Data Engineering II - 1TD076

Assignment 1 - Orchestration and contextualization using docker containers

Teame Hailu Gebru

April 8, 2025

Task 1. Introduction to Docker containers and DockerHub

1. Explain the difference between contextualization and orchestration:

   Docker contextualization refers to the process of preparing the customized environment based on the specific needs of an application. It includes setting up the container with the necessary configuration, dependencies and resources before deployment.

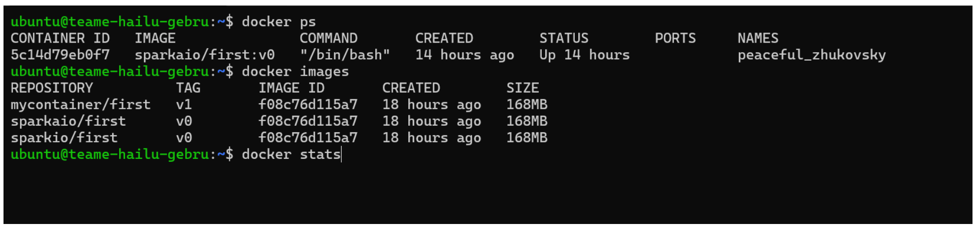
   Where as **Docker orchestration** is automated management of containerized applications ensuring they run efficiently across multiple hosts. It includes scheduling, scaling, networking, monitoring and failure recovery.

2. Explain the following commands and concepts:

   content of the docker file used in this task-The docker file creates a Docker image based on Ubuntu 22.04, updating and upgrading system packages before installing the sl command-line utility. It modifies the PATH environment variable to include /usr/games/, allowing easy access to sl. The default command, CMD [”echo”, ”Data Engineering-II.”], prints a message when the container runs.After entering an interactive shell to manually execute sl and see an ASCII train animation.

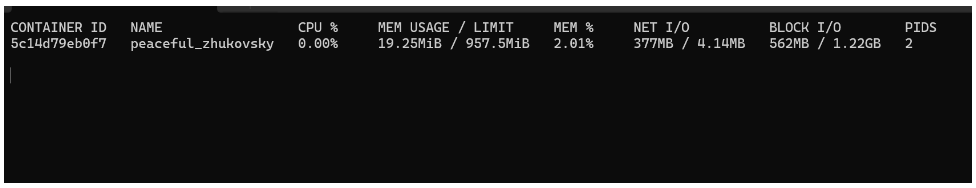
   Explain the command docker run -it mycontainer/first:v1 bash- Docker run creates and starts a new container from specified image. -it combines two flags -i (iterative) keeps the container’s standard input open,allowing user interaction and -t (TTY) allocates a pseudo terminal, making the session behave like normal shell. mycontainer/first:v1 is image name and tag indicating specific task.

   Show the output and explain the following commands- Docker ps shows active containers and **docker images** displays all available docker images in our system.Result for docker image and docker ps is shown in Figure 1 **docker stats**- Displays CPU, memory and network usage for all running containers.Output is shown in figure 2 below.



```
ubuntu@teame-hailu-gebru:~$ docker ps
CONTAINER ID   IMAGE                COMMAND       CREATED        STATUS        PORTS     NAMES
5c14d79eb0f7   sparkaio/first:v0    "/bin/bash"   14 hours ago   Up 14 hours             peaceful_zhukovsky
ubuntu@teame-hailu-gebru:~$ docker images
REPOSITORY          TAG       IMAGE ID       CREATED        SIZE
mycontainer/first   v1        f08c76d115a7   18 hours ago   168MB
sparkaio/first      v0        f08c76d115a7   18 hours ago   168MB
sparkio/first       v0        f08c76d115a7   18 hours ago   168MB
ubuntu@teame-hailu-gebru:~$ docker stats
```

Figure 1: Docker ps and docker images command display



```
CONTAINER ID   NAME                 CPU %     MEM USAGE / LIMIT     MEM %     NET I/O           BLOCK I/O         PIDS
5c14d79eb0f7   peaceful_zhukovsky   0.00%     19.25MiB / 957.5MiB   2.01%     377MB / 4.14MB    562MB / 1.22GB    2
```

Figure 2: Docker stats command display

3. What is the difference between docker run, docker exec and docker build commands?

   Docker run - creates and starts a new container from an image.

   Docker exec - runs a command inside an already running container.

   Docker build - creates docker image from a dockerfile.

4. Create an account on DockerHub and upload your newly built container to your DockerHub area. Explain the usablity of DockerHub. Make you container Publicly available and report the name of you publicly available container.

   DockerHub uses to store, manage, share and distribute docker container images. Name of Publicly available container:**teamehailu/mycontainer:v0**

5. Explain the difference between docker build and docker compose commands. **Docker build** command is used to create a Docker image from a dockerfile. It packages our app and its dependencies into an image that can later run. Whereas, **docker compose** is used to run multi-container applications defined in a docker-compose.yaml file. It can build images and start multiple services (like a web app and database) with one command. In short, docker build creates images, while docker compose runs and manages multi-container setups.

Task 2. Build a multi-container Apache Spark cluster using docker compose

1.Explain your docker compose configuration file?

The docker Compose configuration file sets up a simple Apache Spark cluster using Docker containers. It defines three main services: one Spark master and two Spark workers, all running the Bitnami Spark Docker image. The Spark master container is configured to run in master mode and exposes its web UI on port 8080. It also mounts a host directory to the container to persist data. The two worker containers, spark-worker-1 and spark-worker-2, are set to run in worker mode and are configured to connect to the Spark master using the master's URL (spark://spark-master:7077). Each worker also mounts its own data volume and exposes a web UI on different ports 8081 and 8082 respectively allowing to monitor them individually through a web browser.All services are connected through a custom docker bridge network named spark-network, which allows them to communicate using their container names instead of IP addresses.

2. What is the format of the docker compose compatible configuration file?

A docker compose compatible configuration file is written in YAML format. It starts with a version key to specify the file format version, followed by a services section where each containerized service is defined under a unique name. Within each service, configuration options such as image, container name, ports, environment, volumes, depends on, and networks are used to define how the container should run and interact with others. The file may also include top-level networks and volumes sections to set up shared communication channels and persistent storage. The YAML format relies on proper indentation using spaces (not tabs), and lists are represented with dashes, while key-value pairs use colons. This structured format makes it easy to describe and manage multi container docker applications in a clear and readable way.

3. What are the limitations of docker compose?

   1. **Not Suitable for Production at Scale**
      Docker Compose is primarily designed for development and testing, not for large-scale or distributed production environments.
   2. **Lacks Advanced Orchestration Features**
      It does not support features like auto-scaling, load balancing, or self-healing, which are available in tools like Kubernetes.
   3. **Basic Service Dependency Handling**
      The `depends_on` directive only controls container startup order; it does not wait for a service to be fully ready before starting dependent services.
   4. **Restricted Networking Options**
      Compose supports basic networking, but it lacks more advanced network policies and configurations required in complex setups.
   5. **Single Host Limitation**
      Compose runs containers on a single host, making it unsuitable for managing applications that require multi-node deployments.

Task 3. Introduction to different orchestration and contextualization frameworks

The Role of Runtime Orchestration and Contextualization for Large-Scale Distributed Applications

Runtime orchestration and contextualization are vital in deploying and managing large-scale distributed systems. Orchestration refers to the automated coordination, deployment, and scaling of services across distributed environments. Contextualization, on the other hand, ensures that each container or component is configured with environment-specific parameters, such as network settings, storage access, and credentials. These processes enable distributed applications to be resilient, portable, and efficient across various deployment targets—from development to production.

Modern cloud-native architectures rely on orchestration frameworks to achieve elasticity, high availability, and operational efficiency. In Task 2, we orchestrated a multi-container Spark cluster using Docker Compose. While Compose is user-friendly and ideal for local development, it lacks advanced orchestration features necessary for production-grade systems. This calls for a closer look at more robust orchestration frameworks.

Docker Compose allows developers to define and run multi-container applications using a YAML file. It simplifies service deployment on a single host but does not support automatic scaling, high availability, or multi-node coordination—key requirements in a production Spark environment.

Kubernetes has become the industry standard for container orchestration. It provides powerful features such as automatic bin packing, self-healing (auto-restart and failover), rolling updates and rollbacks, and horizontal scaling. Kubernetes also supports service discovery and network routing across hosts, making it well-suited for dynamic, distributed workloads like Spark. Additionally, tools like the Spark Operator and Custom Resource Definitions (CRDs) streamline Spark deployment, allowing better resource isolation and integration with cloud storage.

Docker Swarm offers a simpler alternative to Kubernetes while still supporting clustering, service replication, rolling updates, and basic load balancing. It is more tightly integrated with the Docker ecosystem and easier to set up but lacks Kubernetes' depth in features like fine-grained resource management and advanced networking.

Apache Mesos with Marathon provides a two-layer architecture where Mesos abstracts resources across a cluster, and Marathon manages long-running services. It supports application placement constraints, health checks, and scaling. Though flexible and powerful, it has seen declining adoption in favor of Kubernetes.

In summary, while Docker Compose is suitable for basic orchestration and development, frameworks like Kubernetes and Docker Swarm can significantly enhance Spark cluster management in production environments. Kubernetes, in particular, offers superior scalability, fault tolerance, and integration with modern DevOps practices, making it the ideal choice for orchestrating large-scale, data-intensive applications.