights for each. You can choose any preferred scripting language.

Troubleshooting

1. If you local cluster is crashing or restarting, probably try increasing the resources allocated to docker: Docker Desktop for Mac user manual

2. If you're using Windows, and if docker is consuming a lot of memory, vmmen process consuming too much memory (Docker Desktop)

3. WSL2 for running docker on Windows: WSL+Docker: Kubernetes on the Windows Desktop

4. Kind known issues: kind – Known Issues

5. Troubleshooting application on K8s: Troubleshoot Applications

---