# Архитектурная Эволюция: Неделя 15 — Контейнеризация и Стратегии Обеспечения Качества в Распределенных Системах

## 1. Введение: Фундаментальный сдвиг парадигмы доставки

Переход к пятнадцатой неделе обучения знаменует собой критический этап в профессиональной трансформации Unity-разработчика в инженера Enterprise-бэкенда. Если предыдущие четырнадцать недель, согласно плану обучения 1, были посвящены внутренней механике кода — от управления памятью и сборки мусора (Недели 1-2) до асинхронности и проектирования REST API (Недели 6-10), — то текущий этап смещает фокус на **среду исполнения** и **гарантии надежности**. В контексте стратегического документа «Разработка бэкенда: план обучения C#» 1, этот модуль служит необходимым мостом между разработкой монолитного сервиса «Календарь» и переходом к сложной микросервисной архитектуре, запланированной на шестнадцатую неделю с внедрением RabbitMQ.

### 1.1. От игрового цикла к контейнерной оркестрации

Для разработчика, пришедшего из индустрии Unity, концепция среды исполнения (Runtime Environment) часто воспринимается как данность. Игровой движок предоставляет абстракцию, скрывающую различия между операционными системами, а билд игры представляет собой самодостаточный артефакт. Если игра запускается на устройстве разработчика, с высокой долей вероятности она запустится и у пользователя, при условии соответствия аппаратным требованиям. В мире бэкенд-разработки понятие «среда» является гораздо более изменчивым и критичным фактором. Код, безупречно работающий в среде Localhost на машине разработчика, может отказать в производственной среде (Production) из-за, казалось бы, тривиальных различий: другой версии системной библиотеки, отсутствующих зависимостей, нюансов конфигурации сети или различий в версиях СУБД.

Пятнадцатая неделя призвана решить две фундаментальные проблемы, с которыми неизбежно сталкивается каждый бэкенд-инженер:

1. **Проблема воспроизводимости среды («It works on my machine»):** Эта проблема решается через **Докеризацию** (Dockerization). Мы переходим от парадигмы доставки исполняемых файлов к парадигме доставки изолированных контейнеров, упаковывающих приложение вместе со всеми его зависимостями, от конфигурации OS до рантайма.NET.
2. **Проблема регрессии и интеграционной целостности:** Решается через внедрение **Интеграционного тестирования**. В отличие от Unit-тестов, проверяющих изолированную логику, интеграционные тесты верифицируют работу системы в сборе. Как указано в исходном плане 1, сервис «Календарь» должен быть протестирован в условиях, максимально приближенных к боевым, с использованием реальной базы данных PostgreSQL, а не ее имитации.

### 1.2. Стратегические цели модуля

К завершению этой недели инженер должен не просто обладать навыком написания Dockerfile, но и глубоко понимать философию Linux-контейнеров, принципы работы слоевой файловой системы (UnionFS) и методы обеспечения идемпотентности развертывания. В области обеспечения качества задача состоит в том, чтобы преодолеть привычку полагаться на ручное тестирование (аналог Play Mode в Unity) и научиться создавать эфемерные окружения для автоматизированных тестов, которые зеркально отражают продакшен-инфраструктуру.

Объектом практической работы выступает сервис **«Календарь»** (Calendar Service), архитектура которого закладывалась на неделях 7-8.1 Наша цель — трансформировать его из локального C# проекта в cloud-native приложение, готовое к горизонтальному масштабированию и развертыванию в кластерах Kubernetes, хотя сам Kubernetes остается за рамками текущего спринта.

Ниже представлена сравнительная таблица, иллюстрирующая смену ментальной модели при переходе к контейнеризации:

| **Характеристика** | **Подход Unity / Desktop** | **Подход Cloud-Native Backend** | **Архитектурное следствие для Недели 15** |
| --- | --- | --- | --- |
| **Среда исполнения** | Управляемая движком (Mono/IL2CPP), жесткая привязка к OS пользователя. | Контейнеризированная (Linux), полная изоляция от хоста. | Необходимость изучения Linux-примитивов (Namespaces, Cgroups). |
| **Зависимости** | Включены в билд (DLLs) или установлены пользователем (DirectX, Drivers). | Явно декларированы в Dockerfile. OS — это тоже зависимость. | Минимизация размера образа для ускорения деплоя и скалирования. |
| **Конфигурация** | ScriptableObjects, PlayerPrefs (внутри билда). | Переменные окружения (Environment Variables), монтируемые файлы. | Вынос конфигурации за пределы кода. Принцип 12-Factor App. |
| **Тестирование** | Ручное (QA), Unit-тесты логики. Сложность автоматизации интеграции. | Автоматизированное интеграционное (Testcontainers). Эфемерные БД. | Тесты должны сами поднимать инфраструктуру (БД) и убивать её после прогона. |

## 2. Стратегия взаимодействия с AI-ментатором: Эмуляция Senior DevOps

Для эффективного освоения материала на 15-й неделе необходимо радикально пересмотреть роль AI-ассистента. Если на первых этапах 1 он выступал в роли тьютора по синтаксису C# или алгоритмического наставника, то теперь он должен принять персону **Senior DevOps Engineer & QA Lead**. Это требует смены тональности общения: от поддерживающей и объясняющей к требовательной, сфокусированной на безопасности, производительности и надежности.

### 2.1. Психологический профиль ментора

В отличие от предыдущих недель, где допускались неоптимальные решения ради учебного процесса, на этой неделе AI должен быть безжалостен к таким аспектам, как запуск процессов от имени root, наличие уязвимостей в базовых образах или нестабильность (flakiness) тестов. Он должен эмулировать поведение строгого системного архитектора, который блокирует Pull Request, если образ весит 800 Мб вместо 150 Мб, или если тесты проходят через раз.

### 2.2. Глобальный системный промпт (System Prompt)

Ниже представлен детализированный промпт, который обучаемый должен использовать для инициализации контекста сессии с LLM (ChatGPT/Claude/Gemini). Этот промпт задает рамки и правила игры на всю неделю.

**Роль:** Ты — Senior DevOps Engineer и Lead QA Automation с 10-летним опытом работы в высоконагруженных распределенных системах на платформе.NET и Azure/AWS. Ты специализируешься на Docker, Linux-контейнерах, CI/CD пайплайнах и стратегиях тестирования (Testing Pyramid).

**Контекст:** Я — разработчик, находящийся в процессе перехода с Unity на Enterprise Backend (согласно плану обучения). У меня есть рабочий Web API сервис "Календарь" (ASP.NET Core, PostgreSQL, EF Core), разработанный на предыдущих этапах. Я понимаю C#, но имею поверхностные знания о Linux и глубокой контейнеризации.

**Твоя задача:** Сопровождать меня в процессе докеризации сервиса и написания надежных интеграционных тестов на 15-й неделе обучения.

**Принципы взаимодействия и Критерии качества:**

1. **Security First:** При проверке Dockerfile ты должен параноидально искать уязвимости: запуск от root, использование устаревших базовых образов, жестко закодированные секреты, лишние права доступа.
2. **Optimization Obsession:** Критикуй размер итогового образа. Требуй использования Multi-stage builds. Объясняй, почему каждый мегабайт имеет значение при масштабировании на тысячи подов.
3. **Realism in Testing:** Если я пытаюсь использовать Mock-объекты для базы данных в интеграционном тесте или предлагаю InMemoryDatabase, останови меня. Настаивай на использовании реальной БД (через Testcontainers). Объясни, почему In-Memory DB — это "ложный друг", скрывающий баги.
4. **Deep Dive Theory:** Не давай просто готовый код. Если я использую команду, спроси меня, понимаю ли я, что она делает на уровне ядра OS (например, чем ENTRYPOINT отличается от CMD или как работают Linux Namespaces).
5. **Troubleshooting Guide:** Когда я неизбежно столкнусь с сетевыми проблемами между контейнерами, не давай решение сразу. Направляй меня через процесс отладки (DNS разрешение, проверка портов, анализ логов, bridge networks).

**Формат ответа:**

* *Анализ:* Краткая оценка моего решения.
* *Code Review:* Подсветка проблемных зон с объяснением рисков.
* *Socratic Question:* Вопрос на засыпку, заставляющий задуматься об архитектурных последствиях (например: "Что произойдет с логами контейнера, если он перезагрузится, и как это решить?").

## 3. Детальный план обучения: Погружение в Контейнеризацию (20 часов)

Программа рассчитана на 5 дней интенсивной работы (по 4 часа в день). Каждый день структурирован следующим образом: теоретический блок (Deep Dive), промпты для исследования (Research Prompts) для самостоятельного изучения с AI, и практическая реализация задач на базе проекта "Календарь".

### День 1: Анатомия Docker и Оптимизация Образов

**Цель:** Понять, как работает изоляция процессов в Linux, и научиться создавать безопасные, минималистичные образы для.NET приложений.

#### 3.1. Теоретический базис: Под капотом контейнера

Unity-разработчики привыкли работать поверх Windows API. Docker требует понимания того, как операционная система управляет процессами. Контейнер — это не "легковесная виртуалка", это обычный процесс Linux, которому "солгали" о том, что он один в системе.

* **Изоляция через Namespaces (Пространства имен):**
  + *PID Namespace:* Процесс внутри контейнера видит себя как PID 1.
  + *Network Namespace:* У контейнера свой сетевой стек (IP, localhost, таблица маршрутизации).
  + *Mount Namespace:* У контейнера своя файловая система.
  + *User Namespace:* Позволяет процессу быть root внутри контейнера, но обычным пользователем снаружи (важно для безопасности).
* **Контроль ресурсов через Cgroups (Control Groups):** Механизм ядра, ограничивающий потребление CPU и RAM. Важно понимать, как.NET GC (Garbage Collector) ведет себя в условиях ограничений памяти (Hard Limit vs Soft Limit). С версии.NET 8 рантайм автоматически распознает лимиты cgroup, но понимание механизма критично для отладки OOM (Out Of Memory) ошибок.
* **Слоевая файловая система (UnionFS/Overlay2):** Каждый шаг в Dockerfile создает новый неизменяемый слой. Понимание кэширования слоев позволяет ускорить сборку с минут до секунд. Это концептуально похоже на инкрементальные билды в Unity, но на уровне файловой системы.

#### 3.2. Стратегия Multi-stage Build

Для компилируемых языков, таких как C#, критически важно разделять среду сборки и среду выполнения.

* **Stage 1 (Build):** Использует образ SDK (содержит компилятор Roslyn, MSBuild, NuGet). Он тяжелый (сотни мегабайт).
* **Stage 2 (Runtime):** Использует образ ASP.NET Core Runtime. Он легкий, содержит только CLR и необходимые библиотеки. Не содержит исходного кода.

#### 3.3. Промпты для исследования (Research Prompts)

Студент должен задать эти вопросы AI-ментору для глубокого погружения:

* *"Объясни механизм Copy-on-Write в контексте слоев Docker-образа. Что произойдет с размером образа, если я в одном слое создам временный файл на 100Мб, а в следующем слое удалю его? Почему этот файл останется в истории слоев?"*
* *"В чем фундаментальная разница между dotnet publish внутри Dockerfile и публикацией локально с последующим копированием DLL в контейнер? Почему CI/CD пайплайны требуют первого подхода?"*
* *"Сравни образы mcr.microsoft.com/dotnet/aspnet:8.0-alpine и 8.0-bookworm-slim. Какие риски несет использование Alpine (musl libc) для.NET приложений, особенно при работе с Globalization и нативными библиотеками (например, System.Drawing или gRPC)?"*

#### 3.4. Практическая задача: Dockerfile для Календаря

Задание: Написать оптимизированный Dockerfile для сервиса Calendar.

Технические требования:

1. Использовать Multi-stage build.
2. Настроить .dockerignore (исключить bin, obj, .git, .idea).
3. Запуск приложения должен производиться не от root пользователя (использовать USER app или создать своего пользователя).
4. Приложение должно слушать порт 8080.
5. Итоговый размер образа должен быть < 200 Мб (для Linux x64).

**Пример архитектуры Dockerfile:**

Dockerfile

# Stage 1: Build  
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build  
WORKDIR /src  
# Копируем сначала только csproj для кэширования restore слоя  
COPY ["Calendar/Calendar.csproj", "Calendar/"]  
RUN dotnet restore "Calendar/Calendar.csproj"  
# Копируем остальной код  
COPY..  
WORKDIR "/src/Calendar"  
RUN dotnet publish "Calendar.csproj" -c Release -o /app/publish /p:UseAppHost=false  
  
# Stage 2: Runtime  
FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS final  
WORKDIR /app  
# Создаем непривилегированного пользователя (best practice)  
# (В образах.NET 8 user 'app' уже существует, нужно просто переключиться)  
USER app  
COPY --from=build /app/publish.  
EXPOSE 8080  
ENV ASPNETCORE\_URLS=http://+:8080  
ENTRYPOINT ["dotnet", "Calendar.dll"]

### День 2: Оркестрация и Сетевое взаимодействие (Docker Compose)

**Цель:** Освоить управление зависимостями и понять, как контейнеры общаются друг с другом. Сервис «Календарь» бесполезен без базы данных.

#### 4.1. Теоретический базис

В Unity объекты на сцене находят друг друга через GetComponent или FindObjectOfType. В микросервисах "поиск" происходит через сеть (Service Discovery).

* **Docker Compose как IaC (Infrastructure as Code):** Декларативное описание инфраструктуры. Позволяет поднять все окружение одной командой docker compose up.
* **Внутренний DNS Docker:** Docker поднимает свой DNS-сервер. Сервисы в одной сети могут обращаться друг к другу по имени сервиса. Например, ConnectionString в C# будет выглядеть так: Host=postgres\_db;Port=5432..., где postgres\_db — это имя сервиса в docker-compose.yml.
* **Persistence (Хранение данных):** Контейнеры эфемерны. Если убить контейнер БД, данные исчезнут.
  + *Volumes (Тома):* Управляются Docker, хранятся в специальной директории на хосте. Лучший выбор для БД.
  + *Bind Mounts:* Прямое отображение папки хоста в контейнер. Удобно для конфигов или кода при разработке.
* **Проблема очередности запуска (Startup Order):** Контейнер БД может запуститься ("Running"), но база внутри еще не инициализировалась. Приложение упадет при попытке подключения. Решение: depends\_on + healthcheck.

#### 4.2. Промпты для исследования

* *"Как работает разрешение имен (DNS resolution) внутри пользовательской сети Docker bridge? Что произойдет, если я попытаюсь обратиться к сервису по localhost внутри контейнера? (Спойлер: я постучусь в сам контейнер, а не в соседний)."*
* *"Исследуй механизм healthcheck в Docker Compose. Как настроить ожидание готовности PostgreSQL (команда pg\_isready) перед запуском сервиса приложения?"*
* *"Как передать конфигурацию (Connection String) в контейнер через переменные окружения, переопределяя значения из appsettings.json? Объясни иерархию провайдеров конфигурации в ASP.NET Core."*

#### 4.3. Практическая задача: Экосистема Календаря

Задание: Создать файл docker-compose.yml и скрипты инициализации.

Требования:

1. Описать два сервиса: calendar-api и calendar-db (PostgreSQL 15).
2. Настроить Volume для сохранения данных БД.
3. Использовать переменные окружения для передачи паролей (через .env файл, который добавлен в .gitignore).
4. Пробросить порт 5000 с хоста на 8080 контейнера приложения.
5. Настроить Healthcheck для БД, чтобы приложение ждало её готовности.

### День 3: Философия Интеграционного Тестирования

**Цель:** Перейти от модульных тестов к проверке сценариев "черного ящика" через HTTP вызовы.

#### 5.1. Теоретический базис

Согласно разделу 5.1 исходного документа 1, интеграционные тесты — это ключевой элемент фазы Enterprise.

* **Пирамида тестирования:** Unit-тесты дешевы и быстры, но они проверяют, правильно ли мы построили компонент. Интеграционные тесты проверяют, правильный ли компонент мы построили и как он взаимодействует с внешним миром.
* **WebApplicationFactory:** Это мощнейший инструмент из пакета Microsoft.AspNetCore.Mvc.Testing. Он позволяет запустить ваше ASP.NET Core приложение **в памяти** (In-Memory Test Server), без необходимости деплоить его на реальный веб-сервер. При этом весь конвейер Middleware, DI-контейнер и маршрутизация работают по-настоящему.
* **Подмена зависимостей:** WebApplicationFactory позволяет "влезть" в DI-контейнер перед стартом приложения и заменить реальные сервисы на моки. Однако, для базы данных мы будем использовать другой подход.

#### 5.2. Промпты для исследования

* *"Почему использование UseInMemoryDatabase (EF Core) считается анти-паттерном для интеграционных тестов? Приведи примеры SQL-запросов или ограничений (Foreign Keys), которые будут работать в In-Memory, но упадут в реальном PostgreSQL."*
* *"Как работает WebApplicationFactory под капотом? Создает ли он реальный сетевой сокет TCP, или общение клиента HttpClient с сервером происходит через прямую передачу объектов в памяти?"*

#### 5.3. Практическая задача: Инфраструктура тестов

Задание: Создать проект Calendar.IntegrationTests.

Шаги:

1. Подключить Microsoft.AspNetCore.Mvc.Testing и xUnit.
2. Реализовать класс-наследник WebApplicationFactory<Program>, который переопределяет конфигурацию (например, читает appsettings.Testing.json).
3. Написать первый "Smoke Test": GET /health или GET /api/events возвращает 200 OK.

### День 4: Продвинутые сценарии с Testcontainers

**Цель:** Реализовать "Золотой стандарт" тестирования в.NET — полную изоляцию с использованием реальной инфраструктуры.

#### 6.1. Теоретический базис

Библиотека **Testcontainers** позволяет C# коду программно управлять Docker-контейнерами.

* **Ephemeral Environments (Эфемерные окружения):** Для каждого прогона тестов создается абсолютно чистая, новая база данных в контейнере. После тестов она уничтожается. Это гарантирует отсутствие "наводок" от предыдущих тестов.
* **IAsyncLifetime:** Интерфейс xUnit, позволяющий выполнять асинхронные операции инициализации (старт контейнера) перед запуском тестов и очистку (стоп контейнера) после.
* **Динамическая конфигурация:** Тест запускает контейнер БД -> Docker назначает случайный свободный порт -> Тест получает Connection String -> Тест передает эту строку в приложение.

#### 6.2. Промпты для исследования

* *"Сравни стратегии управления состоянием БД в тестах: (А) Новый контейнер на каждый тестовый класс vs (Б) Один контейнер на все тесты с очисткой данных (Respawner/TRUNCATE). В чем компромисс между скоростью и изоляцией?"*
* *"Как протестировать сценарии аутентификации, если мы используем внешний Identity Provider (например, Google или Keycloak)? Как настроить TestAuthHandler для обхода проверки токена в интеграционных тестах?"*

#### 6.3. Практическая задача: Реализация Testcontainers

Задание: Переписать тесты с использованием библиотеки Testcontainers.PostgreSql.

Требования:

1. Тест должен поднимать реальный PostgreSQL.
2. Приложение должно автоматически применять миграции EF Core на старте теста к этой новой БД.
3. Написать тест-кейс "Полный цикл события":
   * Создать событие (POST).
   * Проверить, что оно создалось (Assert ID > 0).
   * Запросить список событий (GET) и найти там созданное.
   * Убедиться, что поля (дата, заголовок) совпадают.

**Пример реализации (Snippet):**

C#

public class CalendarTests : IClassFixture<CalendarFactory>, IAsyncLifetime  
{  
 private readonly PostgreSqlContainer \_dbContainer = new PostgreSqlBuilder()  
 .WithImage("postgres:15")  
 .Build();  
  
 public async Task InitializeAsync() {  
 await \_dbContainer.StartAsync();  
 // Передача строки подключения в Factory...  
 }  
 //...  
}

### День 5: CI/CD Foundations и Итоговая задача

**Цель:** Консолидация знаний. Автоматизация процессов сборки и тестирования.

#### 7.1. Теоретический базис

* **Continuous Integration (CI):** Практика, при которой каждое изменение в коде автоматически триггерит сборку и прогон всех тестов. Если тесты с Testcontainers проходят в CI (например, в GitHub Actions), это дает высочайшую гарантию качества.
* **Скриптование:** Умение написать bash или powershell скрипт, который "делает всё", чтобы разработчик не набирал 10 команд вручную.

#### 7.2. Итоговая задача недели: "Непотопляемый Календарь"

**Легенда:** Команда SRE требует гарантий, что сервис готов к деплою. Вы должны предоставить артефакты, доказывающие надежность.

**Комплексное задание:**

1. **Final Dockerfile:** Максимально оптимизированный, безопасный.
2. **Verify Script:** Написать скрипт verify.sh, который:
   * Собирает Docker-образ.
   * Запускает интеграционные тесты (которые сами поднимут свои контейнеры с БД).
   * Если тесты прошли, поднимает окружение через docker compose up -d.
   * Делает тестовый curl запрос к API.
   * Останавливает окружение.
3. **Отчет о покрытии:** Интегрировать сбор покрытия кода (Code Coverage) в тесты и убедиться, что основные сценарии контроллера Calendar покрыты.

## 4. Глубокий анализ инсайтов и связей (Second-Order Insights)

Интеграция материалов недели 15 с общим контекстом документа 1 позволяет выявить неочевидные, но критические взаимосвязи.

### 4.1. Связь с управлением памятью (Неделя 2)

На второй неделе изучались SOH/LOH и сборка мусора. В контексте Docker это знание обретает новый смысл. В контейнерных средах (Kubernetes) часто устанавливаются жесткие лимиты памяти (Memory Limits).

* *Инсайт:* Если приложение потребляет память близко к лимиту контейнера, Kubernetes просто убьет процесс (OOM Kill). В отличие от десктопа, где есть файл подкачки (Swap) и гигабайты RAM, контейнер не прощает утечек.
* *Следствие:* Разработчик должен мониторить метрику Gen2 Size и LOH Size. Неправильная работа с памятью, допустимая на десктопе, в микросервисе приведет к постоянным перезагрузкам пода (CrashLoopBackOff).

### 4.2. Связь с многопоточностью (Неделя 3)

Docker по умолчанию предоставляет доступ ко всем ядрам хоста, но через Cgroups можно ограничить процессорное время (CPU Quota).

* *Инсайт:* Environment.ProcessorCount в.NET может врать внутри контейнера, если лимиты настроены неправильно. Это влияет на размер ThreadPool.
* *Следствие:* При высокой нагрузке (тысячи запросов), если ThreadPool не будет масштабироваться корректно из-за ограничений CPU, возникнет "Thread Starvation". Интеграционные нагрузочные тесты должны выявлять это поведение.

### 4.3. Связь с будущей темой RabbitMQ (Неделя 16)

Докеризация на 15-й неделе — это не самоцель, а пререквизит для недели 16.1

* *Инсайт:* Разработка микросервисной архитектуры с брокером сообщений (RabbitMQ) без Docker практически невозможна на локальной машине разработчика. Сложность ручной установки Erlang, RabbitMQ, плагинов управления и их конфигурации отпугивает новичков.
* *Следствие:* Освоение docker-compose сейчас — это инвестиция, которая позволит на следующей неделе поднять кластер RabbitMQ одной строкой, сосредоточившись на логике сообщений, а не на администрировании серверов.

## 5. Заключение

Пятнадцатая неделя является поворотной точкой в программе обучения. Студент перестает быть "автором кода" и становится "инженером систем". Владение контейнеризацией и автоматизированным тестированием с использованием реальной инфраструктуры (Testcontainers) — это тот навык, который отличает Senior разработчика от Middle. Это обеспечивает фундамент для построения надежных, масштабируемых систем, способных выдерживать нагрузки Enterprise-уровня, к которым готовит данный курс. Следующий шаг — внедрение асинхронного взаимодействия через RabbitMQ — теперь будет базироваться на твердой инфраструктурной почве.

#### Источники

1. Разработка бэкенда: план обучения C#