Activity 1: Core Docker Operations

🎯 Objective

Gain hands-on experience with running containers on your **AWS EC2 (Ubuntu 24.04 LTS)** instance prepared in Activity 0.

By the end of this activity, you should be able to:

- Run and inspect containers, images confidently.
- Work interactively inside containers.
- Run containers as non-root for better security.
- Keep the host clean using housekeeping commands.

✅ Prerequisites

- You have completed **Activity 0** and can SSH into the EC2 instance as `ubuntu`.
- Docker Engine is installed and you can run `docker` without `sudo` (user in `docker` group).

🐳 Task 1 — Run Basic Containers

Goal: Understand how images are executed and how containers behave in both **short-lived** and **long-running** scenarios.

Do's:

- Run a short-lived container (`hello-world`) and long-running ones (`nginx`, `ubuntu`).
- Assign **meaningful names** to all containers.
- Apply a consistent **label** to all containers for easy filtering.
- Practice running in **foreground** and **detached** modes.
- For the web server, **publish ports** so it is accessible externally.

Step-by-step actions

1. Run the short-lived container (`hello-world`)

This tests image pulling and one-shot container execution with meaningful names.

- **Name:** `lab1-hello`
- **Label:** `lab=act1`

Shell

```
docker run --name <?> --label <?> hello-world
docker ps
docker logs <?>
```

What this does & why:

- `docker run` pulls the image if not available locally and runs it.
- `--name <container-name>` sets fixed container name (important for later inspection).
- `--label <key>=<value>` adds metadata for filtering.
- `hello-world` runs once and exits after printing a confirmation message.
- `docker ps` → lists only **running** containers.
- `docker logs` lets us view container logs.

2. Run the long-running `nginx` service (Foreground vs Detached) with published port

Start Nginx in detached mode, accessible on host port `8080`.

- **Name:** `lab1-nginx`
- **Label:** `lab=act1`

Foreground example:

Shell

```
docker run --name <?> --label <?> -p <?> nginx:latest
```

What this does & why:

- `-p <host-port>:<container-port>` maps *container-port* to *host-port*

- Runs the latest Nginx web server image.
- Runs in the terminal, showing logs live.
- Useful for debugging and reading logs directly.
- Can be tested from another terminal by `curl http://localhost:8080`
- Stop with `Ctrl+C`.
- Not ideal for long running apps because it ties up your terminal.

Detached example (recommended for services):

Shell

```
docker run -d --name <?> --label <?> -p <?> nginx:latest
docker ps
```

What this does & why:

- **`-d`** runs the container in the **background**.
- Can be tested from the same terminal by `curl http://localhost:8080`
- With `docker ps` command verify the port mapping is done properly.
- Preferred for long-running or production-like scenarios.
- Standard for persistent services.

3. Run an `ubuntu` container that stays alive (for interactive work)

- **Name:** `lab1-ubuntu`
- **Label:** `lab=act1`

Option A — short-lived keep-alive

Shell

```
docker run -d --name <?> --label <?> ubuntu:latest sleep 300
docker ps
docker exec -it <?> bash
echo ok > /lab1.txt
exit
docker exec -it lab1-ubuntu cat /lab1.txt
```

- `sleep 300` keeps container alive for 5 minutes.
- Allows temporary interactive work.
- `docker exec` runs a command inside an already running container without restarting it.
- `-i` (interactive) keeps STDIN open so you can provide input to the command.
- `-t` (tty) allocates a pseudo-terminal, making the session behave like a normal terminal.
- `exec -it <container> bash` opens Bash shell inside the running container.
- `exec -it <container> cat /lab1.txt` runs `cat` inside the container to display the contents of /lab1.txt on your *host's* terminal.

Option B — long-lived keep-alive (use for autograding)

Shell

```
# Observe container 'lab1-ubuntu' exists
docker ps -a

# Remove it to create container with same name
docker rm lab1-ubuntu

docker run -d --name <?> --label <?> ubuntu:latest tail -f
/dev/null

# Returns "No such file or directory"
docker exec -it lab1-ubuntu cat /lab1.txt  docker exec <?> bash
-c 'echo ok > /lab1.txt'

docker exec -it lab1-ubuntu cat /lab1.txt  # Returns "ok"
```

- `docker ps -a` lists **all** containers, including stopped and exited.
- `docker rm <container-name>` → removes the container.
- `tail -f /dev/null` keeps container running indefinitely (common trick).
- Ideal for scenarios where the grader will connect later.
- `bash -c '<command>'` tells Bash inside container to execute the quoted command.

📌 Verification checklist for successful autograding

- `lab1-hello` exists, exited successfully, logs contain the hello message.
- `lab1-nginx` is **Up** and serves HTTP on port `8080`.
- `lab1-ubuntu` is **Up**, labeled `lab=act1`, and `/lab1.txt` contains `ok`.
- All containers have label `lab=act1`.

Useful commands for this task

Command	**Purpose**	**Example**
`docker run [OPTIONS] IMAGE [CMD]`	Create and start a container	`docker run --name lab1-hello --label lab=act1 hello-world`
`--name NAME`	Assign fixed name	`--name lab1-nginx`
`--label key=value`	Add metadata	`--label lab=act1`
`-d` / `--detach`	Run in background	`docker run -d ... nginx`
`-p host:container`	Publish container port	`-p 8080:80`
`docker ps` / `docker ps -a`	List running / all containers	`docker ps --filter "label=lab=act1"`
`docker logs CONTAINER`	View container logs	`docker logs lab1-hello`
`docker inspect CONTAINER`	Detailed metadata	`docker inspect lab1-nginx`
`docker exec -it CONTAINER CMD`	Run inside container	`docker exec -it lab1-ubuntu bash`
`docker port CONTAINER`	Show port mappings	`docker port lab1-nginx`

Command	Purpose	Example
`docker stop` / `docker rm`	Stop/remove container	`docker stop lab1-nginx && docker rm lab1-nginx`
`curl URL`	Test HTTP from host	`curl -sI http://localhost:8080`

🐳 Task 2 — Master Key CLI for Lifecycle & Inspection

Goal: Learn to list, filter, inspect containers/images in detail, as well as manage their lifecycle.

Do's:

- Use `docker ps` and `docker images` with **filters** and **custom formats** to extract exactly what you need.
- **Inspect** specific metadata fields using `docker inspect` w/o dumping the entire JSON.
- Explore container internals: **IP addresses**, **mounts**, **environment variables**, **commands**, **running processes**.
- Use `docker top`, `docker stats` to monitor live resource usage.

Step-by-step actions

1. List containers

Filter containers by label:

Shell

```shell
docker ps -a --filter "label=lab=act1"
```

Custom output format (name + status only):

Shell

```shell
docker ps -a --format "table {{.Names}}\t{{.Status}}"
```

Discover available fields for formatting:

Shell

```shell
docker ps --format '{{json .}}' | head -n 3 | jq .
```

What this does & why:

- `--filter` → narrow down results (e.g., by label, status, name).
- `--format` → output only the fields you care about (avoids clutter).
- `--format '{{json .}}' | jq .` → shows all available keys for custom formatting.
- `-n 3` shows only top 3 containers' details from the list of all containers.

🖊️ **Student Action:**

Find the **`CreatedAt`** value of container `lab1-nginx` and write it in `ans.json` as:

JSON

```json
{ "lab1-nginx-CreatedAt": "<value>" }
```

2. Manage images

List all images:

Shell

```
docker images
```

Filter by repository name:

Shell

```
docker images --filter=reference="nginx:*"
```

Custom format (repository + size):

Shell

```
docker images --format "table {{.Repository}}\t{{.Size}}"
```

Discover available fields for images:

Shell

```
docker images --format '{{json .}}' | head -n 3 | jq .
```

Remove unused image by ID:

Shell

```
docker rmi <image-id>
```

What this does & why:

- `docker images` → list local images.
- `--filter=reference` → match specific repository/tags.
- `docker rmi` → removes image from local cache (only if unused).
- `--format` key discovery helps automate reporting.

✏️ **Student Action:** Find the **Size** of the `nginx` image and append it to `ans.json`:

JSON

```json
{ "lab1-nginx-size": "<value>" }
```

3. Inspect container metadata (targeted fields)

- **Name:** `lab1-ubuntu` (from Task 1)

Inspect all metadata (JSON output):

Shell

```shell
docker inspect lab1-ubuntu
```

Get container's IP address only:

Shell

```shell
docker inspect --format '{{ .NetworkSettings.IPAddress }}' lab1-ubuntu
```

Get container's mount points:

Shell

```shell
docker inspect --format '{{ json .Mounts }}' lab1-ubuntu | jq
```

Get container’s command:

Shell

```
docker inspect --format '{{ .Config.Cmd }}' lab1-ubuntu
```

What this does & why:

- `docker inspect` → complete metadata in JSON.
- `--format '{{ ... }}'` → extract only the specific field you want (Go syntax).
- Useful for automation, scripts, and clean CLI outputs.

🖊️ **Student Action:** Record the **Image Sha256** of `lab1-ubuntu` in `ans.json`:

JSON

```
{ "lab1-ubuntu-sha256": "<value>" }
```

4. Inspect runtime environment

List processes inside a running container:

Shell

```
docker top lab1-ubuntu
```

View environment variables:

Shell

```
docker inspect --format '{{ json .Config.Env }}' lab1-ubuntu | jq
```

View container filesystem layout from host:

Shell

```
docker inspect --format '{{ .GraphDriver.Data.MergedDir }}'
lab1-ubuntu
```

Monitor resource usage (live stats):

Shell

```
docker stats lab1-ubuntu
```

What this does & why:

- `docker top` → shows running processes inside the container.
- `.Config.Env` → retrieves environment variables at start.
- `.GraphDriver.Data.MergedDir` → shows where the container filesystem is mounted on host.
- `docker stats` → real-time CPU, memory, and network usage monitoring.

🖊️ **Student Action:** Find the value of the environment variable `PATH` from `lab1-ubuntu` and save to `ans.json`:

JSON

```
{ "lab1-ubuntu-path": "<value>" }
```

📌 Verification checklist for successful autograding

- Listed `lab1-nginx-CreatedAt`, `lab1-nginx-size`, `lab1-ubuntu-sha256`, `lab1-ubuntu-path` values in `ans.json`.

Useful commands for this task

Command	Purpose	Example
`docker ps` / `docker ps -a`	List running / all containers	`docker ps -a --filter "label=lab=act1"`
`docker images`	List images	`docker images --filter=reference="nginx:*"`
`--filter key=value`	Narrow listing	`--filter "name=lab1"`
`--format '{{...}}'`	Custom output fields	`--format '{{.Names}}'`
`docker inspect CONTAINER`	View container metadata	`docker inspect lab1-ubuntu`
`docker top CONTAINER`	Show running processes	`docker top lab1-ubuntu`
`docker stats [NAME]`	Live resource usage	`docker stats lab1-ubuntu`
`jq`	Pretty-print JSON	`...|jq`

🐳 Task 3 — Run Containers as Non-Root

Goal: Improve security by avoiding the default `root` user inside containers.

Do's:

- Verify the running user inside a container.
- Use `--user` flag to specify a non-root UID/GID.
- Avoid granting unnecessary privileges to containers.
- Use official images that support non-root operation (or modify them if needed).

Step-by-step actions

1. Check the default user inside a container

Run a temporary Ubuntu container and check the user:

Shell

```
docker run --rm ubuntu:latest whoami
```

What this does & why:

- `whoami` inside container prints the current user (usually `root` by default).
- `--rm` removes the container after it exits (no leftover).
- This helps confirm the container’s default privileges.

🖊️ **Student Action:** Record the **default user** for `ubuntu:latest` in `ans.json`:

JSON

```
{ "ubuntu-default-user": "<value>" }
```

2. Run a container as a specific non-root user

Run with UID 1000 (typical first non-root user on Linux):

Shell

```
docker run --rm \
     --name lab1-nonroot --label lab=act1 \
     --user 1000:1000 ubuntu:latest whoami
```

What this does & why:

- `--user <UID>:<GID>` sets the user and group inside the container.
- Prevents processes from having root privileges.
- Reduces risk if the container is compromised.

🖊️ **Student Action:** Record the **user** shown when running with UID 1000 in `ans.json`:

JSON
```
{ "lab1-nonroot-user": "<value>" }
```

3. Verify user for an interactive container

Start a container and verify UID/GID:

Shell
```
docker run -d --name lab1-nonroot-int --label lab=act1 \
      --user 1000:1000 ubuntu:latest \
      tail -f /dev/null

docker exec -it lab1-nonroot-int bash
whoami
id
exit
```

What this does & why:

- `-it` opens an interactive terminal inside the container.
- `whoami` shows the username (may show UID if no `/etc/passwd` entry).
- `id` shows the UID/GID explicitly.

🖊️ **Student Action:** Record the **UID** from `id` output in `ans.json`:

JSON
```
{ "lab1-nonroot-uid": "<value>" }
```

4. Check processes and permissions

From the **host**, verify the running container's process owner:

Shell

```
# Expected: UID column should not show roo#
docker top lab1-nonroot-int
```

From inside the container verify the user's permissions:

Shell

```
docker exec -it lab1-nonroot-int bash
ls -ld /root
touch /root/testfile  # Expected: Permission denied
```

What this does & why:

- `docker top` shows **host-level** process info (here should be owned by the non-root UID).
- `ls` → list information
- `-l` → long format (permissions, owner, group, size, date, name)
- `-d` → show the directory entry itself, not what's inside it
- `touch` create empty file
- Non-root user should not have permission to write to `/root`.
- Confirms reduced privileges are enforced.

🖊️ **Student Action:** If `/root/testfile` creation fails, autograder test case will pass.

📌 Verification checklist for successful autograding

- `ans.json` contains:
  - `ubuntu-default-user`
  - `lab1-nonroot-user`
  - `lab1-nonroot-uid`

- Default userid for the `lab1-nonroot-int` container must be non-root.

Useful commands for this task

Command	Purpose	Example
`docker run --user UID:GID`	Run as specific user/group	`docker run --user 1000:1000 ...`
`whoami`	Show current username	`whoami`
`id`	Show UID/GID	`id`
`docker top CONTAINER`	Show container processes from host	`docker top lab1-nonroot`

🐳 Task 4 — Image & System Housekeeping

Goal: Keep your Docker environment clean by identifying and removing unused resources.

Do's:

- Identify **dangling** and **unused** images.
- Review disk usage for images, containers, and volumes.
- Use pruning commands carefully — know **exactly** what will be removed before running them.

Step-by-step actions

1. List all images & identify dangling ones

Shell

```
docker image ls
```

- Shows all local images.
- **Dangling images**: untagged (`<none>`) images — often left behind after rebuilds.
- These can usually be removed without breaking anything.

2. Review Docker disk usage

Shell

```shell
docker system df
```

What this does & why:

- Shows how much space is used by images, containers, volumes, and build cache.
- Helps decide if cleanup is needed.

3. Remove dangling images

Shell

```shell
docker image prune
```

- Removes all dangling images (prompts for confirmation).
- Add `--force` to skip confirmation:

4. Remove stopped containers

Shell

```shell
docker container prune
```

- Deletes all stopped containers.
- Add `--filter until=24h` to remove only those older than 24 hours:

Shell

```shell
docker container prune --filter until=24h
```

❗ Note: *If you deleted the* `hello-world` *container, you must recreate it before running all the test cases to ensure they pass.*

5. Remove unused volumes

```shell
docker volume prune
```

- Removes **unused** volumes (not referenced by any container).
- ⚠️ Be careful — this can delete persistent data.

6. Full cleanup (images + containers + networks + build cache)

```shell
docker system prune
```

- Removes all unused containers, networks, and dangling images.
- Add `--volumes` to also remove unused volumes:

```shell
docker system prune --volumes
```

- This removes all dangling **build cache**:

```shell
docker buildx prune
```

📌 Verification checklist for successful autograding

- After cleanup, `docker image ls` should **not** list any `<none>` images.
- `docker system df` should show reduced usage compared to Step 2.

Useful commands for this task

Command	Purpose	Example
`docker image ls`	List local images	`docker image ls`
`docker system df`	Show disk usage	`docker system df`
`docker image prune`	Remove dangling images	`docker image prune --force`
`docker container prune`	Remove stopped containers	`docker container prune --filter until=24h`
`docker volume prune`	Remove unused volumes	`docker volume prune`
`docker system prune`	Full cleanup	`docker system prune --volumes`

📝 Final Verification Checklist for successful autograding

Task 1 — Basic Containers

- `lab1-hello`: exists, exited successfully, logs show hello message.
- `lab1-nginx`: **Up**, serving HTTP on port `8080`, labeled `lab=act1`.
- `lab1-ubuntu`: **Up**, labeled `lab=act1`, `/lab1.txt` contains `ok`.

Task 2 — CLI & Inspection

- `ans.json` contains:
  - `lab1-nginx-CreatedAt`
  - `lab1-nginx-size`
  - `lab1-ubuntu-sha256`
  - `lab1-ubuntu-path`

Task 3 — Non-Root Containers

- `ans.json` contains:
  - `ubuntu-default-user`
  - `lab1-nonroot-user`
  - `lab1-nonroot-uid`
- `lab1-nonroot-int` runs as non-root, `/root/testfile` cannot be created.

Task 4 — Housekeeping

- No `<none>` images remain (`docker image ls`).

🧹 Cleanup After the Activity

Stop/remove any containers you no longer need and reclaim space using the **housekeeping** tools you explored in **Task 4**.

Do's:

- Prefer **`stop`** over **`kill`** for a graceful shutdown.
- Always `prune` resources you no longer need to keep the host clean.

5. Stop and remove containers (lifecycle control)

Gracefully stop a running container:

Shell

```
docker stop lab1-ubuntu
```

Force kill (immediate termination):

Shell

```
docker kill lab1-ubuntu
```

Remove a stopped container:

Shell

```
docker rm lab1-ubuntu
```

What this does & why:

- `stop` → sends **SIGTERM**, allowing graceful shutdown.
- `kill` → sends **SIGKILL**, forcing immediate exit (use only if stop fails).
- Removing stopped containers frees up space and avoids clutter.

Useful commands for this task

Command	Purpose	Example
`docker stop CONTAINER`	Graceful stop	`docker stop lab1-ubuntu`
`docker kill CONTAINER`	Force stop	`docker kill lab1-ubuntu`
`docker rm CONTAINER`	Remove container	`docker rm lab1-ubuntu`
`docker rmi IMAGE`	Remove image	`docker rmi nginx:latest`

🛑 Stop / Terminate Instance (to avoid charges)

- After completing evaluation, it is important to stop the instance so that no extra costs are incurred.
- To **stop** the instance (preserve disk/data): In EC2 Console → select instance → **Instance state** → **Stop instance**.
- To **terminate** (delete resources): **Instance state** → **Terminate instance**.

🏆 Congratulations — You’ve completed Activity 1!

END OF DOCUMENT