МИНИСТЕРСТВО ПРОСВЕЩЕНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«РОССИЙСКИЙ ГОСУДАРСТВЕННЫЙ

ПЕДАГОГИЧЕСКИЙ УНИВЕРСИТЕТ им. А. И. ГЕРЦЕНА»



Направление подготовки   
09.03.01 Информатика и вычислительная техника

Направленность (профиль)

«Технологии разработки программного обеспечения»

**Выпускная квалификационная работа**

Разработка и внедрение CI/CD пайплайна для автоматизации развертывания веб-приложения по подсчету калорий

|  |
| --- |
| **Обучающейся 4 курса**  **очной формы обучения**  **Мельниковой Алены Сергеевны** |
|  |
| **Руководитель выпускной квалификационной работы:**  **кандидат физико-математических наук,**  **доцент кафедры ИТиЭО**  **Власов Дмитрий Викторович** |
|  |

Санкт-Петербург

2025

# СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 3](#_Toc198935785)

[ГЛАВА 1. ТЕОРЕТИЧЕСКИЕ ОСНОВЫ DEVOPS И CI/CD 5](#_Toc198935786)

[1.1 Подходы к разработке программного обеспечения 5](#_Toc198935787)

[1.2 Понятие DevOps и его роль в современном IT 5](#_Toc198935788)

[1. 3 Влияние DevOps на бизнес-показатели 7](#_Toc198935789)

[1.4 Подход CI/CD: непрерывная интеграция и доставка 8](#_Toc198935790)

[1.5 Сравнение CI/CD систем 9](#_Toc198935791)

[1.6 Роль инструментов Infrastructure as Code (IaC) 9](#_Toc198935792)

[1.7 Обзор используемых технологий 10](#_Toc198935793)

[1.8 Обоснование архитектурного подхода 10](#_Toc198935794)

[ГЛАВА 2. 12](#_Toc198935795)

[2.1 Цели и структура MVP 12](#_Toc198935796)

[2.2 Архитектура приложения 12](#_Toc198935797)

[2.3 Реализация backend 14](#_Toc198935798)

[2.4 Реализация frontend 16](#_Toc198935799)

[2.5 Контейнеризация и запуск 17](#_Toc198935800)

[2.6 Результаты и проверка 19](#_Toc198935801)

[ГЛАВА 3. РЕАЛИЗАЦИЯ CI/CD И ПОСТРОЕНИЕ ИНФРАСТРУКТУРЫ РАЗВЕРТЫВАНИЯ 21](#_Toc198935802)

[3.1 Цели и задачи автоматизации 21](#_Toc198935803)

[3.2 Выбор GitHub Actions 21](#_Toc198935804)

[3.3 Пайплайны проекта 21](#_Toc198935805)

[3.4 Серверная инфраструктура 24](#_Toc198935806)

[3.5 Безопасность и переменные 26](#_Toc198935807)

[3.6 Результаты автоматизации 27](#_Toc198935808)

[ЗАКЛЮЧЕНИЕ 31](#_Toc198935809)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ 34](#_Toc198935810)

## **ВВЕДЕНИЕ**

Развитие веб-приложений и увеличение темпов выпуска новых версий программных продуктов требуют надёжных и автоматизированных методов поставки кода. Одной из ключевых практик в современной разработке является внедрение процессов CI/CD (непрерывной интеграции и доставки), позволяющих автоматизировать сборку, тестирование и развертывание программного обеспечения.

Актуальность темы обусловлена необходимостью ускорения релизов, уменьшения человеческого фактора при деплое и обеспечения стабильной и воспроизводимой среды. Особенно важным это становится в проектах с микросервисной архитектурой, где компоненты могут разрабатываться и обновляться независимо.

Целью дипломной работы является разработка и внедрение CI/CD-пайплайна для автоматизации развертывания веб-приложения по подсчёту калорий с разделением на dev- и prod-окружения.

Для достижения поставленной цели необходимо решить следующие задачи:

1. Разработать MVP веб-приложения, включающего frontend и backend.
2. Упаковать компоненты в Docker-контейнеры.
3. Настроить систему docker-compose для локального и серверного запуска.
4. Настроить отдельные окружения для разработки (dev) и продакшна (prod).
5. Реализовать CI/CD пайплайн через GitHub Actions.
6. Обеспечить автоматическую доставку на dev-стенд и ручной деплой на prod.
7. Расширить функционал приложения и протестировать устойчивость пайплайна.

Объектом исследования является процесс автоматизации развертывания программных продуктов.

Предмет исследования — методы и инструменты построения CI/CD пайплайнов для микросервисных веб-приложений с использованием Docker и GitHub Actions.

В работе применяются:

1. методы анализа и синтеза архитектуры приложений;
2. практика контейнеризации компонентов с помощью Docker;
3. использование инструментов CI/CD (GitHub Actions);
4. настройка микросервисной инфраструктуры и среды исполнения;
5. эмуляция производственного процесса релизов.

Работа состоит из введения, трёх глав, заключения, списка использованной литературы и приложений.

В первой главе рассматриваются теоретические основы DevOps и CI/CD, а также обоснование выбора технологий.

Во второй главе описана реализация MVP, контейнеризация и настройка инфраструктуры.

В третьей главе представлена настройка пайплайна, двух окружений, CI/CD, а также тестирование и оценка результата.

# ГЛАВА 1. ТЕОРЕТИЧЕСКИЕ ОСНОВЫ DEVOPS И CI/CD

## 1.1 Подходы к разработке программного обеспечения

История развития подходов к разработке программного обеспечения прошла несколько этапов, каждый из которых повлиял на формирование современных практик DevOps.

На начальном этапе широко применялась модель "водопада" (Waterfall), которая предполагала последовательное выполнение фаз: анализ требований, проектирование, реализация, тестирование и сопровождение. Основным недостатком этой модели было отсутствие гибкости: любые изменения вносились с трудом, а процесс релиза занимал месяцы.

С развитием гибких методологий появилась модель Agile. Agile предложил более гибкий подход с короткими итерациями, тесной связью с заказчиком и регулярными поставками работающего продукта. Однако при этом сохранялась проблема разделения на команды разработки и эксплуатации, что замедляло выход новых функций в продуктив.

DevOps возник как ответ на эти ограничения и стал логичным продолжением развития Agile. Он направлен на объединение команд разработки и эксплуатации, автоматизацию процессов и повышение скорости релизов за счёт тесного взаимодействия всех участников проекта. Таким образом, DevOps стал важным этапом эволюции процессов создания программного обеспечения, сочетая в себе лучшие практики Agile и акцентируя внимание на непрерывной поставке ценности пользователю.

## 1.2 Понятие DevOps и его роль в современном IT

Современное программное обеспечение развивается в условиях высокой конкуренции, что требует от команд разработки постоянного совершенствования процессов поставки, тестирования и поддержки кода. В этих условиях традиционные подходы, при которых разработчики и операционные специалисты действуют разрозненно, теряют актуальность. Ответом на эти вызовы стал методологический подход DevOps (сокращение от Development и Operations), направленный на интеграцию процессов разработки и эксплуатации в единую непрерывную цепочку.

DevOps предполагает тесное сотрудничество между всеми участниками жизненного цикла приложения: разработчиками, тестировщиками, системными администраторами и специалистами по безопасности. Основная цель DevOps — автоматизация и ускорение процессов поставки программного обеспечения при сохранении высокого качества и стабильности продукта.

Ключевые принципы DevOps:

* Непрерывная интеграция и поставка (CI/CD);
* Автоматизация тестирования и сборки;
* Инфраструктура как код (Infrastructure as Code);
* Мониторинг и обратная связь (Observability);
* Культура совместной ответственности (shared ownership).
* DevOps способствует:
* Сокращению цикла выпуска новых версий;
* Снижению количества ошибок за счёт автоматизации;
* Повышению устойчивости и отказоустойчивости систем;
* Быстрому отклику на изменения требований бизнеса.

Сравнение традиционной модели и подхода DevOps

В традиционной модели разработки программного обеспечения команды разработчиков и операционных специалистов существовали изолированно. Разработчики писали код и передавали его команде эксплуатации, которая затем отвечала за развертывание и поддержку. Такой подход создавал множество проблем:

1. низкую скорость выпуска релизов,
2. частые конфликты между командами,
3. сложности в масштабировании и устранении ошибок.

С переходом к DevOps происходит кардинальное изменение культурной парадигмы. DevOps выступает не только как набор инструментов, но и как философия совместной работы. В этой модели:

1. команды объединяются и совместно отвечают за продукт на всех этапах;
2. внедряются практики автоматизации тестирования, сборки и развертывания;
3. особое внимание уделяется мониторингу и быстрой обратной связи от пользователей.

Таким образом, DevOps как культура фокусируется на доверии, прозрачности, автоматизации и постоянном улучшении, что позволяет организациям быстрее адаптироваться к изменениям рынка и достигать высокого качества выпускаемого продукта.

## 1. 3 Влияние DevOps на бизнес-показатели

Внедрение DevOps оказывает значительное влияние на ключевые бизнес-показатели, особенно в сфере IT-продуктов и веб-приложений. Одним из главных преимуществ является увеличение частоты выпусков (deployment frequency). В традиционных подходах релизы происходят с периодичностью от нескольких недель до месяцев, в то время как компании, внедрившие DevOps, могут выпускать обновления ежедневно или даже по нескольку раз в день.

Другим важным показателем является сокращение времени отклика на инциденты (mean time to recovery, MTTR). DevOps-подходы обеспечивают быстрое обнаружение ошибок, благодаря автоматическим тестам и мониторингу, и позволяют оперативно вернуть систему в рабочее состояние.

Кроме того, повышается стабильность релизов (change failure rate), поскольку автоматизация процессов и интеграция проверки кода снижают вероятность ошибок, связанных с человеческим фактором. Также снижается стоимость сопровождения за счёт стандартизации инфраструктуры и использования IaC.

Исследования, такие как ежегодный отчет DORA (DevOps Research and Assessment), подтверждают, что организации с высоким уровнем внедрения DevOps достигают:

* в 46 раз более высокой частоты развертывания;
* в 440 раз более короткого времени отклика на инциденты;
* в 5 раз меньшего уровня сбоев при релизах.

Таким образом, DevOps становится не просто технической инициативой, а стратегическим направлением, влияющим на конкурентоспособность бизнеса и удовлетворённость конечных пользователей.

## 1.4 Подход CI/CD: непрерывная интеграция и доставка

CI/CD — это совокупность практик, направленных на обеспечение постоянной интеграции кода, его автоматического тестирования и доставки на сервер. CI (Continuous Integration) означает регулярное объединение кода всех разработчиков в общую ветку. CD (Continuous Delivery / Deployment) предполагает автоматическую отправку проверенного кода в рабочую среду либо вручную, либо по событию.

Этапы CI/CD пайплайна:

1. Событие в Git (push, PR);
2. Сборка и тесты;
3. Проверка качества кода;
4. Сборка Docker-образа;
5. Деплой на окружение.

Преимущества подхода:

1. Повышение скорости релизов;
2. Минимизация риска ручных ошибок;
3. Обратная связь от пользователей на каждом этапе;
4. Улучшение совместной работы команд.

## 1.5 Сравнение CI/CD систем

Основные сравнительные характеристики CI/CD систем представлены в таблице 1.1.

**Таблица 1.1 – Основные сравнительные характеристики CI/CD систем**

|  |  |  |  |
| --- | --- | --- | --- |
| **Характеристика** | **GitHub Actions** | **GitLab CI** | **Jenkins** |
| Хостинг | Облако | Облако / локально | Локально |
| Простота настройки | Очень высокая | Средняя | Низкая |
| Язык конфигурации | YAML | YAML | Groovy (DSL) |
| Поддержка Docker | Да | Да | Да |
| UI | Современный | Средний | Устаревший |

GitHub Actions был выбран для проекта благодаря его полной интеграции с GitHub, простоте настройки и возможности деплоя на удалённые серверы с помощью SSH.

## 1.6 Роль инструментов Infrastructure as Code (IaC)

Одним из важнейших компонентов практики DevOps является концепция инфраструктуры как кода (Infrastructure as Code, IaC). Этот подход заключается в том, что конфигурация инфраструктуры и серверов описывается в виде программного кода, который можно версионировать, тестировать и автоматически применять.

IaC позволяет:

* быстро и последовательно разворачивать окружения;
* управлять конфигурацией серверов и приложений;
* повышать надёжность и снижать вероятность ошибок;
* интегрировать управление инфраструктурой в CI/CD-процессы.

Наиболее популярные инструменты для IaC включают:

* Terraform — инструмент для управления инфраструктурой в облаках (AWS, Azure, GCP и др.) с использованием декларативного языка конфигурации;
* Ansible — инструмент для автоматизации конфигурации и управления системами, использующий YAML-плейбуки и принцип push-модели без необходимости установки агентов.

Хотя в рамках данного проекта эти инструменты не применялись непосредственно, они являются логическим продолжением эволюции DevOps-практик и могут быть интегрированы на следующих этапах развития проекта для улучшения масштабируемости и управляемости серверной инфраструктуры.

## 1.7 Обзор используемых технологий

Для реализации DevOps-подхода и CI/CD в рамках проекта использовались следующие технологии:

* Docker — для контейнеризации приложений;
* Docker Compose — для управления многоконтейнерной инфраструктурой;
* GitHub Actions — для автоматизации CI/CD;
* Nginx — для проксирования запросов;
* Python/Django — для серверной части API;
* React — для построения интерфейса пользователя.

Каждая из технологий обеспечивает изоляцию, масштабируемость и переносимость компонентов.

## 1.8 Обоснование архитектурного подхода

Проект построен по микросервисной архитектуре с раздельным управлением компонентами frontend и backend. Каждая часть обёрнута в собственный Docker-контейнер и может разрабатываться, тестироваться и развёртываться независимо. Компоненты взаимодействуют через REST API.

В качестве среды развёртывания используются два VPS-сервера: dev и prod. GitHub Actions запускает пайплайны, подключается по SSH и запускает команды деплоя. Все параметры конфигурации выносятся в .env файлы и GitHub Secrets.

Такой подход обеспечивает:

1. Быстрый отклик на изменения кода;
2. Минимизацию ошибок при релизе;
3. Масштабируемость при росте команды или нагрузки;
4. Воспроизводимость среды при разработке и проде.

Глава 1 создаёт теоретическую базу и обосновывает выбор решений, на которых строится весь проект автоматизации.

# ГЛАВА 2.

## 2.1 Цели и структура MVP

Минимально жизнеспособный продукт (MVP, от англ. Minimum Viable Product) — это версия приложения, обладающая минимально необходимым набором функций для запуска и получения обратной связи от пользователей. В рамках проекта целью MVP стало создание функционального веб-приложения, позволяющего пользователю фиксировать съеденные продукты и рассматривать их калорийность.

Основные функции:

* Создание новой записи о приёме пищи;
* Указание калорийности продукта;
* Хранение данных о приёмах пищи с датой и временем;
* Просмотр и фильтрация записей по дате.

Разработка MVP служит не только для отладки основной логики, но и для тестирования инфраструктурных решений: контейнеризации, взаимодействия между фронтендом и бэкендом, а также подготовки среды для CI/CD.

## 2.2 Архитектура приложения

Структура проекта

Проект состоит из двух основных каталогов: backend-app и frontend, каждый из которых содержит независимую структуру файлов и модулей.

Backend (Django):

* backend-app/ — корневая директория серверной части.
  + manage.py — точка входа для административных команд Django;
  + calories/ — основное приложение:
    - models.py — содержит модель Entry;
    - serializers.py — описание сериализатора EntrySerializer;
    - views.py — представление EntryViewSet на основе DRF;
    - urls.py — маршруты, подключаемые к роутеру;
  + backend\_app/ — настройки проекта:
    - settings.py — конфигурация базы данных, установленных приложений, CORS и пр.;
    - urls.py — корневой URL-конфиг, подключает API-маршруты через include.

Frontend (React):

* frontend/ — директория клиентской части:
  + src/ — основной каталог исходного кода:
    - components/ — Form.jsx и EntriesTable.jsx;
    - api/ — файл api.js для обращения к серверу;
    - App.js — основной компонент, объединяющий интерфейс;
    - index.js — точка входа в приложение.

Маршруты

* Клиентский интерфейс доступен по пути / (React SPA);
* API эндпоинт: /api/entries/:
  + GET — получить список записей;
  + POST — добавить новую запись;
  + ?date=YYYY-MM-DD — фильтрация записей по дате.

Проект построен по модульной схеме, позволяющей легко масштабировать функционал: добавлять авторизацию, роли пользователей, дополнительные таблицы и представления. Использование стандартной структуры Django и React обеспечивает читаемость и расширяемость кода.

Веб-приложение построено по классической клиент-серверной архитектуре с разделением на frontend и backend части.

Backend:

* Язык: Python
* Фреймворк: Django + Django REST Framework
* Функции: реализация API, работа с базой данных (SQLite), валидация данных, маршрутизация

Frontend:

* Язык: JavaScript (ES6+)
* Библиотека: React
* Функции: форма ввода данных, отправка запросов к API, таблица с фильтрацией по дате

Обмен данными: осуществляется по HTTP (REST API), данные передаются в формате JSON.

Каждая часть проекта упакована в собственный Docker-контейнер, и управление осуществляется через docker-compose. В перспективе архитектура масштабируема, допускает добавление компонентов (например, авторизация, база PostgreSQL, аналитика).

## 2.3 Реализация backend

API: структура и поведение

Серверная часть предоставляет REST API через Django REST Framework. Основной эндпоинт — /api/entries/, который поддерживает следующие методы:

* GET /api/entries/ — возвращает список всех записей о приёмах пищи.
* GET /api/entries/?date=YYYY-MM-DD — возвращает записи за конкретную дату.
* POST /api/entries/ — создаёт новую запись, принимая name, calories, date в теле запроса.

Пример запроса (POST):

POST /api/entries/ HTTP/1.1

Content-Type: application/json

{

"name": "Овсянка",

"calories": 250,

"date": "2025-05-22"

}

Пример ответа:

{

"id": 1,

"name": "Овсянка",

"calories": 250,

"date": "2025-05-22"

}

Пример запроса (GET):

GET /api/entries/?date=2025-05-22 HTTP/1.1

Accept: application/json

Пример ответа:

[

{

"id": 1,

"name": "Овсянка",

"calories": 250,

"date": "2025-05-22"

},

{

"id": 2,

"name": "Яйца",

"calories": 150,

"date": "2025-05-22"

}

]

Каждый запрос проходит валидацию: calories должно быть положительным числом, name — непустой строкой. Ошибки возвращаются в формате JSON с пояснением причины. Все ответы используют статус-коды HTTP (200, 201, 400 и т. д.), что соответствует REST-подходу.

Серверная часть реализована на Django. Основные элементы:

* Модель Entry: содержит поля name (название продукта), calories (число), date (дата записи);
* EntrySerializer: сериализация/десериализация данных API;
* EntryViewSet: поддерживает методы POST, GET и фильтрацию по дате;
* Маршруты: /api/entries/

Пример модели:

class Entry(models.Model):

name = models.CharField(max\_length=255)

calories = models.PositiveIntegerField()

date = models.DateField()

## 2.4 Реализация frontend

Клиентская часть написана с использованием React. Основные компоненты:

1. Form.jsx — форма для ввода названия и калорийности продукта;
2. EntriesTable.jsx — таблица для отображения записей;
3. api.js — интерфейс взаимодействия с backend.

Данные из формы отправляются через fetch POST-запросом. Таблица обновляется автоматически при добавлении новой записи. Также реализована фильтрация по дате через input типа date.

Пример отправки запроса:

fetch('/api/entries/', {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify({ name, calories, date })

});

## 2.5 Контейнеризация и запуск

Для локального запуска используется docker-compose.

Пример docker-compose.yml:

version: '3.8'

services:

backend:

build: ./backend-app

volumes:

- ./backend-app:/app

ports:

- "8000:8000"

frontend:

build: ./frontend

volumes:

- ./frontend:/app

ports:

- "3000:3000"

Для настройки переменных окружения используется .env. Приложение поднимается командой:

docker compose up --build

**Dockerfile: backend**

FROM python:3.11-slim

WORKDIR /app

COPY requirements.txt ./

RUN pip install --no-cache-dir -r requirements.txt

COPY . .

CMD ["gunicorn", "backend\_app.wsgi:application", "--bind", "0.0.0.0:8000"]

**Dockerfile: frontend**

FROM node:18

WORKDIR /app

COPY package\*.json ./

RUN npm install

COPY . .

CMD ["npm", "start"]

**.env (пример)**

DEBUG=True

DJANGO\_SECRET\_KEY=your\_secret\_key\_here

ALLOWED\_HOSTS=localhost,127.0.0.1

CORS\_ALLOWED\_ORIGINS=http://localhost:3000

Приложение поднимается с помощью команды:

docker compose up --build

Контейнеры обмениваются данными в одной внутренней сети, настроенной через Compose. При необходимости можно масштабировать каждый сервис, задать зависимости и использовать внешние образы БД, кэш-серверов и других компонентов.

Для локального запуска используется docker-compose.

Пример docker-compose.yml:

version: '3.8'

services:

backend:

build: ./backend-app

volumes:

- ./backend-app:/app

ports:

- "8000:8000"

frontend:

build: ./frontend

volumes:

- ./frontend:/app

ports:

- "3000:3000"

Для настройки переменных окружения используется .env. Приложение поднимается командой:

docker compose up --build

## 2.6 Результаты и проверка

Обработка ошибок

На всех этапах взаимодействия пользовательского интерфейса и серверной части предусмотрены механизмы обработки ошибок. Пример:

Неверный формат данных при создании записи:

POST /api/entries/

Content-Type: application/json

{

"name": "Яблоко",

"calories": -100,

"date": "2025-05-23"

}

Ответ сервера:

{

"calories": ["Убедитесь, что это значение больше либо равно 0."]

}

Ошибка при пустом поле:

{

"name": ["Обязательное поле."]

}

Эти сообщения отображаются на клиенте, информируя пользователя о необходимости корректного ввода.

Пользовательские сценарии (Use Cases)

Для демонстрации работы MVP-приложения были выделены следующие типичные пользовательские сценарии:

Сценарий 1: Добавление новой записи о приёме пищи

* Пользователь заходит на веб-страницу.
* Вводит название продукта (например, "Гречка"), калорийность (например, 300 ккал) и дату.
* Нажимает кнопку "Добавить".
* На экране отображается новая запись в таблице.
* В фоновом режиме данные отправляются на сервер и сохраняются в базе.

Сценарий 2: Просмотр истории приёмов пищи

* Пользователь открывает веб-интерфейс.
* В таблице автоматически отображаются все ранее добавленные записи.
* Пользователь может прокручивать список, просматривая детали: название, калории и дату.

Сценарий 3: Фильтрация записей по дате

* Пользователь выбирает конкретную дату в поле фильтрации.
* Таблица обновляется, отображая только записи за выбранную дату.
* Это позволяет проанализировать калорийность за конкретный день.

Сценарий 4: Ошибка при вводе

* Пользователь пытается отправить пустую форму или указывает некорректное значение калорий.
* Сервер возвращает сообщение об ошибке (400 Bad Request), отображаемое в UI.
* Пользователь получает подсказку и исправляет ошибку.

После запуска контейнеров backend доступен по адресу http://localhost:8000, frontend — http://localhost:3000. Проведена ручная проверка корректности отправки и получения данных, фильтрации, обработки ошибок.

MVP успешно выполняет заявленные функции и готов к интеграции в пайплайн CI/CD, а также масштабированию и переходу на полноценную продакшн-инфраструктуру.

# ГЛАВА 3. РЕАЛИЗАЦИЯ CI/CD И ПОСТРОЕНИЕ ИНФРАСТРУКТУРЫ РАЗВЕРТЫВАНИЯ

## 3.1 Цели и задачи автоматизации

Следующим этапом после успешной реализации MVP стало внедрение процессов CI/CD (непрерывной интеграции и доставки). Основной задачей является организация автоматической сборки, тестирования и развертывания компонентов проекта. Это позволяет упростить поддержку проекта, ускорить релизы и повысить надёжность.

Задачи:

1. Автоматизация сборки backend и frontend;
2. Автодеплой на dev-сервер при коммите в main;
3. Ручной деплой на prod по событию;
4. Разделение пайплайнов по частям проекта (backend, frontend, nginx);
5. Безопасное хранение ключей через GitHub Secrets.

## 3.2 Выбор GitHub Actions

GitHub Actions был выбран как CI/CD-среда благодаря:

1. встроенной поддержке GitHub-репозиториев;
2. простой YAML-конфигурации;
3. поддержке подключения к VPS через SSH;
4. широкому сообществу и интеграциям.

Файлы пайплайнов размещаются в .github/workflows/ и запускаются по событиям, например push или workflow\_dispatch.

## 3.3 Пайплайны проекта

Были созданы 3 отдельных workflow:

1. deploy-backend.yml — при изменениях в backend-app/\*\*
2. deploy-frontend.yml — при изменениях в frontend/\*\*
3. deploy-proxy.yml — при изменениях в nginx/\*\* или nginx.conf

Каждый файл .yml находится в директории .github/workflows/ и содержит подробное описание шагов деплоя. Ниже приведён пример пайплайна с комментариями:

name: Deploy Backend # Название пайплайна

on:

push:

branches: [main] # Событие запуска: push в main-ветку

paths:

- 'backend-app/\*\*' # Только если изменены файлы backend

jobs:

deploy:

runs-on: ubuntu-latest # GitHub запускает runner на Ubuntu

steps:

- name: Checkout репозитория

uses: actions/checkout@v3 # Шаг 1: клонирует текущий репозиторий

- name: SSH-деплой на сервер

uses: appleboy/ssh-action@v0.1.7 # Шаг 2: подключение по SSH

with:

host: ${{ secrets.DEV\_HOST }}

username: root

key: ${{ secrets.DEV\_SSH\_KEY }}

script: | # Выполняемые команды на сервере

cd /root/diploma

git pull

docker compose build backend

docker compose up -d backend

Аналогичным образом построены пайплайны для frontend и proxy, с разницей в путях и именах сервисов в Docker Compose. Каждый пайплайн можно расширить — например, добавить шаги с уведомлением в Slack или запуском тестов.

Были созданы 3 отдельных workflow:

* deploy-backend.yml — при изменениях в backend-app/\*\*
* deploy-frontend.yml — при изменениях в frontend/\*\*
* deploy-proxy.yml — при изменениях в nginx/\*\* или nginx.conf.

Пример пайплайна для backend:

name: Deploy Backend

on:

push:

branches: [main]

paths: ['backend-app/\*\*']

jobs:

deploy:

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v3

- name: Deploy via SSH

uses: appleboy/ssh-action@v0.1.7

with:

host: ${{ secrets.DEV\_HOST }}

username: root

key: ${{ secrets.DEV\_SSH\_KEY }}

script: |

cd /root/diploma

git pull

docker compose build backend

docker compose up -d backend

## 3.4 Серверная инфраструктура

Проект использует два VPS:

1. Dev — для автоматического деплоя
2. Prod — для ручного деплоя.

Установка Docker и Docker Compose на сервер

1. Обновить пакеты:

sudo apt update && sudo apt upgrade -y

1. Установить зависимости:

sudo apt install apt-transport-https ca-certificates curl software-properties-common -y

1. Добавить ключ Docker:

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg

1. Добавить репозиторий Docker:

echo \

"deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] \

https://download.docker.com/linux/ubuntu \

$(lsb\_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

1. Установить Docker:

sudo apt update

sudo apt install docker-ce docker-ce-cli containerd.io -y

1. Установить Docker Compose:

sudo curl -L "https://github.com/docker/compose/releases/latest/download/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose

sudo chmod +x /usr/local/bin/docker-compose

1. Проверить установку:

docker --version

docker-compose --version

Установка Nginx:

sudo apt install nginx -y

1. Открыть порты:

sudo ufw allow 'Nginx Full'

sudo ufw allow 22

sudo ufw enable

Обе машины имеют:

* Установленные Docker и docker-compose;
* Открытые порты 80, 443, 8000, 3000;
* Nginx для проксирования;
* SSH-доступ по ключу.

На серверах выполнены:

* git clone репозитория;
* настройка firewall;
* запуск docker compose с .env.

Проект использует два VPS:

* Dev — для автоматического деплоя
* Prod — для ручного деплоя

Обе машины имеют:

* Установленные Docker и docker-compose;
* Открытые порты 80, 443, 8000, 3000;
* Nginx для проксирования;
* SSH-доступ по ключу.

На серверах выполнены:

* git clone репозитория;
* настройка firewall;
* запуск docker compose с .env.

## 3.5 Безопасность и переменные

Для защиты чувствительных данных использованы GitHub Secrets:

1. DEV\_HOST, PROD\_HOST
2. DEV\_SSH\_KEY, PROD\_SSH\_KEY
3. NGINX\_CONF

Это исключает хранение паролей в репозитории. Переменные окружения .env не коммитятся, а создаются вручную на сервере.

Разграничение доступа между dev и prod

Для обеспечения безопасности и изоляции окружений был реализован ручной запуск деплоя на продакшн через событие workflow\_dispatch. Это позволяет запускать пайплайн продакшна только вручную, исключая автоматический деплой при каждом изменении в репозитории.

Пример конфигурации workflow:

name: Deploy to Production

on:

workflow\_dispatch: # Ручной запуск через интерфейс GitHub

jobs:

deploy:

runs-on: ubuntu-latest

steps:

- name: Checkout code

uses: actions/checkout@v3

- name: Deploy to prod

uses: appleboy/ssh-action@v0.1.7

with:

host: ${{ secrets.PROD\_HOST }}

username: root

key: ${{ secrets.PROD\_SSH\_KEY }}

script: |

cd /root/diploma

git pull

docker compose build

docker compose up -d

Таким образом, прод-сервер защищён от несанкционированного обновления, и все изменения можно проверить сначала на dev-окружении. Это повышает надёжность релизов и даёт команде гибкость в управлении циклами поставки.

Для защиты чувствительных данных использованы GitHub Secrets:

* DEV\_HOST, PROD\_HOST
* DEV\_SSH\_KEY, PROD\_SSH\_KEY
* NGINX\_CONF

Это исключает хранение паролей в репозитории. Переменные окружения .env не коммитятся, а создаются вручную на сервере.

## 3.6 Результаты автоматизации

Логирование в рамках CI/CD и контейнеров

Для наблюдаемости процессов и отладки в проекте реализовано базовое логирование как на уровне пайплайнов GitHub Actions, так и внутри Docker-контейнеров.

GitHub Actions автоматически сохраняет лог каждого запуска workflow, включая результаты шагов, ошибки и предупреждения. Это позволяет:

* отслеживать успешные и неуспешные деплои;
* анализировать причины отказов (например, ошибки SSH, docker build);
* просматривать логи в реальном времени через веб-интерфейс GitHub.

Docker-контейнеры логируют поведение сервисов. При запуске backend и frontend можно получить доступ к логам с помощью команд:

docker logs <имя\_контейнера>

Для постоянного мониторинга можно перенаправить stdout/stderr в файл или использовать лог-драйверы Docker (json-file, syslog).

В перспективе возможно внедрение централизованного логирования с помощью ELK-стека (Elasticsearch + Logstash + Kibana) или более лёгких решений типа Loki + Grafana.

Логирование помогает не только в отладке, но и в аудите событий CI/CD, выявлении узких мест и доказательстве надёжности процессов автоматизации.

Проблемы и их решения при внедрении пайплайна

...

Проблемы и их решения при внедрении пайплайна

В процессе внедрения CI/CD пайплайна возникли несколько практических проблем, требующих дополнительных решений и адаптаций:

1. Проблема: ошибка при сборке backend-образа

* Описание: при выполнении docker compose build пайплайн не находил requirements.txt.
* Причина: ошибка в пути копирования в Dockerfile.
* Решение: структура проекта была пересмотрена, файл перенесён и путь в Dockerfile обновлён:

COPY backend-app/requirements.txt ./

2. Проблема: SSH-доступ не устанавливался из GitHub Actions

* Описание: action appleboy/ssh-action не подключался к серверу.
* Причина: некорректный формат SSH-ключа в GitHub Secrets.
* Решение: ключ был перекодирован в формат PEM, без лишних символов перевода строки. Секрет пересоздан с флагом New line disabled.

3. Проблема: контейнеры конфликтовали с предыдущими версиями

* Описание: docker compose up не перезапускал сервис корректно.
* Решение: был добавлен флаг --build и --remove-orphans, обеспечивающий пересборку и очистку:

docker compose up -d --build --remove-orphans

4. Проблема: frontend не отображал корректно API данные

* Описание: CORS-ошибка между портами 3000 (React) и 8000 (Django).
* Решение: в settings.py добавлены:

CORS\_ALLOWED\_ORIGINS = ["http://localhost:3000"]

и установлен пакет django-cors-headers.

5. Проблема: prod-пайплайн запускался случайно

* Решение: переведён на workflow\_dispatch и отключён от событий push, защищая от непреднамеренных релизов.

6. Проблема: скрытые переменные не применялись

* Причина: .env отсутствовал на сервере.
* Решение: .env создан вручную с нужными ключами (SECRET\_KEY, DEBUG и др.).

Таким образом, отладка пайплайна потребовала ручного анализа логов, корректировки Dockerfile и настройки переменных окружения. Эти решения укрепили надёжность деплоя и обеспечили бесперебойную работу пайплайна в автоматическом и ручном режимах.

CI/CD работает стабильно:

1. Backend и frontend деплоятся за 10–15 секунд;
2. Изменения появляются на dev без вмешательства человека;
3. Prod обновляется вручную через интерфейс GitHub;
4. Пайплайны независимы и не мешают друг другу.

Система масштабируема: можно добавить Slack-уведомления, мониторинг, тесты и использовать GitHub Actions для релизов с версионированием. CI/CD-процесс успешно внедрён и сопровождает проект на всех этапах.

CI/CD работает стабильно:

1. Backend и frontend деплоятся за 10–15 секунд;
2. Изменения появляются на dev без вмешательства человека;
3. Prod обновляется вручную через интерфейс GitHub;
4. Пайплайны независимы и не мешают друг другу.

Система масштабируема: можно добавить Slack-уведомления, мониторинг, тесты и использовать GitHub Actions для релизов с версионированием. CI/CD-процесс успешно внедрён и сопровождает проект на всех этапах.

# ЗАКЛЮЧЕНИЕ

В ходе выполнения дипломной работы была поставлена цель — разработать и внедрить CI/CD пайплайн для автоматизации развертывания веб-приложения по подсчёту калорий. Работа охватывала как проектирование и реализацию MVP, так и построение инфраструктуры доставки кода в двух средах: dev и production.

Была проведена теоретическая база: изучены и проанализированы основные концепции DevOps, непрерывной интеграции и доставки, инструменты контейнеризации и автоматизации. На основании анализа было принято обоснованное решение использовать Docker и GitHub Actions в качестве основы инфраструктуры.

В процессе разработки:

1. Реализовано веб-приложение с использованием Django (backend) и React (frontend);
2. Организован REST API для взаимодействия между клиентом и сервером;
3. Выполнена контейнеризация всех компонентов с помощью Docker;
4. Настроен автоматический деплой на dev-сервер при пуше в основную ветку;
5. Обеспечен ручной запуск пайплайна для прод-сервера через workflow\_dispatch;
6. Реализована изоляция окружений, защита переменных и логирование.

Проект подтвердил практическую значимость подхода CI/CD для ускорения поставки программного обеспечения, повышения надёжности и удобства поддержки. Использование Docker позволило упростить развёртывание, а GitHub Actions обеспечил полную автоматизацию процесса.

С технической точки зрения были решены реальные задачи, включая ошибки сборки, CORS-проблемы, SSH-доступ, настройку переменных окружения. Внедрение пайплайна сопровождалось логированием и ручным тестированием.

Использование экспериментального подхода с построением MVP и поэтапным внедрением CI/CD позволило не только подтвердить теоретические гипотезы, но и оперативно выявить и устранить технические проблемы. Такой формат наиболее эффективен при работе с современными архитектурами веб-приложений.

С практической точки зрения, проект также отвечает потребностям общества в вопросах самоконтроля за питанием и здоровьем. В условиях роста интереса к здоровому образу жизни и профилактике заболеваний, цифровые инструменты для подсчёта калорий становятся всё более востребованными. Разработанное приложение может использоваться как индивидуально, так и интегрироваться в системы диетологического сопровождения, спортивных программ и корпоративного велнеса.

Практическая ценность работы заключается в готовом шаблоне для небольших и средних команд, которые могут использовать данную архитектуру и подход для своих проектов. Благодаря модульной структуре и универсальным технологиям, решение легко адаптируется под различные типы веб-приложений — от систем бронирования и интернет-магазинов до образовательных платформ и административных панелей.

CI/CD пайплайн, реализованный в проекте, может быть переиспользован для других продуктов с минимальными изменениями. Поддержка независимых пайплайнов для backend, frontend и nginx позволяет масштабировать разработку на многокомпонентные системы. Выделение среды разработки (dev) и среды эксплуатации (prod) облегчает контроль версий и качество поставки.

Также, контейнеризация всех компонентов предоставляет возможность лёгкой миграции между хостингами, поддержки нескольких сред, интеграции с облачными CI/CD платформами и внедрения кластеризации. Это делает проект хорошей основой не только для учебных, но и для коммерческих решений.

для небольших и средних команд, которые могут использовать данную архитектуру и подход для своих проектов. Результаты могут быть масштабированы, а система легко расширяется дополнительными сервисами, авторизацией, мониторингом, версионированием и аналитикой.

В дальнейшем возможно интегрировать централизованное логирование, систему мониторинга, тестирование и использовать Kubernetes для продвинутой оркестрации контейнеров. Работа стала не только инженерным решением, но и подтверждением важности методологии DevOps в современном жизненном цикле программных продуктов.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ