DevOps – e1Compose

Sang Nguyen

1. Platform information
   - Hardware device for development: Macbook
   - Virtual machine for testing: VMware
   - Operating system for development: macOS Sequoia 15.6.1
   - Operating system for testing: Ubuntu 24.04.1 LTS
   - Docker version in development environment: Docker version 28.0.1
   - Docker version in testing environment: Docker version 28.4.0
   - Docker compose version in development environment: Docker Compose version v2.33.1-desktop.1
   - Docker compose version in testing environment: Docker Compose version v2.39.4
2. Architecture diagram

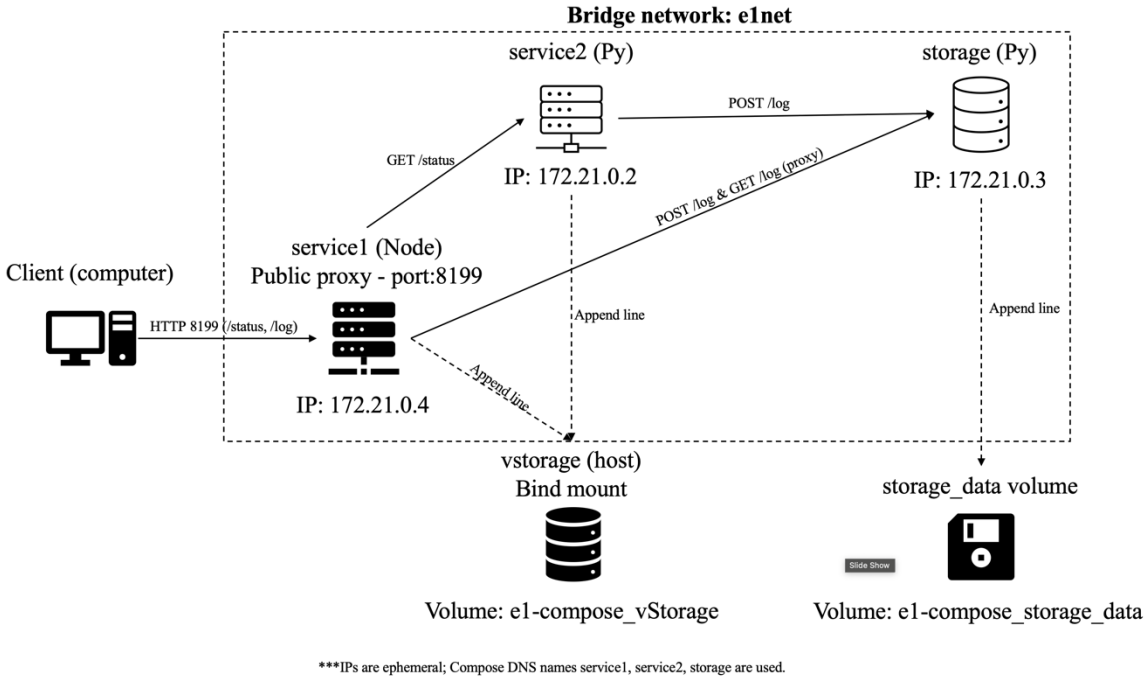

Figure 1: Architecture diagram of e1Compose microservices

3. Status record analysis
   - In this assignment, the aim is to create two interworking services, which are defined and run with Docker Compose. When a status request is sent, the services

return their state, including each service's uptime and the free space of the container's root filesystem.

- In Service1 and Service2, the uptime measurement shows the actual uptime of the current service, which is the service's lifetime since it started, and it resets when the service shuts down or restarts. This helps identify crashes or restarts and provides a quantifiable measure of reliability and performance. The ops team can use uptime as a key indicator to monitor system health and manage incidents, maintaining user trust.
- Meanwhile, the disk-space figure reflects the free space of the container's root filesystem, not the host's free space. In the development environment (macOS), it represents the Linux VM's root. The vStorage volume is bound to a host file store, while the storage_data volume serves as the service-backed store.
- Free space on the root filesystem can diverge from free space on those mount points; therefore, it is better to measure free space on the actual storage targets.

4. Persistent storage solutions
   - vStorage (host bind mount):
     + A named volume bind-mounts the host path ./service1_nodejs/vstorage into service1 and service2 at /app/vstorage. On each /status call, both services append one line directly to the file. This design is simple and makes inspection easy (example: cat/tail).
     + Drawbacks: concurrent writes from multiple containers can interleave or produce partial writes under load. The security is weaker because host users can modify the file or permissions can block containers.
   - storage_data (Docker named volume via Storage service).
     + The Storage (Python) microservice persists to /var/lib/storage/log.txt, backed by the named volume e1-compose_storage_data. Only the Storage service writes the file, while service1 and service2 write via HTTP (POST /log). This single-writer pattern improves consistency and ordering and decouples clients from the storage format. Besides, it enables better observability (the service can expose health/metrics) and supports adding authentication/validation to secure write requests.

+ Drawbacks: adds a small network hop (latency) and creates a single point of failure (the Storage service).

5. Cleanup the services
   - To remove all data from both storages, from project root exercise1/:
     + Use this command to stop the stack (containers) and remove containers and both named volumes: *docker compose down -v*
     + Use this command to clear the host file: *rm -f service1_nodejs/vstorage/log.txt*
6. Project's challenges
   - During development, I encountered challenges with system architecture and debugging.
     + Architecture design: it took time to design how the services should interact to meet the requirements.
     + Inter-service connectivity: the services worked when tested directly on localhost, but Service1 initially couldn't reach Service2 over the e1net bridge.
     + Dockerfile and Compose setup: configuring the Dockerfiles and docker-compose.yaml was challenging, especially the Compose yaml file. I had to ensure services were defined correctly, dependencies declared, environment variables set, and volumes/storage mounted properly.