# Архитектура и теория контейнеризации корпоративных систем на платформе.NET 8: Глубокий анализ для перехода к Enterprise-разработке

## 1. Введение: Смена парадигмы доставки и исполнения кода

Переход к пятнадцатой неделе учебного плана знаменует собой фундаментальный разрыв с практиками локальной разработки, характерными для индустрии Unity и десктопных приложений. Если предыдущие этапы обучения фокусировались на внутренней архитектуре приложения — управлении памятью, асинхронных паттернах и проектировании API, — то текущий модуль переносит фокус на среду исполнения. В контексте разработки Enterprise-бэкенда, и в частности сервиса «Календарь», понимание того, *где* и *как* исполняется код, становится столь же критичным, как и качество самого кода.1

Традиционная модель развертывания, при которой артефакт (исполняемый файл) переносится на сервер с надеждой на совпадение конфигураций, в современных распределенных системах считается устаревшей и опасной. Проблема «It works on my machine» (у меня все работает) является не просто анекдотичным сбоем, а следствием отсутствия изоляции зависимостей. В экосистеме.NET это усугубляется зависимостью от конкретных версий среды выполнения (CLR), системных библиотек (например, OpenSSL или ICU для глобализации) и конфигурации операционной системы. Контейнеризация решает эту проблему радикально: единицей поставки становится не код, а вся среда его исполнения, зафиксированная в неизменяемом (immutable) образе.1

Для инженера, стремящегося к уровню Senior, Docker перестает быть просто утилитой для запуска базы данных. Он становится инструментом управления ресурсами ядра Linux, механизмом изоляции процессов и основой для построения предсказуемых, идемпотентных систем. В данном отчете, составленном с позиции Senior DevOps Engineer, мы проведем декомпозицию технологий контейнеризации, опустимся на уровень системных вызовов ядра Linux и проанализируем специфику работы.NET 8 в ограниченных средах.1

### 1.1. Философия неизменяемой инфраструктуры (Immutable Infrastructure)

Центральной концепцией, лежащей в основе перехода к контейнерам, является неизменяемость. В классическом администрировании серверы часто «мутируют»: администраторы устанавливают обновления, меняют конфигурационные файлы и устанавливают патчи непосредственно на работающих машинах. Это приводит к «дрейфу конфигураций» (Configuration Drift), когда состояние сервера в продакшене начинает отличаться от тестового окружения, делая поведение системы непредсказуемым.

Контейнеризация навязывает подход, при котором контейнеры не обновляются «на лету». При необходимости изменения кода или конфигурации создается новый образ, и старый контейнер заменяется новым. Это гарантирует, что артефакт, прошедший CI/CD пайплайн и интеграционные тесты (которые будут внедряться в рамках этой недели), будет идентичен тому, что запускается в боевой среде. Для сервиса «Календарь» это означает, что любая зависимость — будь то библиотека для работы с PDF или драйвер PostgreSQL — должна быть явно декларирована в Dockerfile и стать частью образа.1

## 2. Анатомия изоляции: Пространства имен ядра Linux (Namespaces)

Существует распространенное заблуждение, что контейнер Docker — это легковесная виртуальная машина. Это фундаментально неверно. Виртуальная машина эмулирует аппаратное обеспечение и загружает полноценное ядро гостевой ОС. Контейнер же — это обычный процесс в операционной системе хоста, которому ядро «лжет» о том, что он является единственным процессом в системе. Эта иллюзия создается с помощью механизма пространств имен (Namespaces).4

Пространства имен разбивают глобальные ресурсы системы (такие как список процессов, сетевой стек, точки монтирования) на изолированные экземпляры. Процесс, помещенный в пространство имен, видит только те ресурсы, которые принадлежат этому пространству. Для.NET-разработчика понимание этих механизмов критично для отладки проблем взаимодействия сервисов и диагностики ошибок доступа.

### 2.1. PID Namespace: Изоляция процессов и проблема PID 1

В глобальной таблице процессов Linux каждый процесс имеет уникальный идентификатор (PID). Однако внутри нового PID Namespace процессы получают свою собственную нумерацию, начиная с 1. Процесс, который запускает контейнер (точка входа, ENTRYPOINT), получает PID 1 внутри этого контейнера, хотя на хосте он может иметь PID 15342.4

Роль PID 1 и обработка сигналов (Graceful Shutdown)

В Linux процесс с PID 1 (обычно systemd или SysVinit) имеет особую роль: он обязан «усыновлять» осиротевшие процессы (zombie reaping) и обрабатывать системные сигналы. Обычные процессы Linux имеют стандартные обработчики сигналов (например, SIGTERM вызывает завершение). Однако для PID 1 ядро Linux отключает эти стандартные обработчики. Если приложение.NET, запущенное как PID 1, явно не подписывается на сигналы, оно будет игнорировать команду docker stop (которая посылает SIGTERM), пока Docker не убьет его принудительно через SIGKILL (обычно через 10 секунд).7

Это критично для сервиса «Календарь». Принудительное завершение (SIGKILL) не дает приложению завершить текущие HTTP-запросы, сохранить состояние или закрыть соединения с базой данных, что может привести к потере данных.

В.NET (начиная с Core 2.1) встроен механизм обработки сигналов. IHost (используемый в ASP.NET Core) автоматически слушает SIGTERM и SIGINT, инициируя процедуру остановки через CancellationToken. Однако, это работает только если сигнал доходит до процесса.

Проблема Shell-оберток:

Если в Dockerfile используется инструкция ENTRYPOINT в формате shell (без скобок):

Dockerfile

ENTRYPOINT dotnet Calendar.dll

Docker запускает это как /bin/sh -c "dotnet Calendar.dll". В этом случае PID 1 получает оболочка sh, а приложение.NET становится дочерним процессом. Оболочка sh часто не пересылает сигналы дочерним процессам. В результате SIGTERM приходит в sh, тот его игнорирует (или завершается, не дожидаясь потомка), а.NET приложение продолжает работать до SIGKILL.

Решение: Всегда использовать exec формат (JSON массив):

Dockerfile

ENTRYPOINT ["dotnet", "Calendar.dll"]

Это замещает процесс оболочки процессом.NET, делая его PID 1 и обеспечивая корректную доставку сигналов.10

### 2.2. Network Namespace: Виртуализация сетевого стека

Network Namespace предоставляет контейнеру собственный сетевой стек: свои интерфейсы (например, eth0), свою таблицу маршрутизации, свои правила iptables и свой localhost. Это объясняет, почему несколько контейнеров могут слушать порт 80 на одной хост-машине без конфликтов — каждый из них слушает порт 80 в своем изолированном пространстве имен.4

В контексте.NET приложения это меняет семантику адресации. localhost или 127.0.0.1 внутри контейнера ссылается исключительно на сам контейнер. Сервис «Календарь» не может обратиться к базе данных, запущенной на хосте или в другом контейнере, используя localhost.

Для взаимодействия контейнеров Docker создает виртуальный мост (bridge network) и запускает внутренний DNS-сервер. Контейнеры могут обращаться друг к другу по именам сервисов (Service Discovery). Например, строка подключения в appsettings.json должна использовать имя сервиса из docker-compose.yml: Host=postgres\_db, а не IP-адрес.

Механизм Veth Pair:

Связь между Network Namespace контейнера и хостом осуществляется через пару виртуальных Ethernet-устройств (veth pair). Один конец «трубы» находится внутри контейнера (выглядит как eth0), другой — подключен к мосту docker0 на хосте. Понимание этого механизма необходимо при диагностике сетевых задержек или проблем с MTU (Maximum Transmission Unit), которые могут возникать при инкапсуляции трафика в облачных средах (Overlay Networks).5

### 2.3. Mount Namespace: Изоляция файловой системы

Mount Namespace позволяет каждому контейнеру иметь свое уникальное дерево файловой системы. Это достигается путем использования системного вызова pivot\_root, который подменяет корневую директорию процесса на директорию, собранную из слоев образа.

Благодаря этому, приложение.NET видит только те библиотеки и файлы, которые были упакованы в образ. Изменения в файловой системе контейнера (создание логов, временных файлов) происходят в специальном верхнем слое (writable layer) и не затрагивают образ или хост-систему, если не используются тома (Volumes).6

Проблема персистентности:

Так как Mount Namespace изолирован и эфемерен (уничтожается вместе с контейнером), любые данные, записанные в «локальную» папку внутри контейнера, будут потеряны. Для баз данных (PostgreSQL в нашем плане обучения) использование Docker Volumes является обязательным. Тома — это механизм «прокалывания» изоляции Mount Namespace, позволяющий мантировать директорию хоста внутрь контейнера.1

### 2.4. User Namespace: Безопасность и сопоставление ID

User Namespace изолирует идентификаторы пользователей (UID) и групп (GID). Это ключевой механизм для реализации безопасности. Он позволяет процессу быть root (UID 0) *внутри* контейнера, но при этом отображаться на непривилегированного пользователя (например, UID 10000) на уровне хоста.

Это предотвращает атаки типа «выход из контейнера» (container breakout). Если злоумышленник скомпрометирует приложение и получит root-права внутри контейнера, на уровне ядра хоста он останется бесправным пользователем и не сможет нанести вред системе (например, загрузить модуль ядра или изменить системные файлы).

В.NET 8 Microsoft внедрила пользователя app (UID 1654) непосредственно в базовые образы, продвигая практику запуска контейнеров без прав root по умолчанию. Это требует внимательного отношения к правам доступа при монтировании внешних томов (bind mounts), так как UID владельца файлов на хосте должен совпадать с UID пользователя внутри контейнера, либо права должны быть открыты для «others».14

## 3. Управление ресурсами: Control Groups (Cgroups)

Если Namespaces отвечают за то, что процесс *видит*, то Cgroups (Control Groups) отвечают за то, что процесс может *использовать*. Без Cgroups контейнер мог бы потребить всю оперативную память или процессорное время хоста, вызвав отказ в обслуживании (DoS) для других сервисов.17

Существует две версии Cgroups: v1 и v2. Современные дистрибутивы Linux и контейнерные среды (Kubernetes 1.25+) активно переходят на Cgroups v2, который предлагает единую иерархию и улучшенные механизмы управления памятью..NET Runtime должен уметь корректно определять версию Cgroups, чтобы правильно считывать лимиты.20

### 3.1. Управление памятью и поведение.NET GC

Управление памятью в контейнерах — один из самых сложных аспектов для.NET-приложений.

**Hard Limit vs Soft Limit:**

* **Hard Limit (memory.max):** Жесткий предел потребления памяти. Если контейнер превышает этот лимит и ядро не может освободить место (через сброс файлового кэша), срабатывает OOM Killer (Out Of Memory Killer), который немедленно убивает процесс. Для.NET приложения это выглядит как внезапное исчезновение без логов (StackOverflowException или OutOfMemoryException внутри управляемого кода возникают редко в таких случаях, чаще процесс убивается ядром раньше).
* **Soft Limit (memory.high):** Мягкий предел. При его превышении ядро начинает агрессивно освобождать память и может принудительно замедлять (троттлить) процесс, но не убивает его.

Адаптация Garbage Collector (GC):

В классическом серверном сценарии (Server GC).NET стремится занять всю доступную память для максимизации пропускной способности (throughput), проводя сборку мусора как можно реже. В контейнере это поведение фатально.

До появления механизмов адаптации,.NET мог прочитать объем памяти хоста (например, 64 ГБ) вместо лимита контейнера (512 МБ) и настроить кучу (Heap) соответствующим образом. Это приводило к мгновенному OOM.

Современный.NET (и особенно.NET 8) автоматически распознает лимиты Cgroups. Однако, стратегия Server GC все еще может быть слишком агрессивной.

В.NET 8 представлен механизм DATAS (Dynamic Adaptation To Application Sizes). Он динамически подстраивает размер кучи в зависимости от реальных потребностей приложения, а не только доступных ресурсов. Это особенно важно для «пульсирующих» нагрузок в микросервисах, позволяя контейнеру потреблять меньше памяти в простое и избегать OOM Kill при пиках.22

Рекомендация для Senior DevOps:

Всегда устанавливать жесткие лимиты памяти в Docker/Kubernetes. Настраивать.NET (переменная DOTNET\_GCHighMemPercent), чтобы GC начинал агрессивную сборку при достижении 75-90% от лимита контейнера, не дожидаясь вмешательства OOM Killer ядра.

### 3.2. CPU Quotas и ThreadPool

Cgroups ограничивают процессорное время через механизм квот (CFS Quota). Если контейнеру выделено 0.5 CPU, это значит, что за каждые 100 мс реального времени он может использовать процессор только 50 мс.

.NET ThreadPool использует количество процессоров для расчета оптимального числа потоков. Если рантайм видит все 64 ядра хоста вместо выделенных 0.5, он может создать слишком много потоков. Это приведет к:

1. Чрезмерному переключению контекста (context switching).
2. «Голоданию» потоков (Thread Starvation), когда потоки не успевают получить квант времени из-за троттлинга Cgroups.  
   В.NET 8 определение доступных ресурсов процессора значительно улучшено, что позволяет ThreadPool масштабироваться корректно относительно установленных лимитов (--cpus).17

## 4. Файловая система контейнеров: UnionFS и Overlay2

Эффективность Docker базируется на использовании слоистой файловой системы (Union Filesystem). Стандарт де-факто сегодня — драйвер **overlay2**.

### 4.1. Механизм Copy-on-Write (CoW)

Образ Docker состоит из набора неизменяемых (read-only) слоев. Когда контейнер запускается, поверх них создается тонкий записываемый слой (writable layer).

* **Чтение:** Если приложение читает файл, драйвер ищет его в слоях сверху вниз. Доступ происходит на скорости, близкой к нативной.
* **Запись:** Если приложение хочет изменить файл, который находится в нижнем (read-only) слое, драйвер сначала **копирует весь файл целиком** в верхний записываемый слой, и только потом применяет изменения. Это и есть Copy-on-Write.28

Влияние на производительность (Пример 100 МБ файла):

Предположим, в базовом образе есть лог-файл размером 100 МБ. Если ваше приложение решит дописать в него одну строчку, драйвер overlay2 будет вынужден скопировать все 100 МБ в записываемый слой контейнера. Это вызовет задержку ввода-вывода (I/O spike) и мгновенно увеличит размер контейнера на 100 МБ.

Вывод: Никогда не модифицируйте файлы, запеченные в образе. Логи должны писаться в stdout (откуда их заберет Docker logging driver) или на внешние тома.28

### 4.2. Whiteout-файлы и непрозрачные директории

Что происходит при удалении файла в контейнере? Так как нижние слои неизменяемы, файл физически не удаляется. Вместо этого в верхнем слое создается специальный файл-маркер, называемый whiteout (в overlay2 это символьное устройство с major/minor 0/0). Этот маркер говорит драйверу: «скрой файл с таким именем из нижних слоев».28

Это имеет критическое значение для безопасности и оптимизации размера образа. Если вы добавите файл с паролями в одном слое (COPY secrets.txt.), а в следующем удалите его (RUN rm secrets.txt), файл останется в истории слоев и будет доступен любому, кто скачает образ. Whiteout-файл лишь скроет его из финального вида файловой системы. Секреты никогда не должны попадать в слои образа.

## 5. Экосистема.NET 8 в контейнерах: Специфика и лучшие практики

Внедрение.NET 8 принесло ряд существенных изменений в работу с контейнерами, направленных на безопасность и оптимизацию.

### 5.1. Безопасность: Non-root User app

Как упоминалось выше, образы.NET 8 теперь включают пользователя app. Это изменение ломает многие старые Dockerfile, которые предполагали работу от root.

Последствия:

1. **Порт 8080:** Порты ниже 1024 в Linux являются привилегированными. Пользователь app не может забиндить порт 80. Поэтому дефолтный порт ASP.NET Core изменен на 8080. Это нужно учитывать при настройке Healthcheck и пробросе портов (EXPOSE 8080).2
2. **Запись в файловую систему:** Попытка записать что-либо в /app или изменить конфигурацию «на лету» приведет к ошибке Permission denied. Если приложению нужно писать временные файлы, необходимо явно создать директорию и передать права пользователю app *до* переключения USER:

Dockerfile

RUN mkdir /app/temp && chown app:app /app/temp  
USER app

Или использовать /tmp, который обычно доступен для записи всем.13

### 5.2. Выбор базового образа: Glibc vs Musl

Выбор между Debian (стандарт) и Alpine (минимализм) — это выбор между совместимостью и размером.

* **Debian (Bookworm-Slim):** Использует библиотеку glibc. Это стандартная библиотека C, используемая в большинстве Linux-систем. Обеспечивает максимальную совместимость с нативными библиотеками (System.Drawing, gRPC, криптография). Рекомендуется для начала.
* **Alpine Linux:** Использует библиотеку musl. Она намного легче, но имеет отличия в реализации некоторых функций (DNS-резолвинг, работа со стеком)..NET поддерживает Alpine (RID linux-musl-x64), но разработчик может столкнуться с трудноуловимыми багами или падением производительности.
* **Ubuntu Chiseled:** Новый тип образов от Canonical и Microsoft. Это distroless-образы (без пакетного менеджера, без shell), оптимизированные для продакшена. Они используют glibc, обеспечивая совместимость, но имеют размер, сопоставимый с Alpine. Это целевой выбор для Enterprise-решений, требующих высокой безопасности.33

**Таблица 2: Сравнительный анализ базовых образов для.NET 8**

| **Характеристика** | **Debian Bookworm-Slim** | **Alpine Linux** | **Ubuntu Chiseled** |
| --- | --- | --- | --- |
| **Размер (сжатый)** | ~75 MB | ~10-15 MB | ~15-20 MB |
| **Библиотека C** | glibc (Стандарт) | musl (Легковесная) | glibc (Стандарт) |
| **Совместимость.NET** | Максимальная | Требует тестирования (RID linux-musl) | Высокая |
| **Shell/Утилиты** | Есть (bash, apt) | Есть (sh, apk) | Нет (Безопасность++) |
| **Глобализация (ICU)** | Включена | Нужно ставить отдельно или invariant | Есть версии -extra |
| **Use Case** | Разработка, Отладка | Экстремальная экономия места | Production Security |

### 5.3. Глобализация и ICU

.NET использует библиотеку ICU для работы с датами, валютами и сортировкой строк. В полных образах (Debian) она предустановлена. В минимальных (Alpine, Chiseled) она часто отсутствует для экономии места.

Если сервис «Календарь» должен поддерживать разные часовые пояса или форматы дат, необходимо либо использовать образы с суффиксом -extra (для Chiseled), либо устанавливать icu-libs в Alpine. В противном случае приложение упадет при старте или переключится в Invariant Globalization Mode (только английский, примитивная сортировка), что недопустимо для бизнес-приложения.38

## 6. Стратегии сборки и оптимизации кэша

Эффективность CI/CD зависит от скорости сборки образов. Docker кэширует слои на основе хеш-сумм файлов.

Проблема COPY..:

Стандартная ошибка новичков — копировать весь исходный код перед восстановлением пакетов:

Dockerfile

COPY..  
RUN dotnet restore

Любое изменение в коде (даже пробел в комментарии) инвалидирует кэш слоя COPY, и dotnet restore будет выполняться заново, скачивая мегабайты пакетов.

Решение: Разделение слоев (Layer Caching)

Необходимо сначала копировать только файлы проектов (.csproj), восстанавливать зависимости, и только потом копировать остальной код. Для сложных решений с множеством проектов это может быть трудоемко. Инструмент dotnet-subset позволяет автоматизировать этот процесс, вытягивая из дерева исходников только файлы, необходимые для restore, сохраняя структуру директорий. Это позволяет Docker использовать кэш для слоя с пакетами NuGet, если список зависимостей не менялся.41

Multi-Stage Builds:

Использование многоэтапной сборки обязательно. Первый этап (build) использует «тяжелый» образ SDK (содержит компиляторы, MSBuild). Второй этап (final) использует легкий образ Runtime (только то, что нужно для запуска). Артефакты (DLL) копируются из первого этапа во второй. Это снижает размер финального образа с 800+ МБ до <150 МБ, уменьшая поверхность атаки и время деплоя.44

## 7. Заключение

Теоретическая база первого дня 15-й недели закладывает фундамент для трансформации сервиса «Календарь» в облачно-нативное приложение. Мы ушли от абстракций игрового движка к суровой реальности системных вызовов Linux.

Понимание Namespaces дает ключ к решению сетевых проблем и вопросов изоляции. Знание Cgroups и поведения.NET GC защищает от нестабильности и OOM-ошибок под нагрузкой. Использование UnionFS и правильных стратегий сборки обеспечивает эффективность доставки. Переход на модель безопасности с пользователем app и read-only файловыми системами приближает разработку к стандартам корпоративной безопасности (DevSecOps).

Эти знания не просто теоретические — они являются прямыми инструкциями к действию при написании Dockerfile и docker-compose.yml, которыми мы займемся в практической части. Следующим шагом станет применение этих принципов для создания идемпотентного окружения интеграционного тестирования с использованием Testcontainers.

**Источники:**

* 1 План обучения: Docker и Тестирование неделя 11
* 1 Архитектурная Эволюция: Неделя 15
* 4 Understanding Linux Namespaces
* 6 Mount Namespaces and nsenter Deep Dive
* 17 Cgroups v2 mechanism
* 15 New non-root 'app' user in Linux images
* 23 Optimize ASP.NET Core memory with DATAS
* 43 dotnet-subset tool
* 28 How overlay2 storage driver works

#### Источники

1. План обучения: Docker и Тестирование неделя 11
2. Secure your .NET cloud apps with rootless Linux Containers - Microsoft Developer Blogs, дата последнего обращения: декабря 6, 2025, <https://devblogs.microsoft.com/dotnet/securing-containers-with-rootless/>
3. .NET application publishing overview - .NET | Microsoft Learn, дата последнего обращения: декабря 6, 2025, <https://learn.microsoft.com/en-us/dotnet/core/deploying/>
4. Understanding Linux Namespaces: A Deep Dive into Container Isolation - WafaTech Blogs, дата последнего обращения: декабря 6, 2025, <https://wafatech.sa/blog/linux/linux-security/understanding-linux-namespaces-a-deep-dive-into-container-isolation/>
5. A deep dive into Linux namespaces - Chord Simple, дата последнего обращения: декабря 6, 2025, <https://ifeanyi.co/posts/linux-namespaces-part-1/>
6. Linux Mount Namespaces and nsenter: Deep Dive - DEV Community, дата последнего обращения: декабря 6, 2025, <https://dev.to/ajinkya_singh_2c02bd40423/mount-namespaces-and-nsenter-deep-dive-39o7>
7. How to send SIGTERM (graceful shutdown) to a .NET Core process in MacOS/Linux, дата последнего обращения: декабря 6, 2025, <https://blog.steadycoding.com/how-to-send-sigterm-and-more-to-a-net-core-process/>
8. Gracefully Shut Down dotnet 8 IHostedService App - Deployed as a Windows Container in AKS - While Scale In or Pod Deallocations - Chaminda's DevOps Journey with MSFT, дата последнего обращения: декабря 6, 2025, <http://chamindac.blogspot.com/2025/03/gracefully-shut-down-dotnet-8.html?m=1>
9. How to do gracefully shutdown on dotnet with docker? - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/40742192/how-to-do-gracefully-shutdown-on-dotnet-with-docker>
10. Understanding CMD, ENTRYPOINT and RUN in Docker - An Interlude Post, дата последнего обращения: декабря 6, 2025, <https://dev.to/kalkwst/interlude-1-cmd-entrypoint-and-run-2926>
11. Docker Best Practices: Choosing Between RUN, CMD, and ENTRYPOINT, дата последнего обращения: декабря 6, 2025, <https://www.docker.com/blog/docker-best-practices-choosing-between-run-cmd-and-entrypoint/>
12. Understanding Docker's CMD and ENTRYPOINT Instructions - CloudBees, дата последнего обращения: декабря 6, 2025, <https://www.cloudbees.com/blog/understanding-dockers-cmd-and-entrypoint-instructions>
13. 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>
14. Isolate containers with a user namespace - Docker Docs, дата последнего обращения: декабря 6, 2025, <https://docs.docker.com/engine/security/userns-remap/>
15. docs/docs/core/compatibility/containers/8.0/app-user.md at main · dotnet/docs · GitHub, дата последнего обращения: декабря 6, 2025, <https://github.com/dotnet/docs/blob/main/docs/core/compatibility/containers/8.0/app-user.md>
16. 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>
17. Using cgroups to limit I/O - andrestc.com, дата последнего обращения: декабря 6, 2025, <https://andrestc.com/post/cgroups-io/>
18. About cgroup v2 - Kubernetes, дата последнего обращения: декабря 6, 2025, <https://kubernetes.io/docs/concepts/architecture/cgroups/>
19. Control Group v2 - The Linux Kernel documentation, дата последнего обращения: декабря 6, 2025, <https://docs.kernel.org/admin-guide/cgroup-v2.html>
20. Kubernetes 1.25: cgroup v2 graduates to GA, дата последнего обращения: декабря 6, 2025, <https://kubernetes.io/blog/2022/08/31/cgroupv2-ga-1-25/>
21. Chapter 26. Configuring resource management by using cgroups-v2 and systemd, дата последнего обращения: декабря 6, 2025, <https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/8/html/managing_monitoring_and_updating_the_kernel/assembly_configuring-resource-management-using-systemd_managing-monitoring-and-updating-the-kernel>
22. Dynamically Adapting To Application Sizes | by Maoni0 - Medium, дата последнего обращения: декабря 6, 2025, <https://maoni0.medium.com/dynamically-adapting-to-application-sizes-2d72fcb6f1ea>
23. Optimize ASP.NET Core memory with DATAS - Thinktecture AG, дата последнего обращения: декабря 6, 2025, <https://www.thinktecture.com/net/optimize-asp-net-core-memory-with-datas/>
24. Garbage collector config settings - .NET - Microsoft Learn, дата последнего обращения: декабря 6, 2025, <https://learn.microsoft.com/en-us/dotnet/core/runtime-config/garbage-collector>
25. Dynamic adaptation to application sizes (DATAS) - .NET | Microsoft Learn, дата последнего обращения: декабря 6, 2025, <https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/datas>
26. Container security fundamentals part 4: Cgroups, дата последнего обращения: декабря 6, 2025, <https://securitylabs.datadoghq.com/articles/container-security-fundamentals-part-4/>
27. What's new in .NET 8 runtime - Microsoft Learn, дата последнего обращения: декабря 6, 2025, <https://learn.microsoft.com/en-us/dotnet/core/whats-new/dotnet-8/runtime>
28. OverlayFS storage driver - Docker Docs, дата последнего обращения: декабря 6, 2025, <https://docs.docker.com/engine/storage/drivers/overlayfs-driver/>
29. Storage drivers - Docker Docs, дата последнего обращения: декабря 6, 2025, <https://docs.docker.com/engine/storage/drivers/>
30. Case Study: How We Reduced a Container Image by 99.7% (800GB to 2GB) | Sealos Blog, дата последнего обращения: декабря 6, 2025, <https://sealos.io/blog/reduce-container-image-size-case-study>
31. Unveiling Whiteout Files: Do you know how file deletions are handled between layers of a Docker image? - - nemunaire's blog, дата последнего обращения: декабря 6, 2025, <https://www.nemunai.re/post/unveiling-whiteout-files/>
32. overlayfs inside docker container - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/67198603/overlayfs-inside-docker-container>
33. What's new in containers for .NET 8 - Microsoft Learn, дата последнего обращения: декабря 6, 2025, <https://learn.microsoft.com/en-us/dotnet/core/whats-new/dotnet-8/containers>
34. [Bug]: .NET 8 Container Permissions Changed Breaking Non-Root App User #1171 - GitHub, дата последнего обращения: декабря 6, 2025, <https://github.com/testcontainers/testcontainers-dotnet/issues/1171>
35. Question about Alpine vs Debian base images and their impact on pod startup performance : r/kubernetes - Reddit, дата последнего обращения: декабря 6, 2025, <https://www.reddit.com/r/kubernetes/comments/1dpqis9/question_about_alpine_vs_debian_base_images_and/>
36. musl vs glibc: Pros, Cons, and Key Differences - TuxCare, дата последнего обращения: декабря 6, 2025, <https://tuxcare.com/blog/musl-vs-glibc/>
37. Chiseled Ubuntu and .NET Containers, дата последнего обращения: декабря 6, 2025, <https://ubuntu.com/containers/chiseled/dotnet>
38. 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>
39. Install .NET on Alpine - Microsoft Learn, дата последнего обращения: декабря 6, 2025, <https://learn.microsoft.com/en-us/dotnet/core/install/linux-alpine>
40. NET globalization and ICU - Microsoft Learn, дата последнего обращения: декабря 6, 2025, <https://learn.microsoft.com/en-us/dotnet/core/extensions/globalization-icu>
41. 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>
42. 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/>
43. 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>
44. Building best practices - Docker Docs, дата последнего обращения: декабря 6, 2025, <https://docs.docker.com/build/building/best-practices/>
45. ASP.NET 8 Web API Dockerfile breakdown | by Jakub Rzepka - Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@jakubrzepka/asp-net-8-web-api-dockerfile-brakedwon-56bd3a17894a>