Семинар 6. Оркестрация контейнеров и микросервисная архитектура ML-моделей (Docker Compose)

---

1. Задание 1. Создание многосервисного проекта

- Создайте следующую структуру проекта:

```
ml_pipeline/
├── preprocess/
│   ├── app.py
│   └── Dockerfile
├── model/
│   ├── app.py
│   └── Dockerfile
├── postprocess/
│   ├── app.py
│   └── Dockerfile
└── docker-compose.yml
```

- В каждом каталоге (`preprocess`, `model`, `postprocess`) создайте простое FastAPI-приложение:
  - `preprocess` принимает изображение и возвращает нормализованный тензор;
  - `model` выполняет «фиктивный» инференс — например, случайный вектор вероятностей;
  - `postprocess` принимает результат модели и возвращает индекс максимума.
- Проверьте каждый сервис отдельно:

```
docker build -t preprocess-service ./preprocess
docker run -p 5001:5000 preprocess-service
```

---

2. Задание 2. Описание системы в Docker Compose

- Создайте файл `docker-compose.yml`:

```
version: "3.9"
services:
  preprocess:
    build: ./preprocess
    ports: ["5001:5000"]
    depends_on: ["model"]
    networks: [ml_net]
```

```
  model:
    build: ./model
    ports: ["5002:5000"]
    networks: [ml_net]

  postprocess:
    build: ./postprocess
    ports: ["5003:5000"]
    depends_on: ["preprocess"]
    networks: [ml_net]

networks:
  ml_net:
    driver: bridge
```

- Запустите всю систему:

```
docker compose up --build
```

- Убедитесь, что все три сервиса поднялись и доступны по именам `preprocess`, `model`, `postprocess`.

---

3. Задание 3. Передача данных через цепочку сервисов

- Добавьте во все приложения маршруты `/health` для проверки состояния.
- В `preprocess/app.py` реализуйте отправку запроса к `model:5000/predict`, а в `postprocess/app.py` — к `preprocess:5000/process`.
- Проверьте полный цикл вызова:

```
curl -X POST http://localhost:5003/finalize -F "file=@test.jpg"
```

- Убедитесь, что запрос проходит через все три контейнера (просмотрите логи).

---

4. Задание 4. Общие данные и тома

- Добавьте общий том для обмена файлами между сервисами:

```
volumes:
  shared_data:
services:
  preprocess:
    volumes: [shared_data:/data]
  model:
    volumes: [shared_data:/data]
```

- В `preprocess` сохраняйте промежуточные данные в `/data/input.npy`, а `model` загружайте их оттуда.
- Проверьте, что данные доступны обоим сервисам.

---

5. Задание 5. Масштабирование и наблюдаемость

- Добавьте директиву для масштабирования `model-service` (в секцию сервиса `model`):

```
deploy:
  replicas: 2
```

- Либо запустите вручную:

```
docker compose up --scale model=2
```

- Проверьте балансировку нагрузки — отправьте серию запросов и посмотрите логи обеих реплик.
- Добавьте healthcheck в сервис `model`:

```
healthcheck:
  test: ["CMD", "curl", "-f", "http://localhost:5000/health"]
  interval: 10s
  retries: 3
```

- Подключите сервисы `prometheus` и `grafana` (официальные образы) и соберите метрики latency и CPU.

---

6. Задание 6. Интеграционный тест

- Напишите простой тест-клиент (например, `test_pipeline.py`):

```
import requests

def test_pipeline():
    r = requests.post(
        "http://localhost:5003/finalize",
        files={"file": open("test.jpg", "rb")},
    )
    assert r.status_code == 200
```

- Добавьте в Compose дополнительный сервис `tests`, который запускает этот скрипт после развёртывания остальных контейнеров.

- Запустите тестовую проверку:

```
docker compose up --build --abort-on-container-exit
```

---

7. Задание 7. Анализ производительности

- Замерьте среднюю задержку:

```
time curl -X POST http://localhost:5003/finalize -F "file=@test.jpg"
```

- Сравните результаты при:
  - последовательном вызове сервисов по HTTP;
  - передаче данных через общий том `/data`.
- Сделайте вывод о влиянии сетевых задержек и кэширования на итоговый throughput системы.

---

8. Задание 8. Остановка и очистка

- Остановите все контейнеры:

```
docker compose down
```

- Удалите тома и сеть, если требуется полный сброс:

```
docker compose down -v
```
