# Архитектурная Трансформация и Непрерывная Интеграция: Глубокий Анализ Экосистемы CI/CD для.NET Enterprise

## 1. Введение: Эпистемологический Сдвиг в Инженерии Программного Обеспечения

Современная разработка корпоративных backend-систем на платформе.NET переживает фундаментальный сдвиг парадигмы, который в учебном плане восемнадцатой недели охарактеризован как "Архитектурная Трансформация". Этот переход знаменует собой отказ от традиционных для GameDev-индустрии практик, где сборка проекта часто представляет собой ручной, недетерминированный процесс, зависящий от уникальной конфигурации рабочей станции разработчика, в пользу строгих, воспроизводимых и автоматизированных конвейеров Continuous Integration (CI). Если в мире Unity создание исполняемого файла — это финальный аккорд творческого процесса, то в мире Enterprise.NET — это лишь начальная точка жизненного цикла сервиса, который должен выживать во враждебной, динамичной среде облачной инфраструктуры.1

Четвертый день обучения фокусируется на критически важном аспекте этой трансформации — построении CI пайплайнов и интеграционном тестировании. Это не просто техническая задача по написанию YAML-скриптов; это внедрение философии, согласно которой код не считается существующим, пока он не прошел через "горнило" автоматизированных проверок в изолированной среде. Переход от локальных сборок к облачным раннерам GitHub Actions требует от инженера глубокого понимания принципов виртуализации, контейнеризации и сетевого взаимодействия, так как исчезает привычная концепция "localhost" как постоянной и неизменной сущности.

В данном отчете мы проведем исчерпывающий анализ архитектуры CI/CD на базе GitHub Actions, детально разберем механизмы работы Service Containers для эмуляции инфраструктурных зависимостей (PostgreSQL, RabbitMQ), исследуем стратегии оптимизации времени сборки через многоуровневое кэширование и предложим решения для классических проблем распределенных систем, таких как гонки (race conditions) при инициализации сервисов.

## 2. Архитектура и Механика GitHub Actions

GitHub Actions (GHA) представляет собой не просто инструмент для запуска скриптов, а полноценную платформу оркестрации рабочих процессов, глубоко интегрированную в событийную модель репозитория. Понимание её внутренней архитектуры необходимо для проектирования надежных пайплайнов, способных масштабироваться вместе с ростом кодовой базы.

### 2.1. Анатомия Workflow: От Декларации к Исполнению

В основе GHA лежит декларативная модель описания процессов. В отличие от императивных подходов, где инженер описывает *как* выполнить задачу шаг за шагом, YAML-конфигурация GHA описывает *желаемое состояние* процесса. Когда происходит событие-триггер (например, git push), GitHub интерпретирует этот файл и строит граф выполнения (Execution Graph).2

#### Иерархия Объектов Исполнения

| **Объект** | **Роль в Архитектуре** | **Жизненный Цикл и Изоляция** |
| --- | --- | --- |
| **Workflow** | Корневая сущность, определяющая весь процесс автоматизации. | Привязан к репозиторию и событию. Хранится в .github/workflows. |
| **Job (Джоб)** | Единица выполнения, назначаемая на конкретный раннер. | Джобы по умолчанию выполняются параллельно. Каждый джоб получает **чистую** виртуальную машину или контейнер. Изоляция на уровне ОС. 3 |
| **Step (Шаг)** | Атомарная операция внутри джоба (скрипт или Action). | Шаги выполняются последовательно внутри одного процесса раннера. Делят одну файловую систему и сетевой стек. 4 |
| **Action** | Переиспользуемый модуль логики (JavaScript или Docker). | Инкапсулирует сложную логику, позволяя переиспользовать код между проектами (например, actions/checkout). |

Критически важным аспектом, который часто упускается из виду начинающими специалистами, переходящими с локальной разработки, является **изоляция джобов**. Файловая система, созданная в одном джобе (например, скомпилированные бинарные файлы в папке bin), не доступна в другом джобе, если не использовать механизмы передачи артефактов (upload-artifact/download-artifact). Это фундаментально отличается от локальной сборки, где состояние сохраняется между запусками.5

### 2.2. Модель Событий и Триггеры

Эффективность CI пайплайна во многом зависит от того, насколько точно настроены условия его запуска. Избыточные запуски приводят к очередям на раннерах и замедлению обратной связи (Feedback Loop), что критично для производительности команды.

Событийная модель GitHub Actions позволяет реагировать на широкий спектр действий. Для задач Continuous Integration наиболее релевантными являются события push и pull\_request.6 Однако, дьявол кроется в деталях конфигурации.

Например, стандартная конфигурация:

YAML

on: [push, pull\_request]

приведет к двойному запуску пайплайна при отправке кода в ветку, для которой открыт PR (один раз на событие push в ветку, второй — на обновление PR). Это пустая трата ресурсов.

**Архитектурная рекомендация**: Для оптимизации ресурсов следует использовать фильтрацию веток и путей.

* **Push**: Ограничить веткой main (или master), так как именно это событие обычно инициирует процесс Continuous Deployment (CD) и публикацию артефактов.
* **Pull Request**: Использовать как основной триггер для CI проверок. Это реализует паттерн "Quality Gate", гарантируя, что "сломанный" код физически не сможет попасть в основную ветку, если настроены политики защиты веток (Branch Protection Rules).6
* **Path Filtering**: Исключение документации (\*\*.md) и конфигурационных файлов, не влияющих на сборку, позволяет избежать ложных запусков CI при обновлении README, что является проявлением культуры бережливого производства (Lean) в DevOps.4

### 2.3. Виртуализация и Среда Выполнения (Runners)

Выбор среды выполнения — это выбор между контролем и удобством. GitHub предоставляет **GitHub-hosted runners** — виртуальные машины, администрируемые GitHub.

Для.NET разработки стандартом де-факто является образ ubuntu-latest. Несмотря на то, что.NET исторически ассоциировался с Windows, современный.NET (Core/5/6/7/8) является кроссплатформенным. Использование Linux-раннеров предпочтительнее по нескольким причинам:

1. **Скорость**: Linux-контейнеры и виртуалки инициализируются и работают быстрее аналогов на Windows.
2. **Стоимость**: Linux-минуты в биллинге GitHub Actions стоят дешевле (или расходуются с коэффициентом 1.0 против 2.0 для Windows).
3. **Соответствие Продакшену**: Большинство современных.NET микросервисов разворачиваются в Linux-контейнерах (Docker/Kubernetes). Тестирование в среде, максимально приближенной к "боевой" (Production Parity), снижает риск возникновения платформо-специфичных багов.1

Каждый запуск на hosted-раннере происходит в "чистой комнате". Раннер — это эфемерная сущность, которая уничтожается сразу после завершения джоба. Это гарантирует отсутствие "дрейфа конфигурации" (Configuration Drift) и побочных эффектов от предыдущих сборок, что является недостижимым идеалом при ручной сборке на машине разработчика.7

## 3. Service Containers: Оркестрация Инфраструктуры в CI

Переход к интеграционному тестированию ставит перед инженером сложную задачу: как обеспечить наличие внешних зависимостей (баз данных, очередей сообщений) в эфемерной среде CI? В отличие от unit-тестов, где зависимости мокаются (mocking), интеграционные тесты требуют "живых" инстансов сервисов.

GitHub Actions предлагает элегантное решение — **Service Containers**. Это Docker-контейнеры, жизненным циклом которых управляет сам раннер GHA. Они запускаются до начала выполнения шагов джоба и останавливаются после его завершения. Это позволяет декларативно описать инфраструктуру, необходимую для тестов, прямо в YAML файле пайплайна.8

### 3.1. Сетевая Топология: Дихотомия Localhost

Понимание сетевого взаимодействия между хостом (где выполняется dotnet test) и сервисным контейнером (где живет PostgreSQL) является камнем преткновения для многих инженеров. Здесь архитектура GHA имеет два принципиально разных режима работы, зависящих от того, где выполняется сам джоб.

#### Режим 1: Джоб на Хосте (Runner Machine)

Это наиболее распространенный сценарий для.NET проектов. Джоб выполняется непосредственно на виртуальной машине (runs-on: ubuntu-latest), а сервисы — в контейнерах Docker.

В этом сценарии GitHub Actions создает "мост" между сетью контейнера и хостом. Ключевым механизмом здесь является **Port Mapping** (проброс портов).

* Контейнер сервиса (например, Postgres) экспонирует порт 5432.
* GHA маппит этот порт на порт хоста (часто тоже 5432, но может быть случайным).
* Тестовый код, запущенный на хосте, обращается к сервису через localhost:5432 или 127.0.0.1:5432.8

**Важное отличие от Docker Compose**: В Docker Compose сервисы видят друг друга по DNS-именам (например, db-host). В GHA при запуске на хосте DNS-имена контейнеров недоступны для процесса джоба. Единственный способ коммуникации — через петлевой интерфейс localhost.7

#### Режим 2: Джоб в Контейнере

Если сам джоб сконфигурирован для запуска внутри контейнера (container: mcr.microsoft.com/dotnet/sdk:8.0), то архитектура меняется.

* GHA создает пользовательскую Docker-сеть (User-defined bridge network).
* И джоб-контейнер, и сервис-контейнеры помещаются в эту общую сеть.
* В этой сети работает внутренний DNS Docker'а. localhost в джобе указывает на сам контейнер джоба, а сервисы доступны по их именам (labels) из YAML файла (например, postgres).8

Для учебного плана 4-го дня мы рассматриваем **Режим 1**, так как он позволяет использовать предустановленные на хосте инструменты (Docker CLI, утилиты) и является более прозрачным для отладки.

### 3.2. Конфигурация PostgreSQL Service: Анатомия YAML

Рассмотрим детально конфигурацию сервиса базы данных, необходимую для выполнения задания дня.

YAML

services:  
 postgres:  
 image: postgres:15-alpine  
 env:  
 POSTGRES\_USER: postgres  
 POSTGRES\_PASSWORD: password  
 POSTGRES\_DB: integration\_tests  
 ports:  
 - 5432:5432  
 options: >-  
 --health-cmd pg\_isready  
 --health-interval 10s  
 --health-timeout 5s  
 --health-retries 5

**Анализ ключевых параметров:**

1. **Image**: Использование тега alpine (например, postgres:15-alpine) является лучшей практикой (Best Practice). Образы на базе Alpine Linux весят значительно меньше (около 50-100 МБ против 300+ МБ для Debian-based), что ускоряет фазу Pulling images и сокращает общее время выполнения пайплайна.11
2. **Environment Variables**: Переменные POSTGRES\_USER, POSTGRES\_PASSWORD, POSTGRES\_DB обязательны для автоматической инициализации экземпляра. Без них контейнер может не запуститься или требовать ручной настройки.12
3. **Ports**: Директива 5432:5432 жестко связывает порт контейнера с портом хоста. Если не указать левую часть (просто - 5432), Docker назначит случайный эфемерный порт на хосте, что сделает невозможным использование статической строки подключения в тестах.8

### 3.3. Оптимизация Производительности: Файловая Система в Памяти (tmpfs)

Интеграционные тесты часто являются IO-bound задачами (ограниченными скоростью ввода-вывода), так как они интенсивно пишут и читают данные из БД. В стандартных облачных раннерах дисковая подсистема (SSD) может быть узким местом, особенно при параллельном выполнении множества тестов.

Поскольку данные в CI не требуют долгосрочного хранения (они нужны только на время жизни джоба), мы можем использовать мощную оптимизацию: перенос данных БД в оперативную память (RAM).

Docker поддерживает монтирование томов типа tmpfs. Это создает файловую систему в оперативной памяти хоста.

Для PostgreSQL папка данных по умолчанию — /var/lib/postgresql/data.

Модифицированная конфигурация options:

YAML

options: >-  
 --mount type=tmpfs,destination=/var/lib/postgresql/data  
 --health-cmd pg\_isready...

**Техническое обоснование**: Операции записи в tmpfs происходят со скоростью доступа к оперативной памяти, что на порядки быстрее SSD. Кроме того, это снижает нагрузку на диск, устраняя задержки, связанные с fsync (синхронизацией данных на диск), который PostgreSQL выполняет для гарантии целостности данных (ACID). В контексте тестов, где падение контейнера означает перезапуск всего джоба, строгим ACID можно пожертвовать ради скорости.14

Однако, следует соблюдать осторожность: стандартные раннеры GitHub имеют ограничение в 7 ГБ RAM. Если база данных разрастется до нескольких гигабайт, процесс может быть убит OOM Killer'ом (Out of Memory) ядра Linux.7

## 4. Проблема Асинхронности: Race Conditions и Health Checks

Одной из самых коварных проблем в автоматизированных системах является недетерминированность порядка запуска сервисов, известная как состояние гонки (Race Condition).

### 4.1. Феноменология Проблемы

Когда GHA запускает контейнер postgres, команда возвращает управление раннеру почти мгновенно, как только процесс внутри контейнера получил PID. Однако, "процесс запущен" не означает "сервис готов принимать соединения".

PostgreSQL требуется время на:

1. Инициализацию структур в памяти.
2. Создание системных файлов.
3. Запуск TCP-листенера.
4. Выполнение init-скриптов (создание пользователя и БД).

Если следующий шаг пайплайна (dotnet test) запустится немедленно, он попытается открыть TCP-соединение к порту 5432. Если листенера еще нет — Connection Refused. Если листенер есть, но БД не создана — ошибка аутентификации или database does not exist.

Это приводит к "мерцающим" (flaky) тестам, которые то проходят, то падают без видимых причин.7

### 4.2. Стратегия Решения: Health Checks

Для решения этой проблемы необходимо внедрить механизм ожидания готовности (Wait-for-Ready). В экосистеме Docker это реализуется через Health Checks. GitHub Actions позволяет нативно использовать их через параметр options.

#### PostgreSQL: pg\_isready

Утилита pg\_isready специально создана для проверки статуса соединения с сервером PostgreSQL.

YAML

options: >-  
 --health-cmd pg\_isready  
 --health-interval 10s  
 --health-timeout 5s  
 --health-retries 5

GitHub Actions будет опрашивать контейнер указанной командой. Шаги джоба не начнутся, пока команда не вернет код 0 (Success). Это гарантирует, что к моменту старта тестов сервер готов.17

Нюанс: Базовый вызов pg\_isready проверяет только доступность сокета. Для полной уверенности лучше проверять доступность конкретной БД и пользователя:

--health-cmd "pg\_isready -U postgres -d integration\_tests".18

#### RabbitMQ: Сложность с Erlang

RabbitMQ — это приложение на Erlang, и его запуск занимает значительно больше времени, чем Postgres. Простого открытия порта 5672 недостаточно.

Для проверки здоровья RabbitMQ рекомендуется использовать встроенную диагностическую утилиту.

YAML

services:  
 rabbitmq:  
 image: rabbitmq:3.12-management  
 ports:  
 - 5672:5672  
 options: >-  
 --health-cmd "rabbitmq-diagnostics -q ping"  
 --health-interval 10s  
 --health-timeout 5s  
 --health-retries 10

Обратите внимание на увеличенное количество ретраев (retries: 10). RabbitMQ может инициализироваться 30-40 секунд на загруженных раннерах. Обычный nc -z localhost 5672 (netcat) здесь ненадежен, так как порт может открыться до полной загрузки приложения.19

## 5. Оптимизация Сборки: Управление Зависимостями и Кэширование

В экосистеме.NET процесс восстановления пакетов NuGet (dotnet restore) может занимать существенную часть времени выполнения CI, особенно в проектах с большим количеством транзитивных зависимостей. Поскольку каждый запуск GHA начинается с "чистого листа", без кэширования пакеты будут скачиваться из интернета каждый раз.

### 5.1. Эволюция Механизмов Кэширования

Долгое время стандартом было использование универсального экшена actions/cache для сохранения папки ~/.nuget/packages. Однако это требовало ручной настройки путей и ключей кэширования, что часто приводило к ошибкам конфигурации (например, кэшированию пустых папок).

Современный подход (Best Practice на 2025 год) предполагает использование встроенных возможностей экшена setup-dotnet.

YAML

- name: Setup.NET  
 uses: actions/setup-dotnet@v4  
 with:  
 dotnet-version: 8.0.x  
 cache: true  
 cache-dependency-path: '\*\*/packages.lock.json'

### 5.2. Алгоритмика Кэширования

Когда параметр cache: true включен, setup-dotnet выполняет следующие действия:

1. **Генерация ключа**: Создается уникальный хэш на основе файлов, описывающих зависимости (обычно \*.csproj, \*.fsproj или централизованный Directory.Packages.props). Если используется режим NuGet Lock Files (packages.lock.json), хэш строится на его основе, что дает максимальную точность.21
2. **Lookup**: Перед началом сборки GHA ищет в глобальном кэше (S3-compatible storage внутри инфраструктуры GitHub) blob с таким ключом.
3. **Restore**: Если кэш найден (Hit), он распаковывается в папку глобального кэша NuGet на раннере. dotnet restore затем использует эти файлы, избегая сетевых запросов.
4. **Save**: После успешного завершения джоба, если кэш не был найден (Miss), текущее состояние папки пакетов упаковывается и загружается в кэш.

**Влияние на производительность**: Для средних проектов это сокращает время шага Restore с 1-2 минут до 5-10 секунд. Однако, важно понимать, что кэш иммутабелен (неизменяем). При изменении хотя бы одной зависимости ключ хэша меняется, и происходит полный "Cache Miss", требующий нового скачивания.23

## 6. Стратегии Интеграционного Тестирования

Наличие поднятых контейнеров — лишь половина дела. Приложение должно уметь с ними работать.

### 6.1. Конфигурация Приложения в Тестах (Overrides)

В.NET приложении настройки (Connection Strings) обычно находятся в appsettings.json. В тестах нам нужно подменить эти настройки на те, которые соответствуют окружению CI (localhost и порты контейнеров).

Существует два паттерна переопределения конфигурации:

1. **Environment Variables (Переменные окружения)**:.NET Configuration Provider по умолчанию читает переменные окружения. Мы можем задать их прямо в YAML шаге dotnet test.  
   YAML  
   - name: Test  
    env:  
    ConnectionStrings\_\_Database: "Host=localhost;Port=5432;Database=integration\_tests;Username=postgres;Password=password"  
    run: dotnet test  
     
   Использование двойного подчеркивания (\_\_) в имени переменной интерпретируется.NET как разделитель секций JSON (ConnectionStrings:Database).12
2. **WebApplicationFactory и In-Memory Configuration**: Более гибкий подход — использование WebApplicationFactory<TStartup> в коде тестов для подмены конфигурации программно.  
   C#  
   protected override void ConfigureWebHost(IWebHostBuilder builder)  
   {  
    builder.ConfigureAppConfiguration((context, config) =>  
    {  
    config.AddInMemoryCollection(new Dictionary<string, string>  
    {  
    {"ConnectionStrings:Database", "Host=localhost;..."}  
    });  
    });  
   }  
     
   Этот метод предпочтительнее для сложных сценариев, но вариант с переменными окружения в YAML проще для понимания и лучше соответствует философии "Infrastructure as Code" (IaC) — конфигурация инфраструктуры видна в файле пайплайна.25

### 6.2. Управление Данными и Изоляция Тестов

Интеграционные тесты, работающие с общей базой данных, подвержены проблеме "грязного состояния". Если Тест А создает запись пользователя, а Тест Б ожидает, что таблица пользователей пуста, порядок выполнения тестов становится критичным.

В CI это усугубляется тем, что тесты могут запускаться параллельно.

Стратегии решения:

1. **Respawn / Checkpoint**: Использование библиотеки (например, Respawn), которая очищает таблицы перед каждым тестом. Это быстро, но требует последовательного выполнения тестов.
2. **Транзакции**: Оборачивание каждого теста в TransactionScope, который не коммитится (rollback) в конце теста. Это обеспечивает отличную изоляцию, но не работает, если тестируемый код сам управляет транзакциями или создает новые потоки/соединения.
3. **Инициализация Схемы**: Критический момент — создание структуры БД. В CI база пуста. Рекомендуется вызывать context.Database.EnsureCreated() в конструкторе базового класса интеграционных тестов. Это гарантирует актуальность схемы.

**Проблема параллелизма xUnit**: По умолчанию xUnit запускает тестовые коллекции параллельно. При работе с одной БД это вызовет хаос. Для интеграционных тестов необходимо отключать параллелизм на уровне сборки `` или использовать семафоры.26

## 7. Observability: Отчетность и Анализ Сбоев

Когда CI пайплайн падает, инженер должен мгновенно понять причину. Стандартный вывод консоли (stdout) часто перегружен информацией. Необходима структурированная отчетность.

### 7.1. Формат TRX и Визуализация

.NET CLI поддерживает генерацию отчетов в формате TRX (Visual Studio Test Results XML).

Команда запуска должна включать логгер:

dotnet test --logger "trx;LogFileName=test\_results.trx"

Сам по себе TRX файл — это XML, который трудно читать человеку. Для интеграции с GitHub Actions используются специализированные Actions, которые парсят TRX и создают наглядные отчеты.

### 7.2. Паттерн "Split Workflow" для Безопасной Отчетности

Популярный экшен dorny/test-reporter создает красивые аннотации в коде (Checks API). Однако он требует прав на запись (checks: write), которые могут быть недоступны, если PR пришел из форка репозитория (по соображениям безопасности GitHub ограничивает права токенов для форков).

Для решения этой проблемы используется архитектурный паттерн разделения:

1. **Workflow 1 (Test)**: Запускает тесты и *загружает* TRX файлы как артефакт (upload-artifact). Этот workflow имеет ограниченные права.  
   YAML  
   - uses: actions/upload-artifact@v4  
    if: always() # Загружать отчеты даже если тесты упали  
    with:  
    name: test-results  
    path:TestResults/\*.trx  
     
   Использование if: always() (или `success() |

| failure()`) здесь критически важно. Без этого условия, при падении тестов (что и является самым интересным случаем) шаг загрузки артефактов будет пропущен, и отчет будет потерян.28

1. Workflow 2 (Report): Триггерится событием workflow\_run (завершение первого workflow). Он выполняется в контексте основного репозитория, имеет права на запись, скачивает артефакты и публикует отчет.  
   Этот паттерн обеспечивает безопасность и надежную визуализацию результатов.29

### 7.3. Аннотации Ошибок (Problem Matchers)

Помимо полных отчетов, GitHub умеет показывать ошибки прямо в диффе кода ("Files changed"). Для этого используются **Problem Matchers** — регулярные выражения, которые парсят лог консоли. Экшен setup-dotnet автоматически подключает матчеры для ошибок компиляции (csc). Для ошибок тестов (например, "Expected: 5, Actual: 3") можно подключить дополнительные матчеры, чтобы подсвечивать конкретные строки в файлах тестов, где произошла ошибка ассерта.30

## 8. Полная Конфигурация Пайплайна (Эталон)

Ниже представлена полная, оптимизированная конфигурация YAML для Дня 4, объединяющая все рассмотренные концепции.

YAML

name: CI & Integration Tests  
  
on:  
 push:  
 branches: [ "main" ]  
 pull\_request:  
 branches: [ "main" ]  
  
permissions:  
 contents: read  
 checks: write # Для публикации отчетов (если в рамках одного репо)  
 pull-requests: write  
  
jobs:  
 build-and-test:  
 name: Build and Test (Linux)  
 runs-on: ubuntu-latest  
   
 # ------------------------------------------------------------------  
 # Service Containers: Декларативное описание инфраструктуры  
 # ------------------------------------------------------------------  
 services:  
 postgres:  
 image: postgres:15-alpine  
 env:  
 POSTGRES\_USER: postgres  
 POSTGRES\_PASSWORD: ci\_password  
 POSTGRES\_DB: integration\_db  
 ports:  
 - 5432:5432  
 # Оптимизация: tmpfs для ускорения IO  
 options: >-  
 --mount type=tmpfs,destination=/var/lib/postgresql/data  
 --health-cmd pg\_isready  
 --health-interval 10s  
 --health-timeout 5s  
 --health-retries 5  
  
 rabbitmq:  
 image: rabbitmq:3.12-management  
 ports:  
 - 5672:5672  
 # Надежный Health Check для Erlang-приложения  
 options: >-  
 --health-cmd "rabbitmq-diagnostics -q ping"  
 --health-interval 10s  
 --health-timeout 5s  
 --health-retries 10  
  
 steps:  
 - name: Checkout Code  
 uses: actions/checkout@v4  
  
 # ------------------------------------------------------------------  
 # Настройка среды и Кэширование  
 # ------------------------------------------------------------------  
 - name: Setup.NET SDK  
 uses: actions/setup-dotnet@v4  
 with:  
 dotnet-version: 8.0.x  
 cache: true  
 cache-dependency-path: '\*\*/packages.lock.json'  
  
 - name: Restore Dependencies  
 run: dotnet restore  
  
 - name: Build  
 run: dotnet build --no-restore --configuration Release  
  
 # ------------------------------------------------------------------  
 # Запуск Тестов с Инъекцией Конфигурации  
 # ------------------------------------------------------------------  
 - name: Run Integration Tests  
 env:  
 # Переопределение настроек для подключения к Service Containers (localhost)  
 ConnectionStrings\_\_Database: "Host=localhost;Port=5432;Database=integration\_db;Username=postgres;Password=ci\_password"  
 RabbitMq\_\_Host: "localhost"  
 RabbitMq\_\_Port: "5672"  
 run: >  
 dotnet test   
 --no-build   
 --configuration Release   
 --logger "trx;LogFileName=test\_results.trx"   
 --results-directory./TestResults  
  
 # ------------------------------------------------------------------  
 # Отчетность  
 # ------------------------------------------------------------------  
 - name: Publish Test Report  
 uses: dorny/test-reporter@v1  
 if: always() # Выполнять даже при ошибках в тестах  
 with:  
 name:.NET Integration Tests Report  
 path:./TestResults/\*.trx  
 reporter: dotnet-trx  
 fail-on-error: true

## 9. Заключение: От Кода к Продукту

Реализация представленного пайплайна трансформирует процесс разработки. Вместо "черного ящика" локальной сборки команда получает прозрачный, воспроизводимый и надежный механизм проверки качества. Использование Service Containers позволяет тестировать код в условиях, максимально приближенных к реальным, выявляя интеграционные ошибки (проблемы с SQL-запросами, миграциями, форматами сообщений очереди) на ранних этапах. Оптимизации через кэширование и tmpfs демонстрируют зрелый инженерный подход к управлению ресурсами. Внедрение этих практик в рамках четвертого дня обучения закладывает фундамент для дальнейших шагов — непрерывной доставки (CD) и развертывания в Kubernetes.

#### Источники

1. План обучения CI/CD и конфигурации .NET неделя 13
2. Workflow syntax for GitHub Actions, дата последнего обращения: декабря 6, 2025, <https://docs.github.com/actions/using-workflows/workflow-syntax-for-github-actions>
3. Understanding GitHub Actions, дата последнего обращения: декабря 6, 2025, <https://docs.github.com/articles/getting-started-with-github-actions>
4. A Deep Dive Into GitHub Actions From Software Development to Data Engineering, дата последнего обращения: декабря 6, 2025, <https://dev.to/alexmercedcoder/a-deep-dive-into-github-actions-from-software-development-to-data-engineering-bki>
5. Building and testing .NET - GitHub Docs, дата последнего обращения: декабря 6, 2025, <https://docs.github.com/actions/guides/building-and-testing-net>
6. Events that trigger workflows - GitHub Docs, дата последнего обращения: декабря 6, 2025, <https://docs.github.com/actions/learn-github-actions/events-that-trigger-workflows>
7. From CI Chaos to Orchestration: Deep Dive into GitHub Actions Service Containers and Docker Compose | by Sreeprad | Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@sreeprad99/from-ci-chaos-to-orchestration-deep-dive-into-github-actions-service-containers-and-docker-compose-7cb2ff335864>
8. Communicating with Docker service containers - GitHub Docs, дата последнего обращения: декабря 6, 2025, <https://docs.github.com/actions/tutorials/communicating-with-docker-service-containers>
9. What are Service Containers - GitHub Actions - KodeKloud Notes, дата последнего обращения: декабря 6, 2025, <https://notes.kodekloud.com/docs/GitHub-Actions/Continuous-Integration-with-GitHub-Actions/What-are-Service-Containers>
10. Creating PostgreSQL service containers - GitHub Docs, дата последнего обращения: декабря 6, 2025, <https://docs.github.com/actions/guides/creating-postgresql-service-containers>
11. Integration Test Postgres using GitHub Actions - DEV Community, дата последнего обращения: декабря 6, 2025, <https://dev.to/kashifsoofi/integration-test-postgres-using-github-actions-3lln>
12. How to connect to Postgres in GitHub Actions - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/57915791/how-to-connect-to-postgres-in-github-actions>
13. example-services/.github/workflows/postgres-service.yml at main · actions/example-services · GitHub, дата последнего обращения: декабря 6, 2025, <https://github.com/actions/example-services/blob/main/.github/workflows/postgres-service.yml>
14. Optimize Postgres Containers for Testing [RE#15] | Babak K. Shandiz's Blog, дата последнего обращения: декабря 6, 2025, <https://babakks.github.io/article/2024/01/26/re-015-optimize-postgres-containers-for-testing.html>
15. Is it possible to use tmpfs on a GitHub Workflow Service? - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/75726151/is-it-possible-to-use-tmpfs-on-a-github-workflow-service>
16. Docker container port binding issue in GitHub Actions CI/CD - Latenode Official Community, дата последнего обращения: декабря 6, 2025, <https://community.latenode.com/t/docker-container-port-binding-issue-in-github-actions-ci-cd/27056>
17. How to wait for container X before starting Y using docker-compose healthcheck - GitHub, дата последнего обращения: декабря 6, 2025, <https://github.com/peter-evans/docker-compose-healthcheck>
18. Request: adding Docker health check to `postgres/README.md` · Issue #2391 - GitHub, дата последнего обращения: декабря 6, 2025, <https://github.com/docker-library/docs/issues/2391>
19. How to check rabbitMQ connection(health check) up or not? - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/69893966/how-to-check-rabbitmq-connectionhealth-check-up-or-not>
20. Actions and services feedback (health check questions) · Issue #3 - GitHub, дата последнего обращения: декабря 6, 2025, <https://github.com/actions/example-services/issues/3>
21. Caching NuGet packages in GitHub Actions - Damir's Corner, дата последнего обращения: декабря 6, 2025, <https://www.damirscorner.com/blog/posts/20240726-CachingNuGetPackagesInGitHubActions.html>
22. actions/setup-dotnet: Set up your GitHub Actions workflow with a specific version of the .NET core sdk, дата последнего обращения: декабря 6, 2025, <https://github.com/actions/setup-dotnet>
23. Dependency caching reference - GitHub Docs, дата последнего обращения: декабря 6, 2025, <https://docs.github.com/en/actions/reference/workflows-and-actions/dependency-caching>
24. Cache nuget packages on github actions : r/dotnet - Reddit, дата последнего обращения: декабря 6, 2025, <https://www.reddit.com/r/dotnet/comments/194qzj7/cache_nuget_packages_on_github_actions/>
25. Integration Testing using Testcontainers in .NET 8 | CodeNx - Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/codenx/integration-testing-using-testcontainers-in-net-8-520e8911d081>
26. Race condition problem during database initialization of some modules · Issue #19156 · abpframework/abp - GitHub, дата последнего обращения: декабря 6, 2025, <https://github.com/abpframework/abp/issues/19156>
27. [BUG] VSCode test runner seems to ignore test collections / test fixtures #825 - GitHub, дата последнего обращения: декабря 6, 2025, <https://github.com/microsoft/vscode-dotnettools/issues/825>
28. Beautiful .NET Test Reports Using GitHub Actions - SeanKilleen.com, дата последнего обращения: декабря 6, 2025, <https://seankilleen.com/2024/03/beautiful-net-test-reports-using-github-actions/>
29. How to set up a testing workflow for a .NET library in 2024 | Bitwarden, дата последнего обращения: декабря 6, 2025, <https://bitwarden.com/blog/how-to-set-up-a-testing-workflow-for-a-net-library/>
30. toolkit/docs/problem-matchers.md at main · actions/toolkit - GitHub, дата последнего обращения: декабря 6, 2025, <https://github.com/actions/toolkit/blob/main/docs/problem-matchers.md>
31. How do I convert xUnit/dotnet test output to annotations in GitHub actions using a problem matcher? - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/76884308/how-do-i-convert-xunit-dotnet-test-output-to-annotations-in-github-actions-using>