# Разработка инфраструктуры программного обеспечения

## Разработка CI/CD (Continious integration / Continious deployment)

Лаборатория RISC-V технологий,
2025 г.

# Проблемы, которые решает CI/CD

- Представим себе некоторую команду из двух человек, работающую над проектом
- Проект написан на C++, в нем написана система сборки и некоторые тесты

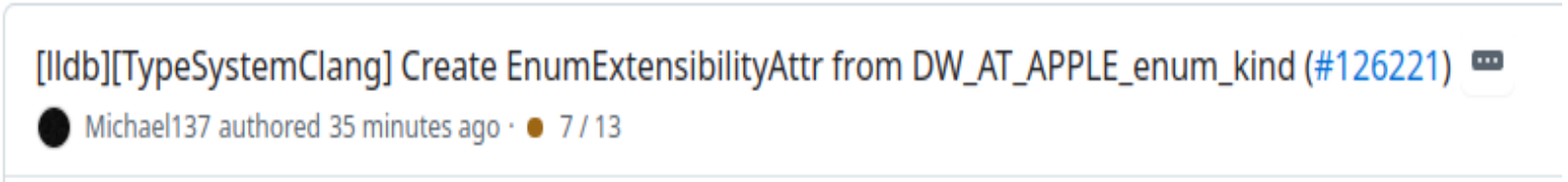
# Проблемы, которые решает CI/CD

- Представим себе некоторую команду из двух человек, работающую над проектом
- Проект написан на C++, в нем написана система сборки и некоторые тесты

Также познакомимся с двумя разработчиками, которые будут помогать нам иллюстрировать проблемы в нашем курсе, а также учиться их решать — Кирилл и Миша

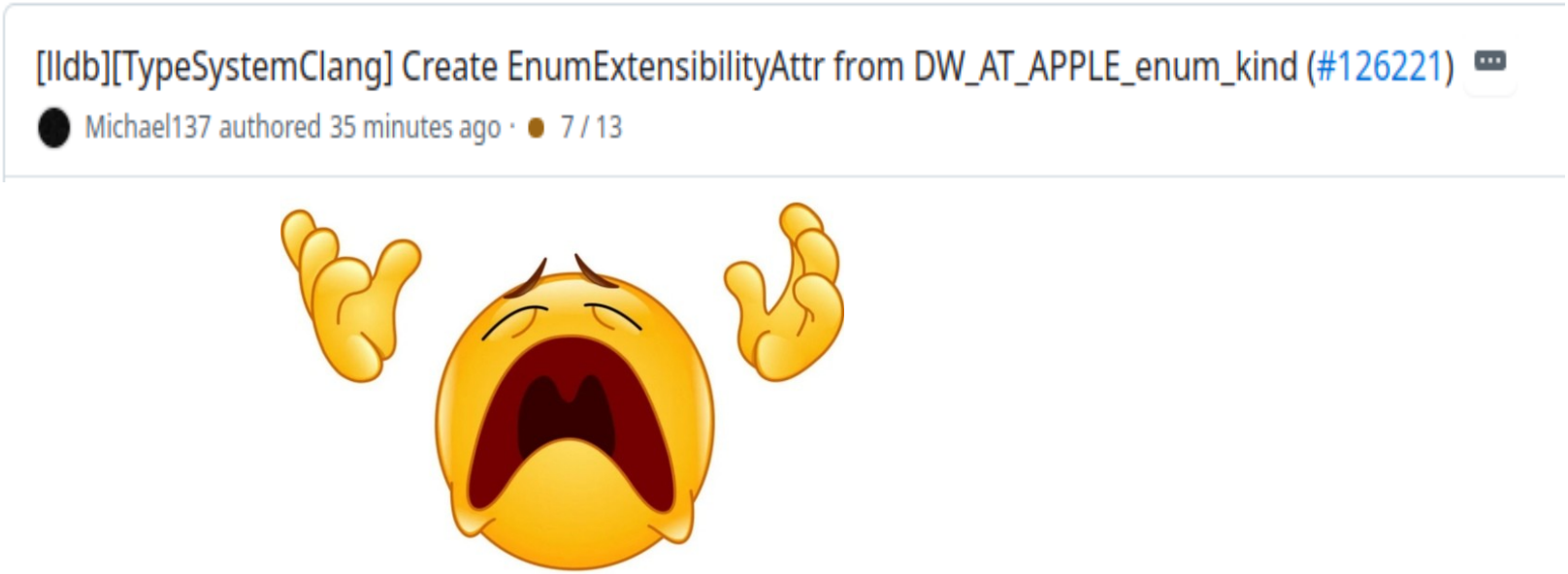
# Проблемы, которые решает CI/CD

- Представим, что в начальный момент времени у каждого разработчика **локально** проект собирался
- После этого происходит коммит Миши, который он запушил в основную ветку для разработки



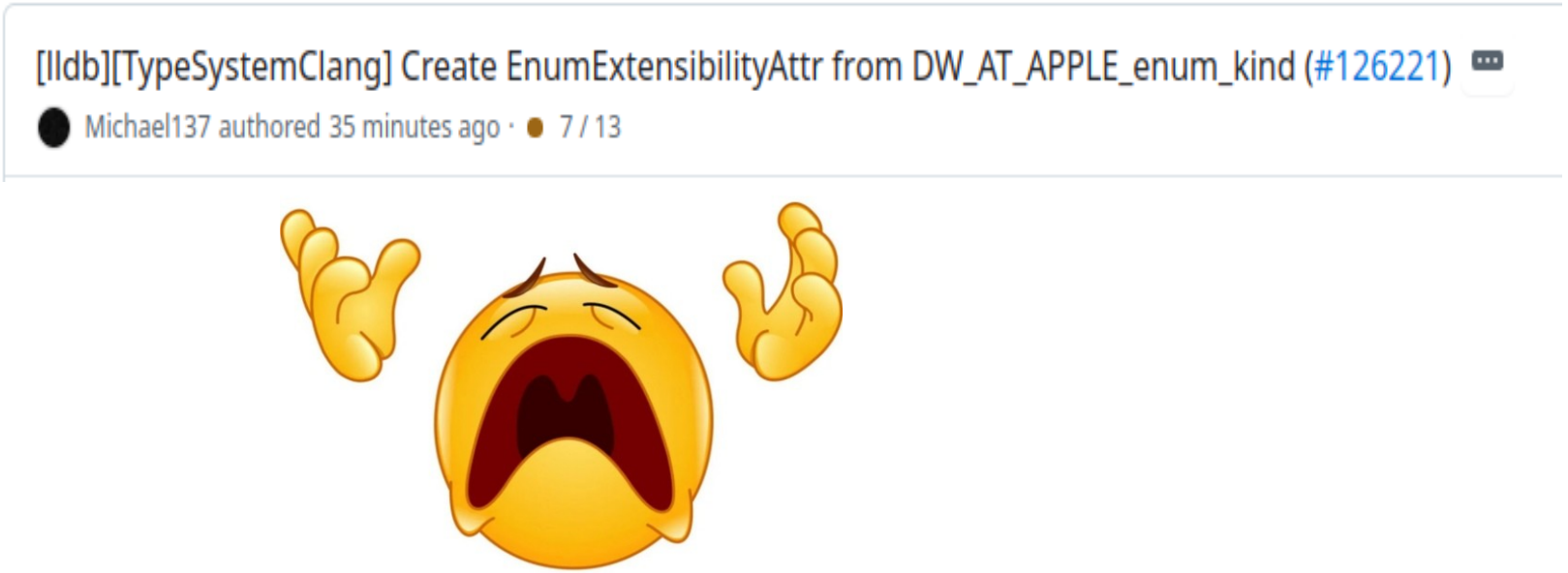
# Проблемы, которые решает CI/CD

- Представим, что до того, как произошли эти три коммита, у каждого разработчика **локально** проект собирался
- После этого происходит коммит Миши, который он запушил в основную ветку для разработки
- Однако Миша забыл прогнать тестирование, и его коммит оказался нерабочим
- **Первая проблема:** текущая рабочая ветка **сломана**
- Когда на данной ветке разработчик всего один — ситуация не такая страшная.



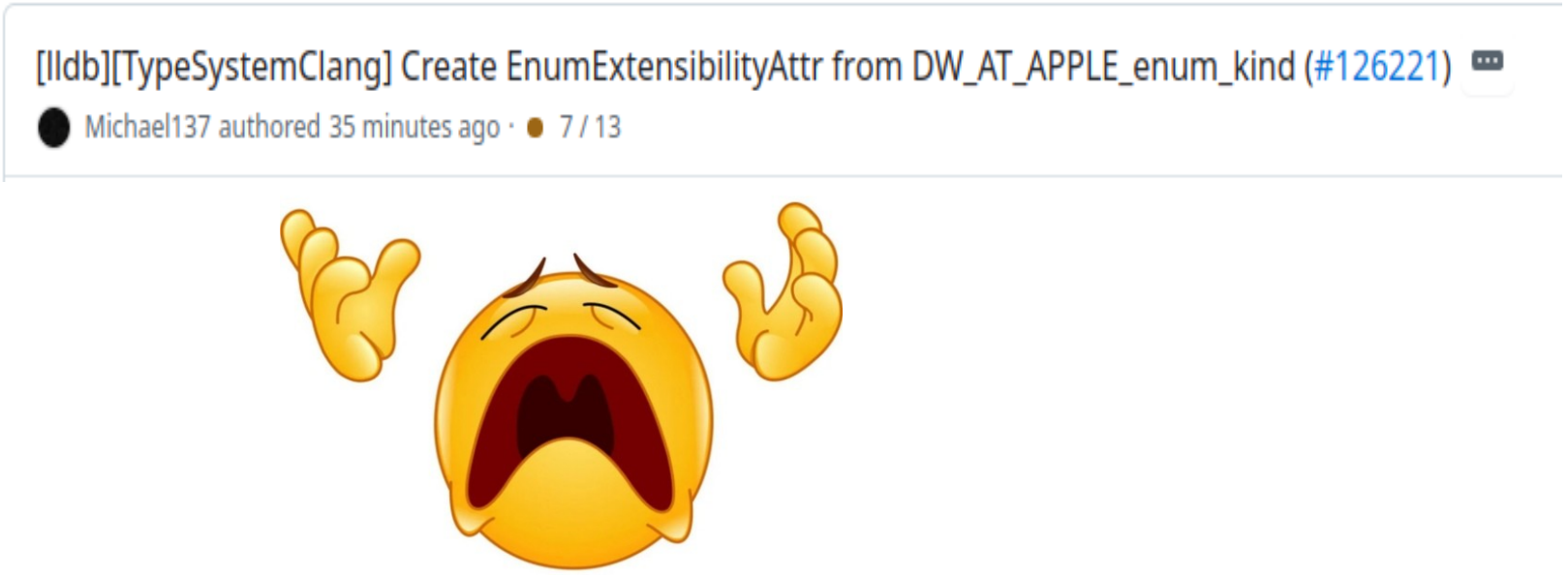
# Проблемы, которые решает CI/CD

- В идеале конечно же перед обновлением удаленный ветки репозитория (перед **git push**), разработчику стоит локально протестировать свои изменения
- Крупные проблемы начинаются, когда на одной ветке работают два или более разработчиков



# Проблемы, которые решает CI/CD

- В идеале конечно же перед обновлением удаленный ветки репозитория (перед **git push**), разработчику стоит локально протестировать свои изменения
- Крупные проблемы начинаются, когда на одной ветке работают два или более разработчиков
- У Кирилла, который только проснулся и решил внести свой вклад в проект — **сломана** сборка / тестирование на главной ветке



# Проблемы, которые решает CI/CD

- Вариант решения: всегда прогонять сборку и тестирование компонента перед созданием коммита в общую ветку

# Проблемы, которые решает CI/CD

- Вариант решения: всегда прогонять сборку и тестирование компонента перед созданием коммита в общую ветку
- **Плохо: в**ысокое влияние человеческого фактора
- А может прогонять сборку и тестирование проекта в автоматизированном формате?
- А только ли сборку и тестирование? Может, стоит запускать что-то еще, что характеризует **состояние** проекта?

# Проблемы, которые решает CI/CD

- Форматирование кода — clang-format (C++), black (Python)
- Качество кода — clang-tidy (C++), pylint (Python)
- Сборка проекта — cmake (C++), wheels (Python)
- Тесты в проекте, **pass rate**
- Инструментарий для пакетирования проекта
- **Внешние зависимости (???)**

- Все эти и еще многие другие свойства определяют **состояние проекта**
- CI/CD и инструменты, направленные на его разработку, решают задачу достижения **максимальной стабильности состояние проекта во времени**

# Проблемы, которые решает CI/CD

- Также CI/CD направлен на то, чтобы подерживать рабочее состояние проекта автоматизированными методами
- Итак, попробуем решить первую задачу автоматизации в нашем проекте — осуществлять некоторый набор действий,/home/stanislav/Downloads/images.png которые проверяет целостность состояния проекта, в автоматике.
- Для этого нам понадобится **CI** сервис
- Самые популярные — **GitHub actions, GitLab CI, Jenkins**



# GitHub actions

- **GitHub actions** —  **CI** сервис на **github**
- Позволяет автоматизировать набор действий в проекте, запускать сборки как на общедоступных серверах — так и на selfhosted
- Бесплатный для коммьюнити пользования
- Предоставляет бесплатные общедоступные серверы, на которых будут прогоняться ваши сборки
- Легко настраиватся

# GitHub actions

- **GitHub actions** —  **CI** сервис на **github**
- Позволяет автоматизировать набор действий в проекте, запускать сборки как на общедоступных серверах — так и на selfhosted
- Бесплатный для коммьюнити пользования
- Предоставляет бесплатные общедоступные серверы, на которых будут прогоняться ваши сборки
- Легко настраиватся

- Точка входа  для создания **Continious integration**  в вашем проекте — описание воркфлоу проекта в yml формате, находящемся в проекте в директории **.github/workflows/workflow.yml**

# Пишем свой первый workflow.yaml

- **name —** название воркфлоу
  **on —** правила, описывающие на чем и когда запускается данной воркфолу
- **push, pull_request:** события
- **branches**: регулярка для бранчей

Пример workflow.yaml

```yaml
name: Dummy workflow

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

jobs:
  main:
    runs-on: ubuntu-22.04
    steps:
    - name: Dummy echo
      echo "In pipeline, ready to roll."
```

# Пишем свой первый workflow.yaml

- **name —** название воркфлоу
  **on —** правила, описывающие на чем и когда запускается данной воркфолу
- **push, pull_request:** события
- **branches**: регулярка для бранчей

- Workflow включает в себя последовательность джоб (jobs) в терминах GitHub actions
- Каждая джоба происходит в новом окружении (новая директория, независима от предыдущей джобы)
- Часто для удобства хочется сделать отдельные джобы для отдельных стадий вашего проекта — например, конфигурацию, сборку и тестирование отделить в разные джобы

Пример workflow.yaml

```yaml
name: Dummy workflow

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

jobs:
  main:
    runs-on: ubuntu-22.04
    steps:
    - name: Dummy echo
      echo "In pipeline, ready to roll."
```

# Пишем свой первый workflow.yaml

- Для каждой джобы нужно указать окружение, в которой она будет запускаться (контейнер либо просто какая-то ОС)
- **Всегда указываете точную версию окружения**
- **НИКАКИХ LATEST**

Пример workflow.yaml

```yaml
name: Dummy workflow

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

jobs:
  main:
    runs-on: ubuntu-22.04
    steps:
    - name: Dummy echo
      echo "In pipeline, ready to roll."
```

# Пишем свой первый workflow.yaml

- Зачем мне писать несколько джоб, когда я могу написать весь код для воркфлоу в одной джобе?

Пример workflow.yaml

```yaml
name: Dummy workflow

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

jobs:
  main:
    runs-on: ubuntu-22.04
    steps:
    - name: Dummy echo
      Run:
        cmake -S llvm -B release/build -G Ninja
        cmake --build release/build
        cmake --install release/build
        ctest
```

# Пишем свой первый workflow.yaml

- Зачем мне писать несколько джоб, когда я могу написать весь код для воркфлоу в одной джобе?
- На самом деле, в целом это нормальная практика. Идея в том, чтобы прогонять разные стадии проекта в разных джобах нужна для возможности их отдельного перезапуска

Пример workflow.yaml

```yaml
name: Dummy workflow

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

jobs:
  main:
    runs-on: ubuntu-22.04
    steps:
    - name: Dummy echo
      Run:
        cmake -S llvm -B release/build -G Ninja
        cmake --build release/build
        cmake --install release/build
        ctest
```

# Пишем свой первый workflow.yaml

- Зачем мне писать несколько джоб, когда я могу написать весь код для воркфлоу в одной джобе?
- На самом деле, в целом это нормальная практика. Идея в том, чтобы прогонять разные стадии проекта в разных джобах нужна для возможности их отдельного перезапуска
- При вносе нового коммита с фиксом никто не гарантирует вам, что стадии которые до этого были успешны, не провалятся
- Так когда же нужно писать несколько джоб?

Пример workflow.yaml

```yaml
name: Dummy workflow

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

jobs:
  main:
    runs-on: ubuntu-22.04
    steps:
    - name: Dummy echo
      Run:
        cmake -S llvm -B release/build -G Ninja
        cmake --build release/build
        cmake --install release/build
        ctest
```

# Пишем свой первый workflow.yaml

- Зачем мне писать несколько джоб, когда я могу написать весь код для воркфлоу в одной джобе?
- На самом деле, в целом это нормальная практика. Идея в том, чтобы прогонять разные стадии проекта в разных джобах нужна для возможности их отдельного перезапуска
- При вносе нового коммита с фиксом никто не гарантирует вам, что стадии которые до этого были успешны, не провалятся
- Так когда же нужно писать несколько джоб?

- Когда функционал некоторой джобы запускается не на каждый коммит и под другими правилами
- Когда функционал джобы довольно сложный
- Когда функционал параллелится
- Когда некоторый функционал является переиспользуемым для других джоб

## Пример workflow.yaml

```yaml
name: Dummy workflow

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

jobs:
  main:
    runs-on: ubuntu-22.04
    steps:
    - name: Dummy echo
      run: |
        cmake -S llvm -B release/build -G Ninja
        cmake --build release/build
        cmake --install release/build
        Ctest

  fpga_test:
    runs-on: ubuntu-22.04
    steps:
      run: |
        ...
```

# Пишем свой первый workflow.yaml

- По аналогии с C/C++ джобы - это **функции**, со своей стоимостью вызова (развертка окружения, контейнера, **передача артефактов**).

Пример workflow.yaml

```yaml
name: Dummy workflow

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

jobs:
  main:
    runs-on: ubuntu-22.04
    steps:
    - name: Dummy echo
      run: |
        cmake -S llvm -B release/build -G Ninja
        cmake --build release/build
        cmake --install release/build
        Ctest

  fpga_test:
    runs-on: ubuntu-22.04
    steps:
      run: |
        ...
```

# Нюансы синтаксиса

- Оформление **steps** через пайп (|) позволяет писать шелловские команды друг за другом подряд

```
run: |
  echo 1
  echo 2
  echo 3
```

- Альтернативно можно каждую команду указыать с тире

```
run:
  - echo 1
  - echo 2
  - echo 3
```

# Переменные окружения

```
run: |
  echo 1
  echo 2
  echo 3
```

- Альтернативно можно каждую команду указыать с тире

```
run:
  - echo 1
  - echo 2
  - echo 3
```