Контейнеризация и упаковка ML-моделей с Docker

Лекция 4. Современные подходы к эксплуатации моделей машинного обучения через контейнерные технологии

Введение: зачем контейнеризация в ML-эксплуатации

Когда инженер машинного обучения завершает обучение модели — будь то ResNet-50 для распознавания изображений или трансформер для обработки текста — он получает не готовый эксплуатационный продукт, а **исследовательский результат**. Это лишь набор весов θ, функций и зависимостей, воспроизводимый только в среде, где модель была обучена.

Чтобы формализовать понятие воспроизводимости, введём обозначение окружения E — множества версий библиотек, системных зависимостей, драйверов и аппаратных параметров. Тогда вычислительный процесс можно записать как функцию:

$$f(x; \theta, E)$$

где x — входные данные, θ — параметры модели, E — среда исполнения. Для корректного переноса модели должно выполняться условие воспроизводимости:

$$f_{\text{prod}}(x; \theta, E) = f_{\text{train}}(x; \theta, E_0)$$

От виртуальных машин к контейнерам

Виртуальные машины

Исторически первой попыткой изоляции окружения стали **виртуальные машины (VM)**. Они позволяли запускать отдельную операционную систему поверх основной, создавая изолированный контур.

$$E_{\text{VM}} = (E_{\text{Host}}, E_{\text{Guest}})$$

Такой подход решал задачу совместимости, но имел существенный изъян: избыточность. Каждая виртуальная машина содержала полную копию ядра ОС, занимала гигабайты памяти и требовала десятки секунд на запуск.

Контейнеры

Переход к контейнеризации стал следующим шагом эволюции. **Контейнер** — это лёгкая форма виртуализации, в которой несколько сред разделяют одно ядро хоста, но имеют независимые пространства имён и ресурсов.

$$C_i = (P_i, F_i, N_i, M_i)$$

где Pi — пространство процессов, Fi — файловая система, Ni — сетевые интерфейсы, Mi — монтированные устройства.

Проблема зависимости окружений в ML

Модели машинного обучения особенно чувствительны к окружению. Рассмотрим пример: исследователь обучил модель ResNet-50 в PyTorch 2.1 на CUDA 12.1 с cuDNN 8.9. Он сохранил веса .pth и передал коллеге, который запустил их на PyTorch 2.0 и CUDA 11.8.

> Внешне код тот же, но поведение модели изменилось: ядра cuDNN вызывают другие реализации свёрток, что даёт расхождение логитов порядка 10-2.

Эта ситуация типична. И она показывает, что артефакт модели состоит не только из весов, но и из кода, движка и окружения:

$$A = (f, \theta, R, E)$$

где f — структура вычислительного графа, θ — параметры, R — runtime (ONNX Runtime, TensorRT, PyTorch Runtime), а E — среда. Потеря хотя бы одного компонента нарушает детерминизм вычислений.

> Модель без фиксированного окружения — это гипотеза, а не продукт.

Контейнер как формализация среды

Docker дал возможность впервые записать эксплуатационное окружение модели как самодостаточный артефакт. Контейнер объединяет код, зависимости, веса и интерпретатор в единый образ, переносимый на любое устройство с установленным Docker-движком.

Если обозначить сборку образа функцией Build, а операцию запуска — Φ, то:

$$E_{\text{prod}} = \Phi(\text{Build}(\text{code}, \text{deps}, \text{config}))$$

Это означает, что эксплуатационная среда Eprod восстанавливается детерминированно из исходных инструкций. Проверка равенства становится экспериментальной процедурой контроля воспроизводимости:

$$f_{\text{train}}(x; \theta, E_0) \approx f_{\text{prod}}(x; \theta, \Phi(D))$$

Контейнеризация и инженерная воспроизводимость

Детерминизация вычислений

Docker позволяет описать среду декларативно: все версии, зависимости, команды установки фиксируются в Dockerfile. Это превращает среду из непредсказуемого контекста в строго определённый объект.

Reproducible build

Процесс сборки можно представить как функцию свёртки, где hash(D) — уникальный идентификатор образа, обеспечивающий детерминированность сборки.

$$D = \mathrm{Build}(\mathrm{source}, \mathrm{deps}, \mathrm{runtime}) \Rightarrow \mathrm{hash}(D)$$

Криптографическая идентичность

Повторная сборка с теми же входными данными должна давать тот же хэш. Это свойство является математическим выражением воспроизводимости среды.

От виртуального окружения к контейнеру

Виртуальное окружение

Виртуальное окружение фиксирует лишь слой Python-зависимостей:

$$E_{\text{virtualenv}} = \{\text{Python}, \text{packages}\}$$

Это решает проблему изоляции пакетов, но не гарантирует бинарную совместимость при переносе между системами.

Docker-контейнер

Контейнер фиксирует весь стек:

$$E_{\text{Docker}} = \{\text{OS}, \text{Python}, \text{CUDA}, \text{cuDNN}, \text{deps}\}$$

Следовательно, Evirtualenv $\subset$ EDocker, и только контейнер гарантирует бинарную совместимость при переносе.

Контейнеризация как смена парадигмы

В контексте курса «Эксплуатация ML-моделей» переход к контейнеризации — это смена инженерной парадигмы: от гибкости к детерминированности.

1

Исследователь

Работает в пространстве гипотез — его цель эксперимент. Гибкость и быстрая итерация важнее воспроизводимости.

2

Инженер эксплуатации

Работает в пространстве контрактов — его цель воспроизводимость. Стабильность и детерминизм важнее гибкости.

Контейнеризация позволяет впервые сформулировать эксплуатацию как математически проверяемый процесс:

$$\exists D : f_{\text{prod}}(x; \theta, \Phi(D)) = f_{\text{train}}(x; \theta, E_0), \forall x \in D_{\text{val}}$$

Это равенство становится определением корректной упаковки модели.

Практическая перспектива

Возьмём простую модель в формате ONNX и скрипт инференса на Flask. Запуск такого кода в чистом окружении Python зависит от установленных пакетов, версии numpy, наличия onnxruntime и даже компилятора OpenBLAS.

Python-скрипт

```
import onnxruntime as ort
from flask import Flask, request, jsonify
import numpy as np

app = Flask(__name__)
session = ort.InferenceSession("model.onnx")

@app.route("/predict", methods=["POST"])
def predict():
    x = np.array(request.json["input"],
                 dtype=np.float32)
    y = session.run(None, {"input": x})
    return jsonify({"output": y[0].tolist()})
```

Dockerfile

```
FROM python:3.10-slim
COPY model.onnx /app/model.onnx
COPY server.py /app/server.py
WORKDIR /app
RUN pip install flask onnxruntime
EXPOSE 8080
CMD ["python", "server.py"]
```

Теперь этот образ можно запустить на любой машине, и результаты инференса будут идентичны независимо от хоста.

Архитектура Docker и механизм контейнеров

Чтобы понять, почему Docker стал основным инструментом упаковки моделей, нужно рассмотреть его внутреннее устройство. Docker — это не просто утилита для сборки образов, а **движок виртуализации на уровне операционной системы**, основанный на системных механизмах Linux: namespaces, cgroups, overlayfs и capabilities.

Изоляция

Процессы контейнера изолированы от хоста и других контейнеров через пространства имён.

Воспроизводимость

Образы детерминированы и могут быть воссозданы с идентичным хэшем.

Управляемость ресурсов

Контроль CPU, памяти и GPU через cgroups обеспечивает предсказуемую производительность.

Концептуальная схема Docker

Логически Docker можно рассматривать как трёхуровневую систему:

01

Образы (images)

Статические шаблоны, описывающие файловую систему и инструкции для создания контейнера.

02

Контейнеры (containers)

Запущенные экземпляры образов, внутри которых выполняется процесс.

03

Демон Docker

Управляющий процесс на хосте, обеспечивающий взаимодействие с ядром Linux и управление жизненным циклом контейнеров.

В формализованной записи образ можно рассматривать как функцию сборки:

$$D = \mathrm{Build}(B, L_1, L_2, \ldots, L_n)$$

где B — базовый слой, а Li — последовательность добавленных слоёв. Контейнер есть динамическое отображение образа:

$$C(t) = \Phi(D, E_{\mathrm{host}}, R(t))$$

Пространства имён (Namespaces)

Ключевым механизмом изоляции в Docker являются **namespaces** — системные области, разделяющие глобальные ресурсы Linux. Каждый контейнер получает собственное пространство имён, и процессы внутри него «видят» только свои объекты.

PID namespace

Изолирует идентификаторы процессов. Процесс PID 1 внутри контейнера — не тот же, что на хосте.

NET namespace

Отделяет сетевые интерфейсы и таблицы маршрутизации. Каждый контейнер имеет собственный виртуальный адаптер eth0.

MNT namespace

Управляет точками монтирования файловых систем.

UTS namespace

Задаёт собственные имена хоста (hostname).

IPC namespace

Отделяет объекты межпроцессного взаимодействия.

USER namespace

Позволяет маппинг пользователей контейнера в пользователей хоста.

$$E_{\text{container}} = \{N_{\text{PID}}, N_{\text{NET}}, N_{\text{MNT}}, N_{\text{IPC}}, N_{\text{UTS}}, N_{\text{USER}}\}$$

Контроль ресурсов: cgroups

Контейнеры используют ресурсы общего ядра, но их потребление должно быть ограничено. Для этого применяется механизм **cgroups (control groups)**, который устанавливает квоты и приоритеты по CPU, памяти, вводу/выводу и GPU.

Каждый контейнер имеет собственное поддерево в иерархии cgroups, и ядро Linux отслеживает использование ресурсов каждым процессом. Например, при запуске контейнера с параметром:

```
docker run --cpus=2 --memory=4g my_model:1.0
```

Docker создаёт для него cgroup с лимитами CPU и памяти. Если процесс выходит за границы лимита, ядро применяет механизм OOM-killer или throttle.

Формально управление можно описать как оптимизационную задачу:

$$\forall C_i : R(C_i) \leq Q_i$$

где R(Ci) — текущий вектор использования ресурсов контейнера, а Qi — заданные квоты.

Файловые системы и OverlayFS

Контейнеры Docker основаны на принципе **слоистой файловой системы** (overlayfs). Каждый слой представляет собой снимок изменений относительно предыдущего. Верхний слой — «записываемый» (writable layer), в котором хранятся изменения, сделанные контейнером во время работы.

Если рассмотреть структуру образа как последовательность слоёв L_1, L_2, ..., L_n, то файловая система контейнера формируется операцией наложения:

$$F_{\text{container}} = L_1 \oplus L_2 \oplus \ldots \oplus L_n$$

где $\oplus$ обозначает монтирование слоёв поверх друг друга с помощью overlayfs. При чтении файл ищется сверху вниз: если он есть в верхнем слое, нижние не просматриваются. При записи создаётся копия в верхнем слое (механизм *copy-on-write*).

Эта структура делает Docker-образы компактными и быстрыми: общий базовый слой python:3.10-slim может быть использован десятками контейнеров без дублирования.

Сеть и взаимодействие контейнеров

Docker создаёт собственный уровень сетевой абстракции через **bridge network**. По умолчанию каждый контейнер получает виртуальный интерфейс, подключённый к мосту docker0 на хосте. Этот мост действует как программный свитч, маршрутизируя пакеты между контейнерами и внешним интерфейсом.

bridge

Стандартная сеть с NAT, подходящая для локальных сервисов

host

Контейнер использует сетевой стек хоста (максимальная производительность, но без изоляции)

none

Без сети, используется для офлайн-задач

custom networks

Пользовательские сети позволяют управлять связями между микросервисами

Слои образа и кэширование

Каждая инструкция Dockerfile создаёт новый слой. Если инструкция не изменилась, Docker использует кэш предыдущей сборки. Это делает процесс **инкрементальным и воспроизводимым**.

Пример Dockerfile

```
FROM python:3.10-slim
RUN pip install onnxruntime flask
COPY model.onnx /app/model.onnx
COPY inference.py /app/inference.py
CMD ["python", "inference.py"]
```

Все зависимости (pip install) будут выполняться только один раз, пока не изменится requirements.txt.

Хэш-функция слоёв

С точки зрения воспроизводимости, можно рассматривать каждый слой как хэш-функцию:

$$h_i = \mathrm{SHA256}(L_i)$$

$$D = \{h_1, h_2, \ldots, h_n\}$$

Суммарный хэш образа H(D) служит его уникальным идентификатором.

Различие между образом и контейнером

Образ (Image)

Статический шаблон, аналог бинарного пакета. Это функция, определяющая структуру среды.

Образ неизменяем и может быть использован для создания множества контейнеров.

Контейнер (Container)

Запущенный экземпляр, процесс с собственным PID и состоянием. Это результат применения функции образа.

Контейнер имеет жизненный цикл: создание, запуск, остановка, удаление.

Можно сказать, что образ — это функция, а контейнер — результат её применения:

$$C = \Phi(D) = \mathrm{Run}(D, \mathrm{env})$$

где Φ — оператор развёртывания, а env — конкретное окружение запуска (порты, тома, переменные).

Математическая модель контейнерной изоляции

Если формализовать процесс запуска, можно записать модель взаимодействия контейнеров и хоста. Пусть Ω — множество ресурсов хоста (CPU, RAM, GPU, сеть, файловая система), и П = {C_1, C_2, ..., C_k} — множество контейнеров.

Тогда для каждого контейнера выполняется:

$$R(C_i) \subseteq \Omega, \quad R(C_i) \cap R(C_j) = \emptyset \text{ для } i \neq j$$

Это свойство **непересечения ресурсов** гарантируется cgroups и namespaces. Контейнер видит только свою часть мира Ω. Однако ядро Linux остаётся общим, что обеспечивает высокую производительность и низкие накладные расходы по сравнению с виртуальными машинами, где создаётся Ωi ≈ Ω.

Docker как формальная реализация воспроизводимой среды

Docker позволяет перейти от неформального понятия «окружение» к математически контролируемому объекту. Если рассматривать среду как отображение:

$$E = (O, D, P)$$

где O — операционная система, D — зависимости, P — параметры исполнения, то Docker реализует отображение:

$$\Psi : (O, D, P) \mapsto H(D)$$

присваивая хэш-идентификатор, который делает окружение адресуемым и сравнимым.

> Два инженера, построив одинаковые Dockerfile, получат идентичные хэши и, следовательно, гарантированно эквивалентные среды.

В этом и заключается суть контейнеризации как технологии научной и инженерной воспроизводимости.

Архитектура Docker: заключение

Архитектура Docker — это не просто набор системных инструментов, а формализованная модель среды, в которой каждая компонента выполняет роль в обеспечении детерминизма.

Слои файловой системы

Обеспечивают компактность и переиспользование базовых образов

Namespaces

Изолируют процессы, сеть и файловую систему контейнеров

OverlayFS

Реализует эффективное наложение слоёв с copy-on-write

Cgroups

Контролируют потребление CPU, памяти и GPU

Для ML-инженера это означает: каждый контейнер — это зафиксированная копия среды инференса, каждый слой образа — шаг воспроизводимого билда, каждый хэш образа — гарантия идентичности модели между машинами.

Построение Dockerfile для ML-моделей

Контейнеризация модели машинного обучения начинается с создания **Dockerfile** — декларативного сценария, описывающего шаги формирования образа. Этот файл представляет собой инженерный контракт, фиксирующий все зависимости, настройки окружения и способ запуска модели.

Именно он превращает модель из исследовательского прототипа в воспроизводимый эксплуатационный артефакт.

Семантика Dockerfile

Dockerfile — это последовательность инструкций, каждая из которых определяет изменение в файловой системе будущего контейнера. Каждая инструкция создаёт новый слой образа, который сохраняется с собственным хэшем.

Формально Dockerfile можно рассматривать как функцию:

$$D = F(I_1, I_2, \ldots, I_n)$$

где Ii — инструкции сборки (FROM, RUN, COPY, CMD, и т. д.). Эта функция детерминирована: если последовательность Ii и входные данные (файлы, пакеты, версии) неизменны, то хэш-идентификатор образа H(D) также неизменен.

Это свойство обеспечивает **reproducible build** — воспроизводимую сборку, критически важную для эксплуатации ML-моделей.

Базовые образы

Первое ключевое решение при построении контейнера — выбор **базового образа**. Он определяет систему, интерпретатор и часто включает в себя CUDA, cuDNN и предустановленные библиотеки.

python:3.10-slim

Минимальный Python-образ на Debian, подходит для CPU-инференса

pytorch/pytorch

Образы от PyTorch с предустановленным CUDA и cuDNN

onnxruntime/onnxruntime

Образы для инференса ONNX-моделей

nvidia/cuda

Используется, если требуется максимальный контроль над CUDA и драйверами

Выбор базового слоя влияет на размер и переносимость. Чем меньше слой, тем быстрее сборка, но тем больше зависимостей нужно устанавливать вручную.

Модель как часть образа

Ключевая инженерная идея — модель должна быть частью контейнера, а не внешним файлом. Если модель копируется командой:

```
COPY model.onnx /app/model.onnx
```

она становится частью слоя, и хэш образа зависит от содержимого файла. Это гарантирует, что версия контейнера всегда соответствует версии модели. Если модель обновилась, хэш слоя изменится, и Docker создаст новый образ.

Эта связь между моделью и образом делает возможным контроль версий на уровне Docker Registry:

$$D_{\text{model}}^{(v)} = \text{Build}(\text{Dockerfile}, \text{model}^{(v)})$$

и каждая версия v фиксируется как тэг (my_model:1.0, my_model:1.1, и т. д.).

Добавление зависимостей

При установке зависимостей в контейнере важно избегать избыточности. Слишком большое количество пакетов увеличивает размер образа и время сборки.

❌ Неправильно

```
RUN pip install torch torchvision \
    onnxruntime flask numpy pandas \
    scikit-learn
```

Такой подход не фиксирует версии и может привести к установке несовместимых пакетов.

✅ Правильно

```
COPY requirements.txt /app/requirements.txt
RUN pip install -r requirements.txt
```

Если в requirements.txt указаны версии (onnxruntime==1.19.0), то окружение фиксируется полностью, и последующие сборки дают идентичный результат.

CMD и ENTRYPOINT: интерфейс инференса

Контейнер — это не просто архив, а **исполняемая среда**. Его интерфейс к внешнему миру задают команды CMD и ENTRYPOINT.

CMD

Определяет команду по умолчанию, которую можно переопределить при запуске

```
CMD ["python", "inference.py",
    "--batch", "32"]
```

ENTRYPOINT

Фиксирует точку входа, которая всегда выполняется

```
ENTRYPOINT ["python",
            "inference.py"]
```

Для ML-инференса часто используется Flask или FastAPI-сервер. В промышленной среде предпочтительнее gRPC-сервисы или REST API с возможностью мониторинга.

Инженерные принципы минимизации размера

Размер Docker-образа напрямую влияет на время сборки и доставки в облако. Образ весом 3–4 ГБ затрудняет CI/CD и развертывание. Поэтому применяются три принципа:

01

Multi-stage builds

В первом этапе используется тяжёлый образ с компиляцией, во втором — минимальный runtime.

02

Удаление временных файлов

После установки зависимостей следует очищать кеши:

```
RUN pip install -r requirements.txt && \
    rm -rf /root/.cache/pip
```

03

Использование slim-баз

Например, python:3.10-slim вместо полного python:3.10.

Такой подход уменьшает размер финального образа в 2–3 раза.

Контроль GPU-доступа

Если модель требует ускорения на GPU, Docker поддерживает доступ к графическим устройствам через **NVIDIA Container Toolkit**. При запуске используется флаг --gpus all:

```
docker run --gpus all -p 8080:8080 my_model:1.0
```

Docker автоматически монтирует драйверы и устройства GPU в контейнер. Однако версии CUDA и драйверов должны быть согласованы: версия CUDA внутри контейнера не может превышать версию драйвера хоста.

Формально можно записать условие совместимости:

$$\mathrm{CUDA}_{\mathrm{container}} \leq \mathrm{Driver}_{\mathrm{host}}$$

Несоблюдение этого равенства вызывает ошибки "CUDA driver version is insufficient for CUDA runtime". Поэтому инженер эксплуатации обязан фиксировать эти зависимости в документации артефакта.

Переменные окружения и конфигурация

Чтобы избежать жёстко закодированных путей и параметров, параметры конфигурации задаются через **ENV** и **ARG**.

Определение в Dockerfile

```
ENV MODEL_PATH=/app/model.onnx
ENV PORT=8080
```

Эти переменные становятся частью образа и доступны всем процессам внутри контейнера.

Переопределение при запуске

```
docker run -e PORT=9090 my_model:1.0
```

Это делает контейнер гибким, но при этом воспроизводимым: базовая структура остаётся той же.

Пример полного Dockerfile

Приведём пример типового Dockerfile для контейнера инференса модели на ONNX Runtime с Flask-сервером:

```
# Базовый образ с поддержкой Python и CUDA
FROM pytorch/pytorch:2.1.0-cuda12.1-cudnn8-runtime

# Устанавливаем рабочую директорию
WORKDIR /app

# Копируем файлы модели и сервера
COPY model.onnx /app/model.onnx
COPY inference.py /app/inference.py
COPY requirements.txt /app/requirements.txt

# Устанавливаем зависимости
RUN pip install --no-cache-dir -r requirements.txt

# Определяем переменные окружения
ENV MODEL_PATH=/app/model.onnx
ENV PORT=8080

# Открываем порт для REST API
EXPOSE 8080

# Запуск сервера
CMD ["python", "inference.py"]
```

Такой Dockerfile фиксирует все ключевые параметры среды: версию CUDA, Python, зависимости, модель и способ запуска. Его хэш будет уникален для конкретной версии артефакта.

Проверка воспроизводимости контейнера

После сборки образа выполняется тест воспроизводимости. Для этого контейнер запускают на разных хостах и сравнивают результаты инференса:

```
docker build -t my_model:1.0 .
docker run --rm my_model:1.0 python -c \
  "import inference; print(inference.test())"
```

Если разность результатов Δ ≤ 10-3 по L2-норме, окружение считается эквивалентным:

$$\Delta = |f_{\text{host1}}(x) - f_{\text{host2}}(x)|_2 \leq 10^{-3}$$

Этот тест фиксируется в CI/CD как контроль воспроизводимости контейнера.

Инженерные соображения при создании Dockerfile

Создание Dockerfile — это не просто копирование инструкций, а проектирование среды, отвечающей требованиям эксплуатации:

1

Детерминированность

Каждый слой фиксирует версию пакета

2

Воспроизводимость

Сборка должна быть идентичной на разных машинах

3

Минимализм

Ничего лишнего не должно попасть в образ

4

Безопасность

Контейнер не должен работать от root и содержать уязвимые пакеты

> Dockerfile — это, по сути, декларативная спецификация среды, аналог математического описания E = {OS, CUDA, deps, config}, превращённого в исполняемый объект.

Воспроизводимость и версионирование окружения

После того как контейнер собран и протестирован локально, возникает следующий этап инженерной зрелости — **контроль воспроизводимости** и **версионирование эксплуатационного окружения**.

Контейнеризация создаёт самодостаточную среду, но без строгого управления версиями она превращается в статичный артефакт, не встроенный в жизненный цикл модели.

В промышленной эксплуатации воспроизводимость означает не только возможность собрать образ повторно, но и гарантировать, что любая версия модели может быть восстановлена в точности с теми зависимостями, которые использовались при её деплое год назад.

Понятие reproducible build

Reproducible build — это процесс сборки, при котором результат полностью определяется входными данными: исходным кодом, зависимостями, аргументами и конфигурацией. Если эти данные не изменились, хэш итогового образа должен быть идентичен.

Формально воспроизводимость можно записать как:

$$\text{Build}(S, D, C) = \text{Build}(S, D, C) \Rightarrow H(D_1) = H(D_2)$$

где S — исходный код и модель, D — зависимости, C — конфигурация сборки, а H(D) — хэш результата. Если хотя бы один из элементов изменился, Docker создаёт новый слой, и хэш образа меняется.

Это свойство делает возможным **детерминированную историю окружений**, что особенно важно при расследовании инцидентов и аудите.

Теги и версии образов

В Docker каждый образ помечается тегом (tag), который является символьным именем конкретного состояния. Например: my_model:1.0, my_model:1.1, my_model:latest

Тег — это человекочитаемый ярлык, но под ним всегда скрыт конкретный SHA256-хэш слоя. Хэш выступает в роли **истинного идентификатора** артефакта:

```
docker images --digests
REPOSITORY  TAG   DIGEST
my_model    1.0   sha256:ab94f3b...
```

Если собрать тот же Dockerfile с теми же файлами, мы получим точно такой же хэш. Таким образом, пара (tag, digest) образует контракт между версией модели и её окружением.

$$E^{(v)} = \mathrm{Build}(\mathrm{Dockerfile}^{(v)}, \mathrm{model}^{(v)}, \mathrm{deps}^{(v)})$$

Immutable builds

В промышленной эксплуатации контейнеры рассматриваются как **неизменяемые артефакты** (*immutable builds*). Это означает, что после сборки образ не редактируется: любые изменения — даже правка зависимостей — требуют сборки нового образа с новым тегом.

1	2
my_model:1.0	my_model:1.1
Исходная версия модели с фиксированными зависимостями	Обновлённая версия с новыми зависимостями или кодом

Такой подход предотвращает «дрейф окружения» — скрытые изменения, которые могут нарушить воспроизводимость. Старый образ остаётся в реестре для воспроизведения предыдущего состояния.

Математически это означает, что функция сборки становится инъективной:

$$\text{Build} : (S, D, C) \mapsto H(D)$$

Контроль зависимостей и версий

Чтобы обеспечить воспроизводимость, необходимо фиксировать версии библиотек. В Docker это реализуется через три механизма:

1 Фиксация версий пакетов в requirements.txt

```
onnxruntime==1.19.0
numpy==1.26.0
flask==3.0.0
```

Это гарантирует, что при повторной сборке установятся те же бинарные пакеты.

2 Закрепление версии базового образа

```
FROM python:3.10.14-slim
```

Если использовать python:latest, воспроизводимость нарушается — базовый слой обновляется автоматически.

3 Логирование хэшей сборки

Docker сохраняет SHA256 каждого слоя. Эти хэши включают версии всех пакетов и файлов, что делает возможной проверку идентичности среды.

Docker Registry и хранение артефактов

Чтобы контейнеры можно было использовать в разных средах, они публикуются в **Docker Registry** — централизованном хранилище артефактов. Это может быть публичный Docker Hub или частный реестр компании (например, в GitLab, Yandex Cloud, AWS ECR).

Процесс публикации включает две команды:

```
docker tag my_model:1.0 registry.company.ru/ml/my_model:1.0
docker push registry.company.ru/ml/my_model:1.0
```

После загрузки реестр хранит метаданные образа — его хэш, размер, дату, зависимости и владельца.

В CI/CD такая публикация становится обязательным шагом: каждый успешный билд должен заканчиваться загрузкой в реестр, а версия контейнера связывается с конкретным коммитом Git и версией модели.

Проверка идентичности контейнеров

Docker позволяет проверить, что два контейнера построены из одного образа. Для этого сравниваются хэши слоёв:

```
docker inspect --format='{{.Image}}' container_id
```

Если SHA256 идентичен, контейнеры функционально эквивалентны. Для более строгого контроля создаются **манифесты сборки**, в которых фиксируются хэши файлов и пакетов:

```
docker run my_model:1.0 pip freeze > manifest.txt
```

При повторной сборке манифест сравнивается построчно, чтобы убедиться в идентичности версий.

Математически это проверка равенства множеств зависимостей:

$$M(E_1) = M(E_2)$$

где M(E) — отображение окружения в список версий библиотек.

Управление версиями CUDA и драйверов

Одним из наиболее уязвимых мест при воспроизводимости является сочетание CUDA и драйверов NVIDIA. В Docker существует строгое соотношение между ними: контейнер использует **CUDA runtime**, а драйвер устанавливается на хосте.

Совместимость задаётся условием:

$$v_{\text{driver}}^{\text{host}} \geq v_{\text{cuda}}^{\text{container}}$$

Если это условие нарушается, инференс на GPU становится невозможным. Поэтому инженеры эксплуатации фиксируют пары (CUDA, Driver) в документации артефакта.

> В крупных компаниях создаются стандартизированные образы — *base CUDA images* — которые обеспечивают стабильность всей экосистемы.

Практика версионирования в CI/CD

В системах непрерывной интеграции версии контейнеров формируются автоматически. Типичный конвейер CI/CD для ML-моделей включает этапы:

01

Build

Сборка Docker-образа

02

Test

Проверка инференса и метрик точности

03

Tag

Генерация версии образа, например по схеме 1.0.3-build.472

04

Push

Загрузка в Docker Registry

05

Deploy

Развёртывание контейнера в тестовую или продакшн-среду

Версия контейнера связывается с коммитом модели:

$$\text{model\_version} = f(\text{git\_commit}, \text{docker\_tag}, \text{hash})$$

Проверка детерминированности сборки

Для критически важных систем (например, медицинские или финансовые модели) проверка воспроизводимости выполняется формально: сборка проводится на двух независимых машинах, а полученные хэши сравниваются.

```
docker build -t build1 .
docker build -t build2 .
docker inspect build1 --format='{{.Id}}'
docker inspect build2 --format='{{.Id}}'
```

Если идентификаторы совпадают, сборка детерминирована. Если нет — анализируется, какие зависимости привели к расхождению.

Формально критерий детерминизма записывается как:

$$H(\text{Build}_A) = H(\text{Build}_B) \Rightarrow \text{Reproducible}$$

Верификация среды и контроль целостности

На заключительном этапе контейнер проходит проверку целостности. Docker использует SHA256 для каждого слоя, поэтому изменение даже одного байта в файле модели приводит к новому хэшу. Это свойство обеспечивает **криптографическую идентичность окружения**.

Инженеры эксплуатации часто хранят контрольные суммы в отдельном артефактном файле:

```
sha256sum model.onnx > checksum.txt
```

При деплое система CI/CD проверяет, что контрольная сумма совпадает с эталонной. Если нет — сборка блокируется.

Формально контроль целостности можно выразить как требование:

$$\forall f \in E, \quad \mathrm{SHA256}(f_{\mathrm{deploy}}) = \mathrm{SHA256}(f_{\mathrm{build}})$$

Инференс внутри контейнера

На предыдущих этапах мы говорили о Docker как о средстве упаковки и воспроизводимости окружения. Теперь настало время рассмотреть, **что происходит внутри контейнера, когда модель начинает выполнять инференс**.

Этот раздел объединяет системное и ML-инженерное понимание: как процесс инференса протекает в среде, изолированной Docker'ом, и как измеряется производительность, воспроизводимость и эффективность вычислений.

Внутренний цикл инференса

Когда контейнер запущен, Docker создаёт для него отдельное пространство имён, монтирует файловую систему, запускает процесс, указанный в CMD, и передаёт ему управление. Для ML-моделей этот процесс обычно представляет собой Python-сервер (например, Flask или FastAPI), который инициализирует сессию инференса и начинает обрабатывать входные запросы.

1. **Инициализация среды**
   Загрузка модели (ONNXRuntime.InferenceSession, torch.jit.load, trt.Runtime)
2. **Принятие входных данных**
   Через REST/gRPC
3. **Предобработка входа**
   Нормализация, ресайз
4. **Выполнение инференса**
   Вычисление выходов модели
5. **Постобработка и возврат результата**
   Форматирование и отправка ответа

$$\forall x \in D, \quad f_{\text{container}}(x; \theta, E_{\text{container}}) = f_{\text{train}}(x; \theta, E_0)$$

ONNX Runtime как ядро инференса

Внутри контейнера обычно используется **ONNX Runtime (ORT)** — кроссплатформенный движок инференса, поддерживающий CPU, GPU и специализированные ускорители. Он оптимизирует граф, управляет памятью и исполняет модель на уровне низкоуровневых библиотек (cuDNN, MKL-DNN, TensorRT).

При инициализации создаётся объект сессии:

```
import onnxruntime as ort
session = ort.InferenceSession("model.onnx",
    providers=["CUDAExecutionProvider"])
```

ONNX Runtime формирует вычислительный план: граф ONNX разбивается на узлы, распределяемые по провайдерам исполнения. Например, GPU выполняет свёртки и матричные операции, CPU — нелинейности и логические операции.

Процесс можно рассматривать как композицию вычислительных операторов:

$$f_{\mathrm{ORT}}(x) = o_k \circ o_{k-1} \circ \ldots \circ o_1(x)$$

Управление ресурсами CPU/GPU внутри контейнера

Docker предоставляет средства для жёсткого контроля над вычислительными ресурсами. С помощью флагов --cpus, --memory, --gpus можно задать квоты, которые будут применены через cgroups.

Пример:

```
docker run --cpus=2 --memory=4g --gpus all my_model:1.0
```

В этом случае контейнер видит только два CPU и 4 ГБ памяти. Для GPU используется интеграция с NVIDIA Container Runtime, который передаёт устройства /dev/nvidia* в пространство контейнера.

Внутренние ограничения можно интерпретировать как ограничения на вектор ресурсов:

$$R = [r_{\text{CPU}}, r_{\text{RAM}}, r_{\text{GPU}}]$$

и задача эксплуатации — минимизировать латентность L(R) при фиксированных ресурсах.

Измерение latency и throughput

Две ключевые метрики производительности модели в контейнере — **latency** (время обработки одного запроса) и **throughput** (число запросов в секунду). Эти параметры измеряются с помощью профилирования внутри контейнера.

Пример кода для измерения

```
import time, numpy as np
x = np.random.randn(1, 3, 224, 224).astype(np.float32)
N = 100
start = time.time()
for _ in range(N):
    _ = session.run(None, {"input": x})
end = time.time()
latency = (end - start) / N * 1000
throughput = N / (end - start)
print(f"Latency: {latency:.2f} ms")
print(f"Throughput: {throughput:.2f} inferences/s")
```

Формулы

$$L = \frac{T_{\text{end}} - T_{\text{start}}}{N}$$

$$T = \frac{N}{T_{\text{end}} - T_{\text{start}}}$$

Эти значения используются для оценки эффективности контейнера при разных конфигурациях.

Параллелизм и batch-инференс

Контейнеризация позволяет масштабировать инференс горизонтально (через оркестрацию) и вертикально (через батчирование). **Batch size** напрямую влияет на latency и throughput. При увеличении батча снижается относительная задержка ввода-вывода, но растёт время отклика для одного запроса.

Зависимость между throughput и batch size можно аппроксимировать как:

$$T(n) = \frac{n}{a + bn}$$

где a — фиксированные накладные расходы на инициализацию, b — время обработки одного элемента. Задача инженера — подобрать n*, при котором T(n) максимален при допустимом L(n).

В контейнере этот параметр задаётся через переменные окружения или конфигурационный файл модели.

Мониторинг потребления ресурсов

Docker позволяет наблюдать использование CPU, памяти и GPU в реальном времени с помощью команды:

```
docker stats
```

Вывод:

```
CONTAINER ID  NAME      CPU %   MEM USAGE / LIMIT  GPU MEM USAGE
a1f6c03b1f    my_model  35.2%   1.8GiB / 4GiB      230MiB / 8192MiB
```

Эти показатели помогают обнаружить узкие места. Для более глубокого анализа применяют nvidia-smi (в GPU-контейнерах) или встроенные средства профилирования ONNX Runtime (enable_profiling=True).

Профиль производительности экспортируется в JSON:

```
session.start_profiling("profile.json")
session.run(None, {"input": x})
session.end_profiling()
```

Профиль показывает, какие операции занимают основное время — типично это Conv, Gemm и Softmax.

Сравнение инференса внутри и вне контейнера

Контейнеризация неизбежно добавляет небольшие накладные расходы, связанные с изоляцией процессов и файловой системы. Однако в случае Docker эти расходы минимальны — обычно не более 1–2 % при работе на GPU и до 5 % на CPU.

Это можно формально выразить как отношение производительности:

$$\eta = \frac{T_{\text{container}}}{T_{\text{native}}} \approx 0.98$$

98%

Эффективность GPU

Производительность контейнера относительно нативного запуска на GPU

95%

Эффективность CPU

Производительность контейнера относительно нативного запуска на CPU

Таким образом, контейнеризация практически не влияет на скорость инференса, но радикально повышает управляемость и воспроизводимость.

Влияние ограничений ресурсов на производительность

Если контейнеру заданы жёсткие квоты CPU или памяти, производительность модели изменяется нелинейно. Например, при ограничении до одного ядра latency может вырасти пропорционально количеству потоков, используемых ONNX Runtime.

Эта зависимость описывается как:

$$L(c) \sim \frac{1}{\min(c, c_{\text{opt}})}$$

где c — количество доступных ядер, а copt — оптимальное число потоков. Для GPU-инференса зависимость слабее, но ограничения по памяти (--memory) могут приводить к выгрузке тензоров и снижению скорости.

Поэтому при проектировании контейнера важно указывать **разумные лимиты**, соответствующие характеристикам хоста.

Воспроизводимость численных результатов

Контейнеризация устраняет вариации версий библиотек, но для строгой детерминированности инференса необходимо также контролировать параметры случайности и аппаратные оптимизации.

Перед запуском инференса обычно фиксируют seed и отключают неустойчивые алгоритмы:

```
import torch, numpy as np, random
torch.manual_seed(0)
np.random.seed(0)
random.seed(0)
```

В ONNX Runtime добавляется параметр:

```
sess_options = ort.SessionOptions()
sess_options.enable_mem_pattern = False
sess_options.graph_optimization_level = \
    ort.GraphOptimizationLevel.ORT_DISABLE_ALL
```

Эти настройки обеспечивают числовое совпадение результатов на разных машинах до машинного эпсилона ε = 10-6.

Контейнеры в нагрузочном тестировании

После настройки инференса контейнер используется для **нагрузочного тестирования** — проверки стабильности модели под реальной нагрузкой. Для этого применяются инструменты ab (ApacheBench), wrk, locust или hey, посылающие параллельные запросы к REST API контейнера.

Пример:

```
hey -n 1000 -c 20 -m POST \
  -H "Content-Type: application/json" \
  -d '{"input": [0.1,0.2,0.3,...]}' \
  http://localhost:8080/predict
```

Результаты теста включают среднюю и максимальную задержку, процент ошибок и пропускную способность. При этом контейнер можно масштабировать горизонтально: запустить несколько экземпляров одного образа с балансировщиком запросов.

Переносимость контейнера между средами

Одна из важнейших черт контейнера — способность работать одинаково в разных средах: локально, на сервере, в облаке, в CI/CD, на GPU-кластере.

Docker реализует **абстракцию вычислительной среды**: образ содержит всё необходимое, кроме ядра и драйверов. Поэтому, если аппаратное обеспечение поддерживает требуемые версии CUDA, контейнер будет вести себя одинаково.

Формально переносимость можно записать как:

$$\forall E_1, E_2 \text{ совместимых}, f(x; \theta, \Phi(D, E_1)) = f(x; \theta, \Phi(D, E_2))$$

Это определение эксплуатационной инвариантности модели.

Практический пример инференса

Рассмотрим пример инференса контейнера my_model:1.0, работающего с ONNX Runtime и REST API.

```
import onnxruntime as ort
from flask import Flask, request, jsonify
import numpy as np, time

app = Flask(__name__)
session = ort.InferenceSession("model.onnx")

@app.route("/predict", methods=["POST"])
def predict():
    start = time.time()
    x = np.array(request.json["input"], dtype=np.float32)
    y = session.run(None, {"input": x})
    latency = (time.time() - start) * 1000
    return jsonify({
        "output": y[0].tolist(),
        "latency_ms": latency
    })
```

Контейнер запускается:

```
docker run -p 8080:8080 my_model:1.0
```

и обрабатывает запросы через curl. Такой контейнер является законченным сервисом, готовым к интеграции в микросервисную архитектуру.

Выводы: инференс внутри контейнера

Инференс внутри контейнера демонстрирует, что контейнеризация не ограничивается упаковкой. Это — форма инженерной детерминизации вычислений, обеспечивающая:

Полная управляемость ресурсов

Контроль CPU, памяти и GPU через cgroups

Измеримость производительности

Точные метрики latency и throughput

Числовая воспроизводимость

Идентичные результаты на разных машинах

Переносимость между средами

Одинаковое поведение локально и в облаке

> Docker превращает модель в **единицу вычислений с контрактом поведения**: заданное окружение гарантирует заданный результат.