Лекция 5: От контейнера к системе

Переход от изолированной модели к многосервисной архитектуре машинного обучения

От одного контейнера к экосистеме

01

Контейнеризация

Упаковка модели в воспроизводимое окружение с фиксированными зависимостями

02

Композиция

Препроцессинг, инференс, постобработка — каждый компонент требует собственного окружения

03

Оркестрация

Управление жизненным циклом множества взаимодействующих контейнеров

Проблема монолитного подхода

Ограничения монолита

- Невозможность неравномерного масштабирования
- Сложность обновления отдельных компонентов
- Конфликты зависимостей библиотек
- Потеря воспроизводимости

Микросервисное решение

Каждый компонент живёт в собственном контейнере с независимым жизненным циклом, версиями и зависимостями

$$f_{\text{system}}(x) = f_{\text{post}}(f_{\text{model}}(f_{\text{pre}}(x)))$$

Контейнер как атом вычисления

Препроцессинг

Нормализация, ресайз, преобразование в тензор

Инференс

Выполнение обученной модели

Постобработка

Декодирование, фильтрация, формирование ответа

Атомы соединяются через сети, тома и контракты интерфейсов, образуя устойчивую систему

От изоляции к оркестрации

1

Docker CLI

Ручной запуск контейнеров, императивное управление

2

Docker Compose

Декларативное описание системы в YAML

3

Kubernetes

Промышленная кластерная оркестрация

Воспроизводимость на уровне системы

Оркестрация обеспечивает детерминированное отображение конфигурации в исполняемую среду:

$$\Psi : \mathcal{C} \to \mathcal{E}$$

где $\mathcal{C}$ — множество Compose-файлов, $\mathcal{E}$ — множество исполняемых окружений

Один и тот же docker-compose.yml всегда порождает идентичную сеть взаимодействий при одинаковых версиях образов

Математическая модель системы

Совокупная задержка многосервисной системы:

$$L_{\text{system}} = \sum_{i=1}^{n} L_i + L_{\text{net}}$$

Пропускная способность ограничена минимальным звеном:

$$T_{\text{system}} = \min_i T_i$$

где L_i — задержка сервиса i, T_i — его throughput

Трёхсервисный пайплайн инференса

01

preprocess-service

Получает изображение, выполняет нормализацию и ресайз, преобразует в тензор

02

model-service

Загружает модель ONNX и выполняет инференс

03

postprocess-service

Вычисляет метки, фильтрует результаты, возвращает JSON

Сервисы взаимодействуют по именам: `http://model:5000`, http://preprocess:5000

Пример Compose-конфигурации

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

  model:
    build: ./model
    ports: ["5002:5000"]

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

Запуск системы одной командой: docker compose up --build

Формализация оркестрации

Оркестрация как отображение декларации в исполняемую среду:

$$\Phi : (D_1, D_2, \ldots, D_n, N, V) \longrightarrow E_{\text{system}}$$

где каждый контейнер D_i определяется параметрами:

$$D_i = (\text{image}_i, \text{cmd}_i, \text{ports}_i, \text{volumes}_i, \text{env}_i)$$

N — виртуальная сеть, V — набор томов

Архитектура Docker Compose

Декларативность

Описание желаемого состояния системы, а не последовательности команд

Детерминизм

Один YAML-файл всегда создаёт идентичную среду

Воспроизводимость

Инфраструктура как код, версионируемая в Git

От императивного к декларативному

Императивный подход

```
docker build -t preprocess .
docker run -d --name model model_image
docker network create my_net
docker run --network my_net preprocess_image
```

Порядок действий критичен, ошибки неявны

Декларативный подход

```
services:
  preprocess:
    build: ./preprocess
  model:
    build: ./model
```

Описание состояния, автоматическая инициализация

Структура docker-compose.yml

1

version

Версия схемы Compose (например, "3.9")

2

services

Определение контейнеров, образов, портов и зависимостей

3

networks

Описание виртуальных сетей

4

volumes

Определение общих томов для данных и моделей

Граф зависимостей: depends_on

Директива `depends_on` формирует ориентированный ациклический граф $G = (V, E)$

Порядок инициализации вычисляется как топологическая сортировка:

$$\pi = \text{toposort}(G)$$

> `depends_on` гарантирует порядок запуска, но не готовность сервиса. Используйте healthchecks для проверки доступности

Healthchecks для надёжности

```
model:
  build: ./model
  healthcheck:
    test: ["CMD", "curl", "-f", "http://localhost:5000/health"]
    interval: 5s
    retries: 5
```

Зависимые сервисы запускаются только после успешной проверки готовности

Система становится самовосстанавливающейся

Единое сетевое пространство

Bridge-сеть

Compose автоматически создаёт виртуальную сеть для всех контейнеров проекта

DNS-резолвинг

Каждый сервис доступен по имени: `http://model:5000`

Изоляция

Проекты изолированы друг от друга, безопасность по умолчанию

DNS как отображение

Внутренний DNS Compose создаёт отображение имён в IP-адреса:

$$\mathrm{DNS}_{\mathrm{compose}} : S_i \mapsto \mathrm{IP}_i$$

Имя сервиса остаётся неизменным при перезапусках, хотя IP может меняться

Это обеспечивает устойчивость к сбоям и делает код переносимым между машинами

Тома: постоянное хранилище

Назначение томов

- Хранение весов моделей между перезапусками
- Обмен промежуточными результатами
- Сохранение логов и метрик
- Кэширование вычислений

Том как отображение:

$$V_k : P_{\text{host}} \leftrightarrow P_{\text{container}}$$

Пример конфигурации томов

```
volumes:
  model_data:

services:
  model:
    volumes:
      - model_data:/models

  preprocess:
    volumes:
      - model_data:/models:ro
```

Флаг `:ro` — доступ только на чтение для preprocess

Модель обновляется в одном месте, остальные сервисы используют её без копирования

Переменные окружения

```
environment:
  - MODEL_PATH=/models/resnet50.onnx
  - LOG_LEVEL=info
```

Параметризация системы без пересборки контейнеров

Чувствительные данные хранятся в `.env` и не попадают в репозиторий

Воспроизводимость Compose

Состояние системы как множество параметров:

$$S = \{D_i, N, V, E\}$$

Docker Compose гарантирует тотальное детерминированное отображение:

$$\Psi : Y \to S$$

где Y — YAML-описание. Любой корректный YAML определяет единственное состояние S

Построение пайплайна инференса

Логическая топология системы

Граф взаимодействий:

$$G = (V, E)$$

$$V = \{\text{preprocess}, \text{model}, \text{postprocess}\}$$

$$E = \{(\text{preprocess}, \text{model}), (\text{model}, \text{postprocess})\}$$

Каждое ребро соответствует HTTP-вызову REST-эндпоинта

Реализация preprocess-service

```
from fastapi import FastAPI, UploadFile
import requests, numpy as np
from PIL import Image

app = FastAPI()

@app.post("/process")
async def process_image(file: UploadFile):
 image = Image.open(file.file).resize((224, 224))
 arr = np.asarray(image).astype(np.float32) / 255.0
 payload = {"tensor": arr.tolist()}
 r = requests.post("http://model:5000/predict", json=payload)
 return r.json()
```

Реализация model-service

```
from fastapi import FastAPI, Request
import onnxruntime as ort, numpy as np

app = FastAPI()
session = ort.InferenceSession("model.onnx")

@app.post("/predict")
async def predict(req: Request):
 data = await req.json()
 x = np.array(data["tensor"], dtype=np.float32)[None, ...]
 output = session.run(None, {"input": x})[0]
 return {"result": output.tolist()}
```

Dockerfile для сервисов

```
FROM python:3.10-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
EXPOSE 5000
CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "5000"]
```

Каждый сервис собирается независимо с собственными зависимостями

Анализ маршрута данных

Последовательность преобразований:

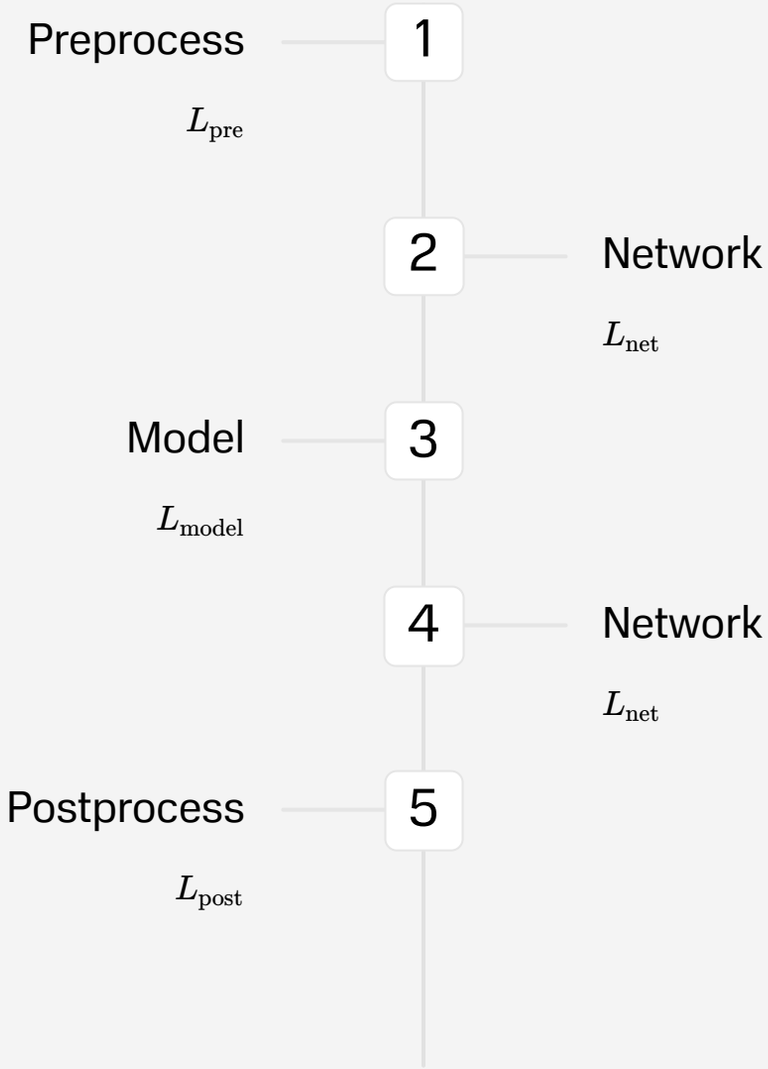
$$x_{\text{raw}} \xrightarrow{f_{\text{pre}}} x_{\text{tensor}} \xrightarrow{f_{\text{model}}} y_{\text{raw}} \xrightarrow{f_{\text{post}}} y_{\text{final}}$$

Время передачи между сервисами:

$$L_{ij}^{\text{net}} = \frac{B_i}{R_{\text{net}}} + L_{\text{handshake}}$$

где B_i — объём данных, R_{net} — пропускная способность сети

Временная диаграмма запроса



Суммарная латентность складывается из времён вычислений и передачи

Throughput системы

Пропускная способность в steady-state:

$$T_{\text{system}} = \frac{1}{L_{\text{system}}}$$

При использовании очереди запросов throughput ограничен минимальным звеном:

$$T_{\text{system}} = \min(T_{\text{pre}}, T_{\text{model}}, T_{\text{post}})$$

Использование общих томов

```
volumes:
  shared_data:

services:
  preprocess:
    volumes:
      - shared_data:/shared
  model:
    volumes:
      - shared_data:/shared
```

Взаимодействие через файловую систему сокращает сетевую задержку

Коэффициент эффективности кэша: $\eta_{\text{cache}} = \frac{T_{\text{net}}}{T_{\text{volume}}}$

Самовосстановление системы

```
model:
  build: ./model
  healthcheck:
    test: ["CMD", "curl", "-f", "http://localhost:5000/health"]
    interval: 10s
    timeout: 5s
    retries: 3
  restart: always
```

Автоматический перезапуск при сбоях обеспечивает непрерывную эксплуатацию

Интеграционное тестирование

```
import requests

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

Тест включается в CI-конвейер для проверки целостности системы

Идемпотентность сервисов

Повторный вызов с теми же параметрами должен давать тот же результат:

$$\forall x, \quad f_{\text{system}}(x) = f_{\text{system}}(x)$$

Достигается через:

- Фиксированные RNG seeds
- Отключение стохастических операций
- Контроль версий библиотек
- Кэширование результатов

Формула производительности

Совокупная производительность системы:

$$T_{\text{system}} = \left(\sum_{i=1}^{n} \frac{1}{T_i} \right)^{-1}$$

Надёжность системы:

$$R_{\text{system}} = \prod_{i=1}^{n} R_i$$

где $R_i = 1 - p_i, p_i$ — вероятность сбоя сервиса i

Свойства пайплайна инференса

Воспроизводимость

Один Compose-файл определяет всё окружение

Изолированность

Каждый сервис живёт в своём контейнере

Композиционность

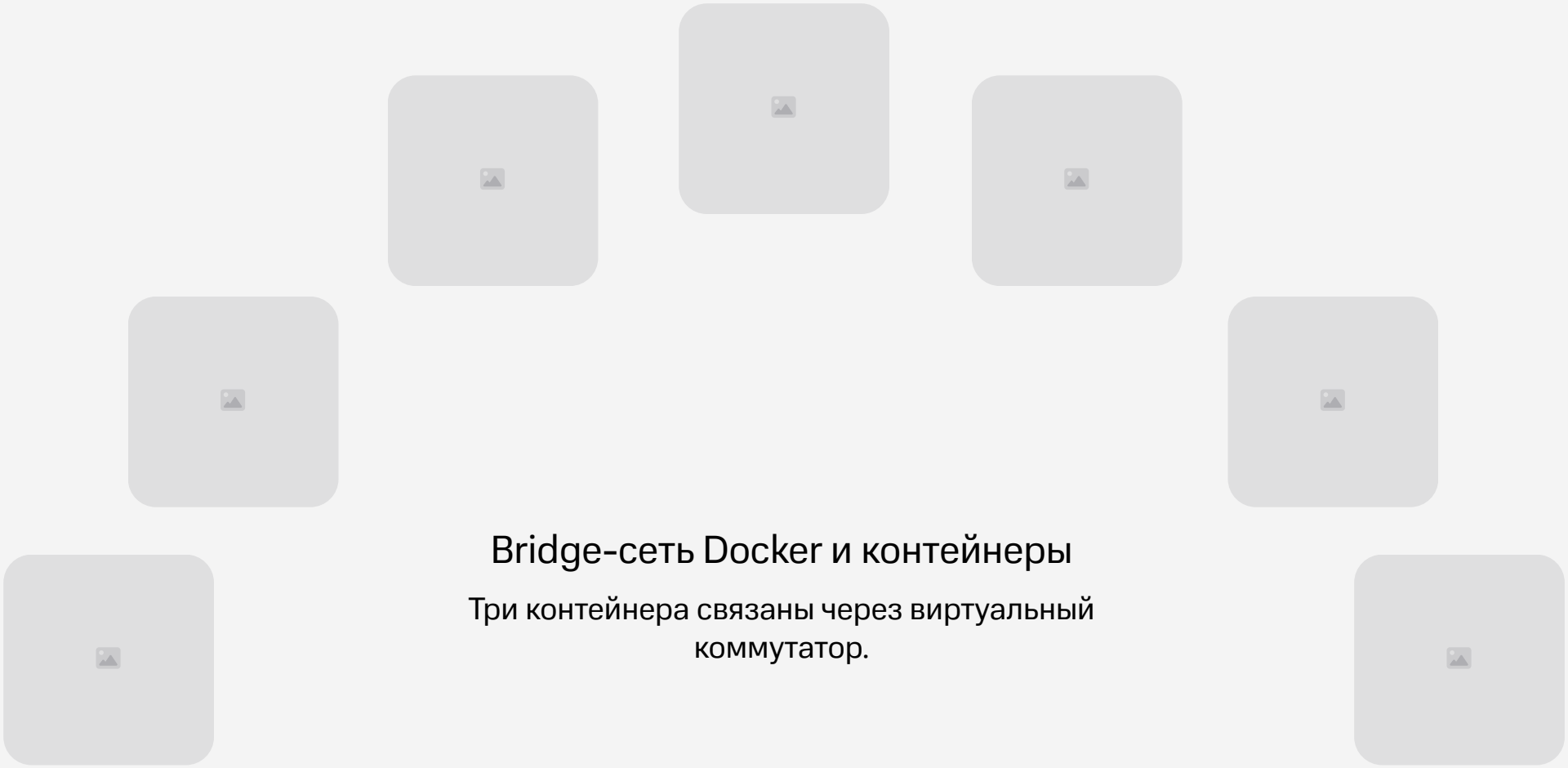
Взаимодействие через стабильные API

Масштабируемость

Независимое масштабирование сервисов

Сетевое взаимодействие

Сеть — главный канал взаимодействия и потенциальный источник проблем в микросервисной системе



Bridge-сеть Docker и контейнеры

Три контейнера связаны через виртуальный коммутатор.

Модель виртуальной сети

Граф контейнеров:

$$G = (V, E)$$

$$V = \{S_1, S_2, \ldots, S_n\}, \quad E = \{(S_i, S_j, \mathrm{REST}_{ij})\}$$

Bridge-сеть создаёт подсеть (например, 172.20.0.0/16), где все контейнеры получают IP-адреса

Пропускная способность близка к скорости RAM — несколько Гбит/с

DNS-резолвинг по именам

Автоматическое создание внутреннего DNS:

$$\mathrm{DNS}_{\mathrm{Compose}} : S_i \rightarrow \mathrm{IP}_i(t)$$

Обращение к сервису по имени: `http://model:5000/`

IP-адрес может меняться при перезапуске, но имя остаётся неизменным — это обеспечивает устойчивость к сбоям

Внутренние и внешние сети

```
networks:
  internal_net:
    internal: true
  external_net:
    driver: bridge

services:
  model:
    networks:
      - internal_net
  api_gateway:
    networks:
      - internal_net
      - external_net
```

Разделение повышает безопасность: только API-шлюз доступен извне

Настройка портов

Маппинг портов

```
ports:
  - "8080:80"
```

Обращения к `localhost:8080` перенаправляются в контейнер на порт 80

Ограничение доступа

```
ports:
  - "127.0.0.1:9000:9000"
```

Порт доступен только с локальной машины

Безопасность сетей

Изоляция по умолчанию

Bridge-сеть изолирует проекты друг от друга

Внутренние сети

Флаг `internal: true` блокирует внешний доступ

Принцип наименьших привилегий

Публикуется только необходимый минимум портов

Диагностика сети

```
docker compose ps
docker compose exec model ping preprocess
docker compose exec model curl http://postprocess:5000/health
```

Проверка связности графа: если между вершинами существует маршрут, система связна

Отсутствие связи указывает на ошибку конфигурации

Сетевые метрики

Время передачи пакета между контейнерами:

$$t_p = t_{\text{serialization}} + t_{\text{bridge}} + t_{\text{deserialization}}$$

Типичные значения: 0.1–0.3 мс для JSON-запроса размером 10 КБ

Время передачи данных размером B байт:

$$L_{\text{net}} = \frac{B}{R_{\text{net}}}$$

Граф взаимодействий

Система как ориентированный граф $G = (V, E)$, где вершины — сервисы, рёбра — каналы связи

Параметры системы:

$$L_{\text{total}} = \sum_{i} L_i + \sum_{(i,j) \in E} L_{ij}^{\text{net}}$$

$$T_{\text{total}} = \min_{i} T_i$$

$$R_{\text{total}} = \prod_{i} R_i$$

Ограничения bridge-сети

1

Bridge

Все контейнеры на одном хосте, удобно для разработки

2

Overlay

Распределённые сети через VXLAN-туннели в Swarm/Kubernetes

Bridge — упрощённый прототип overlay-сети для локальной работы

Принципы сетевого взаимодействия

1 **DNS-имена**

Каждому сервису соответствует имя, доступное в пределах проекта

2 **Разделение сетей**

Внутренние и внешние сети должны быть изолированы

3 **Минимальная публикация**

Только API-gateway обычно доступен извне

4 **Высокая скорость**

Bridge обеспечивает производительность близкую к RAM

Томовые хранилища

Тома — постоянные хранилища для данных, разделяемые между контейнерами

Том как отображение путей:

$$V : P_{\text{host}} \leftrightarrow P_{\text{container}}$$

Изменения внутри контейнера отражаются на хосте и наоборот

Типы данных в ML-системах

Статические артефакты

Веса моделей, конфигурации, словари

Динамические результаты

Выходы промежуточных стадий, кэшированные тензоры

Журналы и метрики

Логи инференса, статистика, временные отчёты

Типы томов Docker

Named volumes

Управляются Docker, хранятся в `/var/lib/docker/volumes`, для постоянных данных

Anonymous volumes

Создаются при запуске, удаляются с контейнером, для временных файлов

Bind mounts

Прямое подключение каталога хоста, удобно для разработки

Общий том для моделей

```
volumes:
  models_storage:

services:
  model:
    volumes:
      - models_storage:/models

  preprocess:
    volumes:
      - models_storage:/models:ro
```

Разделяемое состояние: $D_{\text{shared}} = \{x \in P_{\text{models}} \mid \forall S_i \in V_{\text{access}} : S_i(x) = x\}$

Обмен через файловую систему

Preprocess

```
import numpy as np
np.save("/data/input.npy", tensor)
```

Model

```
import numpy as np
x = np.load("/data/input.npy")
```

Коэффициент эффективности кэша:

$$\eta_{\text{cache}} = \frac{T_{\text{without\_cache}}}{T_{\text{with\_cache}}}$$

Согласованность и конкурентный доступ

Условие согласованности томов:

$$\forall i, j \in V_{\text{access}}, \quad |f_i(x,t) - f_j(x,t)| < \epsilon$$

Предотвращение гонки данных через блокировки:

```
from filelock import FileLock

with FileLock("/data/input.npy.lock"):
 np.save("/data/input.npy", tensor)
```

Структура томов в проекте

/models

Хранилище обученных моделей и весов

/data

Временные данные инференса

/logs

Файлы журналов и метрик

/cache

Кэш промежуточных вычислений

Внешние источники данных

```
volumes:
  shared_nfs:
    driver_opts:
      type: "nfs"
      o: "addr=10.0.0.1,rw"
      device: ":/mnt/shared"
```

Подключение NFS или облачного хранилища для распределённого доступа

Версионирование томов

Содержимое томов должно версионироваться через checksum:

$$\text{hash}_{\text{model}} = \text{SHA256}(W)$$

где W — веса модели

Том становится носителем версии артефакта, управляемым через Model Registry, Git LFS или DVC

Производительность томов

Время доступа к файлу:

$$T_{\text{IO}} = T_{\text{seek}} + T_{\text{read}} + T_{\text{write}}$$

Для временных файлов до 2 ГБ используйте tmpfs (RAM-диск):

```
tmpfs:
  size: 2g
```

Радикальное сокращение задержек, но данные теряются при перезапуске

Заключение: от контейнера к системе

1	3	∞
Контейнер	Сервисы	Масштабирование
Атомарная единица вычисления	Препроцессинг, модель, постобработка	Независимое управление каждым компонентом

Docker Compose — мост между локальной инженерией и промышленной оркестрацией