

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ
ФЕДЕРАЦИИ ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ
по выполнению лабораторных работ
по дисциплине «Проектирование и оптимизация конвейеров CI/CD» для студентов
направления подготовки 09.04.04 Программная инженерия
Направленность (профиль)
«Разработка, мониторинг и управление жизненным циклом инженерных систем»

Ставрополь 2026

ВВЕДЕНИЕ

Лабораторный практикум по дисциплине «Проектирование и оптимизация конвейеров CI/CD» является неотъемлемой частью подготовки магистров по направлению 09.04.04 «Программная инженерия» (профиль «Разработка, мониторинг и управление жизненным циклом инженерных систем»).

Основная цель лабораторного практикума - формирование у обучающихся практических навыков проектирования, реализации, оптимизации и сопровождения сквозных конвейеров непрерывной интеграции, поставки и развертывания (CI/CD) для сложных программных и программно-аппаратных систем, включая системы автономного управления (БПЛА, робототехнические комплексы).

Задачи лабораторного практикума:

1. Освоение базовых принципов CI/CD - настройка первого пайплайна, работа с артефактами, структурирование конфигураций.
2. Автоматизация тестирования - интеграция модульных и интеграционных тестов, анализ покрытия кода, генерация отчетов.
3. Обеспечение безопасности - внедрение статического анализа безопасности (SAST) и анализа зависимостей (SCA) с настройкой Quality Gate.
4. Контейнеризация и управление артефактами - сборка Docker-образов, публикация в registry, версионирование и подпись артефактов.
5. Стратегии развертывания - реализация zero-downtime развертывания (rolling update, blue-green, canary), использование feature flags и GitOps.
6. Специфика встроенных систем - кросс-компиляция прошивок, SITL-тестирование в симуляторах, безопасные OTA-обновления.
7. Оптимизация и мониторинг - ускорение пайплайнов (кэширование, параллелизация, шардирование), сбор метрик эффективности.

Структура лабораторного практикума

Лабораторный практикум состоит из 18 лабораторных работ, которые логически разделены на семь тематических модулей:

Модуль	№ ЛР	Тема
1. Основы CI/CD	1-3	Настройка базового пайплайна, Pipeline as Code, оптимизация времени выполнения
2. Тестирование и качество	4-5	Модульные/интеграционные тесты, покрытие кода
3. Безопасность (DevSecOps)	6-7	SAST, SCA, Quality Gate
4. Артефакты и контейнеризация	8-9	Docker-образы, репозитории артефактов
5. Развертывание и стратегии	10-14	Тестовый стенд, rolling update, blue-green, canary, GitOps
6. Встроенные системы	15-17	Кросс-компиляция, SITL, OTA-обновления
7. Итоговый проект	18	Сквозной пайплайн для автономной системы

Требования к подготовке и выполнению:

1. Предварительная подготовка

Перед выполнением лабораторных работ студент должен:

- Изучить соответствующие разделы лекционного курса и рекомендованной литературы.
- Зарегистрироваться на выбранной CI/CD-платформе (GitLab / GitHub).
- Установить необходимое программное обеспечение (Docker, Minikube, kubectl, Git).
- Ознакомиться с синтаксисом YAML и основами работы с системами контроля версий Git.

2. Порядок выполнения

1. Ознакомиться с теоретическим обоснованием и примерами выполнения.
2. Выбрать вариант задания в соответствии с номером, выданным преподавателем.
3. Разработать конфигурацию CI/CD-пайплайна в соответствии с заданием.
4. Выполнить проверку работоспособности пайплайна в CI-среде.
5. Проанализировать полученные результаты (логи, отчёты, артефакты).
6. Оформить отчёт по установленной форме.
7. Защитить лабораторную работу перед преподавателем (ответы на контрольные вопросы, демонстрация работы пайплайна).

3. Отчётность

Каждая лабораторная работа оформляется в виде электронного отчёта, содержащего:

- Титульный лист с указанием ФИО студента, группы, варианта и номера работы.
- Цель и задачи работы.
- Теоретическое обоснование (кратко).
- Листинги конфигурационных файлов с комментариями.
- Результаты выполнения (скриншоты этапов пайплайна, отчетов, артефактов).
- Анализ полученных результатов.
- Ответы на контрольные вопросы.
- Вывод о проделанной работе.

Лабораторная работа №1. Настройка базового CI/CD-пайплайна

Цель работы: Сформировать практические навыки создания и настройки базового CI/CD-пайплайна для автоматической сборки программного проекта и запуска модульных тестов с использованием современных CI-платформ (GitLab CI / GitHub Actions).

1. ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ

1.1. Понятие CI/CD и его роль в современной разработке

Continuous Integration (CI) - непрерывная интеграция - это практика разработки программного обеспечения, при которой разработчики регулярно (несколько раз в день) объединяют свои изменения в общий репозиторий, после чего автоматически выполняются сборка и тестирование проекта.

Основные цели CI:

- Раннее обнаружение ошибок - проблемы выявляются на ранних стадиях, когда их исправление требует минимальных затрат.
- Сокращение времени интеграции - вместо «ада смерти интеграции» перед релизом, интеграция происходит постоянно и безболезненно.
- Повышение уверенности в качестве кода - зелёный статус пайплайна означает, что код проходит все автоматические проверки.

Continuous Delivery (CD) - непрерывная доставка - это расширение CI, при котором код, прошедший автоматические тесты, автоматически разворачивается на тестовых средах (staging) и готов к выпуску в production. Ключевое отличие от Continuous Deployment заключается в том, что релиз в production может требовать ручного утверждения.

1.2. Архитектура CI/CD-пайплайна

Базовый CI/CD-пайплайн состоит из следующих логических этапов (рисунок 1):

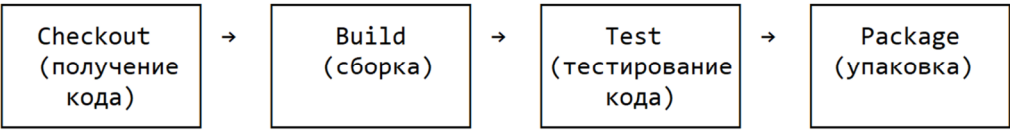


Рисунок 1

Ключевые компоненты:

Компонент	Описание	Примеры
Система контроля версий	Хранилище кода и триггер для запуска пайплайнов	GitLab, GitHub, Bitbucket
CI-сервер / Платформа	Исполняет пайплайны, управляет раннерами	GitLab CI, GitHub Actions, Jenkins
Раннер / Агент	Вычислительный узел, выполняющий задания	GitLab Runner, GitHub Runner, self-hosted
Артефакты	Результаты сборки, передаваемые между этапами	Бинарные файлы, отчеты, логи
Репозиторий артефактов	Хранилище результатов сборки	GitLab Package Registry, Nexus, Artifactory

1.3. Конфигурация пайплайна как код (Pipeline as Code)

Pipeline as Code - это подход, при котором конфигурация CI/CD-пайплайна хранится в том же репозитории, что и исходный код приложения. Это обеспечивает:

- Версионирование - изменения пайплайна проходят code review.
- Воспроизводимость - пайплайн всегда соответствует версии кода.

- Прозрачность - любой разработчик может видеть и предлагать изменения.

Синтаксис для популярных платформ:

Платформа	Файл конфигурации	Базовый синтаксис
GitLab CI	.gitlab-ci.yml	stages, job, script, artifacts
GitHub Actions	.github/workflows/.yml	name, on, jobs, steps, uses, run

1.4. Основные элементы конфигурации GitLab CI

```
# .gitlab-ci.yml

# Определение этапов (порядок выполнения)
stages:
  - build
  - test

# Переменные окружения
variables:
  APP_VERSION: "1.0.0"

# Job для сборки
build-job:
  stage: build
  script:
    - echo "Сборка проекта..."
    - make build
  artifacts:
    paths:
      - dist/

# Job для тестирования
test-job:
  stage: test
  script:
    - echo "Запуск тестов..."
    - make test
  dependencies:
    - build-job
```

1.5. Основные элементы конфигурации GitHub Actions

```
# .github/workflows/ci.yml

name: CI Pipeline

on:
  push:
    branches: [ main, develop ]
  pull_request:
    branches: [ main ]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: '18'

      - name: Install dependencies
        run: npm ci

      - name: Build
        run: npm run build

      - name: Upload artifacts
        uses: actions/upload-artifact@v4
        with:
          name: build-output
          path: dist/

  test:
    runs-on: ubuntu-latest
    needs: build
    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Run tests
        run: npm test
```

1.6. Анализ логов выполнения

Логи выполнения CI/CD-пайплайна содержат:

- Время начала и окончания каждого задания.
- Вывод команд `script / run` (`stdout`, `stderr`).
- Код возврата (`exit code`) - 0 означает успех, ненулевой - ошибку.
- Информацию об артефактах и кэше.

Типичные ошибки при первом запуске:

- Синтаксические ошибки в YAML-файле.
- Отсутствие необходимых зависимостей в среде раннера.
- Неправильные пути к файлам.
- Несовместимость версий инструментов.

2. ПРИМЕРЫ ПРАКТИЧЕСКОГО ВЫПОЛНЕНИЯ

2.1. Пример для Python (GitLab CI)

Структура проекта:

```
my-python-app/  
├── .gitlab-ci.yml  
├── requirements.txt  
├── src/  
│   └── main.py  
└── tests/  
    └── test_main.py
```

Содержимое .gitlab-ci.yml:

```
stages:
  - build
  - test

variables:
  PIP_CACHE_DIR: "$CI_PROJECT_DIR/.cache/pip"

cache:
  paths:
    - .cache/pip
    - venv/

build-job:
  stage: build
  script:
    - python -m venv venv
    - source venv/bin/activate
    - pip install -r requirements.txt
  artifacts:
    paths:
      - venv/
    expire_in: 1 hour

test-job:
  stage: test
  script:
    - source venv/bin/activate
    - pip install pytest
    - pytest tests/ --junitxml=report.xml
  artifacts:
    paths:
      - report.xml
  reports:
    junit: report.xml
```

2.2. Пример для Node.js (GitHub Actions)

Структура проекта:

```
my-node-app/  
├── .github/workflows/ci.yml  
├── package.json  
├── src/  
│   └── index.js  
└── tests/  
    └── index.test.js
```

Содержимое .github/workflows/ci.yml:

```
name: Node.js CI  
  
on:  
  push:  
    branches: [ main ]  
  pull_request:  
    branches: [ main ]  
  
jobs:  
  build-and-test:  
    runs-on: ubuntu-latest  
  
    strategy:  
      matrix:  
        node-version: [18.x, 20.x]  
  
    steps:  
      - uses: actions/checkout@v4  
  
      - name: Use Node.js ${{ matrix.node-version }}  
        uses: actions/setup-node@v4  
        with:  
          node-version: ${{ matrix.node-version }}  
          cache: 'npm'  
  
      - run: npm ci  
      - run: npm run build --if-present
```

```
- run: npm test
```

2.3. Пример для Go (GitLab CI)

Содержимое .gitlab-ci.yml:

```
image: golang:1.21

stages:
  - build
  - test

variables:
  GO111MODULE: "on"
  CGO_ENABLED: "0"

cache:
  paths:
    - .cache/go/

before_script:
  - mkdir -p .cache/go
  - go mod download

build:
  stage: build
  script:
    - go build -o myapp ./cmd/app
  artifacts:
    paths:
      - myapp

test:
  stage: test
  script:
    - go test -v -coverprofile=coverage.out ./...
  artifacts:
    paths:
      - coverage.out
```

Методические указания для выполнения практической части работы

Задачи работы:

1. Регистрация и настройка рабочей среды:

- Зарегистрироваться на платформе GitLab / GitHub (при необходимости).

- Создать репозиторий для размещения исходного кода проекта.
- Настроить доступ к CI/CD-функционалу платформы.

2. Создание конфигурационного файла пайплайна:

- Разработать файл .gitlab-ci.yml или .github/workflows/ci.yml.
- Определить структуру пайплайна (этапы, задания, скрипты).

3. Реализация этапа сборки (build):

- Настроить автоматическую установку зависимостей проекта.
- Реализовать сборку (компиляцию) проекта в CI-среде.
- Настроить сохранение артефактов сборки для последующих этапов.

4. Реализация этапа тестирования (test):

- Настроить автоматический запуск модульных тестов.
- Обеспечить генерацию и сохранение отчётов о тестировании.

5. Анализ результатов выполнения:

- Интерпретировать логи выполнения пайплайна.
- Выявлять и устранять типовые ошибки при первом запуске.
- Оценить корректность работы пайплайна по зелёному статусу.

6. Оформление отчёта:

- Зафиксировать листинги конфигурационных файлов.
- Представить скриншоты успешного выполнения пайплайна.
- Сформулировать выводы о проделанной работе.

3. ВАРИАНТЫ ЗАДАНИЙ

3.1. Базовый уровень

№	Язык/Платформа	Задание
1	Python	Создать пайплайн для скрипта, который вычисляет факториал. Добавить этап сборки (виртуальное окружение) и этап запуска

		модульных тестов (unittest).
2	Python	Создать пайплайн для скрипта, который проверяет, является ли число простым. Добавить этап сборки зависимостей и этап тестирования с pytest.
3	Python	Создать пайплайн для скрипта, который сортирует массив. Добавить этап сборки и этап запуска тестов с проверкой покрытия (coverage.py)
4	JavaScript	Создать пайплайн в GitHub Actions для Node.js-приложения. Добавить этап установки зависимостей (npm ci) и этап запуска тестов (npm test).
5	JavaScript	Создать пайплайн для простой функции сложения. Добавить этап линтинга (ESLint) и этап тестирования (Jest).
6	JavaScript	Создать пайплайн для утилиты работы со строками. Добавить этап сборки (webpack) и этап тестирования
7	Go	Создать пайплайн для консольного калькулятора. Добавить этап сборки (go build) и этап тестирования (go test).
8	Go	Создать пайплайн для HTTP-сервера "Hello World". Добавить этап сборки и этап интеграционного тестирования (curl).
9	Java (Maven)	Создать пайплайн для простого класса Calculator. Добавить этап сборки (mvn compile) и этап тестирования (mvn test).
10	Java (Gradle)	Создать пайплайн для утилиты StringUtils.

		Добавить этап сборки (gradle build) и этап тестирования.
11	C++ (CMake)	Создать пайплайн для программы, выводящей "Hello, CI!". Добавить этап сборки (cmake, make) и этап запуска
12	C (.NET)	Создать пайплайн для консольного приложения. Добавить этап восстановления зависимостей (dotnet restore) и этап сборки/тестирования
13	Ruby	Создать пайплайн для скрипта, вычисляющего среднее арифметическое. Добавить этап установки gem и этап тестов (RSpec).
14	PHP	Создать пайплайн для скрипта, валидирующего email. Добавить этап установки Composer и этап тестирования (PHPUnit).
15	Rust	Создать пайплайн для функции вычисления НОД. Добавить этап сборки (cargo build) и этап тестирования (cargo test).

3.2. Средний уровень

№	Язык/Платформа	Задание
1	Python	Создать пайплайн с двумя этапами: build (установка зависимостей, линтинг flake8) и test (pytest с генерацией JUnit-отчета и coverage). Настроить публикацию отчета как артефакта
2	Python	Создать пайплайн для FastAPI-приложения. Добавить этапы: сборка, линтинг (black, isort), тестирование (pytest), проверка покрытия (>80%).
3	Python	Создать пайплайн для CLI-утилиты. Добавить

		этапы: сборка wheel-пакета, установка пакета в виртуальное окружение, тестирование и проверка вывода через subprocess.
4	JavaScript	Создать пайплайн для React-приложения. Добавить этапы: установка, линтинг (ESLint), форматирование (Prettier), тестирование (Jest), сборка (npm run build). Артефакт - папка build
5	JavaScript	Создать пайплайн для Express API. Добавить этапы: установка, тестирование (Supertest), проверка покрытия (Jest --coverage), публикация отчета.
6	JavaScript	Создать пайплайн для TypeScript-проекта. Добавить этапы: установка, компиляция (tsc), линтинг (ESLint + TypeScript), тестирование (Jest).
7	Go	Создать пайплайн с параллельными заданиями: test (go test) и lint (golangci-lint). Добавить этап сборки бинарного файла и публикации артефакта
8	Go	Создать пайплайн для gRPC-сервера. Добавить этапы: генерация кода из proto, сборка, тестирование, проверка форматирования (gofmt)
9	Java (Maven)	Создать пайплайн с этапами: clean compile, test (с JUnit), package (jar), проверка покрытия (JaCoCo). Артефакт - jar-файл
10	Java (Spring Boot)	Создать пайплайн для REST-контроллера. Добавить этапы: сборка, тестирование (MockMvc), интеграционное тестирование с Testcontainers

11	C++ (CMake)	Создать пайплайн с этапами: конфигурация CMake, сборка (make), запуск unit-тестов (Google Test), сборка документации (Doxygen).
12	C (.NET)	Создать пайплайн с этапами: restore, build, test (coverlet), publish. Настроить запуск тестов на нескольких версиях .NET (6.0, 8.0).
13	Docker + любой язык	Создать пайплайн, который собирает Docker-образ приложения, запускает тесты внутри контейнера, публикует образ в registry
14	Terraform	Создать пайплайн для проверки Terraform-конфигураций: terraform fmt, terraform validate, terraform plan (без применения).
15	Ansible	Создать пайплайн для проверки Ansible-плейбуков: ansible-lint, ansible-playbook --syntax-check, тестирование с molecule

3.3. Продвинутый уровень

№	Язык/Платформа	Задание
1	Python + Docker	Создать пайплайн с multi-stage сборкой Docker-образа. Добавить этапы: линтинг, тестирование, сборка образа, сканирование образа на уязвимости (Trivy), публикация в registry
2	Python + CI матрица	Создать пайплайн с матрицей: Python 3.9, 3.10, 3.11, 3.12; OS: ubuntu-latest, windows-latest, macos-latest. Тесты должны выполняться на всех комбинациях
3	JavaScript + GitHub Actions	Создать пайплайн с manual trigger (workflow_dispatch), позволяющим выбрать

		окружение для развертывания (dev/staging/prod). Добавить conditional steps
4	Go + GitLab CI	Создать пайплайн с использованием динамических child pipelines. Основной пайплайн генерирует дочерний для каждого микросервиса в монорепозитории
5	Java + Jenkins	Создать пайплайн в Jenkins с использованием Jenkinsfile (Declarative Pipeline). Добавить этапы: checkout, build, test, archiveArtifacts, post-действия (email уведомления).
6	Terraform + GitLab CI	Создать пайплайн для инфраструктуры AWS (S3 bucket + EC2). Добавить этапы: init, validate, plan (с сохранением плана как артефакта), apply (только для ветки main).
7	Kubernetes + CI/CD	Создать пайплайн, который собирает образ, публикует в registry, обновляет манифест в Git-репозитории (K8s deployment), запускает kubectl apply.
8	Специфика БПЛА (PX4)	Создать пайплайн для кросс-компиляции прошивки PX4 под Pixhawk. Добавить этапы: клонирование PX4, сборка с помощью make, архивация .px4 файла
9	ROS 2 + CI	Создать пайплайн для ROS 2 пакета. Добавить этапы: установка ROS 2 в контейнере, сборка (colcon build), линтинг (ament_), запуск тестов (colcon test).
10	Gazebo + SITL	Создать пайплайн для симуляции полета дрона в Gazebo. Добавить этапы: установка Gazebo, запуск симуляции, выполнение Python-скрипта

		с автономным полетом, запись видео
11	Hardware-in-the-Loop	Создать пайплайн, взаимодействующий с HiTL-стендом (через API). Добавить этапы: сборка прошивки, загрузка на устройство, запуск тестового сценария, сбор телеметрии
12	ОТА-обновления	Создать пайплайн, который собирает прошивку, подписывает её (GPG/Cosign), публикует на сервер обновлений и генерирует манифест для ОТА-клиента
13	Безопасность	Создать пайплайн с этапами SAST (Semgrep), SCA (Dependency-Check), секретов (TruffleHog), контейнера (Trivy). Пайплайн должен падать при критических уязвимостях
14	Performance CI	Создать пайплайн для нагрузочного тестирования (k6). Добавить этапы: сборка, развертывание на тестовый стенд, запуск k6 теста, сохранение метрик, сравнение с baseline
15	Мониторинг пайплайна	Создать пайплайн, который после выполнения отправляет метрики (время выполнения, статус) в Prometheus через Pushgateway. Добавить дашборд в Grafana

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое Continuous Integration и в чем ее отличие от Continuous Delivery?
2. Какие преимущества дает подход «Pipeline as Code»?
3. Перечислите основные этапы базового CI-пайплайна.
4. Какие артефакты могут передаваться между этапами пайплайна?
5. Для чего используется кэширование в CI/CD? Приведите пример.

6. Чем отличается script от artifacts в GitLab CI?
7. Какие триггеры можно использовать для запуска пайплайна в GitHub Actions?
8. Что означает успешный код возврата (exit code) команды в CI-среде?
9. Какие инструменты можно использовать для модульного тестирования в Python?
10. Как передать переменные окружения в пайплайн безопасным способом?
11. Для чего используется директива needs в GitHub Actions?
12. Какие ошибки возникают наиболее часто при первом запуске пайплайна?
13. Как настроить запуск пайплайна только для определенных веток?
14. Что такое матричная стратегия (matrix strategy) в CI/CD?
15. Как добавить ручное подтверждение (manual approval) перед выполнением этапа?

5. ТРЕБОВАНИЯ К ОТЧЕТУ

Отчет должен содержать следующие разделы:

1. Титульный лист с указанием ФИО студента, группы, варианта.
2. Цель работы - формулируется на основе задания.
3. Теоретическая часть:
 - Краткое описание выбранной CI/CD-платформы (GitLab CI или GitHub Actions).
 - Объяснение ключевых элементов конфигурации, использованных в работе.
4. Практическая часть:
 - Листинг файла конфигурации пайплайна с комментариями.

- Листинг тестируемого кода (если он разрабатывался).

- Листинг тестов.

5. Результаты выполнения:

- Скриншоты успешного выполнения пайплайна в веб-интерфейсе.

- Скриншоты логов каждого этапа.

- Скриншоты артефактов (если создавались).

6. Анализ ошибок (если возникали):

- Описание ошибки.

- Причина возникновения.

- Способ устранения.

7. Ответы на контрольные вопросы (3-5 вопросов по выбору преподавателя).

8. Вывод - что удалось освоить в ходе работы, какие навыки приобретены.

6. КРИТЕРИИ ОЦЕНИВАНИЯ

Критерий	Базовый уровень	Средний уровень	Продвинутый уровень
Пайплайн запускается без ошибок	2 балла	2 балла	2 балла
Все этапы выполнены успешно	2 балла	2 балла	2 балла
Корректно настроена передача артефактов	1 балл	2 балла	2 балла
Корректно настроено кэширование	1 балл		
Настроены параллельные/матричные задания	0 баллов	1 балл	2 балла

Настроены условные этапы (only/if)	0 баллов	1 балл	2 балла
Интегрированы дополнительные инструменты (linter, coverage)	0 баллов	2 балла	2 балла
Качество отчета и ответы на вопросы	2 балла	2 балла	2 балла

Шкала перевода баллов в оценку:

Оценка	Базовый уровень	Средний уровень	Продвинутый уровень
Отлично (5)	8-10 баллов	10-12 баллов	12-14 баллов
Хорошо (4)	6-7 баллов	8-9 баллов	10-11 баллов
Удовлетворительно (3)	4-5 баллов	6-7 баллов	8-9 баллов
Неудовлетворительно (2)	менее 4 баллов	менее 6 баллов	менее 8 баллов

Дополнительные критерии (штрафы):

- Отсутствие скриншотов выполнения пайплайна - минус 1 балл.
- Отсутствие листингов кода - минус 1 балл.
- Неправильное форматирование YAML (синтаксические ошибки) - минус 2 балла.
- Пайплайн не запускается - работа не засчитывается до исправления.

Лабораторная работа № 2. Пайплайн как код и структурирование пайплайна

Цель работы: сформировать практические навыки декомпозиции CI/CD-пайплайна на логические этапы и организации повторного использования кода конфигурации с использованием механизмов шаблонизации (includes, extends, reusable workflows).

1. Теоретическое обоснование

1.1. Многостадийный пайплайн: концепция и преимущества

Многостадийный пайплайн (multi-stage pipeline) - это конвейер CI/CD, разделенный на логические этапы (stages), каждый из которых выполняет определенную функцию в процессе сборки, тестирования и доставки программного обеспечения.

Базовая структура трехстадийного пайплайна (рисунок 1):

Многостадийный пайплайн			
Stage 1 BUILD (сборка)	Stage 2 TEST (тестирование)	Stage 3 PACKAGE (упаковка)	Stage 4 (опционально) DEPLOY (развертывание)
• компиляция • установка зависимостей • линтинг	• unit-тесты • интеграц. тесты • проверка покрытия	• создание артефакта • публикация в registry	• развертывание на стенд • обновление конфигурации

Рисунок 1

Ключевые преимущества многостадийного пайплайна:

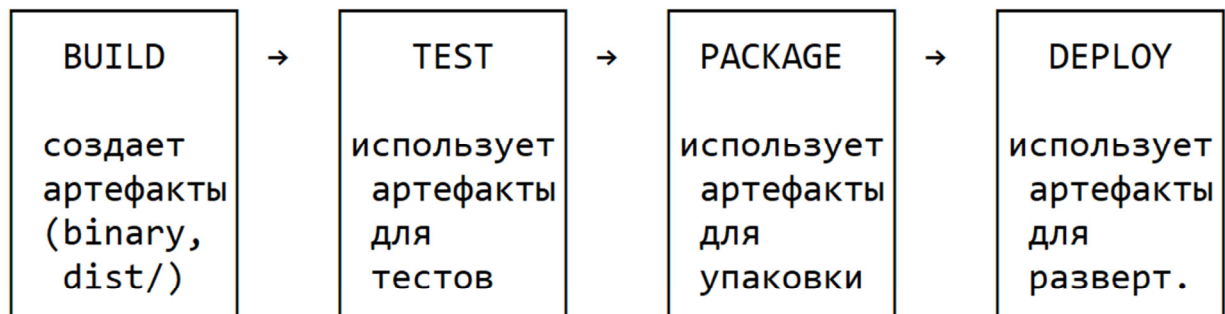
Преимущество	Описание
Четкое разделение ответственности	Каждый этап выполняет строго определенную функцию
Раннее обнаружение ошибок	Ошибка на раннем этапе останавливает

	выполнение последующих
Параллелизация	Независимые задания внутри этапа могут выполняться параллельно
Повторное использование	Результаты этапа (артефакты) передаются дальше
Прозрачность	Легко понять, на каком этапе произошел сбой

1.2. Передача артефактов между этапами

Артефакты - это результаты работы задания, которые сохраняются и могут быть использованы последующими этапами.

Схема передачи артефактов (рисунок 2):



В GitLab CI артефакты настраиваются через ключ artifacts:

```
//yaml
build:
  stage: build
  script:
    - make build
  artifacts:
    paths:
      - dist/
    expire_in: 1 hour
```

В GitHub Actions артефакты настраиваются через actions/upload-artifact и actions/download-artifact:

```
- name: Upload build artifacts
  uses: actions/upload-artifact@v4
  with:
    name: build-output
    path: dist/
```

1.3. Шаблонизация и повторное использование кода (DRY в CI/CD)

Принцип DRY (Don't Repeat Yourself) в контексте CI/CD означает вынесение повторяющихся фрагментов конфигурации в переиспользуемые компоненты.

Подходы к шаблонизации:

Подход	GitLab CI	GitHub Actions
Базовые шаблоны	include	workflow_call
Переменные	variables	env
Повторяющиеся блоки	extends, !reference, anchor	composite actions
Библиотеки шагов	include:file	actions/checkout@v4

Пример использования extends в GitLab CI:

```
# Шаблон для Python-заданий
.python-template:
  image: python:3.11
  before_script:
    - python -m venv venv
    - source venv/bin/activate
    - pip install -r requirements.txt
```

```
# Использование шаблона
test-job:
  extends: .python-template
  stage: test
  script:
    - pytest tests/
```

Пример использования include для выноса общих частей в отдельные файлы:

```
# .gitlab-ci.yml
include:
  - local: '/ci/templates/build.yml'
  - local: '/ci/templates/test.yml'
  - local: '/ci/templates/deploy.yml'

stages:
  - build
  - test
  - deploy
```

Структура проекта с шаблонами (рисунок 3):

```
project/
├── .gitlab-ci.yml          # основной файл (только stages и include)
├── ci/
│   ├── templates/
│   │   ├── build.yml      # шаблоны для сборки
│   │   ├── test.yml       # шаблоны для тестирования
│   │   └── deploy.yml     # шаблоны для развертывания
│   └── variables.yml      # общие переменные
```

Рисунок 3

1.4. Управление последовательностью выполнения

Зависимости между этапами (stages):

- Этапы выполняются последовательно в порядке, определенном в stages.
- Если этап состоит из нескольких заданий (jobs), они выполняются параллельно (по умолчанию).

- Переход к следующему этапу происходит только после успешного завершения всех заданий текущего этапа.

Зависимости между заданиями (jobs):

Механизм	GitLab CI	GitHub Actions
Ожидание задания	needs	needs
Условный запуск	rules, only, except	if
Ручной запуск	when: manual	workflow_dispatch

Пример с needs (задание запускается после указанных):

```
# GitLab CI
deploy:
  stage: deploy
  needs: ["build", "test"] # запустится после build и test
  script:
    - deploy.sh
```

```
# GitHub Actions
deploy:
  needs: [build, test]
  runs-on: ubuntu-latest
  steps:
    - run: ./deploy.sh
```

1.5. Управление окружениями (environments)

Окружения позволяют отслеживать, где развернуто приложение, и управлять развертыванием в разные среды.

Окружение	Назначение	Типичные ограничения
dev	Разработка и отладка	Автоматическое развертывание из develop

test/staging	Тестирование перед релизом	Автоматическое или ручное
production	Боевая среда	Только ручной запуск, одобрение

Пример с окружениями в GitLab CI:

```

deploy_to_dev:
  stage: deploy
  environment:
    name: dev
  script:
    - deploy dev
  only:
    - develop

deploy_to_prod:
  stage: deploy
  environment:
    name: production
  script:
    - deploy prod
  only:
    - main
  when: manual # требует ручного подтверждения

```

2. Примеры практического выполнения

2.1. Пример для Python (GitLab CI) - базовый трехстадийный пайплайн

Структура проекта (рисунок 4):


```
my-python-app/
├── .gitlab-ci.yml
├── requirements.txt
├── src/
│   └── main.py
└── tests/
    └── test_main.py
```

Рисунок 4

Содержимое .gitlab-ci.yml:

```
stages:
  - build
  - test
  - package

variables:
  PIP_CACHE_DIR: "$CI_PROJECT_DIR/.cache/pip"

cache:
  paths:
    - .cache/pip
    - venv/

# Шаблон для Python-заданий
.python-template:
  image: python:3.11-slim
  before_script:
    - python -m venv venv
    - source venv/bin/activate
    - pip install --upgrade pip
    - pip install -r requirements.txt

build:
  stage: build
  extends: .python-template
  script:
    - echo "Сборка проекта..."
    - python -m py_compile src/main.py
```

```

artifacts:
  paths:
    - venv/
  expire_in: 1 hour

test:
  stage: test
  extends: .python-template
  script:
    - pip install pytest pytest-cov
    - pytest tests/ --cov=src --junitxml=report.xml
  artifacts:
    paths:
      - report.xml
    reports:
      junit: report.xml
  dependencies:
    - build

package:
  stage: package
  extends: .python-template
  script:
    - pip install build
    - python -m build --wheel
    - ls -la dist/
  artifacts:
    paths:
      - dist/
    expire_in: 1 week
  dependencies:
    - test

```

2.2. Пример с выносом шаблонов в отдельные файлы (GitLab CI)

Структура проекта (рисунок 5):

```

my-project/
├── .gitlab-ci.yml
├── .gitlab/
│   ├── templates/
│   │   ├── base.yml
│   │   ├── build.yml
│   │   ├── test.yml
│   │   └── package.yml
│   └── variables.yml

```

Рисунок 5

Файл .gitlab/templates/base.yml:

```
.python-base:
  image: python:3.11-slim
  cache:
    paths:
      - .cache/pip
      - venv/
  before_script:
    - python -m venv venv
    - source venv/bin/activate
    - pip install --upgrade pip
    - pip install -r requirements.txt
```

Файл .gitlab/templates/build.yml:

```
.build-template:
  stage: build
  extends: .python-base
  script:
    - echo "Сборка..."
    - python -m py_compile src/*.py
  artifacts:
    paths:
      - venv/
  expire_in: 1 hour
```

Файл .gitlab/templates/test.yml:

```
.test-template:
  stage: test
  extends: .python-base
  script:
    - pip install pytest pytest-cov
    - pytest tests/ --cov=src --junitxml=report.xml
  artifacts:
    paths:
      - report.xml
  reports:
    junit: report.xml
```

Файл .gitlab-ci.yml:

```
include:
  - local: '.gitlab/variables.yml'
```

```

- local: '.gitlab/templates/base.yml'
- local: '.gitlab/templates/build.yml'
- local: '.gitlab/templates/test.yml'
- local: '.gitlab/templates/package.yml'

stages:
  - build
  - test
  - package

build:
  extends: .build-template

test:
  extends: .test-template
  dependencies:
    - build

package:
  extends: .package-template
  dependencies:
    - test

```

2.3. Пример для Node.js (GitHub Actions) с reuse-actions

Структура проекта:

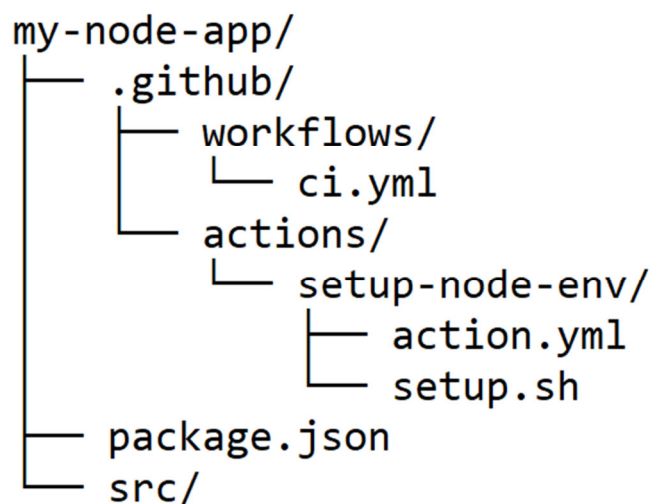


Рисунок 6

Файл .github/actions/setup-node-env/action.yml:

```

name: 'Setup Node.js Environment'
description: 'Setup Node.js with caching and install dependencies'

inputs:
  node-version:
    description: 'Node.js version'
    required: false
    default: '18'

runs:
  using: 'composite'
  steps:
    - name: Setup Node.js
      uses: actions/setup-node@v4
      with:
        node-version: ${ inputs.node-version }
        cache: 'npm'

    - name: Install dependencies
      shell: bash
      run: npm ci

```

Файл `.github/workflows/ci.yml`:

```

name: CI Pipeline

on:
  push:
    branches: [ main, develop ]
  pull_request:
    branches: [ main ]

env:
  NODE_VERSION: '18'

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Setup environment
        uses: ../github/actions/setup-node-env
        with:
          node-version: ${ env.NODE_VERSION }

      - name: Build
        run: npm run build

```

```

- name: Upload build artifacts
  uses: actions/upload-artifact@v4
  with:
    name: build-output
    path: dist/

test:
  runs-on: ubuntu-latest
  needs: build
  steps:
    - name: Checkout code
      uses: actions/checkout@v4

    - name: Setup environment
      uses: ../github/actions/setup-node-env
      with:
        node-version: ${{ env.NODE_VERSION }}

    - name: Download build artifacts
      uses: actions/download-artifact@v4
      with:
        name: build-output
        path: dist/

    - name: Run tests
      run: npm test

    - name: Upload test report
      uses: actions/upload-artifact@v4
      with:
        name: test-report
        path: test-results/

package:
  runs-on: ubuntu-latest
  needs: test
  if: github.ref == 'refs/heads/main'
  steps:
    - name: Checkout code
      uses: actions/checkout@v4

    - name: Setup environment
      uses: ../github/actions/setup-node-env
      with:
        node-version: ${{ env.NODE_VERSION }}

    - name: Download build artifacts
      uses: actions/download-artifact@v4
      with:
        name: build-output

```

```
    path: dist/

- name: Create package
  run: npm pack

- name: Upload package
  uses: actions/upload-artifact@v4
  with:
    name: npm-package
    path: '*.tgz'
```

3. Методические указания выполнения практической части работы

Задачи работы:

1. Проектирование многостадийного пайплайна:

1.1. Спроектировать структуру пайплайна из трёх и более этапов (build → test → package).

1.2. Определить зависимости между этапами и порядок их выполнения.

1.3. Схематично изобразить архитектуру пайплайна.

2. Реализация последовательного выполнения этапов:

2.1. Настроить этап сборки (build) с сохранением артефактов.

2.2. Настроить этап тестирования (test) с использованием артефактов предыдущего этапа.

2.3. Настроить этап упаковки (package) с созданием дистрибутивного артефакта.

3. Вынесение общих частей в шаблоны:

3.1. Идентифицировать повторяющиеся фрагменты конфигурации.

3.2. Реализовать базовые шаблоны с использованием extends / include / composite actions.

3.3. Вынести общие переменные и before_script в отдельные файлы.

4. Настройка передачи артефактов между этапами:

4.1. Настроить сохранение артефактов на этапе build.

4.2. Настроить загрузку артефактов на этапах test и package.

4.3. Проверить корректность передачи данных между этапами.

5. Оптимизация структуры репозитория:

5.1. Организовать файлы шаблонов в директории .gitlab/ или .github/.

5.2. Обеспечить читаемость и поддерживаемость конфигурации.

6. Оформление отчёта:

6.1. Представить листинги основного файла и всех файлов шаблонов.

6.2. Продемонстрировать скриншоты успешного выполнения каждого этапа.

6.3. Обосновать выбранную структуру шаблонизации.

Варианты индивидуальных заданий

3.1. Базовый уровень

Общее требование для базового уровня: создать трехстадийный пайплайн (build → test → package) с передачей артефактов между этапами. Использовать шаблоны для вынесения общих частей (как минимум один шаблон).

№	Язык/Платформа	Дополнительные требования
1	Python	На этапе build - установка зависимостей, на test - pytest, на package - создание wheel.
2	Python	На этапе build - линтинг (flake8), на test - unittest, на package - создание tar.gz архива.
3	Python	На этапе build - проверка типов (mypy), на test - coverage, на package - создание zip-архива.
4	JavaScript (Node.js)	На этапе build - npm ci, на test - npm test, на package - npm pack.
5	JavaScript (Node.js)	На этапе build - ESLint, на test - Jest, на package - создание Docker-образа (без публикации).
6	JavaScript	На этапе build - сборка webpack, на test -

	(Node.js)	Мocha, на package - создание tar.gz архива.
7	Go	На этапе build - go mod download, на test - go test, на package - go build
8	Go	На этапе build - go fmt, на test - go test -cover, на package - создание бинарного файла
9	Java (Maven)	На этапе build - mvn compile, на test - mvn test, на package - mvn package
10	Java (Gradle)	На этапе build - gradle assemble, на test - gradle test, на package - gradle build.
11	C++ (CMake)	На этапе build - cmake configure, на test - ctest, на package - make package
12	C (.NET)	На этапе build - dotnet restore, на test - dotnet test, на package - dotnet publish
13	Ruby	На этапе build - bundle install, на test - rspec, на package - gem build
14	PHP	На этапе build - composer install, на test - phpunit, на package - создание phar-архива
15	Rust	На этапе build - cargo fetch, на test - cargo test, на package - cargo build --release

3.2. Средний уровень

Общее требование для среднего уровня: пайплайн должен включать 4-5 этапов (build → test → lint → package → (опционально) deploy). Шаблоны должны быть вынесены в отдельные файлы. Должна быть настроена передача артефактов между всеми этапами.

№	Язык/Платформа	Дополнительные требования
1	Python	Добавить этап lint (flake8 + black --check) между build и test. Настроить кэширование

		зависимостей.
2	Python	Добавить этап security (bandit) между test и package. Настроить публикацию отчетов как артефактов
3	Python	Добавить этап docs (sphinx-build) после package. Артефакты — HTML-документация.
4	avaScript	Добавить этап lint (ESLint) между build и test. Настроить матрицу версий Node.js (18, 20).
5	JavaScript	Добавить этап type-check (TypeScript) между build и test. Настроить параллельное выполнение lint и type-check.
6	JavaScript	Добавить этап e2e-test (Playwright) после test. Настроить manual approval для e2e-test
7	Go	Добавить этап lint (golangci-lint) между build и test. Настроить параллельное выполнение test на разных ОС (ubuntu, windows).
8	Go	Добавить этап benchmark (go test -bench) после test. Сохранять результаты benchmark как артефакт
9	Java	Добавить этап coverage (JaCoCo) между test и package. Установить порог покрытия 80%.
10	Java	Добавить этап integration-test (Testcontainers) после test. Настроить зависимость от БД в контейнере
11	C++	Добавить этап static-analysis (clang-tidy) между build и test. Настроить генерацию отчетов
12	Docker + любой язык	Пайплайн для Docker-образа. Этапы: build (сборка образа), test (тесты внутри контейнера), package (публикация в registry), security (Trivy).

13	Terraform	Пайплайн для инфраструктуры. Этапы: init, validate, plan (с артефактом plan), apply (только main).
14	Ansible	Пайплайн для плейбуков. Этапы: syntax-check, lint (ansible-lint), test (molecule), package (сборка коллекции).
15	ROS 2	Пайплайн для ROS 2 пакета. Этапы: build (colcon build), test (colcon test), lint (ament_), package (создание deb-пакета).

3.3. Продвинутый уровень

Общее требование для продвинутого уровня: использовать динамические или дочерние пайплайны. Шаблоны должны быть параметризованы. Настроены условные этапы (запуск только для определенных веток/тегов). Реализовано ручное подтверждение для финального этапа.

№	Язык/Платформа	Дополнительные требования
1	Python	Реализовать child pipeline в GitLab CI: основной пайплайн генерирует дочерние для каждого микросервиса
2	Python	Реализовать матричный пайплайн с параметрами: версия Python (3.9-3.12), ОС (ubuntu, macos). Параллельный запуск
3	Python	Реализовать пайплайн с динамическими этапами: список тестов формируется на основе измененных файлов
4	JavaScript	Реализовать пайплайн с conditional stages: этап e2e запускается только при наличии метки «e2e» в PR

5	JavaScript	Реализовать монорепопозиторий (Yarn Workspaces/Nx) с пайплайном, запускающим задачи только для измененных пакетов
6	JavaScript	Реализовать пайплайн с авто-тегированием версии на основе commit message (semantic-release).
7	Go	Реализовать пайплайн с генерацией кода (go generate) перед сборкой. Артефакты - сгенерированный код
8	Go	Реализовать пайплайн с кросс-компиляцией под несколько платформ (linux, windows, darwin).
9	Java	Реализовать пайплайн с автоматической публикацией в Maven Central (Sonatype). С ручным подтверждением
10	Java	Реализовать пайплайн для Spring Boot с интеграционными тестами на testcontainers. Окружения: dev, staging, prod
11	Специфика БПЛА	Пайплайн для PX4/ArduPilot: этапы: build (кросс-компиляция прошивки), test (SITL в симуляторе), package (создание ОТА-пакета), deploy (ручное).
12	ROS 2 + Docker	Пайплайн для ROS 2 с использованием Docker. Этапы: build (сборка образа), test (тесты в контейнере), package (публикация), deploy (на парк роботов).
13	CI/CD для CI/CD	Пайплайн, который проверяет и валидирует другие пайплайны: синтаксис YAML, проверка шаблонов, тестирование на mock-репозитории

14	Multi-cloud	Пайплайн с развертыванием в два облака (AWS и Azure). Параллельное развертывание, сравнение результатов
15	Security-hardened	Пайплайн с этапами SAST, DAST, SCA, container scan. Автоматическое создание issue при уязвимостях

4. Контрольные вопросы

1. Какие преимущества дает разделение пайплайна на этапы (stages)?
2. Для чего используется механизм artifacts в CI/CD? Приведите пример.
3. В чем отличие dependencies от needs в GitLab CI?
4. Как организовать повторное использование кода в CI/CD-пайплайнах?
5. Что такое «шаблон» (template) в контексте CI/CD? Приведите пример использования.
6. Как передать артефакты между заданиями в GitHub Actions?
7. Для чего используется директива extends в GitLab CI?
8. Как организовать вынос общих частей пайплайна в отдельные файлы?
9. В чем отличие последовательного выполнения этапов от параллельного?
10. Как настроить ручное подтверждение (manual approval) перед выполнением этапа?
11. Что такое environment в CI/CD и для чего он используется?
12. Как настроить выполнение задания только для определенной ветки?
13. Какие проблемы решает использование cache в CI/CD?
14. В чем преимущества хранения конфигурации пайплайна в том же репозитории, что и код?

15. Как реализовать условное выполнение этапа на основе результата предыдущего?

5. Требования к отчету

Отчет должен содержать следующие разделы:

1. Титульный лист с указанием ФИО студента, группы, варианта, номера лабораторной работы.

2. Цель работы - формулируется на основе задания.

3. Теоретическая часть:

- Краткое описание выбранной CI/CD-платформы.
- Объяснение понятий: stages, artifacts, dependencies, templates.
- Схема разработанного пайплайна (можно в виде диаграммы).

4. Практическая часть:

- Листинг основного файла конфигурации пайплайна с комментариями.

- Листинг файлов шаблонов (если они вынесены).
- Листинг тестируемого кода (если он разрабатывался).
- Листинг тестов.

5. Результаты выполнения:

- Скриншоты успешного выполнения пайплайна в веб-интерфейсе.
- Скриншоты логов каждого этапа.
- Скриншоты артефактов (папка с результатами сборки, отчеты).

6. Анализ ошибок (если возникали):

- Описание ошибки.
- Причина возникновения.
- Способ устранения.

7. Ответы на контрольные вопросы (3-5 вопросов по выбору преподавателя).

8. Вывод - что удалось освоить, какие навыки приобретены, в чем преимущества шаблонизации.

6. Критерии оценивания

Критерий	Базовый уровень	Средний уровень	Продвинутый уровень
Пайплайн содержит 3 этапа (build/test/package)	2 балла	2 балла	2 балла
Все этапы успешно выполняются	2 балла	2 балла	2 балла
Корректно настроена передача артефактов	2 балла	2 балла	2 балла
Использованы шаблоны/повторное использование кода	1 балл	2 балла	2 балла
Шаблоны вынесены в отдельные файлы	0 баллов	2 балла	2 балла
Настроены условные этапы (only/rules/if)	0 баллов	1 балл	2 балла
Настроены окружения (environments)	0 баллов	0 баллов	1 балл
Реализованы динамические/дочерние пайплайны	0 баллов	0 баллов	2 балл
Качество отчета и ответы на вопросы	2 балла	2 балла	2 балла

Шкала перевода баллов в оценку:

Оценка	Базовый	Средний	Продвинутый

	уровень (макс. 9)	уровень (макс. 13)	уровень (макс. 18)
Отлично (5)	8-9 баллов	11-13 баллов	16-18 баллов
Хорошо (4)	6-7 баллов	9-10 баллов	13-15 баллов
Удовлетворительно (3)	4-5 баллов	6-8 баллов	9-12 баллов
Неудовлетворительно (2)	менее 4 баллов	менее 6 баллов	менее 9 баллов

Дополнительные критерии (штрафы):

- Отсутствие скриншотов выполнения пайплайна - минус 1 балл.
- Отсутствие листингов кода - минус 1 балл.
- Ошибки в синтаксисе YAML - минус 2 балла.
- Пайплайн не запускается - работа не засчитывается до исправления.

Повышающие коэффициенты (бонусы):

- Пайплайн включает более 3 этапов (для базового уровня) - +1 балл.
- Настроена автоматическая публикация отчета о тестировании в веб-интерфейсе - +1 балл.
- Реализован custom Docker-образ для раннера с предустановленными инструментами - +2 балла.

Лабораторная работа №3. Параллельное выполнение и оптимизация времени

Цель работы: Сформировать практические навыки оптимизации времени выполнения CI/CD-пайплайнов за счёт параллелизации заданий, матричных стратегий, шардирования тестов и кэширования зависимостей.

Формируемые компетенции: УК-1, ОПК-2, ПК-1

1. Теоретическое обоснование

1.1. Проблема времени выполнения пайплайнов

По мере роста проекта время выполнения CI/CD-пайплайна имеет тенденцию к увеличению:

- Увеличивается количество тестов (модульных, интеграционных, E2E).
- Растет сложность сборки (больше зависимостей, тяжеловесные фреймворки).
- Добавляются новые этапы (линтинг, проверка безопасности, сборка документации).

Последствия долгих пайплайнов:

- Разработчики реже запускают тесты локально, чаще пушат «сломанный» код.
- Увеличивается время обратной связи (feedback loop), страдает итеративность разработки.
- Долгое ожидание снижает продуктивность команды.

Целевые показатели времени выполнения (согласно DORA):

Метрика	Элитные команды	Высокий уровень	Средний уровень	Низкий уровень
Время выполнения CI	< 1 часа	1-6 часов	1-2 дня	> 1 недели
Время	< 1 часа	< 1 дня	< 1 недели	> 1 недели

Параллелизация тестов	parallel: число (для одного задания)	jobs.<job_id>.strategy.fail-fast

Пример матричной стратегии в GitLab CI:

```
test:
  stage: test
  parallel:
    matrix:
      - PYTHON_VERSION: ["3.9", "3.10", "3.11", "3.12"]
        OS: ["ubuntu", "alpine"]
  script:
    - docker run python:$PYTHON_VERSION python -m pytest tests/
```

Пример матричной стратегии в GitHub Actions:

```
test:
  runs-on: ${ matrix.os }
  strategy:
    matrix:
      node-version: [18.x, 20.x, 22.x]
      os: [ubuntu-latest, windows-latest, macos-latest]
    fail-fast: false # не останавливать другие задания при ошибке одного
  steps:
    - uses: actions/setup-node@v4
      with:
        node-version: ${ matrix.node-version }
    - run: npm test
```

1.3. Кэширование: принципы и практики

Кэширование - это сохранение результатов промежуточных вычислений (зависимости, скомпилированные файлы, кэш инструментов) для повторного использования в последующих запусках.

Что обычно кэшируют:

Тип кэша	Пример	Экономия

		времени
Менеджеры пакетов	npm, pip, maven, gradle, go mod	30-70%
Среда выполнения	виртуальное окружение Python, node_modules	20-50%
Кэш компилятора	.ccache, sccache, .gradle/caches	40-80%
Docker-слои	многослойные образы, buildx cache	50-90%

Схема работы кэша (рисунок 2):

Запуск 1 (первый)
Установка зависимостей → Сохранение в кэш → Выполнение тестов (30 сек) (сохранение) (10 сек) 30 сек + 10 сек = 40 сек

Запуск 2 (повторный)
Загрузка из кэша → (пропуск установки) → Выполнение тестов (5 сек) (10 сек) 5 сек + 10 сек = 15 сек

Экономия: 25 сек (62.5%)

Рисунок 2

Кэширование в GitLab CI:

```
variables:
  PIP_CACHE_DIR: "$CI_PROJECT_DIR/.cache/pip"

cache:
  key: "$CI_COMMIT_REF_SLUG" # ключ кэша (ветка)
  paths:
    - .cache/pip/
    - venv/
  policy: pull-push # (по умолчанию) загрузить и сохранить
```

Кэширование в GitHub Actions:

```
name: Cache pip packages
```

```

uses: actions/cache@v3
with:
  path: ~/.cache/pip
  key: ${{ runner.os }}-pip-${{ hashFiles('requirements.txt') }}
  restore-keys: |
    ${{ runner.os }}-pip-

```

1.4. Стратегии шардирования тестов (test splitting / sharding)

Шардирование (sharding) - это разделение набора тестов на несколько групп (шардов), которые выполняются параллельно.

Подходы к шардированию:

Подход	Описание	Преимущества	Недостатки
По имени файла	Разделение по алфавиту	Простота	Неравномерная нагрузка
По времени выполнения	Анализ истории, балансировка по времени	Равномерная загрузка	Требует сбора метрик
По модулям/пакетам	Функциональное разделение	Логичность	Может быть неравномерным
Случайное (round-robin)	Циклическое распределение	Простота, усреднение	Непредсказуемость

Пример шардирования в GitLab CI:

```

test:
  stage: test
  parallel: 4 # разделить на 4 параллельных задания
  script:
    - pytest tests/ --splits=$CI_NODE_TOTAL --group=$CI_NODE_INDEX

```

Пример шардирования в GitHub Actions:

```

jobs:
  test:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        shard: [1, 2, 3, 4]
    steps:
      - run: pytest --shard=${{ matrix.shard }}/4 tests/

```

1.5. Измерение и сравнение производительности

Ключевые метрики для анализа:

Метрика	Описание	Единица измерения
Total time	Общее время выполнения пайплайна	секунды/минуты
Setup time	Время установки зависимостей	секунды
Test time	Время выполнения тестов	секунды
Build time	Время сборки/компиляции	секунды
Cache hit rate	Доля успешных попаданий в кэш	%

Инструменты для измерения:

- Встроенные таймеры CI-платформы.
- Ручное логирование с time или date.
- Интеграция с Prometheus + Grafana для сбора метрик.

2. Примеры практического выполнения

2.1. Пример для Python: параллелизация тестов и кэширование

Исходный пайплайн (медленный):

```

stages:
  - test

```

```
test:
  stage: test
  script:
    - pip install -r requirements.txt
    - pytest tests/
```

Оптимизированный пайплайн (с кэшем и параллелизацией):

```
stages:
  - test

variables:
  PIP_CACHE_DIR: "${CI_PROJECT_DIR}/.cache/pip"

cache:
  key: "${CI_COMMIT_REF_SLUG}"
  paths:
    - .cache/pip/
    - venv/

before_script:
  - python -m venv venv
  - source venv/bin/activate
  - pip install -r requirements.txt

test:
  stage: test
  parallel: 4
  script:
    - source venv/bin/activate
    - pytest tests/ --split=4 --group=${CI_NODE_INDEX}
  artifacts:
    paths:
      - reports/
```

Сравнение времени:

Конфигурация	Время выполнения
Исходная (без оптимизаций)	320 секунд
С кэшированием	180 секунд
С параллелизацией (4 шарда)	65 секунд
Кэш + параллелизация	50 секунд

2.2. Пример для Node.js (GitHub Actions) с матрицей и кэшем

```
name: Optimized CI Pipeline

on: [push, pull_request]

jobs:
  test:
    runs-on: ${ matrix.os }
    strategy:
      matrix:
        node-version: [18.x, 20.x, 22.x]
        os: [ubuntu-latest, windows-latest]
        fail-fast: false # не останавливать при ошибке одного

    steps:
      - uses: actions/checkout@v4

      - name: Cache Node.js modules
        uses: actions/cache@v3
        with:
          path: ~/.npm
          key: ${ runner.os }-node-${ matrix.node-version }-${
hashFiles('package-lock.json') }
          restore-keys: |
            ${ runner.os }-node-${ matrix.node-version }-

      - name: Install dependencies
        run: npm ci

      - name: Run tests with sharding
        run: npm run test:sharded -- --shard=${ strategy.job-index }/${
strategy.job-total }

      - name: Upload coverage
        uses: actions/upload-artifact@v4
        with:
          name: coverage-${ matrix.os }-node${ matrix.node-version }
          path: coverage/

  lint:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/cache@v3
        with:
          path: ~/.npm
          key: ${ runner.os }-lint-${ hashFiles('package-lock.json') }
```



```
- run: npm ci
- run: npm run lint
```

2.3. Пример для Go: параллельная компиляция и кэш модулей

```
stages:
  - build
  - test

variables:
  GOCACHE: "${CI_PROJECT_DIR}/.gocache"
  GOMODCACHE: "${CI_PROJECT_DIR}/.gomodcache"

cache:
  key: "$CI_COMMIT_REF_SLUG"
  paths:
    - .gocache/
    - .gomodcache/

build:
  stage: build
  parallel:
    matrix:
      - GOOS: ["linux", "windows", "darwin"]
        GOARCH: ["amd64", "arm64"]
  script:
    - go build -o myapp_${GOOS}_${GOARCH} ./cmd/app
  artifacts:
    paths:
      - myapp_*

test:
  stage: test
  parallel: 4
  script:
    - go test -v -shard=$CI_NODE_INDEX/$CI_NODE_TOTAL ./...
```

3. Методические указания выполнения практической части работы

Задачи работы:

1. Анализ времени выполнения исходного пайплайна:

- 1.1. Зафиксировать время выполнения каждого этапа исходной (неоптимизированной) версии.
- 1.2. Выявить «узкие места» - этапы с наибольшей длительностью.
- 1.3. Рассчитать общее время выполнения пайплайна.

2. Настройка кэширования зависимостей:

- 2.1. Определить, какие зависимости и промежуточные файлы подлежат кэшированию.
- 2.2. Настроить кэш для менеджера пакетов (pip, npm, maven, go mod и т.д.).
- 2.3. Измерить и зафиксировать hit rate кэша.

3. Реализация параллелизации заданий:

- 3.1. Разделить независимые задания внутри одного этапа для параллельного выполнения.
- 3.2. Настроить матричную стратегию (по версиям языка, операционным системам).
- 3.3. Проверить корректность параллельного выполнения (отсутствие конфликтов).

4. Реализация шардирования (splitting) тестов:

- 4.1. Разделить набор тестов на несколько групп (шардов).
- 4.2. Настроить параллельный запуск шардов на разных раннерах.
- 4.3. Проанализировать равномерность распределения нагрузки между шардами.

5. Измерение и сравнение производительности:

- 5.1. Зафиксировать время выполнения после каждой оптимизации.
- 5.2. Построить таблицу и график сравнения времени выполнения.
- 5.3. Рассчитать процент ускорения (speedup) и эффективность оптимизаций.

6. Анализ эффективности и формулирование выводов:

- 6.1. Оценить, какой метод оптимизации дал наибольший эффект.

- 6.2. Сравнить теоретическое и реальное ускорение (с учётом накладных расходов).
- 6.3. Сформулировать рекомендации по дальнейшей оптимизации.

7. Оформление отчёта:

- 7.1. Представить таблицу сравнения времени выполнения всех конфигураций.
- 7.2. Включить график или диаграмму ускорения.
- 7.3. Привести скриншоты параллельного выполнения заданий.
- 7.4. Обосновать выбор стратегий оптимизации для конкретного проекта.

Варианты заданий

3.1. Базовый уровень

Общее требование: Настроить кэширование зависимостей. Разделить тесты на 2 параллельных шарда. Предоставить сравнение времени выполнения до и после оптимизации.

№	Язык/Платформа	Особенности проекта
1	Python (pytest)	200+ модульных тестов, зависимости через requirements.txt
2	Python (pytest)	150+ тестов, зависимости через pipenv
3	Python (unittest)	100+ тестов, зависимости через poetry
4	JavaScript (Jest)	300+ unit-тестов, зависимости npm
5	JavaScript (Mocha)	200+ тестов, зависимости yarn
6	JavaScript (Vitest)	150+ тестов, зависимости pnpm
7	Go	100+ тестов, модули go mod
8	Java (Maven)	200+ тестов JUnit, зависимости Maven
9	Java (Gradle)	150+ тестов JUnit, зависимости Gradle
10	C (.NET)	250+ тестов xUnit, NuGet зависимости

11	C++ (CMake)	80+ тестов Google Test, система сборки CMake
12	Ruby (RSpec)	120+ тестов, зависимости bundler
13	PHP (PHPUnit)	100+ тестов, Composer зависимости
14	Rust	90+ тестов, cargo зависимости
15	Kotlin (JVM)	130+ тестов, Gradle зависимости

3.2. Средний уровень

Общее требование: Настроить матричную стратегию по 2-3 параметрам (например, версия языка/ОС). Реализовать шардирование тестов на 4 потока. Сравнить время выполнения 3 различных конфигураций кэша.

№	Язык/Платформа	Особенности проекта
1	Python	Матрица: Python 3.9, 3.11, 3.12; OS: ubuntu, alpine
2	Python	Матрица: Python 3.10, 3.11; OS: ubuntu, macos. Кэш pip и venv
3	Python	Матрица: зависимости (minimal, full). Кэш с разными ключами
4	JavaScript	Матрица: Node 18, 20, 22; OS: ubuntu, windows
5	JavaScript	Матрица: npm, yarn, pnpm (три менеджера пакетов)
6	JavaScript	Матрица: браузерные тесты (Chrome, Firefox, Safari через Playwright)
7	Go	Матрица: Go 1.20, 1.21, 1.22; OS: ubuntu, alpine
8	Go	Матрица: GOOS (linux, windows) × GOARCH (amd64, arm64)
9	Java	Матрица: Java 11, 17, 21; OS: ubuntu,

		windows
10	Java	Матрица: Maven vs Gradle (сравнение времени кэширования)
11	C	Матрица: .NET 6.0, 7.0, 8.0; OS: ubuntu, windows
12	C++	Матрица: GCC 11, 12, 13; Clang 14, 15. Кэш ccache
13	Rust	Матрица: stable, beta, nightly; OS: ubuntu, macos
14	Ruby	Матрица: Ruby 2.7, 3.0, 3.1, 3.2
15	PHP	Матрица: PHP 8.1, 8.2, 8.3; OS: ubuntu, alpine

3.3. Продвинутый уровень

Общее требование: Реализовать динамическое шардирование на основе времени выполнения предыдущих запусков. Настроить сценарий, где часть заданий выполняется параллельно, часть - последовательно. Провести анализ эффективности (график времени, процент ускорения). Интегрировать сбор метрик в Prometheus/Grafana.

№	Язык/Платформа	Особенности проекта
1	Python	Динамическое шардирование: анализ истории тестов в JSON, балансировка по времени
2	Python	Кэш с разными стратегиями (branch, tag, commit) и сравнение hit rate
3	Python + Docker	Параллельная сборка Docker-образов с кэшированием слоев (buildx cache)
4	JavaScript	Динамическое шардирование на основе coverage-данных (тяжелые тесты - в

		отдельный шард)
5	JavaScript + Puppeteer	Параллельный запуск E2E-тестов в headless-браузерах (4+ параллельных инстанса)
6	JavaScript	Анализ времени выполнения: сбор метрик в GitHub Actions, экспорт в JSON
7	Go	Оптимизация кэша: разделение GOCACHE и GOMODCACHE, разные политики
8	Go + Benchmarks	Параллельный запуск бенчмарков, сравнение результатов, автоматическая регрессия
9	Java	Настройка Gradle Build Cache (локальный и удаленный), сравнение времени сборки
10	Java + Testcontainers	Параллельный запуск интеграционных тестов с контейнерами (конфликты портов)
11	C++	Настройка sscache с разными размерами кэша (1GB, 5GB, 10GB) и сравнение hit rate
12	Rust	Настройка sscache (S3 или локальный), сравнение с обычным кэшем
13	Специфика БПЛА	Параллельная симуляция нескольких дронов в Gazebo, кэширование моделей мира
14	ROS 2	Параллельный запуск тестов разных пакетов, кэширование сборки colcon
15	омплексный	Пайплайн с 4 параллельными этапами (build, test, lint, security), динамическое распараллеливание на основе доступных ресурсов раннера

4. Контрольные вопросы

1. Какие факторы влияют на время выполнения CI/CD-пайплайна?
2. В чем отличие параллелизации заданий от матричной стратегии?

3. Как работает кэширование зависимостей в CI/CD? Какие данные обычно кэшируются?
4. Что такое шардирование (sharding) тестов и зачем оно нужно?
5. Какие стратегии балансировки нагрузки при шардировании вы знаете?
6. Как настроить матричную стратегию в GitLab CI? В GitHub Actions?
7. Как измерить эффективность кэширования (hit rate)?
8. Какие риски связаны с параллельным выполнением тестов (race conditions, конфликты портов)?
9. В чем отличие fail-fast от fail-safe при параллельном выполнении?
10. Как настроить кэш с разными ключами для разных веток?
11. Какие инструменты можно использовать для сбора метрик времени выполнения пайплайнов?
12. Как реализовать динамическое шардирование на основе времени выполнения тестов?
13. Почему Docker-слои эффективно кэшируются и как это использовать в CI/CD?
14. Какие ограничения накладывает использование кэша в GitHub Actions (размер, срок хранения)?
15. Как оценить экономическую эффективность оптимизации пайплайна (время разработчиков × стоимость раннеров)?

5. Требования к отчету

Отчет должен содержать следующие разделы:

1. Титульный лист с указанием ФИО студента, группы, варианта.
2. Цель работы - формулируется на основе задания.

3. Теоретическая часть:

- Обоснование необходимости оптимизации пайплайнов.
- Описание выбранных методов оптимизации (параллелизация, кэширование, шардирование).
- Схема оптимизированного пайплайна.

4. Практическая часть:

- Исходная конфигурация пайплайна (без оптимизаций).
- Оптимизированная конфигурация пайплайна (с комментариями изменений).
- Листинг скриптов для шардирования (если применялись).

5. Результаты измерений:

- Таблица сравнения времени выполнения (до/после, для каждого этапа).
- График или диаграмма (столбчатая или линейная) изменения времени.
- Скриншоты логов с указанием времени выполнения.

6. Анализ эффективности:

- Процент ускорения (speedup).
- Эффективность кэширования (hit rate, если доступно).
- Оценка экономии вычислительных ресурсов.

7. Ответы на контрольные вопросы (3-5 вопросов).

8. Вывод - какие методы оптимизации оказались наиболее эффективными, рекомендации.

6. Критерии оценивания

Критерий	Базовый уровень	Средний уровень	Продвинутый уровень
Настроено кэширование зависимостей	2 балла	2 балла	2 балла
Реализовано шардирование	2 балла	2 балла	2 балла

тестов (≥ 2 шардов)			
Приведено сравнение времени (таблица + график)	2 балла	2 балла	2 балла
Достигнуто ускорение $>30\%$	1 балл	1 балл	1 балл
Реализована матричная стратегия (≥ 2 параметров)	0 баллов	2 балла	2 балла
Выполнен анализ hit rate кэша	0 баллов	1 балл	2 балла
Реализовано динамическое шардирование (по времени)	0 баллов	0 баллов	2 балла
Интегрирован сбор метрик в Prometheus/Grafana	0 баллов	0 баллов	2 балла
Качество отчета и ответы на вопросы	2 балла	2 балла	2 балла

Шкала перевода баллов в оценку:

Оценка	Базовый уровень (макс. 9)	Средний уровень (макс. 12)	Продвинутый уровень (макс. 17)
Отлично (5)	8-9 баллов	11-12 баллов	15-17 баллов
Хорошо (4)	6-7 баллов	9-10 баллов	12-14 баллов
Удовлетворительно (3)	4-5 баллов	6-8 баллов	9-11 баллов
Неудовлетворительно (2)	менее 4 баллов	менее 6 баллов	менее 9 баллов

Дополнительные критерии (штрафы):

- Отсутствие скриншотов сравнения времени - минус 1 балл.
- Некорректное сравнение (разные условия выполнения) - минус 2 балла.

- Не указана версия CI-платформы и раннера - минус 1 балл.

Бонусные баллы (макс. +3):

- Ускорение более 50% - +1 балл.
- Ускорение более 70% - +2 балла.
- Настройка удаленного кэша (S3, GCS) - +1 балл.

Лабораторная работа №4. Автоматизация модульного и интеграционного тестирования

Цель работы: сформировать практические навыки интеграции различных уровней автоматизированного тестирования (модульного и интеграционного) в CI/CD-пайплайн с генерацией отчетов о результатах тестирования и их публикацией в качестве артефактов.

Теоретическое обоснование

1. Уровни тестирования и их место в CI/CD

Пирамида тестирования в контексте CI/CD (рисунок 1):

E2E / UI тесты (мало, выборочно)	← медленные, дорогие
Интеграционные тесты (тестирование взаимодействия)	← средние по скорости
Модульные тесты (тестирование отдельных units)	← быстрые, много

Рисунок 1

Уровень	Что тестирует	Скорость	Количество в CI
Модульные тесты (unit)	Отдельные функции, классы, методы	Очень быстрые (мс)	Много (сотни/тысячи)
Интеграционные тесты	Взаимодействие компонентов (БД, API, сервисы)	Средние (сек)	Умеренно
E2E тесты	Полные пользовательские сценарии	Медленные (мин)	Мало (ключевые)

2. Форматы отчетов о тестировании

JUnit XML - стандартный формат для хранения результатов тестов, поддерживаемый большинством CI-платформ.

Структура JUnit XML:

```
<testsuites>
<testsuite name="test_module" tests="5" failures="1" errors="0" skipped="0" time="0.123">
<testcase name="test_addition" classname="math_utils_test" time="0.045"/>
<testcase name="test_subtraction" classname="math_utils_test" time="0.032"/>
<testcase name="test_division_by_zero" classname="math_utils_test" time="0.046">
<failure message="Division by zero should raise exception">AssertionError</failure>
</testcase>
</testsuite>
</testsuites>
```

Поддержка форматов в CI-платформах:

CI-платформа	Формат отчетов	Визуализация
GitLab CI	JUnit XML	Вкладка Tests, график трендов
GitHub Actions	JUnit XML, JSON	Аннотации в PR, Actions summary
Jenkins	JUnit XML	Графики, тренды, история

3. Интеграционное тестирование с контейнерами

Проблема: Интеграционные тесты требуют наличия внешних сервисов (БД, очереди, кэш), которые не всегда доступны в CI-среде.

Решение: Использование контейнеров (Docker) для поднятия зависимостей на время выполнения тестов (рисунок 2).

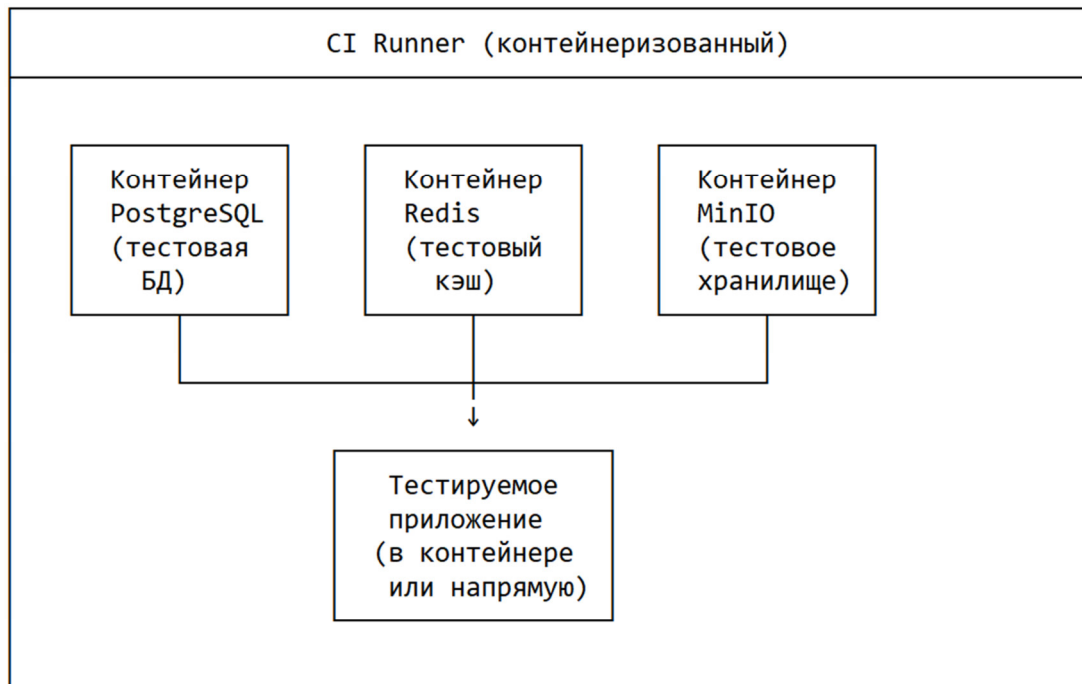


Рисунок 2

Инструменты для интеграционного тестирования с контейнерами:

Инструмент	Язык	Описание
Testcontainers	Java, Go, Python, .NET	Библиотека для управления контейнерами из тестов
Docker	Любой	Оркестрация нескольких контейнеров для тестов
Compose	Любой	Оркестрация нескольких контейнеров для тестов
Podman	Любой	Альтернатива Docker, rootless
GitLab Services	Любой	Встроенный механизм поднятия сервисов в CI

4. Покрытие кода (Code Coverage)

Покрытие кода - метрика, показывающая, какой процент кода выполняется во время тестов.

Инструмент	Язык	Формат отчета
pytest-cov / coverage.py	Python	HTML, XML, JSON
Jest --coverage	JavaScript	HTML, lcov, JSON
go test -cover	Go	Текст, HTML
JaCoCo	Java	HTML, XML
OpenCover	C	XML
gcov / lcov	C/C++	HTML

Настройка порогов покрытия:

```
# Пример для Python
script:
  - pytest --cov=src --cov-fail-under=80 tests/
```

5. Публикация отчетов как артефактов

Артефакты тестирования сохраняются для последующего анализа и визуализации.

```
# GitLab CI
test:
  stage: test
  script:
    - pytest --junitxml=report.xml --cov=src --cov-report=html
  artifacts:
    paths:
      - report.xml
      - htmlcov/
  reports:
    junit: report.xml
  expire_in: 1 week
```

```
# GitHub Actions
- name: Upload test results
  uses: actions/upload-artifact@v4
```

```
with:
  name: test-reports
  path: |
    report.xml
    coverage/
```

Пример практического выполнения

3.1. Пример для Python (GitLab CI) — модульные тесты

Файл.gitlab-ci.yml:

```
stages:
  - test

variables:
  PIP_CACHE_DIR: "$CI_PROJECT_DIR/.cache/pip"

cache:
  paths:
    - .cache/pip/

test:
  stage: test
  image: python:3.11-slim
  before_script:
    - pip install pytest pytest-cov
    - pip install -r requirements.txt
  script:
    - pytest tests/unit/ --junitxml=unit-report.xml --cov=src --cov-report=html -
-cov-report=term
  artifacts:
    paths:
      - unit-report.xml
      - htmlcov/
  reports:
    junit: unit-report.xml
  expire_in: 1 week
```

2. Пример для Python (GitLab CI) - интеграционные тесты с PostgreSQL

Файл .gitlab-ci.yml:

```

stages:
  - integration-test

variables:
  POSTGRES_USER: testuser
  POSTGRES_PASSWORD: testpass
  POSTGRES_DB: testdb
  DATABASE_URL: "postgresql://testuser:testpass@postgres:5432/testdb"

integration-test:
  stage: integration-test
  image: python:3.11-slim
  services:
    - postgres:15
  variables:
    POSTGRES_USER: testuser
    POSTGRES_PASSWORD: testpass
    POSTGRES_DB: testdb
  before_script:
    - pip install pytest pytest-cov psycpg2-binary
    - pip install -r requirements.txt
    - sleep 5 # ожидание готовности PostgreSQL
  script:
    - pytest tests/integration/ --junitxml=integration-report.xml --cov=src --cov-append
  artifacts:
    paths:
      - integration-report.xml
  reports:
    junit: integration-report.xml

```

3. Пример для Node.js (GitHub Actions) с Jest и Testcontainers

Файл `.github/workflows/test.yml`:

```

name: Test Suite

on: [push, pull_request]

jobs:
  unit-tests:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: '20'

```

```

    cache: 'npm'
  - run: npm ci
  - run: npm run test:unit -- --coverage --testResultsProcessor=jest-junit
  - name: Upload coverage
    uses: actions/upload-artifact@v4
    with:
      name: coverage-report
      path: coverage/
  - name: Upload test results
    uses: actions/upload-artifact@v4
    with:
      name: unit-test-results
      path: test-results/

integration-tests:
  runs-on: ubuntu-latest
  needs: unit-tests
  steps:
    - uses: actions/checkout@v4
    - uses: actions/setup-node@v4
      with:
        node-version: '20'
    - run: npm ci
    - name: Run integration tests with Testcontainers
      run: npm run test:integration -- --forceExit
    - name: Upload integration results
      uses: actions/upload-artifact@v4
      with:
        name: integration-test-results
        path: test-results-integration/

```

4. Пример для Java (Maven) с Testcontainers

Файл pom.xml (фрагмент):

```

<dependency>
  <groupId>org.testcontainers</groupId>
  <artifactId>testcontainers</artifactId>
  <version>1.19.3</version>
  <scope>test</scope>
</dependency>
<dependency>

```



```
<groupId>org.testcontainers</groupId>  
<artifactId>postgresql</artifactId>  
<version>1.19.3</version>  
<scope>test</scope>  
</dependency>
```

Файл .gitlab-ci.yml:

```
test:  
  stage: test  
  image: maven:3.9-eclipse-temurin-17  
  services:  
    - docker:dind  
  variables:  
    DOCKER_HOST: tcp://docker:2375  
    DOCKER_TLS_CERTDIR: ""  
  script:  
    - mvn test  
    - mvn jacoco:report  
  artifacts:  
    paths:  
      - target/surefire-reports/  
      - target/site/jacoco/  
  reports:  
    junit: target/surefire-reports/*.xml
```

Методические указания выполнения практической части работы

Задачи работы

1. Настройка модульного тестирования:

- Добавить в пайплайн этап запуска модульных тестов.
- Настроить генерацию отчетов в формате JUnit XML.
- Обеспечить публикацию отчетов как артефактов пайплайна.

2. Настройка интеграционного тестирования:

- Разработать интеграционные тесты, требующие внешних зависимостей (БД, API).
- Настроить поднятие тестовой базы данных (контейнер) в CI-среде.

- Обеспечить выполнение интеграционных тестов после успешного прохождения модульных.

3. Анализ и визуализация результатов:

- Настроить отображение результатов тестов в веб-интерфейсе CI-платформы.

- Настроить публикацию отчетов о покрытии кода (coverage report).

4. Управление окружениями для тестирования:

- Настроить изолированную среду для тестирования (контейнеры, виртуальное окружение).

- Обеспечить автоматическую очистку ресурсов после выполнения тестов.

5. Оформление отчёта:

- Зафиксировать конфигурацию пайплайна с тестовыми этапами.

- Представить скриншоты успешного выполнения и отчетов.

Варианты заданий

1. Базовый уровень

Общее требование: Добавить в пайплайн этап модульного тестирования с генерацией JUnit-отчета. Настроить публикацию отчета как артефакта.

№	Язык/Платформа	Тестовый фреймворк	Генерация отчета
1	Python	pytest	JUnit XML, coverage
2	Python	unittest	JUnit XML (через xmlrunner)
3	Python	pytest	JUnit XML, HTML coverage
4	JavaScript (Node.js)	Jest	JUnit XML, lcov
5	JavaScript	Mocha + Chai	JUnit (mocha-junit-reporter)

	(Node.js)		
6	JavaScript (Node.js)	Vitest	JUnit XML, coverage
7	Go	testing + testify	JUnit (go-junit-report)
8	Go	testing	coverage (go test -cover)
9	Java (Maven)	JUnit 5	Surefire XML, JaCoCo
10	Java (Gradle)	JUnit 5	JUnit XML, JaCoCo
11	C (.NET)	xUnit	JUnit XML, coverlet
12	C (.NET)	NUnit	JUnit XML, OpenCover
13	Ruby	RSpec	JUnit XML, simplecov
14	PHP	PHPUnit	JUnit XML, coverage
15	Rust	cargo test	JUnit (cargo2junit)

4.2. Средний уровень

Общее требование: Добавить интеграционные тесты с использованием контейнера (БД, кэш, очередь). Настроить этапы: модульные тесты → интеграционные тесты (с зависимостью). Публиковать отчеты отдельно.

№	Язык/Платформа	Внешний сервис для интеграционных тестов	Инструмент
1	Python + FastAPI	PostgreSQL	pytest + psycopg2
2	Python + Django	PostgreSQL	pytest-django
3	Python + Flask	Redis	pytest + fakeredis / реальный Redis
4	Python + SQLAlchemy	MySQL	pytest + pymysql

5	JavaScript + Express	MongoDB	Jest + mongodb-memory-server
6	JavaScript + NestJS	PostgreSQL	Jest + Testcontainers
7	JavaScript + Next.js	Redis	Jest + ioredis-mock
8	Go + Gin	PostgreSQL	testify + testcontainers-go
9	Go + Echo	MySQL	testing + dockertest
10	Java + Spring Boot	PostgreSQL	JUnit 5 + Testcontainers
11	Java + Quarkus	Kafka	JUnit 5 + Testcontainers
12	C + ASP.NET Core	SQL Server	xUnit + Testcontainers
13	Ruby on Rails	PostgreSQL	RSpec + database_cleaner
14	PHP + Laravel	MySQL	PHPUnit + Docker Compose
15	Rust + Actix	PostgreSQL	cargo test + testcontainers

4.3. Продвинутый уровень

Общее требование: Настроить несколько контейнерных сервисов (БД + кэш + очередь). Реализовать проверку покрытия кода с пороговым значением (например, >80%). Настроить автоматическую публикацию отчета о покрытии в виде HTML-страницы.

№	Язык/Платформа	Сценарий	Дополнительные требования
1	Python	БД + Redis + MinIO	Настроить параллельный запуск модульных и интеграционных тестов
2	Python	PostgreSQL + RabbitMQ	Настроить retry при недоступности сервисов
3	Python	TimescaleDB + Kafka	Реализовать тесты с временными рядами

4	JavaScript	PostgreSQL + Redis + Elasticsearch	Настроить custom Docker Compose для тестов
5	JavaScript	MongoDB + RabbitMQ + MinIO	Реализовать тесты с очередью сообщений
6	JavaScript	MySQL + Redis + S3 (mock)	Настроить GitHub Actions с матрицей версий БД
7	Go	PostgreSQL + Redis + NATS	Реализовать интеграционные тесты с gRPC
8	Go	MySQL + Kafka + ClickHouse	Настроить генерацию mock-данных для тестов
9	Java	PostgreSQL + Redis + Kafka	Testcontainers + параллельный запуск (JUnit 5 parallel)
10	Java	MySQL + RabbitMQ + Elasticsearch	Настроить динамическое выделение портов
11	C	SQL Server + Redis + Azure Storage (mock)	Реализовать тесты с CosmosDB эмулятором
12	Специфика БПЛА	MAVSDK + симулятор	Интеграционные тесты с симуляцией полета
13	ROS 2	ROS 2 + Gazebo	Тестирование под ROS 2 с симуляцией
14	Микросервисы	2+ сервиса + БД	Тестирование взаимодействия микросервисов
15	Комплексный	Все выше + security scan	Настроить fail пайплайна при покрытии <85%

Контрольные вопросы:

1. В чем отличие модульных тестов от интеграционных?
2. Для чего используется формат JUnit XML в CI/CD?
3. Как настроить генерацию отчета о покрытии кода в выбранном языке?
4. Какие проблемы решает использование Testcontainers?
5. Как поднять тестовую базу данных в GitLab CI с помощью services?
6. В чем отличие services в GitLab CI от Testcontainers?
7. Как настроить порог покрытия кода, при котором пайплайн падает?
8. Какие типы артефактов тестирования стоит сохранять?
9. Как опубликовать отчет о покрытии в виде HTML-страницы?
10. Как избежать конфликта портов при параллельном запуске интеграционных тестов?
11. Как обеспечить очистку тестовых данных после выполнения тестов?
12. Как настроить зависимость интеграционных тестов от модульных (последовательность)?
13. Как визуализировать тренды покрытия кода в GitLab CI?
14. Какие инструменты позволяют запускать тесты с реальными зависимостями в CI?
15. Как протестировать код, работающий с очередями сообщений (RabbitMQ/Kafka)?

Требования к отчету

Отчет должен содержать следующие разделы:

1. Титульный лист с указанием ФИО студента, группы, варианта.
2. Цель и задачи работы.
3. Теоретическая часть:
 - Описание уровней тестирования (unit, integration).
 - Обоснование выбора тестового фреймворка и инструментов.
 - Схема взаимодействия компонентов при интеграционном тестировании.
4. Практическая часть:
 - Листинг конфигурации пайплайна с комментариями.
 - Листинг тестов (примеры модульных и интеграционных тестов).
 - Листинг кода приложения (если разрабатывался).
5. Результаты выполнения:
 - Скриншоты успешного выполнения пайплайна.
 - Скриншоты отчета о тестировании (JUnit XML, вкладка Tests).
 - Скриншоты отчета о покрытии кода (HTML страница).
 - Скриншоты артефактов (файлы отчетов).
6. Анализ покрытия:
 - Таблица с показателями покрытия (по модулям/файлам).
 - Выявление непокрытых участков кода.
7. Ответы на контрольные вопросы (3-5 вопросов).
8. Вывод - какие навыки приобретены, сложности при выполнении.

Критерии оценивания

Критерий	Базовый уровень	Средний уровень	Продвинутый уровень
Настроен этап модульного тестирования	2 балла	2 балла	2 балла
Сгенерирован и опубликован JUnit-отчет	2 балла	2 балла	2 балла

Настроен этап интеграционного тестирования	0 баллов	2 балла	2 балла
Использован контейнер для внешней зависимости	0 баллов	2 балла	2 балла
Настроена публикация отчета о покрытии	1 балл	1 балл	2 балла
Установлен порог покрытия ($\geq 80\%$)	0 баллов	1 балл	2 балла
Использовано 2+ контейнера в интеграционных тестах	0 баллов	0 баллов	2 балла
Настроен параллельный запуск unit и integration	0 баллов	0 баллов	1 балл
Качество отчета и ответы на вопросы	2 балла	2 балла	2 балла

Шкала перевода баллов в оценку:

Оценка	Базовый уровень (макс. 7)	Средний уровень (макс. 12)	Продвинутый уровень (макс. 17)
Отлично (5)	6-7 баллов	11-12 баллов	15-17 баллов
Хорошо (4)	5 баллов	9-10 баллов	12-14 баллов
Удовлетворительно (3)	3-4 балла	6-8 баллов	9-11 баллов
Неудовлетворительно (2)	менее 3 баллов	менее 6 баллов	менее 9 баллов

Дополнительные критерии (штрафы):

- Отсутствие скриншотов отчетов - минус 1 балл.

- Покрытие кода <70% (при установленном пороге) - минус 1 балл.
- Интеграционные тесты не проходят - работа не засчитывается до исправления.

Бонусные баллы (макс. +3):

- Покрытие кода >90% - +1 балл.
- Интеграция с Allure Report - +1 балл.
- Настроена автоматическая публикация coverage badge — +1 балл.

Лабораторная работа №5. Анализ покрытия кода тестами

Цель работы: Сформировать практические навыки настройки инструментов анализа покрытия кода, интерпретации отчетов о покрытии и автоматизации контроля качества тестирования через установку пороговых значений с автоматическим срывом пайплайна при недостижении заданного уровня.

Теоретическое обоснование

1. Понятие покрытия кода (Code Coverage)

Покрытие кода - это метрика, показывающая, какая часть исходного кода программы была выполнена во время прогона тестов.

Метрики покрытия (от наиболее важных к менее важным):

Метрика	Описание	Типичный целевой порог
Строк (Lines)	Доля выполненных строк кода	80-90%
Инструкций (Instructions)	Доля выполненных байт-код/инструкций (Java, .NET)	80-90%

Ветвей (Branches)	Доля выполненных условных ветвей (if/else, switch)	70-80%
Функций (Functions)	Доля вызванных функций/методов	80-90%
Классов (Classes)	Доля загруженных/использованных классов	70-80%

Иерархия метрик покрытия (рисунок 1):

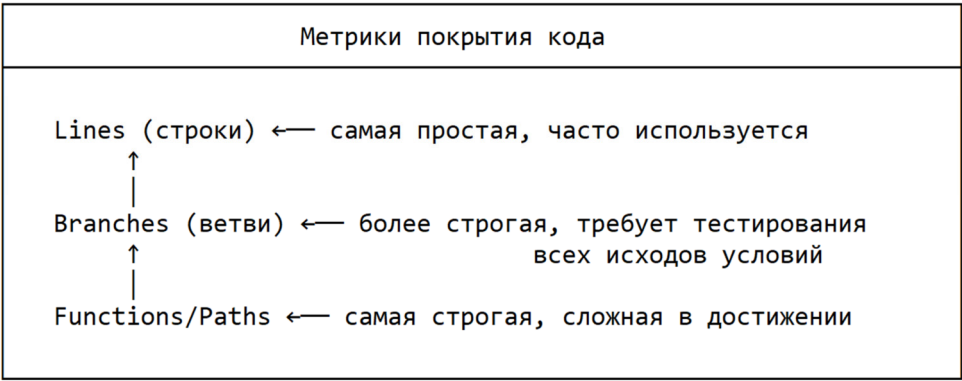


Рисунок 1

2. Инструменты анализа покрытия

Язык	Инструмент	Команда генерации	Форматы отчетов
Python	coverage.py, pytest-cov	pytest --cov=src --cov-report=html	HTML, XML, JSON, LCov
JavaScript	Jest, Istanbul/nyc	jest --coverage	HTML, LCov, JSON, Clover
TypeScript	Jest, Vitest	vitest --coverage	HTML, JSON, LCov
Go	go test -cover	go test -coverprofile=coverage.out	TXT, HTML
Java	JaCoCo, Cobertura	mvn jacoco:report	HTML, XML, CSV

C (.NET)	coverlet, OpenCover	dotnet test --collect:"XPlat Code Coverage"	XML, Cobertura
C/C++	gcov, lcov, gcovr	gcovr --html --output=coverage.html	HTML, XML, JSON
Ruby	SimpleCov	SimpleCov.start	HTML, JSON
PHP	PHPUnit + Xdebug	phpunit --coverage-html	HTML, XML
Rust	tarpaulin, grcov	cargo tarpaulin --out Html	HTML, JSON, XML

3. Принцип работы coverage.py (Python)

Схема работы coverage.py (рисунок 2):



Рисунок 2

Пример HTML-отчета coverage.py (рисунок 3):

модуль: src/calculator.py		покрытие:95%
<pre> 1 def add(a, b): 2 return a + b 4 def divide(a, b): 5 if b == 0: 6 raise ValueError("Division by zero") 7 return a / b </pre>		

Рисунок 3

4. Настройка порогов покрытия

Пороговое значение (threshold) - минимально допустимый уровень покрытия, при недостижении которого пайплайн считается проваленным.

Подходы к установке порогов:

Подход	Описание	Плюсы	Минусы
Единый порог	Один порог для всего проекта (например, 80%)	Простота	Не учитывает критичность модулей
Порог по модулям	Разные пороги для разных модулей	Гибкость	Сложность настройки
Порог по веткам	Более высокий порог для main/develop	Безопасность	Может мешать быстрой разработке
Инкрементальный	Требовать покрытие только для нового кода	Справедливость	Сложность реализации

Пример настройки порога в pytest-cov (bash):

Минимальное покрытие 80%, иначе exit code != 0

pytest --cov=src --cov-fail-under=80

Пример настройки порога в Jest:

//json

```
// jest.config.js
module.exports = {
  coverageThreshold: {
    global: {
      lines: 80,
      branches: 70,
      functions: 80,
      statements: 80
    },
    "./src/critical/*.js": {
      lines: 95 // более строгий порог для критических модулей
    }
  }
};
```

5. Интеграция с CI/CD и срыв пайплайна

Механизм срыва пайплайна (fail pipeline) (рисунок 4):

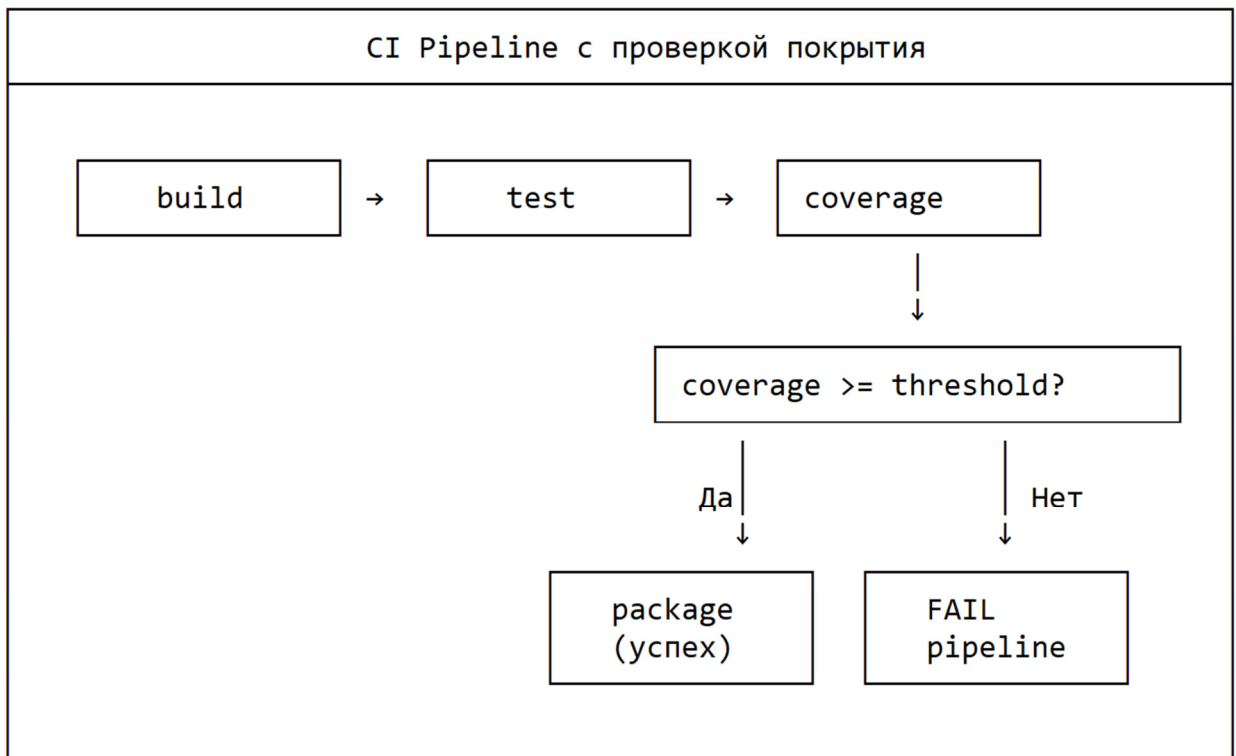


Рисунок 4

Коды выхода (exit codes):

- 0 - успех, порог достигнут
- 1 (или ненулевой) - ошибка, порог не достигнут

CI-платформа интерпретирует ненулевой код выхода как провал задания (job failure).

6. Лучшие практики анализа покрытия

1. Стремиться к 80% покрытия строк как к реалистичному минимуму.
2. Отслеживать динамику - покрытие не должно снижаться со временем.
3. Фокусироваться на критических модулях (ядро бизнес-логики).
4. Не гнаться за 100% - оставшиеся 20% часто требуют непропорционально больших усилий.
5. Использовать разные пороги для разных типов кода (утилиты, UI, бизнес-логика).
6. Интегрировать покрытие в code review - проверять, что новый код покрыт тестами.

Примеры практического выполнения

1. Пример для Python (pytest-cov)

Файл .gitlab-ci.yml:

```
stages:
  - test

variables:
  PIP_CACHE_DIR: "$CI_PROJECT_DIR/.cache/pip"
  MIN_COVERAGE: 80

cache:
  paths:
    - .cache/pip/

coverage-test:
  stage: test
  image: python:3.11-slim
  before_script:
    - pip install pytest pytest-cov
    - pip install -r requirements.txt
  script:
    - pytest tests/ --cov=src --cov-report=html --cov-report=xml --cov-report=term --cov-fail-under=$MIN_COVERAGE
  artifacts:
    paths:
      - htmlcov/
      - coverage.xml
    reports:
      coverage_report:
        coverage_format: cobertura
        path: coverage.xml
    expire_in: 1 week
  coverage: '/TOTAL\s+\d+\s+\d+\s+(\d+%)/'
```

Локальный запуск:

```
# Установка
pip install pytest pytest-cov

# Запуск с порогом 80%
pytest tests/ --cov=src --cov-fail-under=80

# Результат при успехе:
```

```
# ----- coverage: platform linux, python 3.11.0 -----
# Name           Stmts   Miss  Cover
# -----
# src/calculator.py    15     0   100%
# src/utils.py         23     2    91%
# src/db.py            42     5    88%
# -----
# TOTAL                80     7    91%
# Required minimum coverage: 80% -> PASSED

# Результат при неудаче:
# TOTAL                80    20    75%
# Required minimum coverage: 80% -> FAILED
```

3.2. Пример для JavaScript (Jest)

Файл `jest.config.js`:

```
module.exports = {
  collectCoverageFrom: [
    'src/**/*.js',
    '!src/**/*.test.js',
    '!src/index.js'
  ],
  coverageThreshold: {
    global: {
      lines: 80,
      branches: 70,
      functions: 80,
      statements: 80
    }
  },
  coverageReporters: ['html', 'lcov', 'text', 'cobertura']
};
```

Файл `.github/workflows/test.yml`:

```
name: Test with Coverage

on: [push, pull_request]

jobs:
  coverage:
    runs-on: ubuntu-latest
    steps:
```



```

- uses: actions/checkout@v4
- uses: actions/setup-node@v4
  with:
    node-version: '20'
    cache: 'npm'
- run: npm ci
- run: npm test -- --coverage --coverageReporters=lcov --
coverageReporters=text
- name: Upload coverage report
  uses: actions/upload-artifact@v4
  with:
    name: coverage-report
    path: coverage/

```

3. Пример для Java (Maven + JaCoCo)

Файл pom.xml (фрагмент):

```

<plugin>
<groupId>org.jacoco</groupId>
<artifactId>jacoco-maven-plugin</artifactId>
<version>0.8.11</version>
<executions>
<execution>
<goals>
<goal>prepare-agent</goal>
</goals>
</execution>
<execution>
<id>report</id>
<phase>test</phase>
<goals>
<goal>report</goal>
</goals>
</execution>
<execution>
<id>check</id>
<goals>
<goal>check</goal>
</goals>
<configuration>
<rules>
<rule>
<element>PACKAGE</element>
<limits>
<limit>
<counter>LINE</counter>
<value>COVEREDRATIO</value>
<minimum>0.80</minimum>
</limit>
</limits>
</rule>
</rules>
</configuration>
</execution>

```

```
</executions>
</plugin>
```

4. Пример для Go

Файл .gitlab-ci.yml:

```
coverage-test:
  stage: test
  image: golang:1.21
  script:
    - go test -coverprofile=coverage.out ./...
    - go tool cover -html=coverage.out -o coverage.html
    - go tool cover -func=coverage.out
    # Проверка порога (80%)
    - |
      COVERAGE=$(go tool cover -func=coverage.out | grep total | awk '{print $3}'
| sed 's/%//')
      if (( $(echo "$COVERAGE < 80" | bc -l) )); then
        echo "Coverage $COVERAGE% is below threshold 80%"
        exit 1
      fi
  artifacts:
    paths:
      - coverage.html
      - coverage.out
```

Методические указания выполнения практической части задачи

Задачи работы

1. Интеграция инструмента измерения покрытия:

- Выбрать и установить инструмент анализа покрытия для выбранного языка программирования.
- Настроить генерацию отчета о покрытии в процессе выполнения тестов.

2. Генерация и визуализация отчета:

- Создать отчет о покрытии в формате HTML для визуального анализа.

- Создать отчет в машиночитаемом формате (XML, JSON, lcov) для CI-обработки.

3. Настройка порогового значения:

- Установить целевой порог покрытия (например, 80%).
- Настроить автоматическую проверку достижения порога.

4. Автоматизация контроля качества:

- Реализовать срыв (fail) пайплайна при недостижении порога покрытия.

- Настроить публикацию отчета о покрытии как артефакта.

5. Анализ непокрытых участков кода:

- Выявить строки и функции, не покрытые тестами.
- Предложить пути улучшения покрытия.

6. Оформление отчёта:

- Зафиксировать конфигурацию пайплайна с анализом покрытия.
- Представить скриншоты отчета и демонстрацию срыва пайплайна.

Варианты заданий

Базовый уровень

Общее требование: Настроить инструмент измерения покрытия кода. Сгенерировать HTML-отчет о покрытии. Установить порог 70% для строк. Пайплайн должен падать при недостижении порога.

№	Язык	Инструмент	Формат отчета
1	Python	pytest-cov	HTML, term
2	Python	coverage.py	HTML, XML
3	Python	pytest-cov	HTML, JSON
4	JavaScript	Jest	HTML, lcov
5	JavaScript	Vitest	HTML, JSON
6	JavaScript	Mocha + nyc	HTML, lcov

7	TypeScript	Jest	HTML, lcov
8	Go	go test -cover	HTML
9	Go	gocov	HTML, JSON
10	Java	JaCoCo	HTML, XML
11	Java	Cobertura	HTML, XML
12	C (.NET)	coverlet	HTML, Cobertura
13	Ruby	SimpleCov	HTML, JSON
14	PHP	PHPUnit + Xdebug	HTML, XML
15	Rust	tarpaulin	HTML

2. Средний уровень

Общее требование: Настроить пороги по нескольким метрикам (lines, branches, functions). Сгенерировать отчет в машиночитаемом формате (XML/LCov). Настроить публикацию отчета в CI с визуализацией тренда покрытия.

№	Язык	Метрики порога	Дополнительное требование
1	Python	lines (80%), branches (70%)	Интеграция с GitLab coverage badge
2	Python	lines (85%), functions (80%)	Настройка исключения тестов из отчета
3	Python	lines (75%), branches (65%)	Генерация diff-отчета (только измененные файлы)
4	JavaScript	lines (80%), branches (75%), functions (80%)	Интеграция с Codecov
5	JavaScript	lines (85%), statements (85%)	Настройка разных порогов для разных

			директорий
6	JavaScript	lines (70%), branches (70%)	Использование istanbul/nyc
7	Go	lines (80%), functions (80%)	Генерация coverage.out и HTML
8	Go	lines (85%)	Интеграция с Coveralls
9	Java	lines (80%), branches (70%)	Настройка JaCoCo check goal
10	Java	lines (85%), complexity (70%)	Генерация XML для SonarQube
11	C	lines (80%), branches (75%)	Использование coverlet + ReportGenerator
12	C	lines (85%)	Интеграция с Azure Pipelines
13	Ruby	lines (80%), branches (70%)	Настройка SimpleCov с фильтрами
14	PHP	lines (75%), methods (80%)	Интеграция с PHPUnit coverage
15	Rust	lines (80%), functions (85%)	Использование tarpaulin --out xml

4.3. Продвинутый уровень

Общее требование: Настроить разные пороги покрытия для разных модулей/пакетов. Реализовать инкрементальный анализ покрытия (требовать покрытие только для нового/измененного кода). Настроить автоматическое создание issue при падении покрытия.

№	Язык	Сценарий	Дополнительные требования
1	Python	core module (95%), utils	Настройка .coveragerc с

		(80%), legacy (60%)	разными порогами
2	Python	Инкрементальное покрытие (diff-cover)	Интеграция diff-cover в CI
3	Python	Покрывтие + мутационное тестирование	Добавить cosmic-ray или mutmut
4	JavaScript	core (95%), ui (70%), api (85%)	Настройка покрытия в Jest с разными порогами
5	JavaScript	Инкрементальное покрытие	Использование jest-coverage-ratchet
6	TypeScript	strict (90%), permissive (75%)	Настройка разных порогов для .ts и .spec.ts
7	Go	internal (90%), pkg (80%), cmd (70%)	Настройка профилей покрытия для разных пакетов
8	Go	Инкрементальное покрытие + гейты	Использование goveralls с diff
9	Java	core (95%), service (85%), web (70%)	Настройка JaCoCo с разными rules
10	Java	Инкрементальное покрытие	Интеграция с SonarQube и Quality Gate
11	C	Core (95%), Infrastructure (80%), UI (60%)	Настройка coverlet с фильтрацией
12	Специфика БПЛА	Пайплайн для PX4 прошивки	Анализ покрытия кода автопилота
13	ROS 2	Покрывтие для ROS 2 нод	Интеграция с lscov и gscovr
14	Многоязычный	Python + JavaScript + Go	Единый отчет о покрытии (сборка из разных инструментов)

15	Комплексный	Покрывтие + автоматическое создание issue	GitHub Action: при падении порога → создать issue с отчетом

Контрольные вопросы

1. Что такое покрытие кода и какие метрики покрытия существуют?
2. В чем отличие покрытия строк (lines) от покрытия ветвей (branches)?
3. Почему 100% покрытие кода не гарантирует отсутствие ошибок?
4. Какой уровень покрытия считается «хорошим» для production-проекта?
5. Как настроить порог покрытия в pytest-cov?
6. Как интерпретировать HTML-отчет coverage.py (красные/зеленые строки)?
7. Как заставить пайплайн упасть, если покрытие ниже порога?
8. Как исключить из отчета о покрытии тестовые файлы или сгенерированный код?
9. Как настроить разные пороги покрытия для разных модулей в Jest?
10. Как настроить JaCoCo check goal в Maven?
11. Как интегрировать отчет о покрытии в GitLab CI с отображением badge?
12. Какие инструменты позволяют отслеживать динамику покрытия во времени?
13. Что такое инкрементальный анализ покрытия и зачем он нужен?
14. Какие проблемы могут возникнуть при измерении покрытия в асинхронном коде (JS)?
15. Как протестировать код, который работает с внешними API, для достижения высокого покрытия?

Требования к отчету

Отчет должен содержать следующие разделы:

1. Титульный лист с указанием ФИО студента, группы, варианта.
2. Цель и задачи работы.
3. Теоретическая часть:
 - Определение покрытия кода и его метрик.
 - Описание выбранного инструмента анализа покрытия.
 - Обоснование выбранного порогового значения.
4. Практическая часть:
 - Листинг конфигурации пайплайна с настройками покрытия.
 - Листинг конфигурации инструмента (если отдельный файл).
 - Листинг тестов (примеры).
5. Результаты выполнения:
 - Скриншоты успешного выполнения пайплайна (покрытие $\geq$ порога).
 - Скриншот срыва пайплайна (покрытие $<$ порога).
 - Скриншоты HTML-отчета с выделением непокрытых строк.
 - Скриншот артефактов (отчеты о покрытии).
6. Анализ покрытия:
 - Таблица покрытия по модулям/файлам.
 - Выявление и описание непокрытых участков кода.
 - Предложения по улучшению покрытия (какие тесты добавить).
7. Ответы на контрольные вопросы (3-5 вопросов).
8. Вывод - достигнута ли цель, какие сложности возникли, как можно улучшить процесс.

Критерии оценивания

Критерий	Базовый уровень	Средний уровень	Продвинутый уровень
Настроен инструмент измерения	2 балла	2 балла	2 балла

покрытия			
Сгенерирован HTML-отчет о покрытии	2 балла	2 балла	2 балла
Установлен порог покрытия ($\geq 70\%$)	2 балла	2 балла	2 балла
Пайплайн падает при недостижении порога	1 балл	2 балла	2 балла
Настроены пороги по нескольким метрикам	0 баллов	2 балла	2 балла
Сгенерирован машиночитаемый отчет (XML/LCov)	0 баллов	1 балл	2 балла
Настроены разные пороги для разных модулей	0 баллов	0 баллов	2 балла
Реализован инкрементальный анализ покрытия	0 баллов	0 баллов	2 балла
Качество отчета и ответы на вопросы	2 балла	2 балла	2 балла

Шкала перевода баллов в оценку:

Оценка	Базовый уровень (макс. 9)	Средний уровень (макс. 13)	Продвинутый уровень (макс. 18)
Отлично (5)	8-9 баллов	12-13 баллов	16-18 баллов
Хорошо (4)	6-7 баллов	10-11 баллов	13-15 баллов
Удовлетворительно (3)	4-5 баллов	7-9 баллов	10-12 баллов
Неудовлетворительно (2)	менее 4 баллов	менее 7 баллов	менее 10 баллов

Дополнительные критерии (штрафы):

- Отсутствие демонстрации срыва пайплайна - минус 1 балл.
- Не указаны непокрытые строки в анализе - минус 1 балл.
- Покрытие <70% (при базовом уровне) - работа не засчитывается.

Бонусные баллы (макс. +3):

- Покрытие >90% - +1 балл.
- Настроен coverage badge в README - +1 балл.
- Интеграция с Codecov/Coveralls - +1 балл.

Лабораторная работа №6. Статический анализ безопасности (SAST)

Цель работы: Сформировать практические навыки интеграции инструментов статического анализа безопасности (SAST) в CI/CD-пайплайн, настройки правил безопасности, анализа отчетов и исправления типовых уязвимостей (SQL-инъекции, межсайтовый скриптинг и др.) на ранних этапах разработки .

Теоретическое обоснование

1. Концепция SAST и DevSecOps

Static Application Security Testing (SAST) - это метод тестирования безопасности, при котором анализ исходного кода выполняется без его фактического запуска (статически) . SAST-инструменты построчно сканируют код на наличие опасных паттернов: хранение секретов, невалидированный ввод, небезопасные вызовы функций и т.д.

Ключевые принципы DevSecOps в контексте SAST:

Принцип	Описание
Shift Left	Сдвиг безопасности влево — перенос проверок на ранние этапы разработки
Security as Code	Конфигурация безопасности хранится в репозитории и версионизируется
Automation First	Автоматизация проверок без участия человека
Fail Fast	Обнаружение проблем на самых ранних этапах

Почему SAST внедряется именно в CI/CD (рисунок 1):

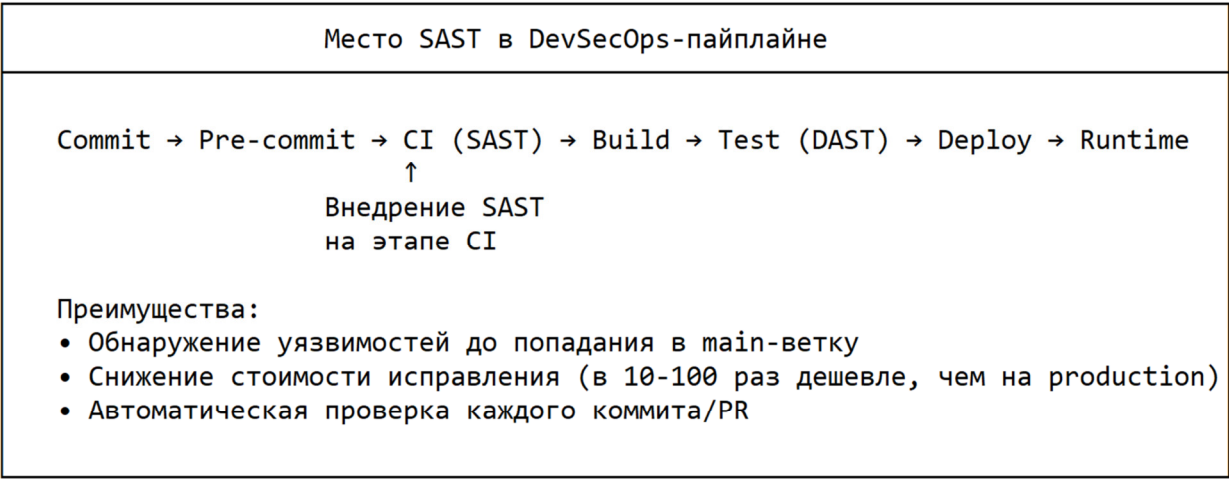


Рисунок 1

2. Сравнение популярных SAST-инструментов

Инструмент	Языки	Формат отчета	Интеграция	Особенности
Semgrep	Python, JS, Go, Java, Ruby, etc.	JSON, SARIF	GitLab, GH Actions	Быстрый, правила на простом языке, OSS-версия
Bandit	Python	JSON, JUnit XML,	Любая	Простой, специализирован

		TXT		для Python
SonarQube	20+ языков	XML, JSON	GitLab, GH Actions, Jenkins	Quality Gate, тренды, интеграция с IDE
CodeQL	Python, JS, Go, Java, C++, C	SARIF	GitHub (бесплатно)	Глубокий анализ поточков данных
GitLab SAST	Множество	JUnit, JSON	GitLab (встроен)	Бесплатный для GitLab, адаптирован под среду
Checkmarx	20+ языков	XML, HTML	Корпоративный	Коммерческий, enterprise- функции

Рекомендации по выбору:

- Для Python-проектов: Bandit (базовый) или Semgrep (продвинутый).
- Для многоязычных проектов: Semgrep или SonarQube.
- Для GitHub-проектов: CodeQL (бесплатно для публичных репозиторий).
- Для GitLab-проектов: GitLab SAST (встроенный) .

3. Типовые уязвимости, обнаруживаемые SAST

Тип уязвимости	Описание	Пример кода (уязвимый)	Исправление
SQL Injection	Прямая конкатенация пользовательск	query = f"SELECT FROM users WHERE id = {user_id}"	Параметризованн ые запросы / ORM

	ого ввода в SQL-запрос		
XSS (Cross-Site Scripting)	Вывод непроверенных данных в HTML	echo " \$user_input "	Экранирование вывода / CSP
Command Injection	Выполнение команд ОС с пользовательским вводом	os.system(f"ping {user_input}")	Валидация ввода / shlex.quote
Path Traversal	Доступ к файлам вне разрешенной директории	open(f"/var/www/{user_file}")	Нормализация пути / allowlist
Hardcoded Secrets	Пароли/токены в коде	API_KEY = "sk-1234567890"	Переменные окружения / secrets manager
Insecure Deserialization	Десериализация недоверенных данных	pickle.loads(data)	JSON / альтернативные форматы

4. Пример уязвимого кода на Python

```
# vulnerable_app.py — пример кода с преднамеренными уязвимостями

import sqlite3
import os
import pickle

# Уязвимость 1: Hardcoded secret (должно быть в переменной окружения)
API_SECRET = "super_secret_key_12345" # уязвимость

def get_user_sql_injection(user_id):
    # Уязвимость 2: SQL Injection
    conn = sqlite3.connect('users.db')
    cursor = conn.cursor()
    # Прямая конкатенация — опасно!
    query = f"SELECT * FROM users WHERE id = {user_id}" # SQL Injection
```

```

cursor.execute(query)
return cursor.fetchall()

def xss_vulnerable(request):
    # Уязвимость 3: XSS (в веб-фреймворке)
    user_name = request.GET.get('name', '')
    # Вывод без экранирования
    return f"<div>Hello, {user_name}!</div>" # XSS

def command_injection(host):
    # Уязвимость 4: Command Injection
    os.system(f"ping -c 1 {host}") # Command Injection

def insecure_deserialization(data):
    # Уязвимость 5: Insecure deserialization
    return pickle.loads(data) # RCE-уязвимость

```

5. Архитектура интеграции SAST в CI/CD (рисунок 2)

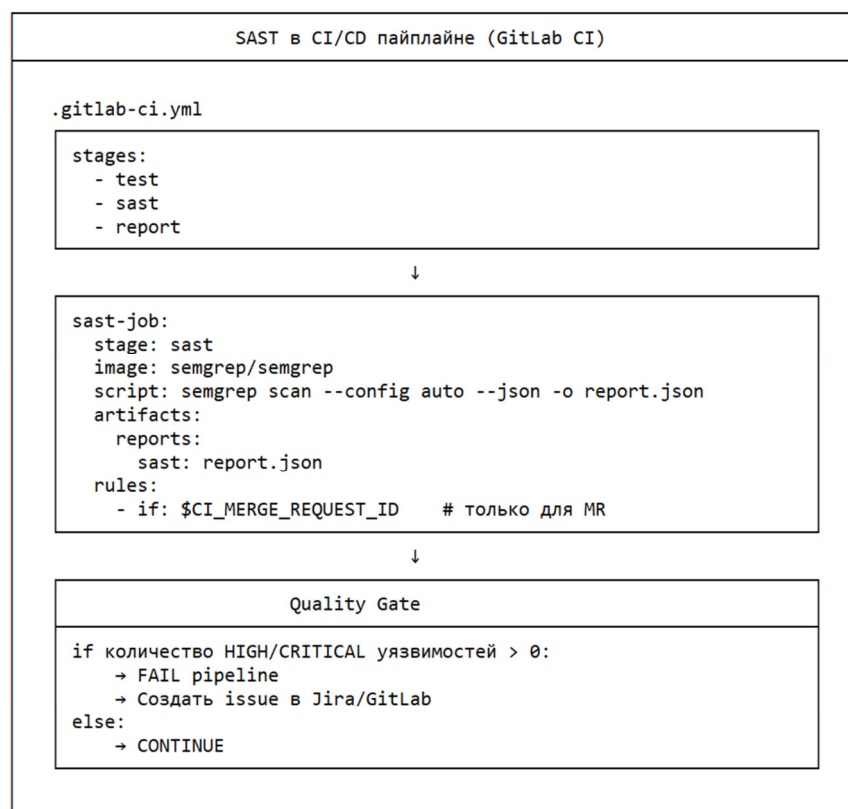


Рисунок 2

Ключевые элементы интеграции :

- Добавление SAST как обязательного шага в CI/CD-конвейер
- Определение триггеров запуска (после каждого коммита / перед развертыванием)

- Установка пороговых значений для уязвимостей (Quality Gate)
- Связывание с дефект-трекером для автоматического создания задач

6. Quality Gate и уровни критичности

Типичная классификация уязвимостей по критичности:

Уровень	Описание	Действие в пайплайне
CRITICAL	Удаленное выполнение кода (RCE), обход аутентификации	FAIL - блокировка
HIGH	SQL Injection, XSS с возможностью компрометации	FAIL - блокировка
MEDIUM	Информационные утечки, CSRF	WARN - логирование, не блокирует
LOW	Рекомендации по улучшению, code smells	INFO - только информация

Пример практического выполнения

1. Пример для Python (GitLab CI + Semgrep)

Файл .gitlab-ci.yml:

```
stages:
  - test
  - sast
  - quality-gate

variables:
  SAST_REPORT: gl-sast-report.json

# Базовый этап тестирования
unit-test:
```

```

stage: test
image: python:3.11-slim
script:
  - pip install pytest
  - pytest tests/ --junitxml=report.xml
artifacts:
  reports:
    junit: report.xml

# SAST-анализ с Semgrep
semgrep-sast:
  stage: sast
  image: semgrep/semgrep
  script:
    - semgrep scan --config auto --json --output $SAST_REPORT
    - semgrep scan --config auto --text # ВЫВОД В КОНСОЛЬ
  artifacts:
    paths:
      - $SAST_REPORT
    reports:
      sast: $SAST_REPORT
    expire_in: 1 week
  rules:
    - if: $CI_MERGE_REQUEST_ID
    - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH
  allow_failure: true # не ломаем пайплайн, только собираем отчет

# Quality Gate — проверка порогов
quality-gate:
  stage: quality-gate
  image: python:3.11-slim
  script:
    - pip install jq
    - |
      # Подсчет критических уязвимостей
      CRITICAL_COUNT=$(jq '.results | map(select(.extra.severity == "ERROR")) |
length' $SAST_REPORT)
      HIGH_COUNT=$(jq '.results | map(select(.extra.severity == "WARNING")) |
length' $SAST_REPORT)
      echo "CRITICAL findings: $CRITICAL_COUNT"
      echo "HIGH findings: $HIGH_COUNT"
      if [ "$CRITICAL_COUNT" -gt 0 ] || [ "$HIGH_COUNT" -gt 0 ]; then
        echo " Quality Gate failed: Found $CRITICAL_COUNT CRITICAL and
$HIGH_COUNT HIGH vulnerabilities"
        exit 1
      else
        echo " Quality Gate passed"
      fi
  dependencies:
    - semgrep-sast
  rules:

```



```
- if: $CI_MERGE_REQUEST_ID
- if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH
```

2. Пример для Python (GitHub Actions + Bandit + Semgrep)

Файл .github/workflows/sast.yml:

```
name: SAST Security Scan

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]
  schedule:
    - cron: '0 0 * * 0' # еженедельное сканирование

jobs:
  bandit-scan:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Set up Python
        uses: actions/setup-python@v5
        with:
          python-version: '3.11'

      - name: Install Bandit
        run: pip install bandit

      - name: Run Bandit scan
        run: bandit -r ./src -f json -o bandit-report.json -lll # -lll: только
HIGH и CRITICAL

      - name: Upload Bandit report
        uses: actions/upload-artifact@v4
        with:
          name: bandit-report
          path: bandit-report.json

      - name: Check for HIGH/CRITICAL issues
        run: |
          HIGH_CRITICAL=$(jq '[.results[] | select(.issue_severity == "HIGH" or
.issue_severity == "CRITICAL")] | length' bandit-report.json)
          if [ "$HIGH_CRITICAL" -gt 0 ]; then
            echo " Found $HIGH_CRITICAL HIGH/CRITICAL vulnerabilities!"
```

```

        exit 1
    fi

semgrep-scan:
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v4

    - name: Semgrep scan
      uses: returntocorp/semgrep-action@v1
      with:
        config: auto
        output: semgrep-report.json

    - name: Upload Semgrep report
      uses: actions/upload-artifact@v4
      with:
        name: semgrep-report
        path: semgrep-report.json

```

3. Пример для Python (Bandit - специализированный инструмент)

Локальный запуск Bandit:

```
//bash
```

Установка

```
pip install bandit
```

Базовое сканирование

```
bandit -r ./src
```

Сканирование с детализацией HIGH и CRITICAL

```
bandit -r ./src -lll
```

Генерация JSON-отчета

```
bandit -r ./src -f json -o bandit-report.json -lll
```

Сканирование с игнорированием определенных проверок

```
bandit -r ./src -s B108,B701  исключить проверки hardcoded и exec
```

Пример вывода Bandit:

```
[main] INFO  profile include tests: None
[main] INFO  profile exclude tests: None
[main] INFO  cli include tests: None
[main] INFO  cli exclude tests: None
[main] INFO  running on Python 3.11.0
Run started:2025-01-15 10:30:00.123456
```

Test results:

```
>> Issue: [B608:hardcoded_sql_expressions] Possible SQL injection vector
Severity: MEDIUM Confidence: HIGH
Location: ./src/db.py:12
More Info:
```

https://bandit.readthedocs.io/en/latest/plugins/b608_hardcoded_sql_expressions.html

```
12      query = f"SELECT * FROM users WHERE id = {user_id}"
```

```
>> Issue: [B105:hardcoded_password_string] Possible hardcoded password
Severity: LOW Confidence: MEDIUM
Location: ./src/config.py:5
More Info:
```

https://bandit.readthedocs.io/en/latest/plugins/b105_hardcoded_password_string.html

```
5      API_SECRET = "super_secret_key_12345"
```

| Ran 45 tests in 0.234s

| Issues found: 2 (MEDIUM: 1, LOW: 1)

4. Пример для JavaScript (Semgrep)

Файл `.github/workflows/semgrep.yml` :

```
name: Semgrep

on:
  pull_request: {}
  push:
    branches:
      - main
      - master

permissions:
  contents: read

jobs:
  semgrep:
    name: semgrep/ci
    runs-on: ubuntu-latest
    container:
      image: semgrep/semgrep
    if: (github.actor != 'dependabot[bot]')
    steps:
      - uses: actions/checkout@v4
      - run: semgrep ci
      env:
        SEMGREP_APP_TOKEN: ${ secrets.SEMGREP_APP_TOKEN }
```

5. Пример исправления уязвимостей

Исходный уязвимый код (SQL Injection):

python

```
def get_user(user_id):
    query = f"SELECT * FROM users WHERE id = {user_id}"
    cursor.execute(query)
    return cursor.fetchone()
```

Исправленный код (параметризованный запрос):

```
def get_user(user_id):
```

```
query = "SELECT * FROM users WHERE id = ?"
cursor.execute(query, (user_id,))
return cursor.fetchone()
```

Исходный уязвимый код (XSS):

```
def render_comment(comment_text):
    return f"<div class='comment'>{comment_text}</div>"
```

Исправленный код (экранирование):

```
from html import escape

def render_comment(comment_text):
    safe_text = escape(comment_text)
    return f"<div class='comment'>{safe_text}</div>"
```

Методические указания выполнения практической части работы

Задачи работы

1. Выбор и настройка SAST-инструмента:

- Выбрать инструмент статического анализа для выбранного языка программирования (Semgrep, Bandit, SonarQube, CodeQL).
- Настроить конфигурацию правил безопасности.
- Интегрировать инструмент в CI/CD-пайплайн.

2. Автоматизация проверок безопасности:

- Добавить этап SAST как обязательный шаг пайплайна .
- Настроить триггеры запуска (pull request, push в main).
- Настроить генерацию отчетов в машиночитаемом формате (JSON, SARIF, JUnit XML).

3. Анализ и интерпретация результатов:

- Запустить SAST-анализ на тестовом проекте с преднамеренными уязвимостями.

- Проанализировать отчет: классифицировать найденные уязвимости по типу и критичности.

- Интерпретировать рекомендации по исправлению.

4. Исправление уязвимостей:

- Внести исправления в код для устранения выявленных проблем.

- Повторно запустить анализ для верификации исправлений.

5. Настройка Quality Gate:

- Установить пороговые значения для критических уязвимостей .

- Настроить срыв пайплайна при обнаружении HIGH/CRITICAL уязвимостей .

- Настроить автоматическое создание issue/задач для исправления .

6. Оформление отчёта:

- Зафиксировать конфигурацию пайплайна с SAST-этапом.

- Представить скриншоты отчета о уязвимостях.

- Продемонстрировать исправления кода.

Варианты заданий

1. Базовый уровень

Общее требование: Интегрировать один SAST-инструмент в пайплайн. Настроить генерацию отчета. Продемонстрировать обнаружение хотя бы одного типа уязвимостей.

№	Язык	SAST-инструмент	Целевые уязвимости
1	Python	Bandit	SQL Injection, hardcoded secrets
2	Python	Semgrep	XSS, command injection
3	Python	Bandit	Insecure deserialization, eval

			usage
4	Python	Semgrep	Path traversal, unsafe file operations
5	JavaScript	Semgrep	DOM-based XSS, eval
6	JavaScript	ESLint (security plugin)	Unsafe innerHTML, no eval
7	TypeScript	Semgrep	Type confusion, unsafe any
8	Go	Semgrep	SQL Injection, unsafe pointer
9	Go	gosec	Hardcoded secrets, unsafe blocks
10	Java	Semgrep	SQL Injection (JDBC), XSS
11	Java	SpotBugs	Command injection, path traversal
12	C	Semgrep	SQL Injection, XSS
13	Ruby	Semgrep	Command injection, unsafe send
14	PHP	Semgrep	SQL Injection, XSS, file inclusion
15	Rust	Semgrep	Unsafe blocks, panic risks

2. Средний уровень

Общее требование: Интегрировать SAST-инструмент с Quality Gate (пороговые значения). Настроить публикацию отчета в SARIF/JUnit формате для визуализации в Security Dashboard. Продемонстрировать исправление 2+ типов уязвимостей.

№	Язык	Инструмент	Quality Gate условие
1	Python	Semgrep + Bandit	0 HIGH/CRITICAL, <5 MEDIUM
2	Python	SonarQube	Quality Gate: Reliability A, Security A
3	Python	GitLab SAST	Использование встроенного SAST

4	JavaScript	Semgrep + ESLint	0 CRITICAL, <3 HIGH
5	JavaScript	CodeQL (GitHub)	Автоматический Quality Gate от GitHub
6	TypeScript	Semgrep	Порог по типам: injection, xss
7	Go	Semgrep + gosec	0 CRITICAL, <2 HIGH
8	Go	CodeQL	Интеграция с GitHub Security tab
9	Java	SonarQube	Quality Gate с пользовательскими правилами
10	Java	Semgrep	Разные пороги для разных директорий
11	C	Semgrep	Интеграция с Azure DevOps
12	Python + Docker	Trivy (fs scan) + Bandit	Сканирование кода и Dockerfile
13	Многоязычный	Semgrep	Единый отчет для Python + JS
14	Python + Terraform	Semgrep + Checkov	SAST для кода и IaC
15	Python + CI/CD	Semgrep + GitLab SAST	Сравнение эффективности двух инструментов

3. Продвинутый уровень

Общее требование: Настроить комплексную SAST-стратегию: pre-commit хуки + CI-сканирование + Quality Gate с блокировкой. Интегрировать с дефект-трекером (автосоздание issue). Реализовать инкрементальное сканирование (только измененные файлы).

№	Язык	Сценарий	Дополнительные требования
1	Python	Pre-commit (Bandit) + CI (Semgrep) +	Автосоздание issue в GitLab

		Gate	
2	Python	Инкрементальное сканирование	Только измененные файлы в MR
3	Python	Кастомизация правил Semgrep	Написание 3+ собственных правил
4	JavaScript	GitHub Advanced Security	Использование CodeQL + Secret Scanning
5	JavaScript	Pre-commit + CI + Gate	Интеграция с Jira
6	TypeScript	Semgrep + SonarQube	Сравнение результатов двух инструментов
7	Go	Инкрементальное сканирование	Использование git diff для оптимизации
8	Go	Кастомные правила Semgrep	Поиск специфичных для проекта паттернов
9	Java	SonarQube + Checkmarx (OSS)	Сравнение enterprise и OSS инструментов
10	C	Microsoft Security Code Analysis	Интеграция с Azure DevOps Security
11	Специфика БПЛА	PX4/ArduPilot прошивка	SAST для C++ кода автопилота
12	ROS 2	Semgrep для ROS-нод	Поиск специфичных уязвимостей ROS
13	Комплексный	SAST + SCA + Secrets	Комбинированный security-пайплайн
14	Security-hardened	SAST с блокировкой merge	Нельзя смержить MR с HIGH уязвимостями
15	AI-assisted	SAST + LLM для рекомендаций	Автоматическое предложение исправлений

Контрольные вопросы

1. Что такое SAST и чем он отличается от DAST?
2. Почему SAST внедряется на этапе CI, а не после развертывания?
3. Какие типы уязвимостей может обнаружить статический анализатор?
4. Какие типы уязвимостей НЕ может обнаружить SAST (ложные негативы)?
5. Что такое Quality Gate в контексте SAST и как его настроить?
6. В чем разница между Semgrep, Bandit и SonarQube?
7. Как интерпретировать отчет SAST (severity, confidence, location)?
8. Как настроить пайплайн на срыв (fail) при обнаружении HIGH-уязвимостей?
9. Как исключить из сканирования тестовые файлы или сгенерированный код?
10. Как интегрировать SAST-отчет в Security Dashboard GitLab/GitHub?
11. Что такое «ложное срабатывание» (false positive) и как с ним работать?
12. Как написать кастомное правило для Semgrep?
13. Как настроить инкрементальное сканирование (только измененные файлы)?
14. Какие секреты (пароли, токены) может найти инструмент статического анализа?
15. Какие метрики можно отслеживать для оценки эффективности SAST в команде?

Требования к отчету

Отчет должен содержать следующие разделы:

1. Титульный лист с указанием ФИО студента, группы, варианта.

2. Цель и задачи работы.
3. Теоретическая часть:
 - Определение SAST и его место в DevSecOps.
 - Описание выбранного инструмента и его возможностей.
 - Обоснование выбора правил безопасности.
4. Практическая часть:
 - Листинг конфигурации пайплайна с SAST-этапом.
 - Листинг уязвимого кода (тестового приложения).
 - Листинг исправленного кода (с комментариями изменений).
5. Результаты выполнения:
 - Скриншоты выполнения SAST-этапа в CI.
 - Скриншоты отчета о найденных уязвимостях.
 - Скриншоты Quality Gate (проход/непроход).
 - Скриншоты артефактов (JSON/SARIF отчеты).
6. Анализ результатов:
 - Таблица найденных уязвимостей (тип, критичность, локация, статус).
 - Описание исправлений для каждой уязвимости.
 - Анализ ложных срабатываний (если есть).
7. Ответы на контрольные вопросы (3-5 вопросов).
8. Вывод - достигнута ли цель, какие сложности возникли, рекомендации по внедрению SAST в команде.

Критерии оценивания

Критерий	Базовый уровень	Средний уровень	Продвинутый уровень
SAST-инструмент интегрирован в пайплайн	2 балла	2 балла	2 балла
Сгенерирован отчет о	2 балла	2 балла	2 балла

уязвимостях			
Обнаружены типовые уязвимости (SQLi, XSS)	2 балла	2 балла	2 балла
Уязвимости исправлены и верифицированы	1 балл	2 балла	2 балла
Настроен Quality Gate (пороги)	0 баллов	2 балла	2 балла
Пайплайн падает при HIGH/CRITICAL уязвимостях	0 баллов	1 балл	2 балла
Настроена публикация в Security Dashboard	0 баллов	1 балл	2 балла
Реализовано инкрементальное сканирование	0 баллов	0 баллов	2 балла
Качество отчета и ответы на вопросы	2 балла	2 балла	2 балла

Шкала перевода баллов в оценку:

Оценка	Базовый уровень (макс. 9)	Средний уровень (макс. 14)	Продвинутый уровень (макс. 18)
Отлично (5)	8-9 баллов	13-14 баллов	16-18 баллов
Хорошо (4)	6-7 баллов	10-12 баллов	13-15 баллов
Удовлетворительно (3)	4-5 баллов	7-9 баллов	10-12 баллов
Неудовлетворительно (2)	менее 4 баллов	менее 7 баллов	менее 10 баллов

Дополнительные критерии (штрафы):

- Отсутствие демонстрации срыва пайплайна - минус 1 балл.
- Не указаны исправления для уязвимостей - минус 1 балл.

- Ложные срабатывания не проанализированы - минус 1 балл.

Бонусные баллы (макс. +3):

- Написано кастомное правило Semgrep - +1 балл.
- Интеграция с дефект-трекером (автосоздание issue) - +1 балл.
- Настроен pre-commit хук для SAST - +1 балл.

Лабораторная работа №7. Анализ зависимостей (SCA)

Цель работы: Сформировать практические навыки автоматической проверки зависимостей проекта на наличие известных уязвимостей с использованием инструментов Software Composition Analysis (SCA), настройки пороговых значений и автоматизации обновления уязвимых компонентов.

Теоретическое обоснование

1. Понятие SCA и его роль в DevSecOps

Software Composition Analysis (SCA) - это практика анализа открытых и сторонних компонентов, используемых в программном проекте, для выявления известных уязвимостей, проблем с лицензированием и устаревших версий зависимостей.

Почему SCA критически важен:

Фактор	Описание
Зависимость от open-source	Современные проекты на 70-90% состоят из сторонних библиотек
Быстрое появление уязвимостей	Новые CVE публикуются ежедневно
Транзитивные зависимости	Уязвимости могут быть скрыты на глубине 3-5 уровней
Supply Chain атаки	Компрометация популярных библиотек (SolarWinds, log4shell)

Автоматизация SCA в CI/CD позволяет:

- Обнаруживать уязвимости до попадания в production
- Снижать стоимость исправления (Shift Left)
- Автоматизировать обновление зависимостей

2.2. Источники данных об уязвимостях

Источник	Описание	Используется в
CVE (Common Vulnerabilities and Exposures)	Публичная база известных уязвимостей	Все SCA-инструменты
CVSS (Common Vulnerability Scoring System)	Система оценки критичности (0-10)	Все SCA-инструменты
NVD (National Vulnerability Database)	Агрегированная база CVE с CVSS	OWASP DC, Snyk
GitHub Advisory Database	Рекомендации по безопасности GitHub	Dependabot
Snyk Vulnerability DB	Собственная база Snyk с приоритизацией	Snyk

Классификация CVSS:

Оценка	Уровень	Действие в пайплайне
9.0 – 10.0	CRITICAL	FAIL - блокировка
7.0 – 8.9	HIGH	FAIL - блокировка
4.0 – 6.9	MEDIUM	WARN - предупреждение
0.1 – 3.9	LOW	INFO - только информация

2.3. Сравнение SCA-инструментов

Инструмент	Лицензия	Языки	Интеграция	Особенности
OWASP Dependency-Check	Open Source (Apache 2.0)	Java, .NET, JS, Python, Ruby	Jenkins, GitHub, GitLab, CLI	Бесплатный, NVD, локальное использование
Snyk	Freemium	20+ языков	GitHub, GitLab, Jenkins, IDE	Developer-first, автопоиск фиксов
Dependabot	Бесплатно (GitHub)	NPM, Maven, pip, Go, etc.	GitHub (встроен)	Авто-PR, интеграция с GitHub Security
JFrog Xray	Коммерческая	Универсальная	JFrog Platform	Глубокий анализ транзитивных зависимостей
Mend (ex-WhiteSource)	Коммерческая	Универсальная	CI/CD, IDE	License compliance

Рекомендации по выбору:

- Для open-source проектов — OWASP Dependency-Check
- Для GitHub-проектов — Dependabot + GitHub Advanced Security
- Для enterprise — Snyk или JFrog Xray

2.4. Типовые уязвимости, обнаруживаемые SCA

Тип уязвимости	Пример	CVE пример	Исправление

RCE (Remote Code Execution)	Log4Shell (log4j)	CVE-2021-44228	Обновление до 2.17.0+
SQL Injection в библиотеке	Sequelize	CVE-2023-22578	Обновление ORM
XSS в фронтенд-библиотеке	jQuery	CVE-2020-11023	Обновление до 3.5.0+
Path Traversal	Express.js	CVE-2022-24999	Обновление
Deserialization RCE	Jackson-databind	CVE-2019-12384	Обновление
DoS (Denial of Service)	Pillow (Python)	CVE-2022-45199	Обновление

2.5. Архитектура интеграции SCA в CI/CD (рисунок 1)

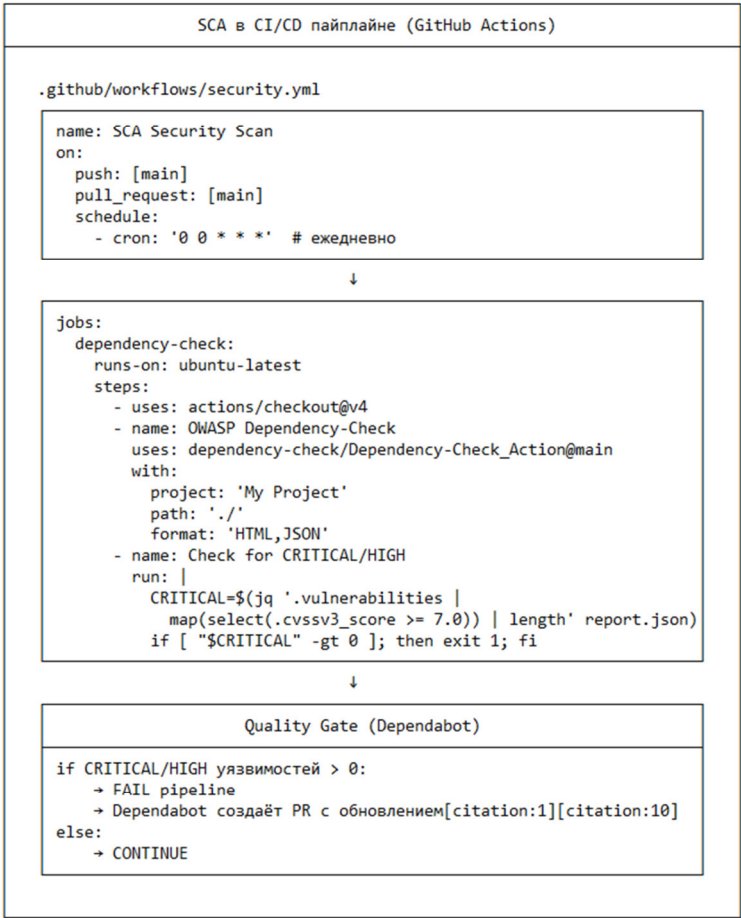


Рисунок 1

2.6. Dependabot: автоматическое обновление зависимостей

Dependabot - встроенный инструмент GitHub для автоматического сканирования и обновления зависимостей.

Файл конфигурации.github/dependabot.yml:

```
version: 2
updates:
  # Python зависимости
  - package-ecosystem: "pip"
    directory: "/"
    schedule:
      interval: "weekly"
      day: "monday"
    open-pull-requests-limit: 10
    reviewers:
      - "team-devops"
    labels:
      - "dependencies"
      - "security"

  # GitHub Actions
  - package-ecosystem: "github-actions"
    directory: "/"
    schedule:
      interval: "weekly"

  # npm зависимости
  - package-ecosystem: "npm"
    directory: "/frontend"
    schedule:
      interval: "daily"
    security-updates-only: true # только security-обновления[citation:1]
```

Режимы работы Dependabot:

- security-updates-only: true - только исправления известных уязвимостей
- Автоматическое создание PR с детальным описанием CVSS
- Интеграция с GitHub Security Dashboard

Примеры практического выполнения

1. Пример для Python (OWASP Dependency-Check + GitHub Actions)

Файл `.github/workflows/sca.yml`:

```
name: SCA Security Scan

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]
  schedule:
    - cron: '0 0 * * 0' # еженедельно

jobs:
  dependency-check:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Setup Python
        uses: actions/setup-python@v5
        with:
          python-version: '3.11'

      - name: Generate requirements.txt (если используем poetry/pipenv)
        run: |
          pip install pip-audit
          pip-audit --requirement requirements.txt --format json --output
audit.json

      - name: Run OWASP Dependency-Check
        uses: dependency-check/Dependency-Check_Action@main
        with:
          project: 'Vulnerable-Python-App'
          path: './'
          format: 'HTML,JSON'
          out: 'reports'
          args: >
            --enableExperimental
            --failOnCVSS 7
            --suppression suppression.xml

      - name: Upload reports
        uses: actions/upload-artifact@v4
        with:
```

```

    name: dependency-check-reports
    path: reports/

- name: Check for HIGH/CRITICAL vulnerabilities
  run: |
    CRITICAL=$(jq '.vulnerabilities | map(select(.cvssv3_score >= 7.0)) |
length' reports/dependency-check-report.json)
    echo "Found $CRITICAL HIGH/CRITICAL vulnerabilities"
    if [ "$CRITICAL" -gt 0 ]; then
        echo "Security gate failed: HIGH/CRITICAL vulnerabilities detected!"
        exit 1
    fi

```

2. Пример для Node.js (Snyk + GitHub Actions)

Файл `.github/workflows/snyk.yml`:

```

name: Snyk Security Scan

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  snyk-scan:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: '20'
          cache: 'npm'

      - name: Install dependencies
        run: npm ci

      - name: Run Snyk to check for vulnerabilities
        uses: snyk/actions/node@master
        env:
          SNYK_TOKEN: ${ secrets.SNYK_TOKEN }
        with:
          args: --severity-threshold=high
          command: test

      - name: Run Snyk monitor (continuous tracking)

```

```
uses: snyk/actions/node@master
env:
  SNYK_TOKEN: ${ secrets.SNYK_TOKEN }
with:
  args: --severity-threshold=high
  command: monitor
```

3. Пример для Go (trivy + GitHub Actions)

Файл .github/workflows/trivy-sca.yml:

```
name: Trivy SCA Scan

on:
  push:
    branches: [ main ]

jobs:
  trivy-scan:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Run Trivy vulnerability scanner
        uses: aquasecurity/trivy-action@master
        with:
          scan-type: 'fs'
          scan-ref: '.'
          format: 'sarif'
          output: 'trivy-results.sarif'
          severity: 'CRITICAL,HIGH'

      - name: Upload Trivy results to GitHub Security
        uses: github/codeql-action/upload-sarif@v3
        with:
          sarif_file: 'trivy-results.sarif'

      - name: Check for vulnerabilities (fail on HIGH/CRITICAL)
        run: |
          trivy fs --severity HIGH,CRITICAL --exit-code 1 --format json .
```

4. Пример настройки Dependabot (автоматические PR)

Файл `.github/dependabot.yml`:

```
version: 2
updates:
  - package-ecosystem: "pip"
    directory: "/"
    schedule:
      interval: "weekly"
    open-pull-requests-limit: 5
    reviewers:
      - "security-team"
    labels:
      - "security"
    commit-message:
      prefix: "🔒"
    groups:
      security-updates:
        applies-to: security-updates
        patterns:
          - "*"

  - package-ecosystem: "npm"
    directory: "/frontend"
    schedule:
      interval: "daily"
    security-updates-only: true
    reviewers:
      - "frontend-team"
```

Пример уязвимого `requirements.txt`

```
//txt

# requirements.txt — с преднамеренными уязвимостями

# уязвимость CVE-2021-44228 (Log4Shell в Java, но аналогия для
Python)

django==3.2.12 # CVE-2023-24580 (Potential denial of service)

# уязвимость CVE-2022-40897 (setuptools)

setuptools==58.0.0

# уязвимость CVE-2022-29217 (JWT)

pyjwt==2.3.0

# уязвимость CVE-2021-33503 (urllib3)
```

urllib3==1.26.0

3.6. Исправление уязвимостей

Исходный уязвимый файл:

django==3.2.12

setuptools==58.0.0

pyjwt==2.3.0

urllib3==1.26.0

Исправленный файл:

django==4.2.7 # фикс CVE-2023-24580

setuptools==68.2.0 # фикс CVE-2022-40897

pyjwt==2.8.0 # фикс CVE-2022-29217

urllib3==2.0.7 # фикс CVE-2021-33503

Методические указания выполнения практической части работы

Задачи работы

1. Выбор и настройка SCA-инструмента:

- Выбрать инструмент для анализа зависимостей (OWASP Dependency-Check, Snyk, Dependabot).
- Настроить конфигурацию для конкретного языка программирования и экосистемы.
- Интегрировать инструмент в CI/CD-пайплайн.

2. Автоматизация проверки зависимостей:

- Добавить этап SCA как обязательный шаг пайплайна.
- Настроить триггеры запуска (pull request, push в main, ежедневное сканирование).

- Настроить генерацию отчетов в машиночитаемом формате (JSON, HTML).

3. Анализ и интерпретация результатов:

- Запустить SCA-анализ на тестовом проекте с преднамеренно устаревшими зависимостями.
- Проанализировать отчет: классифицировать найденные уязвимости по CVSS-оценке и критичности.
- Интерпретировать рекомендации по обновлению.

4. Обновление уязвимых зависимостей:

- Внести изменения в файлы зависимостей (package.json, requirements.txt, go.mod).
- Обновить уязвимые версии до безопасных.
- Повторно запустить анализ для верификации исправлений.

5. Настройка Quality Gate:

- Установить пороговые значения для критических уязвимостей.
- Настроить срыв пайплайна при обнаружении HIGH/CRITICAL уязвимостей.
- Настроить автоматическое создание pull request для обновления зависимостей (Dependabot/Snyk).

6. Оформление отчёта:

- Зафиксировать конфигурацию пайплайна с SCA-этапом.
- Представить скриншоты отчета о уязвимостях.
- Продемонстрировать исправления зависимостей и повторный анализ.

Варианты заданий

Базовый уровень

Общее требование: Интегрировать один SCA-инструмент в пайплайн. Настроить генерацию отчета. Продемонстрировать обнаружение уязвимостей в преднамеренно устаревших зависимостях.

№	Язык	SCA-инструмент	Формат отчета
1	Python	OWASP Dependency-Check	HTML, JSON
2	Python	pip-audit + Safety	JSON, TXT
3	Python	Snyk	HTML, JSON
4	JavaScript/Node.js	npm audit	JSON
5	JavaScript/Node.js	OWASP Dependency-Check	HTML, JSON
6	JavaScript/Node.js	Snyk	HTML, JSON
7	Go	Trivy (filesystem scan)	JSON, SARIF
8	Go	govulncheck	JSON
9	Java (Maven)	OWASP Dependency-Check	HTML, JSON
10	Java (Maven)	Snyk	HTML, JSON
11	C (.NET)	OWASP Dependency-Check	HTML, JSON
12	C (.NET)	dotnet list package --vulnerable	TXT
13	Ruby	bundle-audit	JSON, TXT
14	PHP	Composer Audit	JSON
15	Rust	cargo audit	JSON

Средний уровень

Общее требование: Настроить Quality Gate с пороговыми значениями (fail на HIGH/CRITICAL). Настроить Dependabot или Snyk для автоматических PR обновлений. Интегрировать отчет в Security Dashboard.

№	Язык	Инструмент	Quality Gate условие
1	Python	OWASP DC + Dependabot	0 CRITICAL, <3 HIGH
2	Python	Snyk + Dependabot	Авто-PR для уязвимостей

3	Python	Trivy + GitHub Security	Интеграция с Security Dashboard
4	JavaScript	npm audit + Dependabot	fail на HIGH/CRITICAL
5	JavaScript	Snyk + Dependabot	fail на CVSS > 7.0
6	JavaScript	OWASP DC + GitHub Actions	Авто-PR через Dependabot
7	Go	Trivy + Dependabot	fail на HIGH/CRITICAL
8	Go	govulncheck + GitHub Actions	fail на любую уязвимость
9	Java	OWASP DC + Dependabot	fail на CVSS > 7.0
10	Java	Snyk + GitHub Security	Авто-мониторинг в Dashboard
11	C	OWASP DC + Dependabot (NuGet)	fail на HIGH
12	Python + Docker	Trivy (fs + image)	Сканирование кода и контейнера
13	JavaScript + Docker	Snyk + Trivy	Комбинированный отчёт
14	Многоязычный	Dependabot (multi-ecosystem)	Единая конфигурация
15	Python + CI/CD	pip-audit + GitHub Actions	fail на HIGH

4.3. Продвинутый уровень

Общее требование: Настроить комплексную SCA-стратегию: сканирование на PR, ежедневное сканирование main-ветки, автоматическое создание issue для MEDIUM уязвимостей, suppression-файл для ложных срабатываний, SBOM-генерация.

№	Язык	Сценарий	Дополнительные

			требования
1	Python	OWASP DC + Dependabot + SBOM	Генерация CycloneDX SBOM
2	Python	Snyk + Jira интеграция	Авто-issue при уязвимостях
3	Python	Trivy + GitHub Security	Генерация SARIF для Code Scanning
4	JavaScript	Snyk + Dependabot groups	Группировка обновлений в один PR
5	JavaScript	OWASP DC + suppression	suppression.xml для false positives
6	TypeScript	npm audit + Snyk	Сравнение двух инструментов
7	Go	Trivy + govulncheck	Комбинированное сканирование
8	Go	Dependabot + GitHub Actions	Авто-мерж для patch-версий
9	Java	OWASP DC + Maven	Интеграция в maven lifecycle
10	Java	Snyk + Jenkins pipeline	Корпоративный пайплайн
11	Специфика БПЛА	ROS 2 + Python	Сканирование зависимостей ROS-пакетов
12	Комплексный	SCA + SAST + Secrets	Комбинированный security-пайплайн
13	Supply Chain	SBOM + Dependency Track	Интеграция с OWASP Dependency Track
14	Security-	Auto-merge для security	Авто-мерж при

	hardened	PR	прохождении тестов
15	ML/AI	Python + Hugging Face	Сканирование ML-зависимостей (PyTorch, TF)

Контрольные вопросы:

1. Что такое SCA и чем он отличается от SAST?
2. Какие типы проблем может обнаружить SCA-инструмент?
3. Какие источники данных об уязвимостях используются SCA-инструментами?
4. Что такое CVSS и как интерпретировать CVSS-оценку?
5. Как настроить Dependabot для автоматического обновления зависимостей?
6. В чем отличие Dependabot от Snyk?
7. Как настроить пайплайн на срыв (fail) при обнаружении HIGH-уязвимостей?
8. Что такое транзитивные зависимости и как их анализировать?
9. Как создать suppression-файл для ложных срабатываний в OWASP DC?
10. Как интегрировать SCA-отчет в GitHub Security Dashboard?
11. Что такое SBOM (Software Bill of Materials) и зачем он нужен?
12. Как обновить уязвимую зависимость без нарушения совместимости?
13. Какие риски связаны с автоматическим обновлением зависимостей?
14. Как настроить группировку Dependabot PR для снижения noise?
15. Какие метрики можно отслеживать для оценки эффективности SCA в команде?

Требования к отчету

Отчет должен содержать следующие разделы:

1. Титульный лист с указанием ФИО студента, группы, варианта.
2. Цель и задачи работы.
3. Теоретическая часть:
 - Определение SCA и его место в DevSecOps.
 - Описание выбранного инструмента и его возможностей.
 - Обоснование выбора пороговых значений CVSS.
4. Практическая часть:
 - Листинг конфигурации пайплайна с SCA-этапом.
 - Листинг файла зависимостей (уязвимая версия).
 - Листинг исправленного файла зависимостей (с комментариями).
5. Результаты выполнения:
 - Скриншоты выполнения SCA-этапа в CI.
 - Скриншоты отчета о найденных уязвимостях.
 - Скриншоты Quality Gate (проход/непроход).
 - Скриншоты Dependabot PR (если настроен).
6. Анализ результатов:
 - Таблица найденных уязвимостей (CVE, CVSS, компонент, версия, статус).
 - Описание исправлений для каждой уязвимости.
 - Анализ ложных срабатываний (если есть).
7. Ответы на контрольные вопросы (3-5 вопросов).
8. Вывод - достигнута ли цель, какие сложности возникли, рекомендации по внедрению SCA.

Критерии оценивания

Критерий	Базовый уровень	Средний уровень	Продвинутый уровень
SCA-инструмент интегрирован в пайплайн	2 балла	2 балла	2 балла

Сгенерирован отчет об уязвимостях	2 балла	2 балла	2 балла
Обнаружены уязвимости в зависимостях	2 балла	2 балла	2 балла
Уязвимости исправлены и верифицированы	1 балл	2 балла	2 балла
Настроен Quality Gate (пороги CVSS)	0 баллов	2 балла	2 балла
Пайплайн падает при HIGH/CRITICAL	0 баллов	1 балл	2 балла
Настроен Dependabot (авто-PR)	0 баллов	1 балл	2 балла
Сгенерирован SBOM (CycloneDX/SPDX)	0 баллов	0 баллов	2 балла
Качество отчета и ответы на вопросы	2 балла	2 балла	2 балла

Шкала перевода баллов в оценку:

Оценка	Базовый уровень (макс. 9)	Средний уровень (макс. 14)	Продвинутый уровень (макс. 18)
Отлично (5)	8-9 баллов	13-14 баллов	16-18 баллов
Хорошо (4)	6-7 баллов	10-12 баллов	13-15 баллов
Удовлетворительно (3)	4-5 баллов	7-9 баллов	10-12 баллов
Неудовлетворительно (2)	менее 4 баллов	менее 7 баллов	менее 10 баллов

Дополнительные критерии (штрафы):

- Отсутствие демонстрации срыва пайплайна - минус 1 балл.
- Не указаны CVE номера в анализе - минус 1 балл.

- Отсутствие фиксации версий после обновления - минус 1 балл.

Бонусные баллы (макс. +3):

- Интеграция с Jira/дефект-трекером - +1 балл.
- Настройка группировки Dependabot PR - +1 балл.
- Генерация SBOM и загрузка в Dependency Track - +1 балл.

Лабораторная работа 8. Сборка и публикация Docker-образов

Цель работы: Сформировать практические навыки автоматизации сборки Docker-образов, публикации в container registry, а также применения best practices (multi-stage builds, кэширование, версионирование) в CI/CD-пайплайне.

Тео

2.1. Docker-образы и их место в CI/CD

Docker-образ -это легковесный, автономный исполняемый пакет, содержащий всё необходимое для запуска приложения: код, runtime, системные инструменты, библиотеки и настройки.

Преимущества использования Docker в CI/CD:

Преимущество	Описание
Воспроизводимость	Один и тот же образ работает одинаково на всех средах
Изоляция	Зависимости приложения изолированы от хостовой системы
Скорость развертывания	Запуск контейнера занимает секунды

Масштабируемость	Легко масштабировать горизонтально
Управление версиями	Образы версионированы в registry

Архитектура сборки и публикации в CI/CD (рисунок 1):

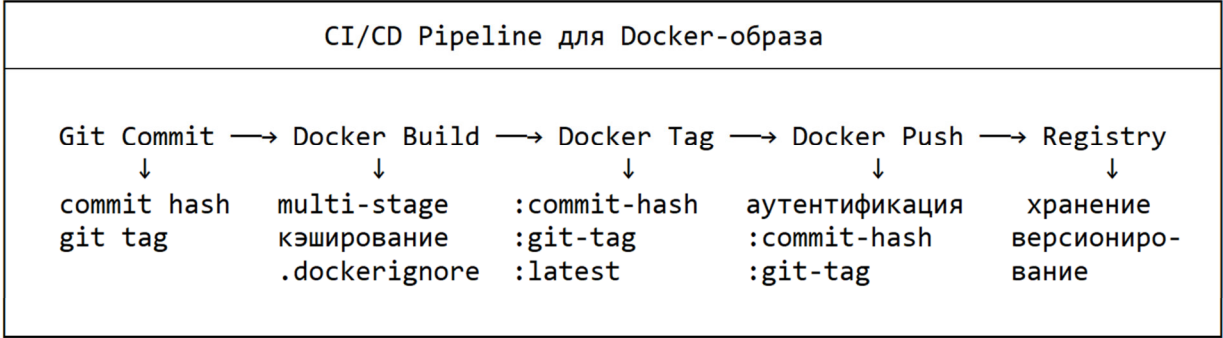


Рисунок 1

2. Best practices написания Dockerfile

Сравнение плохого и хорошего Dockerfile:

Аспект	Плохо	Хорошо
Базовый образ	ubuntu:latest	python:3.11-slim, alpine
Количество слоев	Каждая команда RUN в отдельности	Объединение команд RUN apt-get update && apt-get install -y ...
Кэширование	COPY . /app до установки зависимостей	COPY requirements.txt → RUN pip install → COPY .
Пользователь	root	Создание и переключение на non-root пользователя
Multi-stage	Не используется	Используется для уменьшения размера
.dockerignore	Отсутствует	Исключает .git,

		__pycache__, .env

Пример хорошего Dockerfile (Python):

```
# ===== Stage 1: Builder =====
FROM python:3.11-slim AS builder

WORKDIR /app

# Установка зависимостей отдельно от кода (кэширование)
COPY requirements.txt .
RUN pip install --user --no-cache-dir -r requirements.txt

# ===== Stage 2: Runtime =====
FROM python:3.11-slim

# Установка runtime-зависимостей
RUN apt-get update && apt-get install -y --no-install-recommends \
    curl \
    && rm -rf /var/lib/apt/lists/*

# Создание non-root пользователя
RUN useradd --create-home appuser
USER appuser

WORKDIR /app

# Копирование зависимостей из builder-стадии
COPY --from=builder --chown=appuser:appuser /home/appuser/.local
/home/appuser/.local

# Копирование кода приложения
COPY --chown=appuser:appuser . .

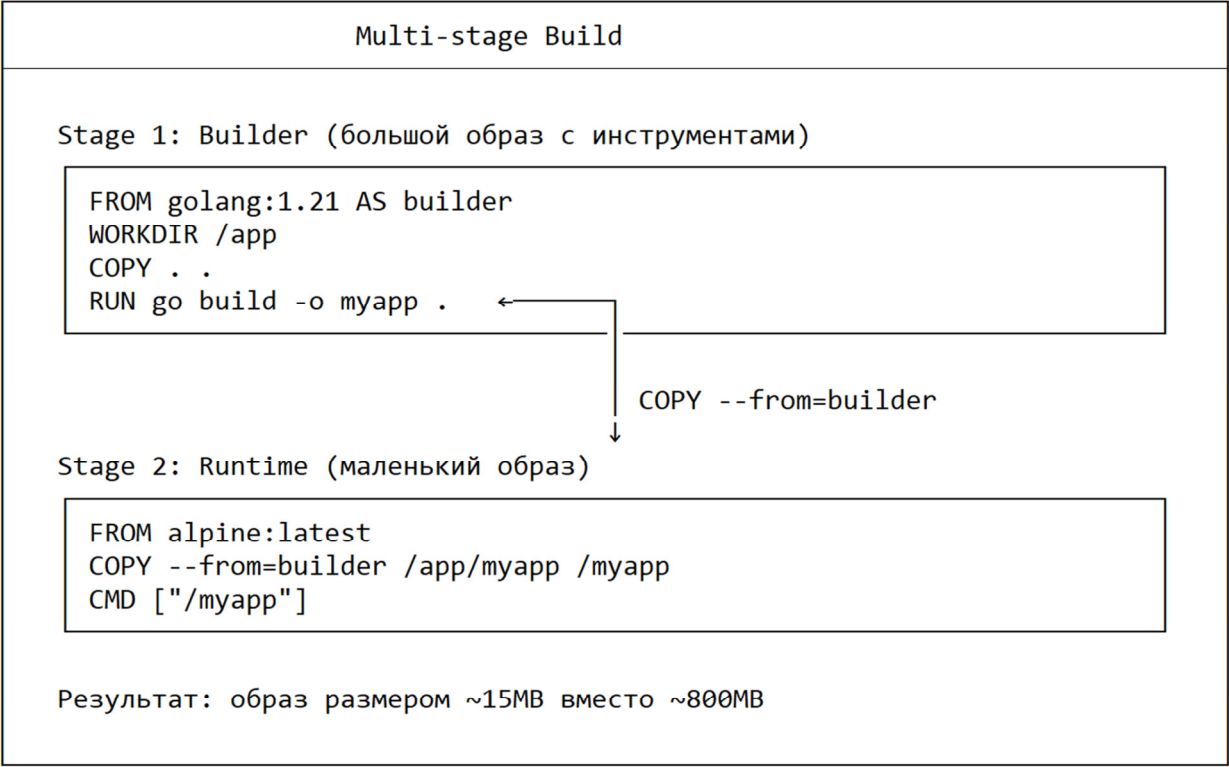
# Добавление локальных Python-пакетов в PATH
ENV PATH=/home/appuser/.local/bin:$PATH

# Проверка здоровья
HEALTHCHECK --interval=30s --timeout=3s --start-period=5s --retries=3 \
    CMD curl -f http://localhost:8000/health || exit 1

EXPOSE 8000
CMD ["python", "app.py"]
```

3. Multi-stage сборка

Multi-stage build - техника, позволяющая использовать несколько базовых образов в одном Dockerfile, копируя только необходимые артефакты между стадиями (рисунок 2).



		версию

Пример версионирования в CI (GitHub Actions):

```
- name: Set up Docker tags
  id: tags
  run: |
    # Short commit hash
    SHORT_SHA=$(git rev-parse --short HEAD)
    echo "sha=${SHORT_SHA}" >> $GITHUB_OUTPUT

    # Git tag (если есть)
    if [[ $GITHUB_REF == refs/tags/* ]]; then
      TAG=${GITHUB_REF#refs/tags/}
      echo "tag=${TAG}" >> $GITHUB_OUTPUT
    fi

- name: Build and push Docker image
  uses: docker/build-push-action@v5
  with:
    tags: |
      myapp:${{ steps.tags.outputs.sha }}
      myapp:${{ steps.tags.outputs.tag }}
      myapp:latest
```

2.5. Кэширование в CI/CD для Docker

Проблема: Сборка Docker-образа с нуля при каждом запуске CI занимает много времени.

Решение: Кэширование слоев Docker.

Стратегии кэширования:

Стратегия	GitHub Actions	GitLab CI
Встроенное кэширование	cache-from: type=gha	cache: [paths]
Регистр-кэш	cache-to: type=gha,mode=max	Не поддерживается
Локальное кэширование	actions/cache	cache: [paths: /var/lib/docker]

Пример кэширования в GitHub Actions:

```
- name: Set up Docker Buildx
  uses: docker/setup-buildx-action@v3

- name: Cache Docker layers
  uses: actions/cache@v3
  with:
    path: /tmp/.buildx-cache
    key: ${{ runner.os }}-buildx-${{ github.sha }}
    restore-keys: |
      ${{ runner.os }}-buildx-

- name: Build and push
  uses: docker/build-push-action@v5
  with:
    cache-from: type=local,src=/tmp/.buildx-cache
    cache-to: type=local,dest=/tmp/.buildx-cache-new,mode=max
```

2.6. Container Registry

Популярные registry:

Реестр	Бесплатный лимит	Интеграция с CI/CD
Docker Hub	1 приватный образ	GitHub Actions, GitLab CI
GitHub Container Registry	Неограниченно (для публичных)	GitHub Actions (нативная)
GitLab Container Registry	Неограниченно	GitLab CI (нативная)
Amazon ECR	500 MB (free tier)	AWS CLI
Azure Container Registry	Платный	Azure DevOps
Google Container Registry	Платный	GCP

Аутентификация в registry:

```
# GitHub Actions – Docker Hub
- name: Log in to Docker Hub
  uses: docker/login-action@v3
  with:
    username: ${ secrets.DOCKER_USERNAME }
    password: ${ secrets.DOCKER_TOKEN }

# GitHub Actions – GitHub Container Registry (GHCR)
- name: Log in to GHCR
  uses: docker/login-action@v3
  with:
    registry: ghcr.io
    username: ${ github.actor }
    password: ${ secrets.GITHUB_TOKEN }
```

Пример практического выполнения

1. Пример для Python (GitHub Actions + GHCR)

Файл Dockerfile:

```
# Multi-stage для Python-приложения
FROM python:3.11-slim AS builder

WORKDIR /app
COPY requirements.txt .
RUN pip install --user --no-cache-dir -r requirements.txt

FROM python:3.11-slim

RUN apt-get update && apt-get install -y --no-install-recommends \
  curl \
  && rm -rf /var/lib/apt/lists/*

RUN useradd --create-home appuser
USER appuser

WORKDIR /app

COPY --from=builder --chown=appuser:appuser /home/appuser/.local
/home/appuser/.local
COPY --chown=appuser:appuser . .

ENV PATH=/home/appuser/.local/bin:$PATH
```

```
HEALTHCHECK --interval=30s --timeout=3s --cmd="curl -f
http://localhost:8000/health || exit 1"

EXPOSE 8000

CMD ["python", "app.py"]
```

Файл .github/workflows/docker.yml:

```
name: Docker Build and Publish

on:
  push:
    branches: [ main ]
    tags:
      - 'v*'
  pull_request:
    branches: [ main ]

env:
  REGISTRY: ghcr.io
  IMAGE_NAME: ${ github.repository }

jobs:
  build-and-push:
    runs-on: ubuntu-latest
    permissions:
      contents: read
      packages: write

    steps:
      - uses: actions/checkout@v4

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v3

      - name: Log in to GHCR
        uses: docker/login-action@v3
        with:
          registry: ${ env.REGISTRY }
          username: ${ github.actor }
          password: ${ secrets.GITHUB_TOKEN }

      - name: Extract metadata (tags, labels)
        id: meta
        uses: docker/metadata-action@v5
        with:
          images: ${ env.REGISTRY }/${ env.IMAGE_NAME }
          tags: |
            type=sha,format=short
            type=ref,event=branch
```

```

        type=ref,event=pr
        type=semver,pattern={{version}}
        type=raw,value=latest,enable={{is_default_branch}}

- name: Build and push Docker image
  uses: docker/build-push-action@v5
  with:
    context: .
    push: ${{ github.event_name != 'pull_request' }}
    tags: ${{ steps.meta.outputs.tags }}
    labels: ${{ steps.meta.outputs.labels }}
    cache-from: type=gha
    cache-to: type=gha,mode=max

```

2. Пример для Node.js (GitLab CI + GitLab Container Registry)

Файл Dockerfile:

```

# Multi-stage для Node.js
FROM node:20-alpine AS builder

WORKDIR /app
COPY package*.json ./
RUN npm ci --only=production

FROM node:20-alpine

RUN apk add --no-cache curl

RUN addgroup -g 1001 -S nodejs && \
    adduser -S nodejs -u 1001

USER nodejs

WORKDIR /app

COPY --from=builder --chown=nodejs:nodejs /app/node_modules ./node_modules
COPY --chown=nodejs:nodejs . .

HEALTHCHECK --interval=30s --timeout=3s --cmd="curl -f
http://localhost:3000/health || exit 1"

EXPOSE 3000
CMD ["node", "server.js"]

```

Файл .gitlab-ci.yml:

```

stages:
  - build
  - publish

variables:
  DOCKER_IMAGE: $CI_REGISTRY_IMAGE:$CI_COMMIT_SHORT_SHA
  DOCKER_TAG_LATEST: $CI_REGISTRY_IMAGE:latest

docker-build:
  stage: build
  image: docker:latest
  services:
    - docker:dind
  script:
    - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD $CI_REGISTRY
    - docker build -t $DOCKER_IMAGE .
    - docker push $DOCKER_IMAGE

docker-publish-latest:
  stage: publish
  image: docker:latest
  services:
    - docker:dind
  script:
    - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD $CI_REGISTRY
    - docker pull $DOCKER_IMAGE
    - docker tag $DOCKER_IMAGE $DOCKER_TAG_LATEST
    - docker push $DOCKER_TAG_LATEST
  rules:
    - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH

```

3. Пример для Go (alpine + scratch)

Файл Dockerfile:

```

# Stage 1: Build (статически скомпилированный бинарник)
FROM golang:1.21-alpine AS builder

RUN apk add --no-cache git

WORKDIR /app

COPY go.mod go.sum ./
RUN go mod download

COPY . .

RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o myapp .

# Stage 2: Runtime (scratch – пустой образ)
FROM scratch

```

```
COPY --from=builder /app/myapp /myapp

# Настройка timezone
COPY --from=builder /usr/share/zoneinfo /usr/share/zoneinfo

EXPOSE 8080

ENTRYPOINT ["/myapp"]
```

Файл [github/workflows/go-docker.yml](https://github.com/workflows/go-docker.yml):

```
name: Go Docker Build

on:
  push:
    tags:
      - 'v*'

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Set up QEMU
        uses: docker/setup-qemu-action@v3

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v3

      - name: Log in to Docker Hub
        uses: docker/login-action@v3
        with:
          username: ${ secrets.DOCKER_USERNAME }
          password: ${ secrets.DOCKER_TOKEN }

      - name: Build and push multi-arch
        uses: docker/build-push-action@v5
        with:
          platforms: linux/amd64,linux/arm64,linux/arm/v7
          push: true
          tags: |
            myapp:${ github.ref_name }
            myapp:latest
```

4. Файл .dockerignore

```
# Git
.git/
```



```
.gitignore

# Python
__pycache__/*
*.py[cod]
*.so
.Python
venv/
.env

# Node.js
node_modules/
npm-debug.log

# IDE
.vscode/
.idea/

# CI/CD
.github/
.gitlab-ci.yml

# Secrets
*.pem
*.key

# Tests
tests/
coverage/
htmlcov/
```

Методические указания для выполнения работы

Задачи работы

1. Написание оптимизированного Dockerfile:

- Разработать Dockerfile с использованием multi-stage сборки.
- Применить best practices (минимальный базовый образ, .dockerignore, слои).
- Обеспечить безопасность (non-root пользователь).

2. Локальная сборка и тестирование образа:

- Собрать образ локально и проверить его работоспособность.
- Оценить размер образа и оптимизировать его.

3. Интеграция сборки в CI/CD:

- Настроить этап сборки Docker-образа в пайплайне.
 - Настроить кэширование слоев для ускорения сборки.
4. Версионирование образов:
- Использовать git commit hash для уникальной метки.
 - Использовать git tag для версионных меток (semantic versioning).
 - Добавить метку latest для последней стабильной версии.
5. Публикация в container registry:
- Аутентификация в registry (Docker Hub / GitLab Container Registry).
 - Настроить push образа после успешной сборки.
6. Оформление отчёта:
- Зафиксировать Dockerfile и конфигурацию пайплайна.
 - Представить скриншоты процесса сборки и опубликованных образов.

Варианты заданий

1. Базовый уровень

Общее требование: Написать Dockerfile с multi-stage сборкой. Настроить сборку образа в CI. Опубликовать образ в registry с тегом commit hash.

№	Язык/Платформа	Базовый образ	Registry
1	Python (Flask)	python:3.11-slim	Docker Hub
2	Python (FastAPI)	python:3.11-alpine	GHCR
3	Python (Django)	python:3.11-slim	GitLab CR
4	Node.js (Express)	node:20-alpine	Docker Hub
5	Node.js (NestJS)	node:20-slim	GHCR
6	Node.js (Next.js)	node:20-alpine	GitLab CR
7	Go (net/http)	golang:1.21-alpine → scratch	Docker Hub
8	Go (Gin)	golang:1.21-alpine → alpine	GHCR
9	Java (Spring Boot)	eclipse-temurin:17-jdk →	Docker Hub

		eclipse-temurin:17-jre	
10	Java (Micronaut)	maven:3.9 → openjdk:17-slim	GitLab CR
11	C (ASP.NET Core)	mcr.microsoft.com/dotnet/sdk:8.0 → aspnet:8.0	GHCR
12	Ruby (Sinatra)	ruby:3.2-slim	Docker Hub
13	PHP (Laravel)	php:8.2-fpm-alpine	GitLab CR
14	Rust (Actix)	rust:1.74 → debian:stable-slim	GHCR
15	Static Site (HTML)	nginx:alpine	Docker Hub

2. Средний уровень

Общее требование: Настроить кэширование Docker-слоев в CI. Добавить версионирование (commit hash + git tag + latest). Опубликовать образ в registry.

№	Язык	Дополнительные требования
1	Python	Добавить healthcheck, non-root пользователь
2	Python	Настроить кэширование pip пакетов в CI
3	Python	Использовать Docker BuildKit для параллельной сборки
4	Node.js	Оптимизировать размер образа (<200MB)
5	Node.js	Добавить multi-arch сборку (amd64 + arm64)
6	Node.js	Настроить labels (org.opencontainers.image.)
7	Go	Использовать scratch образ (размер <15MB)
8	Go	Добавить CA certificates в scratch образ
9	Java	Использование jlink для создания минимального JRE
10	Java	Настройка Spring Boot layers для кэширования
11	C	Использование Alpine base image
12	Rust	Статическая компиляция (musl) для alpine/scratch
13	Специфика БПЛА	Dockerfile для ROS 2 ноды (ubuntu + humble)

14	ML (Python + PyTorch)	Использование GPU-совместимого образа (cuda)
15	PHP	Использование FrankenPHP вместо php-fpm

3. Продвинутый уровень

Общее требование: Настроить multi-arch сборку (linux/amd64, linux/arm64, linux/arm/v7). Реализовать автоматическое обновление latest тега. Настроить CI для автоматической публикации при создании git tag. Использовать SBOM (Software Bill of Materials).

№	Язык	Сценарий	Дополнительные требования
1	Python	Multi-arch + кэширование buildx	Сборка на QEMU эмуляторе
2	Python	GitHub Actions + OIDC для аутентификации	Без long-lived secrets
3	Python	Автоматический релиз при push тега	Semantic versioning
4	Node.js	Multi-arch + arm64 поддержка	GitHub Actions runner (arm64)
5	Node.js	Docker Scout для анализа уязвимостей	Интеграция с security dashboard
6	Node.js	Автоматический push в GHCR + Docker Hub	Два registry одновременно
7	Go	Multi-arch + scratch + distroless	Сравнение размера образов
8	Go	Cosign подпись образов	Подпись и верификация
9	Java	Native image (GraalVM)	Сравнение с обычным JVM-образом
10	Java	jlink + multi-arch	Минимальный

			кастомный JRE
11	Специфика БПЛА	PX4 + ROS 2 в одном образе	Совмещение двух экосистем
12	ROS 2	Docker для Gazebo симуляции	Проброс GPU для рендеринга
13	Security	SBOM generation (Syft, Trivy)	Генерация SPDX/CycloneDX
14	Supply Chain	SLSA level 2+ compliance	Прослеживаемость сборки
15	Комплексный	CI + CD (авто-деплой в k8s после публикации)	Полный пайплайн

Контрольные вопросы

1. Что такое Docker-образ и Docker-контейнер? В чем разница?
2. Для чего используется multi-stage сборка в Dockerfile?
3. Какие best practices при написании Dockerfile вы знаете?
4. Почему важно использовать .dockerignore?
5. Как уменьшить размер Docker-образа? Какие стратегии существуют?
6. Что такое non-root пользователь в Dockerfile и зачем он нужен?
7. Как настроить кэширование Docker-слоев в GitHub Actions?
8. Какие стратегии тегирования Docker-образов вы знаете?
9. Чем отличается Docker Hub от GitHub Container Registry?
10. Как настроить аутентификацию в registry из CI/CD?
11. Что такое Docker BuildKit и какие преимущества он дает?
12. Как собрать образ для нескольких архитектур (amd64, arm64)?
13. Как добавить healthcheck в Dockerfile?
14. Что такое дистрибутив Distroless и в чем его отличие от Alpine?
15. Как проверить безопасность Docker-образа (сканирование уязвимостей)?

Требования к отчету

Отчет должен содержать следующие разделы:

1. Титульный лист с указанием ФИО студента, группы, варианта.
2. Цель и задачи работы.
3. Теоретическая часть:
 - Обоснование выбора базового образа и multi-stage стратегии.
 - Схема сборки и публикации образа.
4. Практическая часть:
 - Листинг Dockerfile с комментариями.
 - Листинг .dockerignore.
 - Листинг конфигурации CI/CD (сборка, тегирование, push).
5. Результаты выполнения:
 - Скриншоты процесса сборки в CI.
 - Скриншоты опубликованных образов в registry (список тегов).
 - Размер образа (до и после оптимизации, если применимо).
6. Анализ результатов:
 - Сравнение размера образа с альтернативными подходами.
 - Анализ времени сборки с кэшированием и без.
7. Ответы на контрольные вопросы (3-5 вопросов).
8. Вывод — достигнута ли цель, какие сложности возникли.

Критерии оценивание

Критерий	Базовый уровень	Средний уровень	Продвинутый уровень
Dockerfile с multi-stage	2 балла	2 балла	2 балла
Сборка образа в CI	2 балла	2 балла	2 балла

Публикация в registry	2 балла	2 балла	2 балла
Версионирование (commit hash)	1 балл	2 балла	2 балла
Настроено кэширование слоев	0 баллов	1 балл	2 балла
Добавлен git tag + latest	0 баллов	1 балл	2 балла
Non-root пользователь + healthcheck	0 баллов	1 балл	2 балла
Multi-arch сборка	0 баллов	0 баллов	2 балла
Качество отчета и ответы	2 балла	2 балла	2 балла

Шкала перевода баллов в оценку:

Оценка	Базовый уровень (макс. 9)	Средний уровень (макс. 13)	Продвинутый уровень (макс. 18)
Отлично (5)	8-9 баллов	12-13 баллов	16-18 баллов
Хорошо (4)	6-7 баллов	10-11 баллов	13-15 баллов
Удовлетворительно (3)	4-5 баллов	7-9 баллов	10-12 баллов
Неудовлетворительно (2)	менее 4 баллов	менее 7 баллов	менее 10 баллов

Бонусные баллы (макс. +3):

- Размер образа <50MB — +1 балл.
- Настройка SBOM (Syft/Trivy) — +1 балл.
- Подпись образа (Cosign) — +1 балл.

Лабораторная работа 9. Управление артефактами

Цель работы: Сформировать практические навыки работы с репозиториями артефактов: настройки хранилища, публикации собранных артефактов (jar, wheel, bin) и их загрузки на последующих этапах пайплайна.

Теоретическое обоснование

1. Понятие артефакта и репозитория артефактов

Артефакт - это результат сборки программного проекта: бинарный файл, пакет, Docker-образ, документация, отчёты о тестировании. Артефакты являются «строительными блоками» для последующих этапов CI/CD и для развертывания.

Репозиторий артефактов - это хранилище, обеспечивающее:

Функция	Описание
Хранение	Централизованное место для всех бинарных артефактов
Версионирование	Сохранение всех версий артефактов, возможность отката
Контроль доступа	Разграничение прав на публикацию и загрузку
Кэширование	Проксирование удалённых репозиторийев (npm, PyPI, Maven Central)
Безопасность	Сканирование артефактов на уязвимости (SCA)

2.2. Сравнение систем управления артефактами

Характеристика	GitLab Package Registry	GitHub Packages	JFrog Artifactory	Sonatype Nexus
Бесплатная	Да (с	Да	Freemium	OSS

версия	ограничениями)	(публичные)		версия
Поддерживаемые форматы	PyPI, Maven, npm, NuGet, Go, Composer, Conan, Terraform	npm, Maven, NuGet, RubyGems, Docker	30+ форматов	Maven, npm, Docker, Raw
Интеграция с CI/CD	Нативная (GitLab CI)	Нативная (GitHub Actions)	Jenkins, GitLab, GitHub	Jenkins, GitLab
Проксирование	Dependency Proxy	Нет	Да	Да
Промоушн артефактов	Ограничен	Нет	Да (с promotion API)	Нет

Рекомендации по выбору:

- GitLab - если весь цикл в GitLab CI/CD
- GitHub Packages - если в GitHub Actions
- Artifactory - для enterprise с множеством форматов
- Nexus - для Maven/Java-проектов и raw-артефактов

3. Форматы артефактов и их особенности

Формат	Типичное расширение	Менеджер пакетов	Пример команды публикации
Python (PyPI)	.whl, .tar.gz	pip / twine	twine upload --repository gitlab dist/
JavaScript (npm)	.tgz	npm / yarn	npm publish --

			registry
Java (Maven)	.jar, .pom	Maven / Gradle	mvn deploy
Go	модуль	go	go mod download
Generic / Raw	любой	curl / API	curl --upload-file

4. Версионирование артефактов

Стратегии версионирования:

Тип	Пример	Когда использовать
Snapshot	myapp-1.2.3-SNAPSHOT.jar	Промежуточные сборки, ночные билды
Release	myapp-1.2.3.jar	Стабильные версии, готовые к production
Commit-based	myapp-7a8d2f1.whl	Уникальная метка для каждого коммита
Semantic Versioning	v1.2.3	Major.Minor.Patch - стандарт для релизов

Semantic Versioning (SemVer) - формат MAJOR.MINOR.PATCH:

- MAJOR - несовместимые изменения API
- MINOR - новая функциональность (обратно совместимая)
- PATCH - исправления ошибок

5. Архитектура управления артефактами в CI/CD (рисунок 1)

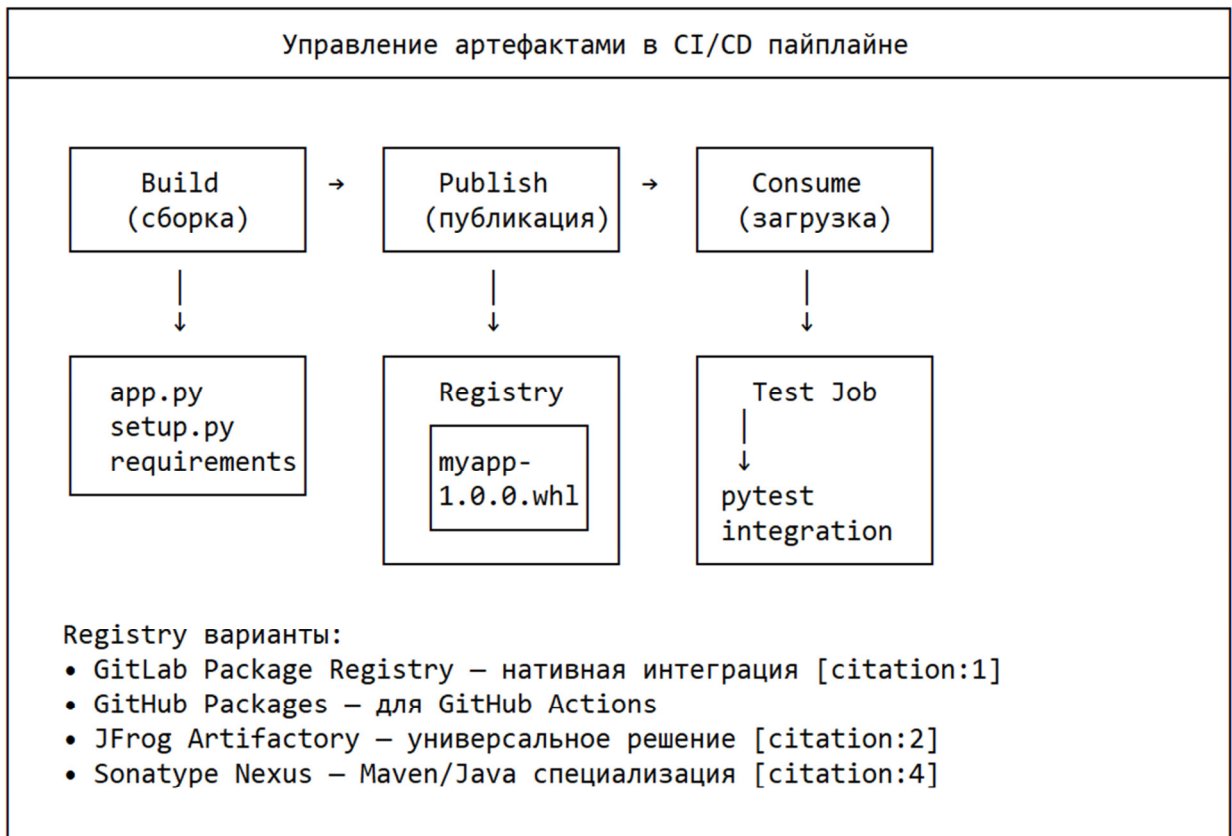


Рисунок 1

6. Публикация в GitLab Package Registry

Настройка CI/CD для PyPI-пакета :

```
stages:
  - build
  - publish

variables:
  TWINE_USERNAME: gitlab-ci-token
  TWINE_PASSWORD: ${CI_JOB_TOKEN}
  TWINE_REPOSITORY_URL: ${CI_API_V4_URL}/projects/${CI_PROJECT_ID}/packages/pypi

build-wheel:
  stage: build
  script:
    - python -m pip install build
    - python -m build
  artifacts:
    paths:
      - dist/

publish-pypi:
```

```

stage: publish
script:
  - pip install twine
  - TWINE_PASSWORD=${CI_JOB_TOKEN} python -m twine upload --repository-url
    ${TWINE_REPOSITORY_URL} dist/*
rules:
  - if: $CI_COMMIT_TAG

```

Настройка для npm-пакета

```

publish-npm:
  stage: publish
  script:
    - echo
    "@${CI_PROJECT_ROOT_NAMESPACE}:registry=https://${CI_SERVER_HOST}/api/v4/projects
    /${CI_PROJECT_ID}/packages/npm/" >> .npmrc
    - echo
    "//${CI_SERVER_HOST}/api/v4/projects/${CI_PROJECT_ID}/packages/npm/:_authToken=${
    CI_JOB_TOKEN}" >> .npmrc
    - npm publish
  rules:
    - if: $CI_COMMIT_TAG

```

7. Публикация в GitHub Packages

Для Maven-артефакта:

```

- name: Publish package
  run: mvn --batch-mode deploy
  env:
    GITHUB_TOKEN: ${ secrets.GITHUB_TOKEN }

```

Для npm-пакета:

```

- name: Publish npm package
  run: |
    echo "//npm.pkg.github.com/:_authToken=${ secrets.GITHUB_TOKEN }" >> .npmrc
    npm publish

```

8. Загрузка артефакта из реестра

Загрузка PyPI-пакета для тестирования:

```
test-with-package:
  stage: test
  script:
    - pip install --index-url
      ${CI_API_V4_URL}/projects/${CI_PROJECT_ID}/packages/pypi/simple my-package
    - pytest tests/
```

Загрузка через API (raw/universal артефакты):

```
download-artifact:
  stage: consume
  script:
    - curl --header "PRIVATE-TOKEN: ${CI_JOB_TOKEN}" \
      "${CI_API_V4_URL}/projects/${CI_PROJECT_ID}/packages/generic/myapp/${V
      ERSION}/myapp.bin" \
      --output myapp.bin
```

Примеры практического выполнения

1. Пример для Python (GitLab CI + PyPI registry)

Структура проекта:

```
my-python-lib/
├── .gitlab-ci.yml
├── pyproject.toml
├── src/
│   └── mylib/
│       └── __init__.py
└── tests/
```

Файл pyproject.toml (фрагмент):

```
[build-system]
requires = ["setuptools>=61.0", "wheel"]
build-backend = "setuptools.build_meta"

[project]
name = "my-python-lib"
```

```
version = "0.1.0"
```

Файл .gitlab-ci.yml:

```
stages:
  - build
  - publish
  - consume

variables:
  TWINE_USERNAME: gitlab-ci-token
  TWINE_PASSWORD: ${CI_JOB_TOKEN}
  TWINE_REPOSITORY_URL: ${CI_API_V4_URL}/projects/${CI_PROJECT_ID}/packages/pypi

build-wheel:
  stage: build
  image: python:3.11-slim
  script:
    - pip install build
    - python -m build
  artifacts:
    paths:
      - dist/
  only:
    - tags

publish-pypi:
  stage: publish
  image: python:3.11-slim
  script:
    - pip install twine
    - python -m twine upload --repository-url ${TWINE_REPOSITORY_URL} dist/*
  only:
    - tags

test-install:
  stage: consume
  image: python:3.11-slim
  script:
    - pip install --index-url
    ${CI_API_V4_URL}/projects/${CI_PROJECT_ID}/packages/pypi/simple my-python-lib
    - python -c "import mylib; print(mylib.__version__)"
  only:
    - tags
```

2. Пример для npm (GitHub Actions + GitHub Packages)

Файл .github/workflows/publish.yml:

```
name: Publish npm package

on:
  push:
    tags:
      - 'v*'

jobs:
  build-and-publish:
    runs-on: ubuntu-latest
    permissions:
      contents: read
      packages: write
    steps:
      - uses: actions/checkout@v4

      - uses: actions/setup-node@v4
        with:
          node-version: '20'
          registry-url: 'https://npm.pkg.github.com'

      - run: npm ci
      - run: npm test
      - run: npm run build
      - run: npm publish
      env:
        NODE_AUTH_TOKEN: ${ secrets.GITHUB_TOKEN }

      - name: Verify installation
        run: npm install @${ github.repository_owner }/${ github.event.repository.name }
```

3. Пример для Raw-артефактов (Nexus + curl)

Публикация бинарного файла в Nexus Raw репозиторий :

```
publish-raw:
  stage: publish
  script:
    - curl -v -u ${NEXUS_USER}:${NEXUS_PASSWORD} \
      --upload-file ./bin/myapp \
      "http://nexus.example.com/repository/raw-repo/myapp/${CI_COMMIT_SHORT_SHA}/myapp"
```

4. Пример для Maven (GitLab CI)

Файл .gitlab-ci.yml:

```
stages:
  - build
  - publish

variables:
  MAVEN_CLI_OPTS: "-s ci-settings.xml --batch-mode"

publish-maven:
  stage: publish
  image: maven:3.9-eclipse-temurin-17
  script:
    - mvn $MAVEN_CLI_OPTS deploy
  only:
    - tags
```

Методические указания выполнения практической части работы

Задачи работы:

1. Настройка репозитория артефактов:

- Выбрать и настроить репозиторий (GitLab Package Registry / GitHub Packages / Nexus / Artifactory).
- Настроить аутентификацию для публикации и загрузки артефактов .

2. Публикация артефакта в репозиторий:

- Собрать артефакт (wheel, jar, бинарный файл, Docker-образ) в CI/CD.
- Настроить этап публикации в package registry .
- Добавить версионирование артефакта (SemVer, commit hash).

3. Загрузка артефакта на последующих этапах:

- Настроить этап, который загружает ранее опубликованный артефакт.
- Использовать артефакт для развертывания или тестирования.

4. Управление версиями и жизненным циклом:

- Настроить политику хранения артефактов (retention).
- Реализовать промоушн артефактов между средами (snapshot → release).

5. Оформление отчёта:

- Зафиксировать конфигурацию пайплайна.
- Представить скриншоты опубликованных артефактов в реестре.

Варианты заданий

Базовый уровень

Общее требование: Настроить публикацию артефакта в реестр. Использовать commit hash в качестве версии. Продемонстрировать успешную публикацию и загрузку артефакта.

№	Язык/Платформа	Тип артефакта	Реестр
1	Python	wheel	GitLab PyPI registry
2	Python	source dist (.tar.gz)	GitLab PyPI registry
3	JavaScript (npm)	.tgz	GitHub Packages
4	JavaScript (npm)	.tgz	GitLab npm registry
5	Maven (Java)	.jar	GitHub Packages
6	Maven (Java)	.jar	GitLab Maven registry
7	Gradle (Java)	.jar	GitLab Maven registry
8	Go	модуль	GitLab Go proxy
9	.NET (C)	.nupkg	GitHub Packages
10	.NET (C)	.nupkg	GitLab NuGet registry
11	Generic	бинарный файл	GitLab Generic registry
12	Generic	бинарный файл	Nexus Raw
13	Ruby	.gem	GitHub Packages

14	PHP	.phar	GitLab registry	Composer
15	Rust	.crate	GitLab registry	(cargo-local-registry)

Средний уровень

Общее требование: Настроить публикацию с версионированием по git tag (SemVer). Добавить этап загрузки артефакта для тестирования. Настроить политику хранения (retention).

№	Язык	Дополнительные требования
1	Python	Автоматическое извлечение версии из git tag в setup.py/pyproject.toml
2	Python	Публикация и snapshot, и release версий
3	Python	Настройка GitLab CI для публикации только при создании тега
4	JavaScript	Автоматический bump версии на основе commit message (semantic-release)
5	JavaScript	Публикация в GitHub Packages и npmjs.org одновременно
6	JavaScript	Настройка .npmrc через CI переменные
7	Maven	Настройка CI/CD для публикации SNAPSHOT и RELEASE версий
8	Maven	Использование CI_JOB_TOKEN для аутентификации в GitLab
9	Go	Настройка Go proxy с sumdb
10	Generic	Публикация нескольких артефактов (bin + config + docs)

11	Generic	Настройка retention policy (удаление старых версий)
12	Python + Docker	Публикация wheel + Docker image в разные реестры
13	Maven + npm	Моно-репозиторий с публикацией двух типов артефактов
14	.NET	Подпись NuGet-пакета перед публикацией
15	Rust	Публикация библиотеки в crates.io через CI

Продвинутый уровень

Общее требование: Реализовать промоушн артефактов между средами (snapshot → staging → release). Настроить автоматическую загрузку артефакта для развертывания. Интегрировать SCA-сканирование артефактов.

№	Язык	Сценарий	Дополнительные требования
1	Python	Промоушн: snapshot (dev) → release (prod)	Разные реестры для разных сред
2	Python	CI/CD с публикацией и автоматическим тестированием	Установка опубликованного пакета в виртуальное окружение
3	Python	SBOM генерация + публикация вместе с артефактом	CycloneDX/SPDX
4	JavaScript	Промоушн через GitHub Environments	manual approval для production
5	JavaScript	Автоматический release на основе conventional commits	semantic-release + GitHub Packages
6	JavaScript	Подпись npm-пакета (Sigstore/Cosign)	Верификация подписи

7	Maven	Промоушн через Artifactory (snapshot → release)	Использование Artifactory API
8	Maven	Интеграция с OWASP Dependency-Check перед публикацией	Блокировка публикации при уязвимостях
9	Go	Публикация модуля с checksum verification	sum.golang.org интеграция
10	Generic	Создание release в GitLab с прикреплением артефактов	Использование release-cli
11	Специфика БПЛА	Публикация прошивки (.px4) в Generic registry	ОТА-пакет + манифест
12	ROS 2	Публикация ROS-пакета (.deb)	Подпись пакета
13	Комплексный	Многоступенчатый пайплайн: build → test → publish → deploy	Артефакт проходит все этапы
14	Security-hardened	Подпись артефакта (GPG/Cosign) + верификация при загрузке	Безопасная цепочка поставок
15	Supply Chain	SBOM + SCA + подпись + реестр	Полный цикл безопасного управления артефактами

Контрольные вопросы

1. Что такое артефакт в контексте CI/CD? Приведите примеры.
2. Для чего нужен репозиторий артефактов (package registry)?
3. В чем отличие GitLab Package Registry от GitHub Packages?
4. Какие стратегии версионирования артефактов вы знаете?
5. Что такое SNAPSHOT-версия в Maven и для чего она используется?
6. Как настроить аутентификацию для публикации в GitLab Package Registry?
7. Как опубликовать Python-пакет в GitLab CI?
8. Как опубликовать npm-пакет в GitHub Packages?
9. Что такое Generic/Raw репозиторий и для чего он используется?
10. Как загрузить артефакт из реестра на последующем этапе пайплайна?
11. Как настроить публикацию только при создании git tag?
12. Что такое retention policy и как её настроить?
13. Какие форматы артефактов поддерживает JFrog Artifactory?
14. Как подписать артефакт (Cosign/GPG) и зачем это нужно?
15. Как организовать промоушн артефактов между средами (dev → staging → prod)?

Требования к отчету

Отчет должен содержать следующие разделы:

1. Титульный лист с указанием ФИО студента, группы, варианта.
2. Цель и задачи работы.
3. Теоретическая часть:
 - Определение артефакта и его роли в CI/CD.

- Описание выбранного реестра артефактов.

- Стратегия версионирования.

4. Практическая часть:

- Листинг конфигурации пайплайна (публикация).

- Листинг конфигурации для загрузки артефакта.

- Листинг файлов сборки (setup.py, package.json, pom.xml).

5. Результаты выполнения:

- Скриншоты опубликованных артефактов в реестре.

- Скриншоты этапов публикации в CI.

- Скриншоты загрузки и использования артефакта.

6. Анализ результатов:

- Таблица версий опубликованных артефактов.

- Сравнение размеров артефактов (если применимо).

7. Ответы на контрольные вопросы (3-5 вопросов).

8. Вывод — достигнута ли цель, какие сложности возникли.

Критерии оценивания

Критерий	Базовый уровень	Средний уровень	Продвинутый уровень
Артефакт собран и опубликован в реестр	2 балла	2 балла	2 балла
Артефакт успешно загружен на др. этапе	2 балла	2 балла	2 балла
Настроено версионирование (commit hash/tag)	1 балл	2 балла	2 балла
Публикация только для тегов (по условию)	0 баллов	1 балл	2 балла
Настроен retention/promotion артефактов	0 баллов	1 балл	2 балла

Подпись артефакта (Cosign/GPG)	0 баллов	0 баллов	2 балла
Интеграция SCA-сканирования артефактов	0 баллов	0 баллов	2 балла
Качество отчета и ответы на вопросы	2 балла	2 балла	2 балла

Шкала перевода баллов в оценку:

Оценка	Базовый уровень (макс. 7)	Средний уровень (макс. 10)	Продвинутый уровень (макс. 16)
Отлично (5)	6-7 баллов	9-10 баллов	14-16 баллов
Хорошо (4)	5 баллов	7-8 баллов	11-13 баллов
Удовлетворительно (3)	3-4 балла	5-6 баллов	8-10 баллов
Неудовлетворительно (2)	менее 3 баллов	менее 5 баллов	менее 8 баллов

Бонусные баллы (макс. +3):

- Публикация в два реестра одновременно - +1 балл.
- Настройка автоматического промоушна (snapshot → release) - +1 балл.
- Генерация SBOM и публикация вместе с артефактом - +1 балл.

Лабораторная работа 10. Развертывание на тестовый стенд

Цель работы: Сформировать практические навыки автоматизации развертывания приложений в тестовую среду с использованием SSH (копирование файлов, удаленное выполнение команд) или Docker Compose, а

также проведения smoke-тестов для проверки работоспособности развернутого приложения.

Теоретическое обоснование

1. Окружения и стратегии развертывания

Окружения (environments) это изолированные инфраструктурные среды, через которые проходит приложение на пути к пользователю.

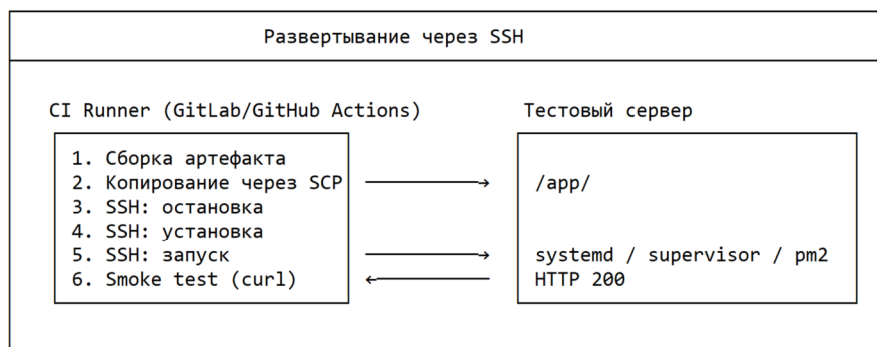
Окружение	Назначение	Типичный владелец
dev	Разработка, быстрые итерации	Разработчики
test / staging	Тестирование перед релизом	QA, автоматизация
production	Боевая среда, реальные пользователи	DevOps, SRE

Стратегии развертывания:

Стратегия	Описание	Когда использовать
SSH / SCP	Копирование файлов и запуск скриптов	Простые приложения, монолиты
Docker Compose	Оркестрация нескольких контейнеров	Многокомпонентные приложения
Kubernetes	Полноценная оркестрация	Микросервисы, масштабирование
Blue-Green	Два одинаковых окружения, переключение трафика	Zero-downtime production

2. Автоматизация через SSH

Схема развертывания по SSH:



Настройка SSH-ключей в CI:

GitHub Actions

```
- name: Setup SSH key
  run: |
    mkdir -p ~/.ssh
    echo "${{ secrets.SSH_PRIVATE_KEY }}" > ~/.ssh/id_rsa
    chmod 600 ~/.ssh/id_rsa
    ssh-keyscan -H "${{ secrets.SERVER_HOST }}" >> ~/.ssh/known_hosts
```

```
# GitLab CI
before_script:
  - mkdir -p ~/.ssh
  - echo "$SSH_PRIVATE_KEY" > ~/.ssh/id_rsa
  - chmod 600 ~/.ssh/id_rsa
  - ssh-keyscan -H $SERVER_HOST >> ~/.ssh/known_hosts
```

3. Автоматизация через Docker Compose

Преимущества Docker Compose для тестового стенда:

Преимущество	Описание
Изоляция	Каждый компонент в своем контейнере
Воспроизводимость	Один docker-compose.yml для всех сред
Зависимости	Автоматический запуск БД, кэша, очередей
Простота	Одна команда для поднятия стенда

Пример docker-compose.yml для тестового окружения:

```
version: '3.8'

services:
  app:
    image: myapp:${TAG:-latest}
    ports:
      - "8080:8080"
    environment:
      - DATABASE_URL=postgresql://user:pass@db:5432/testdb
      - REDIS_URL=redis://redis:6379
    depends_on:
      - db
      - redis
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8080/health"]
      interval: 10s
      timeout: 5s
      retries: 5

  db:
    image: postgres:15-alpine
    environment:
      - POSTGRES_USER=user
      - POSTGRES_PASSWORD=pass
      - POSTGRES_DB=testdb
    volumes:
      - postgres_data:/var/lib/postgresql/data

  redis:
    image: redis:7-alpine

volumes:
  postgres_data:
```

Развертывание через Docker Compose в CI:

```
# Копирование docker-compose.yml на сервер
scp docker-compose.yml user@server:/app/

# Удаленное выполнение
ssh user@server "
  cd /app && \
  docker-compose pull && \
```

```
docker-compose up -d --force-recreate && \
docker-compose ps
"
```

4. Smoke-тестирование

Smoke test - минимальный набор проверок, подтверждающих, что приложение "не дымится" и базово работоспособно.

Что проверять в smoke test:

Тип проверки	Пример команды
HTTP-доступность	curl -f http://localhost:8080/health
БД подключена	curl -f http://localhost:8080/api/status
Сервис отвечает	curl -f http://localhost:8080/ --max-time 10
Логи без ошибок	docker logs app grep -i error

Пример smoke test в CI:

```
# Ожидание готовности сервиса
for i in {1..30}; do
  if curl -s -f http://localhost:8080/health; then
    echo "Service is ready"
    exit 0
  fi
  echo "Waiting for service... ($i/30)"
  sleep 2
done
echo "Service failed to start"
exit 1
```

5. Окружения в GitLab CI

Декларация окружения:

```
deploy_test:
```

```
stage: deploy
environment:
  name: test
  url: https://test.example.com
script:
  - deploy.sh
```

Преимущества декларации окружений:

- Отслеживание деплоев в UI GitLab
- Approvals для production
- История развертываний

Примеры практического выполнения

1. Пример: развертывание Python-приложения через SSH

Файл .gitlab-ci.yml:

```
stages:
  - build
  - deploy_test

variables:
  SERVER_HOST: "test-server.example.com"
  SERVER_USER: "deploy"
  APP_DIR: "/var/www/myapp"

build:
  stage: build
  script:
    - pip install -r requirements.txt
    - python -m compileall .
  artifacts:
    paths:
      - .

deploy_test:
  stage: deploy_test
  environment:
    name: test
    url: http://test.myapp.example.com
  before_script:
```

```

- mkdir -p ~/.ssh
- echo "$SSH_PRIVATE_KEY" > ~/.ssh/id_rsa
- chmod 600 ~/.ssh/id_rsa
- ssh-keyscan -H $SERVER_HOST >> ~/.ssh/known_hosts
script:
# Копирование кода
- scp -r . $SERVER_USER@$SERVER_HOST:$APP_DIR/

# Удаленное выполнение команд
- ssh $SERVER_USER@$SERVER_HOST "
    cd $APP_DIR &&
    python3 -m venv venv &&
    source venv/bin/activate &&
    pip install -r requirements.txt &&
    sudo systemctl restart myapp
"

# Smoke test
- |
for i in {1..30}; do
    if curl -s -f http://test.myapp.example.com/health; then
        echo "Deploy successful!"
        exit 0
    fi
    echo "Waiting for service... ($i/30)"
    sleep 2
done
echo "Service failed to start"
exit 1

```

2. Пример: развертывание через Docker Compose (GitHub Actions)

Файл docker-compose.test.yml:

```

version: '3.8'

services:
  app:
    image: ghcr.io/${{ github.repository_owner }}/myapp:${{ github.sha }}
    ports:
      - "8080:8080"
    environment:
      - NODE_ENV=test
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8080/health"]
      interval: 10s
      timeout: 5s

```

```
retries: 5
```

Файл .github/workflows/deploy-test.yml:

```
name: Deploy to Test

on:
  push:
    branches: [ develop ]

jobs:
  deploy:
    runs-on: ubuntu-latest
    environment:
      name: test
      url: http://test.myapp.example.com
    steps:
      - uses: actions/checkout@v4

      - name: Setup SSH
        run: |
          mkdir -p ~/.ssh
          echo "${{ secrets.SSH_PRIVATE_KEY }}" > ~/.ssh/id_rsa
          chmod 600 ~/.ssh/id_rsa
          ssh-keyscan -H "${{ secrets.SERVER_HOST }}" >> ~/.ssh/known_hosts

      - name: Deploy with Docker Compose
        run: |
          scp docker-compose.test.yml "${{ secrets.SERVER_USER }}"@"${{ secrets.SERVER_HOST }}:/app/docker-compose.yml
          ssh "${{ secrets.SERVER_USER }}"@"${{ secrets.SERVER_HOST }}" "
            cd /app && \
            docker-compose pull && \
            docker-compose up -d --force-recreate
          "

      - name: Smoke test
        run: |
          sleep 10
          curl -f http://test.myapp.example.com/health
```

3. Пример: развертывание Node.js приложения с PM2 через SSH

Файл ecosystem.config.js (PM2):

```
module.exports = {  
  apps: [{  
    name: 'myapp',  
    script: 'dist/server.js',  
    env: {  
      NODE_ENV: 'test',  
      PORT: 3000  
    },  
    instances: 2,  
    exec_mode: 'cluster'  
  }]  
};
```

Файл .gitlab-ci.yml:

```
deploy_test:  
  stage: deploy_test  
  script:  
    # Копирование  
    - scp -r dist/ package.json ecosystem.config.js  
      $SERVER_USER@$SERVER_HOST:/app/  
  
    # Удаленная установка и запуск через PM2  
    - ssh $SERVER_USER@$SERVER_HOST "  
      cd /app &&  
      npm ci --production &&  
      pm2 delete myapp || true &&  
      pm2 start ecosystem.config.js &&  
      pm2 save  
    "  
  
    # Smoke test  
    - curl -f http://test.myapp.example.com:3000/health
```

Методические указания выполнения практической части работы

Задачи работы

1. Настройка тестового окружения:

- Подготовить удаленный сервер или локальный стенд для развертывания.

- Настроить конфигурацию окружения «test» (переменные, порты, доступы).

2. Автоматизация развертывания через SSH:

- Настроить безопасное подключение по SSH (ключи, известные хосты).

- Реализовать копирование артефактов на удаленный сервер (scp/rsync).

- Выполнить удаленные команды для установки и запуска приложения.

3. Автоматизация развертывания через Docker Compose:

- Написать docker-compose.yml для тестового окружения.

- Настроить запуск контейнеров на тестовом стенде.

4. Проверка работоспособности (smoke test):

- Реализовать автоматическую проверку HTTP-эндпоинтов (curl, wget).

- Настроить проверку логов контейнера/приложения.

- Настроить срыв пайплайна при неудачной проверке.

5. Оформление отчёта:

- Зафиксировать конфигурацию развертывания в CI/CD.

- Представить скриншоты успешного развертывания и smoke-тестов.

Варианты заданий

Базовый уровень

Общее требование: Настроить автоматическое развертывание на тестовый стенд (SSH или Docker Compose). Реализовать smoke test (curl HTTP-эндпоинта). Продемонстрировать успешное развертывание.

№	Язык/Платформа	Способ	Smoke test

		развертывания	
1	Python (Flask)	SSH + systemd	curl /health
2	Python (FastAPI)	SSH + gunicorn	curl /health
3	Python (Django)	SSH + uWSGI	curl /admin/
4	Node.js (Express)	SSH + PM2	curl /health
5	Node.js (NestJS)	SSH + systemd	curl /api/status
6	Node.js (Next.js)	SSH + npm start	curl /
7	Go (net/http)	SSH + binary	curl /health
8	Go (Gin)	Docker Compose	curl /ping
9	Java (Spring Boot)	Docker Compose	curl /actuator/health
10	Java (Micronaut)	SSH + java -jar	curl /health
11	C (ASP.NET)	Docker Compose	curl /health
12	Ruby (Sinatra)	SSH + rackup	curl /health
13	PHP (Laravel)	SSH + php-fpm + nginx	curl /health
14	Static Site	SSH + nginx	curl /
15	Docker-образ	Docker Compose	curl /health

Средний уровень

Общее требование: Развертывание с несколькими контейнерами (app + db + redis). Использование Docker Compose. Настроить проверку логов на наличие ошибок.

№	Язык	Стек (контейнеры)	Дополнительные требования
	Python	app + PostgreSQL + Redis	healthcheck для БД
	Python	app + MySQL +	Проверка логов на ошибки

		Memcached	
	Python	app + MongoDB + RabbitMQ	Ожидание готовности БД
	Node.js	app + PostgreSQL + Redis	docker-compose pull перед up
	Node.js	app + MySQL + Elasticsearch	Проверка индексов Elastic
	Node.js	app + MongoDB + MinIO	Smoke test на upload файла
	Go	app + PostgreSQL + Redis	Graceful shutdown
	Go	app + MySQL + Nginx	Nginx проксирование
	Java	app + PostgreSQL + Kafka	Проверка топиков Kafka
	Java	app + MySQL + Redis	Docker Compose profiles
	C	app + SQL Server + Redis	Проверка подключения к БД
	Ruby	app + PostgreSQL + Sidekiq	Проверка очередей
	PHP	app + MySQL + Nginx + PHP-FPM	Полный LEMP стек
	Python + Celery	app + Redis + Celery worker	Проверка Celery задач
	Специфика БПЛА	ROS 2 + Gazebo	Smoke test: запуск симуляции

Продвинутый уровень

Общее требование: Развертывание с Blue-Green или Canary стратегией на тестовом стенде. Автоматический откат при провале smoke test. Интеграция с окружениями CI/CD (environment).

№	Язык	Стратегия	Дополнительные требования
1	Python	Blue-Green (2 порта)	Переключение балансировщика
2	Python	Canary (10% трафика)	Проверка метрик ошибок
3	Python	A/B тестирование	Разные версии на разных портах
4	Node.js	Blue-Green + Nginx	Автоматический откат
5	Node.js	Canary + Prometheus	Сравнение метрик
6	Node.js	Feature flags + развертывание	Постепенное включение
7	Go	Blue-Green + Traefik	Автоматическое переключение
8	Go	Canary + Grafana	Визуализация метрик
9	Java	Blue-Green + Spring Cloud	Интеграция с Discovery Service
10	Java	Canary + Micrometer	Проверка метрик в Prometheus
11	Специфика БПЛА	Симулятор + реальный стенд	НITL-тестирование
12	Kubernetes	Helm + Argo CD	GitOps на тестовый кластер
13	Комплексный	Blue-Green + Smoke + Rollback	Полный цикл с откатом
14	Security-hardened	SSH + Bastion host	Развертывание через jump host
15	Cloud-native	AWS ECS + Fargate	Развертывание в cloud

Контрольные вопросы

1. Какие окружения (environments) используются в CI/CD и для чего?
2. В чем отличие развертывания через SSH от Docker Compose?
3. Как безопасно хранить SSH-ключи в CI/CD (GitLab/GitHub)?
4. Что такое smoke test и чем он отличается от полного регрессионного тестирования?
5. Как настроить healthcheck в Docker Compose?
6. Как проверить, что контейнер успешно запустился, через CI?
7. Как выполнить откат (rollback) при провале smoke test?
8. Что такое Blue-Green развертывание? Какие преимущества?
9. Как настроить окружение в GitLab CI с URL для отображения?
10. Как выполнить удаленную команду через SSH из CI/CD?
11. Как скопировать файлы на удаленный сервер из CI/CD?
12. Как проверить логи контейнера на наличие ошибок в CI?
13. Как настроить ожидание готовности сервиса перед smoke test?
14. В чем отличие Docker Compose от Kubernetes для тестового стенда?
15. Как настроить автоматический деплой только для ветки develop?

Требования к отчету

Отчет должен содержать следующие разделы:

1. Титульный лист с указанием ФИО студента, группы, варианта.
2. Цель и задачи работы.
3. Теоретическая часть:
 - Обоснование выбора способа развертывания (SSH / Docker Compose).
 - Схема развертывания.
4. Практическая часть:
 - Листинг конфигурации CI/CD (развертывание).

- Листинг docker-compose.yml (если используется).

- Листинг скрипта smoke test.

5. Результаты выполнения:

- Скриншоты этапа развертывания в CI.

- Скриншоты успешного smoke test.

- Скриншот запущенного приложения на тестовом стенде.

6. Анализ результатов:

- Время развертывания.

- Проблемы и их решение.

7. Ответы на контрольные вопросы (3-5 вопросов).

8. Вывод — достигнута ли цель, какие сложности возникли.

Критерии оценивания

Критерий	Базовый уровень	Средний уровень	Продвинутый уровень
Развертывание на тестовый стенд настроено	2 балла	2 балла	2 балла
Smoke test реализован	2 балла	2 балла	2 балла
Использованы безопасные SSH-ключи	1 балл	2 балла	2 балла
Docker Compose с несколькими контейнерами	0 баллов	2 балла	2 балла
Healthcheck в Docker Compose	0 баллов	1 балл	2 балла
Blue-Green / Canary развертывание	0 баллов	0 баллов	2 балла
Автоматический откат при провале	0 баллов	0 баллов	2 балла

Качество отчета и ответы на вопросы	2 балла	2 балла	2 балла

Шкала перевода баллов в оценку:

Оценка	Базовый уровень (макс. 7)	Средний уровень (макс. 11)	Продвинутый уровень (макс. 16)
Отлично (5)	6-7 баллов	10-11 баллов	14-16 баллов
Хорошо (4)	5 баллов	8-9 баллов	11-13 баллов
Удовлетворительно (3)	3-4 балла	6-7 баллов	8-10 баллов
Неудовлетворительно (2)	менее 3 баллов	менее 6 баллов	менее 8 баллов

Бонусные баллы (макс. +3):

- Настройка окружения с URL в GitLab - +1 балл.
- Автоматический откат при провале smoke test - +1 балл.
- Интеграция с мониторингом (Prometheus/Grafana) - +1 балл.

Лабораторная работа №11. Стратегия Rolling Update

Цель работы: Сформировать практические навыки реализации стратегии rolling update (обновление без остановки сервиса) с использованием оркестраторов (Docker Swarm / Kubernetes), настройки параметров обновления (maxSurge, maxUnavailable) и наблюдения за процессом обновления.

Теоретическое обоснование

1. Стратегии обновления приложений

Стратегия	Описание	Downtime	Скорость	Сложность
Recreate	Остановка всех старых подов, запуск новых	Есть (минуты)	Быстрая	Низкая
Rolling Update	Постепенная замена старых подов новыми	Нет	Средняя	Средняя
Blue-Green	Два окружения, переключение трафика	Нет	Мгновенная	Высокая
Canary	Постепенный перенос трафика на новую версию	Нет	Гибкая	Высокая

Rolling Update - стратегия, при которой новые версии приложения постепенно заменяют старые без остановки сервиса. В каждый момент времени часть трафика идёт на старую версию, часть - на новую.

2. Rolling Update в Kubernetes

Параметры rolling update в Kubernetes (Deployment):

Параметр	Описание	Значение по умолчанию
maxSurge	Максимальное количество дополнительных подов сверх желаемого количества	25%
maxUnavailable	Максимальное количество подов, которые могут быть недоступны во	25%

	время обновления	
minReadySeconds	Минимальное время, в течение которого под должен быть готов, прежде чем считаться доступным	0
progressDeadlineSeconds	Время, за которое обновление должно прогрессировать	600

Пример Deployment с rolling update:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
spec:
  replicas: 5
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 2          # может быть абсолютным или %
      maxUnavailable: 1   # может быть абсолютным или %
  minReadySeconds: 10
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
        - name: app
          image: myapp:v2
          readinessProbe:
            httpGet:
              path: /health
              port: 8080
            initialDelaySeconds: 5
            periodSeconds: 5
```


Процесс rolling update в Kubernetes (рисунок 1):

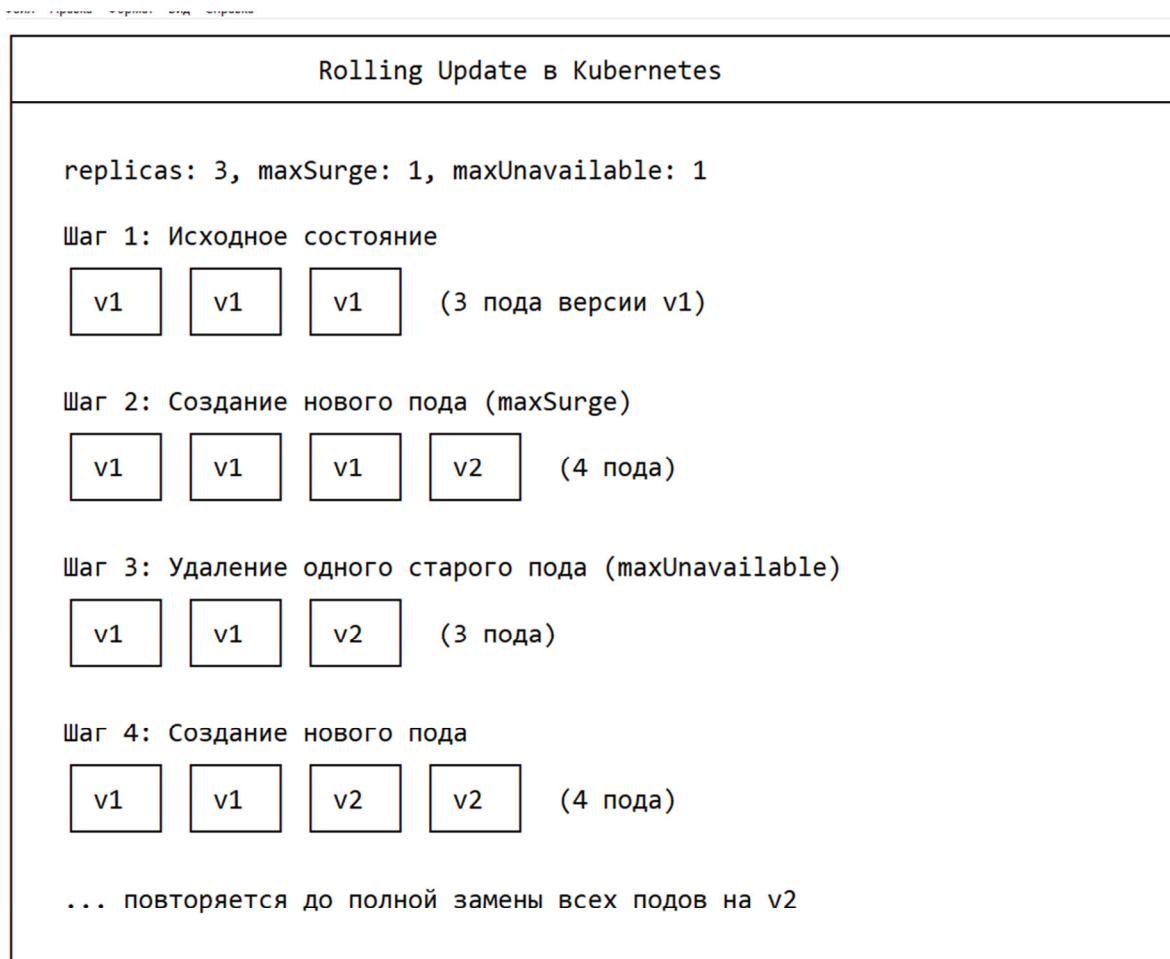


Рисунок 1

2.3. Rolling Update в Docker Swarm

Параметры rolling update в Docker Swarm (service):

Параметр	Описание	Команда
--update-parallelism	Количество контейнеров, обновляемых одновременно	docker service update --update-parallelism 2
--update-delay	Задержка между обновлениями групп	--update-delay 10s
--update-failure-action	Действие при ошибке (pause/continue/rollback)	--update-failure-action rollback

--update-monitor	Время мониторинга после обновления	--update-monitor 30s

Пример сервиса Docker Swarm с rolling update:

```
docker service create \
  --name myapp \
  --replicas 5 \
  --update-parallelism 2 \
  --update-delay 10s \
  --update-failure-action rollback \
  --update-monitor 30s \
  --publish 8080:8080 \
  myapp:v1
```

Обновление сервиса:

```
docker service update --image myapp:v2 myapp
```

4. Readiness Probe и Liveness Probe

Readiness Probe - определяет, готов ли контейнер принимать трафик. При неудаче контейнер исключается из балансировки.

Liveness Probe - определяет, жив ли контейнер. При неудаче контейнер перезапускается.

```
readinessProbe:
  httpGet:
    path: /health
    port: 8080
  initialDelaySeconds: 5
  periodSeconds: 5
  successThreshold: 1
  failureThreshold: 3
```

```
livenessProbe:
  httpGet:
    path: /health
    port: 8080
  initialDelaySeconds: 30
  periodSeconds: 10
  failureThreshold: 3
```

5. Автоматический откат (rollback)

Kubernetes:

```
Откат к предыдущей версии
#kubectl rollout undo deployment/myapp

# Откат к конкретной версии
kubectl rollout undo deployment/myapp --to-revision=2

# Просмотр истории
kubectl rollout history deployment/myapp
```

Docker Swarm:

```
# Откат к предыдущей версии
docker service rollback myapp
```

6. Интеграция с CI/CD

Пример пайплайна с rolling update в Kubernetes:

```
deploy_k8s:
  stage: deploy
  script:
    # Обновление образа в deployment
    - kubectl set image deployment/myapp
      app=${CI_REGISTRY_IMAGE}:${CI_COMMIT_SHORT_SHA}

    # Ожидание завершения обновления
```

```
- kubectl rollout status deployment/myapp --timeout=5m

# Smoke test
- curl -f https://myapp.example.com/health
environment:
  name: production
```

Примеры практического выполнения

1. Пример для Kubernetes (Minikube + kubectl)

Файл deployment.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
  labels:
    app: myapp
spec:
  replicas: 5
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
  minReadySeconds: 10
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
        - name: myapp
          image: myapp:latest
          ports:
            - containerPort: 8080
          readinessProbe:
            httpGet:
```

```

        path: /health
        port: 8080
        initialDelaySeconds: 5
        periodSeconds: 5
    livenessProbe:
        httpGet:
            path: /health
            port: 8080
            initialDelaySeconds: 30
            periodSeconds: 10
---
apiVersion: v1
kind: Service
metadata:
  name: myapp
spec:
  selector:
    app: myapp
  ports:
    - port: 80
      targetPort: 8080
  type: LoadBalancer

```

Файл .gitlab-ci.yml:

```

stages:
  - build
  - deploy

variables:
  IMAGE_TAG: $CI_REGISTRY_IMAGE:$CI_COMMIT_SHORT_SHA

build:
  stage: build
  script:
    - docker build -t $IMAGE_TAG .
    - docker push $IMAGE_TAG

deploy:
  stage: deploy
  script:

```

```
# Настройка kubectl
- kubectl config set-cluster k8s --server=$K8S_SERVER
- kubectl config set-credentials gitlab --
token=$K8S_TOKEN
- kubectl config set-context gitlab --cluster=k8s --
user=gitlab
- kubectl config use-context gitlab

# Обновление образа
- kubectl set image deployment/myapp myapp=$IMAGE_TAG

# Ожидание завершения rolling update
- kubectl rollout status deployment/myapp --timeout=5m

# Smoke test
- kubectl port-forward service/myapp 8080:80 &
- sleep 10
- curl -f http://localhost:8080/health
environment:
  name: production
```

2. Пример для Docker Swarm

Файл docker-compose.yml:

```
version: '3.8'

services:
  myapp:
    image: myapp:latest
    ports:
      - "8080:8080"
    deploy:
      replicas: 5
      update_config:
        parallelism: 2
        delay: 10s
        failure_action: rollback
        monitor: 30s
        order: start-first
      rollback_config:
        parallelism: 2
```

```
    delay: 5s
  healthcheck:
    test: ["CMD", "curl", "-f",
"http://localhost:8080/health"]
    interval: 10s
    timeout: 5s
    retries: 3
```

Файл .github/workflows/deploy-swarm.yml:

```
name: Deploy to Docker Swarm

on:
  push:
    tags:
      - 'v*'

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Deploy to Swarm
        env:
          DOCKER_HOST: ${ secrets.DOCKER_HOST }
          DOCKER_CERT_PATH: ${ secrets.DOCKER_CERT_PATH }
        run: |
          docker stack deploy -c docker-compose.yml myapp
          sleep 10
          curl -f http://swarm.example.com/health
```

3. Пример с мониторингом обновления (kubectl rollout status)

```
#!/bin/bash

# Обновление образа
kubectl set image deployment/myapp myapp=myapp:v2

# Ожидание с выводом прогресса
```

```
echo "Waiting for rollout to complete..."
kubectl rollout status deployment/myapp --watch --
timeout=300s

# Проверка результата
if [ $? -eq 0 ]; then
    echo "Rollout completed successfully"
else
    echo "Rollout failed, rolling back..."
    kubectl rollout undo deployment/myapp
    exit 1
fi
```

Методические указания выполнения практической части работы

Задачи работы

1. Настройка оркестратора для rolling update:

- Выбрать оркестратор (Docker Swarm или Kubernetes).
- Развернуть тестовый кластер (локальный или удалённый).
- Настроить сервис/деплоймент с поддержкой rolling update.

2. Конфигурация параметров обновления:

- Настроить maxSurge и maxUnavailable для контроля скорости обновления.

- Настроить minReadySeconds и failureThreshold для проверки

готовности подов.

3. Модификация CI/CD пайплайна:

- Добавить этап обновления образа в оркестраторе.
- Настроить последовательное обновление экземпляров.

4. Наблюдение и верификация:

- Отслеживать процесс обновления в реальном времени (kubectl rollout status / docker service ps).

- Выполнить smoke-тесты во время обновления.
- Настроить автоматический откат при ошибке.

5. Оформление отчёта:

- Зафиксировать конфигурацию оркестратора и пайплайна.
- Представить скриншоты процесса обновления.

Варианты заданий

Базовый уровень

Общее требование: Развернуть приложение в оркестраторе (K8s / Swarm). Настроить rolling update с базовыми параметрами.

Продемонстрировать обновление без остановки сервиса.

№	Оркестратор	Язык/Платформа	Параметры
1	Minikube (K8s)	Python (Flask)	replicas=3, maxSurge=1, maxUnavailable=0
2	Minikube (K8s)	Node.js (Express)	replicas=3, maxSurge=1, maxUnavailable=1
3	Minikube (K8s)	Go (net/http)	replicas=5, maxSurge=2, maxUnavailable=1
4	Minikube (K8s)	Java (Spring Boot)	replicas=2, maxSurge=1, maxUnavailable=0
5	Minikube (K8s)	Ruby (Sinatra)	replicas=4, maxSurge=1, maxUnavailable=1
6	Docker Swarm	Python (Flask)	replicas=3, parallelism=1, delay=5s
7	Docker Swarm	Node.js (Express)	replicas=3, parallelism=2, delay=10s
8	Docker Swarm	Go (Gin)	replicas=5, parallelism=2, delay=5s
9	Docker Swarm	PHP (Laravel)	replicas=2, parallelism=1, delay=10s

10	Kind (K8s)	Python (FastAPI)	replicas=3, maxSurge=1, maxUnavailable=1
11	Kind (K8s)	Node.js (NestJS)	replicas=4, maxSurge=2, maxUnavailable=1
12	Kind (K8s)	Go (Echo)	replicas=3, maxSurge=1, maxUnavailable=0
13	K3s	Python (Django)	replicas=3, maxSurge=1, maxUnavailable=1
14	K3s	Node.js (Next.js)	replicas=2, maxSurge=1, maxUnavailable=0
15	Docker Swarm	Static Site (nginx)	replicas=3, parallelism=2, delay=5s

Средний уровень

Общее требование: Настроить readinessProbe/livenessProbe. Добавить minReadySeconds. Реализовать автоматический откат при ошибке readinessProbe. Встроить rolling update в CI/CD пайплайн.

№	Оркестратор	Язык	Дополнительные требования
1	Minikube	Python	Readiness probe + откат при ошибке
2	Minikube	Node.js	Liveness probe + minReadySeconds=15
3	Minikube	Go	Оба probe + failureThreshold=3
4	Minikube	Java	Spring Boot Actuator для healthcheck
5	Kind	Python	Откат через kubectl rollout undo
6	Kind	Node.js	CI/CD с ожиданием rollout status
7	Docker Swarm	Python	--update-failure-action rollback
8	Docker Swarm	Node.js	--update-monitor 30s

9	Docker Swarm	Go	Healthcheck в Dockerfile
10	Minikube	Python + PostgreSQL	Проверка связи с БД в readinessProbe
11	Minikube	Node.js + Redis	Проверка Redis в readinessProbe
12	Kind	Go + MySQL	initContainer для миграций
13	K3s	Python	Prometheus метрики во время обновления
14	K3s	Node.js	Smoke test во время обновления
15	Docker Swarm	Go	--update-order start-first

Продвинутый уровень

Общее требование: Реализовать Blue-Green через два Deployment с переключением Service. Настроить canary deployment с процентным распределением трафика (Istio/Argo Rollouts). Интегрировать анализ метрик для автоматического отката.

№	Оркестратор	Стратегия	Дополнительные требования
1	Kubernetes	Blue-Green	Два Deployment + переключение Service
2	Kubernetes	Canary (Istio)	10% → 50% → 100% трафика
3	Kubernetes	Canary (Argo Rollouts)	Анализ метрик Prometheus
4	Kubernetes	A/B тестирование	Разные версии по заголовкам
5	Kubernetes	Shadow deployment	Трафик дублируется на новую версию
6	Kubernetes	Blue-Green +	Автоматический откат при провале

		Smoke	
7	Kubernetes	Canary + Flagger	Автоматический анализ метрик
8	Kubernetes	Progressive Delivery	Анализ логов ошибок
9	Kubernetes + Istio	Canary + Kiali	Визуализация трафика
10	Kubernetes + Argo	Rollouts + AnalysisTemplate	Кастомные метрики
11	Специфика БПЛА	ROS 2 + K8s	Rolling update для ROS-нод
12	Edge	K3s + ARM	Rolling update на Raspberry Pi
13	Multi-cloud	EKS + GKE	Canary между облаками
14	GitOps	Argo CD + Rollouts	Управление через Git
15	Full-stack	K8s + Prometheus + Grafana	Дашборд мониторинга обновления

Контрольные вопросы

1. Что такое rolling update и для чего он используется?
2. В чем отличие rolling update от recreate?
3. Какие параметры rolling update существуют в Kubernetes?
4. Что означают параметры maxSurge и maxUnavailable?
5. Как настроить rolling update в Docker Swarm?
6. Что такое readinessProbe и зачем он нужен при rolling update?
7. В чем отличие readinessProbe от livenessProbe?
8. Как отследить процесс rolling update в Kubernetes?
9. Как выполнить автоматический откат при ошибке?

10. Что такое minReadySeconds и как он влияет на обновление?
11. Как проверить, что сервис не был недоступен во время обновления?
12. Какую стратегию выбрать для zero-downtime deployment?
13. Как настроить rolling update через CI/CD пайплайн?
14. В чем отличие rolling update от blue-green deployment?
15. Как настроить canary deployment с процентным распределением трафика?

Требования к отчету

Отчет должен содержать следующие разделы:

1. Титульный лист с указанием ФИО студента, группы, варианта.
2. Цель и задачи работы.
3. Теоретическая часть:
 - Описание стратегии rolling update.
 - Сравнение с другими стратегиями.
 - Обоснование выбора параметров.
4. Практическая часть:
 - Листинг манифеста Kubernetes (deployment.yaml).
 - Листинг docker-compose.yml (для Swarm).
 - Листинг CI/CD конфигурации.
5. Результаты выполнения:
 - Скриншоты процесса rolling update (kubectl get pods -w).
 - Скриншоты успешного обновления в CI/CD.
 - Скриншоты smoke test во время обновления.
6. Анализ результатов:
 - Таблица времени обновления.
 - Наблюдаемое поведение при разных параметрах.

7. Ответы на контрольные вопросы (3-5 вопросов).
8. Вывод - достигнута ли цель.

Критерии оценивания

Критерий	Базовый уровень	Средний уровень	Продвинутый уровень
Rolling update настроен	2 балла	2 балла	2 балла
Обновление без остановки сервиса	2 балла	2 балла	2 балла
Readiness/liveness probe настроены	0 баллов	2 балла	2 балла
Параметры maxSurge/maxUnavailable	1 балл	2 балла	2 балла
Интеграция в CI/CD пайплайн	0 баллов	1 балл	2 балла
Автоматический откат при ошибке	0 баллов	0 баллов	2 балла
Canary / Blue-Green реализованы	0 баллов	0 баллов	2 балла
Качество отчета и ответы	2 балла	2 балла	2 балла

Шкала перевода баллов:

Оценка	Базовый (макс. 7)	Средний (макс. 11)	Продвинутый (макс. 16)
Отлично (5)	6-7	10-11	14-16
Хорошо (4)	5	8-9	11-13
Удовлетворительно (3)	3-4	6-7	8-10
Неудовлетворительно (2)	<3	<6	<8

Лабораторная работа №12. Стратегия Blue-Green Deployment

Цель работы: Сформировать практические навыки реализации стратегии Blue-Green Deployment (развертывание с мгновенным переключением окружений), обеспечивающей нулевое время простоя (zero-downtime) при обновлении приложения.

Теоретическое обоснование

1. Концепция Blue-Green Deployment

Blue-Green Deployment - это стратегия развертывания, при которой поддерживаются два идентичных производственных окружения: Blue (текущая активная версия) и Green (новая версия, готовящаяся к релизу) (рисунок 1).

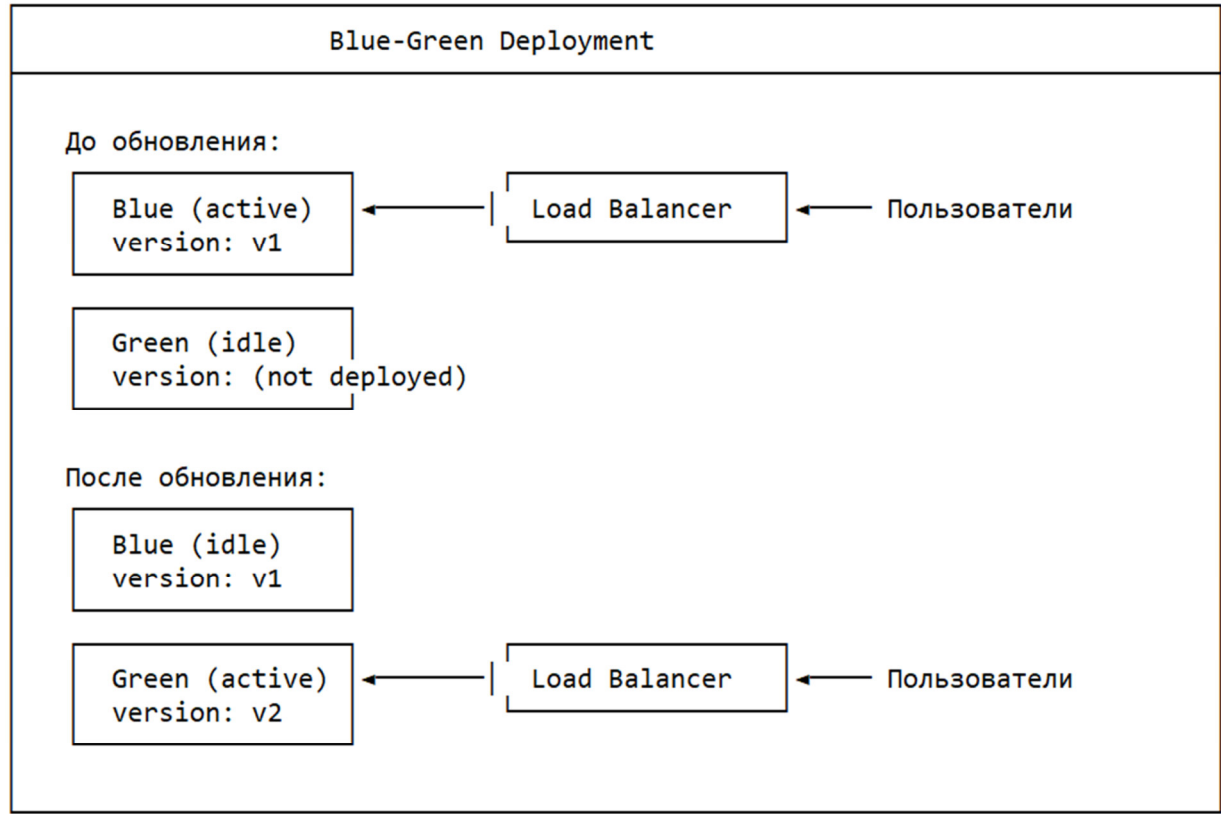


Рисунок 1

2. Преимущества и недостатки Blue-Green

Преимущество	Описание

Zero-downtime	Переключение происходит мгновенно (изменение балансировщика)
Мгновенный откат	Достаточно переключить балансировщик обратно
Изоляция окружений	Новая версия не влияет на работающую
Возможность тестирования	Green окружение можно тестировать до переключения

Недостаток	Описание
Удвоение ресурсов	Требуется вдвое больше вычислительных мощностей
Синхронизация данных	Проблема миграций БД (требуется совместимость)
Сложность	Требуется больше автоматизации

3. Реализация в Kubernetes

Структура ресурсов Kubernetes для Blue-Green:

```
# blue-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-blue
  labels:
    app: myapp
    version: blue
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
      version: blue
  template:
    metadata:
      labels:
        app: myapp
        version: blue
    spec:
      containers:
```



```

- name: app
  image: myapp:v1
---
# green-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-green
  labels:
    app: myapp
    version: green
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
      version: green
  template:
    metadata:
      labels:
        app: myapp
        version: green
    spec:
      containers:
        - name: app
          image: myapp:v2
---
# service.yaml (селектор переключается между blue/green)
apiVersion: v1
kind: Service
metadata:
  name: myapp
spec:
  selector:
    app: myapp
    version: blue # ← переключаем на green при релизе
  ports:
    - port: 80
      targetPort: 8080

```

4. Переключение трафика через Service

Способы переключения:

Способ	Команда	Время переключения

kubectl patch service	kubectl patch service myapp -p '{"spec":{"selector":{"version":"green"}}}'	<1 секунда
kubectl edit service	Интерактивное редактирование	Зависит от оператора
Helm upgrade	helm upgrade --set active=green	1-5 секунд
Ingress weight	nginx.ingress.kubernetes.io/canary-weight	Постепенное

5. Реализация в Docker Swarm

```
# Создание двух стэков (blue и green)
docker stack deploy -c docker-compose.blue.yml myapp_blue
docker stack deploy -c docker-compose.green.yml myapp_green

# Переключение балансировщика (через обновление прокси)
docker service update --label-add
traefik.http.routers.myapp.rule="Host(`myapp.com`)" \
--label-add traefik.http.services.myapp.loadbalancer.server.port=80 \
myapp_green_web
```

6. Проблема синхронизации баз данных

Ключевая проблема Blue-Green: миграции БД должны быть обратно совместимы.

Подходы к решению:

Подход	Описание	Риски
Миграции с поддержкой обратной совместимости	Новая версия работает со старой схемой БД	Сложность
Миграции только вперёд	После переключения нельзя откатиться	Откат невозможен
БД вне окружений	Общая БД для Blue и Green	Миграции должны быть совместимы
Две БД + репликация	Blue→Green	Высокая сложность

	репликация, после переключения - наоборот	

7. Интеграция с CI/CD пайплайном

```
# GitLab CI pipeline для Blue-Green
stages:
  - deploy-green
  - smoke-test
  - switch-traffic
  - cleanup-blue

variables:
  ACTIVE_COLOR: blue # или green

deploy-green:
  stage: deploy-green
  script:
    # Определяем, какой цвет неактивен
    - if [ "$ACTIVE_COLOR" == "blue" ]; then INACTIVE="green"; else
INACTIVE="blue"; fi
    # Деплой в неактивное окружение
    - kubectl set image deployment/myapp-$INACTIVE
app=$CI_REGISTRY_IMAGE:$CI_COMMIT_SHA
    - kubectl rollout status deployment/myapp-$INACTIVE

smoke-test-green:
  stage: smoke-test
  script:
    - curl -f http://myapp-$INACTIVE.example.com/health

switch-traffic:
  stage: switch-traffic
  script:
    - kubectl patch service myapp -p
'{"spec":{"selector":{"version":"$INACTIVE}}}'
    - echo "ACTIVE_COLOR=$INACTIVE" >> .env

cleanup-blue:
  stage: cleanup-blue
  script:
    - kubectl scale deployment/myapp-$ACTIVE_COLOR --replicas=1
```

Примеры практического выполнения

1. Пример для Kubernetes (Minikube) с переключением Service

Файл blue-green.sh:

```
#!/bin/bash

# Конфигурация
APP_NAME="myapp"
NAMESPACE="default"
ACTIVE_COLOR="${1:-blue}" # blue или green

# Определение неактивного цвета
if [ "$ACTIVE_COLOR" == "blue" ]; then
    INACTIVE_COLOR="green"
else
    INACTIVE_COLOR="blue"
fi

echo "Active: $ACTIVE_COLOR, Inactive: $INACTIVE_COLOR"

# Деплой в неактивное окружение
echo "Deploying to $INACTIVE_COLOR environment..."
kubectl apply -f ${APP_NAME}-${INACTIVE_COLOR}-deployment.yaml
kubectl rollout status deployment/${APP_NAME}-${INACTIVE_COLOR} --timeout=5m

# Smoke-тест неактивного окружения
echo "Smoke testing $INACTIVE_COLOR environment..."
INACTIVE_URL="http://${APP_NAME}-${INACTIVE_COLOR}.local"
curl -f $INACTIVE_URL/health || exit 1

# Переключение трафика
echo "Switching traffic to $INACTIVE_COLOR..."
kubectl patch service $APP_NAME -p
'{"spec":{"selector":{"version":"$INACTIVE_COLOR}}}'

# Smoke-тест активного окружения
echo "Smoke testing new active environment..."
curl -f http://$APP_NAME.local/health || exit 1

echo "Blue-Green deployment completed. Active environment: $INACTIVE_COLOR"
```

Файл myapp-blue-deployment.yaml:

```
apiVersion: apps/v1
```

```

kind: Deployment
metadata:
  name: myapp-blue
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
      version: blue
  template:
    metadata:
      labels:
        app: myapp
        version: blue
    spec:
      containers:
      - name: app
        image: myapp:v1
        ports:
        - containerPort: 8080
        readinessProbe:
          httpGet:
            path: /health
            port: 8080

```

Файл myapp-service.yaml:

```

apiVersion: v1
kind: Service
metadata:
  name: myapp
spec:
  selector:
    app: myapp
    version: blue # изначально активен blue
  ports:
  - port: 80
    targetPort: 8080
  type: LoadBalancer

```

2. Пример для GitHub Actions + Kubernetes

Файл .github/workflows/blue-green.yml:

```

name: Blue-Green Deployment
on:

```

```

push:
  tags:
    - 'v*'

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Configure kubectl
        uses: azure/setup-kubectl@v3
        with:
          kubeconfig: ${ secrets.KUBECONFIG }

      - name: Determine colors
        id: colors
        run: |
          CURRENT=$(kubectl get service myapp -o
jsonpath='{.spec.selector.version}')
          if [ "$CURRENT" == "blue" ]; then
            echo "inactive=green" >> $GITHUB_OUTPUT
          else
            echo "inactive=blue" >> $GITHUB_OUTPUT
          fi

      - name: Deploy to inactive environment
        run: |
          envsubst < k8s/deployment.yaml | kubectl apply -f -
        env:
          COLOR: ${ steps.colors.outputs.inactive }
          IMAGE_TAG: ${ github.ref_name }

      - name: Wait for rollout
        run: |
          kubectl rollout status deployment/myapp-${ steps.colors.outputs.inactive } --timeout=5m

      - name: Smoke test inactive
        run: |
          kubectl port-forward service/myapp-${ steps.colors.outputs.inactive }
8080:80 &
          sleep 10
          curl -f http://localhost:8080/health

      - name: Switch traffic
        run: |
          kubectl patch service myapp --type='json' \
            -p='[{"op": "replace", "path": "/spec/selector/version", "value":
"${ steps.colors.outputs.inactive }"}]'

```

```

- name: Verify switch
  run: |
    sleep 5
    curl -f http://myapp.example.com/health

- name: Scale down old environment
  run: |
    kubectl scale deployment/myapp-`${ steps.colors.outputs.active }` --
replicas=1

```

3. Пример для Docker Swarm + Traefik

Файл docker-compose.blue.yml:

```

version: '3.8'

services:
  web:
    image: myapp:v1
    deploy:
      replicas: 3
      labels:
        - "traefik.enable=true"
        - "traefik.http.routers.myapp.rule=Host(`myapp.example.com`)"
        - "traefik.http.services.myapp.loadbalancer.server.port=80"
    networks:
      - traefik

networks:
  traefik:
    external: true

```

Файл switch.sh:

```

#!/bin/bash

# Определение активного стэка
if docker stack ls | grep -q "myapp_blue"; then
  ACTIVE="blue"
  INACTIVE="green"
else
  ACTIVE="green"
  INACTIVE="blue"
fi

```

```
echo "Active: $ACTIVE, Inactive: $INACTIVE"

# Деплой в неактивный стэк
docker stack deploy -c docker-compose.${INACTIVE}.yml myapp_${INACTIVE}

# Ожидание готовности
sleep 10

# Переключение Traefik (обновление labels)
docker service update --label-add
traefik.http.routers.myapp.rule="Host(`myapp.example.com`)" \
  --label-add traefik.http.services.myapp.loadbalancer.server.port=80 \
  myapp_${INACTIVE}_web

# Удаление старого стэка
docker stack rm myapp_${ACTIVE}
```

Методические указания выполнения практической части работы

Задачи работы

1. Настройка двух идентичных окружений (Blue и Green):
 - Создать два независимых набора ресурсов (Deployment/Service) с разными метками.
 - Настроить общий балансировщик трафика (LoadBalancer / Ingress).
2. Модификация CI/CD пайплайна:
 - Автоматизировать развертывание новой версии в неактивное окружение.
 - Реализовать переключение трафика между окружениями.
 - Добавить smoke-тестирование после переключения.
3. Реализация механизма отката:
 - Автоматический откат при провале smoke-тестов.
 - Ручной откат к предыдущей версии.
4. Наблюдение и верификация:
 - Отслеживать состояние окружений в реальном времени.
 - Проверить отсутствие потери трафика во время переключения.

Варианты заданий

Базовый уровень

Общее требование: Создать два окружения (Blue и Green). Настроить общий балансировщик (Service/Ingress/LoadBalancer). Реализовать переключение трафика между окружениями.

№	Оркестратор	Язык/Платформа	Способ переключения
1	Minikube	Python (Flask)	kubectl patch service
2	Minikube	Node.js (Express)	kubectl patch service
3	Minikube	Go (net/http)	kubectl edit service
4	Minikube	Java (Spring Boot)	kubectl patch service
5	Minikube	Ruby (Sinatra)	kubectl apply -f service.yaml
6	Kind	Python (FastAPI)	kubectl patch service
7	Kind	Node.js (NestJS)	kubectl patch service
8	Kind	Go (Gin)	kubectl patch service
9	K3s	Python (Django)	kubectl patch service
10	K3s	Node.js (Next.js)	kubectl patch service
11	Docker Swarm	Python (Flask)	Traefik label update
12	Docker Swarm	Node.js (Express)	Traefik label update
13	Docker Swarm	Go (Echo)	Nginx reload
14	Docker Swarm	PHP (Laravel)	HAProxy reload
15	Minikube + Ingress	Static Site (nginx)	Ingress переключение

Средний уровень

Общее требование: Автоматизировать Blue-Green в CI/CD. Добавить smoke-тестирование после переключения. Реализовать автоматический откат при ошибке.

№	Оркестратор	Язык	Дополнительные требования
	Minikube	Python	CI/CD + smoke test + откат
	Minikube	Node.js	CI/CD + проверка логов
	Minikube	Go	CI/CD + автоматический откат
	Kind	Python	GitHub Actions + smoke test
	Kind	Node.js	GitLab CI + откат
	K3s	Go	Argo CD + Blue-Green
	Minikube	Python + PostgreSQL	Миграции БД (совместимость)
	Minikube	Node.js + MongoDB	Проверка данных после переключения
	Minikube	Go + Redis	Проверка кэша
	Docker Swarm	Python	GitHub Actions + откат
	Docker Swarm	Node.js	GitLab CI + smoke test
	Docker Swarm	Go	Автоматический откат при 500 ошибках
	Minikube + Ingress	Python	Переключение через Ingress rules
	Minikube + Nginx	Node.js	Переключение через Nginx upstream
	Minikube	Java	Actuator + healthcheck

Продвинутый уровень

Общее требование: Реализовать Blue-Green с управлением через GitOps (Argo CD). Настроить синхронизацию БД между окружениями (репликация). Интегрировать анализ метрик для автоматического отката.

№	Оркестратор	Стратегия	Дополнительные требования
	Kubernetes	Argo CD + Blue-Green	Управление через Git
	Kubernetes	Flux + Blue-Green	GitOps с автоматической синхронизацией
	Kubernetes	Blue-Green + Istio	Переключение через VirtualService
	Kubernetes	Blue-Green + Replication	Репликация БД blue→green
	Kubernetes	Blue-Green + Prometheus	Автооткат по метрикам ошибок
	Kubernetes	Blue-Green + Flagger	Анализ метрик и автооткат
	Kubernetes	Blue-Green + Kiali	Визуализация трафика
	Kubernetes + AWS	Blue-Green + RDS	Переключение Route53
	Kubernetes + GCP	Blue-Green + Cloud SQL	Переключение Cloud Load Balancer
	Docker Swarm	Blue-Green + Consul	Service discovery переключение
	Специфика БПЛА	ROS 2 + K8s	Blue-Green для ROS Master
	Edge	K3s + ARM	Blue-Green на Raspberry Pi кластере
	Multi-cloud	EKS + GKE	Blue-Green между облаками
	Security-hardened	Istio + mTLS	Безопасное переключение
	Full-stack	K8s +	Дашборд состояния Blue-Green

		Prometheus + Grafana	

Контрольные вопросы

1. Что такое Blue-Green Deployment? В чем его отличие от Rolling Update?
2. Какие преимущества даёт Blue-Green перед другими стратегиями?
3. Каков главный недостаток Blue-Green с точки зрения ресурсов?
4. Как переключить трафик между окружениями в Kubernetes?
5. Как решить проблему миграций баз данных при Blue-Green?
6. Что произойдёт, если новая версия несовместима со старой схемой БД?
7. Как выполнить откат к предыдущей версии в Blue-Green?
8. Как настроить Blue-Green в Docker Swarm?
9. Как интегрировать Blue-Green в CI/CD пайплайн?
10. Как проверить, что переключение прошло без потери трафика?
11. Что такое «гидридный подход» (Blue-Green + Canary)?
12. Как автоматизировать откат при провале smoke-тестов?
13. Какую роль играет Ingress в Blue-Green deployment?
14. Как настроить Blue-Green через GitOps (Argo CD)?
15. Какие метрики нужно отслеживать для принятия решения об откате?

Требования к отчету

Отчет должен содержать:

1. Титульный лист (ФИО, группа, вариант).
2. Цель и задачи работы.
3. Теоретическая часть:

- Описание стратегии Blue-Green.
- Сравнение с Rolling Update.
- Обоснование выбора способа переключения.

4. Практическая часть:

- Листинг манифестов Kubernetes (blue, green, service).
- Листинг CI/CD конфигурации.
- Листинг скриптов переключения.

5. Результаты выполнения:

- Скриншоты окружений до переключения.
- Скриншоты процесса переключения.
- Скриншоты smoke-тестов.
- Скриншоты отката (если применимо).

6. Анализ результатов:

- Время переключения.
- Проверка zero-downtime (логи, нагрузка).

7. Ответы на контрольные вопросы (3-5).

8. Вывод.

Критерии оценивания

Критерий	Базовый уровень	Средний уровень	Продвинутый уровень
Два окружения (Blue/Green) созданы	2 балла	2 балла	2 балла
Переключение трафика реализовано	2 балла	2 балла	2 балла
Zero-downtime подтверждён	1 балл	2 балла	2 балла
Интеграция в CI/CD пайплайн	0 баллов	2 балла	2 балла
Smoke-тестирование после переключения	0 баллов	1 балла	2 балла
Автоматический откат при ошибке	0 баллов	0 баллов	2 балла
Решение проблемы миграций БД	0 баллов	0 баллов	2 балла

Качество отчета и ответы	2 балла	2 балла	2 балла

Шкала перевода баллов:

Оценка	Базовый (макс. 7)	Средний (макс. 11)	Продвинутый (макс. 16)
5	6-7	10-11	14-16
4	5	8-9	11-13
3	3-4	6-7	8-10
2	<3	<6	<8

Лабораторная работа №13. Поэтапное развертывание (Canary Deployment) и управление функциями (Feature Toggles)

Цель работы: Сформировать у магистранта практические навыки реализации стратегии canary-развертывания (поэтапное развертывание на часть пользователей) и интеграции системы feature flags (переключателей функций) для управления доступом к новым возможностям приложения без повторного развертывания.

Теоретическое обоснование

2.1. Canary Deployment: концепция и преимущества

Canary Deployment - это стратегия развертывания, при которой новая версия приложения постепенно получает всё большую долю трафика, начиная с небольшого процента пользователей (обычно 5-10%). Название происходит от практики использования канареек в угольных шахтах для обнаружения токсичных газов.

Ключевые принципы:

Принцип	Описание
Малый blast radius	Ошибки в новой версии влияют только на небольшой процент пользователей
Поэтапное увеличение	Трафик увеличивается постепенно (10% → 25% → 50% → 100%)
Автоматический откат	При ухудшении метрик canary автоматически отключается
Валидация в production	Новая версия тестируется на реальном трафике

Сравнение с другими стратегиями:

Стратегия	Риск	Ресурсы	Rollback	Требования
Rolling Update	Средний	0% доп.	Медленный	Только K8s
Blue-Green	Низкий	+100%	Мгновенный	Двойные ресурсы
Canary	Минимальный	+10-20%	Мгновенный	Service Mesh / Ingress

2. Canary с Ingress Controller (Nginx)

Nginx Ingress Controller поддерживает canary-развертывание через специальные аннотации :

Аннотация	Описание
nginx.ingress.kubernetes.io/canary: "true"	Включает canary-режим
nginx.ingress.kubernetes.io/canary-	Процент трафика на canary (0-100)

weight	
nginx.ingress.kubernetes.io/canary-by-header	Маршрутизация по HTTP-заголовку
nginx.ingress.kubernetes.io/canary-by-cookie	Маршрутизация по cookie

Приоритет аннотаций: canary-by-header > canary-by-cookie > canary-weight

Пример конфигурации canary Ingress:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: canary-ingress
  annotations:
    nginx.ingress.kubernetes.io/canary: "true"
    nginx.ingress.kubernetes.io/canary-weight: "10"
spec:
  ingressClassName: nginx
  rules:
  - host: myapp.example.com
    http:
      paths:
      - path: /
        backend:
          service:
            name: myapp-canary
            port:
              number: 80
```

3. Canary с Istio VirtualService

Istio предоставляет более мощные возможности для canary-развертывания :

```
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
```



```

    name: myapp
  spec:
    hosts:
    - myapp
    http:
    - route:
        - destination:
            host: myapp-stable
            weight: 90
        - destination:
            host: myapp-canary
            weight: 10

```

Возможности Istio:

- Маршрутизация на основе HTTP-заголовков и cookie
- Постепенное изменение весов без перезагрузки
- Интеграция с Prometheus для метрик

2.4. Автоматический canary-анализ с Argo Rollouts

Argo Rollouts обеспечивает прогрессивную доставку с автоматическим анализом метрик :

```

apiVersion: argoproj.io/v1alpha1
kind: Rollout
metadata:
  name: myapp
spec:
  replicas: 10
  strategy:
    canary:
      canaryService: myapp-canary
      stableService: myapp-stable
      trafficRouting:
        istio:
          virtualService:
            name: myapp
            routes:
            - primary
  steps:
  - setWeight: 10

```

```

- pause: {duration: 5m}
- setWeight: 20
- pause: {duration: 5m}
- setWeight: 50
- pause: {duration: 10m}
- setWeight: 100
analysis:
  templates:
  - templateName: success-rate
  startingStep: 1
  args:
  - name: service-name
    value: myapp-canary.default.svc.cluster.local

```

AnalysisTemplate для проверки метрик:

```

apiVersion: argoproj.io/v1alpha1
kind: AnalysisTemplate
metadata:
  name: success-rate
spec:
  metrics:
  - name: success-rate
    interval: 30s
    successCondition: result[0] >= 0.99 # 99% успешных запросов
    failureLimit: 3
    provider:
      prometheus:
        address: http://prometheus.monitoring:9090
        query: |
          sum(rate(
            istio_requests_total{destination_service=~"{{args.service-
name}}",response_code!~"5.*"}[5m]
          )) /
          sum(rate(
            istio_requests_total{destination_service=~"{{args.service-
name}}}")[5m]
          ))

```

5. Feature Flags: концепция и инструменты

Feature Flag (переключатель функций) - механизм, позволяющий включать или выключать функциональность приложения без повторного развертывания .

Основные возможности:

Функция	Описание
Процентный rollout	Постепенное включение для % пользователей
Targeting	Включение для конкретных пользователей/групп
Multivariate flags	Несколько вариантов (А/В тестирование)
Remote configuration	Изменение поведения без кода

Популярные инструменты feature flags:

Инструмент	Лицензия	Особенности
LaunchDarkly	Коммерческая	Мощные SDK, аналитика, А/В тесты
Unleash	Open Source	Локальная оценка, privacy-first
Flagsmith	Open Source	Удалённая конфигурация
PostHog	Open Source	Встроенная аналитика и А/В тесты
GrowthBook	Open Source	Warehouse-native, эксперименты

6. Пример feature flag в коде (Node.js с LaunchDarkly)

```
// Инициализация клиента LaunchDarkly
const LaunchDarkly = require('launchdarkly-node-server-sdk');
const client = LaunchDarkly.init(process.env.LD_SDK_KEY);

// Использование feature flag
app.get('/api/feature', async (req, res) => {
  const user = { key: req.user.id };

  // Проверка feature flag
  const showNewFeature = await client.variation(
    'new-checkout-flow',
    user,
    false // значение по умолчанию
  );
};
```

```

if (showNewFeature) {
  res.json({ feature: 'new-checkout', version: 'v2' });
} else {
  res.json({ feature: 'legacy-checkout', version: 'v1' });
}
});

```

7. Интеграция canary и feature flags

Почему стоит комбинировать:

Сценарий	Только canary	Canary + Feature Flags
Тестирование на внутренних пользователях	Требуется отдельный кластер	Маршрутизация по заголовку + флаг
Постепенное включение функции	Только на уровне трафика	Два уровня контроля
A/B тестирование	Ограничено	Полноценное
Откат при ошибке	Автоматический (по метрикам)	Мгновенный (через флаг)

Примеры практического выполнения

1. Пример: canary с Nginx Ingress (базовый)

Файл stable-ingress.yaml:

```

apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: myapp-stable
spec:
  ingressClassName: nginx
  rules:
  - host: myapp.example.com
    http:
      paths:
      - path: /

```

```
    backend:
      service:
        name: myapp-stable
        port:
          number: 80
```

Файл canary-ingress.yaml:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: myapp-canary
  annotations:
    nginx.ingress.kubernetes.io/canary: "true"
    nginx.ingress.kubernetes.io/canary-weight: "10"
spec:
  ingressClassName: nginx
  rules:
  - host: myapp.example.com
    http:
      paths:
      - path: /
        backend:
          service:
            name: myapp-canary
            port:
              number: 80
```

Файл .gitlab-ci.yml (обновление canary):

```
stages:
  - deploy-canary
  - analyze
  - promote

deploy-canary:
  stage: deploy-canary
  script:
    - kubectl set image deployment/myapp-canary
myapp=$CI_REGISTRY_IMAGE:$CI_COMMIT_SHA
    - kubectl rollout status deployment/myapp-canary
  environment:
    name: canary

analyze:
  stage: analyze
```

```

script:
- |
    # Мониторинг ошибок в canary
    ERROR_RATE=$(curl -s
"http://prometheus:9090/api/v1/query?query=rate(http_requests_total{app=myapp-
canary,status=~'5..'}[5m])" | jq '.data.result[0].value[1]')
    if (( $(echo "$ERROR_RATE > 0.01" | bc -l) )); then
        echo "Error rate too high: $ERROR_RATE"
        exit 1
    fi
after_script:
- |
    if [ $CI_JOB_STATUS == "failed" ]; then
        kubectl delete ingress myapp-canary || true
    fi

promote:
  stage: promote
  script:
    - kubectl patch ingress myapp-canary -p
'{"metadata":{"annotations":{"nginx.ingress.kubernetes.io/canary-
weight":"100"}}}'
  when: manual

```

Пример: canary с Argo Rollouts (продвинутый)

Файл rollout.yaml:

```

apiVersion: argoproj.io/v1alpha1
kind: Rollout
metadata:
  name: myapp
spec:
  replicas: 10
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
        - name: myapp
          image: myapp:stable
          ports:
            - containerPort: 8080

```

```

    readinessProbe:
      httpGet:
        path: /health
        port: 8080
  strategy:
    canary:
      canaryService: myapp-canary
      stableService: myapp-stable
      trafficRouting:
        nginx:
          stableIngress: myapp-stable
      steps:
        - setWeight: 10
        - pause: {duration: 2m}
        - setWeight: 25
        - pause: {duration: 2m}
        - setWeight: 50
        - pause: {duration: 5m}
        - setWeight: 100
      analysis:
        templates:
          - templateName: error-rate
        startingStep: 1

```

Пример: feature flag с Unleash (self-hosted)

Файл unleash-config.js:

3.4. Пример: canary с заголовками (A/B тестирование)

```

const { initialize } = require('unleash-client');

const unleash = initialize({
  url: 'http://unleash.example.com/api/',
  appName: 'myapp',
  instanceId: 'production-1',
  customHeaders: { Authorization: process.env.UNLEASH_TOKEN }
});

// Middleware для проверки feature flag
function featureFlagMiddleware(flagName) {
  return (req, res, next) => {
    const userId = req.user?.id || 'anonymous';
    const isEnabled = unleash.isEnabled(flagName, { userId });

```

```

    res.locals[flagName] = isEnabled;
    next();
  };
}

app.use('/api/new-feature', featureFlagMiddleware('new-payment-flow'));
app.get('/api/new-feature', (req, res) => {
  if (res.locals['new-payment-flow']) {
    res.json({ flow: 'new', version: '2.0' });
  } else {
    res.json({ flow: 'legacy', version: '1.0' });
  }
});

```

Тестирование: `curl -H "X-Canary-Version: v2" http://myapp.example.com/`

Методические указания для выполнения практической части работы

Задачи работы:

1. Настройка canary-развертывания:

- Создать два набора ресурсов (stable и canary) с разными версиями приложения.
- Настроить маршрутизацию трафика (например, 90% на stable, 10% на canary) с использованием Ingress/VirtualService .
- Реализовать поэтапное увеличение доли трафика на canary (10% → 50% → 100%).

2. Интеграция системы feature flags:

- Выбрать и настроить систему feature flags (LaunchDarkly, Unleash, Flagsmith, PostHog) .
- Реализовать в коде проверку feature flag для новой функциональности.
- Настроить удалённое включение/выключение функции через консоль.

3. Автоматизация canary-анализа:

- Настроить сбор метрик (error rate, latency) для stable и canary версий.
 - Реализовать автоматический откат при превышении порога ошибок (например, >1% 5xx ошибок) .
 - Настроить CI/CD пайплайн для прогрессивного увеличения трафика.
4. Наблюдение и верификация:
- Отслеживать распределение трафика между версиями.
 - Выполнить нагрузочное тестирование для проверки canary-анализа.
5. Оформление отчёта:
- Зафиксировать конфигурации canary-развертывания и feature flags.
 - Представить скриншоты процесса и результатов анализа.

Варианты заданий

Базовый уровень

Общее требование: Настроить canary-развертывание (10% трафика).
 Реализовать ручное переключение веса. Продемонстрировать работу двух версий параллельно.

№	Оркестратор	Язык/Платформа	Способ canary
1	Minikube + Nginx	Python (Flask)	canary-weight: 10
2	Minikube + Nginx	Node.js (Express)	canary-weight: 10
3	Minikube + Nginx	Go (net/http)	canary-weight: 10
4	Kind + Nginx	Python (FastAPI)	canary-weight: 15
5	Kind + Nginx	Node.js (NestJS)	canary-weight: 15
6	K3s + Nginx	Go (Gin)	canary-weight: 10
7	Minikube + Istio	Python (Flask)	VirtualService weight

8	Minikube + Istio	Node.js (Express)	VirtualService weight
9	Kind + Istio	Go (Echo)	VirtualService weight
10	Minikube + Nginx	Ruby (Sinatra)	canary-weight: 10
11	Minikube + Nginx	PHP (Laravel)	canary-weight: 10
12	Kind + Nginx	Java (Spring Boot)	canary-weight: 10
13	K3s + Nginx	C (ASP.NET)	canary-weight: 10
14	Minikube + Nginx	Static Site (nginx)	canary-weight: 10
15	Kind + Nginx	Rust (Actix)	canary-weight: 10

Средний уровень

Общее требование: Настроить canary с автоматическим анализом метрик (error rate). Реализовать автоматический откат при ошибках. Интегрировать feature flag (Unleash/Flagsmith).

№	Оркестратор	Canary инструмент	Feature flag
1	Minikube	Argo Rollouts	Unleash
2	Minikube	Flagger	Flagsmith
3	Kind	Argo Rollouts	LaunchDarkly
4	K3s	Flagger	PostHog
5	Minikube + Istio	Argo Rollouts	Unleash
6	Kind + Istio	Flagger	Flagsmith
7	EKS (local)	Argo Rollouts	GrowthBook
8	Minikube	Nginx + custom script	Unleash
9	Kind	Argo Rollouts	PostHog
10	K3s	Flagger + Prometheus	Flagsmith
11	Minikube	Argo Rollouts + AnalysisTemplate	Unleash

12	Kind	Nginx + Prometheus query	LaunchDarkly
13	Minikube + Istio	VirtualService + Flagger	Unleash
14	K3s	Argo Rollouts	Flagsmith
15	Minikube	Canary + Header routing	Unleash

4.3. Продвинутый уровень (15 вариантов)

Общее требование: Реализовать canary с кастомными метриками (latency P99, бизнес-метрики). Настроить прогрессивную доставку с AnalysisTemplate. Интегрировать feature flag с A/B тестированием. Реализовать canary по заголовкам для внутренних пользователей.

№	Оркестратор	Сценарий	Дополнительные требования
	Mini kube + Istio	Анализ latency P99	Flagger + Prometheus
	Kind + Nginx	Canary + A/B тест	Feature flag для вариантов A/B
	K3s + Argo	Бизнес-метрики (conversion)	AnalysisTemplate с SQL запросом
	Minikube + Istio	Canary по заголовкам + cookie	VirtualService + match
	Kind + Argo	Multi-step canary (10/25/50/100)	Pause + analysis
	EKS (local)	Canary + Blue-Green гибрид	Argo Rollouts
	Minikube	Безопасный	canary-by-header + LDAP

		canary (только internal users)	
	Kind + Flagger	Автоматический rollback на 5xx	Порог: 1% ошибок
	Специфика БПЛА	Canary для ROS нод	+ feature flag для алгоритма
	K3s + Istio	Canary + Kiali визуализация	Мониторинг трафика
	Minikube	Feature flag rollout (10% → 50% → 100%)	Unleash стратегия
	Kind + Argo	A/B тестирование UI	Feature flag + canary вместе
	Minikube + Nginx	Canary + нагрузочное тестирование	k6 во время canary
	K3s	Canary + Grafana дашборд	Визуализация метрик
	Комплексный	Полный progressive delivery	Argo + Istio + Prometheus + Grafana

Контрольные вопросы

1. Что такое canary deployment и в чем его отличие от rolling update?
2. Как настроить canary с Nginx Ingress? Какие аннотации используются?
3. В чем преимущество Istio VirtualService для canary-развертывания?
4. Что такое Argo Rollouts и какие возможности он предоставляет?
5. Как настроить автоматический откат при превышении порога ошибок?

6. Что такое feature flag и для чего он используется?
7. Какие инструменты feature flags вы знаете? В чем их отличия?
8. Как выполнить canary-развертывание только для внутренних пользователей?
9. Как настроить canary по HTTP-заголовкам в Nginx Ingress?
10. Что такое AnalysisTemplate в Argo Rollouts?
11. Какие метрики наиболее важны для canary-анализа?
12. Как интегрировать feature flag с canary-развертыванием?
13. В чем отличие canary от A/B тестирования?
14. Как настроить процентный rollout feature flag в LaunchDarkly?
15. Какие метрики latency (P99) важны для canary-анализа и как их настроить?

Требования к отчету

Отчет должен содержать следующие разделы:

1. Титульный лист с указанием ФИО студента, группы, варианта.
2. Цель и задачи работы.
3. Теоретическая часть:
 - Описание canary-стратегии и её преимуществ.
 - Обоснование выбора инструментов (Ingress/Istio/Argo).
 - Описание feature flag системы.
4. Практическая часть:
 - Листинг манифестов Kubernetes (Deployment, Service, Ingress/VirtualService).
 - Листинг конфигурации Argo Rollouts (если используется).
 - Листинг кода приложения с feature flag.
 - Листинг CI/CD пайплайна.
5. Результаты выполнения:

- Скриншоты распределения трафика (10/90).
- Скриншоты canary-анализа и автоматического отката.
- Скриншоты включения feature flag через консоль.
- Скриншоты нагрузки и метрик.

6. Анализ результатов:

- Таблица изменения весов.
- Анализ метрик ошибок во время canary.

7. Ответы на контрольные вопросы (3-5 вопросов).

8. Вывод - достигнута ли цель, какие сложности возникли.

Критерии оценивания

Критерий	Базовый уровень	Средний уровень	Продвинутый уровень
Canary-развертывание настроено	2 балла	2 балла	2 балла
Трафик распределяется (10/90)	2 балла	2 балла	2 балла
Возможно ручное изменение веса	1 балл	2 балла	2 балла
Feature flag интегрирован	0 баллов	2 балла	2 балла
Автоматический анализ метрик	0 баллов	1 балл	2 балла
Автоматический откат при ошибках	0 баллов	1 балл	2 балла
Canary по заголовкам/cookie	0 баллов	0 баллов	2 балла
Качество отчета и ответы	2 балла	2 балла	2 балла

Шкала перевода баллов в оценку:

Оценка	Базовый (макс. 7)	Средний (макс. 12)	Продвинутый (макс. 16)
5	6-7	11-12	14-16
4	5	9-10	11-13
3	3-4	6-8	8-10
2	<3	<6	<8

Бонусные баллы (макс. +3):

- Интеграция с Prometheus/Grafana - +1 балл.
- Настройка canary по заголовкам + cookie - +1 балл.
- A/B тестирование на основе feature flags - +1 балл.

Лабораторная работа №14. GitOps с Argo CD

Цель работы: Сформировать практические навыки реализации GitOps-подхода с использованием Argo CD для декларативного управления разворачиванием приложений в Kubernetes, где Git-репозиторий выступает единственным источником истины (single source of truth) о состоянии системы

Теоретическое обоснование

1. Концепция GitOps

GitOps - это парадигма непрерывной доставки, использующая Git-репозитории как единственный источник истины для декларативного описания инфраструктуры и приложений .

Основные принципы GitOps:

Принцип	Описание
Декларативность	Вся конфигурация системы описывается декларативно (манифесты Kubernetes)
Версионирование в Git	Git - единственный источник истины, вся конфигурация хранится в Git
Автоматическая синхронизация	Изменения в Git автоматически применяются к целевой среде
Наблюдаемость	Агенты в среде отслеживают расхождения и

	сообщают о них

Преимущества GitOps:

- Повышенная безопасность - доступ к кластеру через Git, а не через прямые учётные записи
- Упрощённый откат - достаточно откатить commit в Git
- Улучшенная наблюдаемость - все изменения проходят через Git и подлежат аудиту
- Согласованность сред - один и тот же Git-репозиторий для dev/staging/prod

2.2. Argo CD: архитектура и компоненты

Argo CD - это декларативный инструмент непрерывной доставки (GitOps) для Kubernetes, поддерживающий манифесты, Helm, Kustomize и другие форматы .

Архитектура Argo CD (рисунок 1):

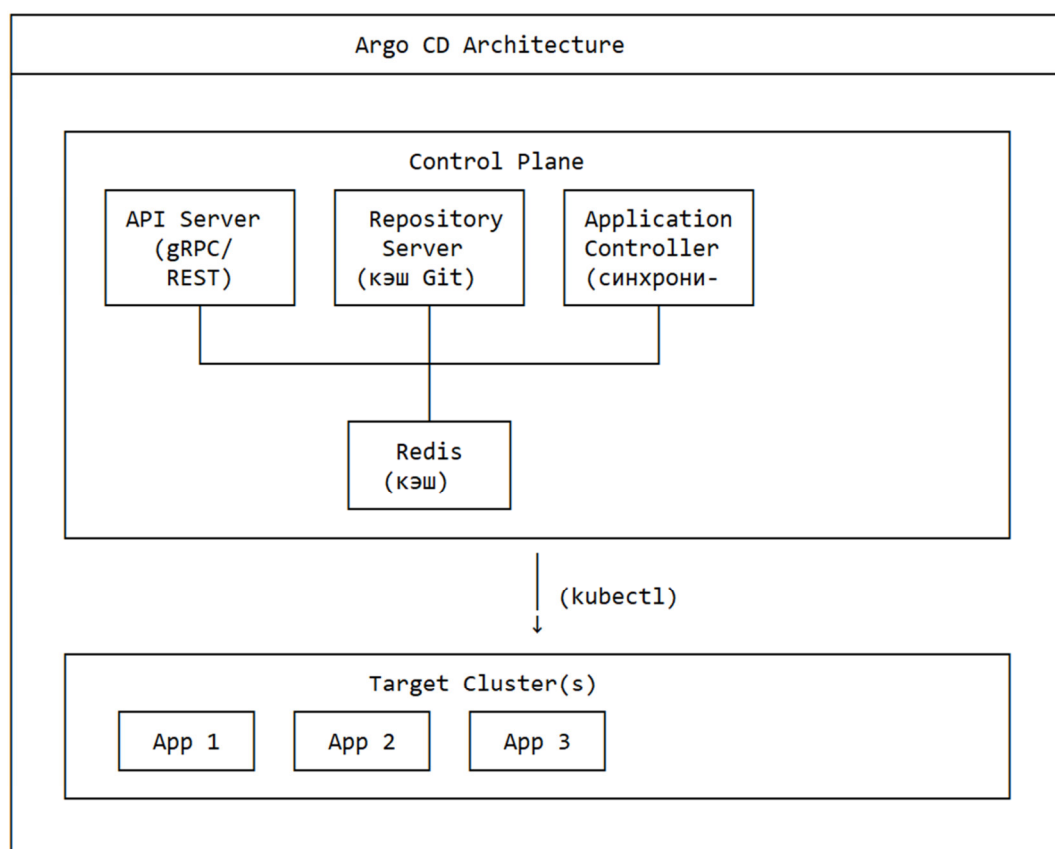


Рисунок 1

Основные компоненты :

Компонент	Функция
API Server	Предоставляет Web UI, CLI и API для управления приложениями. Выполняет аутентификацию, авторизацию и RBAC
Repository Server	Клонирует Git-репозитории, кэширует их и генерирует Kubernetes-манифесты из Helm/Kustomize
Application Controller	Сравнивает желаемое состояние (из Git) с реальным состоянием (в кластере). При расхождении инициирует синхронизацию
Redis	Кэш для Repository Server

3. Установка Argo CD

Установка в Kubernetes-кластер (Minikube/Kind) :

```
# Создание namespace
kubectl create namespace argocd

# Установка Argo CD
kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml

# Установка CLI (Linux)
curl -sSL -o /usr/local/bin/argocd https://github.com/argoproj/argo-cd/releases/latest/download/argocd-linux-amd64
chmod +x /usr/local/bin/argocd

# Установка CLI (Mac)
brew install argocd
```

Получение пароля администратора :

```
# Получение начального пароля
kubectl -n argocd get secret argocd-initial-admin-secret -o
jsonpath="{.data.password}" | base64 -d

# Или с помощью CLI
argocd admin initial-password -n argocd
```

Доступ к Web UI (порт-форвардинг):

```
kubectl port-forward svc/argocd-server -n argocd 8443:443
```

Затем открыть <https://localhost:8443>, логин: admin, пароль: полученный на предыдущем шаге.

Смена пароля через CLI :

```
argocd login localhost:8443 --username admin --password <initial-password> --insecure
argocd account update-password --current-password <current> --new-password <new>
```

4. Создание Git-репозитория для Argo CD

Рекомендуемая структура репозитория (рисунок 2):

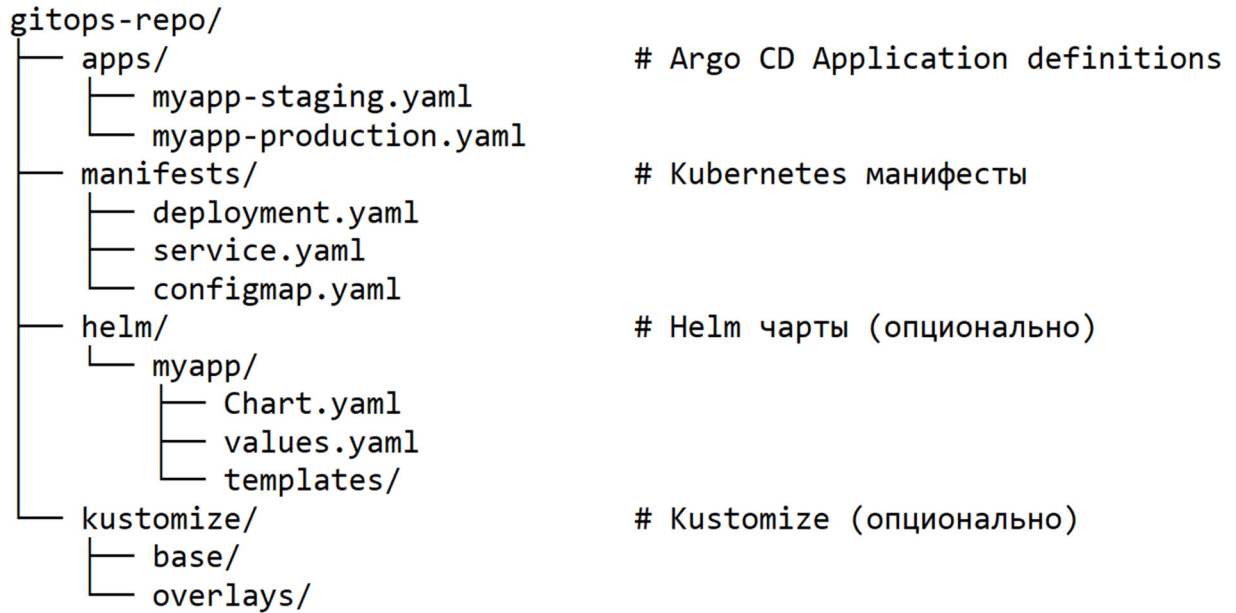


Рисунок 2

5. Создание Application в Argo CD

Application CRD - пример :

```
apiVersion: argoproj.io/v1alpha1  
kind: Application  
metadata:  
  name: myapp  
  namespace: argocd  
spec:  
  # Целевое окружение  
  destination:  
    namespace: default  
    server: https://kubernetes.default.svc # локальный кластер  
  # Источник (Git)  
  source:  
    path: manifests # путь к манифестам  
    repoURL: https://github.com/user/gitops-repo  
    targetRevision: main # ветка  
  # Политика синхронизации  
  syncPolicy:  
    automated: # автоматическая синхронизация  
[citation:1]
```

```
    prune: true                                # удалять ресурсы, отсутствующие в
Git                                             # восстанавливать после ручных
    selfHeal: true                             изменений
    syncOptions:
      - CreateNamespace=true
      # Проект (для multi-tenancy)
    project: default
```

Создание Application через CLI :

2.6. GitOps-цикл: синхронизация и откат

```
argocd app create myapp \
  --repo https://github.com/user/gitops-repo \
  --path manifests \
  --dest-server https://kubernetes.default.svc \
  --dest-namespace default \
  --sync-policy automated
```

Жизненный цикл GitOps с Argo CD (рисунок 3):

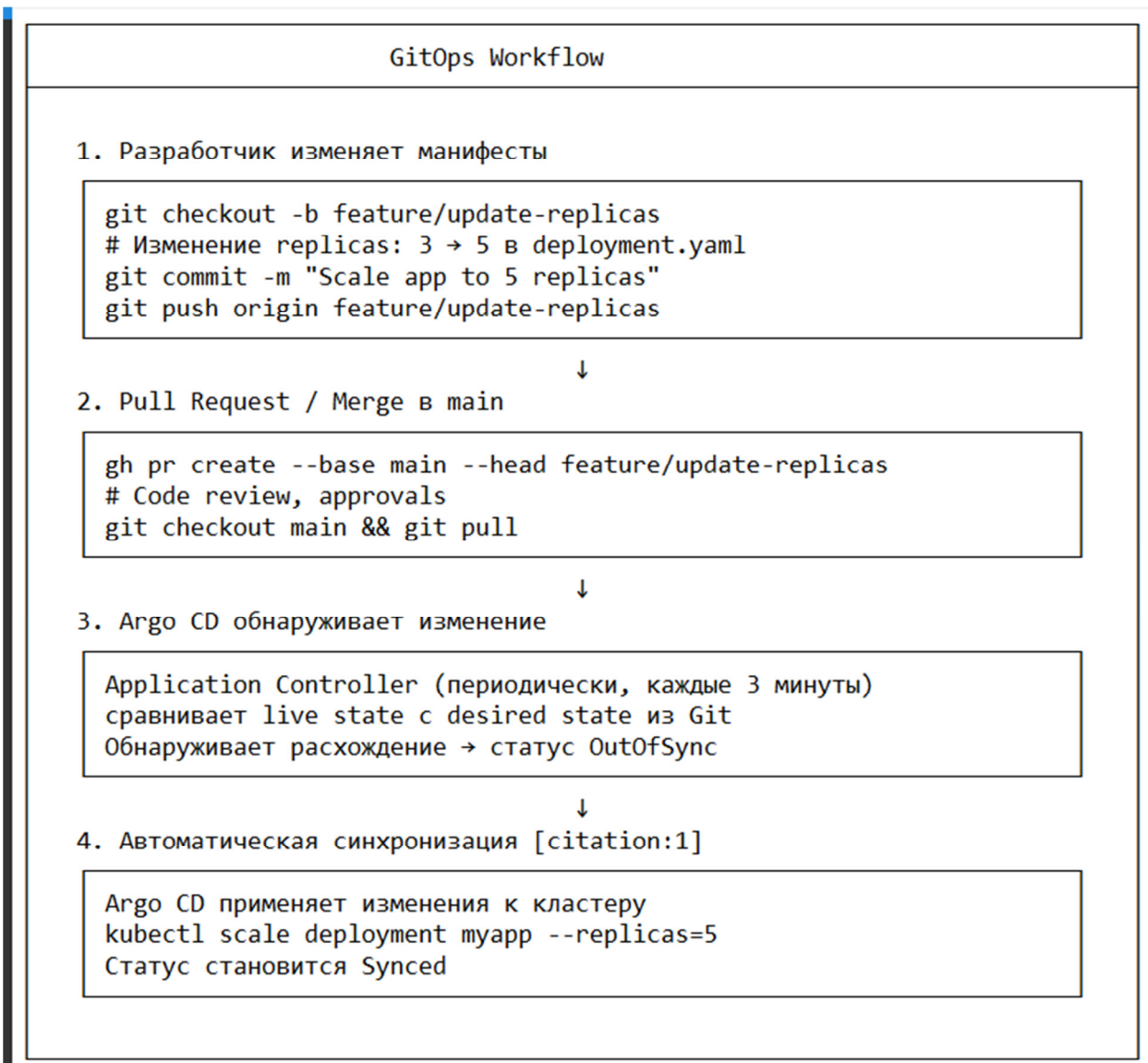


Рисунок 3

Откат (rollback) через Argo CD :

```
# Просмотр истории
argocd app history myapp

# Откат к предыдущей ревизии
argocd app rollback myapp 1

# Через Web UI: History → Rollback
```

7. Интеграция Argo CD с CI/CD

Типовая архитектура: CI собирает образ → GitOps обновляет манифесты → Argo CD применяет :

```
- name: Update manifest with new image tag
  run: |
    cd gitops-repo
    yq eval -i '.spec.template.spec.containers[0].image = "myapp:${{ github.sha }}"' manifests/deployment.yaml
    git config user.name "CI Bot"
    git add .
    git commit -m "Update image to ${{ github.sha }}"
    git push
```

Либо использование Argo CD Image Updater:

```
# Application аннотации для автообновления образа
metadata:
  annotations:
    argocd-image-updater.argoproj.io/image-list: myapp=myregistry/myapp
    argocd-image-updater.argoproj.io/myapp.update-strategy: latest
```

Примеры практического выполнения

Пример 1. Базовый GitOps с простыми манифестами

Тестовое приложение (nginx):

```
# manifests/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-demo
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.25
```

```

        ports:
          - containerPort: 80
---
# manifests/service.yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-demo
spec:
  selector:
    app: nginx
  ports:
    - port: 80
      targetPort: 80
  type: ClusterIP

```

Application manifest:

```

# apps/nginx-app.yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: nginx-demo
  namespace: argocd
spec:
  destination:
    namespace: default
    server: https://kubernetes.default.svc
  source:
    path: manifests
    repoURL: https://github.com/student/argocd-demo.git
    targetRevision: main
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
    syncOptions:
      - CreateNamespace=true
  project: default

```

Пример 2. GitOps с Helm чартом

Структура репозитория (рисунок 4):

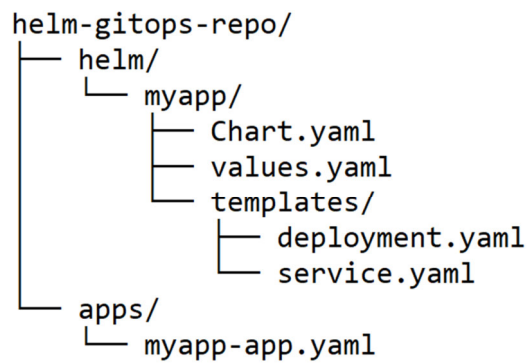


Рисунок 4

Application для Helm :

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: myapp-helm
  namespace: argocd
spec:
  destination:
    namespace: default
    server: https://kubernetes.default.svc
  source:
    repoURL: https://github.com/student/helm-gitops-repo.git
    targetRevision: main
    path: helm/myapp
    helm:
      valueFiles:
        - values.yaml
      parameters:
        - name: replicaCount
          value: "3"
  syncPolicy:
    automated:
      prune: true
```

Пример 3: Argo CD с Kustomize

Структура репозитория (рисунок 5):

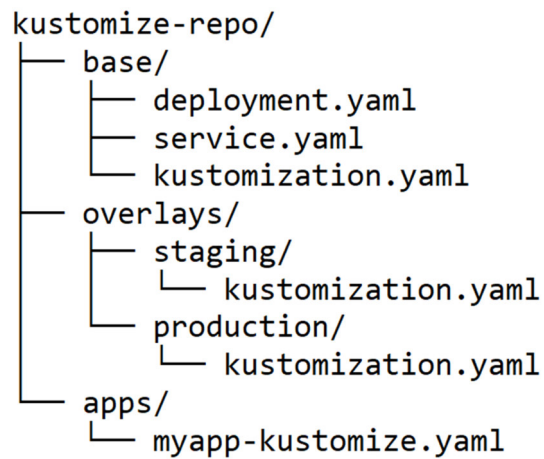


Рисунок 5

Application для Kustomize :

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: myapp-staging
  namespace: argocd
spec:
  destination:
    namespace: staging
    server: https://kubernetes.default.svc
  source:
    repoURL: https://github.com/student/kustomize-repo.git
    targetRevision: main
    path: overlays/staging
  syncPolicy:
    automated: {}
```

Пример 4. ApplicationSet для множества окружений

```
apiVersion: argoproj.io/v1alpha1
kind: ApplicationSet
metadata:
  name: myapp-appset
  namespace: argocd
spec:
  generators:
  - list:
      elements:
      - environment: staging
        namespace: staging
```

```

- environment: production
  namespace: production
template:
  metadata:
    name: 'myapp-{{environment}}'
  spec:
    project: default
    source:
      repoURL: https://github.com/student/gitops-repo.git
      targetRevision: main
      path: manifests/{{environment}}
    destination:
      server: https://kubernetes.default.svc
      namespace: '{{namespace}}'
  syncPolicy:
    automated: {}

```

Методические указания выполнения практической части работы

Задачи работы

1. Установка и настройка Argo CD:

- Развернуть Argo CD в Kubernetes-кластере (Minikube/Kind/K3s) .
- Настроить доступ к Web UI и CLI.
- Получить и сменить пароль администратора.

2. Создание Git-репозитория с манифестами:

- Создать репозиторий для хранения Kubernetes-манифестов.
- Разместить в репозитории декларативные описания приложения

(Deployment, Service) .

- Структурировать манифесты (по окружениям, по компонентам).

3. Настройка синхронизации Argo CD с Git:

- Добавить Git-репозиторий как источник в Argo CD.
- Создать Application, указывающий на репозиторий и целевую среду

.

- Настроить автоматическую синхронизацию (AutoSync) .

4. Демонстрация GitOps-цикла:

- Внести изменения в манифесты в Git-репозитории.
- Убедиться, что Argo CD автоматически применил изменения в

кластере .

- Продемонстрировать откат (rollback) к предыдущей версии.

5. Оформление отчёта:

- Зафиксировать конфигурацию Argo CD и Git-репозитория.
- Представить скриншоты процесса синхронизации и UI Argo CD.

Варианты заданий

Базовый уровень

Общее требование: Установить Argo CD, создать Git-репозиторий с манифестами простого приложения, настроить автоматическую синхронизацию. Продемонстрировать, что изменение манифестов в Git приводит к изменению в кластере.

№	Приложение	Формат манифестов	Количество ресурсов
1	nginx	plain YAML	Deployment + Service
2	httpd	plain YAML	Deployment + Service
3	python-flask (hello-world)	plain YAML	Deployment + Service
4	nodejs-express	plain YAML	Deployment + Service
5	go-http-server	plain YAML	Deployment + Service
6	redis	plain YAML	Deployment + Service
7	postgres	plain YAML	Deployment + Service + PVC
8	mongodb	plain YAML	Deployment + Service + PVC
9	nginx + configmap	plain YAML	Deployment + Service + ConfigMap
10	httpd + secret	plain YAML	Deployment + Service + Secret
11	nginx + ingress	plain YAML	Deployment + Service + Ingress

12	alpine (sleep)	plain YAML	Deployment
13	busybox (sleep)	plain YAML	Deployment
14	nginx + multiple replicas	plain YAML	Deployment + Service
15	custom docker image (your app)	plain YAML	Deployment + Service

Средний уровень

Общее требование: Использовать Helm или Kustomize. Настроить ApplicationSet для нескольких окружений (staging/production). Реализовать автоматическую синхронизацию с self-heal и prune.

№	Инструмент	Приложение	Особенности
1	Helm	nginx	Разные values для staging/production
2	Helm	python-flask	Values с replicaCount и image tag
3	Helm	postgres	С PVC и secrets
4	Helm	redis	С configmap для конфигурации
5	Helm	nodejs-app	Сборка из Helm чарта
6	Kustomize	nginx	Base + overlay staging/production
7	Kustomize	python-app	Base с configMapGenerator
8	Kustomize	go-app	Base + patches для разных окружений
9	Kustomize	java-app	Base с images tags
10	Helm + Kustomize	nginx	Helm чарт + Kustomize поверх
11	ApplicationSet	nginx	Генерация для 2-3 окружений
12	ApplicationSet	python-app	С git-генератором (папки)
13	ApplicationSet	go-app	С cluster-генератором

14	ApplicationSet	nodejs-app	С list-генератором
15	Helm + ApplicationSet	redis	Matrix-генератор

Продвинутый уровень

Общее требование: Настроить Argo CD Image Updater для автоматического обновления образов. Реализовать PreSync/PostSync hooks для миграций. Интегрировать Argo CD с GitHub Actions для обновления манифестов.

№	Технология	Сценарий	Дополнительные требования
1	Image Updater	Автообновление образа при push в registry	Sync волны
2	Hooks	PreSync hook для миграции БД	Job + hook annotation
3	Hooks	PostSync hook для smoke тестов	Pod с curl проверкой
4	CI/CD integration	GitHub Actions обновляет тег в Git	yq + commit + push
5	CI/CD integration	GitLab CI → обновление манифестов	Автоматический MR
6	Multi-cluster	Argo CD управляет 2 кластерами	Добавление remote cluster
7	Multi-cluster	Hub-and-Spoke архитектура	Registry cluster + CLI
8	Argo Rollouts	Canary deployment	Rollout + AnalysisTemplate
9	Argo Rollouts	Blue-Green через Argo CD	Rollout + Service
10	Специфика	ROS 2 приложение	Helm + Argo CD для ROS

	БПЛА		нод
11	Secret management	Sealed Secrets с Argo CD	Bitnami Sealed Secrets
12	Secret management	External Secrets Operator	Интеграция с Vault
13	Observability	Argo CD + Prometheus	Метрики синхронизации
14	GitOps + CI	Полный цикл: CI → Git → Argo CD	GitHub Actions + Argo CD
15	Комплексный	App of Apps + ApplicationSet	Иерархическая структура

Контрольные вопросы

1. Что такое GitOps? Какие принципы лежат в его основе?
2. В чем отличие GitOps от классического CI/CD?
3. Какие компоненты входят в архитектуру Argo CD?
4. Какую роль выполняет Application Controller?
5. Что такое «single source of truth» в GitOps?
6. Как установить Argo CD в Kubernetes-кластер?
7. Как получить и сменить пароль администратора Argo CD?
8. Что такое Application в Argo CD?
9. Какие политики синхронизации поддерживает Argo CD?
10. Что означают статусы Synced и OutOfSync?
11. Как настроить автоматическую синхронизацию с Git?
12. Как выполнить откат (rollback) к предыдущей версии?
13. Какие форматы манифестов поддерживает Argo CD?
14. Что такое ApplicationSet и для чего он нужен?
15. Как интегрировать Argo CD с существующим CI/CD пайплайном?

Требования к отчету

Отчет должен содержать следующие разделы:

1. Титульный лист с указанием ФИО студента, группы, варианта.
2. Цель и задачи работы.
3. Теоретическая часть:
 - Описание GitOps и его принципов.
 - Архитектура Argo CD.
 - Обоснование выбора формата манифестов (plain/Helm/Kustomize).
4. Практическая часть:
 - Листинг команд установки Argo CD.
 - Листинг манифестов приложения (Deployment, Service).
 - Листинг Application (или ApplicationSet).
 - Структура Git-репозитория.
5. Результаты выполнения:
 - Скриншоты Web UI Argo CD (Application list, Sync status).
 - Скриншоты CLI вывода (argocd app list, argocd app get).
 - Скриншоты внесения изменений в Git и автоматической синхронизации.
 - Скриншоты подтверждения применения изменений в кластере (kubectl get pods).
6. Анализ результатов:
 - Таблица времени синхронизации после изменения.
 - Сравнение поведения при AutoSync и Manual Sync.
7. Ответы на контрольные вопросы (3-5 вопросов).
8. Вывод — достигнута ли цель, какие сложности возникли.

Критерии оценивания

Критерий	Базовый уровень	Средний уровень	Продвинутый уровень

Argo CD установлен и работает	2 балла	2 балла	2 балла
Git-репозиторий создан, манифесты добавлены	2 балла	2 балла	2 балла
Application создан, синхронизация работает	2 балла	2 балла	2 балла
Автоматическая синхронизация (AutoSync) настроена	1 балл	2 балла	2 балла
Использован Helm или Kustomize	0 баллов	2 балла	2 балла
Настроен ApplicationSet для нескольких окружений	0 баллов	1 балл	2 балла
Интеграция с CI (обновление манифестов)	0 баллов	0 баллов	2 балла
Качество отчета и ответы на вопросы	2 балла	2 балла	2 балла

Шкала перевода баллов в оценку:

Оценка	Базовый (макс. 9)	Средний (макс. 13)	Продвинутый (макс. 16)
5	8-9	12-13	14-16
4	6-7	10-11	11-13
3	4-5	7-9	8-10
2	<4	<7	<8

Бонусные баллы (макс. +3):

- Настройка self-heal (восстановление после ручных изменений) - +1 балл.
- Настройка Argo CD Image Updater - +1 балл.
- Реализация PreSync/PostSync hooks - +1 балл.

Лабораторная работа №15

CI/CD для встроенной системы (кросс-компиляция)

Цель работы: Сформировать практические навыки настройки CI/CD-пайплайна для встроенных систем, включая конфигурацию кросс-компилятора (ARM GCC), автоматизацию сборки прошивки и управление артефактами (bin/hex файлы) .

Теоретическое обоснование

1. Специфика CI/CD для встроенных систем

CI/CD для встраиваемых систем имеет ряд ключевых отличий от разработки обычного ПО :

Аспект	Обычное ПО	Встроенные системы
Целевая платформа	x86/x64 (та же, что и хост)	ARM, RISC-V, AVR (другая архитектура)
Инструменты сборки	gcc/clang на хосте	Кросс-компилятор (arm-none-eabi-gcc)
Тестирование	На хосте	Симуляция (QEMU) или HIL-стенды
Артефакты	ELF, Docker образ	.bin, .hex, .elf, S-record
Развертывание	Облако/сервер	ОТА-обновления, программатор

Ключевые принципы CI/CD для встроенных систем :

- Контейнеризация окружения - использование Docker для воспроизводимости сборки .

- Автоматизация кросс-компиляции - единая конфигурация для всех целевых платформ .

- Раннее тестирование - unit-тесты на хосте, интеграционные в симуляторе .

2. Кросс-компиляция для ARM

Кросс-компиляция - это процесс компиляции кода для целевой архитектуры, отличной от архитектуры хоста .

Популярные кросс-компиляторы:

Тулчейн	Целевые архитектуры	Лицензия
arm-none-eabi-gcc	ARM Cortex-M/R (bare-metal)	GNU GPL
arm-linux-gnueabi-gcc	ARM с Linux (hard float)	GNU GPL
riscv64-unknown-elf-gcc	RISC-V	GNU GPL
avr-gcc	AVR (Arduino)	GNU GPL

Установка ARM GCC в GitHub Actions :

```
- name: Install ARM GCC
  run: |
    wget https://developer.arm.com/-
/media/Files/downloads/gnu/13.2.rel1/binrel/arm-gnu-toolchain-13.2.rel1-x86_64-
arm-none-eabi.tar.xz
    tar -xf arm-gnu-toolchain-13.2.rel1-x86_64-arm-none-eabi.tar.xz
    echo "$(pwd)/arm-gnu-toolchain-13.2.rel1-x86_64-arm-none-eabi/bin" >>
$GITHUB_PATH
```

3. Переход с Keil/IDE на GCC

Многие встроенные проекты изначально разрабатываются в проприетарных IDE (Keil, IAR). Переход на GCC требует :

1. Замена синтаксиса ассемблера - ARMASM (Keil) → GAS (GNU)
2. Замена линковщика - .sct (Keil) → .ld (GNU linker script)
3. Замена синтаксиса inline assembly - __asm → __asm__ volatile
4. Настройка Makefile/CMake для автоматизации сборки

Сравнение синтаксиса startup файла :

Keil (ARMASM)	GCC (GAS)
AREA RESET, CODE, READONLY	.section .vector_table, "a", %progbits
EXPORT __Vectors	global __Vectors
DCD __initial_sp	.long __initial_sp
__irq USART1_IRQHandler	__attribute__((interrupt("IRQ")))

4. Makefile для кросс-компиляции

Пример Makefile для STM32 :

```
# Toolchain
PREFIX = arm-none-eabi-
CC = $(PREFIX)gcc
AS = $(PREFIX)as
LD = $(PREFIX)ld
OBJCOPY = $(PREFIX)objcopy

# Target configuration
MCU = cortex-m4
FPU = fpv4-sp-d16
FLOAT-ABI = hard

# Compiler flags
CFLAGS = -mcpu=$(MCU) -mthumb -mfpu=$(FPU) -mfloat-abi=$(FLOAT-ABI)
CFLAGS += -Wall -Werror -g -O2
CFLAGS += -DSTM32F407xx -DUSE_HAL_DRIVER

# Linker flags
LDFLAGS = -mcpu=$(MCU) -mthumb -mfpu=$(FPU) -mfloat-abi=$(FLOAT-ABI)
LDFLAGS += -specs=nosys.specs -specs=nano.specs
LDFLAGS += -T stm32f407.ld
```

```

# Source files
SRCS = src/main.c src/system_stm32f4xx.c startup_stm32f407xx.s
OBJS = $(SRCS:.c=.o)
OBJS := $(OBJS:.s=.o)

# Target
TARGET = firmware

all: $(TARGET).elf $(TARGET).bin $(TARGET).hex

$(TARGET).elf: $(OBJS)
    $(CC) $(LDFLAGS) $^ -o $@
    $(PREFIX)size $@

%.bin: %.elf
    $(OBJCOPY) -O binary $< $@

%.hex: %.elf
    $(OBJCOPY) -O ihex $< $@

clean:
    rm -f $(OBJS) $(TARGET).elf $(TARGET).bin $(TARGET).hex

.PHONY: all clean

```

5. Использование Docker для воспроизводимой сборки

Контейнеризация обеспечивает идентичность окружений на всех этапах:

Dockerfile для embedded CI:

```

FROM ubuntu:22.04

RUN apt-get update && apt-get install -y \
    build-essential \
    cmake \
    git \
    wget \
    xz-utils

# Установка ARM GCC
RUN wget https://developer.arm.com/-
/media/Files/downloads/gnu/13.2.rel1/binrel/arm-gnu-toolchain-13.2.rel1-x86_64-
arm-none-eabi.tar.xz && \

```

```
tar -xf arm-gnu-toolchain-13.2.rel1-x86_64-arm-none-eabi.tar.xz -C /opt && \
rm arm-gnu-toolchain-13.2.rel1-x86_64-arm-none-eabi.tar.xz

ENV PATH="/opt/arm-gnu-toolchain-13.2.rel1-x86_64-arm-none-eabi/bin:${PATH}"

WORKDIR /workspace
```

2.6. Артефакты встроенной сборки

Формат	Описание	Использование
.elf	Executable Linkable Format	Отладка, загрузка через отладчик
.bin	Raw binary	Загрузка через программатор, ОТА
.hex	Intel HEX (текстовый)	Загрузка через программатор
.map	Memory map	Анализ использования памяти
.lst	Listing file	Ассемблерный листинг

Публикация артефактов в GitHub Actions :

```
- name: Upload firmware artifacts
  uses: actions/upload-artifact@v4
  with:
    name: firmware-${{ github.sha }}
    path: |
      build/*.bin
      build/*.hex
      build/*.elf
      build/*.map
```

Примеры практического выполнения

1. Пример для STM32 (GitHub Actions)

Файл .github/workflows/build.yml:

```
name: STM32 Firmware CI

on:
  push:
    branches: [ main ]
```

```

tags:
  - 'v*'
pull_request:
  branches: [ main ]

env:
  TARGET: stm32f407
  PROJECT_DIR: firmware

jobs:
  build:
    runs-on: ubuntu-22.04
    steps:
      - uses: actions/checkout@v4

      - name: Install ARM GCC
        run: |
          wget https://developer.arm.com/-
/media/Files/downloads/gnu/13.2.rel1/binrel/arm-gnu-toolchain-13.2.rel1-x86_64-
arm-none-eabi.tar.xz
          tar -xf arm-gnu-toolchain-13.2.rel1-x86_64-arm-none-eabi.tar.xz
          echo "$(pwd)/arm-gnu-toolchain-13.2.rel1-x86_64-arm-none-eabi/bin" >>
$GITHUB_PATH

      - name: Cache toolchain
        uses: actions/cache@v3
        with:
          path: arm-gnu-toolchain-*
          key: ${{ runner.os }}-arm-gcc-13.2

      - name: Build firmware
        working-directory: ${{ env.PROJECT_DIR }}
        run: make clean && make all

      - name: Generate version info
        run: |
          echo "VERSION=${GITHUB_REF_NAME#v}" >> $GITHUB_ENV
          echo "COMMIT=$(git rev-parse --short HEAD)" >> $GITHUB_ENV
          echo "BUILD_TIME=$(date -u +%Y-%m-%dT%H:%M:%SZ)" >> $GITHUB_ENV

      - name: Create version file
        run: |
          cat > version.txt << EOF
          Version: ${{ env.VERSION }}
          Commit: ${{ env.COMMIT }}
          Build: ${{ env.BUILD_TIME }}
          Target: ${{ env.TARGET }}
          EOF

      - name: Upload artifacts
        uses: actions/upload-artifact@v4

```

```

with:
  name: stm32-firmware-${{ github.sha }}
  path: |
    ${ env.PROJECT_DIR }}/build/*.bin
    ${ env.PROJECT_DIR }}/build/*.hex
    ${ env.PROJECT_DIR }}/build/*.elf
    ${ env.PROJECT_DIR }}/build/*.map
    version.txt

- name: Create GitHub Release
  if: startsWith(github.ref, 'refs/tags/')
  uses: softprops/action-gh-release@v1
  with:
    files: |
      ${ env.PROJECT_DIR }}/build/*.bin
      ${ env.PROJECT_DIR }}/build/*.hex
      ${ env.PROJECT_DIR }}/build/*.elf
    body: |
      Firmware for ${ env.TARGET }
      Commit: ${ env.COMMIT }

```

2. Пример для FreeRTOS (GitLab CI)

Файл .gitlab-ci.yml:

```

stages:
  - build
  - test
  - package

variables:
  GIT_SUBMODULE_STRATEGY: recursive

cache:
  key: "$CI_COMMIT_REF_SLUG"
  paths:
    - toolchain/

before_script:
  - apt-get update -qq && apt-get install -y -qq make gcc-arm-none-eabi

build-freertos:
  stage: build
  script:
    - cd freertos_demo
    - make -j$(nproc)
    - arm-none-eabi-size build/freertos.elf
  artifacts:
    paths:

```

```

    - freertos_demo/build/*.bin
    - freertos_demo/build/*.hex
  expire_in: 1 week

test-unit:
  stage: test
  script:
    - cd tests
    - make && ./unit_tests
  allow_failure: false

package-release:
  stage: package
  script:
    - mkdir release
    - cp freertos_demo/build/*.bin release/
    - cp freertos_demo/build/*.hex release/
    - tar -czf freertos-firmware.tar.gz release/
  artifacts:
    paths:
      - freertos-firmware.tar.gz
  only:
    - tags

```

3. Пример для Zephyr RTOS

Файл `.github/workflows/zephyr.yml`:

```

name: Zephyr RTOS Build

on:
  push:
    branches: [ main ]

jobs:
  build-zephyr:
    runs-on: ubuntu-22.04
    container:
      image: zephyrprojectrtos/zephyr-build:latest
    steps:
      - uses: actions/checkout@v4
        with:
          path: app

      - name: Initialize West
        run: |
          west init -l app

```



```

west update

- name: Build for nRF52840
  run: |
    west build -b nrf52840dk_nrf52840 app
    cp build/zephyr/zephyr.bin firmware-nrf52840.bin
    cp build/zephyr/zephyr.hex firmware-nrf52840.hex

- name: Build for STM32F407
  run: |
    west build -b stm32f4_disco app
    cp build/zephyr/zephyr.bin firmware-stm32f407.bin
    cp build/zephyr/zephyr.hex firmware-stm32f407.hex

- name: Upload artifacts
  uses: actions/upload-artifact@v4
  with:
    name: zephyr-firmware
    path: |
      firmware-*.bin
      firmware-*.hex

```

3.4. Пример для Arduino (CLI)

Файл .gitlab-ci.yml:

```

arduino-build:
  stage: build
  image: ubuntu:22.04
  before_script:
    - apt-get update && apt-get install -y wget xz-utils
    - wget https://downloads.arduino.cc/arduino-cli/arduino-
cli_latest_Linux_64bit.tar.gz
    - tar -xf arduino-cli_latest_Linux_64bit.tar.gz
    - ./arduino-cli core install arduino:avr
  script:
    - ./arduino-cli compile --fqbn arduino:avr:uno --output-dir build
    - cp build/*.hex firmware.hex
  artifacts:
    paths:
      - firmware.hex

```

**Методические указания выполнения практической части
практики**

Задачи работы:

1. Настройка кросс-компилятора в CI-среде:

- Установить и настроить ARM GCC тулчейн (arm-none-eabi-gcc) в CI-раннере .

- Настроить переменные окружения для кросс-компиляции (CC, CFLAGS, LDFLAGS) .

- Организовать кэширование тулчейна для ускорения последующих сборок.

2. Сборка прошивки для целевой платформы:

- Написать Makefile/CMakeLists.txt для кросс-компиляции .

- Выполнить сборку бинарного файла прошивки (.bin/.hex) .

- Настроить линковку и генерацию карты памяти (map-файла).

3. Управление артефактами сборки:

- Сохранить собранные артефакты (.bin, .hex, .elf) .

- Настроить публикацию артефактов в CI/CD.

- Добавить метаданные (версия прошивки, время сборки, commit hash).

4. Автоматизация и тестирование:

- Настроить запуск unit-тестов на хост-машине (если применимо) .

- Добавить статический анализ кода (cppcheck, clang-tidy) .

- Интегрировать проверку кодстайла.

5. Оформление отчёта:

- Зафиксировать конфигурацию пайплайна.

- Представить скриншоты процесса сборки и артефактов.

Варианты заданий

Базовый уровень

Общее требование: Настроить кросс-компилятор для целевой платформы. Выполнить сборку прошивки. Сохранить .bin/.hex артефакты.

№	Целевая платформа	RTOS/Фреймворк	Тулчейн
1	STM32F103 (Blue Pill)	Bare metal + CMSIS	arm-none-eabi-gcc
2	STM32F407 Discovery	STM32 HAL	arm-none-eabi-gcc
3	STM32F429 (Nucleo)	FreeRTOS	arm-none-eabi-gcc
4	nRF52840 (BLE)	Zephyr	arm-none-eabi-gcc
5	ESP32	ESP-IDF	xtensa-esp32-elf
6	RP2040 (Raspberry Pico)	C SDK	arm-none-eabi-gcc
7	AVR (Arduino Uno)	Arduino	avr-gcc
8	AVR (Arduino Mega)	Bare metal	avr-gcc
9	RISC-V (SiFive)	Freedom SDK	riscv64-unknown-elf
10	LPC1768 (mbed)	Bare metal	arm-none-eabi-gcc
11	SAMD21 (Arduino MKR)	Arduino	arm-none-eabi-gcc
12	IMX RT1062 (Teensy)	Teensyduino	arm-none-eabi-gcc
13	CH32V203 (RISC-V)	WCH SDK	riscv-none-elf
14	Tiva C (TM4C123)	TivaWare	arm-none-eabi-gcc
15	MSP430	TI-RTOS	msp430-elf-gcc

Средний уровень

Общее требование: Настроить Docker-контейнер с тулчейном. Добавить unit-тесты (Ceedling/Unity). Добавить статический анализ (cppcheck). Настроить версионирование прошивки.

№	Платформа	Дополнительные требования
1	STM32F407	Добавить юнит-тесты с Unity framework
2	STM32F103	Настроить cppcheck для MISRA-C
3	nRF52840	Добавить SAST (CodeQL) для embedded C

4	ESP32	Настроить Docker-образ с ESP-IDF
5	RP2040	Добавить проверку форматирования clang-format
6	AVR	Настроить симуляцию в QEMU
7	RISC-V	Добавить автотестирование в симуляторе Spike
8	STM32F429	Настроить кэширование тулчейна в CI
9	nRF52840	Добавить SCA анализ (OWASP Dependency Check)
10	LPC1768	Настроить генерацию map-файла и анализ памяти
11	SAMD21	Добавить проверку code coverage (gcov)
12	STM32F103	Настроить parallel build (make -j)
13	ESP8266	Интеграция с PlatformIO
14	CH32V203	Добавить проверку утечек памяти через Valgrind (хост)
15	Tiva C	Настроить pre-commit хуки для embedded кода

Продвинутый уровень

Общее требование: Настроить HIL-тестирование (Hardware-in-the-Loop) через USB-connected устройство. Реализовать OTA-обновление прошивки. Добавить статический анализ кода (clang-tidy, clang-analyzer). Интегрировать SBOM генерацию.

№	Платформа	Сценарий	Дополнительные требования
1	STM32F407	OTA-обновление через UART/YModem	Подпись прошивки
2	nRF52840	HIL-тестирование через pytest + pySerial	Автоматическая проверка ответа
3	ESP32	OTA через HTTP (партиции)	А/В схема + откат

4	RP2040	QEMU симуляция + интеграционные тесты	Модульные тесты в симуляторе
5	STM32F103	IAR → GCC миграция + CI проверка	Сравнение размера .bin
6	RISC-V	Renode симуляция	Тестирование периферии
7	ARM Cortex-M	Анализ покрытия кода (gcov/lcov)	Отчёт в HTML
8	STM32F429	CAN-bus HIL тестирование	Проверка обмена по CAN
9	ESP32-C3	Wi-Fi OTA + автоматический rollback	Мониторинг ошибок
10	nRF52840	BLE тестирование с центральным устройством	Проверка соединения
11	Кроссплатформа	Сборка для 3+ архитектур	Matrix strategy в CI
12	Специфика БПЛА	PX4 прошивка + симуляция в Gazebo	SITL тестирование
13	Автомобильный	AUTOSAR (Vector) + CI	Генерация CDD/ARXML
14	Security	Подпись прошивки + верификация загрузки	Secure Boot + CI
15	Комплексный	CI + HIL + OTA + SBOM	Полный пайплайн DevSecOps

Контрольные вопросы

1. Что такое кросс-компиляция и почему она необходима для встроенных систем?
2. Какие тулчейны используются для ARM Cortex-M микроконтроллеров?
3. В чем отличие .bin файла от .hex? Когда используется каждый формат?
4. Как настроить кэширование тулчейна в CI/CD?
5. Что такое linker script и какие изменения нужны при переходе с Keil на GCC?
6. Как запустить unit-тесты для embedded кода на хост-машине?
7. Какие инструменты статического анализа подходят для C/C++ embedded проектов?
8. Как добавить версию прошивки (git tag, commit hash) в бинарный файл?
9. Что такое SITL (Software-in-the-Loop) и HIL (Hardware-in-the-Loop) тестирование?
10. Как настроить OTA-обновления через CI/CD пайплайн?
11. Какие меры безопасности нужно применять при OTA-обновлениях?
12. Как организовать сборку для нескольких целевых платформ в одном пайплайне?
13. Что такое QEMU и как его использовать для тестирования прошивок?
14. Как проверить, что размер прошивки не превышает доступную flash память?
15. Какие артефакты сборки нужно сохранять для production-релизов?

Требования к отчету

Отчет должен содержать следующие разделы:

1. Титульный лист с указанием ФИО студента, группы, варианта.
2. Цель и задачи работы.

3. Теоретическая часть:

- Обоснование выбора тулчейна и целевой платформы.
- Описание процесса кросс-компиляции.

4. Практическая часть:

- Листинг Makefile/CMakeLists.txt.
- Листинг конфигурации CI/CD (GitHub Actions/GitLab CI).
- Листинг Dockerfile (если используется).

5. Результаты выполнения:

- Скриншоты сборки в CI/CD.
- Скриншоты опубликованных артефактов.
- Вывод команды `file firmware.bin` (подтверждение архитектуры).
- Размер прошивки (текстовый вывод).

6. Анализ результатов:

- Таблица размеров сегментов (.text, .data, .bss).
- Время сборки (с кэшем и без).
- Сравнение с IDE-сборкой (если применимо).

7. Ответы на контрольные вопросы (3-5 вопросов).

8. Вывод - достигнута ли цель, какие сложности возникли.

Критерии оценивания

Критерий	Базовый уровень	Средний уровень	Продвинутый уровень
Кросс-компилятор настроен	2 балла	2 балла	2 балла
Сборка прошивки успешна	2 балла	2 балла	2 балла
Артефакты (.bin/.hex) сохранены	2 балла	2 балла	2 балла
Unit-тесты на хосте	0 баллов	2 балла	2 балла
Статический анализ (cppcheck)	0 баллов	1 балл	2 балла
Docker-контейнер с тулчейном	0 баллов	1 балл	2 балла

HIL/SITL тестирование	0 баллов	0 баллов	2 балла
Качество отчета и ответы	2 балла	2 балла	2 балла

Шкала перевода баллов в оценку:

Оценка	Базовый (макс. 8)	Средний (макс. 12)	Продвинутый (макс. 16)
5	7-8	11-12	14-16
4	5-6	9-10	11-13
3	3-4	6-8	8-10
2	<3	<6	<8

Бонусные баллы (макс. +3):

- Настройка кэширования тулчейна - +1 балл .
- Интеграция с ОТА-репозиторием - +1 балл .
- Генерация SBOM (CycloneDX) - +1 балл.

Лабораторная работа №16. Интеграция симулятора в пайплайн (SITL)

Цель работы: Сформировать практические навыки настройки и интеграции симуляционных сред (Gazebo/Webots) в CI/CD-пайплайн для автоматизированного тестирования робототехнических и автономных систем с использованием подхода Software-in-the-Loop (SITL).

1.2. Задачи работы

1. Настройка headless-симуляции в CI-среде:

- Разработать Docker-образ с предустановленным симулятором (Gazebo/Webots) и ROS 2 .

- Настроить виртуальный дисплей (Xvfb) или EGL для headless-рендеринга .

- Обеспечить запуск симуляции без графического интерфейса.

2. Создание тестового сценария:

- Разработать простой мир/сцену для симуляции.

- Реализовать сценарий автономного движения (перемещение в заданную точку) .

- Интегрировать ROS 2 узлы для управления роботом.

3. Автоматизация тестирования в CI/CD:

- Настроить GitHub Actions/GitLab CI для запуска симуляции .

- Реализовать автоматический сбор метрик успешности теста.

- Настроить публикацию отчётов и логов как артефактов .

4. Анализ и валидация:

- Собрать метрики (время достижения цели, статус выполнения).

- Настроить срыв пайплайна при недостижении целевых показателей.

5. Оформление отчёта:

- Зафиксировать конфигурацию пайплайна и Docker-образа.

- Представить скриншоты процесса и результаты тестирования.

Теоретическое обоснование

1. Software-in-the-Loop (SITL) тестирование

SITL (Software-in-the-Loop) - метод тестирования, при котором исполняемый код (например, алгоритмы автономной навигации) запускается в симулированной среде, имитирующей физический мир, датчики и исполнительные механизмы робота .

Преимущества SITL в CI/CD:

Преимущество	Описание
Безопасность	Ошибки алгоритмов не приводят к повреждению оборудования
Воспроизводимость	Каждый тест выполняется в идентичных условиях
Масштабируемость	Возможность параллельного запуска множества тестов
Скорость	Симуляция может работать быстрее реального времени

Когда использовать SITL в CI:

- Тестирование алгоритмов навигации и планирования пути
- Валидация ROS 2 узлов перед развертыванием на реальном роботе
- Регрессионное тестирование после изменений в URDF/конфигурациях

2. Headless-симуляция: технические аспекты

Проблема: Симуляторы (Gazebo, Webots, Ignition) традиционно требуют наличия графического дисплея для рендеринга, что делает их сложными для запуска в CI-средах без GPU.

Решения:

Решение	Описание	Применение
Xvfb (X Virtual Framebuffer)	Эмуляция X11-дисплея в памяти, программный	Универсальное решение, работает на

	рендеринг	любом CPU
EGL Headless Mode	Рендеринг напрямую через EGL без оконной системы, аппаратное ускорение	Современные версии Gazebo (Ignition Fortress+)
Software Rendering (Mesa)	Программная реализация OpenGL через llvmpipe	Для CI без GPU, но медленнее

Пример запуска Gazebo с Xvfb:

```
# Запуск виртуального дисплея
Xvfb :99 -screen 0 1024x768x24 &
export DISPLAY=:99

# Запуск симуляции (headless)
ign gazebo -s --headless-rendering world.sdf
```

Важно: Для работы Gazebo в Docker-контейнере может потребоваться флаг `--privileged` для доступа к разделяемой памяти .

3. Интеграция симулятора с ROS 2

Типовая архитектура тестового пайплайна (рисунок 1):

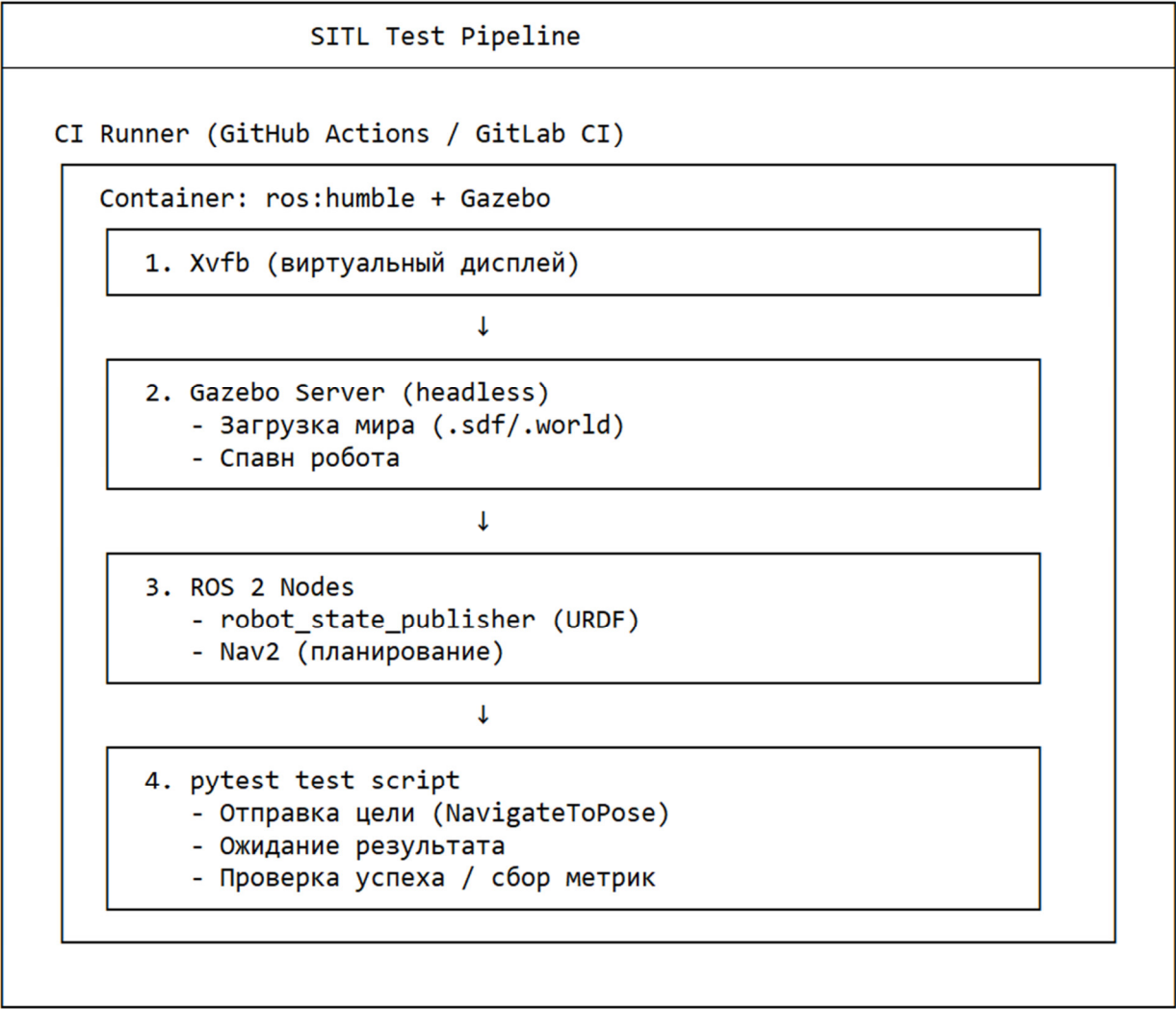


Рисунок 1

4. Инструменты для SITL-тестирования

Инструмент	Лицензия	Интеграция с ROS	Особенности
Gazebo (Ignition)	Apache 2.0	ros_ign_gazebo	Поддержка EGL headless, физика, сенсоры
Webots	Apache 2.0	webots_ros2	Встроенная поддержка ROS 2, CI-friendly
PX4 SITL	BSD	px4_ros_com	Специализирован для дронов
AirSim	MIT	airsim_ros_pkgs	Реалистичная

			физика, Unreal/Unity

5. Сбор метрик успешности

Ключевые метрики для SITL-теста:

Метрика	Описание	Целевое значение
Goal reached	Достигнута ли целевая точка	true/false
Time to goal	Время достижения цели	< 60 сек
Path length	Длина пройденного пути	< 1.5 × прямого расстояния
Collisions	Количество столкновений	0
Average velocity	Средняя скорость	> 0.5 м/с

Пример сбора метрик в pytest:

```
import pytest
import time

def test_robot_reaches_goal():
    start_time = time.time()
    success = navigate_to_goal(x=2.0, y=1.5)
    duration = time.time() - start_time

    assert success, "Robot failed to reach goal"
    assert duration < 60.0, f"Time exceeded: {duration}s"
```

Примеры практического выполнения

Пример: Gazebo + ROS 2 (GitHub Actions)

Файл Dockerfile.ci:

```
FROM ros:humble-ros-base

# Установка Gazebo и инструментов
RUN apt-get update && apt-get install -y \
    ros-humble-gazebo-ros-pkgs \
```

```
xvfb \
python3-pytest \
python3-colcon-common-extensions \
&& rm -rf /var/lib/apt/lists/*

# Установка virtual display
RUN echo "source /opt/ros/humble/setup.bash" >> /root/.bashrc

WORKDIR /ros_ws
```

Файл tests/test_navigation.py:

```
import pytest
import rclpy
from rclpy.action import ActionClient
from nav2_msgs.action import NavigateToPose

class TestNavigation:
    @pytest.fixture(autouse=True)
    def setup(self):
        rclpy.init()
        self.node = rclpy.create_node('test_node')
        self.client = ActionClient(self.node, NavigateToPose, 'navigate_to_pose')
        yield
        self.node.destroy_node()
        rclpy.shutdown()

    def test_robot_reaches_goal(self):
        # Ожидание сервера действий
        assert self.client.wait_for_server(timeout_sec=10.0), "Nav2 not ready"

        # Отправка цели
        goal = NavigateToPose.Goal()
        goal.pose.header.frame_id = 'map'
        goal.pose.pose.position.x = 2.0
        goal.pose.pose.position.y = 1.5
        goal.pose.pose.orientation.w = 1.0

        future = self.client.send_goal_async(goal)
        rclpy.spin_until_future_complete(self.node, future, timeout_sec=60.0)

        result = future.result()
        assert result.status == 4, f"Navigation failed with status {result.status}"
```

Файл .github/workflows/simulation.yml:

```
name: Simulation Tests

on:
```

```
pull_request:
  branches: [main]

jobs:
  gazebo-test:
    runs-on: ubuntu-22.04
    container:
      image: ghcr.io/your-org/robot-ci:latest
      options: --privileged

    steps:
      - uses: actions/checkout@v4

      - name: Build ROS workspace
        run: |
          source /opt/ros/humble/setup.bash
          colcon build --symlink-install

      - name: Run simulation tests
        run: |
          source /opt/ros/humble/setup.bash
          source /ros_ws/install/setup.bash

          # Start virtual display
          Xvfb :99 -screen 0 1024x768x24 &
          export DISPLAY=:99

          # Launch simulation in background
          ros2 launch my_pkg test_world.launch.py headless:=true &
          SIM_PID=$!

          # Wait for Gazebo to initialize
          sleep 20

          # Run tests
          pytest tests/test_navigation.py -v --junitxml=test-results.xml
          TEST_EXIT=$?

          kill $SIM_PID
          exit $TEST_EXIT

      - name: Upload test results
        uses: actions/upload-artifact@v4
        with:
          name: test-logs
          path: |
            test-results.xml
            /ros_ws/log/
```

2. Пример: Webots + ROS 2

Файл `.github/workflows/webots-ci.yml`:

```
name: Webots CI

on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-22.04
    container:
      image: cyberbotics/webots:R2025a-ubuntu22.04
      options: --privileged

    steps:
      - uses: actions/checkout@v4

      - name: Setup ROS 2
        run: |
          apt-get update && apt-get install -y ros-humble-webots-ros2

      - name: Build workspace
        run: |
          source /opt/ros/humble/setup.bash
          colcon build

      - name: Run Webots tests
        run: |
          # Webots headless mode with virtual display
          xvfb-run -a webots --mode=fast --no-rendering \
            --stdout --stderr worlds/test_world.wbt
```

3. Пример: PX4 SITL (дрон)

Файл `.github/workflows/px4-sitl.yml`:

```
name: PX4 SITL Tests

on:
  push:
    branches: [main]

jobs:
  px4-test:
    runs-on: ubuntu-22.04
```



```

steps:
  - uses: actions/checkout@v4

  - name: Run PX4 SITL simulation
    run: |
      # Запуск симуляции PX4
      make px4_sitl gazebo

      # Отправка миссии через MAVSDK
      python3 scripts/run_mission.py --takeoff-alt 10 --waypoints
      "10,10;20,20"

      # Проверка успешности выполнения миссии
      python3 scripts/check_mission.py --expect-success

```

4. Сборка Docker-образа для CI

```

# Сборка образа с Gazebo и ROS 2
docker build -f Dockerfile.ci -t ghcr.io/your-org/robot-ci:latest .

# Публикация в GitHub Container Registry
docker push ghcr.io/your-org/robot-ci:latest

```

Варианты заданий

Базовый уровень

Общее требование: Настроить SITL-тест с простым перемещением робота в симуляторе. Реализовать автоматический запуск в CI. Сохранить артефакты (логи, результаты).

№	Симулятор	Платформа/Робот	Сценарий
1	Gazebo (Ignition)	TurtleBot3	Перемещение на 2 метра вперёд
2	Gazebo (Ignition)	Diff-drive robot	Поворот на 90 градусов
3	Gazebo (Classic)	Husky	Объезд препятствия
4	Webots	e-puck	Движение по квадрату

5	Webots	TurtleBot3	Перемещение к маркеру
6	Webots	Nao robot	Ходьба вперёд (5 шагов)
7	PX4 SITL	Quadrotor	Взлёт на 5 метров
8	PX4 SITL	Quadrotor	Полёт по квадрату
9	ArduPilot SITL	Rover	Движение к точке GPS
10	Gazebo + ROS 2	Custom robot	Подъезд к зарядной станции
11	Gazebo + ROS 2	Robot with arm	Захват объекта (grasp)
12	Webots + ROS 2	Amazon Kiva	Движение по складу
13	Gazebo	PR2	Открытие двери (simplified)
14	Webots	Pioneer 3AT	Следование по линии
15	Gazebo + LIDAR	Differential drive	Обнаружение и остановка перед стеной

Средний уровень

Общее требование: Настроить сложный сценарий с навигацией (move_base/Nav2). Реализовать сбор метрик (время, путь, коллизии). Настроить автоматическую публикацию отчёта.

№	Симулятор	Стек навигации	Сценарий	Метрики
1	Gazebo	Nav2 (ROS 2)	Навигация между 3 точками	Время, путь, столкновения
2	Gazebo	move_base (ROS 1)	Обход динамического препятствия	Успех, время перепланирования
3	Webots	Webots controller	Поиск и захват	Успех, время

			объекта	
4	Gazebo + SLAM	gmapping + Nav2	Исследование неизвестной области	Покрытие карты
5	PX4 SITL	Offboard control	Пролёт через ворота	Точность прохождения
6	Gazebo	Navigation waypoints +	Парковка в заданную зону	Точность финальной позиции
7	Webots	Pure pursuit	Движение по гоночной трассе	Время круга
8	Gazebo + AprilTag	Nav2 + визуальная локализация	Навигация по тегам	Точность позиционирования
9	PX4 SITL	Mission mode	Выполнение миссии из 5 waypoints	Время, расход энергии
10	Gazebo + RGB-D	Nav2 + RTAB-Map	Навигация в помещении	Карта, позиционирование
11	Webots + ROS 2	ROS 2 Control	Отслеживание траектории	Ошибка слежения
12	Gazebo + ROS 2	TEB Planner	Проезд узкого коридора	Успех, время
13	ArduPilot SITL	Auto mode	Облёт препятствий	Успех, минимальная дистанция
14	Gazebo + ROS 2	DWB Controller	Следование за человеком	Сохранение дистанции
15	Webots	PID controller	Балансировка	Время удержания

			(Segway)	

Продвинутый уровень

Общее требование: Настроить параллельный запуск нескольких симуляций (matrix strategy). Реализовать анализ метрик и автоматический откат при превышении порогов. Интегрировать с системой визуализации (Grafana).

№	Симулятор	Сценарий	Параметры тестирования	Анализ
1	Gazebo	Навигация с разными параметрами планировщика	5 конфигураций	Сравнение времени
2	Gazebo	Тестирование при разных погодных условиях	Туман, дождь, ночь	Успешность по условиям
3	PX4 SITL	Полёт с разными ветровыми нагрузками	Скорость ветра 0-15 м/с	Стабильность, дрейф
4	Webots	Массовое тестирование датчиков	10 конфигураций датчиков	Точность, шум
5	Gazebo	А/В тестирование алгоритмов навигации	2 алгоритма × 10 сценариев	Статистический анализ

6	PX4 SITL + AWS	Параллельное выполнение 5 миссий	5 EC2 инстансов	Сравнение результатов
7	Gazebo + ROS 2	Регрессионное тестирование	Коммиты main ветки	Детекция регрессий
8	Gazebo + CI	Тестирование с нагрузкой до 100 симуляций	100 параллельных процессов	Масштабируемость
9	Специфика БПЛА	PX4 + Gazebo + ROS 2	Полная миссия: взлёт-облёт-посадка	20+ метрик
10	Gazebo + CV	Навигация с камерой и YOLO	Распознавание объектов	Точность детекции
11	Webots + RL	Обучение с подкреплением в CI	1000 эпизодов обучения	Графики наград
12	Gazebo + K8s	Оркестрация симуляций в кластере	Масштабирование до 50 подов	Cost/performance
13	Gazebo + Prometheus	Сбор метрик в реальном времени	Grafana дашборд	Визуализация CI/CD
14	PX4 SITL + Data Analysis	Анализ bag-файлов из CI	Автоматическая генерация отчёта	Сравнение версий
15	Комплексный	Полный CI/CD: сборка → симуляция	End-to-end pipeline	Full traceability

		→ анализ → деплой		

Контрольные вопросы

1. Что такое SITL (Software-in-the-Loop) тестирование и чем оно отличается от HITL?
2. Почему для запуска симуляции в CI необходим виртуальный дисплей (Xvfb)?
3. Какие преимущества даёт headless-режим в Gazebo (Ignition) с EGL?
4. Как проверить, что симуляция успешно запустилась в headless-режиме?
5. Какие метрики можно собирать при тестировании автономной навигации?
6. Как настроить автоматический срыв пайплайна при недостижении целевой точки?
7. В чем отличие между Gazebo Classic и Ignition Gazebo?
8. Как настроить Docker-образ с Gazebo и ROS 2 для CI/CD?
9. Что такое `--privileged` флаг в Docker и почему он может потребоваться?
10. Как собрать и сохранить логи симуляции в артефакты?
11. Как организовать параллельный запуск нескольких тестовых сценариев?
12. Какие ограничения SITL-тестирования нужно учитывать?
13. Как интегрировать симуляцию с навигационным стеком Nav2?
14. Что такое `sleep` в тестовом пайплайне и как его можно заменить на динамическое ожидание?
15. Как автоматизировать сбор и анализ bag-файлов из CI?

Требования к отчету

Отчет должен содержать следующие разделы:

1. Титульный лист с указанием ФИО студента, группы, варианта.

2. Цель и задачи работы.

3. Теоретическая часть:

- Описание SITL-подхода и его места в CI/CD робототехнических систем.

- Обоснование выбора симулятора и конфигурации headless-режима.

- Схема тестового пайплайна.

4. Практическая часть:

- Листинг Dockerfile для CI-образа.

- Листинг тестового сценария (launch-файлы, pytest).

- Листинг конфигурации CI/CD (GitHub Actions/GitLab CI).

- Скрипт сбора метрик (если разрабатывался).

5. Результаты выполнения:

- Скриншоты успешного выполнения пайплайна.

- Скриншоты артефактов (логи, отчёты, bag-файлы).

- Таблица собранных метрик.

6. Анализ результатов:

- Время выполнения тестов.

- Анализ стабильности симуляции.

- Проблемы и их решение.

7. Ответы на контрольные вопросы (3-5 вопросов).

8. Вывод — достигнута ли цель, какие навыки приобретены.

Критерии оценивания

Критерий	Базовый уровень	Средний уровень	Продвинутый уровень
Headless-симуляция запущена в CI	2 балла	2 балла	2 балла

Тестовый сценарий выполнен успешно	2 балла	2 балла	2 балла
Собраны и сохранены артефакты (логи)	2 балла	2 балла	2 балла
Настроен сбор метрик (время, статус)	0 баллов	2 балла	2 балла
Docker-образ для CI создан и опубликован	0 баллов	1 балл	2 балла
Пайплайн падает при недостижении метрик	0 баллов	1 балл	2 балла
Параллельный запуск симуляций	0 баллов	0 баллов	2 балла
Качество отчета и ответы	2 балла	2 балла	2 балла

Шкала перевода баллов в оценку:

Оценка	Базовый (макс. 8)	Средний (макс. 12)	Продвинутый (макс. 16)
Отлично (5)	7-8	11-12	14-16
Хорошо (4)	5-6	9-10	11-13
Удовлетворительно (3)	3-4	6-8	8-10
Неудовлетворительно (2)	<3	<6	<8

Бонусные баллы (макс. +3):

- Интеграция с системой визуализации (Grafana) - +1 балл.
- Автоматический анализ bag-файлов - +1 балл.
- Настройка динамического ожидания вместо sleep - +1 балл.

Лабораторная работа №17. Безопасные OTA-обновления

Цель работы: Сформировать практические навыки создания безопасного конвейера для обновлений «по воздуху» (Over-the-Air, OTA), включая сборку прошивки, генерацию цифровой подписи (Cosign) или хэш-суммы, публикацию OTA-пакета на сервере обновлений и (опционально) симуляцию получения обновления устройством.

Теоретическое обоснование

1. OTA-обновления и их безопасность

OTA (Over-the-Air) обновления - это механизм удалённой доставки и установки новых версий прошивки или программного обеспечения на устройства без физического подключения.

Ключевые угрозы безопасности OTA:

Угроза	Описание	Последствия
Подмена пакета	Злоумышленник подменяет прошивку на этапе передачи	Установка вредоносного ПО
Откат версии (rollback)	Установка старой уязвимой версии	Эксплуатация исправленных уязвимостей
Перехват и анализ	Прослушивание канала передачи	Утечка конфиденциальных данных
Компрометация ключей	Утечка приватного ключа подписи	Полная компрометация системы обновлений

Меры безопасности OTA:

Мера	Назначение	Реализация

Криптографическая подпись	Гарантия подлинности и целостности пакета	Cosign, GPG, Minisign
HTTPS/TLS 1.3	Защита канала передачи	Шифрование трафика
End-to-End Encryption (E2EE)	Сквозное шифрование пакета	Защита от сервера обновлений
Атомарное обновление	Защита от повреждения устройства	A/B-разделы, rollback
Защита от отката	Блокировка установки старых версий	Счётчик версий, подпись

2. Инструменты для подписи артефактов

Cosign (Sigstore) - инструмент для подписи контейнеров и артефактов с поддержкой прозрачности.

Особенность	Описание
Поддержка ключей	Локальные ключи, KMS, ключи OpenID Connect (OIDC)
Форматы подписи	DSSE, in-toto attestations
Интеграция с CI/CD	GitHub Actions, GitLab CI, Jenkins
Прозрачность	Интеграция с Rekor (прозрачный лог)

Пример подписи с Cosign:

```
# Генерация ключевой пары
cosign generate-key-pair

# Подпись артефакта
cosign sign-blob --key cosign.key firmware.bin > firmware.bin.sig

# Верификация
cosign verify-blob --key cosign.pub --signature firmware.bin.sig firmware.bin
```

Minisign - простая утилита для подписи файлов, используемая в OTA-системах.

GPG - универсальный инструмент для шифрования и подписи.

3. The Update Framework (TUF)

TUF (The Update Framework) - открытый фреймворк для защиты систем обновлений, обеспечивающий защиту от атак отката, компрометации ключей и подмены метаданных.

Ключевые особенности TUF:

Функция	Описание
М-of-N подпись	Множественные подписи для утверждения обновлений
Ротация ключей	Безопасная смена ключей без обновления клиентов
Делегирование	Разные команды управляют своими компонентами
Защита от отката	Нельзя установить старую версию после обновления

Проекты на основе TUF:

- Sigstore - использует TUF для защиты корневых сертификатов.
- TUF-on-CI - интеграция TUF с CI/CD пайплайнами.
- RSTUF - масштабируемый TUF для реестров пакетов.

4. Архитектура безопасного OTA-пайплайна (рисунок 1)

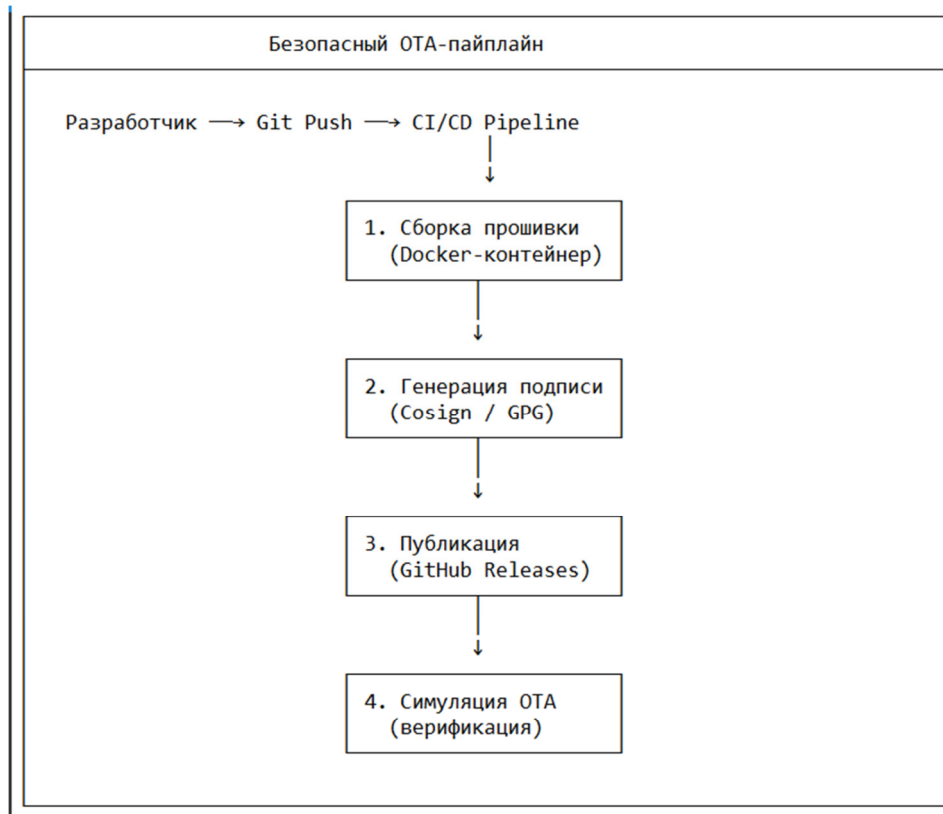


Рисунок 1

5. Безопасность CI/CD пайплайна

Ключевые меры безопасности:

Мера	Описание
Управление секретами	Хранение ключей в CI/CD секретах (GitHub Secrets, GitLab Variables)
Изолированные сборки	Использование Docker-контейнеров для воспроизводимости
RBAC (Role-Based Access Control)	Разграничение прав на публикацию обновлений
Автоматическое сканирование уязвимостей	SAST, SCA, DAST перед подписью
Аудит и логирование	Отслеживание всех операций с обновлениями

6. Процесс подписи и верификации

Подпись OTA-пакета (на стороне CI):

```
# 1. Генерация ключей (однократно)
cosign generate-key-pair

# 2. Подпись артефакта
cosign sign-blob --key cosign.key firmware.bin > firmware.bin.sig

# 3. Генерация манифеста
cat > manifest.json << EOF
{
  "version": "1.2.0",
  "url": "https://releases.example.com/firmware.bin",
  "sha256": "$(sha256sum firmware.bin | cut -d' ' -f1)",
  "signature": "$(cat firmware.bin.sig)",
  "pubkey_url": "https://releases.example.com/cosign.pub"
}
EOF
```

Верификация на устройстве (симуляция):

```
# 1. Загрузка манифеста
curl -O https://releases.example.com/manifest.json

# 2. Загрузка прошивки
curl -O $(jq -r '.url' manifest.json)

# 3. Проверка хэша
echo "$(jq -r '.sha256' manifest.json) firmware.bin" | sha256sum --check

# 4. Проверка подписи
curl -O $(jq -r '.pubkey_url' manifest.json)
cosign verify-blob --key cosign.pub --signature <(jq -r '.signature'
manifest.json) firmware.bin
```

Примеры практического выполнения

1. Пример: OTA-пайплайн с Cosign (GitHub Actions)

Файл.github/workflows/ota-release.yml:

```

name: Secure OTA Release

on:
  push:
    tags:
      - 'v*'

env:
  FIRMWARE_FILE: firmware.bin
  ARTIFACT_DIR: artifacts

jobs:
  build-and-sign:
    runs-on: ubuntu-latest
    permissions:
      contents: write
      packages: write
      id-token: write # для OIDC

    steps:
      - uses: actions/checkout@v4

      - name: Set up Cosign
        uses: sigstore/cosign-installer@v3
        with:
          cosign-release: 'v2.2.4'

      - name: Build firmware
        run: |
          mkdir -p ${ env.ARTIFACT_DIR }
          # Кросс-компиляция прошивки
          make clean && make all
          cp build/firmware.bin ${ env.ARTIFACT_DIR }/${ env.FIRMWARE_FILE }

      - name: Generate version info
        run: |
          VERSION=${GITHUB_REF_NAME#v}
          echo "VERSION=$VERSION" >> $GITHUB_ENV
          echo "VERSION=$VERSION" > ${ env.ARTIFACT_DIR }/version.txt
          echo "COMMIT=${GITHUB_SHA::7}" >> ${ env.ARTIFACT_DIR }/version.txt
          echo "BUILD_TIME=$(date -u +%Y-%m-%dT%H:%M:%SZ)" >> ${ env.ARTIFACT_DIR }/version.txt

      - name: Generate key pair (if not exists)
        run: |
          if [ ! -f cosign.key ]; then
            cosign generate-key-pair
          fi

```

```

- name: Sign firmware with Cosign
  run: |
    # Подпись артефакта
    cosign sign-blob \
      --key cosign.key \
      ${ env.ARTIFACT_DIR }/${ env.FIRMWARE_FILE } \
      > ${ env.ARTIFACT_DIR }/${ env.FIRMWARE_FILE }.sig

- name: Generate manifest
  run: |
    cat > ${ env.ARTIFACT_DIR }/manifest.json << EOF
    {
      "version": "${ env.VERSION }",
      "commit": "${ github.sha }",
      "build_time": "$(date -u +'%Y-%m-%dT%H:%M:%SZ')",
      "firmware": "${ env.FIRMWARE_FILE }",
      "firmware_url": "https://github.com/${ github.repository
    }}/releases/download/${ github.ref_name }/${ env.FIRMWARE_FILE }",
      "signature_url": "https://github.com/${ github.repository
    }}/releases/download/${ github.ref_name }/${ env.FIRMWARE_FILE }.sig",
      "sha256": "$(sha256sum ${ env.ARTIFACT_DIR }/${ env.FIRMWARE_FILE
    }} | cut -d' ' -f1)",
      "pubkey_url": "https://raw.githubusercontent.com/${ {
    github.repository }}/main/cosign.pub"
    }
    EOF

- name: Create GitHub Release
  uses: softprops/action-gh-release@v1
  with:
    files: |
      ${ env.ARTIFACT_DIR }/*
    body: |
      ## OTA Release ${ env.VERSION }
      - **Commit:** ${ github.sha }
      - **Build:** ${ env.BUILD_TIME }
      - **Signature:** Cosign (Sigstore)

- name: Verify signature (self-test)
  run: |
    cosign verify-blob \
      --key cosign.pub \
      --signature ${ env.ARTIFACT_DIR }/${ env.FIRMWARE_FILE }.sig \
      ${ env.ARTIFACT_DIR }/${ env.FIRMWARE_FILE }
    echo " Signature verification passed!"

```

2. Пример: OTA-пайплайн с Memfault (специализированная OTA-платформа)

Файл `.github/workflows/memfault-ota.yml`:

```
name: Memfault OTA Release

on:
  push:
    tags:
      - 'v*'

env:
  HARDWARE_VERSION: "mp"
  SOFTWARE_TYPE: "firmware"
  COHORT: "auto-ota"

jobs:
  memfault-release:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Build firmware (Zephyr example)
        run: |
          west build -b nrf52840dk_nrf52840
          cp build/zephyr/zephyr.elf firmware.elf
          cp build/zephyr/app_update.bin firmware.bin

      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.10'

      - name: Install Memfault CLI
        run: pip install memfault-cli

      - name: Upload symbols and create OTA release
        run: |
          # Upload debug symbols
          memfault \
            --org-token ${ secrets.MEMFAULT_ORG_TOKEN } \
            --org ${ secrets.MEMFAULT_ORG } \
            --project ${ secrets.MEMFAULT_PROJECT } \
            upload-mcu-symbols firmware.elf

          # Upload OTA payload
          memfault \
            --org-token ${ secrets.MEMFAULT_ORG_TOKEN } \
```



```

--org ${ secrets.MEMFAULT_ORG } \
--project ${ secrets.MEMFAULT_PROJECT } \
upload-ota-payload \
--hardware-version ${ env.HARDWARE_VERSION } \
--software-type ${ env.SOFTWARE_TYPE } \
--software-version ${ github.ref_name } \
firmware.bin

# Deploy release to cohort
memfault \
--org-token ${ secrets.MEMFAULT_ORG_TOKEN } \
--org ${ secrets.MEMFAULT_ORG } \
--project ${ secrets.MEMFAULT_PROJECT } \
deploy-release \
--release-version ${ github.ref_name } \
--cohort ${ env.COHORT }

```

3. Пример: Симулятор устройства для проверки ОТА

Файл `simulate_ota.py`:

```

#!/usr/bin/env python3
"""
Симулятор устройства для проверки ОТА-обновлений
"""

import json
import hashlib
import requests
import subprocess
import os

class OTASimulator:
    def __init__(self, manifest_url, pubkey_path):
        self.manifest_url = manifest_url
        self.pubkey_path = pubkey_path
        self.current_version = None

    def check_for_updates(self):
        """Проверка наличия обновлений"""
        response = requests.get(self.manifest_url)
        manifest = response.json()

        print(f"📄 Доступна версия: {manifest['version']}")
        print(f"    Commit: {manifest['commit']}")

```

```

    print(f"    Build: {manifest['build_time']}")

    return manifest

def verify_firmware(self, firmware_path, manifest):
    """Верификация прошивки"""
    # 1. Проверка хэша
    sha256 = hashlib.sha256()
    with open(firmware_path, 'rb') as f:
        for chunk in iter(lambda: f.read(4096), b''):
            sha256.update(chunk)

    if sha256.hexdigest() != manifest['sha256']:
        raise ValueError("❌ Хэш не совпадает! Файл повреждён.")
    print("✅ Хэш совпадает")

    # 2. Проверка подписи (Cosign)
    result = subprocess.run([
        'cosign', 'verify-blob',
        '--key', self.pubkey_path,
        '--signature', manifest['signature_url'],
        firmware_path
    ], capture_output=True, text=True)

    if result.returncode != 0:
        raise ValueError(f"❌ Подпись недействительна: {result.stderr}")
    print("✅ Подпись действительна")

def download_firmware(self, url, output_path):
    """Загрузка прошивки"""
    print(f"📄 Загрузка прошивки из {url}")
    response = requests.get(url, stream=True)
    with open(output_path, 'wb') as f:
        for chunk in response.iter_content(chunk_size=8192):
            f.write(chunk)
    print("✅ Загрузка завершена")

def install_firmware(self, firmware_path):
    """Установка прошивки (симуляция)"""
    print("🔧 Установка прошивки...")
    # Здесь был бы реальный процесс прошивки
    # Например: dfu-util, flash_tool, etc.
    print("✅ Установка завершена")

def run(self):
    """Основной цикл симулятора"""
    print("🏠 Симулятор OTA устройства запущен")

    # Проверка обновлений
    manifest = self.check_for_updates()

```

```

# Загрузка прошивки
firmware_path = "/tmp/firmware.bin"
self.download_firmware(manifest['firmware_url'], firmware_path)

# Верификация
self.verify_firmware(firmware_path, manifest)

# Установка
self.install_firmware(firmware_path)

print(f"🔧 Устройство обновлено до версии {manifest['version']}")

if __name__ == "__main__":
    import sys
    if len(sys.argv) < 2:
        print("Usage: simulate_ota.py <manifest_url>")
        sys.exit(1)

    simulator = OTASimulator(sys.argv[1], "cosign.pub")
    simulator.run()

```

Методические указания к выполнению практической части работы

Задачи работы:

1. Настройка безопасного конвейера сборки прошивки:
 - Настроить кросс-компиляцию прошивки для целевого устройства.
 - Обеспечить детерминированную воспроизводимость сборки (Docker-контейнер).
 - Добавить метаданные в прошивку (версия, время сборки, commit hash).
2. Генерация цифровой подписи OTA-пакета:
 - Сгенерировать ключевую пару для подписи (Cosign, GPG, Minisign).
 - Подписать OTA-пакет (прошивку) с использованием приватного ключа.
 - Сохранить подпись и публичный ключ для верификации.
3. Публикация OTA-пакета на сервере обновлений:
 - Настроить CI/CD для публикации подписанного пакета.

- Публикация в реестр артефактов (GitLab Package Registry, GitHub Releases, S3).

- Генерация манифеста обновления (версия, URL, хэш-сумма, подпись).

4. Симуляция получения обновления устройством (опционально):

- Реализовать скрипт-симулятор устройства, проверяющий обновления.

- Выполнить верификацию подписи OTA-пакета.

- Промоделировать процесс обновления и отката.

5. Оформление отчёта:

- Зафиксировать конфигурацию пайплайна и процесс подписи.

- Представить скриншоты подписанных артефактов.

Варианты заданий

Базовый уровень

Общее требование: Собрать прошивку, сгенерировать хэш-сумму (SHA256), опубликовать OTA-пакет в GitHub Releases. Создать простой манифест обновления.

№	Целевая платформа	Формат прошивки	Способ подписи
1	STM32F103 (Blue Pill)	.bin	SHA256 + хэш в манифесте
2	ESP32	.bin	SHA256 + хэш в манифесте
3	Raspberry Pi Pico	.uf2	SHA256 + хэш в манифесте
4	nRF52840	.hex	SHA256 + хэш в манифесте
5	AVR (Arduino Uno)	.hex	SHA256 + хэш в манифесте
6	ARM (Cortex-M)	.bin	SHA256 + хэш в манифесте
7	RISC-V (SiFive)	.bin	SHA256 + хэш в манифесте

8	Linux ARM (Raspberry Pi)	.tar.gz	SHA256 + хэш в манифесте
9	Zephyr RTOS (any)	.bin	SHA256 + хэш в манифесте
10	FreeRTOS (any)	.bin	SHA256 + хэш в манифесте
11	MyNewt (any)	.img	SHA256 + хэш в манифесте
12	Apache NuttX	.bin	SHA256 + хэш в манифесте
13	RIOT OS	.bin	SHA256 + хэш в манифесте
14	Mbed OS	.bin	SHA256 + хэш в манифесте
15	Custom C++ app (x86)	.exe/.elf	SHA256 + хэш в манифесте

2. Средний уровень

Общее требование: Настроить подпись OTA-пакета с помощью Cosign (локальные ключи). Реализовать верификацию подписи в симуляторе устройства. Публикация манифеста и подписи в GitHub Releases.

№	Целевая платформа	Инструмент подписи	Дополнительные требования
1	STM32F407	Cosign	Верификация в симуляторе
2	ESP32	Cosign	Автоматическая генерация ключей
3	nRF52840	GPG	Интеграция с GitHub Secrets
4	Raspberry Pi Pico	Minisign	Хранение ключей в CI
5	Zephyr + nRF	Cosign + Memfault	Публикация символов
6	Linux ARM	Cosign	OTA для edge-устройств
7	FreeRTOS + STM32	GPG	Атомарное обновление (A/B)
8	ESP32	Cosign + OTA HTTPS	Шифрование канала
9	STM32 + MCUBoot	Cosign + signed images	Интеграция с bootloader

10	Raspberry Pi	Cosign + apt	ОТА для Linux-пакетов
11	Zephyr + qemu	Cosign	Симуляция на QEMU
12	MyNewt + nRF	GPG	ОТА через BLE
13	Apache NuttX + STM32	Minisign	Размер прошивки < 256KB
14	RIOT OS + native	Cosign	Симуляция на хосте
15	Custom app + Docker	Cosign	ОТА для контейнера

Продвинутый уровень

Общее требование: Реализовать полный безопасный ОТА-пайплайн: сборка → подпись (Cosign с OIDC) → публикация → симуляция устройства с верификацией. Настроить защиту от отката (rollback protection). Интегрировать TUF для управления метаданными.

№	Платформа	Особенности	Требования
	Zephyr + nRF	MCUBoot + TUF	Защита от отката
	ESP32	ESP-IDF OTA + Cosign	Атомарное обновление
	STM32	TF-M + Secure Boot	Аппаратная защита
	Linux ARM	SWUpdate + Cosign	Полный rootfs OTA
	Zephyr + QEMU	TUF-on-CI	Интеграция TUF в CI
	ESP32	HTTPS + mTLS	Двусторонняя аутентификация
	nRF52840	BLE OTA + подпись	Ограниченный канал
	STM32 + FreeRTOS	ОТА через UART	Симуляция на хосте

	Raspberry Pi	RAUC + Cosign	A/B-разделы
	Zephyr + Golioth	Golioth OTA + Cosign	Облачная OTA-платформа
	Специфика БПЛА	PX4 + OTA	Полётный контроллер
	ROS 2 + ARM	OTA для ROS-нод	Обновление без остановки
	Multi-target	3+ платформы	Матричная сборка + подпись
	Full-stack	CI + HIL + OTA + SBOM	Полный DevSecOps
	Enterprise	Cosign + Vault + TUF	KMS, ротация ключей

Контрольные вопросы

1. Какие основные угрозы существуют для OTA-обновлений и как они защищаются?
2. В чем различие между хэш-суммой (SHA256) и цифровой подписью?
3. Для чего нужен манифест обновления? Какие поля он должен содержать?
4. Как работает Cosign? Какие форматы подписи поддерживает?
5. Что такое The Update Framework (TUF) и какие атаки он предотвращает?
6. Как настроить автоматическую подпись OTA-пакета в GitHub Actions?
7. Как защитить приватный ключ подписи в CI/CD?
8. Что такое атака отката (rollback attack) и как от неё защититься?
9. Как симулировать получение OTA-обновления устройством?
10. Как интегрировать верификацию подписи в процесс установки прошивки?

11. Какие инструменты поддерживают A/B-разделы для безопасного OTA?
12. Что такое Memfault и как он упрощает OTA-процесс?
13. Как настроить поэтапный rollout обновления (cohorts/canary)?
14. Какие метрики нужно отслеживать для мониторинга OTA-обновлений?
15. Как обеспечить воспроизводимость сборки для детерминированной подписи?

Требования к отчету

Отчет должен содержать следующие разделы:

1. Титульный лист с указанием ФИО студента, группы, варианта.
2. Цель и задачи работы.
3. Теоретическая часть:
 - Описание угроз безопасности OTA.
 - Обоснование выбора инструментов подписи.
 - Схема безопасного OTA-пайплайна.
4. Практическая часть:
 - Листинг конфигурации CI/CD (сборка, подпись, публикация).
 - Листинг скрипта симуляции устройства (если применимо).
 - Пример манифеста обновления.
5. Результаты выполнения:
 - Скриншоты успешной сборки и подписи в CI.
 - Скриншоты опубликованных OTA-пакетов в GitHub Releases.
 - Вывод симулятора устройства (верификация подписи).
 - Логи проверки хэша и подписи.
6. Анализ результатов:
 - Таблица размеров прошивки и подписи.
 - Время выполнения пайплайна.

- Анализ уязвимостей, закрытых подписью.

7. Ответы на контрольные вопросы (3-5 вопросов).

8. Вывод — достигнута ли цель, какие навыки приобретены.

Критерии оценивания

Критерий	Базовый уровень	Средний уровень	Продвинутый уровень
Прошивка собрана и опубликована	2 балла	2 балла	2 балла
Сгенерирована хэш-сумма/подпись	2 балла	2 балла	2 балла
Создан манифест обновления	2 балла	2 балла	2 балла
Подпись выполнена (Cosign/GPG)	0 баллов	2 балла	2 балла
Верификация подписи в симуляторе	0 баллов	1 балл	2 балла
Интеграция с GitHub Releases	1 балл	2 балла	2 балла
Защита от отката (rollback)	0 баллов	0 баллов	2 балла
Качество отчета и ответы	2 балла	2 балла	2 балла

Шкала перевода баллов в оценку:

Оценка	Базовый (макс. 9)	Средний (макс. 13)	Продвинутый (макс. 16)
Отлично (5)	8-9	12-13	14-16
Хорошо (4)	6-7	10-11	11-13
Удовлетворительно (3)	4-5	7-9	8-10
Неудовлетворительно (2)	<4	<7	<8

Бонусные баллы (макс. +3):

- Интеграция с TUF (The Update Framework) - +1 балл.

- A/B-разделы с атомарным обновлением - +1 балл.

- Подпись с аппаратным ключом (YubiKey) - +1 балл.

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

по организации и проведению самостоятельной работы по дисциплине
«Проектирование и оптимизация конвейеров CI/CD» для студентов
направления подготовки

09.04.04 Программная инженерия Направленность (профиль)
«Разработка, мониторинг и управление жизненным циклом инженерных
систем»

Ставрополь 2026

ВВЕДЕНИЕ

Целью самостоятельной работы является закрепление и углубление теоретических знаний, полученных на лекциях, а также формирование практических навыков проектирования, реализации и оптимизации сквозного CI/CD-пайплайна для автономной робототехнической системы (на примере БПЛА или наземного робота) в рамках выполнения индивидуального сквозного проекта.

Сквозной проект выполняется студентом индивидуально или в малых группах (2-3 человека) в течение всего семестра параллельно с изучением дисциплины и выполнением лабораторных работ. Защита проекта проводится на 18-м лабораторном занятии.

Тема проекта: «Разработка и внедрение CI/CD-пайплайна для автономной системы (БПЛА/робот) с обеспечением безопасности, тестирования в симуляции и развертывания».

Результат проекта: Работающий CI/CD-пайплайн в выбранной CI-платформе (GitLab CI / GitHub Actions), включающий все этапы: от компиляции до развертывания, с документацией и демонстрацией.

Рекомендуемые платформы для выполнения:

- CI/CD: GitLab CI (рекомендуется для образовательных целей) или GitHub Actions.
- Оркестрация: Minikube / Kind / K3s (локальный Kubernetes).
- Симуляция: Gazebo (с ROS 2) или Webots.
- Реестр артефактов: GitLab Package Registry / GitHub Packages / Docker Hub.
- Язык программирования: Python / C++ / Go (на выбор студента).

1. ФОРМИРУЕМЫЕ КОМПЕТЕНЦИИ

В результате выполнения самостоятельной работы и защиты сквозного проекта студент должен обладать следующими компетенциями:

Код	Компетенция	Уровень освоения
ПК-1	Способен проектировать, реализовывать и оптимизировать сквозные конвейеры CI/CD	Высокий
ПК-3	Способен реализовывать стратегии развертывания (canary, blue-green, rolling update)	Высокий

2. ТЕМЫ ДЛЯ САМОСТОЯТЕЛЬНОГО ИЗУЧЕНИЯ

В ходе выполнения сквозного проекта студенту рекомендуется самостоятельно изучить следующие темы:

№	Тема для самостоятельного изучения	Рекомендуемые источники	Трудоёмкость (часы)
1	Основы работы с Git (ветки, теги, подмодули)	[1, гл. 2-4]	4
2	Базовый синтаксис YAML для CI/CD конфигураций	[2, раздел «Pipeline syntax»]	2
3	Установка и настройка Minikube / Kind	[3, «Getting started»]	4
4	Основы Docker: Dockerfile, слои, кэширование	[4, «Best practices»]	6
5	Основы ROS 2 (ноды, топики, сервисы, launch-файлы)	[5, «Tutorials»]	8
6	Настройка headless-	[6, «Headless rendering»]	6

	симуляции Gazebo (Xvfb, EGL)		
7	Основы безопасности CI/CD (управление секретами, SAST, SCA)	[7, «DevSecOps»]	6
8	Основы Cosign для подписи артефактов	[8, «Cosign quickstart»]	4
9	Основы мониторинга (Prometheus, Grafana)	[9, «Getting started»]	6
10	GitOps с Argo CD (базовые концепции)	[10, «Argo CD overview»]	6

Общая трудоёмкость самостоятельной работы: 108 часа (примерно 6-8 часов в неделю).

3. ТАБЛИЦА СООТВЕТСТВИЯ ЛАБОРАТОРНЫХ РАБОТ И ЭТАПОВ ПРОЕКТА

Сквозной проект выполняется параллельно с лабораторными работами. Каждая лабораторная работа добавляет новый этап в пайплайн.

№ ЛР	Тема лабораторной работы	Этап сквозного проекта	Результат этапа
1	Настройка базового CI/CD-пайплайна	Настройка CI-платформы, создание репозитория	Базовый пайплайн (checkout, сборка, тесты)
2	Pipeline as Code и структурирование	Организация пайплайна (stage build/test/package)	Многостадийный пайплайн

3	Параллельное выполнение и оптимизация	Оптимизация времени выполнения	Кэширование, параллелизация, шардирование
4	Автоматизация модульного и интеграционного тестирования	Добавление unit и интеграционных тестов	Отчёты JUnit, покрытие
5	Анализ покрытия кода тестами	Настройка порога покрытия	Fail pipeline при <80%
6	Статический анализ безопасности (SAST)	Добавление Semgrep/Bandit	Отчёт об уязвимостях, Quality Gate
7	Анализ зависимостей (SCA)	Добавление OWASP DC / Snyk	Сканирование зависимостей
8	Сборка и публикация Docker-образов	Контейнеризация приложения	Публикация образа в registry
9	Управление артефактами	Публикация артефактов (wheel, bin)	Артефакты в package registry
10	Развертывание на тестовый стенд	Развертывание в Kubernetes (dev)	Работающее приложение в кластере
11	Стратегия Rolling Update	Реализация rolling update	Zero-downtime обновление
12	Стратегия Blue-Green Deployment	Реализация blue-green	Мгновенное переключение
13	Canary Deployment и Feature Flags	Реализация canary (10%	Поэтапное развертывание

		трафика)	
14	GitOps с Argo CD	Настройка GitOps	Автосинхронизация из Git
15	CI/CD для встроенной системы (кросс-компиляция)	Кросс-компиляция прошивки (опционально)	.bin/.hex артефакты
16	Интеграция симулятора в пайплайн (SITL)	Добавление SITL-тестов	Тесты в Gazebo/Webots
17	Безопасные OTA-обновления	Подпись OTA-пакета (Cosign)	Безопасные обновления
18	Защита сквозного проекта	Презентация и демонстрация	Итоговый пайплайн

4. ПОРЯДОК ВЫПОЛНЕНИЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

4.1. Организационные этапы

Этап	Содержание	Срок	Форма контроля
0	Выбор темы проекта и технологического стека	1-я неделя	Согласование с преподавателем
1	Настройка репозитория и базового пайплайна (ЛР 1-3)	2-4 недели	Проверка преподавателем
2	Добавление тестирования и анализа (ЛР 4-7)	5-8 недели	Проверка преподавателем
3	Контейнеризация и развертывание (ЛР 8-10)	9-11 недели	Проверка преподавателем
4	Стратегии развертывания (ЛР 11-14)	12-15	Проверка

		недели	преподавателем
5	Специализированные этапы (ЛР 15-17)	16-17 недели	Проверка преподавателем
6	Подготовка отчёта и презентации	18-я неделя	Защита проекта

4.2. Технологический стек (по выбору студента)

Студент может выбрать один из следующих вариантов стека:

Вариант	CI/CD	Оркестрация	Симуляция	Язык
A (рекомендуемый)	GitLab CI	Minikube	Gazebo + ROS 2	Python
B	GitHub Actions	Kind	Webots	C++
C	GitLab CI	K3s	Gazebo	Go

4.3. Требования к итоговому пайплайну

Итоговый пайплайн должен включать следующие этапы (обязательный минимум):

1. Checkout - получение кода из репозитория.
2. Build - сборка приложения / кросс-компиляция.
3. Lint & Format - проверка стиля кода (black, flake8, clang-format).
4. SAST - статический анализ безопасности (Semgrep/Bandit).
5. Unit Tests - модульные тесты с генерацией отчёта JUnit.
6. Coverage - проверка покрытия (порог 80%).
7. SCA - анализ зависимостей (OWASP DC / Trivy).
8. Docker Build - сборка Docker-образа.
9. Docker Push - публикация образа в registry.

10. SITL Tests - тесты в симуляторе (Gazebo/Webots).
11. Deploy (dev) - развертывание в dev-окружение.
12. Smoke Test - проверка работоспособности после деплоя.
13. Canary / Rolling Update - стратегия обновления.
14. OTA Sign - подпись OTA-пакета (Cosign).

4.4. Структура отчёта по проекту

Отчёт должен содержать:

1. Титульный лист (ФИО, группа, тема).
2. Введение (актуальность, цель, задачи).
3. Архитектура пайплайна (схема, описание этапов).
4. Конфигурация CI/CD (листинги с комментариями).
5. Результаты тестирования (скриншоты, отчёты, метрики).
6. Демонстрация работы (скриншоты или видео развертывания).
7. Заключение (достигнутые результаты, выводы).
8. Список литературы.

5. КРИТЕРИИ ОЦЕНИВАНИЯ СКВОЗНОГО ПРОЕКТА

Критерий	Вес	Отлично (5)	Хорошо (4)	Удовл. (3)	Неуд. (2)
Полнота этапов пайплайна	25 %	14+ этапов	11-13 этапов	8-10 этапов	<8 этапов
Качество конфигурации	20 %	Best practices, DRY	Есть замечания	Много дублирования	Нечитаема
Тестирование	15 %	>80%, отчёты	70-80%	60-70%	<60%

покрытие					
Развертывание и стратегии	15 %	3+ стратегии	2 стратегии	1 стратегия	Нет
Безопасность	10 %	SAST+SCA+подпись	SAST+SCA	SAST	Нет
Документация и отчёт	10 %	Полный, структурированный	Есть недочёты	Минимальный	Отсутствует
Защита (презентация)	5%	Уверенно, ответы	С небольшими ошибками	Слабо	Не готов

Шкала итоговой оценки:

Суммарный балл	Оценка
90-100%	5 (отлично)
75-89%	4 (хорошо)
60-74%	3 (удовлетворительно)
<60%	2 (неудовлетворительно)

6. СПИСОК ЛИТЕРАТУРЫ И ИСТОЧНИКОВ

6.1. Основная литература

1. Брайант Д. «Git. Исчерпывающее руководство». — ДМК-Пресс, 2022. (Есть в РФ)
2. Хиттерман Б. «CI/CD с GitLab. Практическое руководство». — Питер, 2021. (Есть в РФ)

6.2. Дополнительная литература

3. Фарли Д. «Непрерывная доставка программного обеспечения». — Вильямс, 2020.

4. Пул К. «Docker в действии». — ДМК-Пресс, 2021.

6.3. Онлайн-ресурсы

Ресурс	Описание	Ссылка
GitLab Documentation	Официальная документация GitLab CI	https://docs.gitlab.ru/
GitHub Docs (зеркало)	Документация GitHub Actions	https://docs.github.com/ru
Kubernetes Documentation	Документация K8s на русском	https://kubernetes.io/ru/docs/
Docker Documentation	Документация Docker	https://docs.docker.com/
ROS 2 Documentation	Документация ROS 2 (Humble)	https://docs.ros.org/en/humble/
Gazebo (Ignition) Tutorials	Тьюториалы Gazebo	https://gazebosim.org/docs
Semgrep Documentation	Документация Semgrep	https://semgrep.dev/docs
Cosign Documentation	Документация Cosign	https://docs.sigstore.dev/cosign/overview

Argo CD Documentation	Документация Argo CD	https://argo-cd.readthedocs.io/
OWASP Dependency-Check	Документация OWASP DC	https://jeremylong.github.io/DependencyCheck/

6.4. Рекомендуемые видеокурсы (доступны на Rutube)

- «DevOps для начинающих» — Слёрм (бесплатные уроки).
- «GitLab CI с нуля» — канал «DevOps Journey» (перевод).
- «Kubernetes для разработчиков» — OTUS (открытые вебинары).

7. ПРИМЕРНЫЙ ГРАФИК ВЫПОЛНЕНИЯ ПРОЕКТА

Неделя	Лабораторные работы	Этап проекта	Отчётность
1-2	ЛР 1-2	Выбор стека, настройка репозитория	Проверка репозитория
3-4	ЛР 3-4	Базовый пайплайн, тесты	Демонстрация
5-6	ЛР 5-6	Покрытие, SAST	Проверка отчётов
7-8	ЛР 7-8	SCA, Docker образ	Публикация образа
9-10	ЛР 9-10	Артефакты, деплой в K8s	Демонстрация
11-12	ЛР 11-12	Rolling update, Blue-Green	Проверка стратегий
13-14	ЛР 13-14	Canary, GitOps	Демонстрация
15-16	ЛР 15-16	Кросс-компиляция, SITL	Проверка
17	ЛР 17	ОТА, подпись	Проверка подписи
18	—	Защита проекта	Презентация

8. ВОПРОСЫ ДЛЯ ЗАЩИТЫ ПРОЕКТА

1. Опишите архитектуру вашего CI/CD-пайплайна. Сколько этапов?
Почему именно такая структура?
2. Как вы обеспечили воспроизводимость сборки?
3. Какие стратегии развертывания вы реализовали? В чем их отличие?
4. Как вы настроили проверку покрытия кода? Какое пороговое значение и почему?
5. Какие инструменты SAST/SCA вы использовали? Какие уязвимости нашли?
6. Как организовано безопасное хранение секретов в вашем пайплайне?
7. Как вы настроили SITL-тестирование? Какие сценарии автоматизированы?
8. Как выполняется подпись OTA-пакета? Как устройство проверяет подпись?
9. Какие метрики вы собираете для мониторинга успешности развертывания?
10. Что бы вы улучшили в текущей реализации при наличии большего времени?

9. ПРИЛОЖЕНИЯ

Приложение А. Шаблон репозитория для проекта

Рекомендуемая структура Git-репозитория:

```

student-project/
├── .gitlab-ci.yml                # Основной файл CI/CD
├── .gitlab/
│   └── templates/              # Шаблоны пайплайна
│       ├── build.yml
│       ├── test.yml
│       ├── deploy.yml
│       └── security.yml
├── src/                         # Исходный код приложения
│   ├── main.py
│   └── ...
├── tests/                      # Тесты
│   ├── unit/
│   ├── integration/
│   └── sitl/
├── docker/
│   └── Dockerfile
├── k8s/                        # Kubernetes манифесты
│   ├── deployment.yaml
│   ├── service.yaml
│   └── ingress.yaml
├── sim/                        # Конфигурация симуляции
│   ├── world.sdf
│   └── launch.py
├── docs/                      # Документация
│   └── report.md
└── README.md

```

Приложение Б. Чек-лист самопроверки перед защитой

- Пайплайн запускается без ошибок.
- Покрытие кода $\geq 80\%$.
- SAST-анализ не выдаёт HIGH/CRITICAL уязвимостей.
- SCA-анализ не выдаёт HIGH/CRITICAL уязвимостей.
- Docker-образ собран и опубликован.
- Приложение развёрнуто в Kubernetes (Minikube/Kind).
- Реализована минимум одна стратегия развёртывания (rolling/canary/blue-green).
- SITL-тест проходит успешно.
- ОТА-пакет подписан и верифицирован.
- Отчёт оформлен по требованиям.
- Презентация подготовлена.

