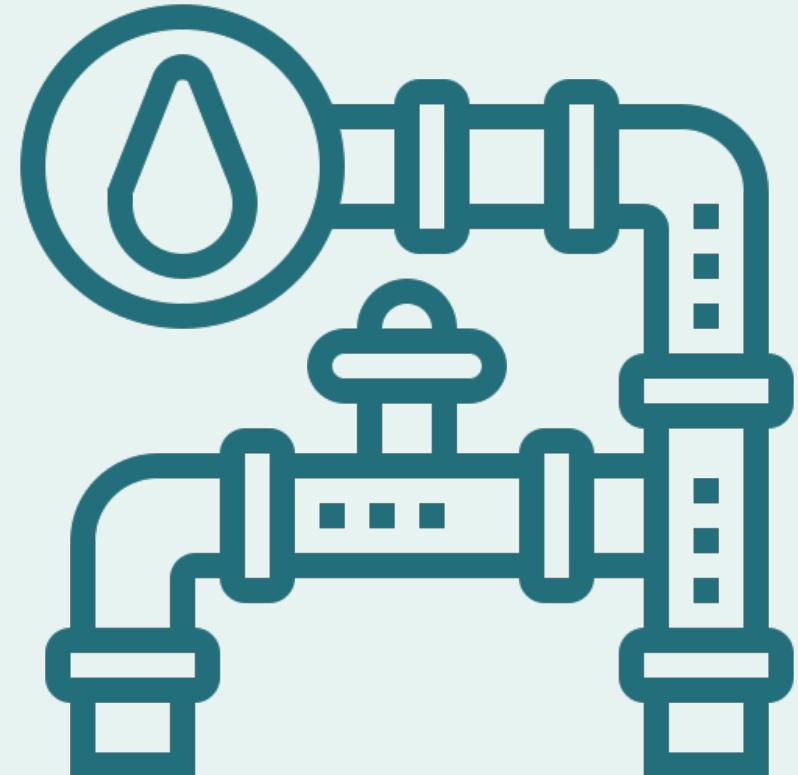


# Современный CI/CD пайплайн для python микросервисов



Денис Аникин

<https://xfenix.ru/>

# Денис Аникин

Кто я такой:

— работаю в Райффайзен банке



<https://xfenix.ru>

# Денис Аникин

Кто я такой:

- работаю в Райффайзен банке
- team lead в команде Chat



<https://xfenix.ru>

# Денис Аникин

Кто я такой:

- работаю в Райффайзен банке
- team lead в команде Chat
- community lead в Python Community



<https://xfenix.ru>

# Денис Аникин

Кто я такой:

- работаю в Райффайзен банке
- team lead в команде Chat
- community lead в Python Community
- fullstack: разрабатываю на back на python и front на typescript, занимаюсь devops



<https://xfenix.ru>

# Цели рассказа

Расскажу коротко какие цели преследовал

## Помочь начать писать пайплайны

Важно для тех, кто никогда их  
ещё не писал

# Цели рассказа

Расскажу коротко какие цели преследовал

## Помочь начать писать пайплайны

Важно для тех, кто никогда их  
ещё не писал

## Натолкнуть на мысли

Для тех, кто уже пишет давно,  
я принес свой субъективный  
взгляд на вещи

# Цели рассказа

Расскажу коротко какие цели преследовал

## Помочь начать писать пайплайны

Важно для тех, кто никогда их  
ещё не писал

## Натолкнуть на мысли

Для тех, кто уже пишет давно,  
я принес свой субъективный  
взгляд на вещи

## Поделиться наработками

Всем остальным может  
пригодиться мой «суповой  
набор» на гитхабе к этой  
презентации



Ah, I see you're a man of culture as well.

# Термины

# Термины

Нудная часть

DevOps: не человек   , а набор практик



# Термины

Нудная часть

CI: continuous integration. Это когда все часто отправляют свои правки в главную ветку

# Термины

Нудная часть

CI: continuous integration. Это когда все часто отправляют свои правки в главную ветку

Мы все в основном играем в continuous isolation 

# Термины

Нудная часть

CD:

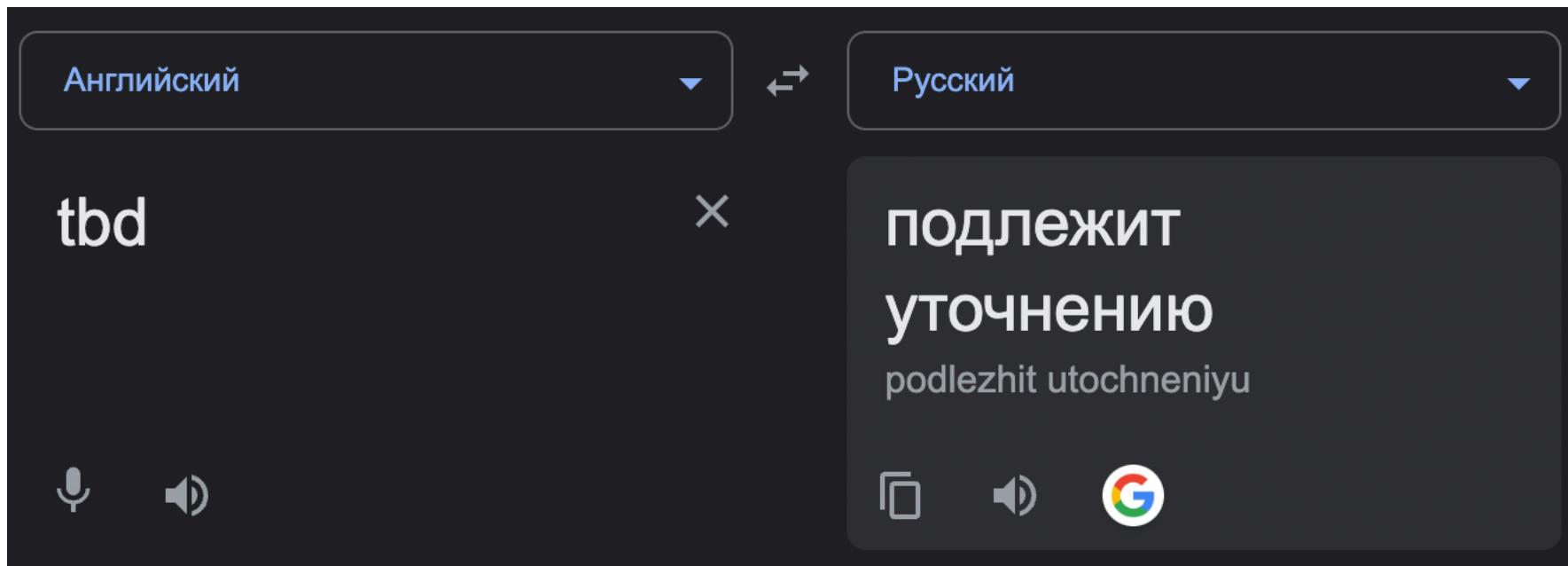
- CDL: Continuous DeLivery
- CDP: Continuous DePloyment

Это когда у вас (полу)автоматически доставляются релизы

# Термины

Нудная часть

TBD: ?



# Термины

Нудная часть

TBD: trunk based development

# Термины

Нудная часть

TBD: trunk based development

А теперь моя своеобразная схема! Я рисовал как мог.



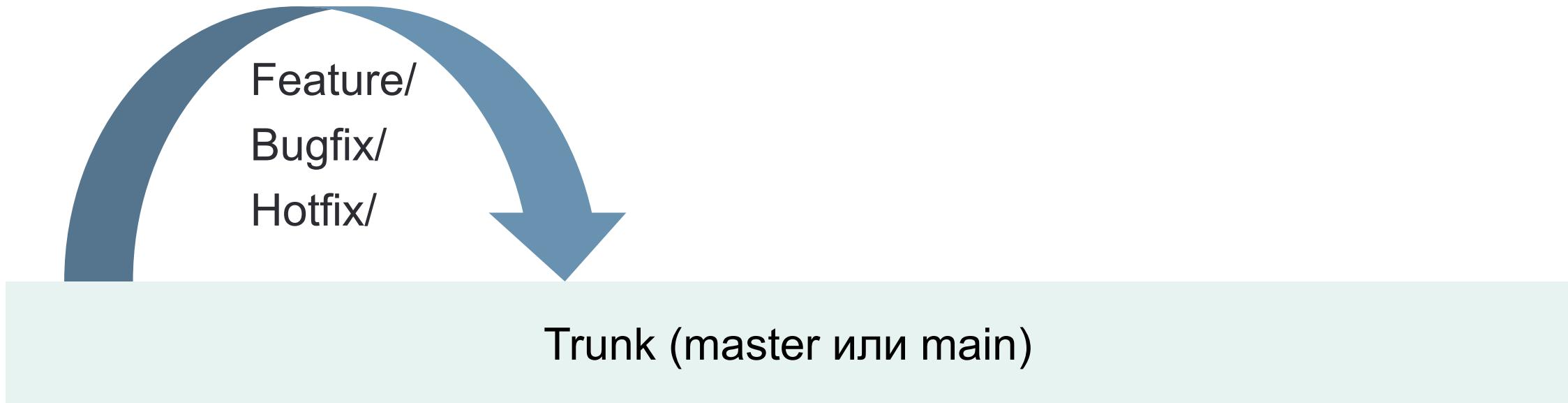
# TBD

Мой взгляд

Trunk (master или main)

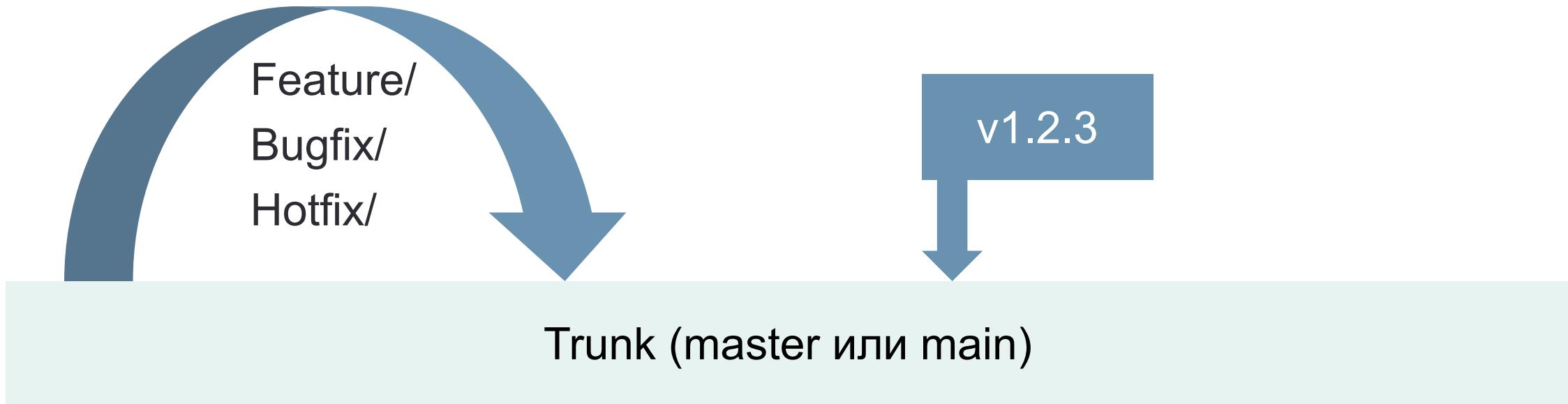
# TBD

Мой взгляд



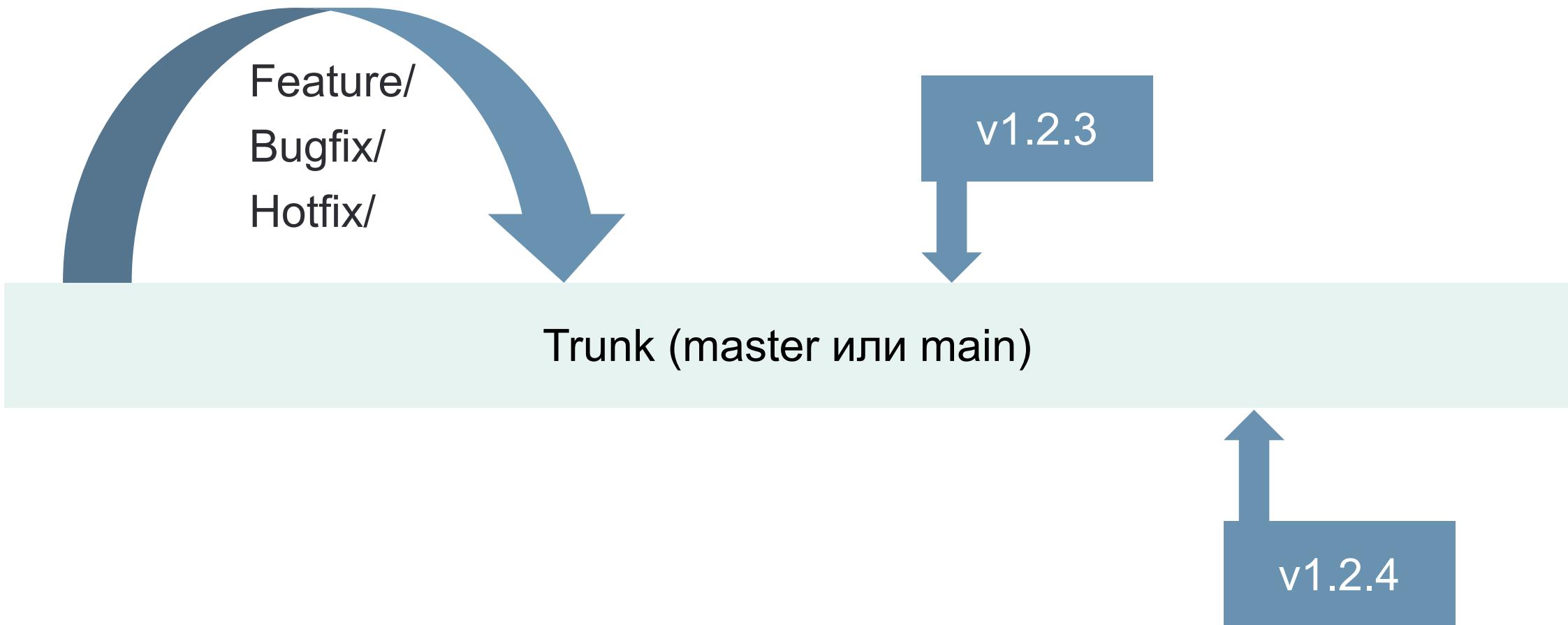
# TBD

Мой взгляд



# TBD

Мой взгляд



# Термины

Нудная часть

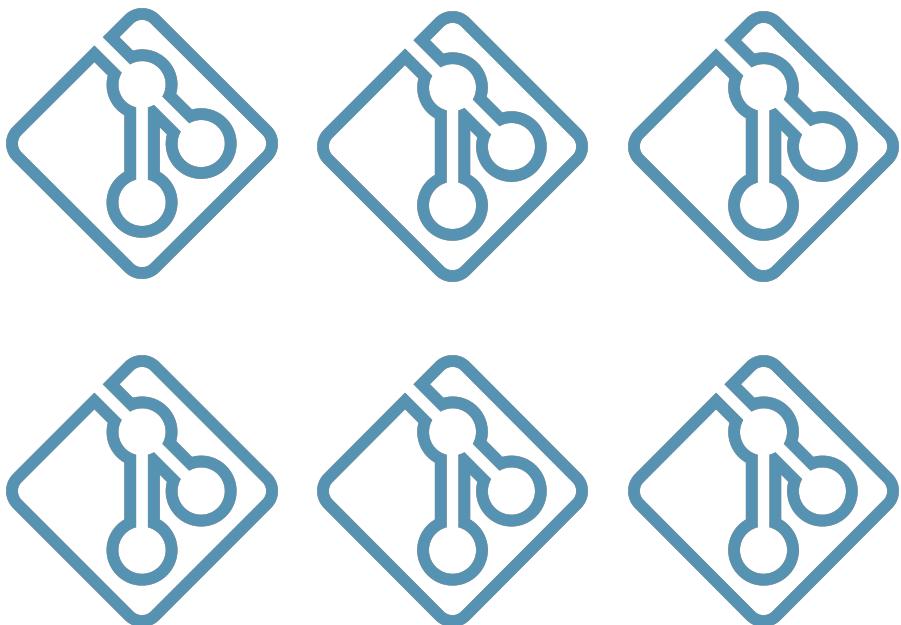
Монорепозиторий



# Термины

Нудная часть

Мульти репозиторий



**Из чего состоит  
приемлемый  
пайплайн?**

# Из чего состоит приемлимый пайплайн?

Давайте немного о CI части

— Сборка

# Из чего состоит приемлемый пайплайн?

Давайте немного о CI части

- Сборка
- Статический анализ (линтинг)

# Из чего состоит приемлимый пайплайн?

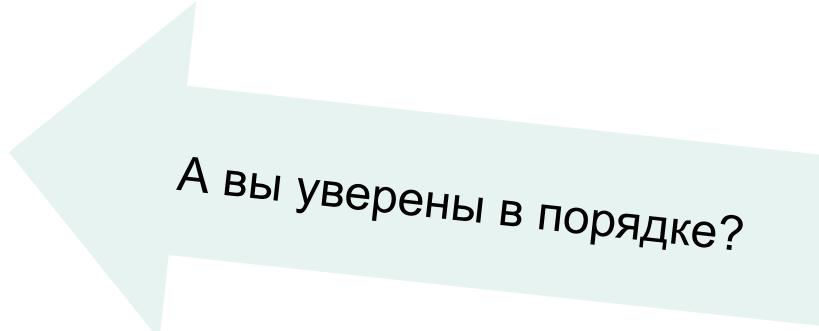
Давайте немного о CI части

- Сборка
- Статический анализ (линтинг)
- Тесты

# Из чего состоит приемлемый пайплайн?

Давайте немного о CI части

- Сборка
- Статический анализ (линтинг)
- Тесты



А вы уверены в порядке?

# А из чего состоит хороший пайплайн?

Давайте немного о CI части

- Сборка
- Статический анализ (линтинг)
- Тесты
- Безопасность?

# А из чего состоит хороший пайплайн?

Давайте немного о CI части

- Сборка
- Статический анализ (линтинг)
- Тесты
- Безопасность?
- Чистка «соплей»

x

# Кратко об истории создания нашего пайплайна

# 2019 год

Мы начали разработку чат-бота втроем

— У нас был bamboo





## Create a new plan

Configure plan Configure tasks

### Configure tasks

Each plan has a default job when it is created. In this section, you can configure the Tasks for this plan's default job. You can add more jobs to this plan once the plan has been created.

A task is an operation that is run on a Bamboo working directory using an [executable](#). An example of task would be the execution of a script, a shell command, an Ant Task or a Maven goal. [Learn more about tasks](#).

**Source Code Checkout**  
Checkout Default Repository

Final tasks Are always executed even if a previous task fails  
Drag tasks here to make them final

Add task

#### Source Code Checkout configuration

How to use the Source Code Checkout task

##### Task description

Checkout Default Repository

Disable this task

You can check out one or more repositories with this Task. You can choose to check out the Plan's *Default Repository* or specify a *Specific Repository*. You can add additional repositories to this Plan via the [Plan configuration](#).

##### Repository\*

Bitbucket Cloud - Totw Repo

*Default* always points to Plans default repository.

##### Checkout Directory

(Optional) Specify an alternative sub-directory to which the code will be checked out.

Force Clean Build

Removes the source directory and checks it out again prior to each build. This may significantly increase build times.

+ Add repository

**Save** Cancel

### Enable this plan?

Yes please!

By selecting this option your plan will be available for building and change detection straight away.  
do not select this option if you have advanced configuration changes to make after creation.

**Create** Cancel

# 2020 год

- Число репозиториев перешагнуло за 20
- Появился gitlab

# О двойной докеризации

HEY BRO



NICE PIPELINE

# О «двойной докеризации»

Это важно

Очень часто разработчики делают пайплайн так:

```
do-all-things:  
  stage: test  
  image: python:3.8-slim  
  script:  
    - apt-get update -y  
    - apt-get install -y pgdev kerberos enchant-2  
    - pytest .
```



# А ЧТО ТУТ НЕ ТАК?

Сборка неповторяема / не идемпотентна

```
do-all-things:  
  stage: test  
  image: python:3.8-slim  
  script:  
    - apt-get update -y  
    - apt-get install -y pgdev kerberos enchant-2  
    - pytest .
```

# А ЧТО ТУТ НЕ ТАК?

Дублируем Dockerfile

```
do-all-things:  
  stage: test  
  image: python:3.8-slim  
  script:  
    - apt-get update -y  
    - apt-get install -y pgdev kerberos enchant-2  
    - pytest .
```

# Что делаю я в пайплайне?

Итак, «двойная докеризация»

```
almost-real-pipeline:  
  script:  
    ...  
    - docker build . -t fancy-tag  
    - docker run -t fancy-tag:latest bash -c "pytest ."  
    ...
```

# В чём помогает «двойная» докеризация?

## В изоляции

Специфика проектов остается  
в этих проектах и не протекает  
в общий пайплайн

# В чём помогает «двойная» докеризация?

## В изоляции

Специфика проектов остается в этих проектах и не протекает в общий пайплайн

## В устраниении дупликации

Все дубликаты, характерные для devops автоматизаций можно устранить!

# В чём помогает «двойная» докеризация?

## В изоляции

Специфика проектов остается в этих проектах и не протекает в общий пайплайн

## В устраниении дупликации

Все дубликаты, характерные для devops автоматизаций можно устранить!

## В ускорении!

Ускоряем обновление на новые версии с помощью низкой связанности



# Есть проблемы

«Двойная докеризация» бьет в псину

- Во-первых, вам нужно коннектиться к докер демону (у gitlab хороший гайд) из «второго слоя»

[https://docs.gitlab.com/ee/ci/docker/using\\_docker\\_build.html](https://docs.gitlab.com/ee/ci/docker/using_docker_build.html)



# Есть проблемы

«Двойная докеризация» бьет в псину

- Во-первых, вам нужно коннектиться к докер демону (у gitlab хороший гайд) из «второго слоя»

[https://docs.gitlab.com/ee/ci/docker/using\\_docker\\_build.html](https://docs.gitlab.com/ee/ci/docker/using_docker_build.html)

- Копирование файлов и монтирование — боль

# Есть проблемы

«Двойная докеризация» бьет в псину

- Во-первых, вам нужно коннектиться к докер демону (у gitlab хороший гайд) из «второго слоя»

[https://docs.gitlab.com/ee/ci/docker/using\\_docker\\_build.html](https://docs.gitlab.com/ee/ci/docker/using_docker_build.html)

- Копирование файлов и монтирование — боль 

# Есть проблемы

«Двойная докеризация» бьет в псину

- Во-первых, вам нужно коннектиться к докер демону (у gitlab хороший гайд) из «второго слоя»

[https://docs.gitlab.com/ee/ci/docker/using\\_docker\\_build.html](https://docs.gitlab.com/ee/ci/docker/using_docker_build.html)

- Копирование файлов и монтирование — боль 
- Связываться с сайд контейнерами из services непросто

**Наконец уже к  
пайплайну!**

# Примеры к презентации, код

<https://github.com/xfenix/pycon2022>



# Ключевые особенности

Это всё на Gitlab

— «Двойная докеризация» 

# Ключевые особенности

Это всё на Gitlab

- «Двойная докеризация» 
- Централизация

# Ключевые особенности

Это всё на Gitlab

- «Двойная докеризация» 
- Централизация
- Пресеты

# Ключевые особенности

Это всё на Gitlab

- «Двойная докеризация» 
- Централизация
- Пресеты
- Использование include, reference, template для избегания основной боли пайплайнов: дупликации кода

# Ключевые особенности

Это всё на Gitlab

- «Двойная докеризация» 
- Централизация
- Пресеты
- Использование include, reference, template для избегания основной боли пайплайнов: дупликации кода
- Тестируем и отправляем в продакшн один и тот же образ

**FINALLY**

**WE ARE THE SAME**



# Лейаут: центральный репозиторий

```
└── cd
    ├── bot.yml
    └── helm.yml
└── ci
    ├── docker-build.yml
    ├── frontend.yml
    ├── pypi-poetry.yml
    ├── pypi.yml
    ├── python-postgres.yml
    └── python.yml
└── common
    > scripts
    > tests
        ├── ci-basic.yml
        ├── ci-python.yml
        ├── group-vars-chat-bot.yml
        ├── group-vars-chat-platform.yml
        ├── group-vars-chat.yml
        ├── jobs.yml
        └── stages.yml
```

# Лейаут: репозиторий сервиса

**variables:**

```
DSN_VARIABLE_NAME: "NV_CHB_DB_DSN_SHARED"  
PYLINT_ARGS: "--load-plugins pylint_django base core"
```

**include:**

- **project:** "chat/misc/generic-cicd"  
**file:** "common/group-vars-chat-bot.yml"
- **project:** 'chat/misc/generic-cicd'  
**file:** 'ci/python-postgres.yml'

**Вот теперь точно  
пайплайн**



# Шаг 1

Проверяем переменные

```
.vars-check-job:  
  extends: .default-job  
  stage: .pre  
  before_script:  
    - set -u
```

# Шаг 1

Проверяем переменные

```
.vars-check-job:  
  extends: .default-job  
  stage: .pre  
  before_script:  
    - set -u
```

```
check-front-necessary-variables:  
  extends: .vars-check-job  
  script:  
    - echo $PROJECT_SLUG $MAIN_USER
```

## Шаг 2

Собираем образ и пушим в registry. Здесь есть разные важные детали

```
.build-docker-job:  
  stage: build  
  variables:  
    STORAGE_DRIVER: vfs  
  script:  
    - set -x  
    ...  
    - buildah bud --jobs=0 --format=docker -t $TAG_FOR_CI ./Dockerfile  
    - buildah push $DOCKER_TAG_FOR_CI
```

# Примечание

`set -x`

в каждом пайплане вам будет полезен!

## Шаг 2: как генерируются теги

Деталь 1

```
VERSION=$(if [[ $CI_COMMIT_TAG ]]; then  
    echo ${CI_COMMIT_TAG/v}; else echo latest; fi)
```

```
NAME=$(if [[ $CI_COMMIT_TAG ]]; then  
    echo release; else echo $CI_COMMIT_BRANCH; fi)
```

```
$REGISTRY/$PROJECT/$CI_PROJECT_NAME/$NAME:$VERSION
```

```
# comany-hub.com/service-name/release:1.2.3  
# comany-hub.com/service-name/master:latest
```

## Шаг 2: кладем в образ мета-инфо

Деталь 2

```
echo "{\"version\": \"$DOCKER_IMAGE_VERSION\", \"service\": \"$CI_PROJECT_NAME\",  
\"project\": \"$PROJECT_SLUG\"}” > status.json
```

# Шаг 3: готовим окружение

Статические проверки (линтинг): подготовка

script:

```
...
- >-
podman run
-v $TOOLSET_PATH:$TOOLSET_PATH
-t $DOCKER_TAG_FOR_CI
bash -c "if id my-user > /dev/null 2>&1; then runAs='--user my-user'; fi &&
pip3 install $runAs -U 'pylint==2.12.2' 'mypy==0.942' 'isort==5.10.1'
'black==22.3.0' &&
...
...
```



## Шаг 3: pylint

Статические проверки (линтинг): pylint с поддержкой auto-discovery механизма, fail-under, baseline

```
/$TOOLSET_SCRIPTS_PATH/lint.py
-project_dir=$PROJ_DIR_IN_DOCKER
-action=lint_pylint
-pylint_args='$PYLINT_ARGS'
-pylint_score=${PYLINT_SCORE:-10.0} &&
```

## Шаг 3: mypy

Статические проверки (линтинг): mypy с поддержкой baseline, «fail under»

```
/$TOOLSET_SCRIPTS_PATH/lint.py
-project_dir=$PROJ_DIR_IN_DOCKER
-action=lint_mypy
-mypy_score=${MYPY_SCORE:-0} &&
```

# Шаг 3: остальное

Банально

```
python -m isort **/*.py --check --diff &&
```

```
python -m black **/*.py --check --diff
```

## Шаг 3

Что можно добавить ещё из полезного

— Radon (метрики кода)

## Шаг 3

Что можно добавить ещё из полезного

- Radon (метрики кода)
- Eradicate (обнаружение комментированного кода)

## Шаг 3

Что можно добавить ещё из полезного

- Radon (метрики кода)
- Eradicate (обнаружение комментированного кода)
- Vulture (обнаружение мертвого кода)

## Шаг 3

Что можно добавить ещё из полезного

- Radon (метрики кода)
- Eradicate (обнаружение комментированного кода)
- Vulture (обнаружение мертвого кода)
- Prospector (метрики кода, можно заменить pylint в теории)

## Шаг 3

Что можно добавить ещё из полезного

- Radon (метрики кода)
- Eradicate (обнаружение комментированного кода)
- Vulture (обнаружение мертвого кода)

или

- Prospector (разные тулы)
- Pylama — все вместе (разные тулы)

## Шаг 3: за рамками python

- Hadolint
- Kube-linter



# Шаг 4

## Тестирование

```
...
services:
  - name: postgres:14-alpine
    command: ["postgres", "-c",
              "shared_buffers=256MB", "-c", "max_connections=420"]
  - name: keydb:x86_64_v6.3.0
script:
  - export BEFORE_TESTS_HOOK="${BEFORE_TESTS_HOOK:-alembic upgrade head}"
  ...
  - export DB_IP_ADDR=$(cat /etc/hosts | grep "runner-" | awk '{print $1}'')
```



## Шаг 4

### Тестирование

```
- podman run  
...  
--add-host="$DB_HOST:${TEST_DB_IP_ADDR}"  
--add-host="$SENTINEL_NAME-0:${TEST_DB_IP_ADDR}"  
-e $DSN_NAME=postgresql://$PG_USER:$PG_PASS@$DB_HOST:$DB_PORT/$PG_DB  
bash -c "eval ${BEFORE_TESTS_HOOK:-} && pytest ."
```

...

## Шаг 4

### Примечания

— Мы используем для линтинга, тестов и дальнейшего деплоя один и тот же образ

## Шаг 4

### Примечания

- Мы используем для линтинга, тестов и дальнейшего деплоя один и тот же образ
- Лейаут докер образов такой: есть имя master и у него 1 тег latest, а так же release и теги — версии. Т.е. выглядит так:  
release:1.2.3

# Шаг 5

Б — безопасность



# Шаг 5

Б — безопасность



# Шаг 5

Б — безопасность



## Шаг 5

Б — безопасность



## Шаг 5

Б — безопасность



## Шаг 5

Б — безопасность



# Шаг 6

Пушим образ

```
publish-docker-image:  
  script:  
    - set -x  
    - !reference [.prepare-build-tag, script]  
    - echo ${FULL_DOCKER_TAG?-Undefined FULL_DOCKER_TAG var}  
    - echo ${DOCKER_TAG_FOR_CI?-Undefined DOCKER_TAG_FOR_CI var}  
    - !reference [.podman-login, script]  
    - skopeo copy docker://$DOCKER_TAG_FOR_CI docker://$FULL_DOCKER_TAG
```

**А что с CD?**

# Автоворсионирование

Под TBD

- GitVersion
- Semantic-release
- ...

# Автоворсионирование

Под TBD

- GitVersion
- Semantic-release
- ...
- Напишем свое!

# Автоверсионирование

Кратко

— Свой скрипт

Х



РИСУНОК  
ПИТОН КОДА

# Автоворсионирование

Кратко

- Свой скрипт
- Ищем нужный маркер ветки

# Автоверсионирование

Кратко

- Свой скрипт
- Ищем нужный маркер ветки
- Бампаем нужную часть

semver

# Автоверсионирование

Кратко

- Свой скрипт
- Ищем нужный маркер ветки
- Бампаем нужную часть  
semver

```
auto-semver:  
  stage: deploy  
  script:  
    - python -m pip install semver GitPython  
    - python ./auto-semver.py version  
  rules  
    if: $CI_COMMIT_BRANCH == "master"
```

# И заканчивая

Надо бы за собой почистить

— Можно чистить в конце пайплайна (здесь есть проблемы)

# И заканчивая

Надо бы за собой почистить

- Можно чистить в конце пайплайна (здесь есть проблемы)
- Можно чистить с помощью scheduled CI раз в сутки

# И заканчивая

Надо бы за собой почистить

- Можно чистить в конце пайплайна (здесь есть проблемы)
- Можно чистить с помощью scheduled CI раз в сутки
- Раз в сутки ходим и чистим release имя, удаляя все кроме N  
(3-5 обычно достаточно) последних версий/тегов



**MY NOSE ITCHES**

LCI 1998

# Возвращаемся назад!

У нас есть проблема с чисткой

— Пишем свой скрипт

# Возвращаемся назад!

У нас есть проблема с чисткой

- Пишем свой скрипт
- В репозиторий добавляем два «курсор»-тега: release-prod, release-prev

# Возвращаемся назад!

У нас есть проблема с чисткой

- Пишем свой скрипт
- В репозиторий добавляем два «курсор»-тега: release-prod, release-prev
- При каждом деплое мы просто удаляем старые и вешаем release-prod/release-prev теги «на тот тег», который деплоим

# Возвращаемся назад!

У нас есть проблема с чисткой

- Пишем свой скрипт
- В репозиторий добавляем два «курсор»-тега: release-prod, release-prev
- При каждом деплое мы просто удаляем старые и вешаем release-prod/release-prev теги «на тот тег», который деплоим

```
auto-semver:  
  stage: deploy  
  script:  
    - python -m pip install semver GitPython  
    - python ./auto-version.py mark  
  rules:  
    - if: '$CI_COMMIT_TAG =~ /^v\d+\.\d+\.\d+$/'
```

# Немного про deploy

— Если у вас docker/docker-compose, то вы делаете pull/restart/up/start

# Немного про deploy

- Если у вас docker/docker-compose, то вы делаете pull/restart/up/start
- Если у вас helm, всё чуть сложнее (но об этом в кулуарах, смотрите на гитхабе и не стесняйтесь писать мне лично)

# Немного про deploy

- Если у вас docker/docker-compose, то вы делаете pull/restart/up/start
- Если у вас helm, всё чуть сложнее (но об этом в кулуарах, смотрите на гитхабе и не стесняйтесь писать мне лично)
- И здесь есть куча всяких особенностей с формированием окружений под разные среды: staging, production, dev



# Спасибо!

Денис Аникин

<https://xfenix.ru/>

