

# Code Quality Workshop

# Статический анализ кода и артефакты в CI

Настройка автоматических проверок качества

Практический воркшоп

# Пару слов о себе






## Максим Гусев

Lead SRE Observability Team в Dodo Engineering

11 лет в IT, последние 5 лет - SRE инженер

Читаю лекции о SRE и DevOps, автор нескольких курсов и воркшопов по DevOps

## Что мы изучим

-  Интеграция линтеров в CI-процесс
-  Автоматическое форматирование кода
-  Проверка типов и безопасности
-  Работа с артефактами
-  Визуализация и отчеты

# Проблема

## Без статического анализа:

Developer → Write Code → Commit → Review ❌



Стиль разный  
Баги не найдены  
Типы не проверены  
Уязвимости пропущены

## Со статическим анализом:

Developer → Write → Auto-check ✅ → Auto-format ✅ → Commit → CI ✅



Линтеры



Форматтеры



Отчеты

# Что такое статический анализ?

Статический анализ = проверка кода без его выполнения

Находит:

- Потенциальные баги
- Проблемы со стилем кода
- Уязвимости безопасности
- Сложный код
- Неоптимальные конструкции
- Отсутствие документации

## Преимущества:

- Быстро (секунды, но не всегда)
- Не требует запуска
- Находит проблемы до production

# Категории инструментов

## Статический анализ

1. Линтеры
  - └ Проверка стиля и ошибок
2. Форматтеры
  - └ Автоматическое оформление
3. Type Checkers
  - └ Проверка типов
4. Security Scanners
  - └ Поиск уязвимостей



# Python - Линтеры

## Flake8

```
flake8 app/  
# E501: line too long  
# F401: imported but unused  
# W503: line break before operator
```

### Что проверяет:

- PEP 8 compliance
- Синтаксические ошибки
- Неиспользуемые импорты
- Длина строк

# Python - Линтеры

## Pylint

```
pylint app/  
# C0103: Invalid name  
# R0913: Too many arguments  
# W0612: Unused variable
```

### Что проверяет:

- Качество кода
- Соглашения об именовании
- Сложность кода
- Потенциальные ошибки
- Рефакторинг opportunities

# Python - Форматтеры

## Black - "The Uncompromising Code Formatter"

```
# До
def very_long_function_name(param1,param2,param3,param4):
    x=param1+param2
    return x

# После Black
def very_long_function_name(
    param1, param2, param3, param4
):
    x = param1 + param2
    return x
```

**Философия:** Нет дискуссий о стиле - Black решает за вас

# Python - Форматтеры

## isort - Сортировка импортов

```
# До
import os
from flask import Flask
import sys
from app.models import User
import json

# После isort
import json
import os
import sys

from flask import Flask
from app.models import User
```

**Группы:** FUTURE → STDLIB → THIRDPARTY → FIRSTPARTY → LOCALFOLDER

# Python - Type Checking

## муру - Статическая проверка типов

```
# Код с type hints
def add_user(name: str, age: int) -> User:
    return User(name, age)
# муру найдет ошибку:
add_user("Alice", "25") # ✗ Expected int, got str
# муру одобрит:
add_user("Alice", 25)   # ✓ Correct types
```

### Преимущества:

- Ловит ошибки до runtime
- Улучшает автодополнение IDE
- Документация в коде

# Python - Security

## Bandit - Поиск уязвимостей

# Bandit найдет проблемы:

```
import pickle # B403: pickle unsafe
password = "hardcoded123" # B105: hardcoded password
os.system(user_input) # B605: shell injection risk
requests.get(url, verify=False) # B501: SSL verification disabled
```

### Проверяет:

- SQL injection
- Command injection
- Hardcoded secrets
- Crypto issues

# JavaScript - Линтеры

## ESLint

// ESLint найдет проблемы:

```
var x = 1;  // ✗ Unexpected var, use const/let
function foo(){  // ✗ Missing space before {
  console.log(x)  // ✗ Missing semicolon
}
if(x=1){  // ✗ Assignment in condition
  return true
}
```

Правила: 250+ встроенных правил

# JavaScript - Форматтеры

## Prettier - Opinionated Code Formatter

```
// До
const user={name:"Alice",age:25,email:"alice@example.com"};
function greet(name){console.log("Hello "+name)}

// После Prettier
const user = {
  name: "Alice",
  age: 25,
  email: "alice@example.com",
};

function greet(name) {
  console.log("Hello " + name);
}
```

**Интеграция:** Работает с ESLint



# Конфигурационные файлы

## Python:

```
.pylintrc          # Pylint config  
.flake8            # Flake8 config  
mypy.ini           # mypy config  
pyproject.toml     # Black + isort + pytest
```

## JavaScript:

```
.eslintrc.json     # ESLint config  
.prettierrc.json   # Prettier config  
package.json       # Scripts
```

**Цель:** Единообразие в команде

# Интеграция в CI - Workflow

```
name: Code Quality

on: [push, pull_request]

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

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

      - name: Lint with Flake8
        run: |
          pip install flake8
          flake8 app/ tests/
```

# Артефакты - Что это?

Артефакт = файл или набор файлов, сохраненный после workflow

## Примеры артефактов:

- HTML отчеты (coverage, lint)
- JSON данные для интеграций
- Собранные приложения
- Результаты тестов
- Скриншоты
- Метрики производительности

**Хранение:** 90 дней по умолчанию (настраивается)

## Создание артефактов

- `name: Generate coverage report`  
`run: |`  
`pytest --cov=app --cov-report=html`
- `name: Upload coverage`  
`uses: actions/upload-artifact@v4`  
`with:`  
`name: coverage-report`  
`path: htmlcov/`  
`retention-days: 30`

**Результат:** Отчет доступен для скачивания в GitHub UI

# Скачивание артефактов

```
- name: Download artifact
  uses: actions/download-artifact@v4
  with:
    name: coverage-report
    path: ./reports
```

## Использование:

- Передача между jobs
- Анализ результатов
- Deployment артефактов

# Типы отчетов

## Coverage Report (HTML)

```
htmlcov/  
├── index.html          # Главная страница  
├── app_service.html    # Покрытие модуля  
└── style.css
```

## Lint Report (JSON)

```
{  
  "file": "app/service.py",  
  "line": 42,  
  "message": "Line too long",  
  "severity": "warning"  
}
```





# Типы отчетов

## Test Report (JUnit XML)





```
<testsuite tests="15" failures="0" errors="0">  
  <testcase name="test_user_creation"/>  
</testsuite>
```

# Визуализация отчетов

## HTML Reports:

-  Красивый UI
-  Интерактивность
-  Графики и таблицы
-  Drill-down по файлам

## JSON Reports:

-  Машиночитаемые
-  Интеграция с API
-  Автоматическая обработка
-  Trending analysis



## Комментарии в PR

```
- name: Comment PR
  uses: actions/github-script@v7
  with:
    script: |
      const coverage = fs.readFileSync('coverage.txt', 'utf8');
      github.rest.issues.createComment({
        issue_number: context.issue.number,
        owner: context.repo.owner,
        repo: context.repo.repo,
        body: `## Coverage Report\n\n${coverage}`
      });
```

**Результат:** Автоматический комментарий с метриками в PR

# Quality Gate

```
quality-gate:
  needs: [lint, test, security]
  runs-on: ubuntu-latest
  steps:
    - name: Check results
      run: |
        echo "✅ All checks passed!"
        echo "Lint: ✅"
        echo "Tests: ✅"
        echo "Security: ✅"
```

**Quality Gate** = точка принятия решения о качестве

# Параллельное выполнение

```
jobs:
  python-lint:      # Выполняется параллельно
    ...

  python-format:    # Выполняется параллельно
    ...

  js-lint:           # Выполняется параллельно
    ...

  quality-gate:      # Ждет все предыдущие
    needs: [python-lint, python-format, js-lint]
    ...
```

**Преимущество:** Быстрее в 3-5 раз

# Best Practices

## DO:

- Используйте все инструменты в комбинации
- Автоматизируйте форматирование
- Сохраняйте отчеты как артефакты
- Установите пороги качества (coverage > 80%)
- Настройте pre-commit hooks
- Комментируйте PR с результатами
- Делайте отчеты красивыми (HTML)

# Best Practices

## ✗ DON'T:

- Не игнорируйте предупреждения
- Не коммитьте неотформатированный код
- Не отключайте проверки без причины
- Не сохраняйте артефакты вечно
- Не делайте проверки слишком строгими сразу
- Не забывайте про JavaScript

# Метрики качества

Что измерять:

Метрика	Хорошо	Отлично
Coverage	> 70%	> 90%
Pylint Score	> 7.0	> 9.0
Complexity	< 15	< 10
Security Issues	0 high	0 any
Type Hints	> 50%	100%

# Roadmap внедрения

## Неделя 1-2: Базовая настройка

- Установить линтеры локально
- Настроить конфигурации
- Интегрировать в IDE

## Неделя 3-4: CI/CD

- Добавить в GitHub Actions
- Настроить артефакты
- Создать отчеты

# Roadmap внедрения

## Неделя 5-6: Команда

- Обучить команду
- Настроить pre-commit hooks
- Установить quality gates

## Неделя 7-8: Оптимизация

- Анализ метрик
- Тюнинг правил
- Автоматизация



# Pre-commit hooks

```
# .pre-commit-config.yaml
repos:
  - repo: https://github.com/psf/black
    rev: 24.1.1
    hooks:
      - id: black

  - repo: https://github.com/PyCQA/flake8
    rev: 7.0.0
    hooks:
      - id: flake8
```

# Pre-commit hooks

**Установка:**

```
pip install pre-commit  
pre-commit install
```

**Результат:** Проверки ДО коммита

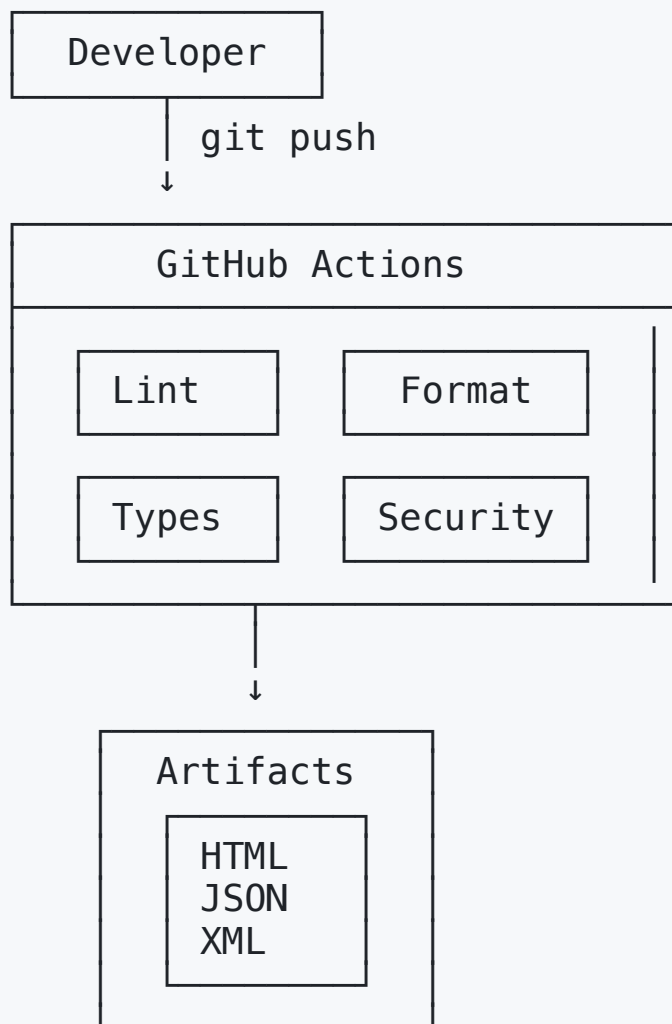
# Интеграции

## GitHub Actions + External Services:

- **SonarQube** - качество кода
- **CodeClimate** - метрики и тренды
- **Codecov** - визуализация coverage
- **Snyk** - безопасность зависимостей
- **Dependabot** - обновление зависимостей

**API:** Отправка JSON отчетов

# Архитектура решения



## Пример полного pipeline

```
name: Complete Quality Pipeline

jobs:
  lint:          # 1. Проверка стиля
  format:        # 2. Проверка форматирования
  types:         # 3. Проверка типов
  security:      # 4. Проверка безопасности
  test:          # 5. Тесты
  coverage:      # 6. Покрытие
  artifacts:     # 7. Сохранение отчетов
  quality-gate:  # 8. Финальная проверка
```

**Время выполнения:** ~5-10 минут (параллельно)

# Демонстрация отчетов

## Coverage Report:

- Цветная подсветка кода
- Покрытие по файлам
- Непокрытые строки
- Статистика по модулям

## Flake8 Report:

- Список всех проблем
- Группировка по файлам
- Severity levels
- Быстрая навигация

# Демонстрация отчетов

## Bandit Report:

- Уровни риска (High/Medium/Low)
- Описание уязвимостей
- Рекомендации по исправлению

# ROI статического анализа

## Экономия времени:

- 🕒 Review быстрее на 30-50%
- 🐛 Меньше багов в production
- 🔄 Меньше итераций review

## Улучшение качества:

- 📈 Coverage растёт
- 🎯 Единый стиль кода
- 🛡️ Меньше уязвимостей

## Экономия денег:

- 💰 1 час настройки = 10 часов сэкономленного review



## **Проблемы и решения**

**Проблема: "Слишком много предупреждений"**

**Решение:** Начинайте постепенно, отключите строгие правила

**Проблема: "CI слишком долго"**

**Решение:** Кэширование + параллелизация

**Проблема: "Команда сопротивляется"**

**Решение:** Покажите выгоды, автоматизируйте форматирование

**Проблема: "Конфликты между инструментами"**

**Решение:** Используйте совместимые конфигурации (Black + Flake8)

# Практическое задание

1. Форкните проект
2. Добавьте новый endpoint в API
3. Напишите тесты
4. Создайте PR
5. Посмотрите на проверки:
  - Линтеры должны пройти
  - Форматирование должно быть правильным
  - Coverage не должен упасть
6. Скачайте артефакты с отчетами

# Ресурсы для изучения

## Документация:

- [Flake8](#)
- [Pylint](#)
- [Black](#)
- [mypy](#)
- [Bandit](#)
- [ESLint](#)
- [Prettier](#)

## Книги:

- "Clean Code" - Robert Martin
- "The Art of Readable Code" - Boswell & Foucher

# GitHub Actions - Artifacts API

## Upload:

```
uses: actions/upload-artifact@v4
with:
  name: my-artifact
  path: reports/
  retention-days: 30
  if-no-files-found: error
```

## Download:

```
uses: actions/download-artifact@v4
with:
  name: my-artifact
  path: ./downloaded
```

## Advanced - Matrix + Artifacts

```
strategy:
  matrix:
    python: ['3.9', '3.10', '3.11']

steps:
  - name: Test
    run: pytest --html=report-${{ matrix.python }}.html

  - name: Upload
    uses: actions/upload-artifact@v4
    with:
      name: report-python-${{ matrix.python }}
      path: report-*.html
```

**Результат:** Отдельный отчет для каждой версии

# Continuous Quality



**Цель:** Постоянное улучшение качества

# Итоги

## Мы изучили:

- ✓ Линтеры (Flake8, Pylint, ESLint)
- ✓ Форматтеры (Black, isort, Prettier)
- ✓ Type Checking (mypy)
- ✓ Security (Bandit)
- ✓ Артефакты в GitHub Actions
- ✓ Генерацию отчетов
- ✓ Визуализацию результатов
- ✓ Quality Gates

## Теперь вы можете:

- 🚀 Настроить полный Code Quality pipeline
- 🚀 Генерировать и сохранять отчеты

# Спасибо за внимание! 🎉

Links:



Telegram:

@fadeinflames

Материалы воркшопа:

GitHub:

<https://github.com/fadeinflames/github-codequality-workshop>



## Бонус: Чек-лист внедрения

- ☐ Установить линтеры локально
- ☐ Настроить конфигурационные файлы
- ☐ Добавить в IDE
- ☐ Создать GitHub Actions workflows
- ☐ Настроить артефакты
- ☐ Генерировать HTML отчеты
- ☐ Добавить комментарии в PR
- ☐ Установить quality gates
- ☐ Настроить pre-commit hooks
- ☐ Обучить команду
- ☐ Сбирать метрики
- ☐ Регулярно review правил

# Бонус: Quick Start Commands

```
# Python
pip install flake8 pylint black isort mypy bandit
flake8 app/
pylint app/
black app/
isort app/
mypy app/
bandit -r app/
```

```
# JavaScript
npm install -D eslint prettier
npm run lint
npm run format
```

```
# Отчеты
pytest --cov=app --cov-report=html
flake8 app/ --format=html --htmldir=reports/
```

## Бонус: .pre-commit-config.yaml

```
repos:
  - repo: https://github.com/psf/black
    hooks:
      - id: black

  - repo: https://github.com/pycqa/isort
    hooks:
      - id: isort

  - repo: https://github.com/pycqa/flake8
    hooks:
      - id: flake8

  - repo: https://github.com/pre-commit/mirrors-prettier
    hooks:
      - id: prettier
        types_or: [javascript, jsx, ts, tsx]
```