Foundational Cloud Infrastructure and Performance Analysis of a Containerized LLM Inference Service using the Docker Ecosystem

Sina Hakimzadeh

March 19, 2025

1 Introduction

In this homework, we containerized an LLM inference application using Docker and compared different ways of deploying an application on it. We started by running the application locally to understand the application workflow and how to prepare it. Then, we moved on to creating a Docker image and running the application with Docker using a local image. After that, we pushed it to Docker Hub using GitHub Actions and also explored Docker Compose and Swarm to understand what each one does.

2 Step Explaination

In this section, we describe each homework step and command to understand the workflow.

2.1 Setting Up and Testing the Inference Application

The application is an inference application that contains a flask-based sentiment analysis API and uses PyTorch and transformer libraries to access a pre-trained LLM model. The application receives a text from user via an HTTP request, and analyzes the sentiment using the model. In the end, the API returns whether the text is positive, negative, or neutral.

Listing 1: Building the Docker image

```
$ curl -X POST -H "Content-Type: application/json" -d '{"text":"This movie was
    ↪ fantastic!"}' http://localhost:5001/infer
```

This command uses the `curl` tool to send an HTTP request to the local host where our application is listening. The output is :

Listing 2: Building the Docker image

```
    {"input_text":"This movie was fantastic!","sentiment":"POSITIVE"}
```

The application code logs some details in a file named `system_inference_metrics.csv`, allowing us to track information such as inference latency. Additionally, there is a script named `latency_test.sh` that sends 160 HTTP requests to the application for testing.(Figure 1)

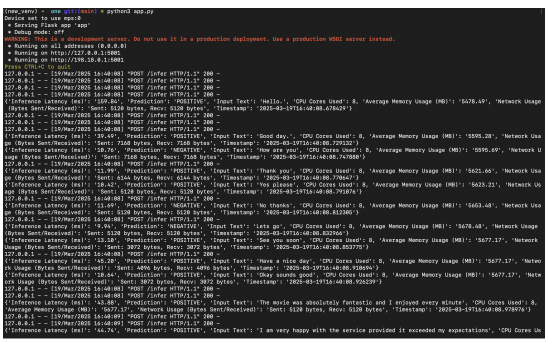

Figure 1: Running application with no Docker

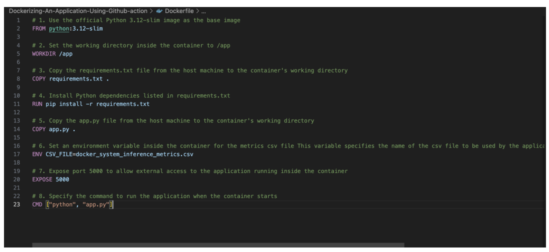

Figure 2: Dockerfile

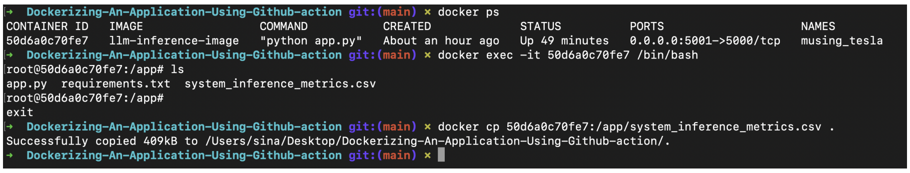

Figure 3: Access to container files and copy it to host

2.2 Containerization with Docker

After we understand the workflow of the application, we goes on containerizing it using docker.

but why would we use containerization?(Objective 1)

We use Docker and containerization for our applications to ensure consistency, scalability, and efficiency in deployment. The benefits of using container:

- Containers package an application with all its dependencies (libraries, configurations, and runtime).
- Containers use host operating system, unlike virtual machines; thus, containers are lightweight.
- Each container runs in isolation, meaning one application won't affect another.
- No need to manually install dependencies on different environments.
- Some tools, like Docker Compose, make it easy to manage multi-container applications.

As mentioned above, Docker containers need to be created with specific configurations so that everything is ready when we run them. To instruct Docker on how to create the image, we must provide a `Dockerfile` for Docker to understand what it should do.

Figure 2 shows all the configurations added to the file:

- `FROM` sets the base image for the Docker container. The `python:3.12-slim` image is an official Python image that includes Python 3.12 on a minimal Debian-based Linux distribution.
- `WORKDIR` sets the working directory for all subsequent Dockerfile instructions. Any commands like `COPY`, `RUN`, and `CMD` will operate within this directory (`/app`) inside the container.
- `COPY` copies a file from your host device to the Docker image. Here, it copies *requirements.txt*, which contains the Python requirements. Additionally, *app.py* is copied in step 5.
- `RUN` runs the `pip install -r requirements.txt` command inside the container to install all the dependencies specified in the `requirements.txt` file.
- `ENV` sets an environment variable `CSV_FILE` inside the container with the value `docker_system_inference_metrics.c`
- `EXPOSE` tells Docker to expose port 5000 of the container to the host machine. The containerized application is expected to run a web server (e.g., Flask), and by exposing port 5000, you allow external access to the application.
- `CMD` is the default command that will be executed when the container starts. In this case, it runs the Python script `app.py`.

All these steps are followed in order to create a Docker image by running Listing 3 , so the produced image can be run on any computer without considering the application's dependencies or the operating system requirements.

Listing 3: Building the Docker image

```
docker build -t llm-inference-image .
```

Table 1 contains some common Docker commands that can be used to manage Docker images. We use `docker run llm-inference-image` to run our image. We will discuss the metrics of running a container in Section 3. After running the container, by using `docker exec -it container_id /bin/bash`, we get access to the container shell and can access our file. By using `docker cp <container_id>:<source_path> <destination_path_on_host>`, we can copy a file from the container to our host device, as shown in Figure 3.

Command	Description
`docker run {image-name}`	Run a container from an image.
`docker run -d {image-name}`	Run a container in detached mode.
`docker stats`	Display runtime statistics of containers.
`docker images`	List all Docker images on the local machine.
`docker ps`	List running containers.
`docker ps -a`	List all containers, including stopped ones.
`docker build -t {image-name} .`	Build an image from a Dockerfile with a specified name.
`docker pull {image-name}`	Download an image from Docker Hub or a registry.
`docker push {image-name}`	Push an image to a registry (e.g., Docker Hub).
`docker exec -it {container-name} {command}`	Execute a command inside a running container.
`docker stop {container-name}`	Stop a running container.
`docker start {container-name}`	Start a stopped container.
`docker rm {container-name}`	Remove a stopped container.
`docker rmi {image-name}`	Remove an image from the local machine.
`docker logs {container-name}`	View the logs of a container.

Table 1: Common Docker Commands

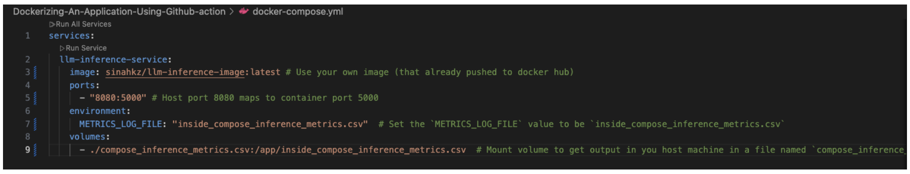

Figure 4: docker-compose.yml file

2.3 Docker Compose (Objective 2)

Docker Compose makes it easy to manage multi-container Docker applications with a simple `docker-compose.yml` file. Instead of running multiple `docker run` commands manually, you can define everything in one place and start everything with a single command. The benefit of using docker compose is:

- **Easier setup & configuration:** Instead of writing long `docker run` commands, you define everything once in *docker-compose.yml*.
- **Easy Environment Variable Management:** You can manage environment variables with a *.env* file instead of passing them in docker run commands.
- **Future Expansion:** you can easily expand your service by modifying the *docker-compsoe.yml*
- **Persistent Storage with Volumes:** Docker Compose makes it easy to keep data safe by automatically managing storage, even if containers stop or get removed.

Figure 4 shows the *docker-compose.yml*. We defined a service named *llm-inference-service* that uses an image pulled from Docker Hub named *sinahkz/llm-inference-image:latest*, maps host port 8080 to container port 5000, sets an environment variable, and finally sets a volume that links a file from the host machine to a location inside the container to save data. Start Docker services with the `docker-compose up -d` command, and all services defined in *docker-compose.yml* will run. Every changes in *inside_compose_inference_metrics.csv* file will be stored in *compose_inference_metrics.csv*. We discuss about docker compose metrics in section 3. In the end use `docker-compose down` command to stop and remove the running service.

2.4 Orchestration & Scaling With Swarm(Objective 3)

Docker Swarm is a built-in tool in Docker that helps manage multiple containers across different machines (called nodes). It makes it easy to run and scale applications reliably by ensuring they remain available,

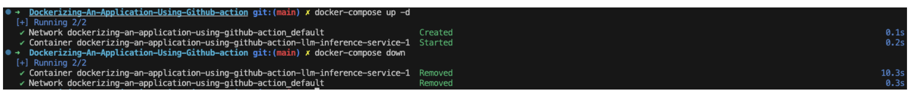

Figure 5: Creating and removing a docker service using docker compose

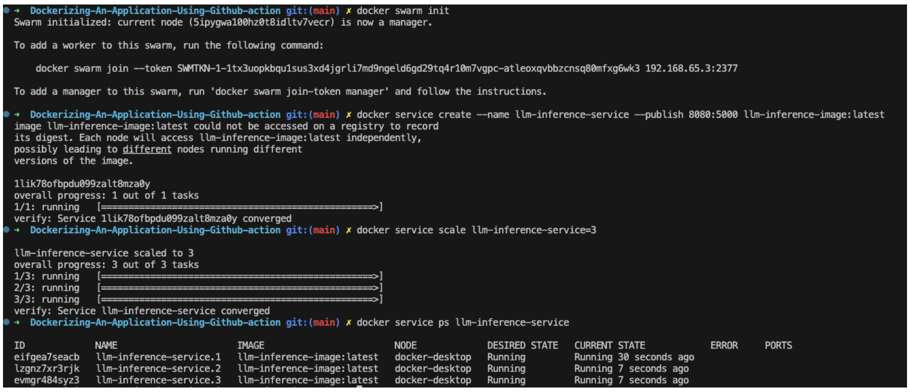

Figure 6: Deploy App Using Swarm and Creating Replica

even if some machines fail.

In other words, Docker Swarm lets you control multiple Docker containers on different machines as if they were part of a single system.

Why use Swarm when Docker Compose exists?

There are multiple situations where Swarm is preferred over Docker Compose. Docker Compose manages multi-container applications on a **single machine**, whereas Swarm orchestrates multiple containers across multiple machines. In other words, Swarm includes cluster management. Swarm orchestration can automatically scale an application across multiple machines and ensure high availability through container redundancy. These features make Swarm a great choice for production-ready orchestration, while Docker Compose is more suitable for development and testing before deployment.

Use `docker swarm init` to create a Swarm cluster and make the current machine the **Swarm manager**. The manager node is responsible for orchestrating services, scheduling tasks, and managing worker nodes. If we are running Docker on a single machine, it acts as both the manager and the worker.

Listing 4: Creating a service using Swarm

```
docker service create --name llm-inference-service --publish 8080:5000
    ↪ llm-inference-image:latest
```

We create a service using command 4. The `--name` flag specifies the service name, while `--publish` maps a port from the host to the containers. Finally, we provide the Docker image name. Now our service is running, and we can use the previous commands to send HTTP requests to it. The application has been successfully deployed, and the service is available based on the configured settings.

Listing 5: Creating Replicas

```
docker service scale llm-inference-service=3
```

This command increases the number of running containers (replicas) from 1 to 3. The Swarm manager automatically distributes these replicas across available nodes. Each container listens on port 5000 inside its instance, while traffic is load-balanced across them.

What are the benefits of creating replicas?

More replicas provide better scalability and higher availability. Additionally, if one container fails, another can continue handling requests.
To verify the service deployment, we use `docker service ps llm-inference-service` to check the running containers. Figure 6 illustrates all the steps involved in creating replicas using Swarm. We will discuss the metrics in Section 3.

3 Metrics & Comparison(Objectives 4,5,6)

Profiling an application without Docker requires additional tools, and I tried to simulate the metrics measurement using the **psutil** package in Python. It is evident that running an application without Docker can perform much faster with lower latency(as shown in figure 9 & table 2), but it cannot be distributed and also cannot provide **scalability**, **high availability**, and a **fault-tolerant** system.

Listing 6: Creating Replicas

```
docker stats
```

Name	CPU %	Memory	Network I/O	Avg. Latency (ms)
No Docker	800%	5762.08MB	6.372KB / 5.985KB	18.6
Simple Run	430%	270.5MB	274MB / 3.39MB	29.1
Docker Compose	415%	470MB	285MB / 4.46MB	29.02
Swarm (Single Rep.)	447.2%	358MB	286MB / 7.9MB	31.37
Swarm (3 Rep. Avg.)	146%	405.7MB	286MB / 6.04MB	23.01

Table 2: Comparison of Performance Metrics Across Deployment Scenarios

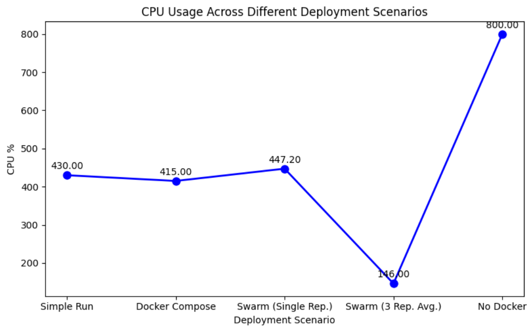

Figure 7: CPU Usage Across Different Deployment Scenarios

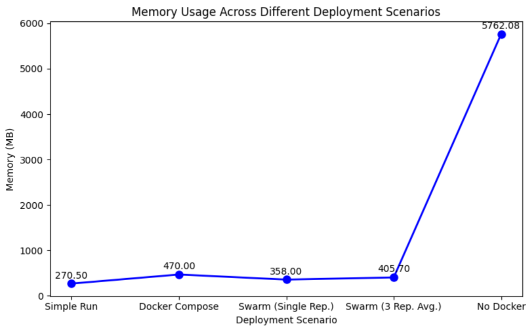

Figure 8: Memory Usage Across Different Deployment Scenarios

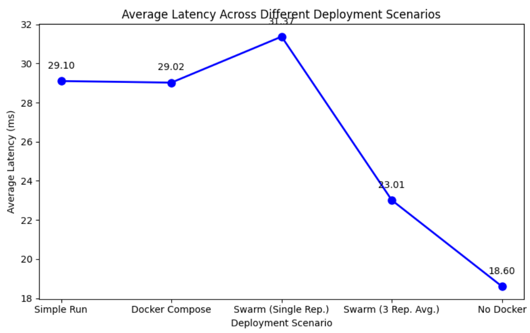

Figure 9: Average Latency Across Different Deployment Scenarios

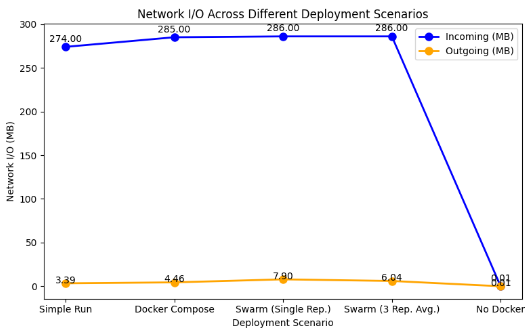

Figure 10: Network I/O Across Different Deployment Scenarios

Listing 6 allows us to access the real-time CPU, memory, and network usage of containers. My running results are listed in Table 2 for all different Docker container running methods.

Latency Improvement with Docker Swarm

The results show that using three replicas in Docker Swarm significantly improves latency by about 20% (Figure 9). This happens because when a request arrives, the Swarm load balancer distributes the requests among replicas to achieve better latency and prevent congestion. However, for a single replica, latency increased because of the load balancer task. It is evident that Docker Compose and Simple run have almost the same latency because there is no built-in load balancer in Docker Compose to create more latency.

Impact on Memory and Network Traffic

On the other hand, using Swarm with multiple replicas consumes more memory and generates more network traffic compared to other methods because it is more complex. The difference between Docker Compose and a simple `docker run` command lies in the overhead of managing multiple containers. As a result, it consumes more memory and network traffic to handle them.

CPU Usage in Docker Swarm

A key observation is that CPU usage is highest in Swarm with a single replica (447.2%). This is likely due to the extra overhead from Swarm's scheduling and networking. However, when using three replicas, CPU usage per container drops to around 146%, showing that the workload is better spread across multiple instances. But it is almost the same for others.

Network Traffic and Further Testing

While the three-replica setup improves latency, it doesn't significantly reduce network traffic compared to the single-replica setup. A good explaination about why this happened in from ChatGPT:
In Docker Swarm, adding more replicas can help balance the workload, but it doesn't always reduce network traffic. This happens because the total data transmitted may stay the same, especially if replicas are accessing shared resources like a database. Additionally, internal communication between containers for tasks like service discovery can still generate network traffic. So, while scaling with replicas improves performance, it doesn't always reduce the amount of data being transferred.

3.1 Pros & Cons of Running an Application in Different Scenario?

Running on your own system

Running an application on your system is more easier and faster for the first deployment; Additionally, it does not have any management overhead. Also you have full control of how your application is deployed. But it's hard to scale applications to handle increased load without setting up more servers or complex solutions. Running directly on the system means the application shares resources with other apps and could potentially cause conflicts, performance issues, or security risks.

Running in Docker

Docker container gives you **isolation**, **portability**, and **consistency** across different machines. But Docker is challenging for its new users and takes time to learn; also, it has some overhead compared to direct execution because of management. As a result, it is not useful for small and non-distributed applications.

Running with Docker Compose

Docker compose simplifies containers configuration and allows you to create and manage multiple container easily. But it introduces more overhead. If the application is simple and doesn't require multiple containers, Docker Compose adds unnecessary complexity. In the end, docker compose is a perfect tool for big application to test their application before deployment.

Running with Docker Swarm

Docker Swarm is a container orchestration tool that enables automatic scaling of containers across multiple machines(**Scalability**). It's great for distributed applications that need to scale easily. Swarm provides built-in high availability by running replicas of containers across different nodes(**high-availability**).

4 Github Action(Bonus)

Github action is a perfect tool to automate some tasks for deplyments like creating a docker image and upload it on docker hub automatically. In this homework i did this by writing a simple workflow for github action.

```
name: ci
on:
  push:
    branches:
      - "main"
jobs:
  docker:
    runs-on: ubuntu-latest
    steps:
    - name: Checkout
      uses: actions/checkout@v4
```

```
      - name: Set up QEMU
        uses: docker/setup-qemu-action@v3

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v3

      - name: Login to Docker Hub
        uses: docker/login-action@v3
        with:
          username: ${{ secrets.DOCKER_USERNAME }}
          password: ${{ secrets.DOCKER_PASSWORD }}
      - name: Build and push
        uses: docker/build-push-action@v5
        with:
          context: .
          file: ./Dockerfile
          platforms: linux/amd64,linux/arm64
          push: true
          tags: |
            sinahkz/llm-inference-image:latest
            sinahkz/llm-inference-image:${{ github.run_number }}
```

I defined an action named `ci` and pushed it to the main branch. Its job is to build and push a Docker image to Docker Hub. The GitHub action provides some pre-written procedures for us to use. `Checkout` fetches the repository code so the workflow can access it. `QEMU` enables emulation, allowing Docker to build images for multiple CPU architectures. `Docker buildx` enables Buildx, an enhanced Docker builder that supports multi-platform builds. Before creating and pushing the image to Docker Hub, we first log into our account in the `Login to Docker Hub` step. Finally, the GitHub action builds and pushes the image with the specified name.
In Github action, if any code changes are committed to the repository, it can trigger the action flow, depending on the configuration in the workflow YAML file. Github repository link

5 Conclusion

In conclusion, this homework project allowed us to explore containerization and deployment strategies using Docker, Docker Compose, and Docker Swarm. By containerizing a sentiment analysis application, we gained a deeper understanding of how containerization provides consistency, scalability, and efficiency in deploying applications. We demonstrated the advantages of Docker in packaging dependencies and simplifying the deployment process. Furthermore, Docker Compose and Swarm enabled us to scale and manage multi-container applications with ease, while Swarm offered high availability and fault tolerance through replication.

The performance metrics revealed that while Docker-based deployments introduced slightly higher latency and resource usage compared to running the application directly, they provided significant benefits in terms of scalability and fault tolerance. Swarm, in particular, stood out for its ability to manage multiple replicas and distribute the load across nodes, further enhancing the application's reliability.