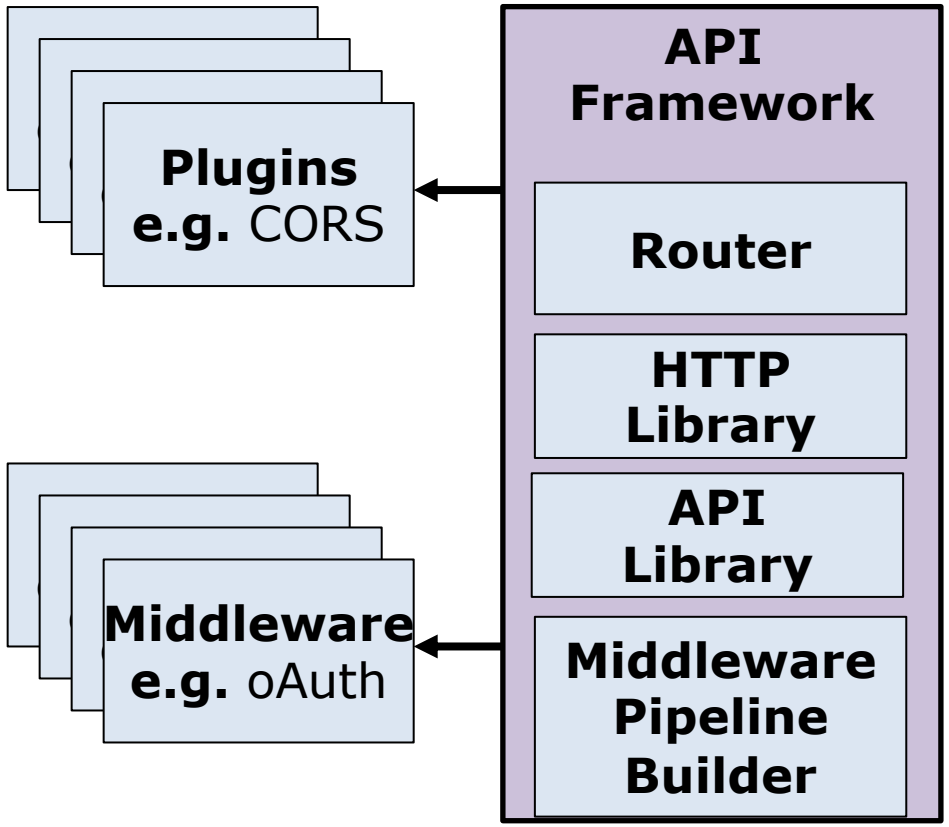
SE 577
Software Architecture

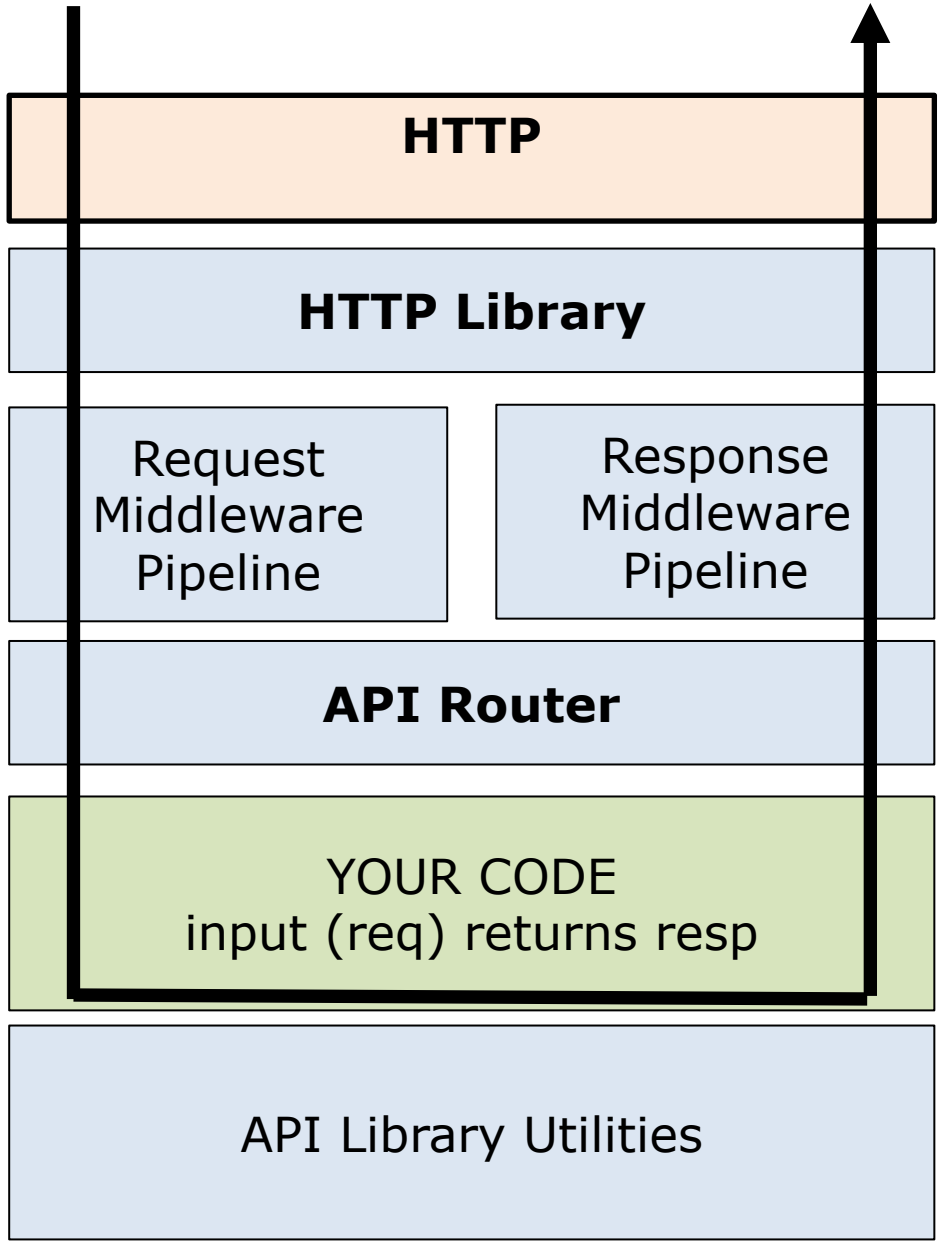
Linux, Docker & Kubernetes (k8s) Architecture

API Architecture



Most API Frameworks/Libraries
Use a Repository Style Architecture

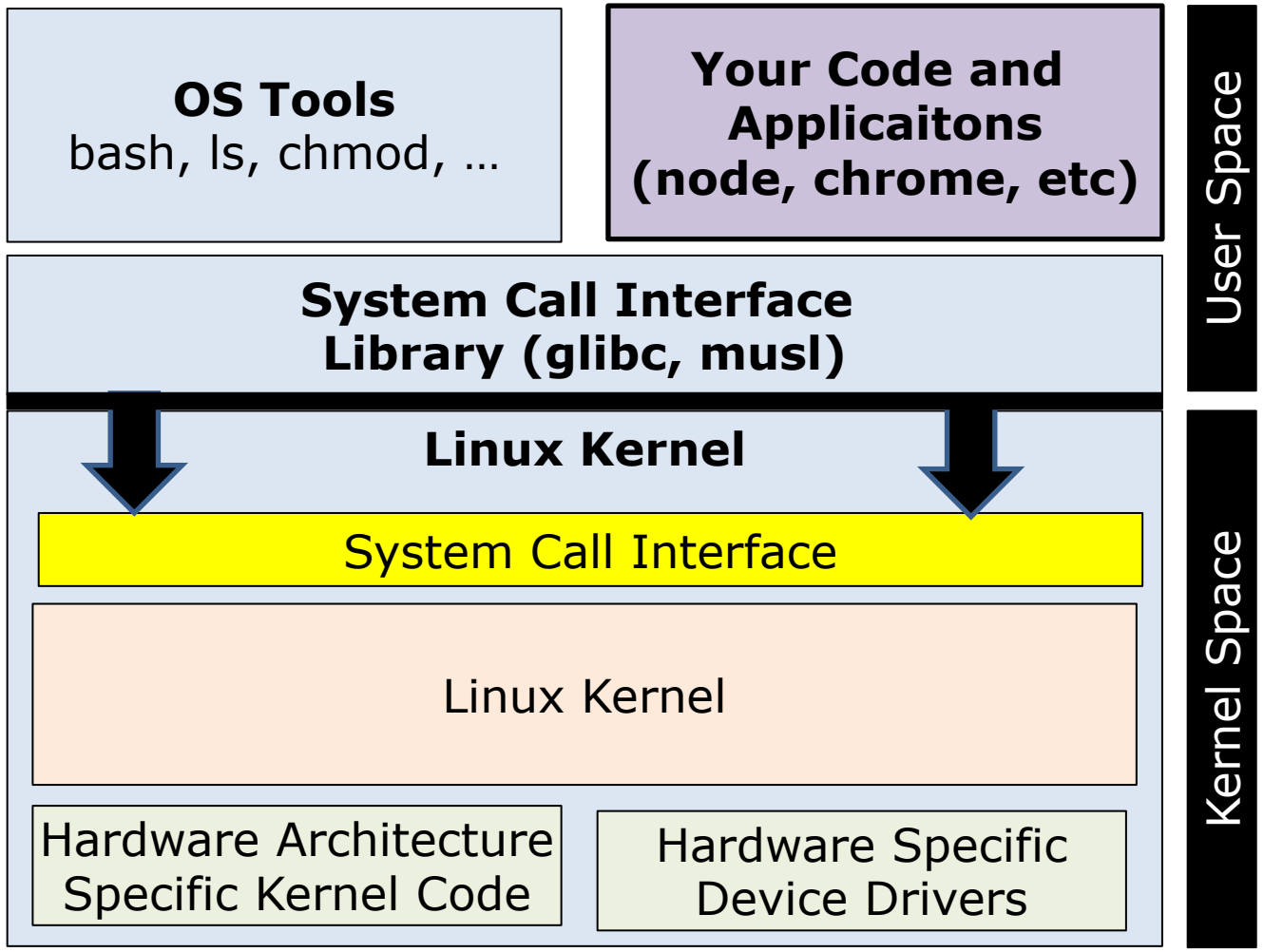
We have used several in this course
Koa, Fastify, GoLang Gin, Ktor, etc



How you should think about this lecture and the remainder of the lectures in this class

- We will be looking at real world things that have very interesting architectures
- You might be thinking as I introduce and walk through some of the material – “Isn't this an architecture class?” – It is, but to get to some of the interesting aspects of the architecture, tradeoffs, constraints, etc – we need to have a fundamental understanding of the technology itself

Linux Architecture



Note the actual linux architecture is significantly more complex, but this should suffice for our needs

Linux is a layered architecture, the kernel sits close to the hardware and even has some hardwre specific code (e.g., x-64 vs ARM).

The code we run, tools, custom code, etc sits at the top layer

User code leverages the system call interface to bridge between user space and kernel space

When you pick a Linux distribution, they basically provide the stuff in the blue boxes

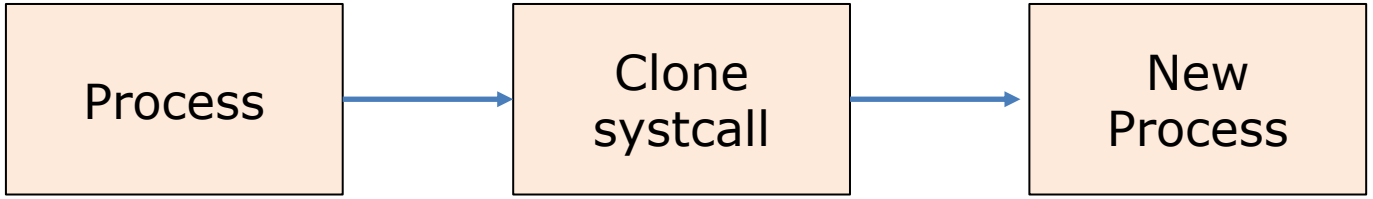
Linux Namespaces

When linux boots, it synthesizes process Id (pid) 1 - its called the `init` process, often implemented by the `systemd` process. This process is used to spawn all other processes in the system.

One way linux can create a child process is via the `clone` syscall. By default the new child process runs in the same execution context as the parent. **Processes in the same execution context share a common linux namespace.**

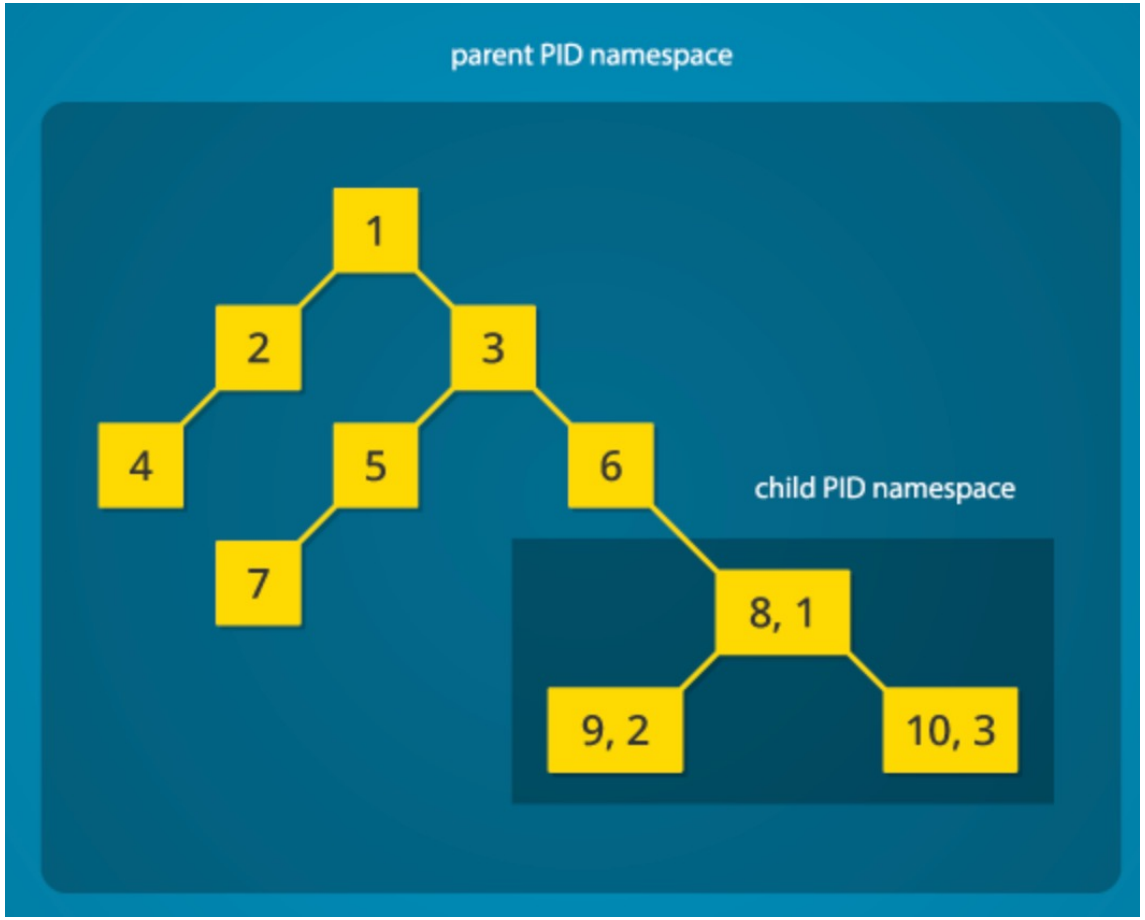
The clone system call also provides options to allow the child to be put in separate namespaces.

Linux currently supports 7 different namespaces – CGroup, IPC, Network, Mount (filesystem), PID, Time, User (user permissions and groups), UTS



Processes that have attributes that run in a common namespace are not isolated, processes with different namespaces are

Linux Namespaces – Example, cloning a child process into a new process/pid name space



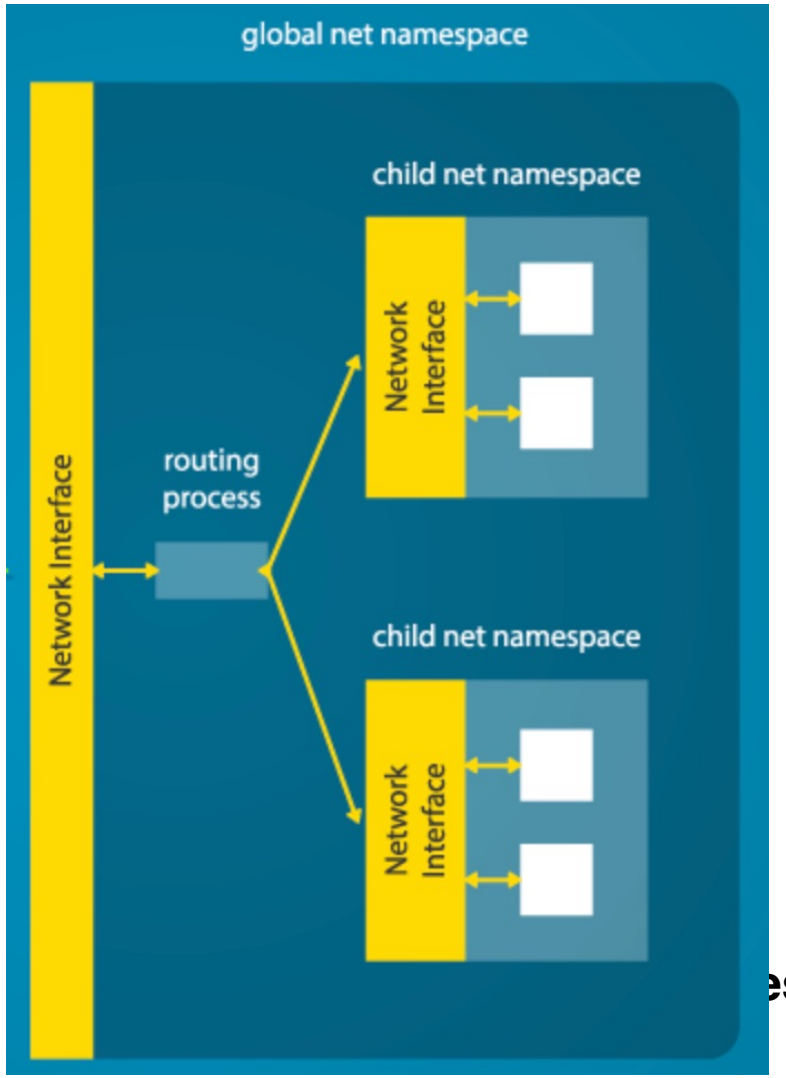
Ultimately every process is given a unique id in the guest operating system. Thus if you listed all processes in the system you would see PID 1-10.

However look at pid 6 it cloned PID8, and PID8 cloned both PID9 and PID10

The cool thing is that if you asked PID 8 what its process ID is it would come back as 1; Same for pid9 and 10, they would come back as 2 and 3.

Thus everything in the child namespace can see each other, but it cannot see anything in the partent namespace.

Linux Namespaces – Example, cloning a child process into a new network namespace



Ultimately every process is given a unique id in the guest operating system. Thus if you listed all processes in the system you would see PID 1-10.

However look at pid 6 it cloned PID8, and PID8 cloned both PID9 and PID10

The cool thing is that if you asked PID 8 what its process ID is it would come back as 1; Same for pid9 and 10, they would come back as 2 and 3.

espace can see each other, but it cannot see anything in the partent

Linux Namespaces – allow for logical isolation in linux

Namespace types

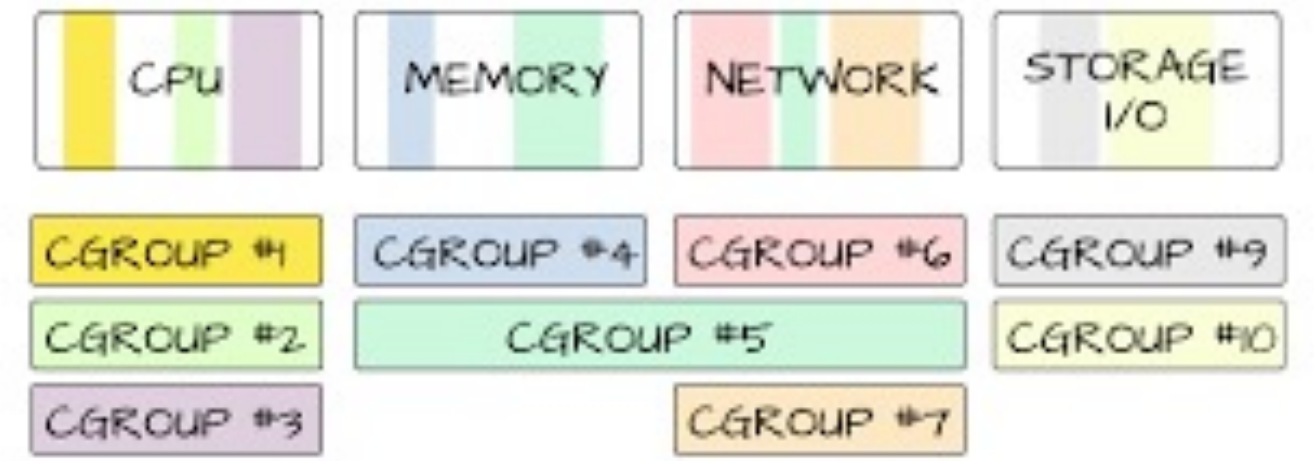
The following table shows the namespace types available on Linux. The second column of the table shows the flag value that is used to specify the namespace type in various APIs. The third column identifies the manual page that provides details on the namespace type. The last column is a summary of the resources that are isolated by the namespace type.

Namespace	Flag	Page	Isolates
Cgroup	**CLONE_NEWCGROUP**	**cgroup_namespaces**(7)	Cgroup root directory
IPC	**CLONE_NEWIPC**	**ipc_namespaces**(7)	System V IPC, POSIX message queues
Network	**CLONE_NEWNET**	**network_namespaces**(7)	Network devices, stacks, ports, etc.
Mount	**CLONE_NEWNS**	**mount_namespaces**(7)	Mount points
PID	**CLONE_NEWPID**	**pid_namespaces**(7)	Process IDs
Time	**CLONE_NEWTIME**	**time_namespaces**(7)	Boot and monotonic clocks
User	**CLONE_NEWUSER**	**user_namespaces**(7)	User and group IDs
UTS	**CLONE_NEWUTS**	**uts_namespaces**(7)	Hostname and NIS domain name

If you clone a child process into a new namespace for all namespace types, it will for all practical purposes be fully isolated from the parent.

Also, if the isolated process creates new processes it will inherit by default the same namespace as the parent thus allowing process groups to be isolated

Linux Cgroups – managing resource limits on collections of processes - /sys/fs/cgroup/…



Cgroups allow resource limits to be attached to a collection of processes.

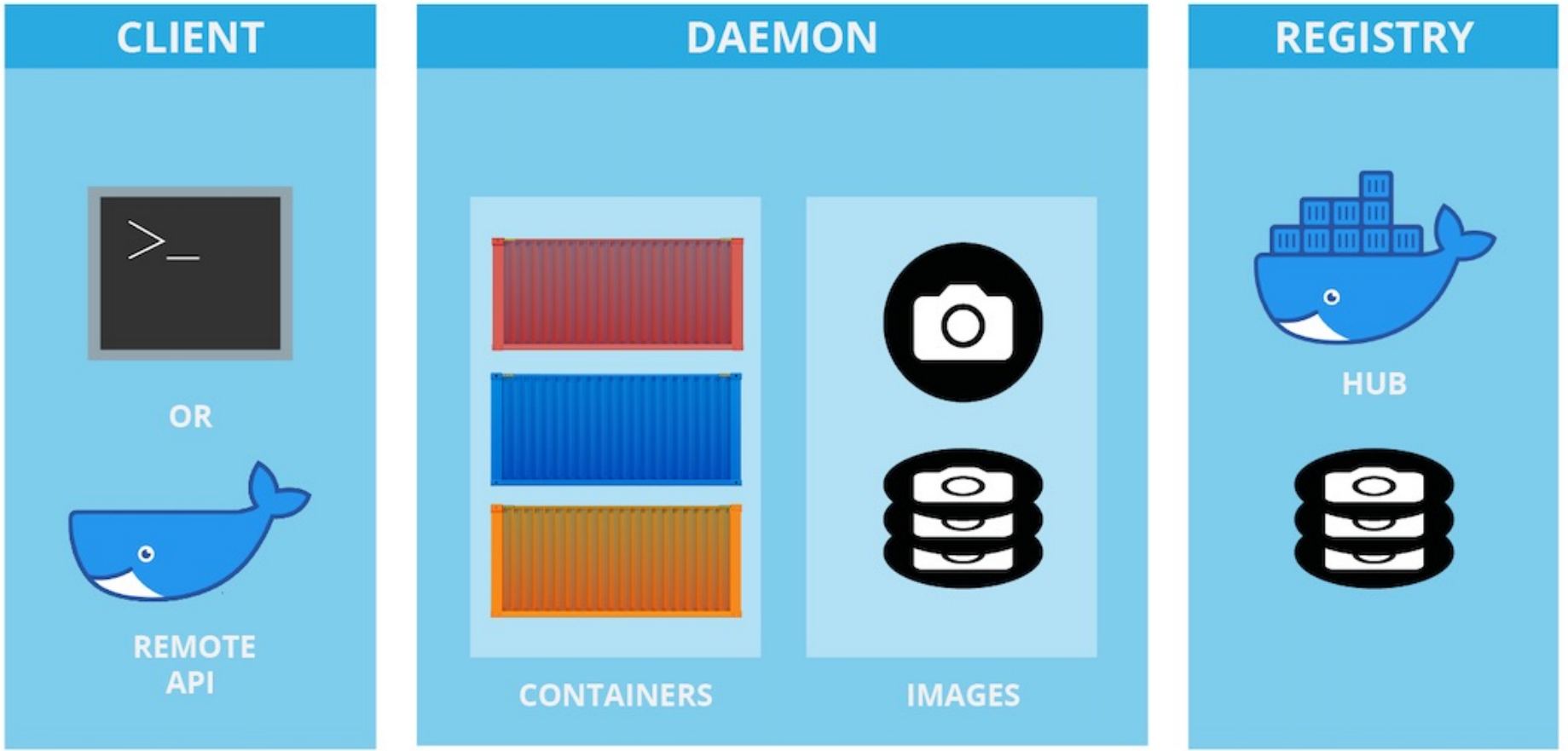
For example you might want to restrict how much network and file I/O can be used and also cap CPU at ½ of a core

```yaml
resources:
  requests:
    memory: "64Mi"
    cpu: "250m"
  limits:
    memory: "128Mi"
    cpu: "500m"
```

In this example we are requesting an initial allocation of 64Mb of memory and 250 millicores (1/4 of a physical core), and allowing it to burst up to the limit of 128Mb of memory and 500 millicores (1/2 of a physical core

Docker

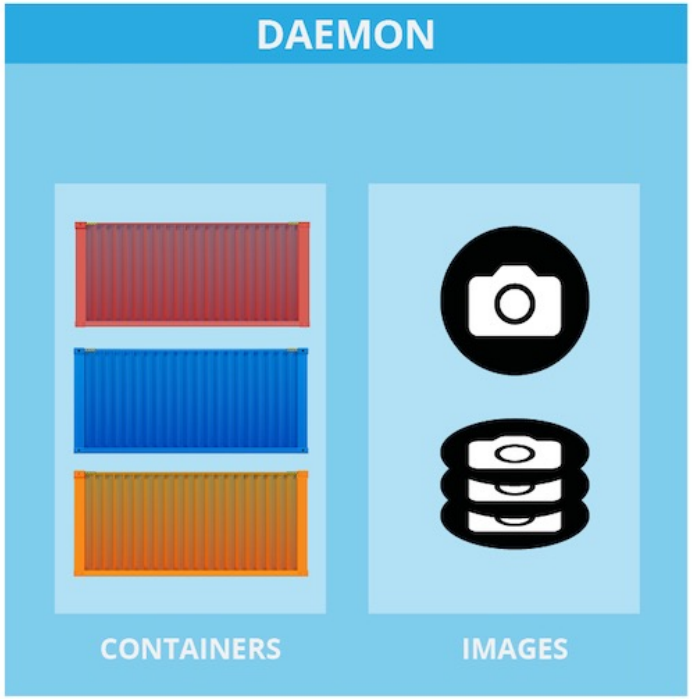
Now that we know how linux can isolate process groups (namespaces) and resources (cgroups) we can better understand the docker architecture



Managing Docker | Running Containers | Getting Containers

Docker Daemon and Linux

On windows and Mac the docker daemon is required and runs a customized, container optimized version of linux in a hypervisor. On linux machines the daemon does not use a hypervisor because it has access to the linux kernel



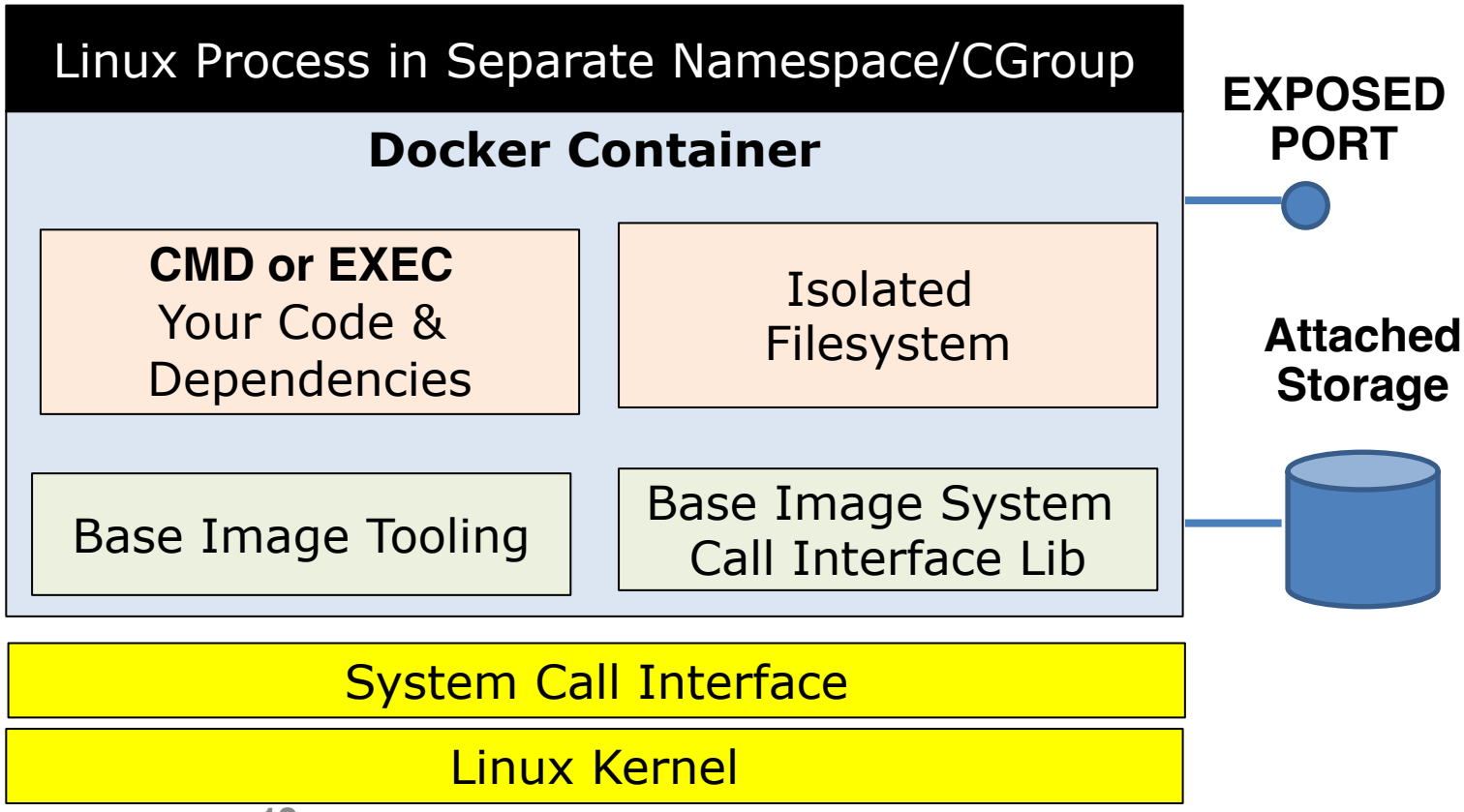
General Purpose Linux Distributions

Container Optimized Linux Distributions

The docker container

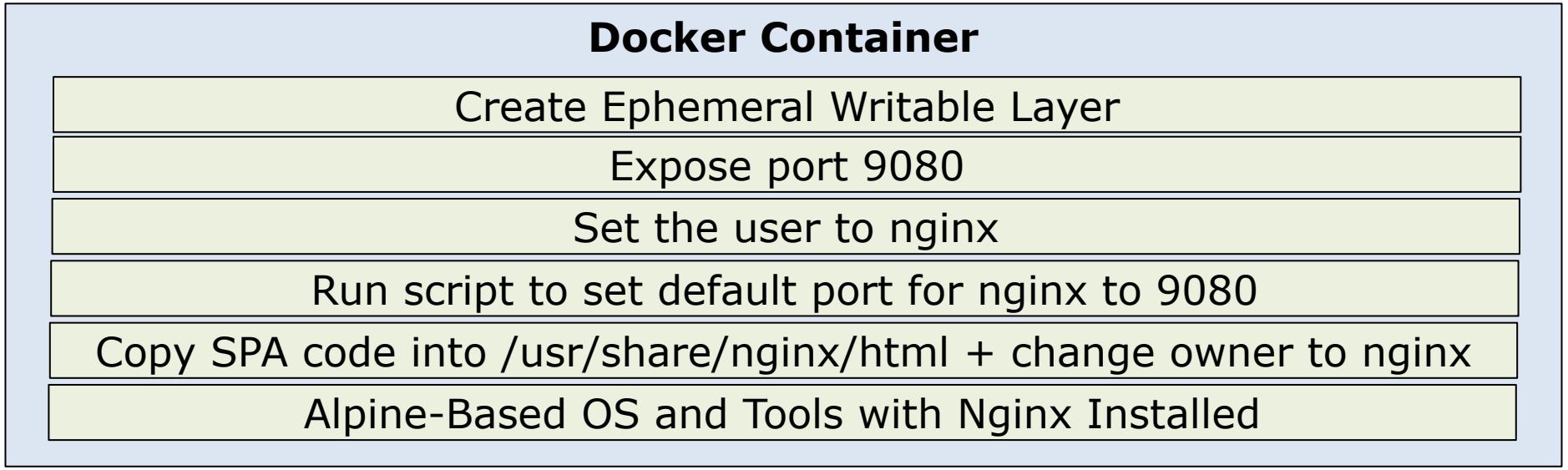
The docker container is a portable package containing everything needed to run your workload. It adheres to the open container initiative specification https://opencontainers.org/

The shipping container transformed logistics



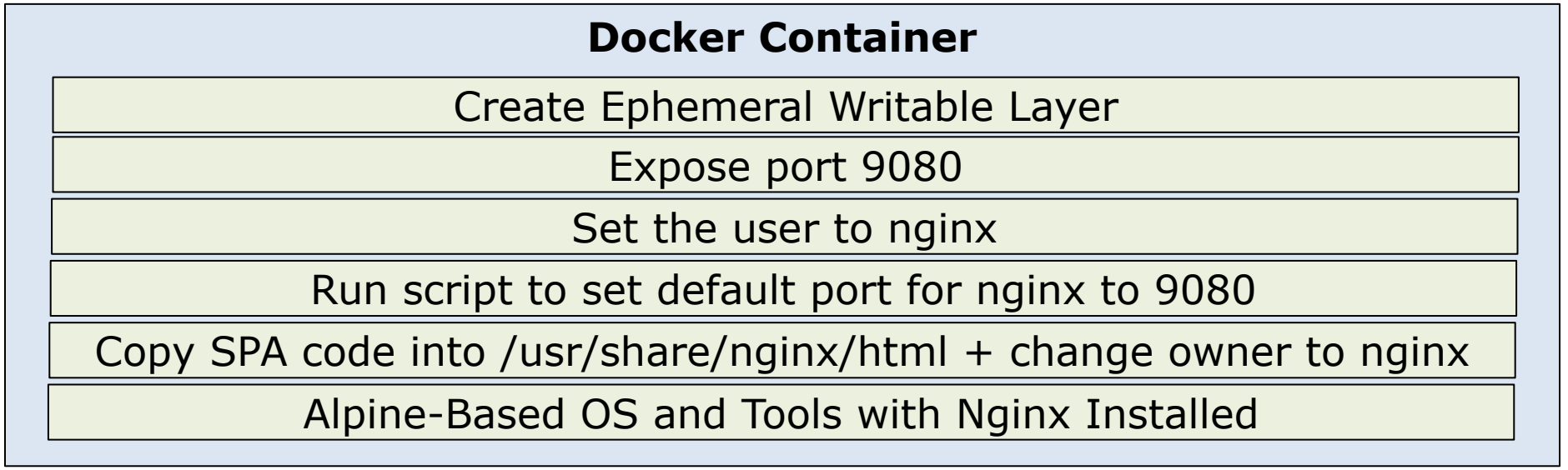
Building Docker Containers

```
FROM nginx:alpine

COPY --chown=nginx:nginx ../dist/spa /usr/share/nginx/html
RUN chmod g+rwx /var/cache/nginx /var/run /var/log/nginx

RUN sed -i.bak 's/listen\(.*\)80;/listen 9080;/' /etc/nginx/conf.d/default.conf

USER nginx

EXPOSE 9080
```

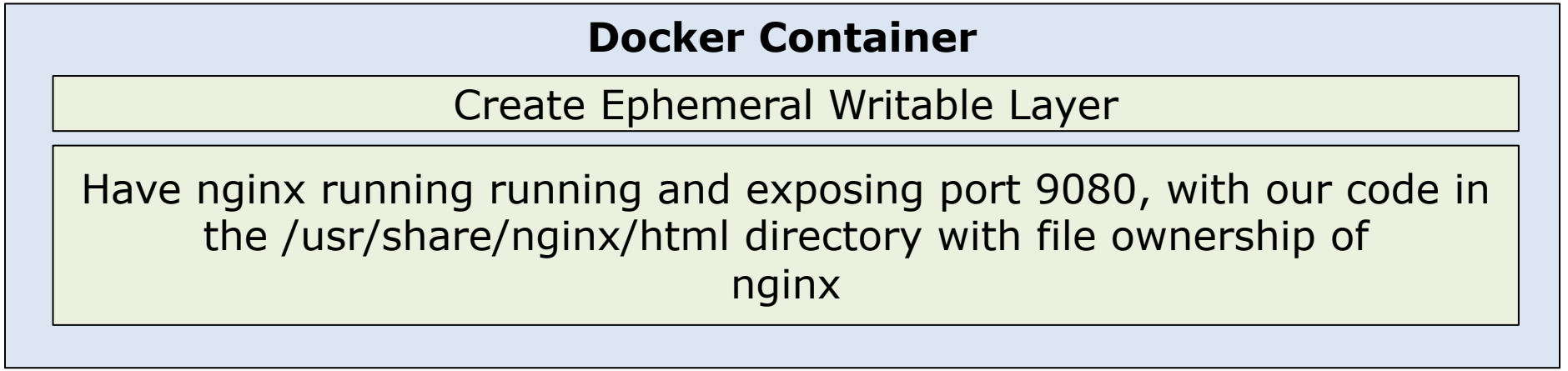


Building Docker Containers

```
docker build -t architecting-software/se577-demo-app -f Dockerfile .
```




```
docker build --squash -t architecting-software/se577-demo-app -f Dockerfile .
```



Building Docker Containers

Its not uncommon to create a build process where you create a build container first and then use the output of that build container to create your final container

Build Container

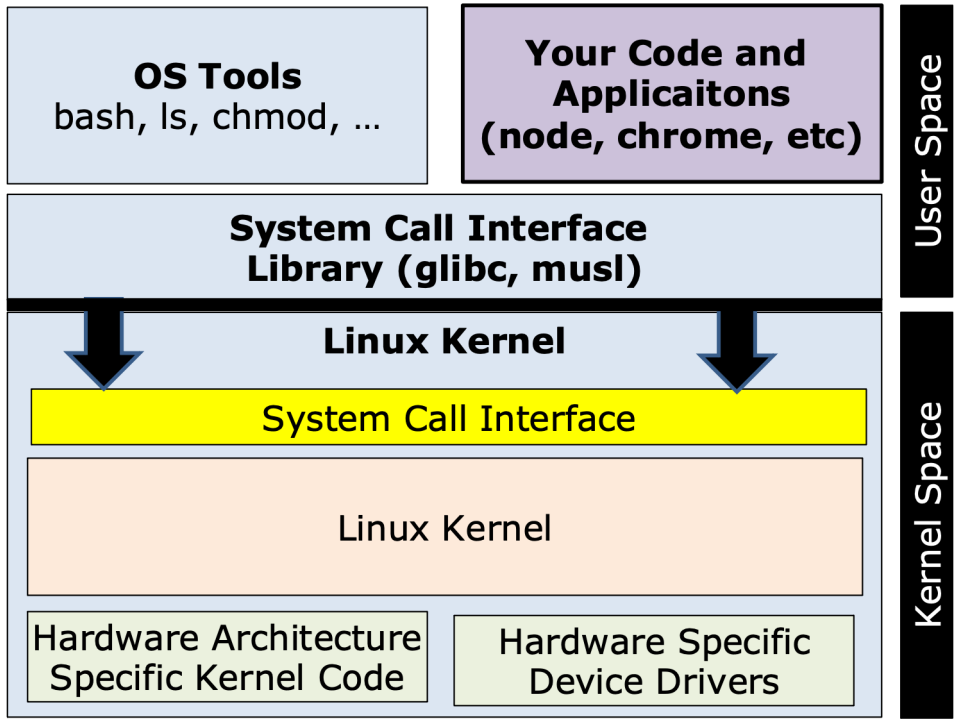
Run Container

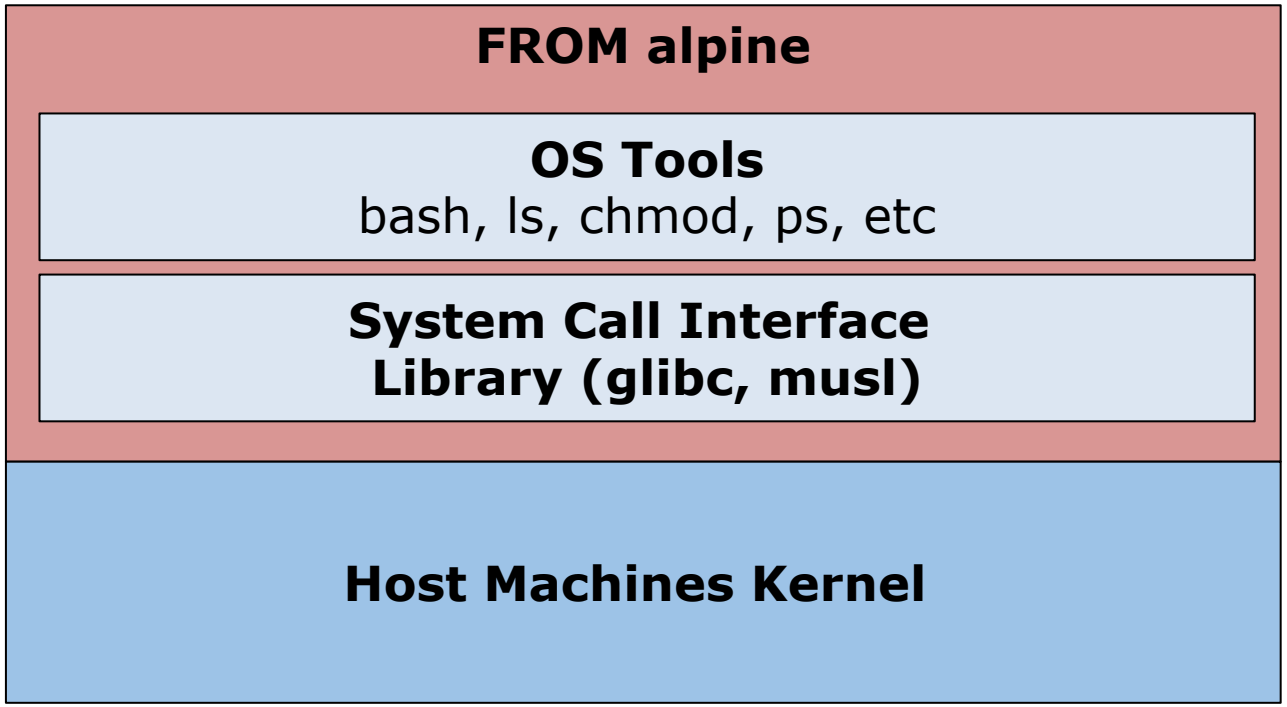
```
FROM golang:latest as builder
LABEL stage=builder
#PHASE 1:  Build the GO Binary - aka builder container
WORKDIR /go/src/drexel.edu/bc-service/go
COPY    go* ./
COPY    src ./src
RUN     CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -v -o bin/bcservice ./src

#PHASE2: Build the final container, notice the directory naming follows the package prefix in go.mod
FROM alpine
WORKDIR /bin/
COPY --from=builder /go/src/drexel.edu/bc-service/go/bin ./
ENTRYPOINT ["/bin/bcservice"]
EXPOSE  9095
```

What OS Kernel does a container use?

Note the base container is created from an OS image in this example

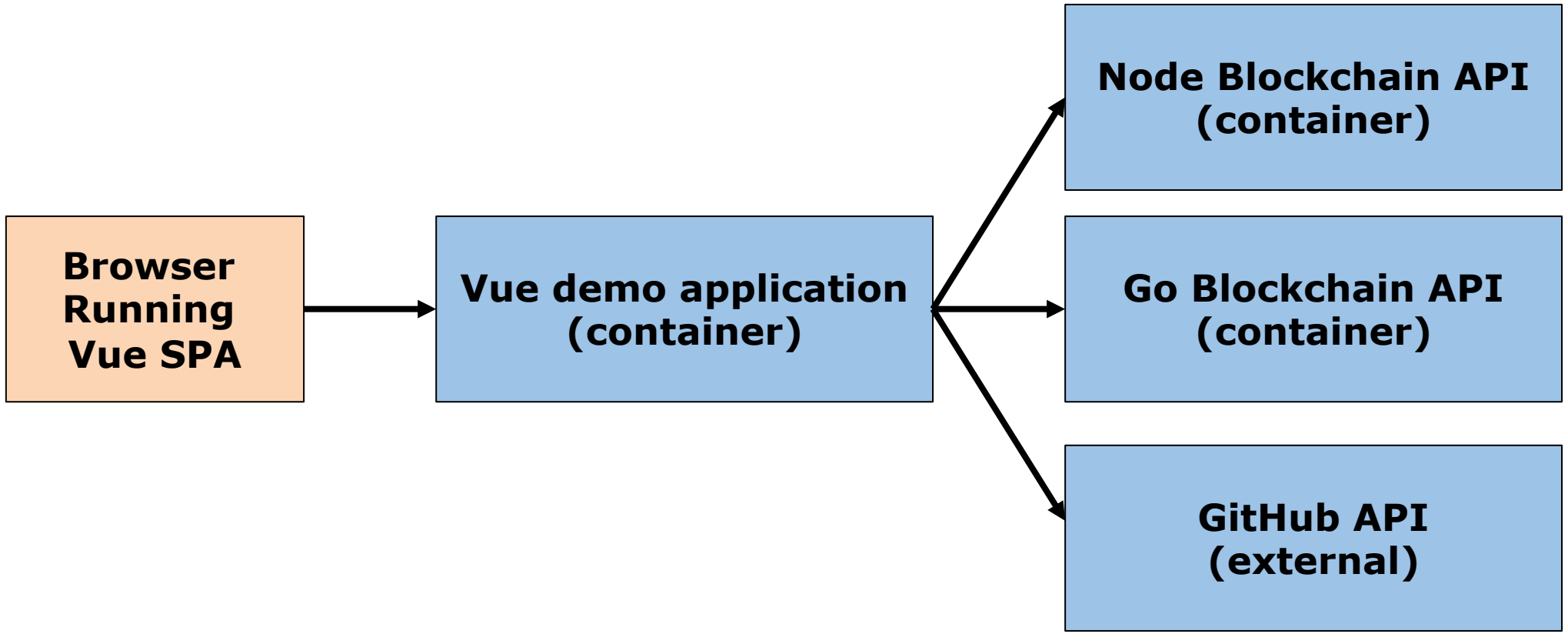
```
#PHASE2: Build the final container, notice the directory naming follows the package prefix in go.mod
FROM alpine
WORKDIR /bin/
COPY --from=builder /go/src/drexel.edu/bc-service/go/bin ./
ENTRYPOINT ["/bin/bcservice"]
EXPOSE  9095
```





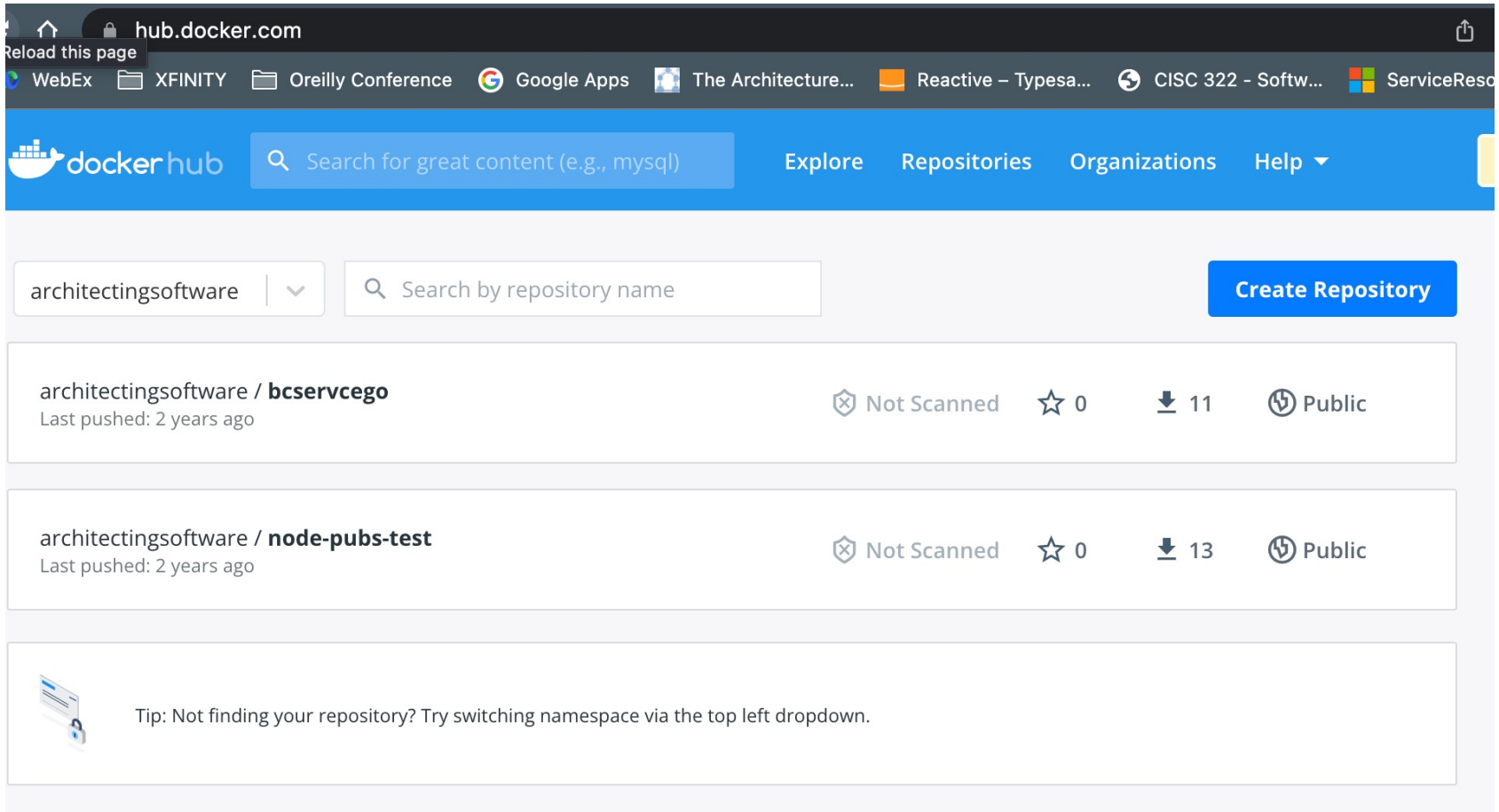
Architecting for containers

Generally containers run one process, and are expected to do one thing. Thus we need a way for containers to work together to build useful architectures



Container Repositories

After containers are created, they are generally pushed to a container repository. These could be public or private



Manage Container Versions

Push Containers on Build

Pull Containers When Needed

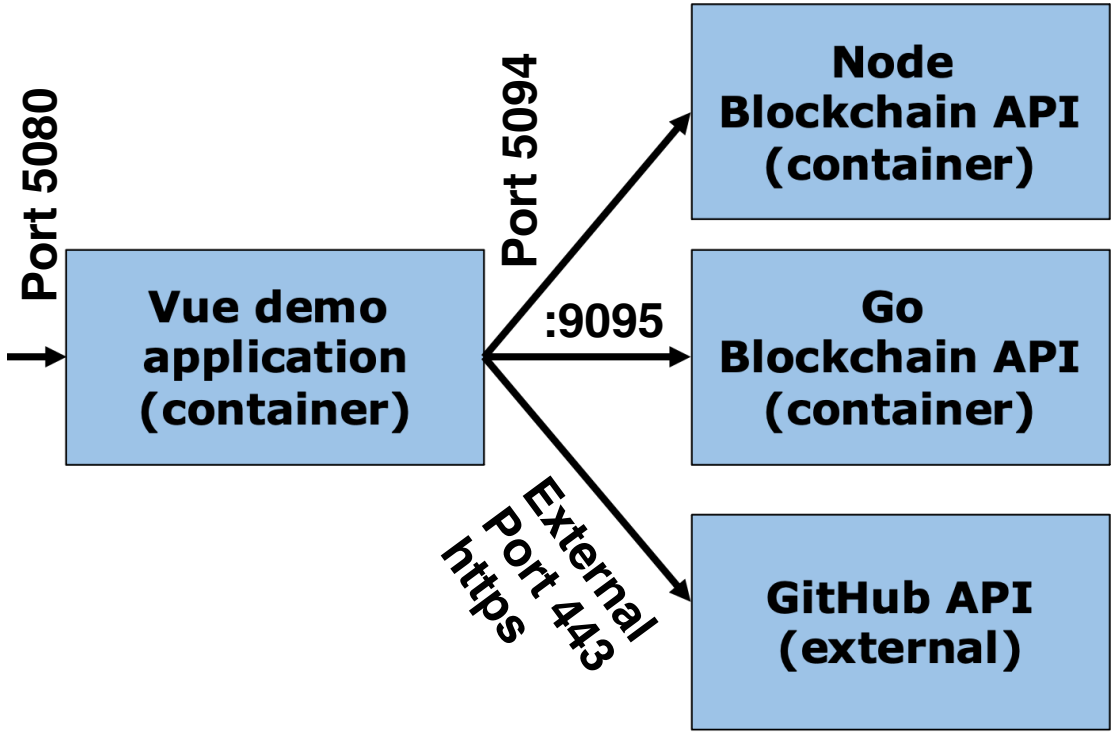
Container Repositories

Container repositories also keep track of the container layers and can do other things like scan for security vulnerabilities

Using Docker-Compose

Generally containers run one process, and are expected to do one thing. Thus we need a way for containers to work together to build useful architectures

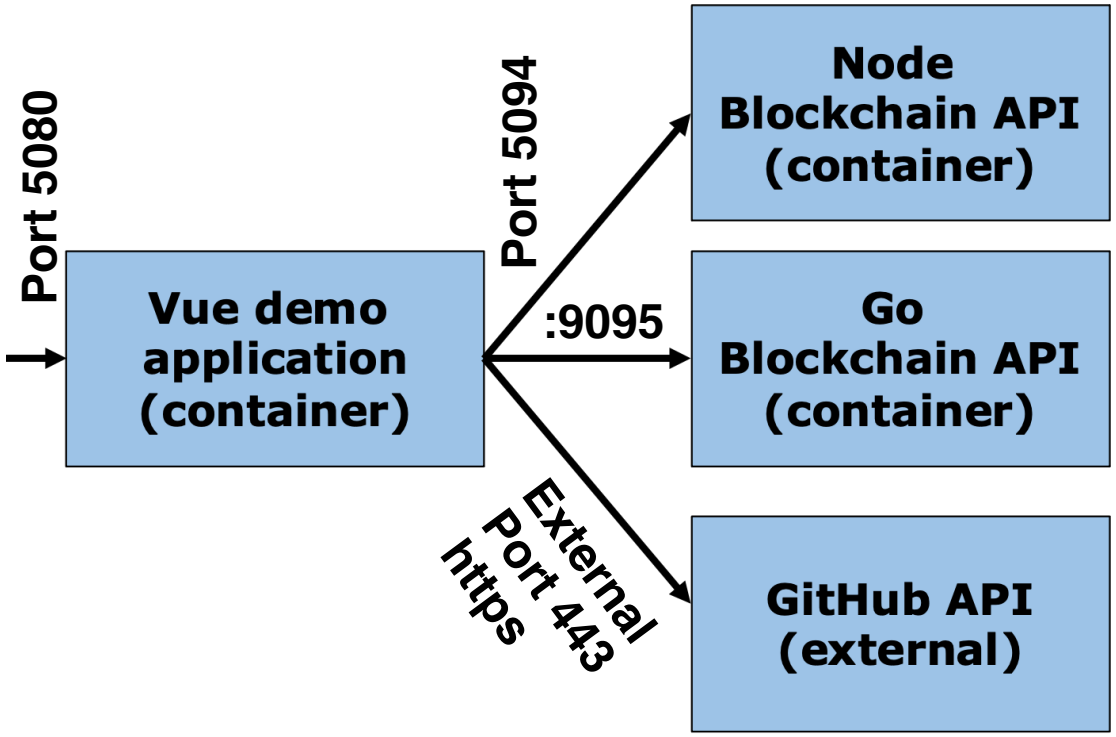
```yaml
version: '3.8'
services:
  gowebservice:
    image: 'architecting-software/bc-service-go'
    ports:
      - '5095:9095'
  nodewebservice:
    image: 'architecting-software/bc-service-node-fastify'
    ports:
      - '5094:9094'
  webserver:
    image: 'architecting-software/se577-demo-app'
    ports:
      - '5080:9080'
```



Docker compose allows us to bring up collections of containers via docker-compose up, and docker-compose down

Architectural Issues with Docker-Compose

Docker compose does not supervise the running environment



What happens if the process in this container crashes?

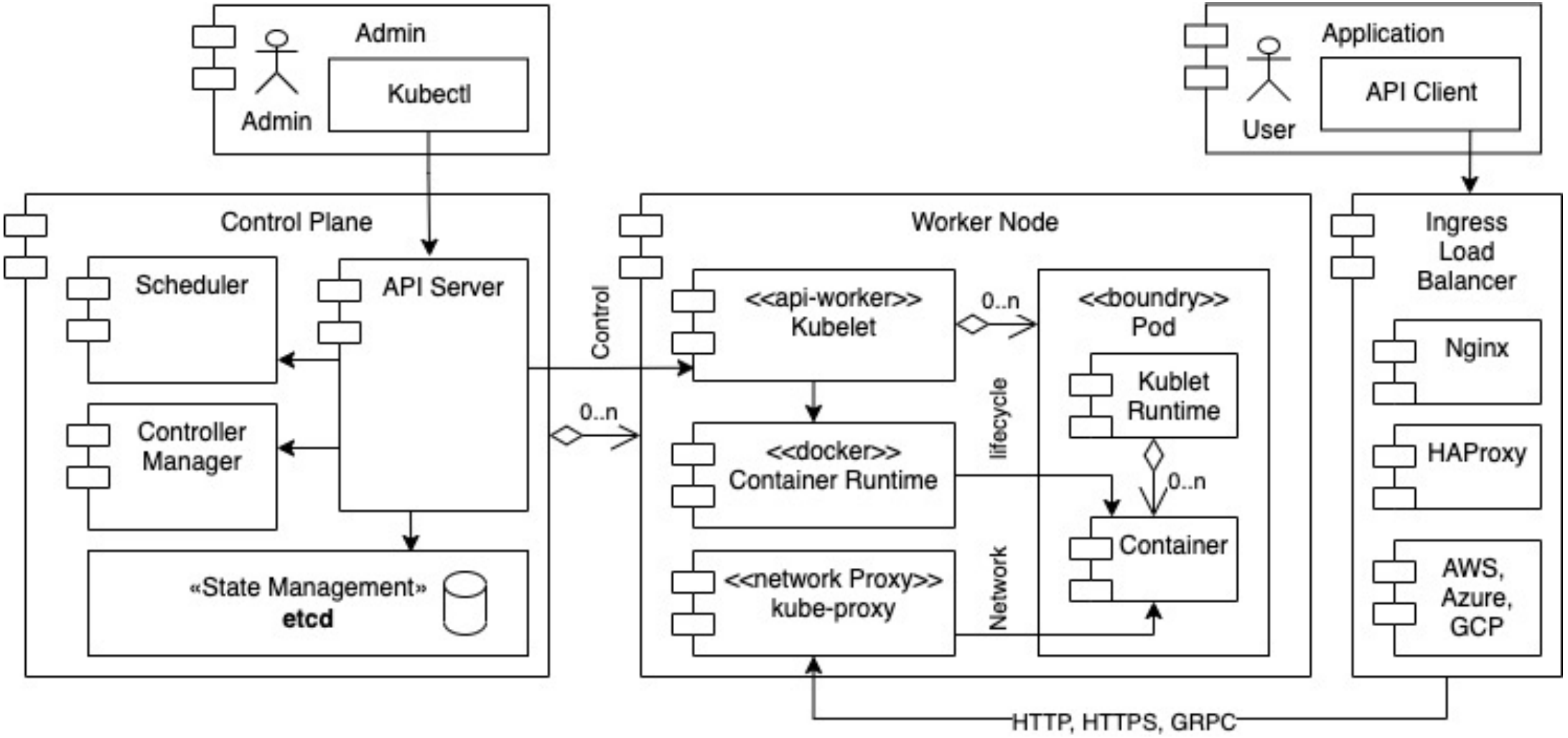
What happens if this container is getting slammed with traffic?

Running Containers at Scale In Production

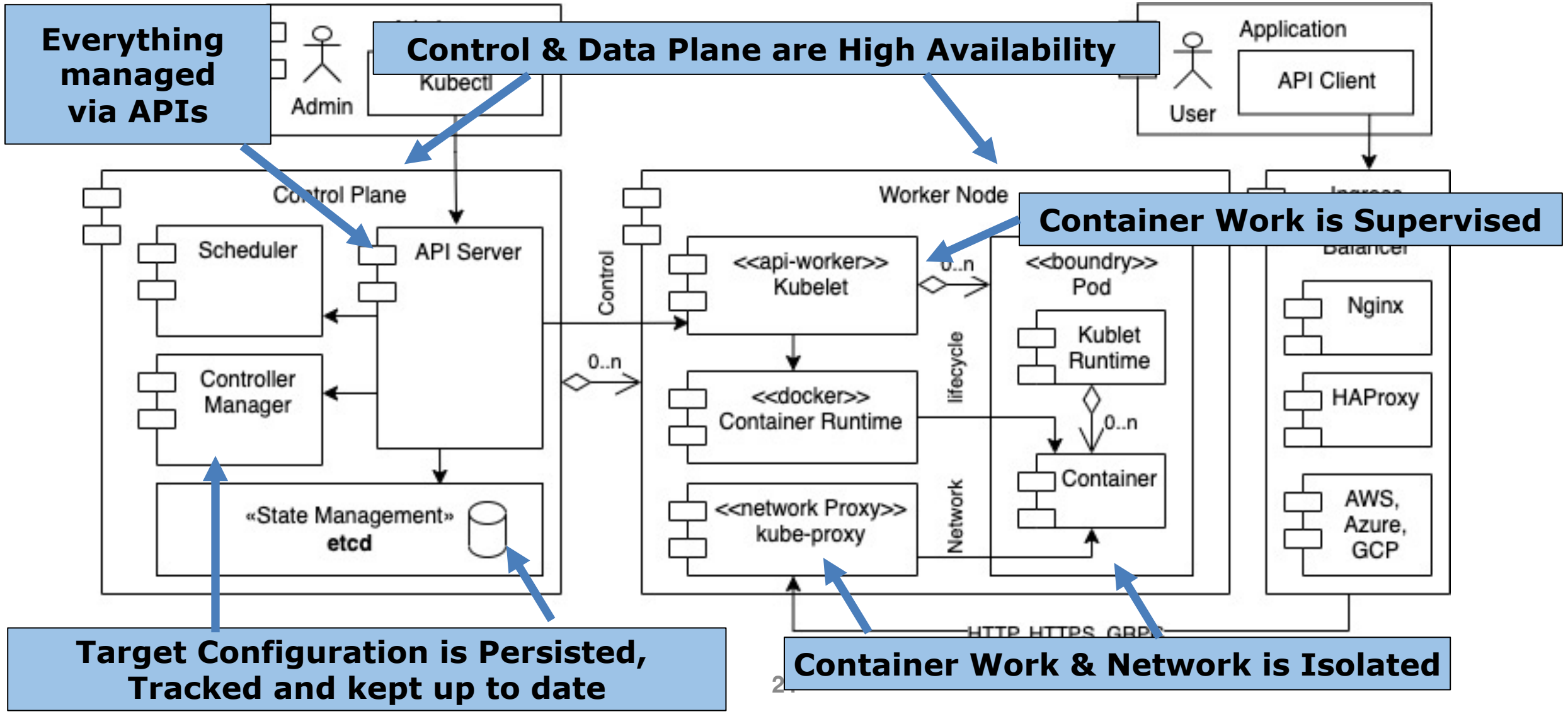
- Containers are ephemeral – this is a big deal – every time its run it has the same starting state
- Containers can be distributed via container repositories and inherit other benefits like ensuring the proper containers are used, they are secure, etc
- But containers alone are not enough:

  - They need to be “supervised” – are they healthy, have they crashed, do they need to be restarted?
  - They need to be distributed to prevent against issues when their runtime crashes
  - They need to be aware of load so they can autoscale up, and autoscale down
  - They need to be isolated, so only container workloads that interact with each other can interact with each other

Running Containers at Scale In Production – Kubernetes or k8s

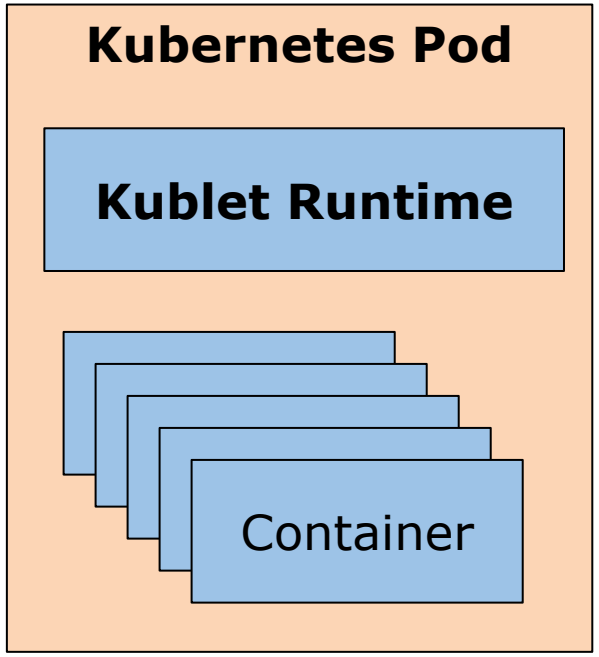


Running Containers at Scale In Production – Kubernetes or k8s

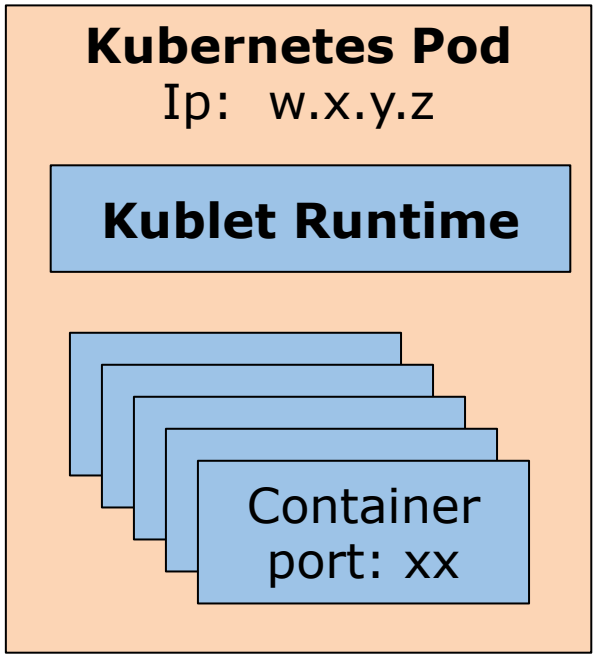


Running Containers at Scale In Production – Kubernetes or k8s - Pods

Most of the time pods run one container, the kubelet manages the pod runtime



Namespaces are used to govern what is shared between containers in the "pod" such as IPC, and things across the Kubernetes cluster such as network



Each k8s pod is given a routable IP address

Each container in the pod can be reached via a port on that IP address

Running Containers at Scale In Production – Kubernetes or k8s – Core K8s Objects

Managing Kubernetes can be complex, but you can get by with just a few of the k8s objects