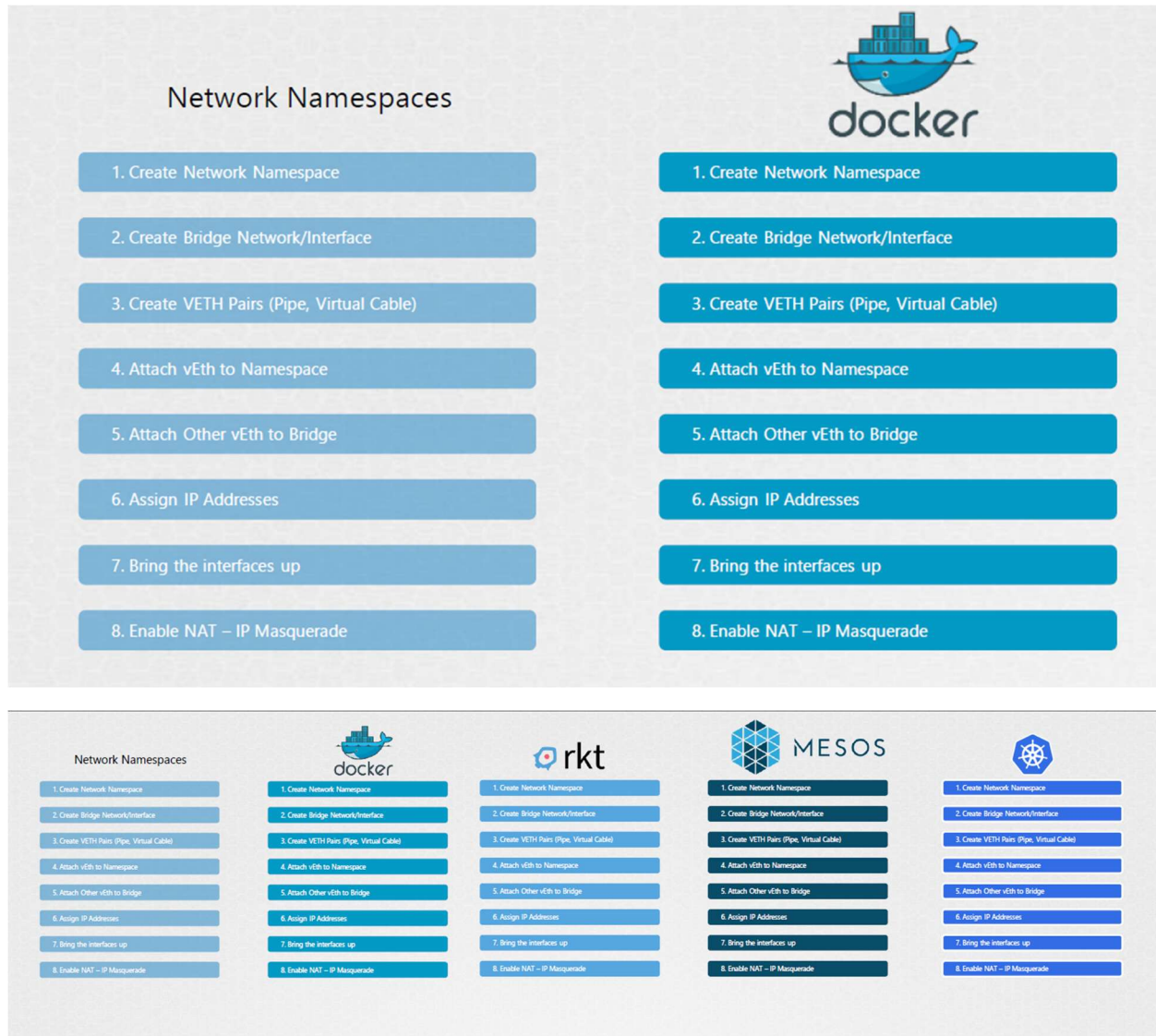
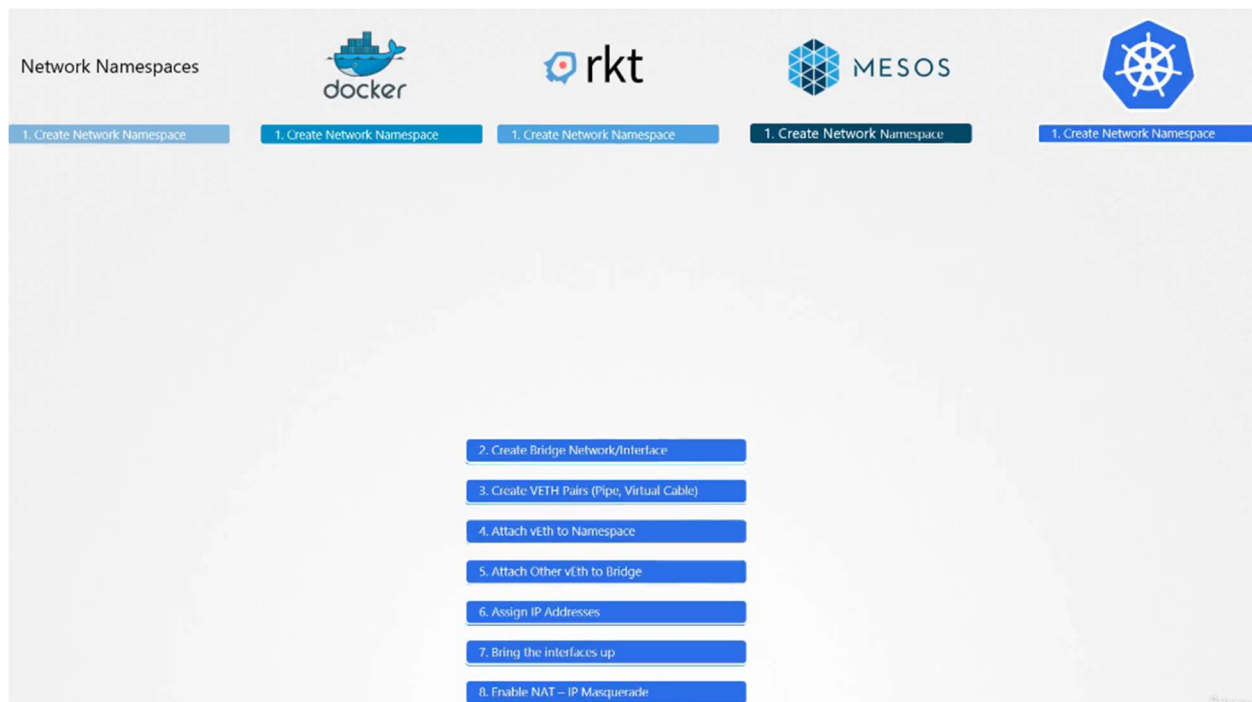


So far, we saw how network name spaces work, as in, how to create an isolated network namespace environment within our system, we saw how to connect multiple such name spaces through a bridge network, how to create virtual cables or pipes with virtual interfaces on either end, and then how to attach each end to end name space and the bridge with also how to assign IP and bring them up and finally, enable NAT or IP masquerade for external communication, et cetera. We then saw how Docker did it for its bridge networking option. It was pretty much the same way except that it uses different naming patterns. Well, other container solutions solve their networking challenges in kind of the same way, like Rocket or Mesos Containerizer or any other solutions that work with containers and requires to configure networking between them like Kubernetes.

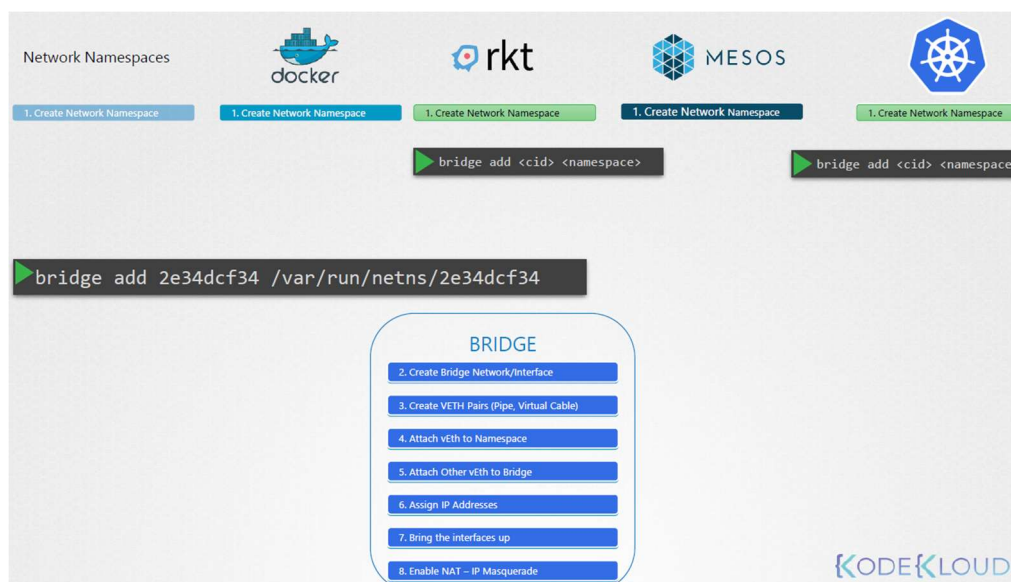


If we are all trying to solve the same networking challenges by researching and finally identifying a similar approach with our own little minor differences, why code and develop the same solution multiple times? Why not just create a single standard approach that everyone can follow? So we take all of these ideas from the different solutions and move all the networking portions of it into a single program or code.



And since this is for the bridge network, we call it Bridge. So we created a program or script that performs all the required tasks to get the container attached to a bridge network.

For example, you could run this program using its name, Bridge and specify that you want to add this container to a particular network name space. The Bridge program takes care of the rest so that the container runtime environments are relieved of those tasks. For example, when Rocket or Kubernetes creates a new container, they call the Bridge Program and pass the container ID and namespace to get networking configured for that container.



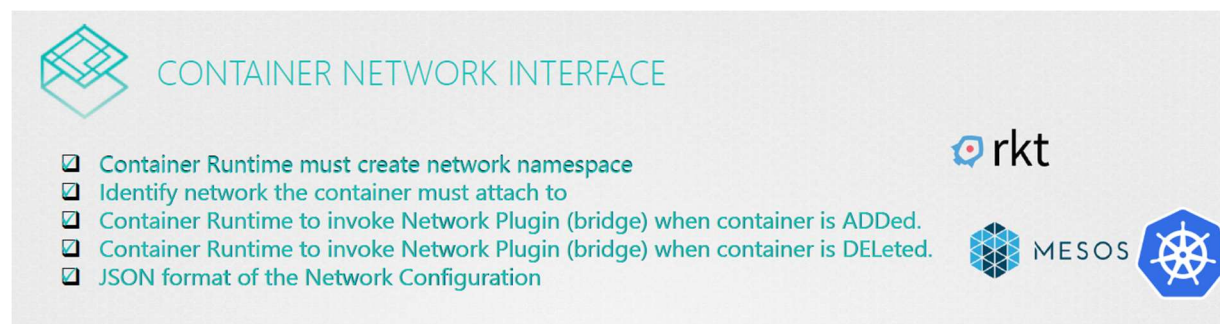
So what if you wanted to create such a program for yourself, maybe for a new networking type? If you were doing so, what arguments and commands should it support? How do you make sure the program you create will work currently with these run times? How do you know that container run times like Kubernetes or Rocket will invoke your program correctly?

That's where we need some standards defined. A standard that defines how a program should look, how container run times will invoke them so that everyone can adhere to a single set of standards and develop solutions that work across run times.

That's where container network interface comes in. The CNI is a set of standards that define how programs should be developed to solve networking challenges in a container runtime environments. The programs are referred to as plugins.

In this case, Bridge program that we have been referring to is a plugin for CNI. CNI defines how the plugin should be developed and how container run times should invoke them.

CNI defines a set of responsibilities for container run times and plugins. For container run times, CNI specifies that it is responsible for creating a network name space for each container. It should then identify the networks the container must attach to, container run time must then invoke the plugin when a container is created using the add command and also invoke the plugin when the container is deleted using the del command. It also specifies how to configure a network plugin on the container runtime environment using a JSON file.

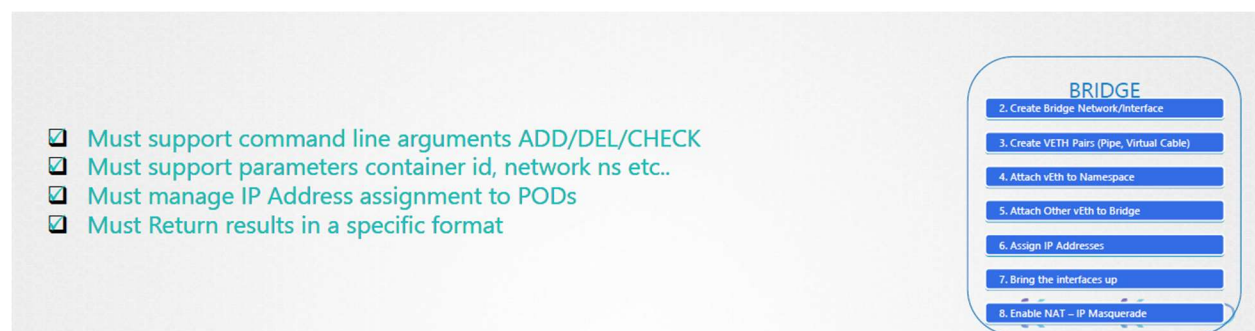


**CONTAINER NETWORK INTERFACE**

- ❑ Container Runtime must create network namespace
- ❑ Identify network the container must attach to
- ❑ Container Runtime to invoke Network Plugin (bridge) when container is ADDED.
- ❑ Container Runtime to invoke Network Plugin (bridge) when container is DELETED.
- ❑ JSON format of the Network Configuration

Logos: rkt, MESOS, and a ship's wheel icon.

On the plugin side, it defines that the plugin should support add, del and check command line arguments and that these should accept parameters like container and network namespace. The plug-in should take care of assigning IP addresses to the pods and any associated routes required for the containers to reach other containers in the network. At the end, the results should be specified in a particular format.



- ❑ Must support command line arguments ADD/DEL/CHECK
- ❑ Must support parameters container id, network ns etc..
- ❑ Must manage IP Address assignment to PODs
- ❑ Must Return results in a specific format

**BRIDGE**

2. Create Bridge Network/Interface
3. Create VETH Pairs (Pipe, Virtual Cable)
4. Attach vEth to Namespace
5. Attach Other vEth to Bridge
6. Assign IP Addresses
7. Bring the interfaces up
8. Enable NAT -- IP Masquerade

As long as the container run times and plugins adhere to these standards, they can all live together in harmony. Any run time should be able to work with any plugin.

CNI comes with a set of supported plugins already such as Bridge, VLAN, IP VLAN, MAC VLAN, one for Windows as well as IPAM plugins like Host Local and DHCP. There are other plugins available from third party organizations as well. Some examples are Weave, Flannel, Cilium, VMware NSX, Calico, Infoblox, et cetera.



All of these container run times implement CNI standards so any of them can work with any of these plugins. But there is one that is not in this list. Docker.

Docker does not implement CNI. Docker has its own set of standards known as CNM which stands for container network model which is another standard that aims at solving container networking challenges similar to CNI but with some differences.



Due to the differences, these plugins don't natively integrate with Docker, meaning you can't run a Docker container and specify the network plugin to use a CNI and specify one of these plugins. But that doesn't mean you can't use Docker with CNI at all. You just have to work around it yourself. For example, create a Docker container without any network configuration and then manually invoke the bridge plugin yourself. That is pretty much how Kubernetes does it. When Kubernetes creates Docker containers, it creates them on the none network. It then invokes the configured CNI plugins who take care of the rest of the configuration.



## CONTAINER NETWORK INTERFACE

```
❌ docker run --network=cni-bridge nginx
```

```
▶ docker run --network=none nginx
```

```
▶ bridge add 2e34dcf34 /var/run/netns/2e34dcf34
```



CONTAINER NETWORK MODEL (CNM)