# Архитектурная Парадигма Контейнеризации в.NET 8: Глубокий Анализ Теории Третьего Дня

## 1. Введение: Фундаментальная Трансформация Ментальных Моделей

Переход от разработки клиентских приложений на движке Unity к созданию высоконагруженных распределенных систем на платформе.NET Enterprise знаменует собой не просто смену инструментария, но и радикальную перестройку инженерного мышления. В центре этой трансформации, достигающей своего апогея на восемнадцатой неделе учебного плана, находится концепция доставки программного обеспечения. Если для разработчика игровых клиентов "билд" представляет собой монолитный, статичный артефакт — исполняемый файл или пакет (APK, EXE), который финализируется на машине разработчика и доставляется пользователю в неизменном виде, то в мире бэкенд-инженерии мы оперируем понятием "образа контейнера" как динамической, конфигурируемой единицы развертывания.1

Теория третьего дня посвящена глубокому изучению технологий контейнеризации с использованием Docker в контексте экосистемы.NET 8. Это не просто изучение синтаксиса Dockerfile, а погружение в архитектурные принципы создания иммутабельной инфраструктуры (Immutable Infrastructure). В отличие от игрового клиента, который работает на устройстве пользователя в единственном экземпляре, бэкенд-сервис функционирует во враждебной, постоянно изменяющейся среде облачного кластера, где экземпляры приложения могут создаваться и уничтожаться сотни раз в минуту в ответ на колебания трафика. В таких условиях требования к артефакту сборки кардинально меняются: на первый план выходят скорость инициализации, детерминизм окружения, минимизация вектора атаки и эффективность использования сетевых и дисковых ресурсов.1

Современный стандарт разработки, диктуемый методологией 12-Factor App, требует строгого разделения стадий сборки (Build), релиза (Release) и выполнения (Run). Контейнеризация является тем технологическим клеем, который обеспечивает это разделение. Однако, наивный перенос практик локальной разработки в контейнерную среду — например, копирование всего исходного кода в образ и компиляция внутри продакшен-контейнера — является критической архитектурной ошибкой, приводящей к созданию уязвимых, тяжеловесных и медленных систем. Цель данного документа — предоставить исчерпывающий теоретический базис для проектирования оптимизированных, безопасных и стандартизированных Docker-образов, соответствующих лучшим практикам 2025 года, включая использование дистрибутивов Ubuntu Chiseled, многоэтапных сборок (Multi-Stage Builds) и продвинутых стратегий кэширования зависимостей.2

Мы рассмотрим физику процесса сборки на уровне файловой системы OverlayFS, проанализируем анатомию базовых образов Microsoft, разберем механизмы инвалидации кэша при восстановлении NuGet-пакетов и обсудим новейшие стандарты безопасности, внедренные в.NET 8, такие как работа под непривилегированным пользователем по умолчанию. Этот документ призван стать настольной книгой для инженера, стремящегося превратить процесс сборки из рутины в инженерное искусство.

## 2. Физика Контейнеризации: От Монолита к Слоям

Понимание внутреннего устройства Docker-образов необходимо для написания эффективных инструкций сборки. В отличие от виртуальной машины, которая эмулирует аппаратное обеспечение и содержит полное ядро операционной системы, контейнер — это изолированный процесс, разделяющий ядро хост-системы, но имеющий собственное пространство имен (User Namespace, PID Namespace, Network Namespace) и корневую файловую систему.

### 2.1. Файловая Система UnionFS и Механизм Copy-on-Write

Ключевой концепцией, отличающей Docker-образ от простого архива с файлами, является слоистая архитектура, реализуемая через Union File System (обычно драйвер overlay2 на Linux). Образ состоит из серии неизменяемых (read-only) слоев. Каждая инструкция в Dockerfile (такая как COPY, RUN, ADD) создает новый слой, который представляет собой дельту изменений относительно предыдущего слоя.4

Когда контейнер запускается, Docker добавляет поверх стопки неизменяемых слоев образа один тонкий слой для записи (Container Layer). Все изменения, которые приложение производит во время работы (запись логов, временных файлов), происходят именно в этом слое. Если приложению необходимо изменить файл, находящийся в одном из нижних, неизменяемых слоев, драйвер файловой системы копирует этот файл в верхний слой (механизм Copy-on-Write) и уже там производит модификацию.

Эта архитектура имеет критические последствия для процесса сборки.NET приложений:

1. **Неизменность истории:** Если вы случайно скопировали секретный ключ или огромную папку bin/Debug с временными файлами в одном слое, а в следующем слое удалили их командой RUN rm -rf..., физически данные *остаются* в нижнем слое образа и могут быть извлечены злоумышленником. Размер образа при этом не уменьшается.
2. **Эффективность кэширования:** Docker кэширует каждый слой по отдельности. При повторной сборке Docker проверяет, изменились ли входные данные для конкретной инструкции (хэш файлов для COPY или строка команды для RUN). Если изменений нет, слой берется из кэша. Это фундаментальный принцип, на котором строится вся оптимизация времени сборки CI/CD пайплайнов.5

### 2.2. Контекст Сборки (Build Context) и .dockerignore

Процесс сборки начинается не с выполнения команд Dockerfile, а с подготовки контекста сборки. Docker CLI (клиент) упаковывает указанную директорию (обычно корень репозитория) и отправляет её демону Docker (серверу), который может находиться как на локальной машине, так и на удаленном узле сборки.

Для Unity-разработчика, привыкшего к наличию тяжелых папок Library, Temp или Assets, важно понимать, что в мире.NET также существуют директории, которые никогда не должны попадать в контекст сборки: bin, obj, .vs, .git. Если не исключить эти папки, демон Docker будет получать сотни мегабайт ненужных бинарных файлов перед началом каждого билда, что драматически замедляет процесс и может привести к непредсказуемому поведению (например, попытке запустить Windows DLL в Linux контейнере).

Файл .dockerignore работает аналогично .gitignore, но предназначен для демона Docker. Правильно настроенный .dockerignore является первым рубежом оптимизации.4

**Таблица 1: Рекомендуемая конфигурация .dockerignore для Enterprise.NET проектов**

| **Паттерн Исключения** | **Обоснование** | **Потенциальный Риск при Отсутствии** |
| --- | --- | --- |
| \*\*/.git | Исключает всю историю версий и метаданные Git. | Утечка истории коммитов, увеличение размера контекста, случайное попадание секретов из старых коммитов. |
| \*\*/.vs, \*\*/.idea | Исключает файлы конфигурации IDE (Visual Studio, Rider). | Загрязнение образа локальными настройками разработчика. |
| \*\*/bin, \*\*/obj | Исключает результаты локальной компиляции. | Конфликт архитектур (попытка запустить Windows-сборку в Linux), инвалидация кэша при каждом локальном билде. |
| \*\*/TestResults | Исключает отчеты о тестах. | Ненужное увеличение размера контекста. |
| \*\*/node\_modules | Исключает локальные JS-зависимости (если есть фронтенд). | Огромный размер контекста, проблемы с кросс-платформенной совместимостью нативных модулей. |
| docker-compose\*.yml | Исключает файлы оркестрации. | Не требуются внутри контейнера, могут содержать конфигурацию среды. |

Использование шаблонов, предоставляемых командой docker init, является хорошей отправной точкой, но для сложных решений требуется ручная настройка.7

## 3. Выбор Базового Образа: Фундамент Вашего Приложения

Инструкция FROM задает основу всего приложения. В экосистеме.NET 8 выбор базового образа перестал быть тривиальной задачей "взять последний available". Microsoft предоставляет широкий спектр образов, оптимизированных под различные сценарии использования, и ошибочный выбор на этом этапе может привести к проблемам с производительностью, безопасностью или совместимостью.2

### 3.1. Иерархия Образов Microsoft

Microsoft поддерживает три основных типа образов, каждый из которых предназначен для определенного этапа жизненного цикла приложения:

1. **SDK (mcr.microsoft.com/dotnet/sdk)**:
   * **Состав:** Содержит компилятор Roslyn, систему сборки MSBuild, CLI инструменты, полный набор библиотек разработки.
   * **Размер:** Очень большой (около 800 МБ - 1 ГБ).
   * **Назначение:** Исключительно для этапа сборки (Build Stage) и запуска тестов. Никогда не должен использоваться для запуска приложения в продакшене, так как наличие компилятора и SDK на боевом сервере — это подарок для хакера, позволяющий скомпилировать и запустить любой вредоносный код.5
2. **ASP.NET Core Runtime (mcr.microsoft.com/dotnet/aspnet)**:
   * **Состав:** Содержит среду выполнения.NET Runtime (CLR) и набор библиотек ASP.NET Core. Оптимизирован для веб-приложений.
   * **Размер:** Средний (около 200 МБ для Debian-версии).
   * **Назначение:** Стандартный выбор для запуска веб-сервисов и API.
3. **.NET Runtime (mcr.microsoft.com/dotnet/runtime)**:
   * **Состав:** Только CLR, без библиотек ASP.NET.
   * **Назначение:** Для консольных приложений, воркеров (Worker Services), демонов, не обрабатывающих HTTP-трафик.

### 3.2. Дилемма Дистрибутивов: Alpine vs. Debian vs. Ubuntu Chiseled

Выбор операционной системы, лежащей в основе базового образа, является критическим архитектурным решением.

#### 3.2.1. Alpine Linux: Ловушка Минимализма

Долгое время Alpine Linux считался золотым стандартом для контейнеров благодаря своему крошечному размеру (образ весит около 100 МБ). Однако, Alpine использует библиотеку musl libc вместо стандартной glibc, используемой в большинстве дистрибутивов Linux (Debian, Ubuntu, CentOS).

**Риски использования Alpine в Enterprise.NET:**

* **Совместимость:** Многие нативные библиотеки (например, драйверы БД, gRPC, криптография) компилируются под glibc. Запуск их на Alpine может требовать сложных манипуляций или приводить к падению производительности.
* **Глобализация (Globalization):** musl не имеет встроенной поддержки ICU (International Components for Unicode). Это означает, что операции с датами, строками, валютами и сортировкой будут работать некорректно или приложение упадет с ошибкой Process terminated. Couldn't find a valid ICU package installed on the system. Решением является либо ручная установка пакета icu-libs (что увеличивает размер образа), либо включение режима InvariantGlobalization, который отключает специфичное для культур форматирование.6 Для финансовых или международных приложений режим Invariant часто неприемлем.
* **Отладка:** Отсутствие привычных утилит затрудняет диагностику проблем внутри контейнера.

#### 3.2.2. Ubuntu Chiseled: Революция Безопасности

В.NET 8 Microsoft совместно с Canonical представили образы **Ubuntu Chiseled** ("Высеченные"). Это реализация концепции "Distroless" — образов, содержащих только само приложение и его прямые зависимости времени выполнения.3

**Преимущества Chiseled образов:**

* **Отсутствие Shell:** В образе физически отсутствуют /bin/sh и /bin/bash. Это делает невозможным выполнение произвольных команд через docker exec или через уязвимости класса Remote Code Execution (RCE). Злоумышленник не может "провалиться в консоль", потому что консоли нет.
* **Отсутствие Пакетного Менеджера:** Нет apt или apk. Нельзя доустановить хакерские утилиты (curl, nmap) внутри контейнера.
* **Non-Root по умолчанию:** Образы настроены на работу от непривилегированного пользователя.
* **Размер:** Сопоставим с Alpine (около 110 МБ), но при этом использует стандартную glibc, обеспечивая 100% совместимость с экосистемой Linux.

**Вывод:** Для новых Enterprise-проектов на.NET 8 использование образов 8.0-jammy-chiseled является наиболее предпочтительным выбором, сочетающим безопасность Distroless и совместимость Ubuntu.3

## 4. Стратегии Сборки: Многоэтапность и Оптимизация Кэша

Эффективный Dockerfile должен обеспечивать минимальное время сборки при внесении изменений в код. Это достигается за счет использования **Multi-Stage Builds** и агрессивного кэширования слоев восстановления зависимостей.

### 4.1. Многоэтапная Сборка (Multi-Stage Builds)

Эта техника позволяет использовать разные базовые образы для разных этапов создания артефакта в рамках одного Dockerfile. Мы используем тяжелый SDK-образ для компиляции кода, а затем копируем только полученные DLL-файлы в легкий Runtime-образ.

**Аналогия с GameDev:**

* **Build Stage:** Это редактор Unity. В нем есть все исходники, несжатые текстуры, инструменты импорта.
* Runtime Stage: Это скомпилированный APK. В нем нет редактора, только оптимизированные данные.  
  Мы никогда не отдаем игроку редактор Unity. Точно так же мы не должны отдавать в продакшен SDK-образ.2

### 4.2. Парадокс dotnet restore и Слоистое Кэширование

Операция dotnet restore (восстановление NuGet пакетов) является одной из самых ресурсоемких. Она требует анализа графа зависимостей и скачивания файлов из сети. В плохом Dockerfile эта операция выполняется при каждом изменении кода.

**Наивный (Неэффективный) Подход:**

Dockerfile

COPY..  
RUN dotnet restore  
RUN dotnet publish

В этом сценарии, если вы измените хотя бы один символ в файле Program.cs или даже в README.md, хэш слоя COPY.. изменится. Следовательно, кэш всех последующих слоев будет инвалидирован, и RUN dotnet restore запустится заново, скачивая все пакеты с нуля. Это убивает производительность CI/CD.14

Оптимизированный Подход (Layered Approach):

Идея состоит в том, чтобы скопировать только файлы описания зависимостей (.csproj, .sln, NuGet.config) до копирования остального исходного кода. Поскольку список зависимостей меняется гораздо реже, чем бизнес-логика, слой dotnet restore будет браться из кэша в большинстве сборок.

Dockerfile

# 1. Копируем только файлы проектов  
COPY  
# 2. Восстанавливаем зависимости (слой кэшируется надолго)  
RUN dotnet restore "MyService/MyService.csproj"  
# 3. Копируем остальной код  
COPY..  
# 4. Собираем (build) - этот слой будет пересобираться часто, но он быстр  
RUN dotnet build

### 4.3. Проблема Монорепозиториев и Инструмент dotnet-subset

В реальных проектах структура решения часто бывает сложной: десятки проектов, вложенные папки, общие библиотеки. Ручное перечисление всех COPY ProjectA.csproj... становится утомительным и подверженным ошибкам. Если вы добавите новый проект в Solution, но забудете добавить COPY в Dockerfile, сборка упадет.

Для решения этой проблемы сообщество разработало инструмент **dotnet-subset**. Он анализирует файл решения (.sln) и копирует во временную директорию только те файлы, которые необходимы для восстановления зависимостей (.csproj, .props, .targets, .sln), сохраняя при этом иерархию папок.16

**Пример использования dotnet-subset в Dockerfile:**

Dockerfile

# Этап подготовки исходников для restore  
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS restore-env  
WORKDIR /source  
COPY..  
# Устанавливаем инструмент  
RUN dotnet tool install --global dotnet-subset  
# Извлекаем подмножество файлов, необходимых для restore  
RUN dotnet subset restore MySolution.sln --root-directory /source --output /restore\_subset  
  
# Этап сборки  
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build  
WORKDIR /src  
# Копируем только подготовленную структуру  
COPY --from=restore-env /restore\_subset.  
# Выполняем restore (теперь он идеально кэшируется)  
RUN dotnet restore MySolution.sln  
# Копируем остальной код и собираем  
COPY..  
RUN dotnet publish -c Release -o /app/publish

Этот подход делает Dockerfile устойчивым к изменениям структуры проекта и гарантирует максимальную эффективность кэша.18

### 4.4. Кэширование за пределами слоев: BuildKit Cache Mounts

Даже с оптимизацией слоев, иногда dotnet restore всё же должен выполниться (например, при обновлении версии пакета). По умолчанию Docker каждый раз начинает с пустого локального кэша NuGet внутри контейнера. Современный движок сборки Docker BuildKit позволяет монтировать специальные кэш-директории, которые сохраняются между сборками разных образов на одной машине.

Синтаксис --mount=type=cache позволяет указать Docker, что определенная папка (/root/.nuget/packages) должна быть сохранена на хосте и переиспользована.

Dockerfile

RUN --mount=type=cache,id=nuget,target=/root/.nuget/packages \  
 dotnet restore "MyService.csproj"

Это работает аналогично локальной разработке: если пакет уже был скачан ранее (даже для другого проекта), он не будет скачиваться повторно. Это критически важно для CI-агентов, которые собирают множество разных микросервисов.19

## 5. Архитектура Безопасности: Запуск без Root

Безопасность контейнеров — это не опция, а фундаментальное требование для Enterprise систем. Исторически, процессы внутри контейнеров запускались от имени пользователя root (UID 0). Это создавало значительный риск: если злоумышленник находил уязвимость в приложении или рантайме, позволяющую "сбежать" из контейнера (Container Breakout), он оказывался в хост-системе с полными правами суперпользователя.

### 5.1. Парадигма Non-Root в.NET 8

Начиная с.NET 8, Microsoft радикально изменила политику безопасности по умолчанию. Теперь все официальные образы ASP.NET Core содержат предопределенного пользователя (обычно с именем app и UID 1654 или другим, в зависимости от ОС), и конфигурация рантайма оптимизирована для работы под этим пользователем.22

Это изменение влечет за собой ряд архитектурных последствий, которые необходимо учитывать при проектировании Dockerfile и docker-compose.yml.

### 5.2. Сдвиг Портов (Port Shift): 80 -> 8080

В операционных системах Linux порты с номерами ниже 1024 считаются *привилегированными*. Только процесс с правами root может начать слушать (bind) порт 80. Поскольку.NET 8 нацелен на работу без root, стандартный порт для ASP.NET Core был изменен с 80 на 8080.22

**Что это значит для разработчика:**

* Если в docker-compose.yml вы напишете ports: - "80:80", ваше приложение будет недоступно, так как внутри контейнера никто не слушает порт 80.
* Правильный маппинг теперь: ports: - "8080:8080" или ports: - "80:8080" (перенаправление внешнего 80 на внутренний 8080).
* Для изменения порта внутри контейнера (например, если этого требует корпоративный стандарт) необходимо использовать переменную окружения ASPNETCORE\_HTTP\_PORTS.

### 5.3. Проблема Прав Доступа к Томам (Volume Permission Hell)

Переход на non-root пользователя часто вызывает болезненные проблемы при локальной разработке с использованием Docker Compose, особенно при монтировании локальных директорий (Bind Mounts).

Типичный сценарий отказа:

Приложение пытается писать логи в папку /app/logs, которая смонтирована с хост-машины (-./logs:/app/logs). При запуске приложение падает с исключением UnauthorizedAccessException: Access to the path is denied.

**Анатомия проблемы:**

1. На хост-машине (особенно Linux) папка ./logs принадлежит текущему пользователю (например, UID 1000) или root.
2. Внутри контейнера процесс работает под пользователем app (UID 1654).
3. Файловая система Linux проверяет права: пользователь 1654 пытается писать в папку, принадлежащую пользователю 1000. Если права доступа не 777 (разрешено всем), операция блокируется.

**Архитектурные решения:**

* **Использование Docker Volumes:** Именованные тома (docker volume create) управляются самим Docker, который корректно инициализирует права доступа. Это предпочтительный способ для баз данных.
* **Настройка прав на хосте:** Можно изменить владельца папки на хосте (chown 1654:1654./logs), но это "грязное" решение, загрязняющее хост-систему.
* **Принудительный UID (для Dev):** В docker-compose.yml можно переопределить пользователя для среды разработки: user: "${UID}:${GID}", чтобы внутри контейнера процесс работал под ID вашего локального пользователя. Это обеспечивает совместимость прав доступа.25

## 6. Синтез: Эталонный Dockerfile для.NET 8

На основе проанализированных требований и лучших практик, мы можем сформулировать эталонный Dockerfile, который должен стать результатом практической работы 3-го дня. Этот файл демонстрирует применение всех изученных концепций: Multi-stage сборка, оптимизация кэша NuGet, использование dotnet-subset, BuildKit Cache Mounts, безопасный базовый образ и работа без root.

Dockerfile

# --- Stage 1: Подготовка (Restore Environment) ---  
# Используем SDK для вычисления графа зависимостей  
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS restore-env  
WORKDIR /source  
COPY..  
# Установка инструмента для оптимизации копирования проектов  
RUN dotnet tool install --global dotnet-subset  
# Создание минимального слепка репозитория для restore  
RUN /root/.dotnet/tools/dotnet-subset restore MySolution.sln \  
 --root-directory /source \  
 --output /restore\_subset  
  
# --- Stage 2: Сборка (Build Environment) ---  
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build  
WORKDIR /src  
  
# Копируем ТОЛЬКО файлы проектов из предыдущего этапа  
# Это гарантирует, что слой restore не будет инвалидирован изменениями в коде  
COPY --from=restore-env /restore\_subset.  
  
# Выполняем восстановление пакетов с использованием кэш-маунта  
# --mount=type=cache сохраняет пакеты на хосте между билдами  
RUN --mount=type=cache,id=nuget,target=/root/.nuget/packages \  
 dotnet restore MySolution.sln  
  
# Теперь копируем полный исходный код  
COPY..  
  
# Собираем и публикуем приложение  
# Используем --no-restore, так как мы уже сделали это  
# Используем Release конфигурацию  
WORKDIR "/src/MyService"  
RUN --mount=type=cache,id=nuget,target=/root/.nuget/packages \  
 dotnet publish "MyService.csproj" \  
 --configuration Release \  
 --no-restore \  
 --output /app/publish  
  
# --- Stage 3: Рантайм (Production Environment) ---  
# Используем Ubuntu Chiseled: малый размер, нет shell, нет root  
# Образ mcr.microsoft.com/dotnet/aspnet:8.0-jammy-chiseled  
FROM mcr.microsoft.com/dotnet/aspnet:8.0-jammy-chiseled AS final  
WORKDIR /app  
  
# Копируем скомпилированные артефакты из этапа build  
COPY --from=build /app/publish.  
  
# Явно декларируем порт (8080 для.NET 8)  
EXPOSE 8080  
ENV ASPNETCORE\_HTTP\_PORTS=8080  
  
# Переключаемся на безопасного пользователя (хотя в chiseled это дефолт)  
# Использование переменной APP\_UID делает код переносимым  
USER $APP\_UID  
  
# Определяем точку входа  
# Используем JSON-синтаксис массива для корректной передачи сигналов OS (SIGTERM)  
ENTRYPOINT

### 6.1. Детальный Анализ Инструкций

* **FROM... AS...**: Именование этапов (restore-env, build, final) позволяет легко ссылаться на них в командах COPY --from, делая скрипт читаемым.
* **dotnet tool install... dotnet-subset**: Интеграция инструмента прямо в пайплайн сборки устраняет необходимость ручного управления зависимостями копирования.16
* **RUN --mount=type=cache...**: Эта "магическая" строка превращает процесс сборки из "скачивания интернета" в мгновенную операцию локального копирования кэшированных пакетов.19
* **ENTRYPOINT ["dotnet",...]**: Использование формы массива (["exec", "arg"]) критически важно. Если использовать строковую форму (ENTRYPOINT dotnet MyService.dll), Docker запустит процесс через оболочку /bin/sh -c. Это приведет к тому, что сигналы остановки (например, когда Kubernetes пытается корректно завершить под) не будут доходить до приложения, и оно будет убиваться принудительно (SIGKILL) после таймаута, что может привести к потере данных или некорректному закрытию соединений с БД.28

## 7. Интеграция с CI/CD и Жизненный Цикл Образа

Понимание того, как создан Dockerfile, неразрывно связано с тем, как этот образ будет использоваться в CI/CD пайплайнах (тема 4-го дня), но фундамент закладывается именно здесь.

### 7.1. Тэгирование и Версионирование

В Unity "версия билда" часто зашивается в PlayerSettings. В Docker версионирование происходит снаружи, на уровне тэгов реестра (Registry).

Критически важно не использовать тег latest для релизов в продакшен. Тег latest является мутабельным — он перезаписывается с каждым новым пушем. Если вы деплоите myapp:latest, вы не знаете, какой именно код сейчас работает.

Правильная стратегия — использовать Immutable Tags, основанные на хэше коммита Git (SHA) или семантической версии (SemVer). Это обеспечивается на этапе CI, но Dockerfile должен быть готов к этому (не содержать хардкода версий внутри).2

### 7.2. Воспроизводимость (Reproducibility)

Использование детерминированных базовых образов (указание конкретной версии 8.0-jammy-chiseled вместо просто 8.0) и фиксация зависимостей через dotnet restore --locked-mode (с использованием lock-файлов NuGet) позволяет достичь Святого Грааля DevOps — **Reproducible Builds**. Это гарантия того, что если мы соберем образ из коммита месячной давности, мы получим бинарно идентичный (или функционально эквивалентный) результат, что критично для отладки старых версий, работающих у клиентов.

## 8. Заключение: От Кода к Инфраструктуре

Теория третьего дня знаменует собой качественный скачок в подготовке.NET инженера. Мы уходим от восприятия приложения как набора C# файлов к восприятию его как **инфраструктурного компонента**.

Навык написания оптимизированных Dockerfile имеет прямой экономический эффект:

* **Снижение Time-to-Market:** Ускорение билдов за счет кэширования позволяет разработчикам быстрее доставлять фичи.
* **Снижение Cloud Costs:** Маленькие (Chiseled) образы требуют меньше места в реестрах (ACR/ECR) и меньше трафика при скачивании на узлы кластера (Node Pulling).
* **Повышение Надежности:** Исключение "flaky" зависимостей (типа musl в Alpine) и использование non-root пользователей делает систему устойчивой к сбоям и атакам.

В следующих днях учебного плана эти контейнеры станут кирпичиками, из которых мы будем строить автоматизированные конвейеры CI/CD в GitHub Actions, но прочность всего здания зависит от качества этих кирпичей, заложенного сегодня.

### Рекомендуемые Материалы для Углубленного Изучения

1. **Документация Docker:** "Best practices for writing Dockerfiles" — Библия оптимизации слоев.
2. **Microsoft Learn:** "Containerize a.NET app" — официальные руководства по специфике.NET 8.2
3. **GitHub:** Репозиторий dotnet-subset — изучение исходного кода инструмента для понимания алгоритма фильтрации зависимостей.16
4. **Security Advisories:** Изучение базы данных уязвимостей (CVE) для сравнения профилей безопасности Ubuntu, Debian и Alpine.

#### Источники

1. План обучения CI/CD и конфигурации .NET неделя 13
2. Containerize an app with Docker tutorial - .NET - Microsoft Learn, дата последнего обращения: декабря 6, 2025, <https://learn.microsoft.com/en-us/dotnet/core/docker/build-container>
3. Announcing .NET Chiseled Containers - Microsoft Developer Blogs, дата последнего обращения: декабря 6, 2025, <https://devblogs.microsoft.com/dotnet/announcing-dotnet-chiseled-containers/>
4. 9 Tips for Containerizing Your .NET Application | Docker, дата последнего обращения: декабря 6, 2025, <https://www.docker.com/blog/9-tips-for-containerizing-your-net-application/>
5. Run an ASP.NET Core app in Docker containers - Microsoft Learn, дата последнего обращения: декабря 6, 2025, <https://learn.microsoft.com/en-us/aspnet/core/host-and-deploy/docker/building-net-docker-images?view=aspnetcore-10.0>
6. Creating smaller and more secure docker images for .NET core | by Jeroen Verhaeghe, дата последнего обращения: декабря 6, 2025, <https://medium.com/@jeroenverhaeghe/creating-smaller-and-more-secure-docker-images-for-net-core-8c74101e9027>
7. Add `dotnet new` template for `.dockerignore` file · Issue #51129 - GitHub, дата последнего обращения: декабря 6, 2025, <https://github.com/dotnet/sdk/issues/51129>
8. Official .NET Docker images - Microsoft Learn, дата последнего обращения: декабря 6, 2025, <https://learn.microsoft.com/en-us/dotnet/architecture/microservices/net-core-net-framework-containers/official-net-docker-images>
9. How to solve "Globalization Invariant Mode is not supported" without adding the ICU libraries on the docker container? - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/77896912/how-to-solve-globalization-invariant-mode-is-not-supported-without-adding-the>
10. Testing Chiseled Ubuntu container images with .NET - my tech ramblings, дата последнего обращения: декабря 6, 2025, <https://www.mytechramblings.com/posts/testing-chiseled-ubuntu-containers-with-dotnet/>
11. .NET Chiselled Ubuntu Container Images Now Generally Available - InfoQ, дата последнего обращения: декабря 6, 2025, <https://www.infoq.com/news/2023/12/dotnet-ubuntu-chiselled-ga/>
12. Announcement: New approach for differentiating .NET 8+ images · dotnet dotnet-docker · Discussion #4821 - GitHub, дата последнего обращения: декабря 6, 2025, <https://github.com/dotnet/dotnet-docker/discussions/4821>
13. Development workflow for Docker apps - .NET - Microsoft Learn, дата последнего обращения: декабря 6, 2025, <https://learn.microsoft.com/en-us/dotnet/architecture/microservices/docker-application-development-process/docker-app-development-workflow>
14. How should I optimize my DotNet docker instances? | by Ted Spence | CodeX - Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/codex/how-should-i-optimize-my-dotnet-docker-instances-129b0156cb4e>
15. Dockerfiles: Why a separate 'build' stage when dotnet publish handles it all - Reddit, дата последнего обращения: декабря 6, 2025, <https://www.reddit.com/r/dotnet/comments/1m21mmv/dockerfiles_why_a_separate_build_stage_when/>
16. nimbleways/dotnet-subset: A .NET tool that copies a subset of files from a repository to a directory. The tool is mainly used in Dockerfiles to optimize the docker build caching for "dotnet restore" instructions. - GitHub, дата последнего обращения: декабря 6, 2025, <https://github.com/nimbleways/dotnet-subset>
17. dotnet-subset 0.2.0 - NuGet, дата последнего обращения: декабря 6, 2025, <https://www.nuget.org/packages/dotnet-subset/0.2.0>
18. Docker build caching for .NET applications done right with dotnet-subset, дата последнего обращения: декабря 6, 2025, <https://blog.theodo.ma/docker-build-caching-for-dotnet-applications-done-right-with-dotnet-subset/>
19. Optimal Dockerfile for .NET ASP.NET Core | Container Builds - Depot.dev, дата последнего обращения: декабря 6, 2025, <https://depot.dev/docs/container-builds/optimal-dockerfiles/dotnet-aspnetcore-dockerfile>
20. Optimize cache usage in builds - Docker Docs, дата последнего обращения: декабря 6, 2025, <https://docs.docker.com/build/cache/optimize/>
21. Docker Samples - Nuget Caching - Why is this a separate build step in all the examples? #6123 - GitHub, дата последнего обращения: декабря 6, 2025, <https://github.com/dotnet/dotnet-docker/discussions/6123>
22. NET 8.0 Container Images Now Available · dotnet dotnet-docker · Discussion #4995 - GitHub, дата последнего обращения: декабря 6, 2025, <https://github.com/dotnet/dotnet-docker/discussions/4995>
23. Breaking change: New non-root 'app' user in Linux images - .NET - Microsoft Learn, дата последнего обращения: декабря 6, 2025, <https://learn.microsoft.com/en-us/dotnet/core/compatibility/containers/8.0/app-user>
24. Containerize .NET applications with .NET 8 - Red Hat Developer, дата последнего обращения: декабря 6, 2025, <https://developers.redhat.com/articles/2023/10/31/containerize-dotnet-applications-dotnet8>
25. Docker non-root access volume for `dotnet/runtime` image - permission denied, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/78661360/docker-non-root-access-volume-for-dotnet-runtime-image-permission-denied>
26. Docker non-root access volume for runtime image - C# - Answer Overflow, дата последнего обращения: декабря 6, 2025, <https://www.answeroverflow.com/m/1254715245226754129>
27. [Help Needed] How to mount a directory as a non-root user in a container, дата последнего обращения: декабря 6, 2025, <https://forums.docker.com/t/help-needed-how-to-mount-a-directory-as-a-non-root-user-in-a-container/141661>
28. The Complete Guide to Writing Dockerfiles for .NET Applications - C# Corner, дата последнего обращения: декабря 6, 2025, <https://www.c-sharpcorner.com/article/the-complete-guide-to-writing-dockerfiles-for-net-applications/>