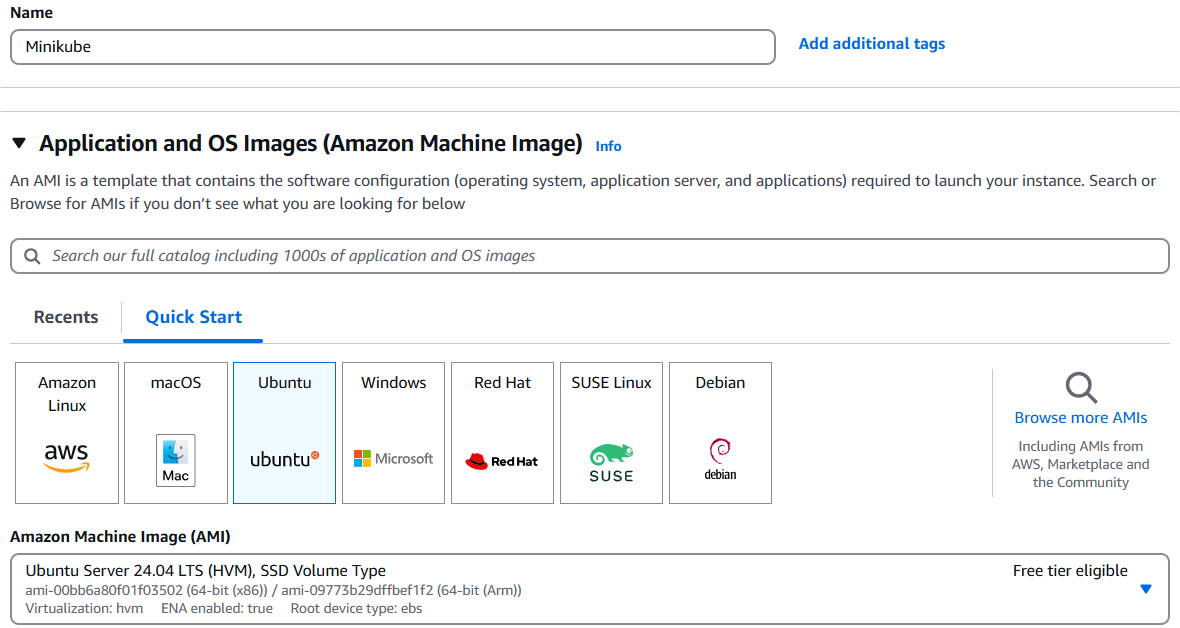
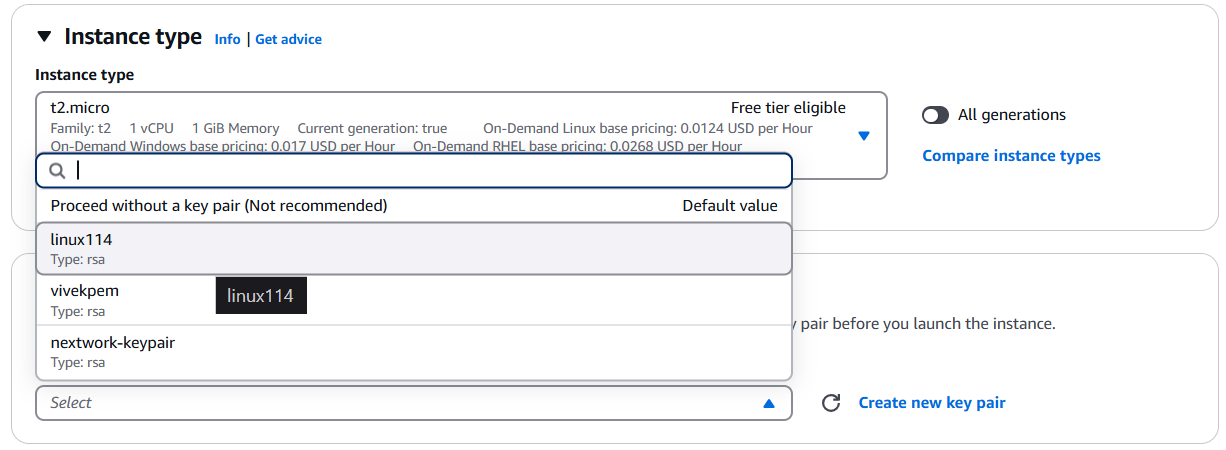
**Execution of Docker Compose Concepts and Networking Concepts.**

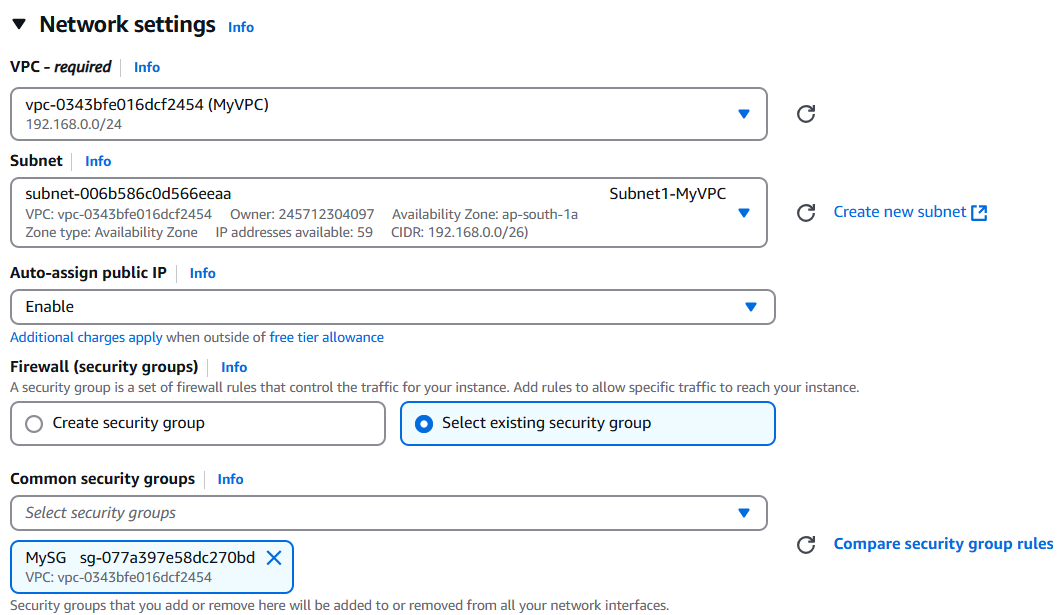
* Launch an EC2 Instance
  + **Enter the Name of the Instance**, eg: **DockerCompose**
  + Choose **Ubuntu Server 24.04 LTS (HVM)** under **Amazon Machine Image(AMI)**



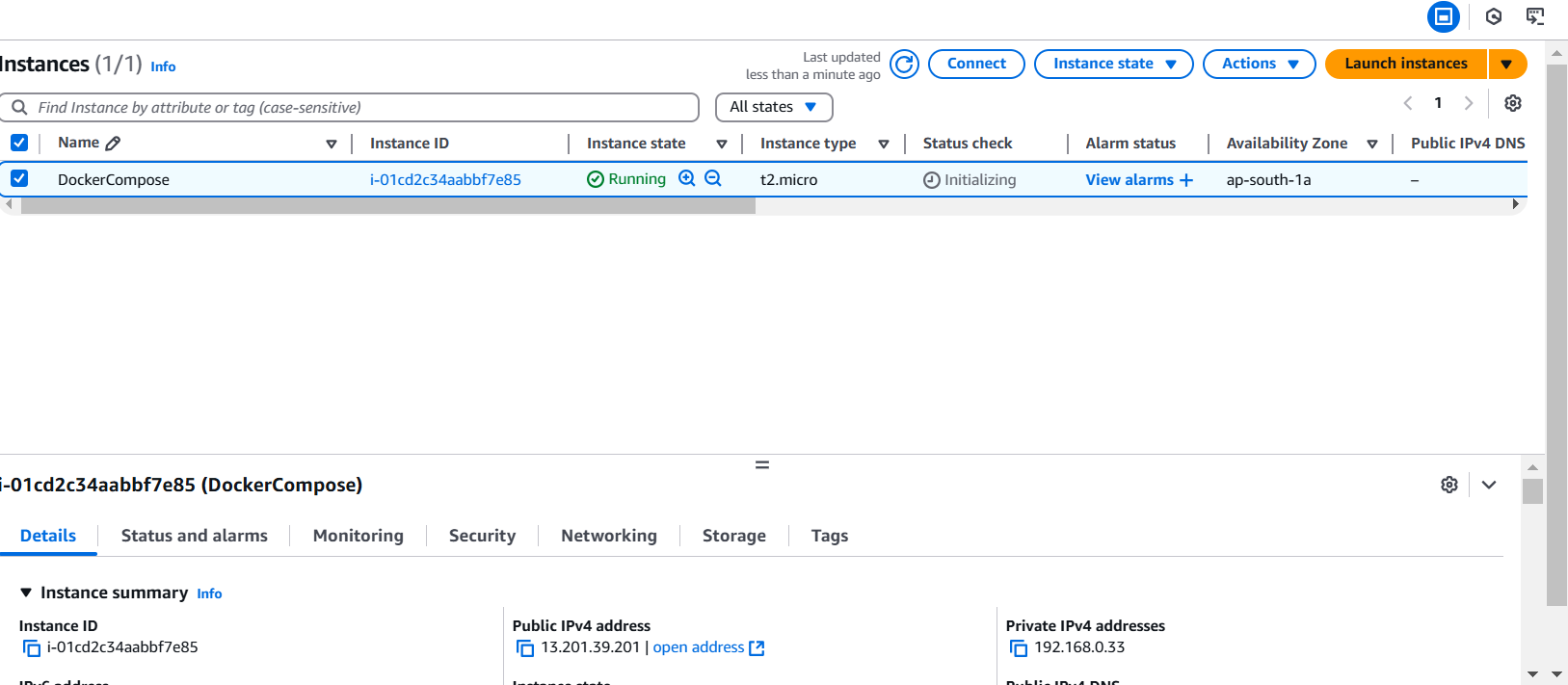
* + Choose **t2.micro** under **Instance type**.
  + Under **Key pair (login)**, give your key pair name eg:linux114 is my keypair.

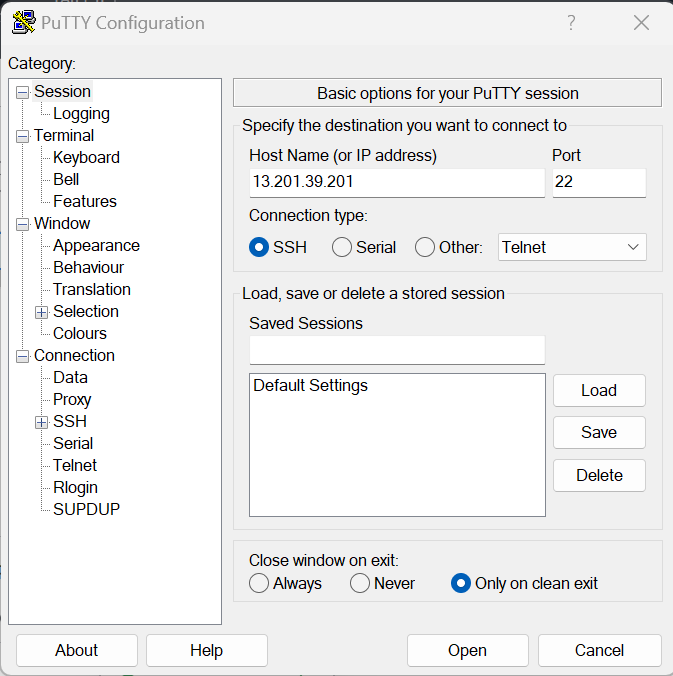


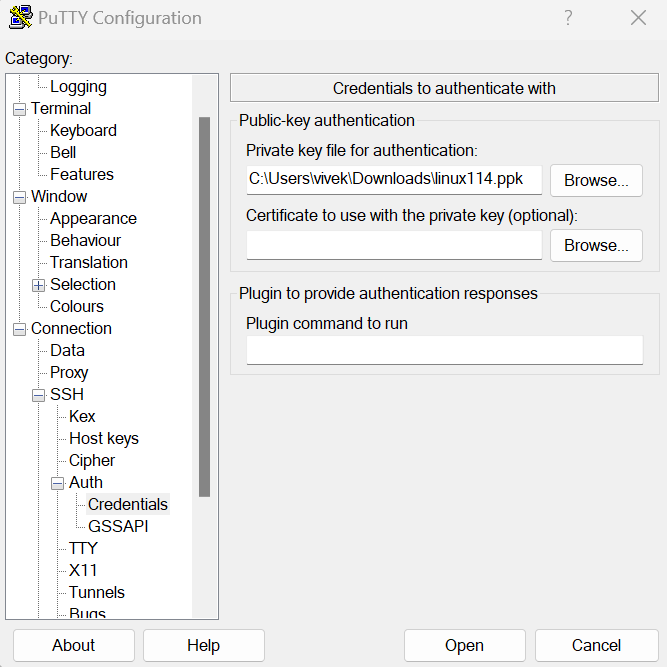
* + Back to our EC2 instance setup, head to the **Network settings** section and click **Edit.**
  + Give Your customised VPC and Public Subnet, **Enable** Auto Assign Public IP and Select your own customised Security Group.

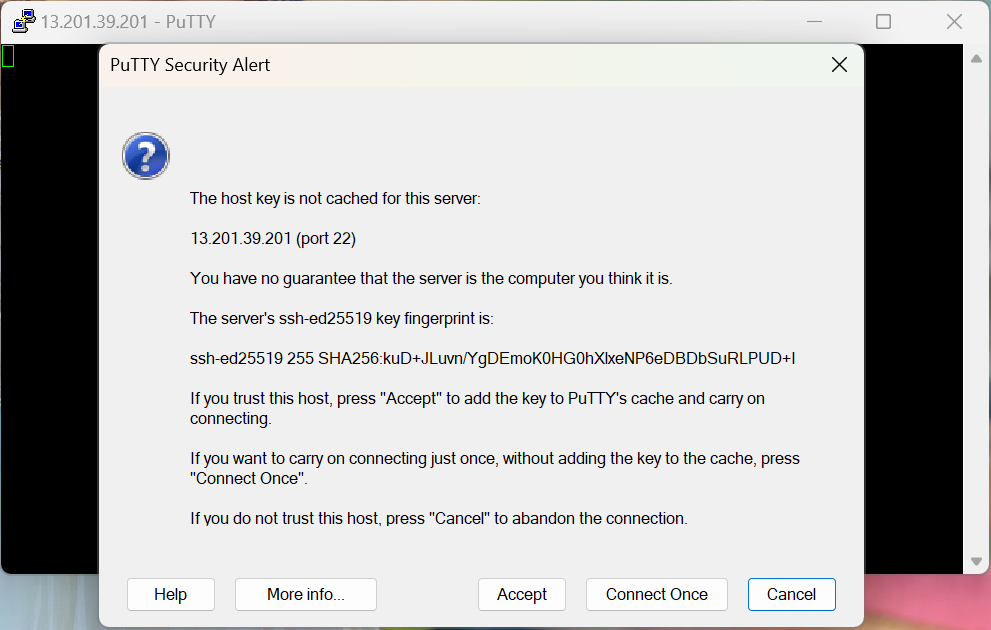


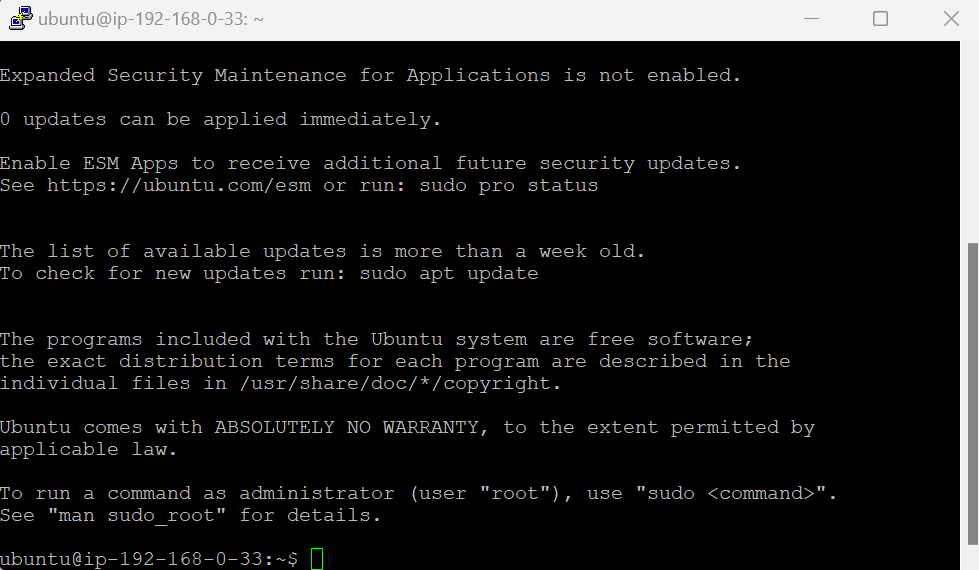
* + choose **Launch instance**
  + Connect to Instance via Putty (**note: login as : ubuntu**)





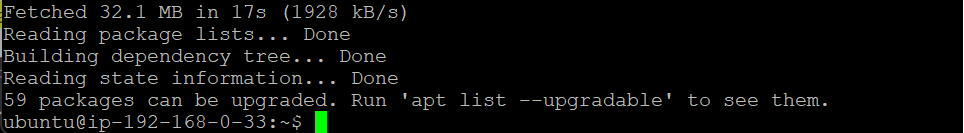




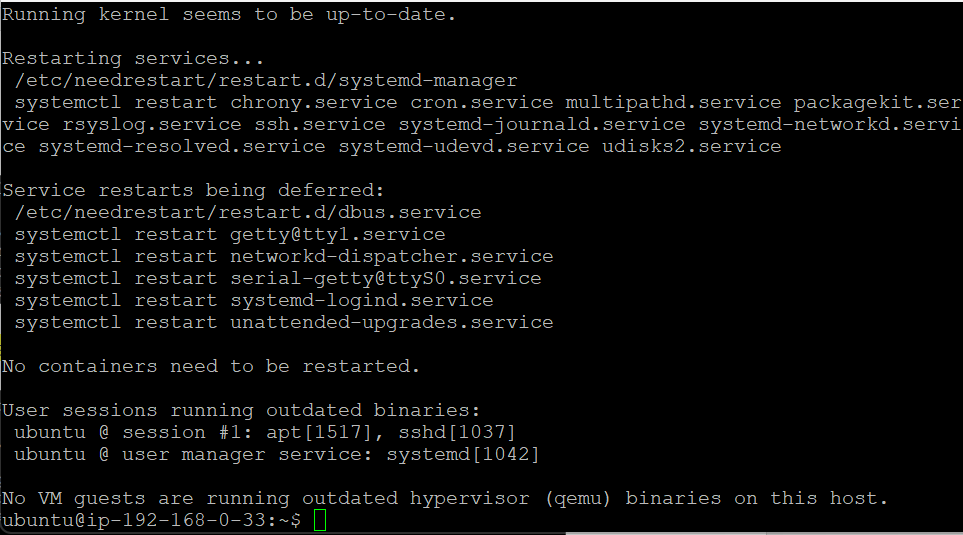


* + Now that we have connected to our EC2 Instance, lets install docker in the EC2.
  + The commands for installation of docker are as follows:

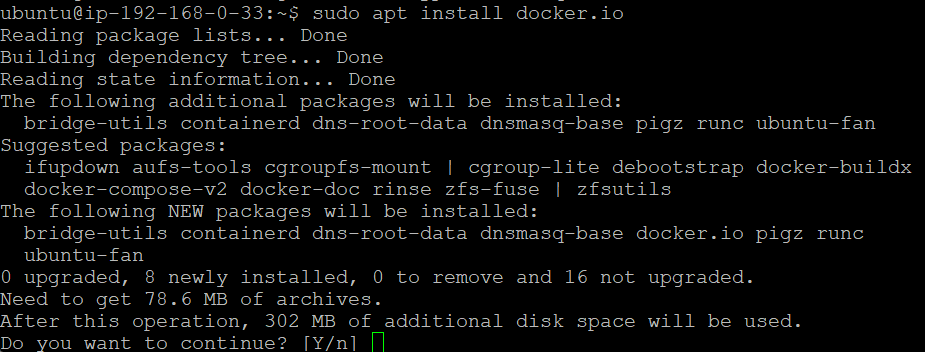
sudo apt update



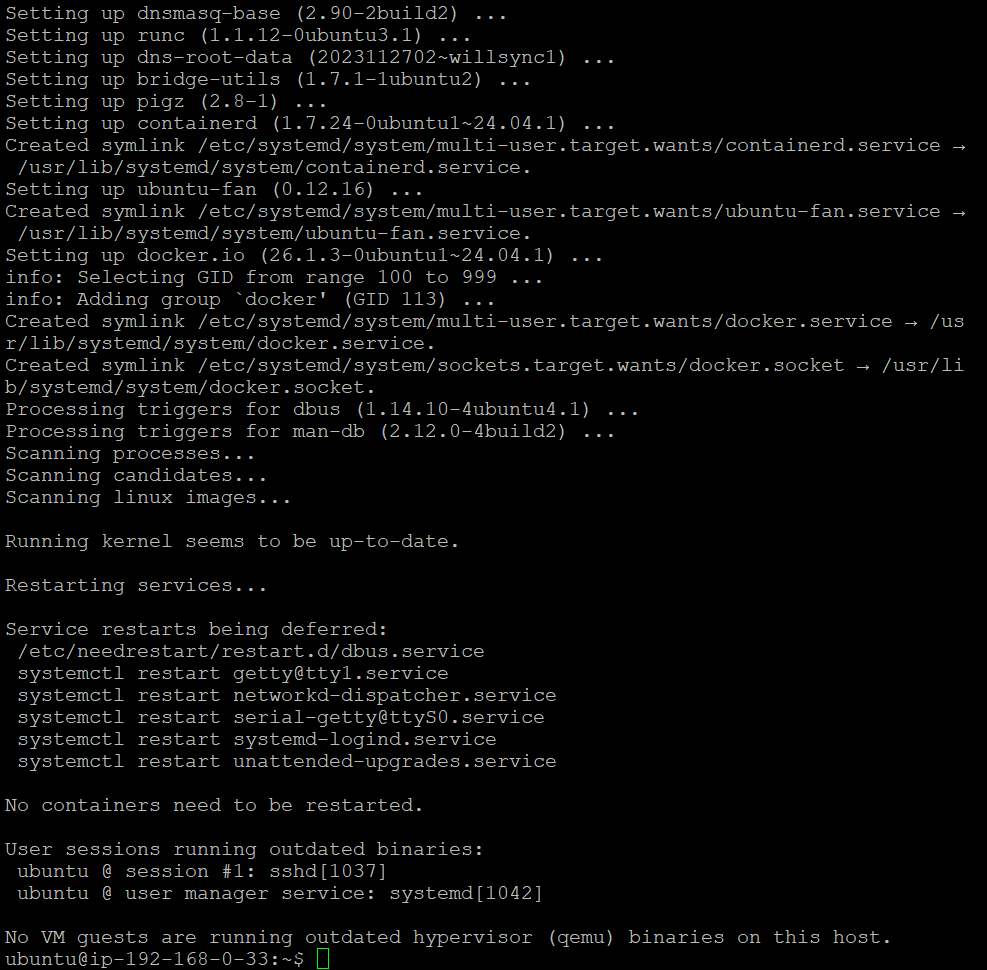
sudo apt upgrade -y



sudo apt install docker.io



Type Y



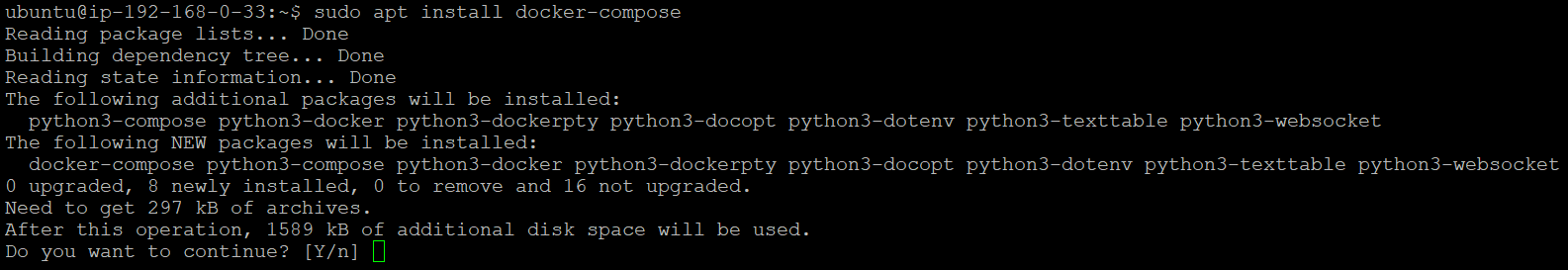
sudo chmod 666 /var/run/docker.sock



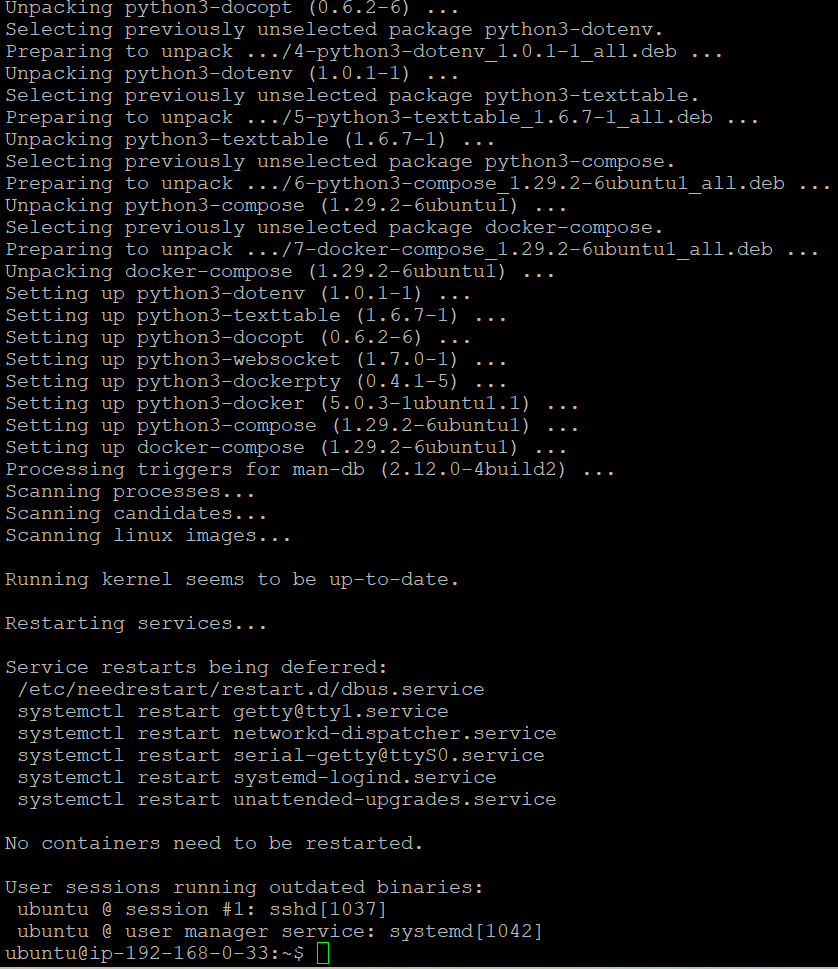
* + Run docker ps command to see the status.



* + Run sudo apt install docker-compose to install Docker compose.



Type Y

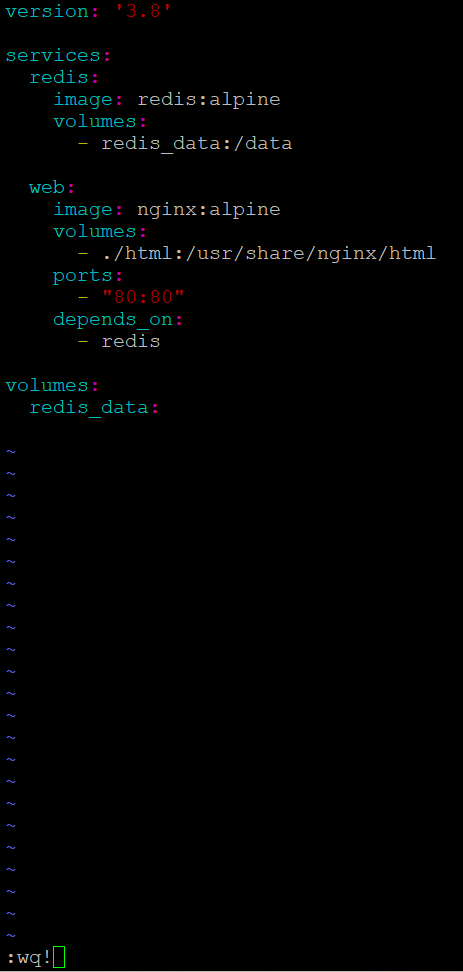


* + Create a folder named **compose1** by running the command mkdir compose1
  + Enter into the folder with the command cd compose1



* + Create a file named **docker-compose.yml** with edit access by running vim docker-compose.yml, then paste the available Docker Compose YAML script into the file as save the file with :wq! command





The explanation for the yaml script is as follows:

1. **version: '3.8'**

Specifies the version of the Docker Compose file format. In this case, it’s version 3.8, which allows access to features available in that version.

1. **services:**  
   Defines the individual containers (services) that will be part of this multi-container setup.
2. **redis:**  
   This is the name of the first service, which runs the Redis container.
3. **image: redis:alpine**

The Docker image to use for the Redis service. The alpine variant is a smaller, lightweight version of Redis built on Alpine Linux.

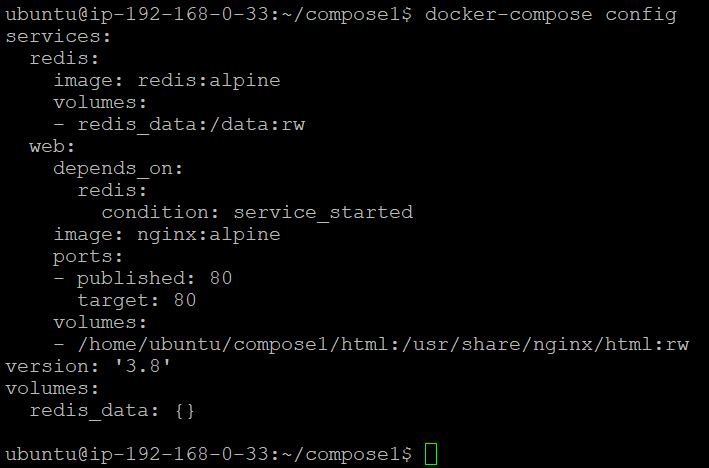
1. **volumes:**  
   Declares that the Redis data should be stored in a persistent volume. This mounts the redis\_data volume to the /data directory inside the Redis container.
2. **web:**  
   This is the second service, which runs the Nginx web server.
3. **image: nginx:alpine**

Specifies the Docker image to use for the Nginx web server. The alpine variant of Nginx is also lightweight and efficient.

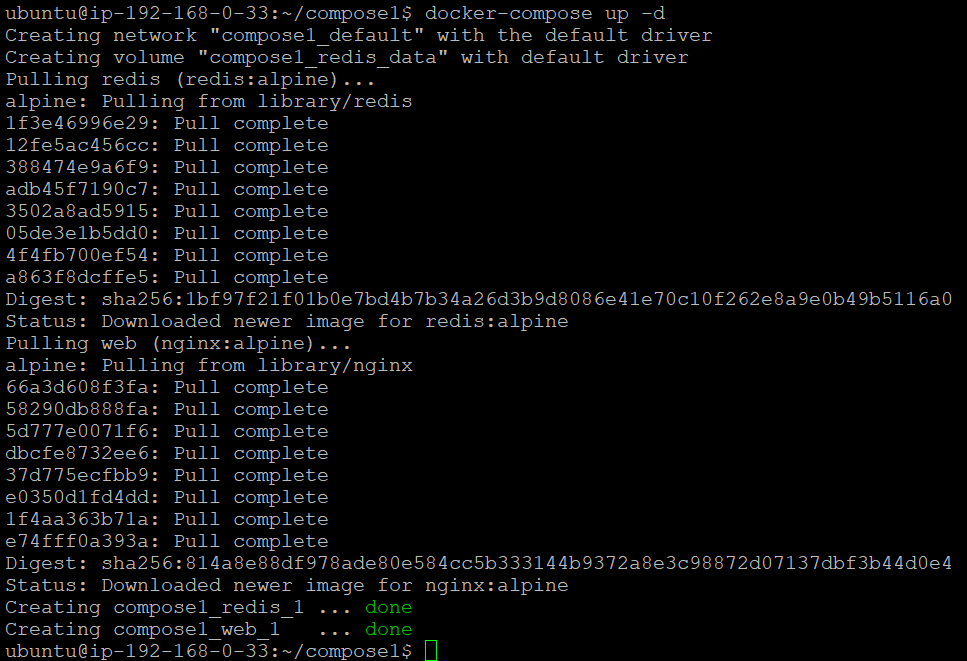
1. **volumes:**  
   Mounts the local ./html directory to /usr/share/nginx/html inside the container. This allows you to serve static files stored in the local html folder through Nginx.
2. **ports:**  
   Maps port 80 on the host machine to port 80 inside the container. This allows external access to the web server via the host’s IP on port 80.
3. **depends\_on:**  
   Ensures that the **web** service starts only after the **redis** service is up and running. This is useful if your web service depends on Redis.
4. **volumes:**  
   Declares a named volume redis\_data to persist Redis data. This ensures data isn't lost when the container is removed or restarted.

This configuration sets up a multi-container application with a **Redis** service for data persistence and an **Nginx** service to serve static files, with persistent storage for Redis and web content.

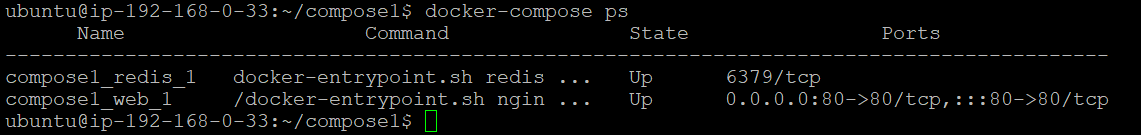
* + To validate your docker-compose.yml file, run the command docker-compose config



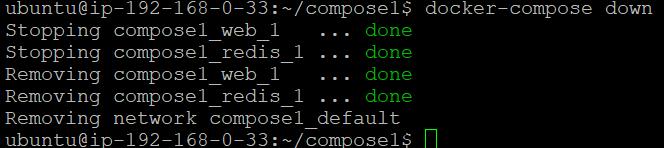
* + Run the command docker-compose up -d to build, create, and start the services defined in the docker-compose.yml file in detached mode, allowing them to run in the background while keeping the terminal available for other tasks.



* + Now execute the command docker-compose ps to display the status of the containers for the services defined in the docker-compose.yml file, including container names, service associations, current states, and exposed ports.

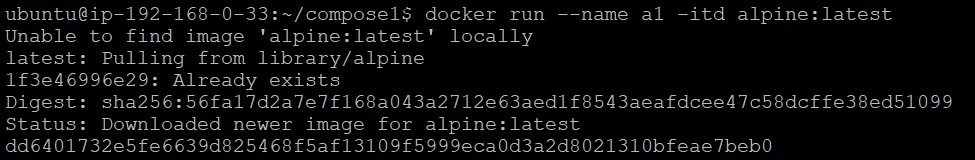


* + Now execute the command docker-compose down to stop and remove the containers, networks, and volumes defined in the docker-compose.yml file, effectively cleaning up the environment created by Docker Compose.

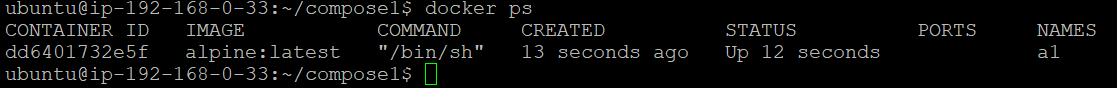


**Networking Concepts:**

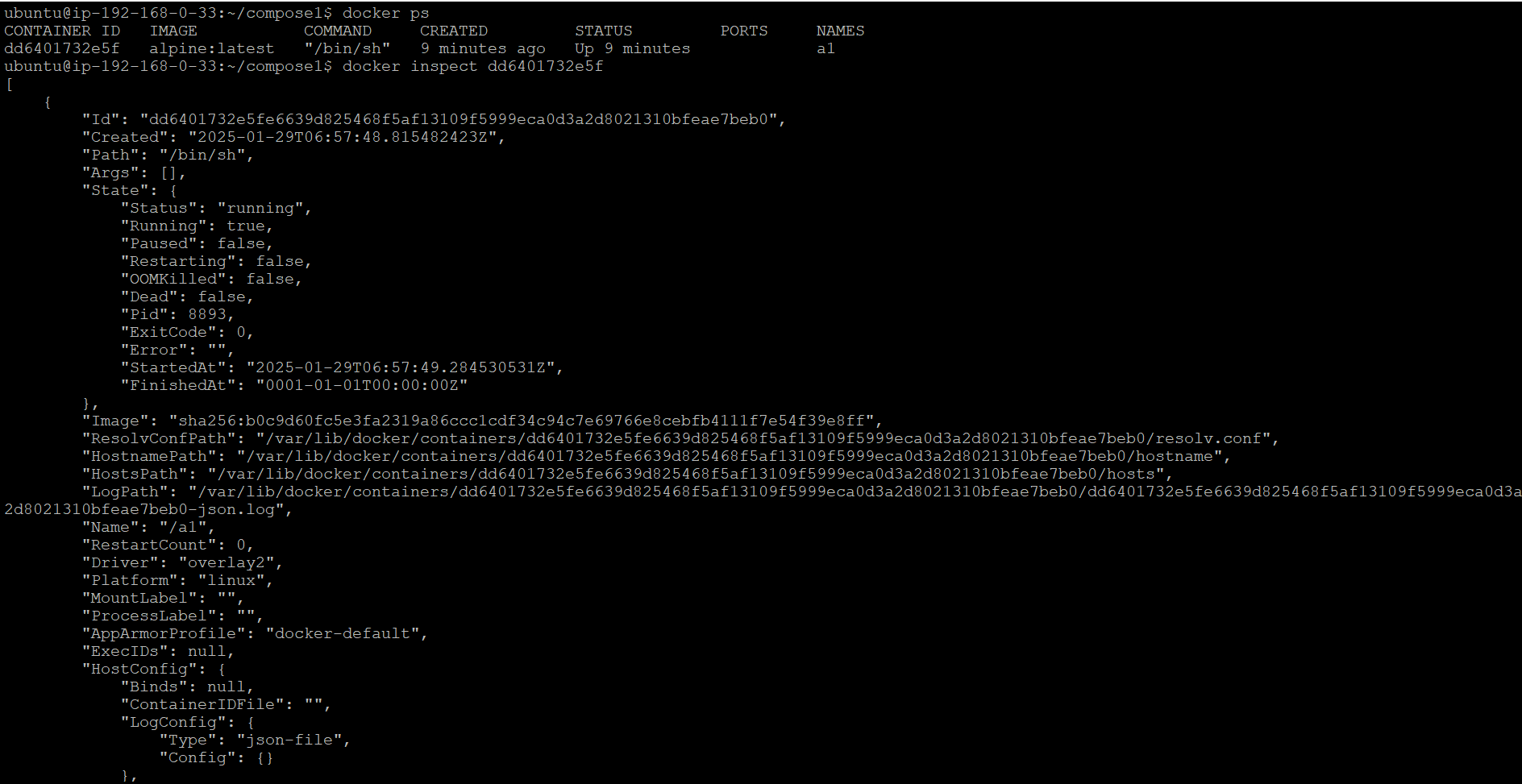
* + Execute the command docker run --name a1 -itd alpine:latest, this will create and run a new container named **a1** using the **Alpine Linux** image (alpine:latest). The -it flags allow for interaction with the container through an interactive terminal, while the -d flag runs the container in **detached mode**, meaning it will run in the background without taking up the terminal. This setup initializes a lightweight Alpine Linux container, which is useful for running simple tasks or creating a testing environment.

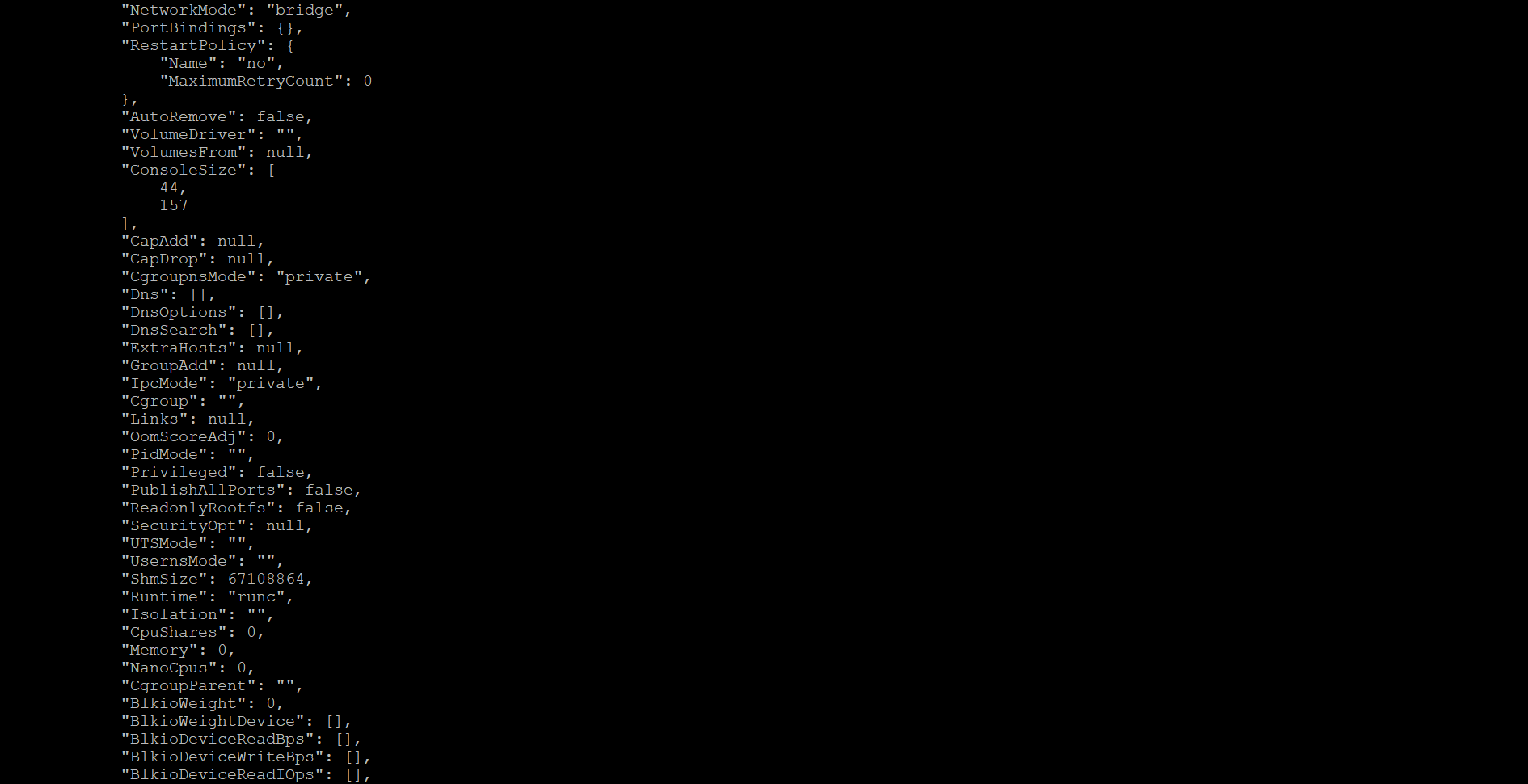


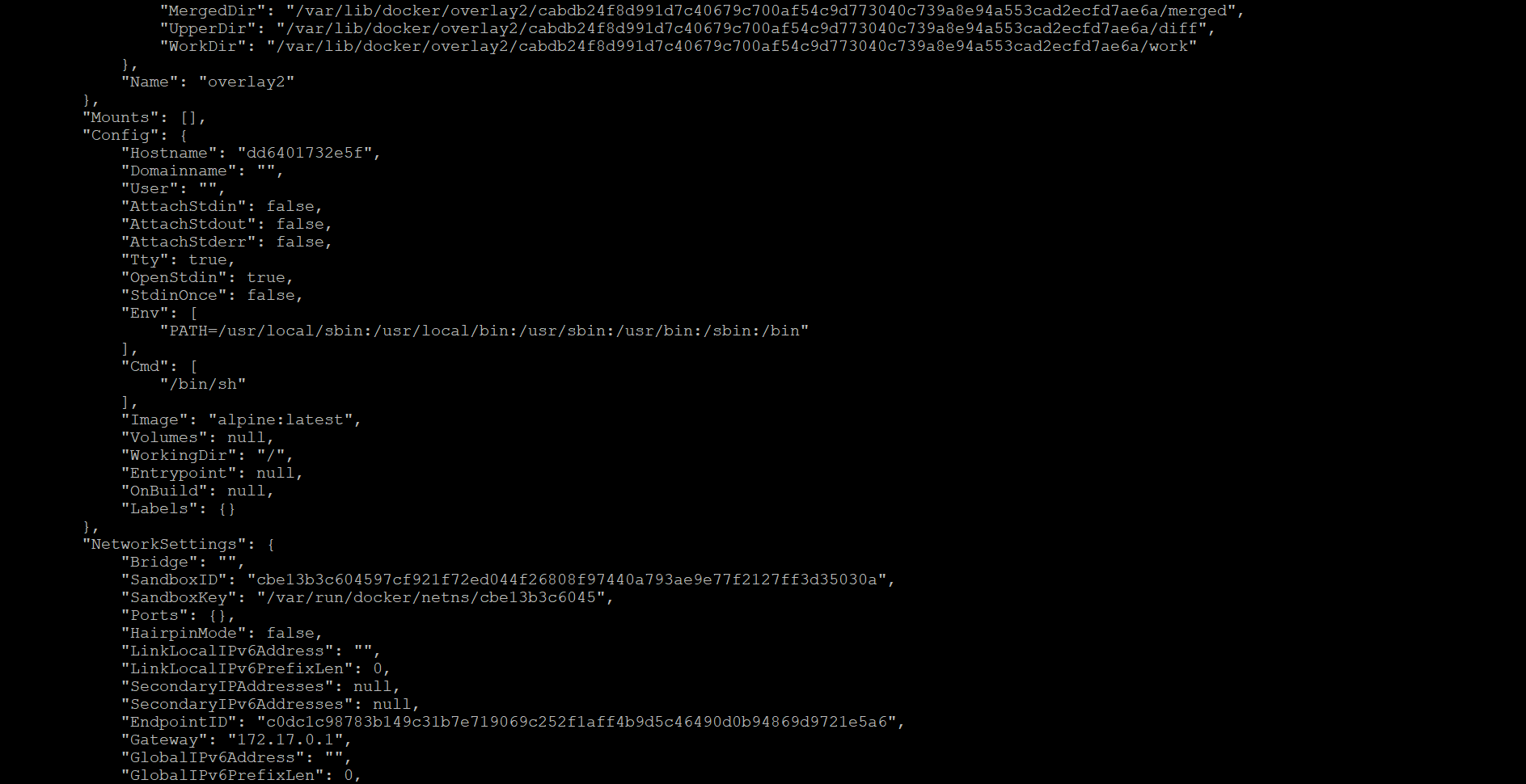
* + Now execute the command docker ps to check the newly created container from the previous command and see its current status.

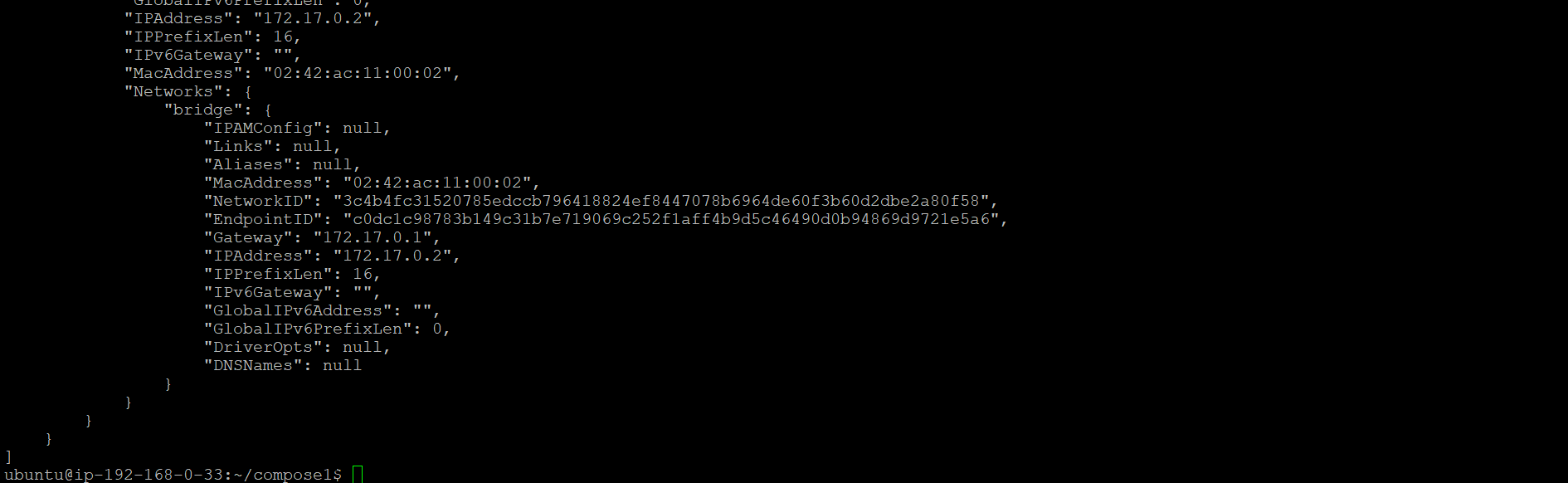


* + Now run the command docker inspect <containerID> in my case docker inspect dd6401732e5f followed by the container ID of the **a1** container to obtain detailed information about the container. You can obtain the container ID by running the docker ps command.









We can observe the container’s network settings from the docker inspect command we used to obtain detailed information about the container as follows:

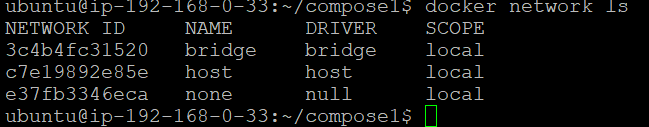
* **IPAddress**: 172.17.0.2
* **Gateway**: 172.17.0.1
* **Network Settings**:
  + **Network Mode**: bridge
  + **Network ID**:

3c4b4fc31520785edccb796418824ef8447078b6964de60f3b60d2dbe2a80f58

* + **Endpoint ID**:

c0dc1c98783b149c31b7e719069c252f1aff4b9d5c46490d0b94869d9721e5a6

* + **Link Local IPv6 Address**: N/A
  + **Link Local IPv6 Prefix Length**: 0
  + **Secondary IP Addresses**: N/A
  + **Secondary IPv6 Addresses**: N/A
  + **Global IPv6 Address**: N/A
  + **Global IPv6 Prefix Length**: 0
  + **DNS Names**: N/A
* From the above data, we can observe that the **a1** container has a network mode set to **bridge**.
* We can obtain the available networks in Docker by running the command docker network ls.

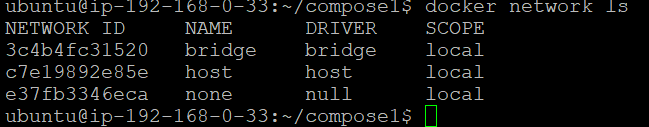


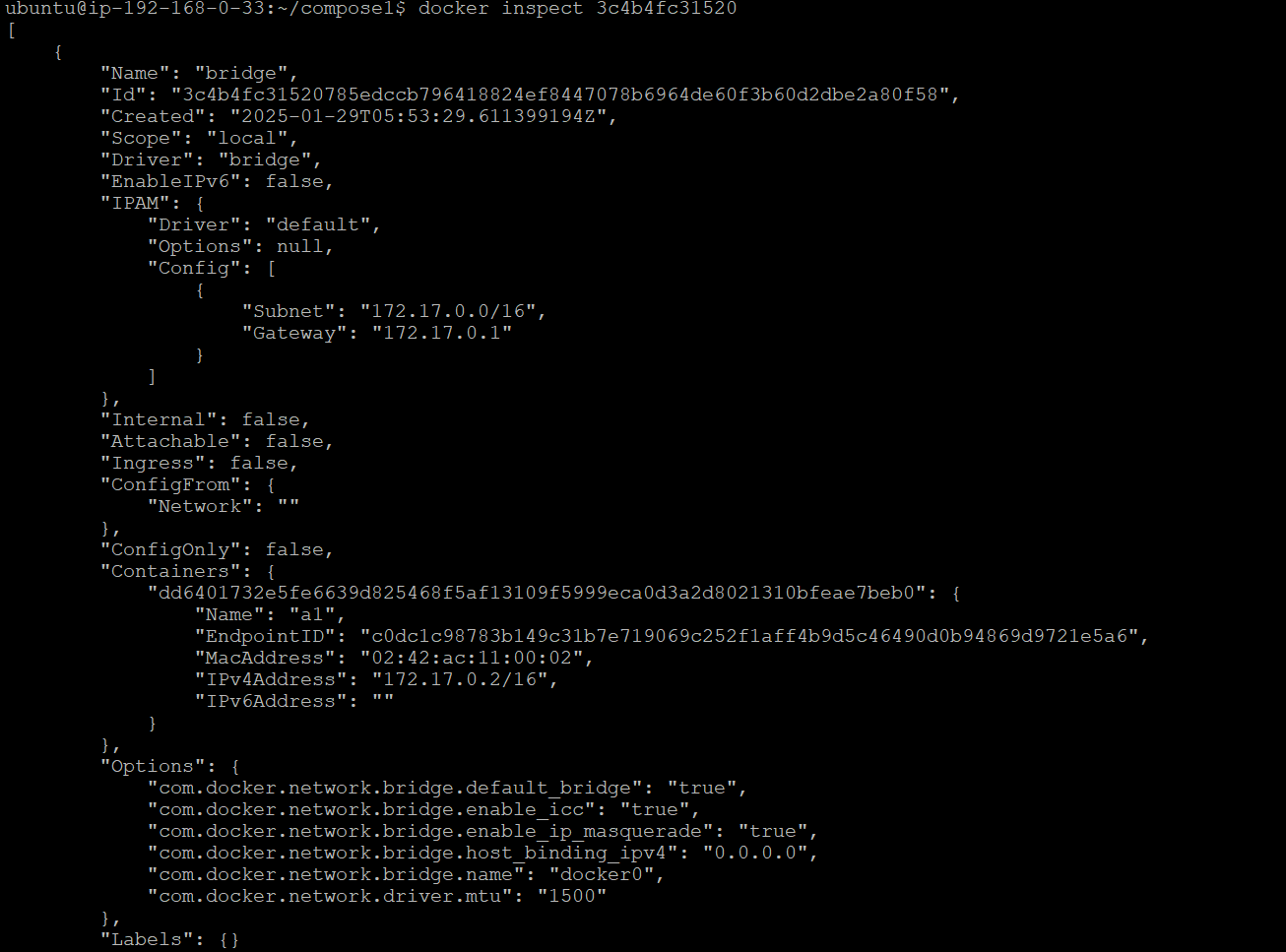
In Docker, there are three main types of networks: **bridge**, **host**, and **none**.

* The **bridge** network is the default network for containers when none is specified. It creates a private internal network on your host system, and containers on this network can communicate with each other.
* The **host** network allows containers to share the host's network stack, which means the container uses the host's IP address and can directly communicate with the external network.
* The **none** network isolates the container from any network, meaning it cannot communicate with other containers or the outside world.

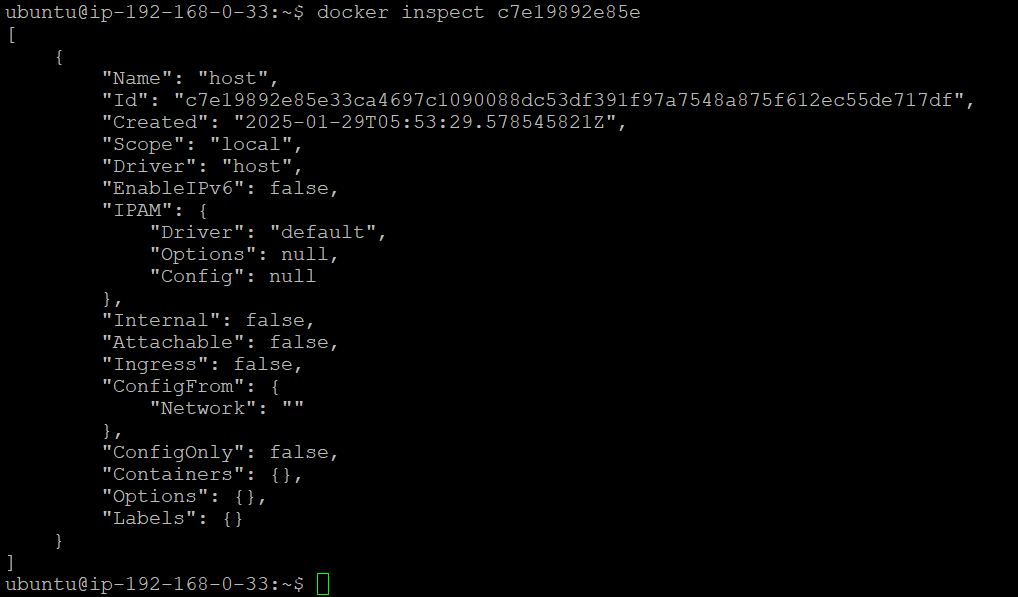
By default, Docker containers are connected to the **bridge** network unless otherwise specified.

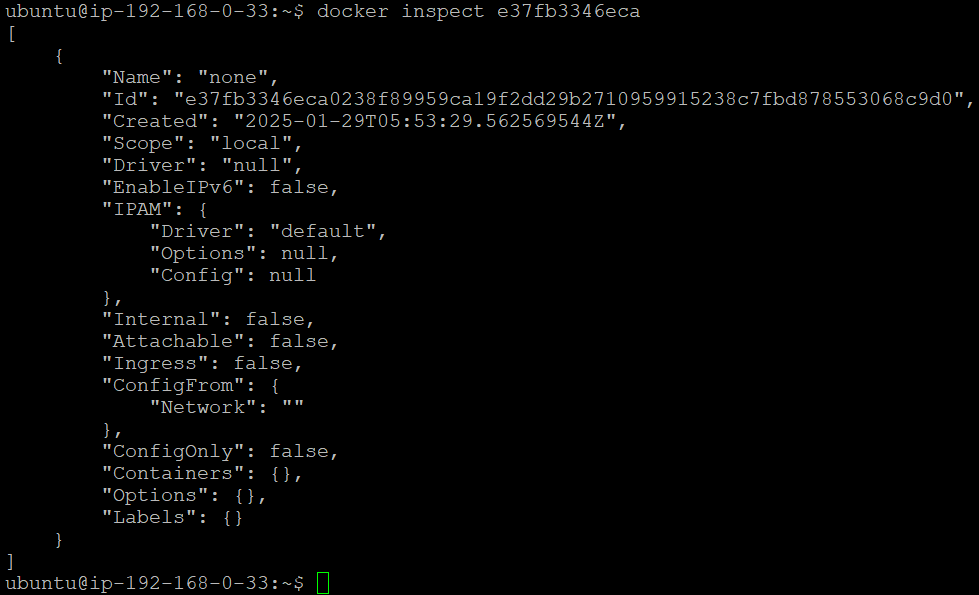
* We can inspect the networks using the command docker inspect <bridge network-id>.  
  In my case scenario, to inspect the details of each network, I can use the docker inspect command followed by the network ID. For instance, **to inspect the Bridge network**, I would run docker inspect 3c4b4fc31520, **for the Host network**, I would run docker inspect c7e19892e85e, and **for the None network**, I would run docker inspect e37fb3346eca. These commands will provide detailed information about the respective networks, including network settings, connected containers, and other relevant configurations.



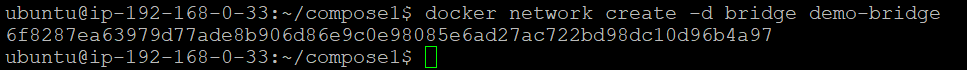


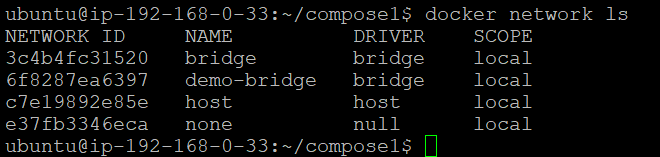
By default, any container created will be added to the Bridge network. In this case, we can see from the output that the container "a1" we created is listed under the "Containers" section of the Bridge network. This indicates that when containers are created, they are allocated to the Bridge network by default. The container "a1" has an assigned IP address of 172.17.0.2/16 within the Bridge network, confirming that it is part of this network and able to communicate with other containers on the same network.





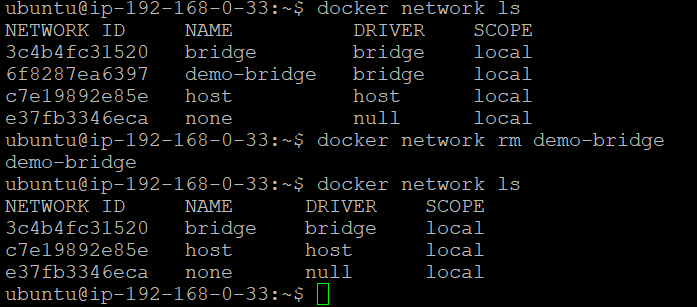
* Containers created on the same bridge network can communicate with each other using their IP addresses, which simplifies the process of setting up multi-container applications. To create a new network of the desired type, you can use the command docker network create -d bridge <networkname>. For example, to create a network named **demo-bridge**, the command would be docker network create -d bridge demo-bridge.



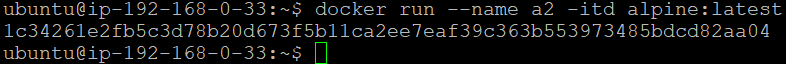


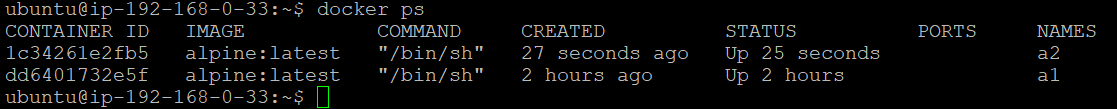
A new bridge network demo-bridge has been successfully created.

* To delete a network in Docker, use the command docker network rm <network\_name\_or\_id>, for example, docker network rm demo-bridge to **remove the demo-bridge network.**

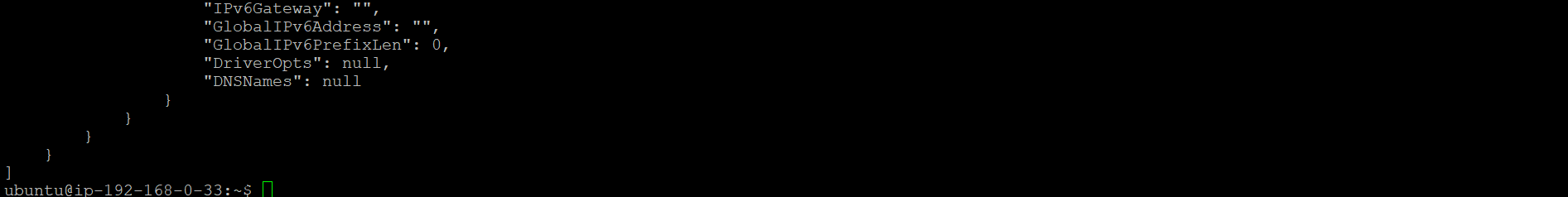
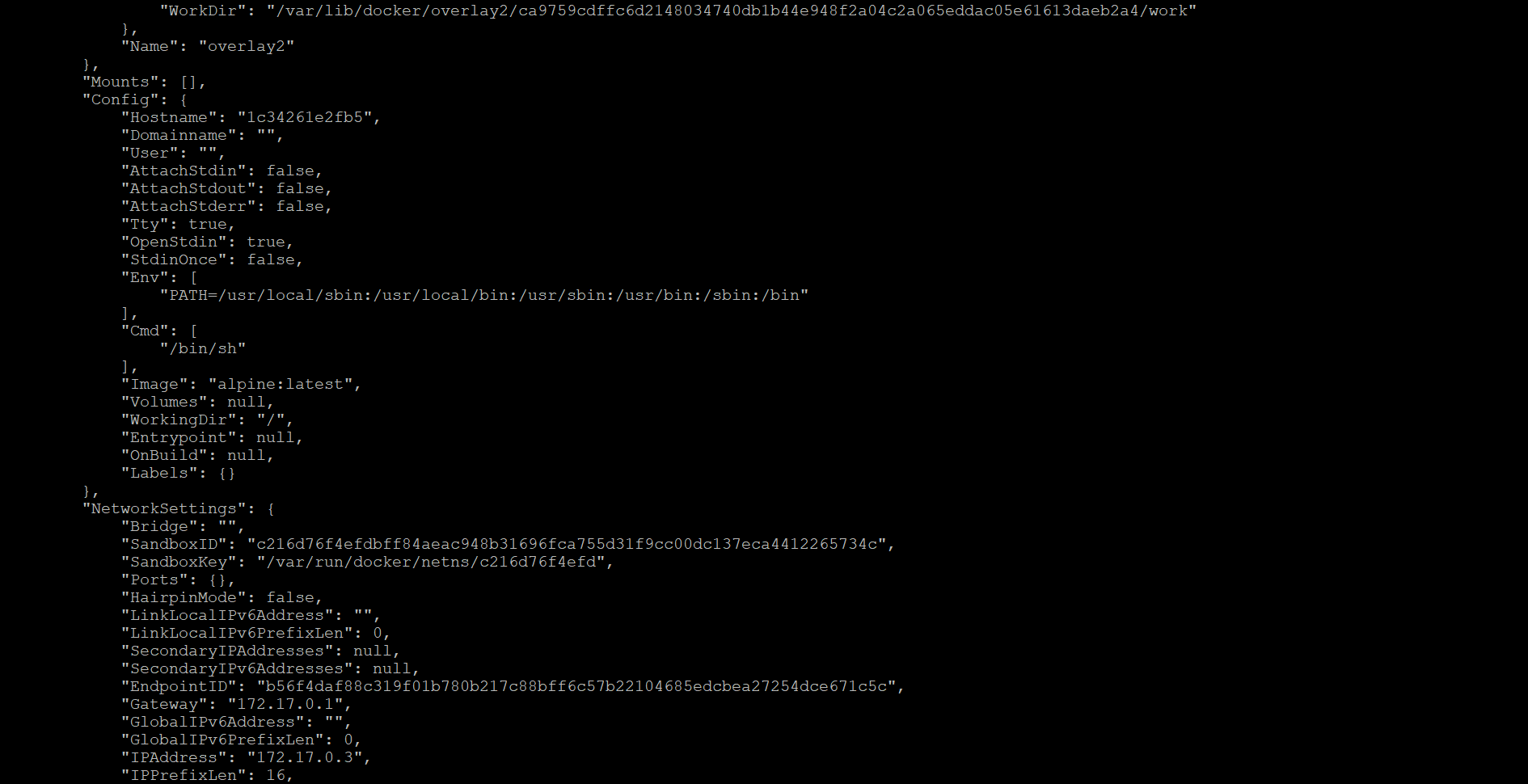
****

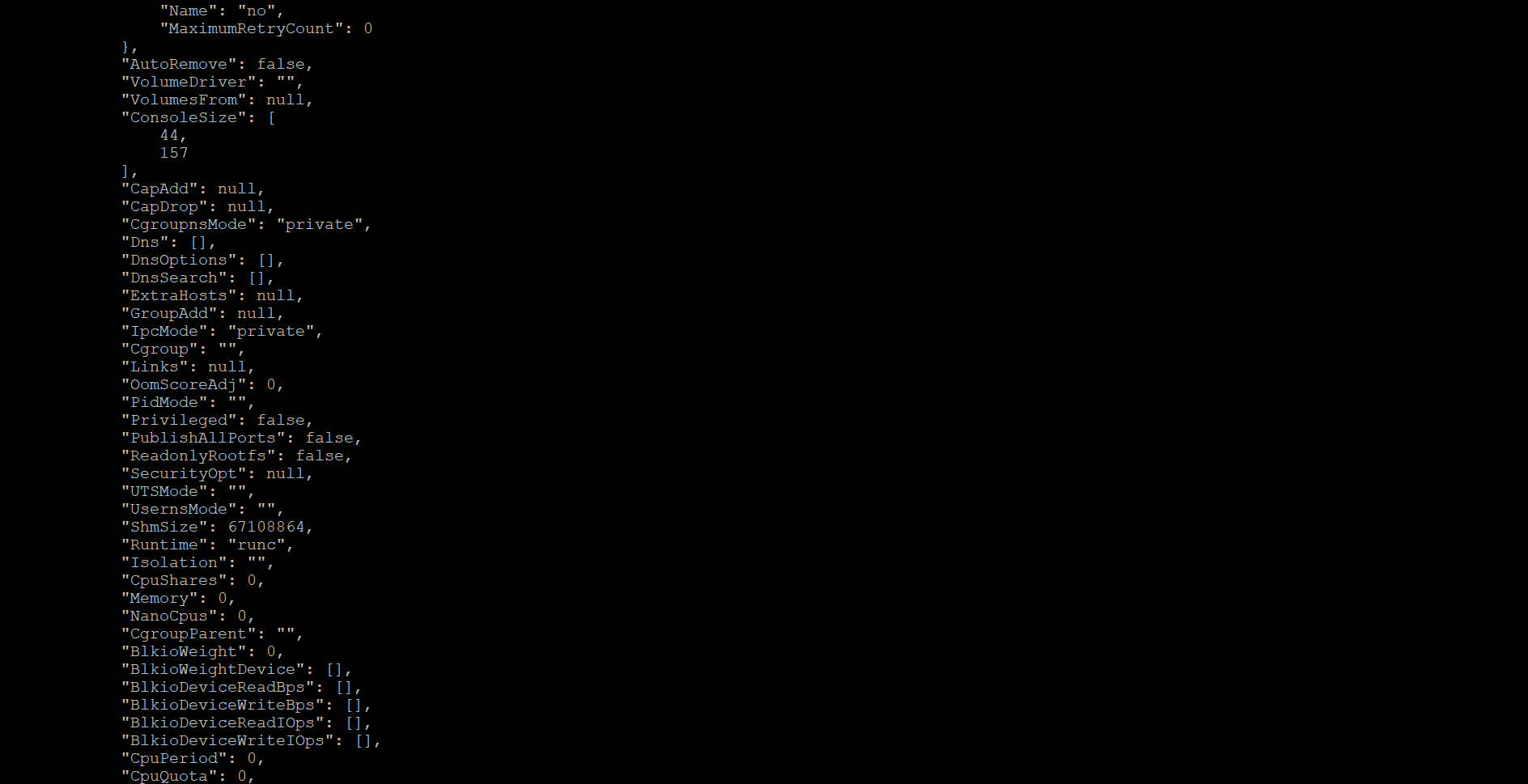
* Now, let's **create another container named a2** by executing the command docker run --name a2 -itd alpine:latest. After that, check the networks of both containers by running the docker inspect commands using their container IDs (which you can obtain from docker ps), like so: docker inspect <container\_id\_of\_a1> and docker inspect <container\_id\_of\_a2>.

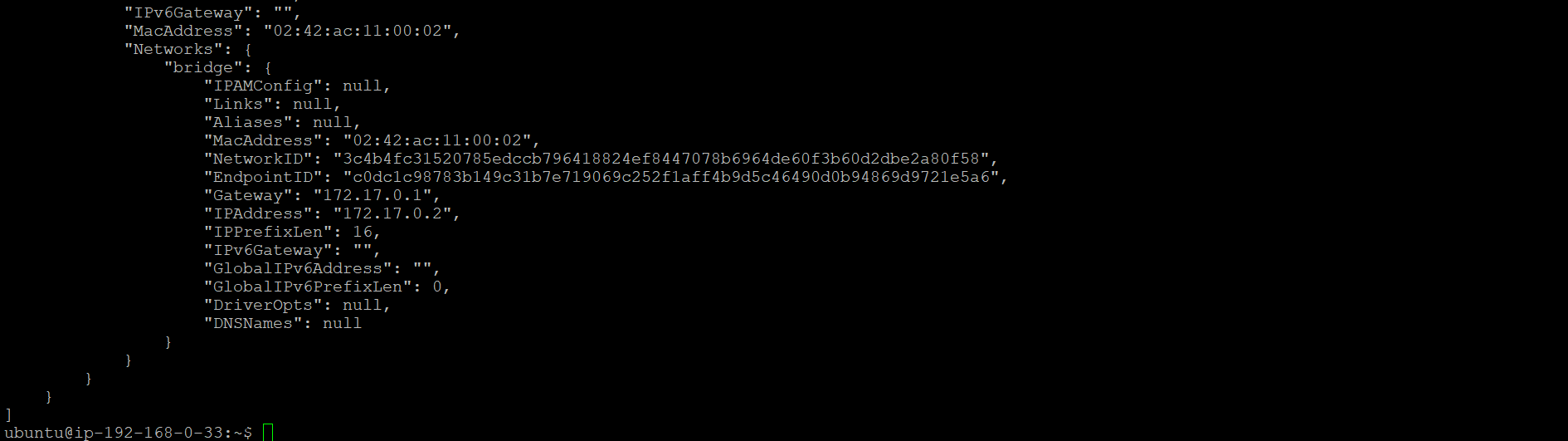
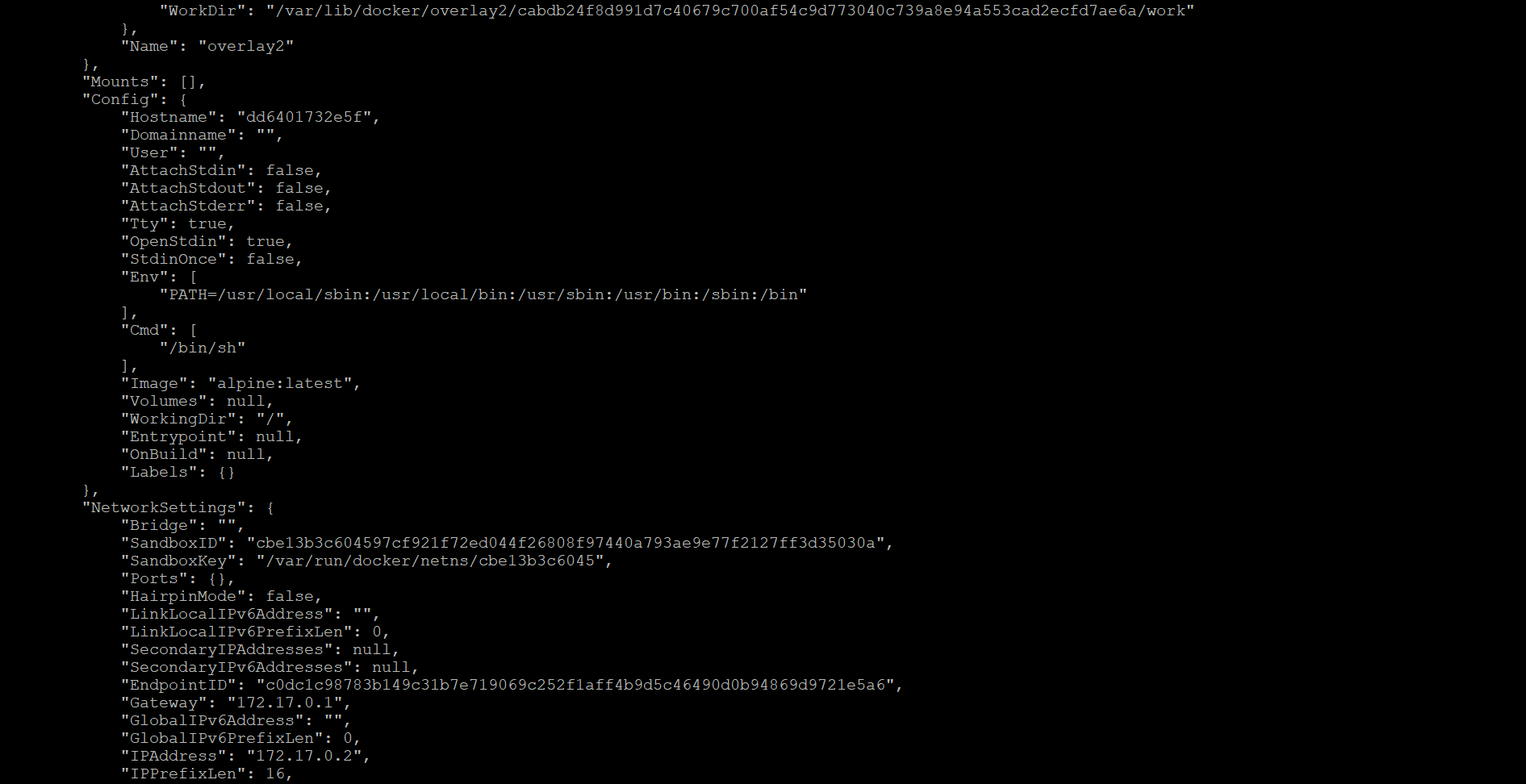
****

****

****

****

****

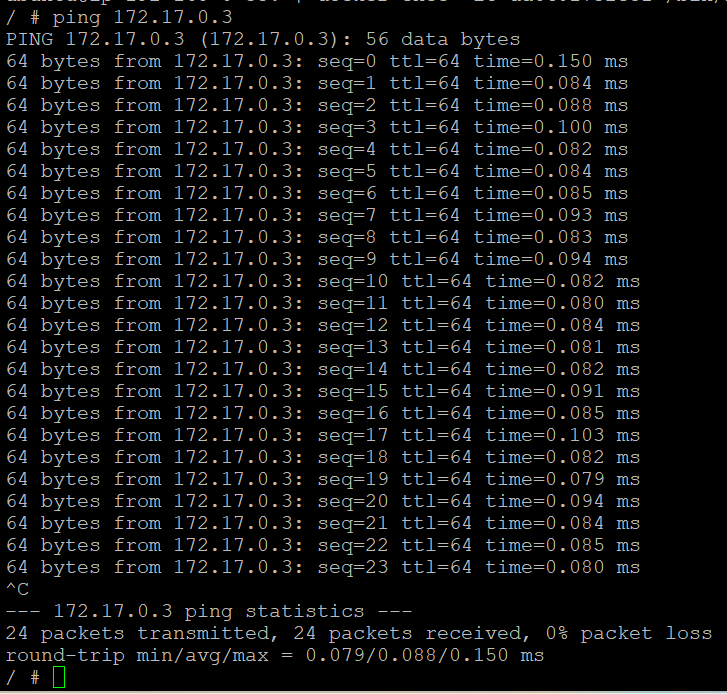
****

**Upon inspecting both containers, we find that both are present inside the Bridge network by default.**

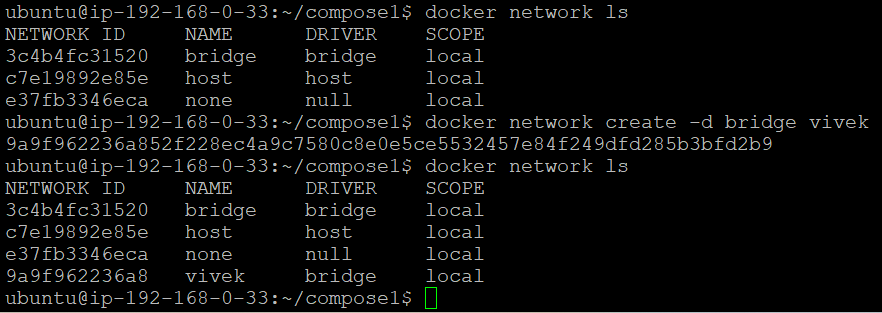
Upon inspecting both containers, we observed that the IP addresses of the containers are as follows: for **container a1**, **the IP address is 172.17.0.2**, and for **container a2, the IP** **address is 172.17.0.3.**

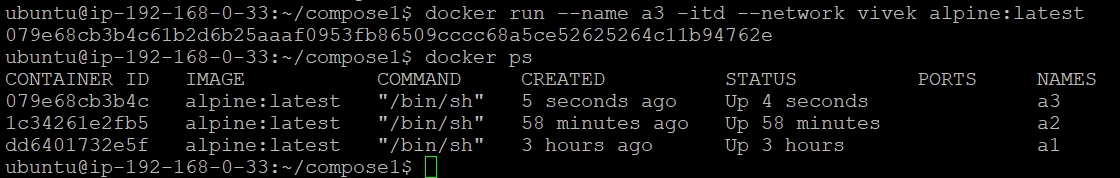
* Now, **enter** the **a1 container by executing** the command docker exec -it <container\_id\_of\_a1> /bin/sh. **This command allows you to open an interactive shell (/bin/sh) inside the a1 container**, enabling you to run commands directly within the container's environment.
* Once **inside the a1 container**, try to **ping the IP address of container a2** by using the ping 172.17.0.3 command. **Since both containers are on the same bridge network**, **the a1 container will be able to reach the a2 container**, and **the ping request will succeed, confirming the network connectivity between the two containers**.

****

****

* Now, let's **create a new bridge network called vivek** by executing the command docker network create -d bridge vivek. **Once the network is created**, we will **launch a new container a3** with the following command: docker run --name a3 -itd --network vivek alpine:latest. This ensures that the container a3 is connected to the newly created vivek network.

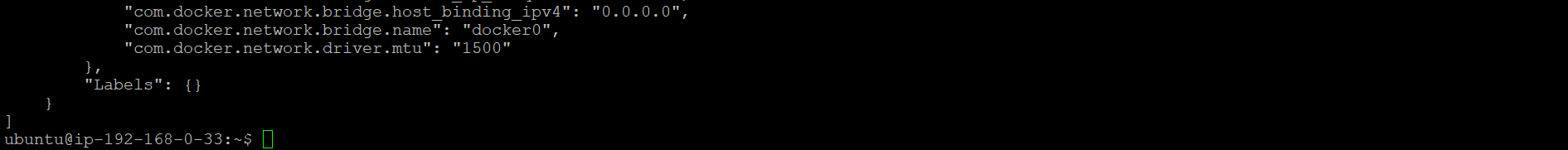




* Now, let's inspect both the bridge and vivek networks.

To inspect the bridge and vivek networks using their network IDs, first, you can list all available networks using the command docker network ls to get the network IDs. Once you have the network IDs, you can inspect the bridge network with the command docker inspect 3c4b4fc31520 and the vivek network with the command docker inspect 9a9f962236a8. This will provide detailed information about each network.



Upon inspection, we will observe that the bridge network contains containers a1 and a2, while the vivek network contains the newly created container a3.

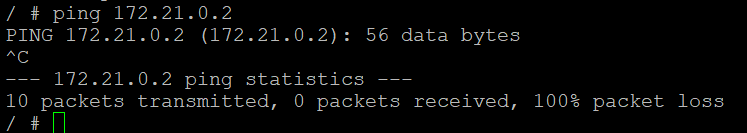
Here's a table displaying the IP addresses of the containers in their respective networks:

| **Container** | **Network** | **IP Address** |
| --- | --- | --- |
| a1 | bridge | 172.17.0.2 |
| a2 | 172.17.0.3 |
| a3 | vivek | 172.21.0.2 |

This table shows that containers a1 and a2 are in the bridge network with IPs 172.17.0.2 and 172.17.0.3, respectively, while container a3 is in the vivek network with the IP 172.21.0.2.

* Now, let's enter the shell of the a1 container and attempt to ping the IP address of the a3 container. First, execute the command docker exec -it dd6401732e5f /bin/sh to access the shell of the a1 container. Once inside the shell, run ping 172.21.0.2 to send a ping request to the IP address of the a3 container. Since a1 and a3 are in different networks (bridge and vivek), the ping request will likely fail because containers in separate networks are isolated from each other unless specific routing or network configurations are in place to allow communication between them.

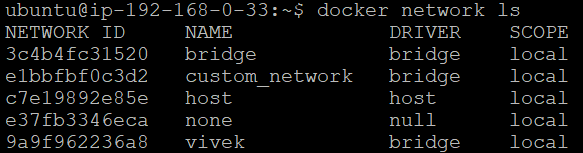




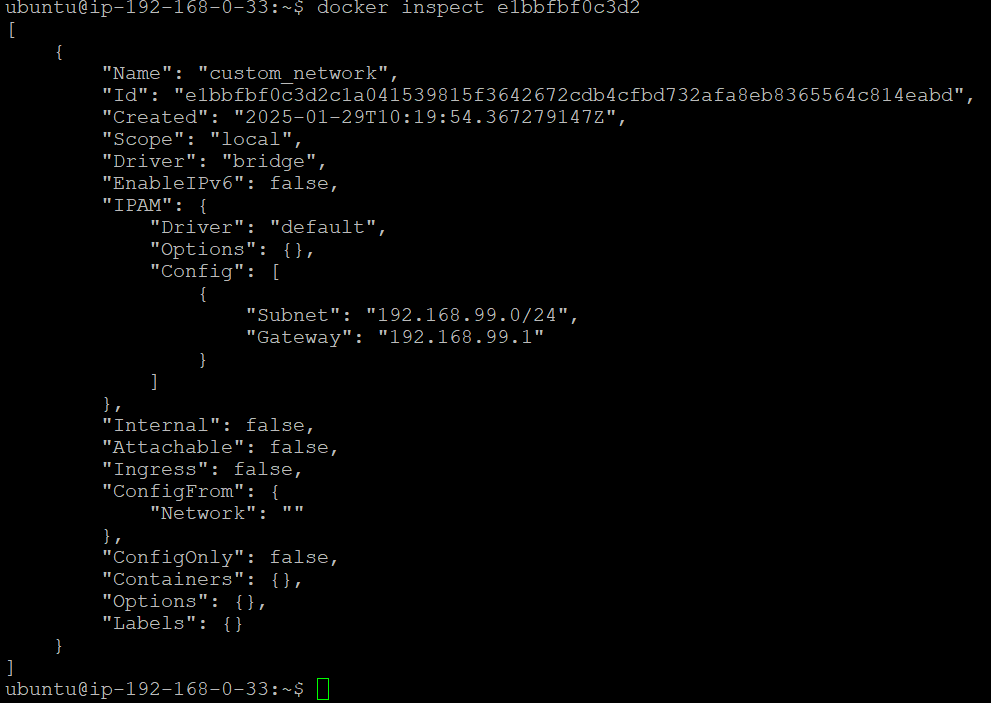
* Now, let's create a new Docker network with a custom subnet and gateway. To do this, execute the following command: docker network create --subnet=192.168.99.0/24 --gateway=192.168.99.1 custom\_network

This command **creates a new network named custom\_network** **with a subnet of 192.168.99.0/24 and a gateway of 192.168.99.1**. By specifying the --subnet and --gateway options, we configure the network to use these custom settings, ensuring that the containers connected to this network will use IP addresses within the defined range.

****

****

upon inspecting the new custom\_network using its network ID, by executing the command docker inspect e1bbfbf0c3d2 we get the detailed information about the network, and upon inspection, you will observe that it has the same subnet (192.168.99.0/24) and gateway (192.168.99.1) that were allocated during the creation of the network. This confirms that the network was set up correctly with the specified configuration.

****