# Архитектурная спецификация и теоретический базис непрерывной интеграции (CI/CD) для Enterprise-систем: Глубокий анализ пятого дня обучения

## Аннотация

Данный документ представляет собой исчерпывающее теоретическое и практическое руководство, разработанное в рамках пятнадцатой недели программы трансформации Unity-разработчиков в инженеров Enterprise-бэкенда. Фокус исследования сосредоточен на материалах пятого дня обучения, который посвящен консолидации знаний о контейнеризации, автоматизированном тестировании и скриптинге для создания надежных локальных конвейеров непрерывной интеграции (Local CI Pipelines).

Цель документа — не просто предоставить инструкции, но сформировать глубокое инженерное понимание фундаментальных принципов, лежащих в основе надежности распределенных систем. Анализ охватывает философию CI/CD, архитектуру защитного скриптинга на Bash, внутреннее устройство управления ресурсами в Testcontainers, метрики покрытия кода и стратегии минимизации поверхности атаки в Docker-образах. Документ предназначен для хранения в корпоративной базе знаний (Google Drive) и использования в качестве эталонного стандарта при выполнении финального задания «Непотопляемый Календарь».

## Глава 1. Эпистемология Непрерывной Интеграции: Сдвиг Парадигмы

### 1.1. От кустарного производства к индустриальному конвейеру

В процессе эволюции разработчика переход от создания изолированных клиентских приложений (Unity/Desktop) к разработке распределенных бэкенд-систем требует радикального пересмотра отношения к процессу сборки и доставки кода. В экосистеме GameDev традиционно доминировала парадигма «редких, но массивных билдов», где процесс сборки мог занимать часы и часто требовал ручного вмешательства релиз-инженеров. В противоположность этому, методология Enterprise-бэкенда опирается на принципы **Непрерывной Интеграции (Continuous Integration - CI)**, где частота интеграции кода обратно коррелирует с рисками.1

Фундаментальный принцип CI гласит: «Версионируйте всё» (Version Everything).2 Это означает, что «Истиной» (Single Source of Truth) является не состояние рабочей станции ведущего разработчика, а состояние репозитория системы контроля версий. Любое изменение в коде должно триггерить автоматизированную цепочку событий, направленную на верификацию целостности системы. Этот процесс, известный как пайплайн (Pipeline), является воплощением индустриализации разработки ПО: ручной труд заменяется детерминированными алгоритмами.3

### 1.2. Экономика раннего обнаружения ошибок (Fail Fast)

Экономическая целесообразность внедрения CI объясняется концепцией «Fail Fast» (Быстрый отказ). Стоимость исправления ошибки растет экспоненциально по мере её продвижения по этапам жизненного цикла ПО: ошибка, найденная разработчиком локально, стоит копейки; ошибка, попавшая в Production, может стоить репутации и миллионов долларов.2

На пятом дне обучения мы фокусируемся на создании **локального CI**. Это критический этап, часто игнорируемый начинающими инженерами. Локальный CI (реализованный через скрипты типа verify.sh) позволяет сократить цикл обратной связи (Feedback Loop) с десятков минут (ожидание очереди на CI-сервере) до секунд. Если код не проходит верификацию на машине разработчика, он не имеет права быть отправленным (Push) в удаленный репозиторий.4

### 1.3. Архитектура пирамиды тестирования в контексте CI

Эффективный пайплайн CI невозможен без структурированной стратегии тестирования. В отличие от ручного тестирования (Play Mode в Unity), которое проверяет поведение системы «глазами пользователя», автоматизированные тесты в бэкенде делятся на уровни абстракции, формируя Пирамиду Тестирования.1

| **Уровень** | **Цель** | **Инструментарий.NET** | **Роль в локальном CI (День 5)** |
| --- | --- | --- | --- |
| **Unit Tests** | Проверка изолированной бизнес-логики (алгоритмы, валидация). Без ввода/вывода (I/O). | xUnit, NUnit | Первая линия обороны. Должны выполняться мгновенно. Блокируют дальнейшие этапы при сбое. |
| **Integration Tests** | Верификация взаимодействия компонентов с внешними зависимостями (БД, брокеры сообщений). | Testcontainers | Самый сложный этап. Требует управления жизненным циклом эфемерных контейнеров. |
| **System/E2E Tests** | Проверка собранного артефакта (Docker Image) в окружении, приближенном к боевому. | Docker Compose, cURL, Bash | Финальная проверка «черного ящика» перед релизом. Гарантия того, что образ корректно стартует. |

Интеграция этих уровней в единый исполняемый сценарий является основной задачей пятого дня. Мы переходим от запуска тестов через IDE (Visual Studio / Rider) к запуску через консольные утилиты, что является пререквизитом для автоматизации.5

## Глава 2. Архитектура надежного скриптинга: Bash как связующее звено

В мире Enterprise-разработки, несмотря на доминирование высокоуровневых языков (C#, Java, Go), «клеем», соединяющим различные инструменты (Docker,.NET CLI, git), остается Bash. Для C# разработчика это часто является зоной дискомфорта, однако написание надежных, идемпотентных скриптов — обязательный навык Senior-инженера.1

### 2.1. Парадигма «Defensive Scripting» (Защитное программирование)

По умолчанию интерпретатор Bash оптимизирован для интерактивной работы пользователя, а не для автоматизации критических процессов. Он «прощает» ошибки: если команда не найдена или вернула код ошибки, Bash просто переходит к следующей строке. В контексте CI-пайплайна такое поведение недопустимо, так как может привести к ситуации «False Positive» (Ложноположительный результат), когда билд падает, но система сообщает об успехе.7

Для трансформации Bash в надежный инструмент автоматизации необходимо использовать режим строгой обработки ошибок, активируемый директивой set -euo pipefail. Рассмотрим анатомию этой конструкции детально.

#### 2.1.1. set -e (errexit): Принцип нулевой терпимости

Флаг -e инструктирует оболочку немедленно завершить выполнение скрипта, если любая выполняемая команда возвращает ненулевой код возврата (exit code!= 0).9 В философии Unix код 0 означает успех, а любое другое число — специфическую ошибку.

Без этого флага возможен катастрофический сценарий:

1. dotnet build падает с ошибкой компиляции.
2. Скрипт игнорирует это и пытается запустить docker build.
3. Docker собирает образ на основе старых бинарных файлов.
4. В продакшн уходит код недельной давности вместо нового патча.

#### 2.1.2. set -u (nounset): Защита от неопределенности

Флаг -u заставляет скрипт падать при попытке разыменования переменной, которая не была объявлена.10 Это критически важно для предотвращения деструктивных действий из-за опечаток.

* *Сценарий риска:* Команда очистки rm -rf /${DIR\_NAME}/\*.
* *Без -u:* Если DIR\_NAME не определена (например, опечатка DIR\_NME), Bash развернет переменную в пустую строку. Команда превратится в rm -rf //\*, что может уничтожить корневую файловую систему (в зависимости от защиты OS).
* *С -u:* Скрипт аварийно завершится с сообщением unbound variable, предотвратив катастрофу.

#### 2.1.3. set -o pipefail: Целостность конвейеров

В Bash exit code конвейера (cmd1 | cmd2 | cmd3) по умолчанию равен коду возврата *последней* команды (cmd3).

* *Проблема:* Если мы выполняем dotnet test | tee test\_results.log, и тесты упали (код 1), но запись в лог прошла успешно (код 0), весь пайплайн вернет 0. CI-система посчитает, что тесты прошли.11
* *Решение:* Опция pipefail гарантирует, что если *хотя бы одна* команда в цепочке упала, весь пайплайн вернет код ошибки. Это обеспечивает честность отчетов о сборке.8

### 2.2. Управление сигналами и гарантированная очистка ресурсов (trap)

В CI-процессах скрипты часто инициируют запуск фоновых процессов или создание временных ресурсов (Docker-контейнеров, временных файлов, сетевых портов). Если скрипт прерывается аварийно (по set -e или вмешательством пользователя через Ctrl+C / SIGINT), эти ресурсы остаются «висеть», приводя к утечкам и конфликтам при последующих запусках.13

Механизм trap в Bash позволяет перехватывать псевдо-сигналы (такие как EXIT) и выполнять функцию-обработчик (cleanup handler) независимо от причины завершения скрипта.

**Архитектурный паттерн идемпотентного скрипта:**

Bash

#!/bin/bash  
set -euo pipefail  
  
# Определение функции очистки  
function cleanup {  
 echo "🧹 Запуск процедуры очистки ресурсов..."  
 # Гарантированная остановка контейнеров, даже если они не были запущены  
 docker compose down -v --remove-orphans 2>/dev/null |  
  
| true  
 # Удаление временных файлов  
 rm -f./temp\_coverage\_report.xml  
}  
  
# Регистрация перехватчика на выход из процесса  
trap cleanup EXIT  
  
echo "🚀 Начало процесса верификации..."  
# Тело скрипта  
#...

Этот паттерн реализует свойство **идемпотентности**: скрипт можно запускать бесконечное количество раз, и каждый запуск будет начинаться с чистого состояния, не оставляя «мусора» после себя.7

### 2.3. Сравнение инструментов скриптинга для CI/CD

В индустрии ведутся споры о выборе инструмента для скриптов автоматизации. Анализ источников позволяет составить сравнительную таблицу.10

| **Характеристика** | **Bash** | **Python** | **PowerShell (Core)** | **Рекомендация для Дня 5** |
| --- | --- | --- | --- | --- |
| **Нативность** | Встроен во все Linux-образы и CI-агенты. | Требует установки интерпретатора и библиотек. | Требует установки.NET Runtime. | **Bash** — из-за минимализма зависимостей. |
| **Управление процессами** | Первоклассная поддержка пайплайнов и I/O перенаправления. | Требует громоздких оберток (subprocess). | Хорошая объектная модель, но вербозный синтаксис. | **Bash** — лучший выбор для склеивания консольных команд. |
| **Обработка ошибок** | Требует явного set -euo pipefail. | Исключения (Exceptions) из коробки. | Исключения и $ErrorActionPreference. | **Bash** с правильными флагами достаточен для скриптов < 100 строк. |
| **Читаемость** | Падает с ростом сложности логики. | Высокая, легко поддерживать сложную логику. | Высокая, знакома C# разработчикам. | **Bash** — для простых линейных сценариев verify.sh. |

Для задачи «Непотопляемый Календарь» (скрипт верификации) Bash является оптимальным выбором благодаря своей вездесущности в среде Linux-контейнеров.10

## Глава 3. Оркестрация контейнеров и проблема гонки (Race Conditions)

Одной из центральных тем пятого дня является взаимодействие компонентов в распределенной системе. При переходе от монолита к микросервисам (или сервису с внешней БД) возникает проблема синхронизации запуска.

### 3.1. Анатомия процесса запуска: Running vs Ready

В Docker существует фундаментальное различие между состояниями контейнера Running и Ready.15

* **Running:** Процесс внутри контейнера (PID 1) запущен. Docker daemon считает контейнер активным.
* **Ready:** Приложение внутри контейнера выполнило инициализацию (загрузило конфиги, подняло сокеты, проверило целостность данных) и готово принимать входящие соединения.

Для сложных систем, таких как PostgreSQL, разрыв между Running и Ready может составлять от нескольких секунд до минут (в случае восстановления журнала транзакций WAL). Если приложение (сервис «Календарь») попытается подключиться к БД сразу после старта контейнера, оно получит исключение ConnectionRefused или Cannot connect to server.

### 3.2. Эволюция стратегий ожидания (Wait Strategies)

Исторически инженеры решали эту проблему с помощью самописных скриптов типа wait-for-it.sh, которые в цикле проверяли доступность TCP-порта.16

* *Недостаток:* Доступность порта (TCP ACK) не гарантирует готовность приложения. Сервер БД может открыть порт, но все еще инициализировать таблицы, отвергая запросы.

Современный подход, обязательный к применению в рамках 5-го дня, базируется на нативных **Healthchecks** Docker Compose.15

**Спецификация реализации Healthcheck для PostgreSQL:**

YAML

services:  
 db:  
 image: postgres:15  
 healthcheck:  
 # Использование нативной утилиты pg\_isready внутри контейнера  
 test:  
 interval: 5s # Как часто проверять  
 timeout: 5s # Сколько ждать ответа  
 retries: 5 # Сколько раз ошибиться перед статусом Unhealthy  
 start\_period: 10s # Льготный период на старте

### 3.3. Декларативное управление зависимостями

Использование Healthcheck позволяет трансформировать императивную логику ожидания в декларативную конфигурацию depends\_on.15

YAML

services:  
 calendar-api:  
 depends\_on:  
 db:  
 condition: service\_healthy # Ждать не просто старта, а "Зеленого" статуса

В скрипте verify.sh это позволяет использовать команду docker compose up -d --wait. Флаг --wait заставляет Docker клиент блокировать выполнение скрипта до тех пор, пока все сервисы не перейдут в состояние Healthy. Это элегантное решение полностью устраняет проблему "Race Condition" при запуске E2E тестов и является "Best Practice" для современных CI-пайплайнов.16

## Глава 4. Стратегия оптимизации контекста сборки (Build Context)

В задании пятого дня особое внимание уделяется оптимизации Dockerfile. Однако оптимизация начинается еще до первой команды FROM.

### 4.1. Механизм передачи контекста

При выполнении команды docker build. Docker CLI архивирует *все* содержимое текущей директории (Build Context) и отправляет его демону Docker. В мире.NET разработки папки проекта часто содержат тяжелые артефакты: bin/, obj/, .git/, .vs/, файлы локальных тестов. Размер контекста может достигать сотен мегабайт или даже гигабайт.18

**Негативные последствия большого контекста:**

1. **Замедление CI:** Передача гигабайта данных демону занимает время, особенно если демон находится на удаленной машине.
2. **Инвалидация кэша:** Если в контекст попадают файлы, которые меняются при каждой сборке (например, логи или временные файлы), Docker может ложно решить, что исходные данные изменились, и сбросить кэш слоев.20
3. **Безопасность:** Риск случайного включения в образ файлов с секретами (.env, приватные ключи).20

### 4.2. Роль и синтаксис .dockerignore

Файл .dockerignore работает аналогично .gitignore, но для демона Docker. Правильная настройка этого файла — первый шаг к профессиональному Dockerfile.21

**Рекомендованная конфигурация для.NET проекта:**

\*\*/.git

\*\*/.vs

\*\*/.vscode

\*\*/bin

\*\*/obj

\*\*/TestResults

\*\*/.user

\*\*/.lock.json

.env

Dockerfile

docker-compose.yml

verify.sh

Исключение папок bin и obj критически важно, так как мы должны гарантировать, что сборка происходит внутри контейнера с нуля ("Clean Build"), а не использует артефакты с машины разработчика, которые могут быть несовместимы с Linux-средой контейнера.18

### 4.3. Multi-stage Builds: Разделение сред

Согласно заданию 1, финальный образ должен быть оптимизирован. Паттерн **Multi-stage build** позволяет использовать разные базовые образы для разных этапов.22

1. **Build Stage:** Использует "тяжелый" образ SDK (mcr.microsoft.com/dotnet/sdk), содержащий весь инструментарий (MSBuild, Roslyn, NuGet). Здесь происходит компиляция.
2. **Runtime Stage:** Использует минималистичный образ Runtime (mcr.microsoft.com/dotnet/aspnet), содержащий только CLR и необходимые нативные библиотеки.

В финальный образ копируются только скомпилированные DLL файлы из Build Stage. Это снижает размер образа с ~800 Мб до ~100-150 Мб и удаляет исходный код из продакшн-окружения, что является требованием безопасности.1

## Глава 5. Инженерные аспекты Интеграционного Тестирования

Пятый день знаменует отказ от имитации (Mocking) базы данных в пользу работы с реальной инфраструктурой.

### 5.1. Критика In-Memory Database в Enterprise

Библиотека Microsoft.EntityFrameworkCore.InMemory долгое время была популярна для тестирования. Однако в Enterprise-разработке ее использование считается анти-паттерном для интеграционных тестов.23

**Фундаментальные отличия и риски:**

* **Реляционная целостность:** In-Memory база не обеспечивает проверку Foreign Keys и других Constraints так, как это делает реальная СУБД. Тест может пройти успешно, записав некорректные данные, но в продакшене код упадет.23
* **Специфичный SQL:** Использование специфичных для PostgreSQL функций (JSONB операции, ILIKE, массивы) невозможно или работает иначе в In-Memory провайдере.24
* **Изоляция транзакций:** Поведение транзакций в памяти отличается от MVCC (Multiversion Concurrency Control) в PostgreSQL.

Вывод: In-Memory тесты дают ложное чувство безопасности ("False Negative"). Для надежности ("True Positive") необходимо тестировать на том же движке, который используется в продакшене.25

### 5.2. Testcontainers: Управление эфемерными средами

Библиотека **Testcontainers** решает проблему сложности настройки Docker для тестов. Она позволяет управлять контейнерами программно из C# кода.24

#### 5.2.1. Механизм Ryuk (Resource Reaper)

Критическим аспектом работы Testcontainers является очистка ресурсов. Если процесс тестов (xUnit runner) аварийно завершится (Crash), контейнеры базы данных могли бы остаться работать вечно, потребляя ресурсы CI-агента.

Для предотвращения этого Testcontainers использует паттерн "Sidecar" — запускает специальный служебный контейнер **Ryuk**.27

* При старте тестов клиентская библиотека устанавливает TCP-соединение с Ryuk.
* Ryuk мониторит это соединение.
* Если соединение разрывается (процесс тестов умер), Ryuk начинает агрессивную очистку: удаляет все контейнеры, сети и тома, помеченные лейблами текущей сессии.
* Это гарантирует стерильность среды даже в случае фатальных сбоев.

*Инсайт:* В некоторых CI-средах (например, с жесткими политиками безопасности) запуск привилегированного контейнера Ryuk может быть заблокирован. В таких случаях необходимо явно конфигурировать стратегию очистки или отключать Ryuk через переменную TESTCONTAINERS\_RYUK\_DISABLED=true, принимая на себя риски ручной очистки.28

#### 5.2.2. Сетевая магия: Динамический маппинг портов

При параллельном запуске тестов (особенно в CI) нельзя полагаться на фиксированные порты (например, 5432). Если два джоба запустятся одновременно на одном агенте, возникнет конфликт портов.25

Testcontainers решает это, пробрасывая порт контейнера на *случайный свободный* порт хоста.

* Внутри контейнера: 5432.
* На хосте: 32768 (например).  
  Приложение получает строку подключения динамически во время выполнения теста: container.GetConnectionString(). Это обеспечивает полную изоляцию параллельных процессов тестирования.30

## Глава 6. Метрики Качества и Code Coverage

В задании требуется интегрировать отчет о покрытии кода. Понимание нюансов метрик покрытия необходимо для установки адекватных Quality Gates.

### 6.1. Line Coverage vs. Branch Coverage

Большинство инструментов по умолчанию показывают **Line Coverage** (покрытие строк). Это метрика "тщеславия". Можно выполнить 100% строк метода, но пропустить критические логические ветвления (например, отсутствие else блока или проверки на null).31

**Branch Coverage** (покрытие ветвлений) — более строгая метрика. Она требует, чтобы каждое логическое условие (if, while, тернарный оператор) было вычислено как в true, так и в false.31

* *Пример:* var result = isSuccess? "Ok" : "Fail";
* Один прогон теста даст 100% Line Coverage, но только 50% Branch Coverage.
* Для задачи "Непотопляемый Календарь" необходимо стремиться к максимизации именно Branch Coverage.

### 6.2. Проблема Async/Await и скрытая сложность

В C# использование async/await приводит к тому, что компилятор Roslyn генерирует сложный класс конечного автомата (State Machine) для управления асинхронностью. Этот сгенерированный код содержит множество ветвлений и проверок состояний, которые отсутствуют в исходном коде разработчика.34

Инструменты покрытия (Coverlet) анализируют IL-код. В результате они видят эти скрытые ветвления. Поскольку разработчик не может написать тесты для внутренних переходов автомата компилятора, метрика Branch Coverage для асинхронных методов часто искусственно занижается (невозможно достичь 100%).36

*Практический вывод:* При настройке CI Quality Gate для асинхронного кода не следует ставить порог 100% для Branch Coverage. Значения 80-90% являются более реалистичными и отражают покрытие бизнес-логики, а не шума компилятора.33

### 6.3. Интеграция в CI: Fail Build on Threshold

Скрипт верификации должен не просто генерировать отчет, но и падать, если покрытие недостаточно. В.NET это реализуется через параметры MSBuild для Coverlet.37

Bash

dotnet test \  
 /p:CollectCoverage=true \  
 /p:Threshold=80 \  
 /p:ThresholdType=branch \  
 /p:CoverletOutputFormat=cobertura

Если покрытие ветвлений упадет ниже 80%, команда dotnet test вернет ненулевой код возврата, и благодаря set -e, весь скрипт verify.sh остановится, сигнализируя о проблеме качества.

## Глава 7. Реализация Финального Задания: «Непотопляемый Календарь»

Консолидируя теорию, мы подходим к архитектуре финального решения. Задача — создать "черный ящик", который на входе получает исходный код, а на выходе дает гарантию работоспособности.1

### 7.1. Спецификация скрипта verify.sh

Скрипт должен реализовывать следующий алгоритм действий, каждый шаг которого обоснован теорией выше:

1. **Bootstrapping:**
   * Активация set -euo pipefail.
   * Проверка пререквизитов (наличие Docker,.NET SDK).
2. **Static Analysis & Unit Tests:**
   * Запуск быстрых модульных тестов.
   * Fail Fast: если логика сломана, нет смысла тратить время на сборку Docker.
3. **Integration Testing (White Box):**
   * Запуск тестов с Testcontainers.
   * Генерация отчетов покрытия.
   * Проверка порогов (Thresholds).
4. **Artifact Assembly:**
   * Сборка Docker-образа приложения (docker build).
   * Здесь проверяется корректность Dockerfile и возможность сборки в изолированной среде.
5. **Environment Orchestration (Black Box Smoke Test):**
   * Запуск полного окружения через docker compose up -d --wait.
   * Это проверяет конфигурацию сети, переменных окружения и совместимость с реальной БД.
6. **Verification:**
   * Выполнение HTTP-запроса к API (curl -f http://localhost:8080/health).
   * Флаг -f (fail) в curl важен для поддержки set -e.
7. **Teardown:**
   * Очистка через trap.

### 7.2. Переход к облачным CI системам

Локальный скрипт verify.sh — это прототип пайплайна для GitHub Actions или GitLab CI. В облаке шаги будут идентичны, но будут выполняться в декларативном YAML-формате.39

Использование Testcontainers в GitHub Actions упрощает workflow, так как отпадает необходимость в секции services (Service Containers). Тесты становятся самодостаточными: они сами поднимают нужные сервисы, используя Docker-сокет хоста.39 Это делает миграцию с локальной машины в облако бесшовной.

## Заключение

Пятый день 15-й недели обучения является поворотным моментом. Инженер переходит от написания кода, который "просто работает", к созданию систем, которые "доказывают свою работоспособность".

Внедрение практик защитного скриптинга, контейнерной оркестрации и строгого контроля метрик покрытия создает фундамент для перехода к микросервисной архитектуре на следующей неделе. Без автоматизированной верификации (CI), управление сложностью распределенной системы (RabbitMQ, множество сервисов) станет невозможным. Созданный скрипт verify.sh и инфраструктура тестов — это главные артефакты, демонстрирующие зрелость инженера и готовность к работе в Enterprise-среде.

#### Источники

1. План обучения: Docker и Тестирование неделя 11
2. 8 Essential CI/CD Best Practices for 2025 | DocuWriter.ai, дата последнего обращения: декабря 6, 2025, <https://www.docuwriter.ai/posts/ci-cd-best-practices>
3. CI/CD best practices: Our top 15 tips | Gatling Blog, дата последнего обращения: декабря 6, 2025, <https://gatling.io/blog/ci-cd-best-practices>
4. Best Practices for Successful CI/CD | TeamCity CI/CD Guide - JetBrains, дата последнего обращения: декабря 6, 2025, <https://www.jetbrains.com/teamcity/ci-cd-guide/ci-cd-best-practices/>
5. Optimizing Development: A Comprehensive Guide to CI/CD Best Practices, дата последнего обращения: декабря 6, 2025, <https://www.nucamp.co/blog/coding-bootcamp-back-end-with-python-and-sql-optimizing-development-a-comprehensive-guide-to-cicd-best-practices>
6. 25 common Linux bash script examples to get you started - Hostinger, дата последнего обращения: декабря 6, 2025, <https://www.hostinger.com/tutorials/bash-script-example>
7. Building a Production-Grade Automated Deployment Script - DEV Community, дата последнего обращения: декабря 6, 2025, <https://dev.to/ursulaonyi/building-a-production-grade-automated-deployment-script-3fgj>
8. set -e in Bash: Why Your Script Fails Without Warning | Xygeni, дата последнего обращения: декабря 6, 2025, <https://xygeni.io/blog/set-e-in-bash-why-your-script-fails-without-warning/>
9. Force docker-compose to rebuild a container - Technical Support - Toradex Community, дата последнего обращения: декабря 6, 2025, <https://community.toradex.com/t/force-docker-compose-to-rebuild-a-container/28545>
10. Best practices we need to follow in Bash scripting in 2025… | by Prasanna Kumar Yempada, дата последнего обращения: декабря 6, 2025, <https://medium.com/@prasanna.a1.usage/best-practices-we-need-to-follow-in-bash-scripting-in-2025-cebcdf254768>
11. Dockerfile best practices: clarify behavior of pipefail in RUN steps · Issue #11427 · docker/docs - GitHub, дата последнего обращения: декабря 6, 2025, <https://github.com/docker/docker.github.io/issues/11427>
12. Jenkins pipeline sh: always use pipefail - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/47416617/jenkins-pipeline-sh-always-use-pipefail>
13. Using Trap to Exit Bash Scripts Cleanly - Putorius, дата последнего обращения: декабря 6, 2025, <https://www.putorius.net/using-trap-to-exit-bash-scripts-cleanly.html>
14. Python vs Bash scripts in CI/CD pipeline : r/devops - Reddit, дата последнего обращения: декабря 6, 2025, <https://www.reddit.com/r/devops/comments/oizpkz/python_vs_bash_scripts_in_cicd_pipeline/>
15. Control startup and shutdown order with Compose - Docker Docs, дата последнего обращения: декабря 6, 2025, <https://docs.docker.com/compose/how-tos/startup-order/>
16. Wait for Services to Start in Docker Compose: wait-for-it vs Healthcheck | by Pavel Loginov, дата последнего обращения: декабря 6, 2025, <https://medium.com/@pavel.loginov.dev/wait-for-services-to-start-in-docker-compose-wait-for-it-vs-healthcheck-e0248f54962b>
17. Forget wait-for-it, use docker-compose healthcheck and depends\_on instead - denhox.com, дата последнего обращения: декабря 6, 2025, <https://www.denhox.com/posts/forget-wait-for-it-use-docker-compose-healthcheck-and-depends-on-instead/>
18. Mastering the .dockerignore File: Boosting Docker Build Efficiency | by Fedi Bounouh, дата последнего обращения: декабря 6, 2025, <https://medium.com/@bounouh.fedi/mastering-the-dockerignore-file-boosting-docker-build-efficiency-398719f4a0e1>
19. build context for docker image very large - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/26600769/build-context-for-docker-image-very-large>
20. Master the dockerignore File To Double Efficiency - CyberPanel, дата последнего обращения: декабря 6, 2025, <https://cyberpanel.net/blog/dockerignore-file>
21. Build context - Docker Docs, дата последнего обращения: декабря 6, 2025, <https://docs.docker.com/build/concepts/context/>
22. Building best practices - Docker Docs, дата последнего обращения: декабря 6, 2025, <https://docs.docker.com/build/building/best-practices/>
23. Integration Tests with In Memory DB vs Real DB on Docker : r/dotnet - Reddit, дата последнего обращения: декабря 6, 2025, <https://www.reddit.com/r/dotnet/comments/1603gul/integration_tests_with_in_memory_db_vs_real_db_on/>
24. What is Testcontainers, and why should you use it?, дата последнего обращения: декабря 6, 2025, <https://testcontainers.com/guides/introducing-testcontainers/>
25. Testcontainers Best Practices - Docker, дата последнего обращения: декабря 6, 2025, <https://www.docker.com/blog/testcontainers-best-practices/>
26. Getting started with Testcontainers for .NET, дата последнего обращения: декабря 6, 2025, <https://testcontainers.com/guides/getting-started-with-testcontainers-for-dotnet/>
27. Ryuk the Resource Reaper - Worldline Engineering Blog, дата последнего обращения: декабря 6, 2025, <https://blog.worldline.tech/2023/01/04/ryuk.html>
28. Resource Reaper - Testcontainers for .NET, дата последнего обращения: декабря 6, 2025, <https://dotnet.testcontainers.org/api/resource_reaper/>
29. Continuous Integration - Testcontainers for .NET, дата последнего обращения: декабря 6, 2025, <https://dotnet.testcontainers.org/cicd/>
30. Networking and communicating with containers - Testcontainers for Java, дата последнего обращения: декабря 6, 2025, <https://java.testcontainers.org/features/networking/>
31. Differences between Line and Branch coverage - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/8229236/differences-between-line-and-branch-coverage>
32. Test Coverage Analysis with Coverlet in .NET | by Victor Magalhães - Medium, дата последнего обращения: декабря 6, 2025, <https://victormagalhaes-dev.medium.com/test-coverage-analysis-with-coverlet-in-net-2e38df3c6ed7>
33. How much code coverage is enough? Best practices for coverage - Graphite, дата последнего обращения: декабря 6, 2025, <https://graphite.com/guides/code-coverage-best-practices>
34. Incorrect branch coverage with async/await test · Issue #623 - GitHub, дата последнего обращения: декабря 6, 2025, <https://github.com/tonerdo/coverlet/issues/623>
35. Code coverage for .NET Core projects with async methods is not that straightforward … for now | Emil Craciun, дата последнего обращения: декабря 6, 2025, <https://emilcraciun.net/posts/code-coverage/>
36. Branch coverage Issue - async await #1177 - GitHub, дата последнего обращения: декабря 6, 2025, <https://github.com/coverlet-coverage/coverlet/issues/1177>
37. Dotnet Unit test with Coverlet- How to get coverage for entire solution and not just a project, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/53255065/dotnet-unit-test-with-coverlet-how-to-get-coverage-for-entire-solution-and-not>
38. Threshold value not working · Issue #48 · coverlet-coverage/coverlet - GitHub, дата последнего обращения: декабря 6, 2025, <https://github.com/tonerdo/coverlet/issues/48>
39. Running Testcontainers Tests Using GitHub Actions and Testcontainers Cloud - Docker, дата последнего обращения: декабря 6, 2025, <https://www.docker.com/blog/running-testcontainers-tests-using-github-actions/>
40. 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>