

# Автоматизация Part 1

## CI/CD/CD/CT

Борисенко Глеб

ФТиАД2021

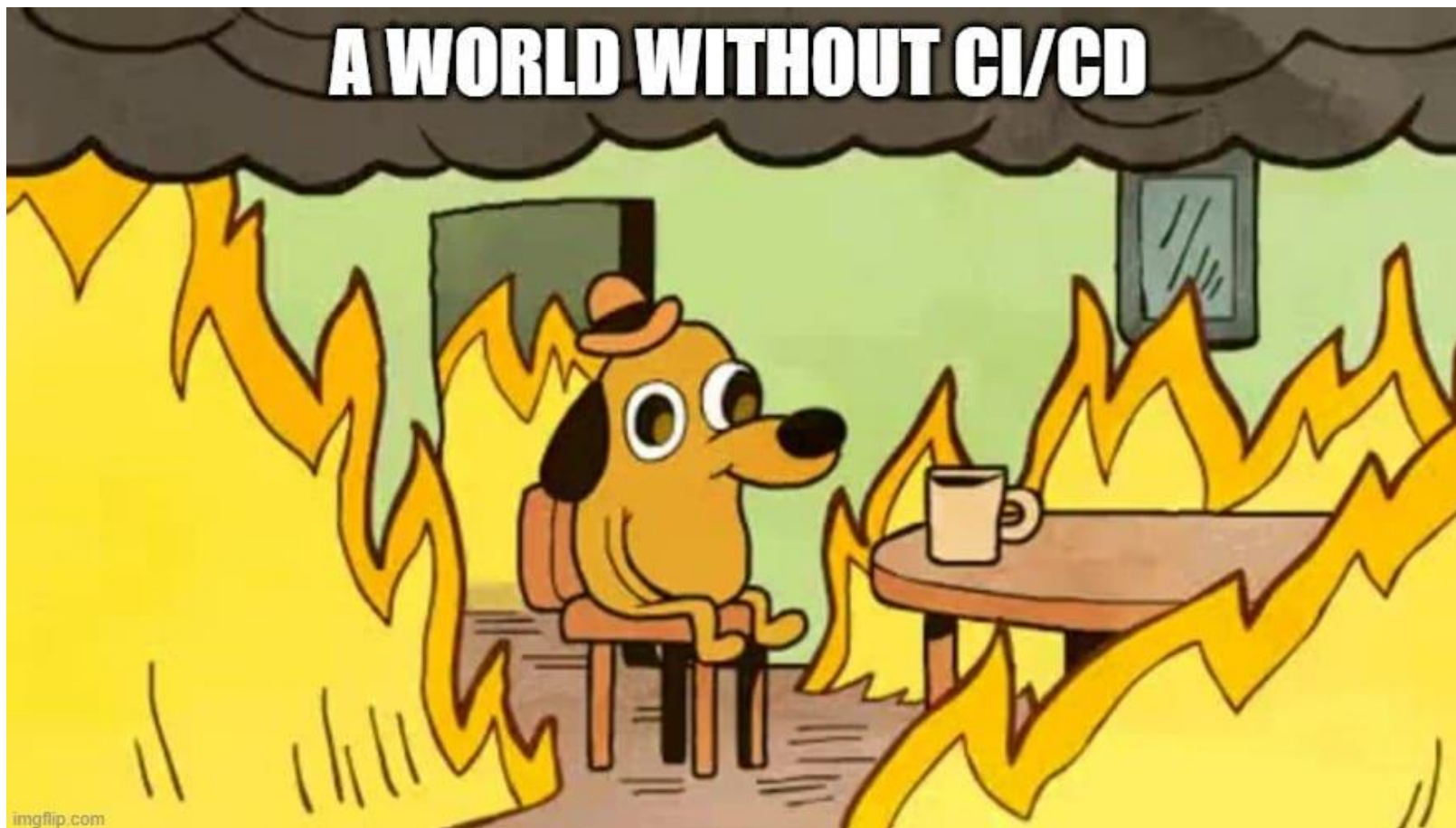
# В прошлый раз что было

- Посмотрели подробнее про разработку
- Посмотрели принципы ООП
- Расшифровали буквы SOLID
- Глянули паттерны GRASP
- Услышали общие базовые принципы проектирования кода

# Про что поговорим

- Зачем это нужно
- Расшифровка аббревиатур из названия
- Примеры джоб стандартные
- Инструменты обзорно:
  - Gitlab CI
  - ~~Leereöööu~~ Jenkins (одним слайдом)

Зачем это нужно (коротко, мемом)



# Зачем это нужно (подробнее)

Автоматизация хорошая помогает:

- Не думать об управлении/движении ваших моделек и кода
- Контролировать все эти движения
- Гибко, прозрачно, быстро доводить ваш код до прода
- Уменьшить количество ошибок (обычных и глупых)
- Сразу работать с красивыми, удобными, преобразованными данными (про это уже в следующей лекции поговорим преимущественно)
- Убрать зависимость от других людей по максимуму

# Какой должна быть автоматизация

- Гибкой
- Прозрачной
- Контролируемой
- Нужной
- Автоматической 😊
- Надежной
- Воспроизводимой, если мы говорим про МЛ

# Какие вообще ваши процессы/движения?

- Написали код -> Пройти через тесты/анализаторы -> Собрать пакет или контейнер -> Задеплоить
- Обновили модель -> Подложить в код
- Обучить/Переобучить
- Сделать инференс и посчитать метрики
- Модель должна дообучаться/переобучаться регулярно
- A/B тестирование
- Накидывайте еще :3

# А теперь к буковкам

- CI – Continuous Integration: сборка и тесты
- CD – Continuous Delivery: раскатка на тестовую/прод среду по кнопке
- CD – Continuous Deployment: то же самое, но автоматически; иногда под Deployment имеют в виду Delivery, но это нехорошо
- CT – Continuous Training: «бесшовное» дообучение
- Можно выделить еще пару букв



# Отличия CI/CD классической разработки от ML

- В классической разработке у нас есть по сути только код, которым мы и управляем
- В ML же у нас есть несколько «размерностей»:
  - Код
  - Данные
  - Модель
- Из-за этих размерностей и возникают свои особенности:
  - Данные нужно валидировать (схема, дрифт)
  - Валидировать модели (метрики считать, должна быть воспроизводимость)
  - Дополнительные процессы с пере-/дообучением (СТ)

# Наш CI

- Написали код, делаем merge request
- Код проходит тестирование
- Код собирается в пакет или контейнер
- Пакет или контейнер пушится в хранилище

## Особенности:

- Валидация данных и моделей нередко происходит не здесь, а в пайплайнах
- Но можно (а возможно, и нужно) как часть теста сделать и здесь
- Если модель обучается не в пайплайне, то пушится и модель

# Наш CD

- Проверяем/разворачиваем/готовим инфраструктуру
- Деплоим наш код (пайплайны и сервисы)

# Наш СТ

- Запустить код обучения модели
- Отвалидировать данные и обучить модель в пайплайне
- Проверить ее, положить в хранилище
- При необходимости – заменить ею уже использующуюся в другом коде

## Особенности

- Это делать уже лучше не в инструменте для CI/CD (кроме последнего пункта)

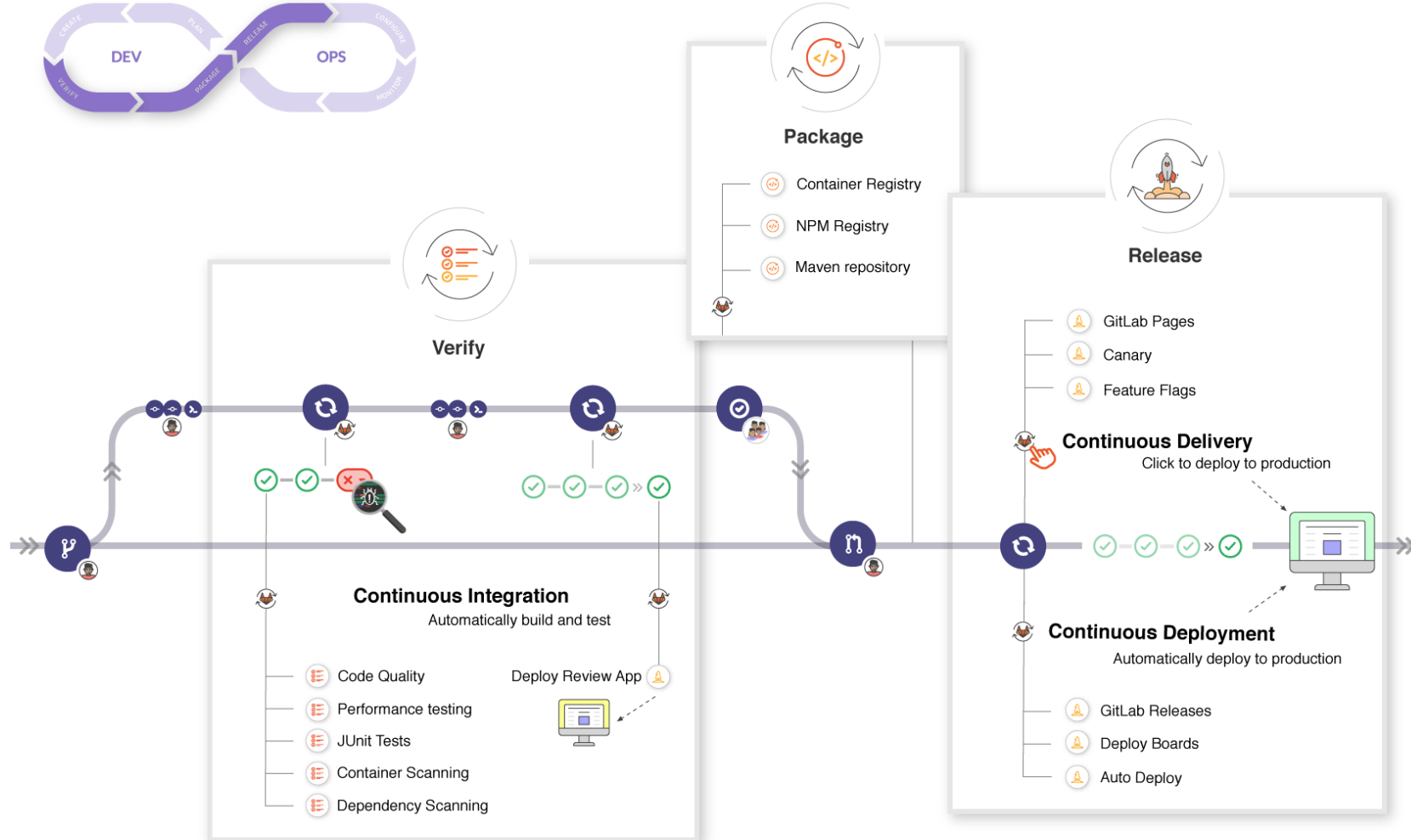
# GitOPS

- GitOPS – это когда Git единственный источник правды и происходит непрерывный процесс синхронизации из него.
- Весьма интересный подход, которые помогают осуществить некоторые технологии, но в любом случае все крутится вокруг гита
- Подробнее поговорим через лекцию

# А теперь посмотрим инструменты

- Что мы посмотрим:
  - Gitlab CI
  - Jenkins (одним слайдом)
- Что еще вообще есть:
  - Github Actions
  - Travis CI
  - Circle CI
  - Argo CD
- У каждого свои плюсы и минусы, при выборе стоит просто загуглить “<one\_tech> vs <another\_tech>”, и узнаете разницу

# Gitlab



# Это база

- Файл `.gitlab-ci.yml` в корне репозитория отвечает за ваши пайплайны
- При запуске пайплайна создается и выполняется граф
- Каждый «шажочек» пайплайна – это джоба
- Каждая джоба выполняется в контейнере на сервере с Gitlab runner
- Есть разные способы построения пайплайнов:  
[https://docs.gitlab.com/ee/ci/pipelines/pipeline\\_architectures.html](https://docs.gitlab.com/ee/ci/pipelines/pipeline_architectures.html)



# Gitlab Runners

- Это что-то вроде серверов, на которых выполняется джоба
- Их можно настроить по-разному
- Например, чтобы джобы выполнялись локально, или в кластере Kubernetes

# Как вообще выглядит ямлик

```
1  image: python:latest
2
3  pages:
4    script:
5      - pip install sphinx sphinx-rtd-theme
6      - cd doc ; make html
7      - mv build/html/ ../public/
8    artifacts:
9      paths:
10       - public
11    only:
12      - master
```

# Артефакты и кэш

- Артефакты и кэш передаются между джобами
- Кэш просто передается, и все
- А артефакты хранятся некоторое время, вы можете загрузить их сами через UI
- Передаются между этапами артефакты с помощью dependencies

# Stage

If stage is not defined, the job uses the test stage by default.

```
stages:  
  - build  
  - cleanup_build  
  - test  
  - deploy  
  - cleanup
```

```
build_job:  
  stage: build  
  script:  
    - make build
```

```
cleanup_build_job:
```

# before\_script

- Use `before_script` to define an array of commands that should run before each job's script commands, but after artifacts are restored.

```
1  before_script:  
2    - python -V # Print out python version for debugging  
3    - pip install virtualenv  
4    - virtualenv venv  
5    - source venv/bin/activate
```

# Переменные

Packages and registries

Infrastructure

Monitor

Analytics

Wiki

Snippets

Settings

General

Integrations

Webhooks

Repository

Merge requests

CI/CD

Packages and registries

## Variables

Collapse

Variables store information, like passwords and secret keys, that you can use in job scripts. [Learn more](#).

Variables can be:

- Protected:** Only exposed to protected branches or protected tags.
- Masked:** Hidden in job logs. Must match masking requirements. [Learn more](#).

Environment variables are configured by your administrator to be [protected](#) by default.

| Type     | ↑ Key    | Value | Options   | Environments  |
|----------|----------|-------|-----------|---------------|
| Variable | TEST_ONE | ***** | Protected | All (default) |

Add variable

Reveal values

# Webhook (pipeline trigger)

## Pipeline triggers

[Collapse](#)

Trigger a pipeline for a branch or tag by generating a trigger token and using it with an API call. The token impersonates a user's project access and permissions. [Learn more](#).

Manage your project's triggers

### Description

[Add trigger](#)

No triggers exist yet. Use the form above to create one.

These examples show how to trigger this project's pipeline for a branch or tag.

In each example, replace `TOKEN` with the trigger token you generated and replace `REF_NAME` with the branch or tag name.

### Use cURL

```
curl -X POST \
  --fail \
  -F token=TOKEN \
  -F ref=REF_NAME \
  https://gitlab.com/api/v4/projects/23219995/trigger/pipeline
```

### Use .gitlab-ci.yml

# Merge request

workflow:

rules:

- if: `$CI_PIPELINE_SOURCE == 'merge_request_event'`



# Постановка на расписание



## Scheduling Pipelines





The pipelines schedule runs pipelines in the future, repeatedly, for specific branches or tags. Those scheduled pipelines will inherit limited project access based on their associated user.

Learn more in the [pipeline schedules documentation](#).



All 1 Active 1 Inactive 0

New schedule

| Description | Target   | Last Pipeline | Next Run    | Owner  |   |
|-------------|----------|---------------|-------------|--|---|
| daily       | 🔗 master | None          | in 14 hours |  Gleb Borisenko |    |

# Как это дело дебажить локально

- В контейнере в баше тестить все команды
- Поставить gitlab-runner локально, ставить команду слип в джобы, заходить в контейнеры и дебажить

# Бонус: Gitlab Pages

- Позволяет запустить статический сайт из пайплайна
- В отдельном стейдже прописывается установка и запуск, например, джекилл, путь public делается артефактом, ииииии....  
Все.

# Jenkins

- Более сложный и обширный инструмент для запуска пайплайнов
- Здесь немного другая терминология. Пайплайн из gitlab – джоба в Jenkins, а джоба из gitlab – stage в Jenkins.
- Пайплайны пишутся на своем языке, Groovy. Но он очень похож на yaml конфиги из gitlab, просто дает больше возможностей «из коробки» (без необходимости лезть в скрипты на другом языке)
- Есть классный UI для ручного запуска джоб, с полноценным выбором параметров выполнения в формочках
- Также есть триггеры и хуки

# Итог

1. Continuous Integration включает в себя этапы тестирования и сборки приложения.
2. Continuous Delivery (Deployment) включает в себя этапы настройки окружений и деплоя в них.
3. Цель CI/CD - сократить разрыв между разработкой приложений и управлением их работой с помощью автоматизации процессов.
4. Не так важно, какой инструмент выбрать для CI/CD, главное - использовать его.
5. Для одного и того же проекта можно собирать разные пайплайны CI/CD – выбирайте наиболее простой вариант.

# Смотрите в следующей лекции

- Поговорим про пайплайны и оркестрацию
- Затронем технологии (Airflow или Dagster, большого значения не имеет)
- А на след-след лекции посмотрим версионирование: DVC, MFlow