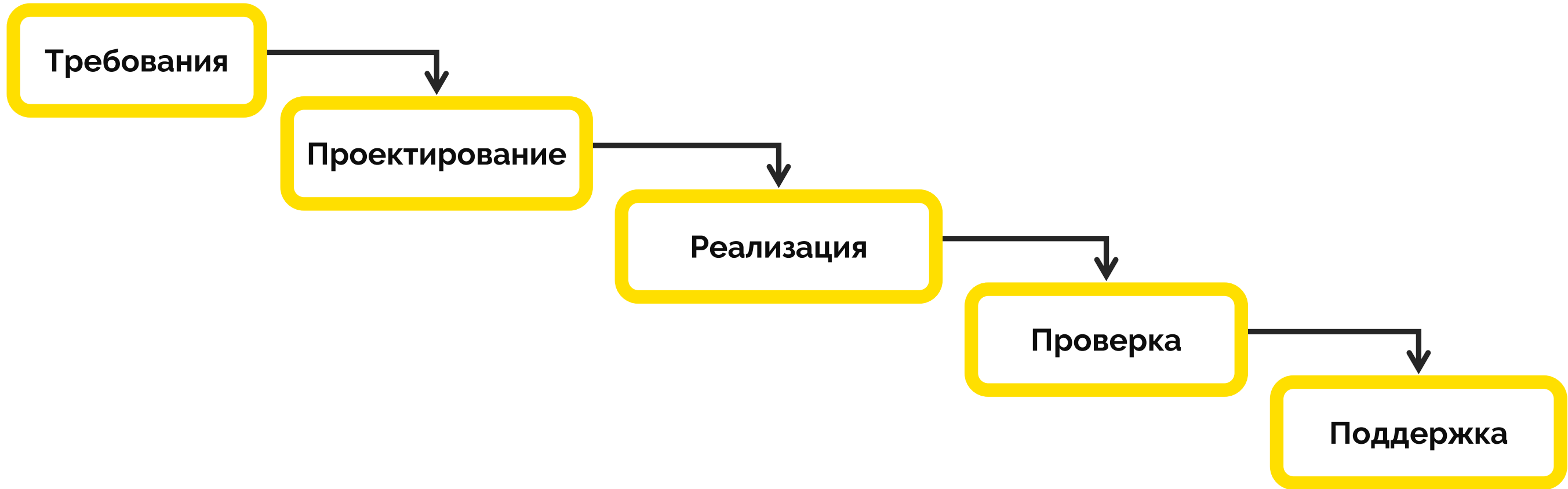




Сборка и непрерывная интеграция

Вспоминаем каскадную модель

Модель водопада [W. W. Royce, 1970]



Вспоминаем проблемы каскадной модели

1. Долгий цикл вывода нового функционала – **месяцы и годы**
2. Долгая обратная связь – **исправление занимало месяцы**
3. Сложность координации работы разных команд – **затраты на координацию**
4. Сложность исправления на поздних стадиях – **затраты на повторные циклы тестирования и релизы**
5. Большие трудозатраты на релиз – **затраты на сборку и внедрение «тяжелого» монолита**


Что хочет бизнес?

1. **Быстро** – вывод функционала за дни и недели
2. **Качественно** – исправление за часы или дни
3. **Недорого** – убрать лишние затраты

Это проблема!



Решение проблемы - 20 лет развития



1991г - Грэди Буч заложил теоретические основы **CI** в книге "Object-Oriented Analysis and Design with Applications", 1997г - Кент Бек и Рон Джеффрис в методологии Extreme Programming (XP) сделали **CI** одним из ключевых принципов разработки


2001г - группа экспертов опубликовала "**Манифест Agile**", который заложил основу для гибкой разработки и частых поставках, **CI** стал важной частью процессов **Agile**

2006г - Martin Fowler в статье "**Continuous Integration**" описал ключевые принципы частой интеграции кода, которые легли в основу **TBD**

2009г - Джон Оллспо и Пол Хэммонд представляют доклад "Десять деплоев в день: кооперация разработки (Dev) и эксплуатации (Ops) во Flickr" на конференции Velocity

2009г - Патрик Дебуа и Эндрю Шафер представили термин "**DevOps**" на конференции DevOpsDay, который заложил основу для автоматизации жизненного цикла, объединив Dev и Ops, разрыв между которыми замедлял поставку ПО, **CI** и **CD** стала основой **DevOps**

Решение проблемы - 20 лет развития



2009г - Джефф Хамбл и Дэвид Фарли выпустили книгу "Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation", ставшую основой для **CD**

2010г - появление термина "Continuous Deployment", поставка в прод после успешного прохождения тестов


2011г - Джеймс Льюис и Мартин Фаулер опубликовали знаковую статью "Microservices", где сформулировали основные принципы **MSA** (микросервисной архитектуры).

2011г – Пол Хэммонд в статье "Trunk-Based Development (TBD) explained" противопоставляет **TBD** традиционным моделям **GitFlow**

2013г - книга "The Phoenix Project" Джина Кима, Кевина Бера и Джорджа Спаффорда значительно ускорила популяризацию **DevOps**

2014г - появление контейнеров docker - один микросервис один контейнер

Решение проблемы - 20 лет развития



2014г - Mitchell Hashimoto из HashiCorp анонсирует выпуск Terraform, который использует декларативный язык HCL для создания **IaC** в облаке AWS

2015г - появление оркестратора **K8s** - управление множеством контейнеров

2015г - Пол Хэммонд в статье о **DevOps** обозначил **TBD** как ключевую практику для быстрой и безопасной разработки

2016г - Пол Хэммонд в статье "Why Trunk-Based Development Works Best" показывает преимущества **TBD** в больших командах, особенно в **DevOps** и **CI/CD**

2018г - Jez Humble, Nicole Forsgren, Gene Kim в статье "Accelerate" показали, что компании, использующие **TBD**, разрабатывают ПО быстрее и с меньшими рисками

Решение получилось комплексным

Уровень процессов

- Методологии Agile
- Методология DevOps

Уровень архитектуры

- MSA
- TBD
- CI \ CD

Уровень инфраструктуры

- IaC
- Docker \ K8s
- Prometheus \ Grafana \ Victoria
- ELK
- Sentry \ OpenTelemetry

Continuous Integration

непрерывная интеграция

Continuous Integration - методология разработки, направленная на регулярное и автоматическое объединение изменений в коде.

Ключевые особенности CI:

Частые коммиты - разработчики регулярно вносят свои изменения в общий репозиторий (минимум один раз в день). Это помогает быстро выявлять конфликты и устранять их на ранней стадии.

Автоматизация сборки – ускоряет разработку, каждое изменение кода автоматически запускает процесс сборки, что позволяет убедиться в его работоспособности и корректности.

Автоматическое тестирование - после сборки выполняются автоматизированные тесты (модульные, интеграционные и другие), чтобы проверить, не нарушили ли изменения существующую функциональность.

Раннее выявление ошибок - CI обеспечивает быстрое обнаружение дефектов благодаря частым проверкам кода. Это снижает стоимость исправления ошибок, так как они устраняются до того, как попадут в продакшн.



Continuous Integration

непрерывная интеграция

Ключевые особенности CI:

Единое рабочее пространство - все разработчики работают с одной основной веткой кода, что упрощает управление версиями и минимизирует вероятность конфликтов при слиянии изменений.

Сокращение цикла обратной связи - быстрая интеграция и тестирование позволяют разработчикам оперативно получать информацию о состоянии их изменений и быстро вносить правки.

Continuous Integration

непрерывная доставка

Continuous Delivery - Методология позволяет автоматически готовить приложение к выпуску в любое время. Она расширяет практики Continuous Integration (CI), добавляя автоматизацию и стандартизацию процессов доставки программного обеспечения на этапах после сборки и тестирования.

Ключевые особенности CD:

Готовность к выпуску в любое время - приложение всегда находится в состоянии, готовом к выпуску на production. Это достигается за счет строгого контроля качества и автоматизации.

Постоянное тестирование - помимо модульных тестов, используются интеграционные, функциональные, нагрузочные и другие виды тестирования для проверки готовности приложения.

Автоматизация развертывания - все изменения кода, прошедшие автоматические тесты в CI, автоматически подготавливаются к развертыванию на различных средах (например, тестовой, staging или production).

Многослойные среды - изменения проходят через несколько сред (например, dev → test → staging → production), где каждая среда имитирует production как можно точнее.

Контроль версий артефактов - все артефакты сборки (бинарные файлы, конфигурации) версионизируются и отслеживаются для обеспечения предсказуемости развертывания.

Continuous Integration

непрерывное развёртывание

Continuous Deployment - методология при которой изменения в коде автоматически проходят все этапы проверки (сборка, тестирование, доставка) и разворачиваются в production-среде без необходимости ручного вмешательства.

Ключевые особенности CD:

Полная автоматизация - все этапы — от написания кода до его развертывания в production — полностью автоматизированы.

Непрерывное обновление - любое изменение, прошедшее тесты, немедленно развертывается в production без ручной проверки.

Высокая надежность тестирования - автоматические тесты должны быть настолько качественными и полными, чтобы исключить вероятность ошибок в production.

Малые и частые изменения - Deployment предполагает выпуск меньших изменений по сравнению с Delivery, что снижает риски и упрощает откат изменений при необходимости.

Быстрая обратная связь от пользователей - Изменения становятся доступными пользователям сразу после успешного развертывания, что позволяет быстрее реагировать на их отзывы.

Integration vs Delivery vs Deployment

Continuous Integration (CI) - фокусируется на автоматической сборке и тестировании кода.

Continuous Delivery (CD) - является расширением Continuous Integration, обеспечивает автоматическую доставку изменений в среду развертывания (например, dev, staging, qa), и ручное в production.

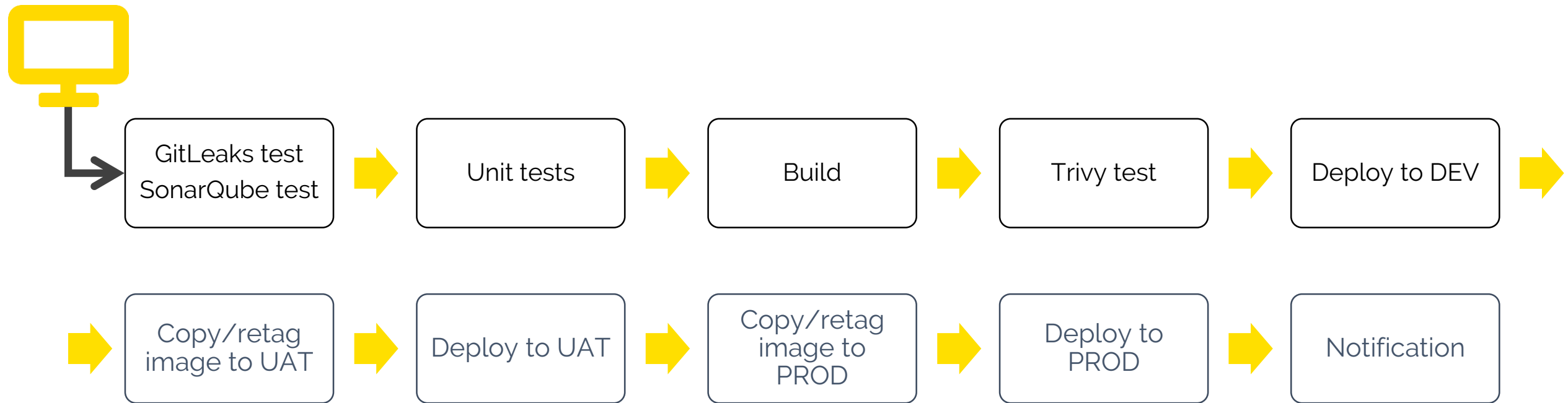
Continuous Deployment (CD) - является расширением Continuous Delivery, изменения автоматически доставляются в production без ручного вмешательства.



Правила эффективного CI/CD

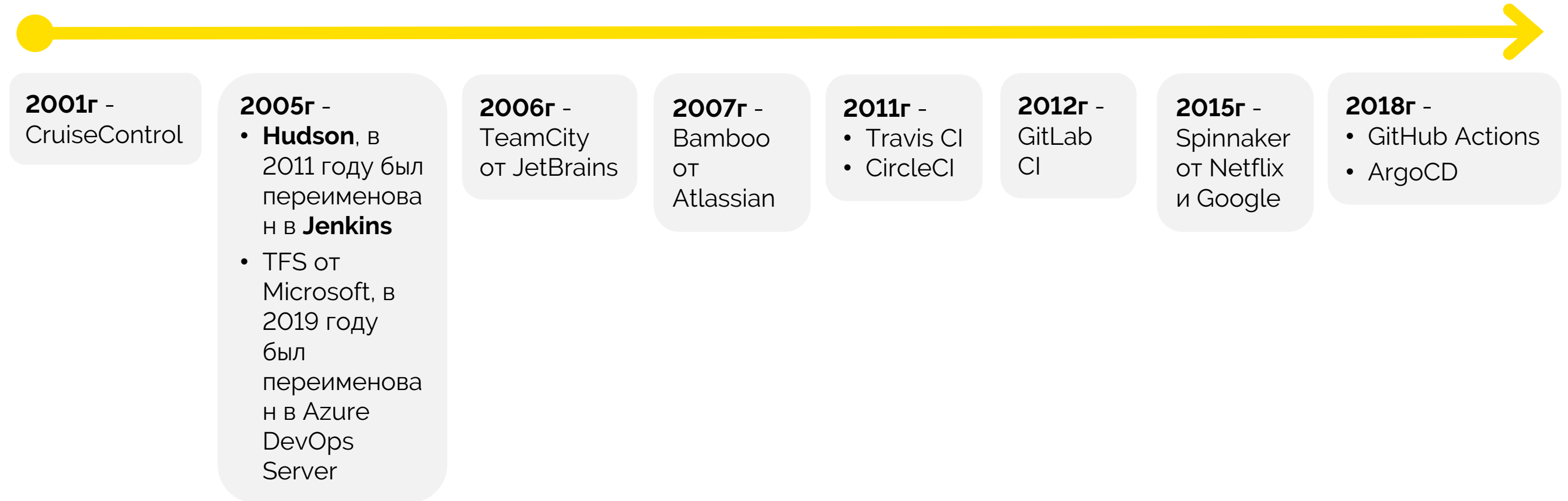
- Пайплайн должен быть быстрым (<30 минут) и простым
- Тесты должны быть изолированными и повторяемыми
- Использование Feature Toggles
- Использование безопасных методов deploy (uat, stage, canary, blue-green)
- Консистентность окружений через IaC
- Возможность отката изменений
- Сотрудничество и коммуникация между Dev и Ops
- Мониторинг метрик
- Быстрая обратная связь

PipeLine при использовании TBD



На UAT и PROD окружения попадает один и тот же образ с другим docker-тегом
После деплоя на PROD высылается автоматическое оповещение о релизе.

Инструменты CI / CD





Популярные инструменты

По количеству найденный результатов в google:

- Github Actions - **74млн**
- Gitlab - **30млн**
- Jenkins - **25млн**
- Azure Pipelines + Azure DevOps Server - **15млн**
- Bitbucket + Bamboo - **8млн**

Особенности конфигурации

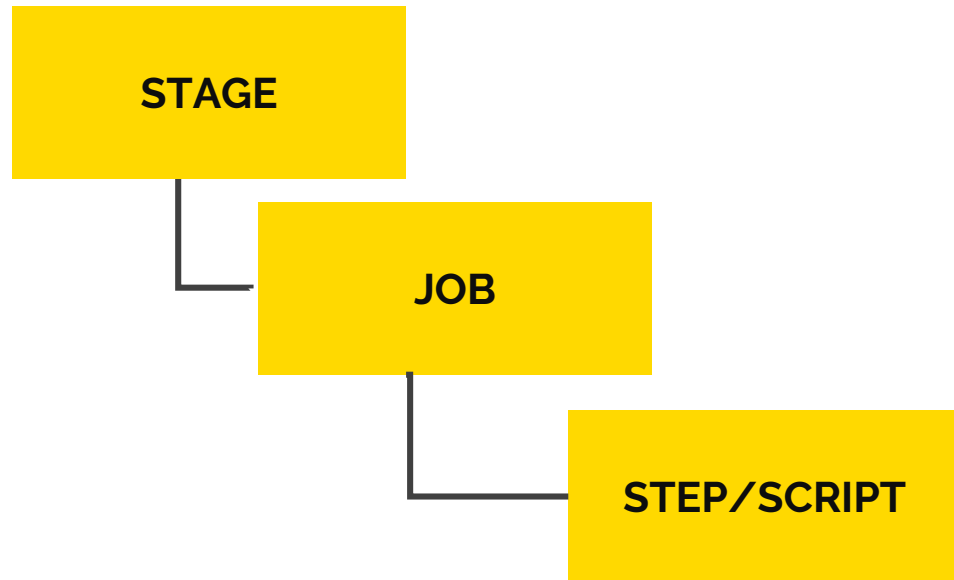
Каждая система имеет свой конфигурационный файл и синтаксис:

- Gitlab - .gitlab-ci.yml
- Github - .github/workflows/<workflow_name>.yml
- Jenkins - Jenkinsfile (groovy подобный синтаксис)
- Azure DevOps - azure-pipelines.yml

Каждая система характеризуется собственным сервером и раннерами на которых происходит сборка

Появилась выделенная должность DevOps - инженер который понимает 2-3 языка разметки

Основные компоненты pipeline



Stage - build / test / deploy выполняются последовательно, определяют набор заданий, которые выполняются на раннерах

Job - объединяет команды, могут выполняться параллельно на разных раннерах

Step/Script - последовательность команд внутри job, выполняются последовательно внутри одного раннера

Azure DevOps Server – Hello World!

```
trigger:
  branches:
    include:
      - main # Триггер только для ветки main

stages:
- stage: HelloWorld
  displayName: "Hello World Stage"
  jobs:
    - job: PrintHello
      pool:
        vmImage: 'ubuntu-latest'
      steps:
        - script: echo "Hello World from Azure DevOps!"
          displayName: "Hello World"

- stage: Build
  displayName: "Build Stage"
  dependsOn: HelloWorld
  jobs:
    - job: BuildApp
      pool:
        vmImage: 'ubuntu-latest'
      steps:
        - script: echo "Building..."
          displayName: "Build"
```

GitHub Actions– Hello World!

```
name: CI Pipeline
on:
  push:
    branches:
      - main # Триггер только для main

jobs:
  hello-world:
    name: "Hello World Stage"
    runs-on: ubuntu-latest
    steps:
      - name: Print Hello
        run: echo "Hello World from GitHub Actions!"

  build:
    name: "Build Stage"
    runs-on: ubuntu-latest
    needs: hello-world # Зависит от этапа hello-world
    steps:
      - name: Build
        run: echo "Building..."
```

GitLab– Hello World!

```
stages:
  - hello_world
  - build

hello-world-job:
  stage: hello_world
  script:
    - echo "Hello World from GitLab!"
  only:
    - main # Триггер только для main

build-job:
  stage: build
  script:
    - echo "Building..."
  only:
    - main
```

Jenkins– Hello World!

```
pipeline {
  agent any
  triggers {
    // Настраивается в интерфейсе Jenkins:
    // "Poll SCM" или через webhook (например, GitHub hook trigger).
    // Пример для опроса каждую минуту:
    pollSCM('* * * * *')
  }
  stages {
    stage('Hello World') {
      steps {
        echo 'Hello World from Jenkins!'
      }
      // Выполнять только для ветки main:
      when { branch 'main' }
    }
    stage('Build') {
      steps {
        echo 'Building...'
      }
      when { branch 'main' }
    }
  }
}
```

Основные типы триггеров на запуск pipeline

- **По событиям репозитория** – push/pull/merge request, можно фильтровать по веткам, тегам, путям, подходит для автоматической сборки и деплоя.
- **Ручной запуск** – по кнопке от пользователя, полезно для деплоя в прод с согласованием.
- **По расписанию** - в заданное время (например, каждую ночь), подходит для периодических задач - бэкапы, очистка данных.
- **Webhook** - при получении HTTP-запроса (POST) от внешней системы, интеграция с Telegram, Jira, требует настройки секретов.
- **Запуск при успехе другого пайплайна** - после успешного завершения другого пайплайна или этапа, подходит для создания цепочек задач (например, сборка → тесты → деплой).
- **При изменении файлов** - если изменены файлы в определенных директориях (например, docs/**), экономит ресурсы, пропуская ненужные этапы.

Секреты в pipeline

Secrets (секреты) в CI/CD — конфиденциальные данные, которые система использует для выполнения определенных операций, но которые должны оставаться в тайне, к таким данным относятся API-ключи, Пароли, Токены, Сертификаты, учетные данные.

Серверы CI/CD часто требуют доступа к широкому спектру таких секретов для тестирования, сборки и развертывания приложений. Секреты используются на всех этапах CI/CD — от доступа к репозиториям исходного кода до развертывания приложений в производственных средах.

Проблемы управления секретами:

- Встраивание секретов непосредственно в конфигурационные файлы небезопасно
- Стандартные механизмы CI/CD - систем для передачи через переменные окружения небезопасны
- Отсутствие централизованного хранения небезопасно
- Без автоматизации ротации возникают операционные издержки



Секреты в pipeline

Рекомендации по работе с секретами

Внедрение принципа наименьших привилегий

- Создание сервисных аккаунтов для разных этапов pipeline и разных уровней доступа
- Регулярный аудит и отзыв неиспользуемых разрешений

Регулярная ротация секретов

- Использование разовых паролей
- Автоматизация ротации
- Поддержка версионности секретов

Включение аудита

- Отслеживание обращений к секретам
- Мониторинг изменений конфигурации
- Настройка оповещений о подозрительной активности

Защита конфигурации CI/CD pipeline

- Защита веток в репозитории
- Сканирование зависимостей и контейнеров

Использование систем управления секретами

- HashiCorp Vault
- Google Secrets Manager
- Azure KeyVault

GitLab– Секреты

Project Variables

- Настраиваются в Settings -> CI/CD -> Variables.
- Можно пометить как Masked (скрыть в логах) и Protected (только для защищенных веток).

Group Variables

На уровне группы проектов.

File Variables

Секреты, сохраняемые в файл (например, сертификаты).

```
job:
  script:
    - echo $API_KEY # Если переменная API_KEY создана в настройках
```

Azure DevOps Server – Секреты

Pipeline Variables (Pipelines → Edit → Variables)

Помечаются как Secret (значение маскируется).

Variable Groups (Pipelines → Library → Variable groups)

- Группы переменных, которые можно подключить к пайплайну.
- Интеграция с Azure Key Vault.

Service Connections (Project Settings → Service Connections)

Токены для доступа к внешним сервисам (AWS, Docker Hub и т.д.).

```
steps:  
  - bash: echo $(SECRET_KEY) # Синтаксис $(VAR_NAME)
```

GitHub Actions – Секреты

Repository Secrets

- Хранятся на уровне репозитория.
- Доступны всем workflow в этом репозитории.

Organization Secrets

- На уровне организации.
- Можно ограничить доступ к определенным репозиториям.

Environment Secrets

Привязываются к окружению (например, production).

Требуют approval для запуска.

```
steps:  
  - name: Use secret  
    run: echo ${ secrets.API_KEY }
```

Тегирование образов в pipeline



Проблема – имеем десятки коммитов в день, десятки сборок в день, десятки артефактов в день, несколько окружений. А еще можем на один и тот же коммит делать по несколько сборок. Как связывать между собой образы с разных окружений, разных сборок и разных коммитов?

Что имеем:

- **Коммит** – есть короткий и полный хэш коммита, также можем проставить тэг
- **Сборка** – есть номер сборки (buildid)
- **Артефакт** – есть имя и тэги

Тегирование образов в pipeline

Рекомендации по работе с секретами

SemVer – для обозначения версий используется формат **MAJOR.MINOR.PATCH (v1.2.3)**.



Плюсы – общепринятый, понятный формат, показывает какие изменения внесены (новая функциональность, исправления, критические обновления).



Минусы - сложность управления, требует дисциплины в команде для правильного обновления версий, без дополнительных меток (например, для разных веток) теги могут повторяться, сложно связывать между собой версии разных микросервисов из состава одного сервиса.

Чаще всего используют при работе по GitFlow.

Тегирование образов в pipeline

Комбинированный – на основе даты/времени/хэша/номера сборки build-20250414171125 / build-50054



Плюсы – идентификация каждого микросервиса не зависит от других.

Минусы – сложно связывать между собой версии разных микросервисов из состава одного сервиса.

Чаще всего используют при работе по TBD.

Тегирование образов в pipeline

Наша практика

- Используем **TBD**
- В название артефакта добавляем постфикс с именем окружения **(dev/qa/uat/stage/prod)** – так проще группировать, администрировать артефакты в registry и применять политики очистки
- К артефакту добавляем тег с номером сборки **buildid**, от тега хэшем отказались, т.к. один и тот же коммит можем пересобирать несколько раз
- Привязка хэша коммита к сборке есть в истории сборок
- После деплоя артефакта на окружение, к соответствующему коммиту в git добавляем тэг с именем окружения и номером сборки – **deployed_to_prod_952295**

Тегирование образов в pipeline

Наша практика

The screenshot displays the GitLab web interface for the repository `te-tfs-build-cleaner`. The left sidebar shows navigation options: Overview, Boards, Repos, Files, Commits, Pushes, Branches, Tags, Pull requests, Pull Request Dashboard, Repository templates, and Pipelines. The main content area shows the commit history for the `master` branch. The commit history list includes the following entries:

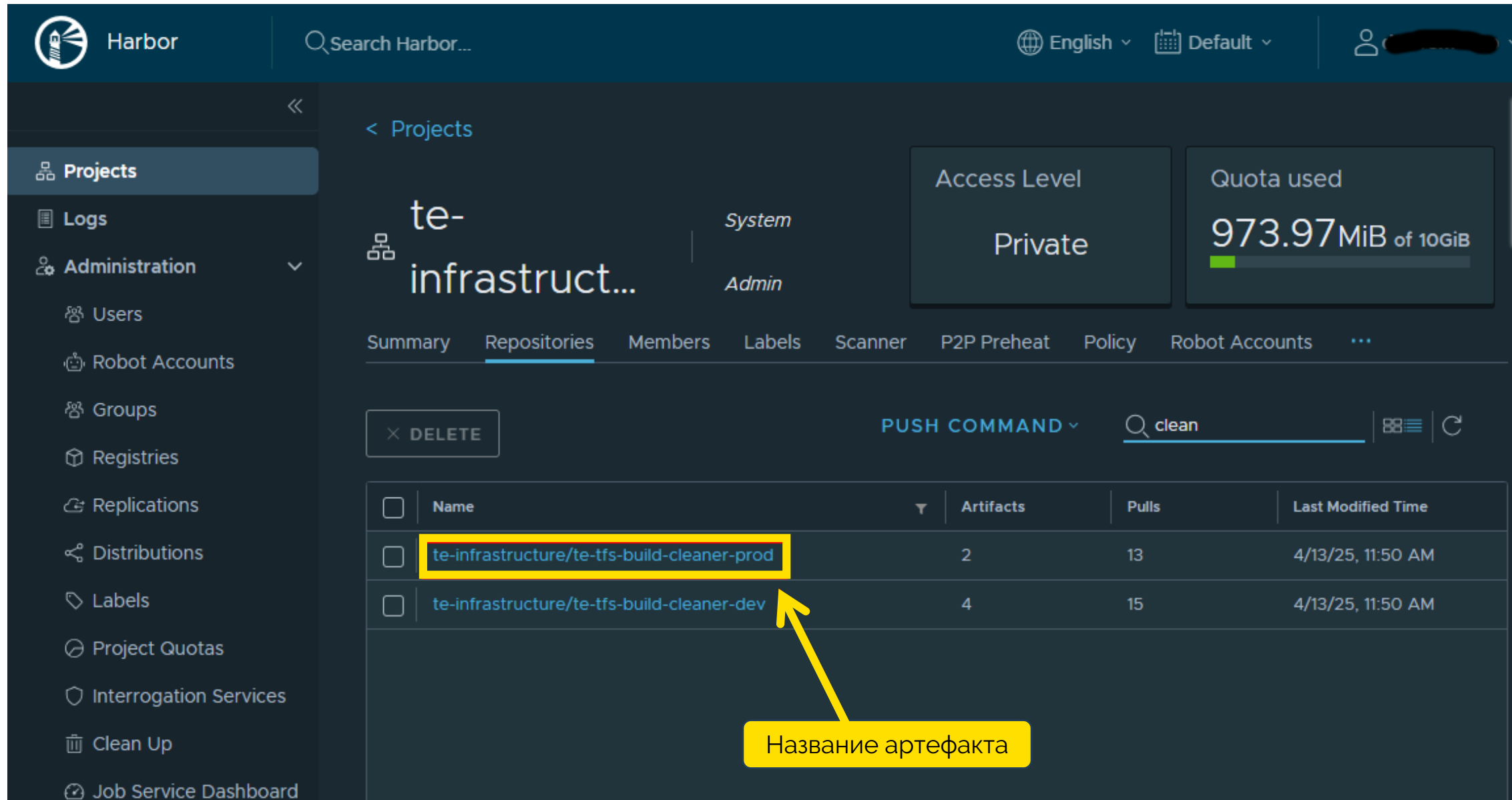
- Updated values-prod.yaml** (commit `0e4a81f4`, 18 мар. at 17:08) with a tag `deployed_to_prod_952295`.
- edit repo** (commit `b50b4fd6`, 13 мар. at 15:14).
- edit namespace** (commit `4f5a7e0c`, 13 мар. at 14:33).
- edit namespace** (commit `8ee7f792`, 13 мар. at 14:31).
- Deleted te-tfs-cleaner-groups-perm-handler.ps1** (commit `5a9321d7`, 5 мар. at 11:07).
- Added te_tfs_build_cleaner.ps1** (commit `162566f2`, 5 мар. at 11:06).
- alert test done** (commit `315b6cba`, 22 янв. at 14:20).

Annotations in the image include:

- A red arrow pointing from the **Commit** label in the sidebar to the commit history list.
- A red arrow pointing from the `deployed_to_prod_952295` tag in the commit history to the **Окружение и сборка** (Environment and Build) section.

Тегирование образов в pipeline

Наша практика



The screenshot shows the Harbor web interface. The left sidebar contains navigation links: Projects, Logs, Administration (Users, Robot Accounts, Groups, Registries, Replications, Distributions, Labels, Project Quotas, Interrogation Services, Clean Up, Job Service Dashboard). The main content area shows the 'te-infrastruct...' project details. The 'Repositories' tab is selected, displaying a table of repositories. A yellow box highlights the repository 'te-infrastructure/te-tfs-build-cleaner-prod', and a yellow arrow points to it from a label 'Название артефакта'.

Harbor

Search Harbor...

English

Default

< Projects

te-infrastruct...

System

Admin

Access Level

Private

Quota used

973.97MiB of 10GiB

Summary Repositories Members Labels Scanner P2P Preheat Policy Robot Accounts

DELETE

PUSH COMMAND

clean

<input type="checkbox"/>	Name	Artifacts	Pulls	Last Modified Time
<input type="checkbox"/>	te-infrastructure/te-tfs-build-cleaner-prod	2	13	4/13/25, 11:50 AM
<input type="checkbox"/>	te-infrastructure/te-tfs-build-cleaner-dev	4	15	4/13/25, 11:50 AM

Название артефакта

Тегирование образов в pipeline

Наша практика

The screenshot shows the Harbor web interface. The left sidebar contains navigation links: Projects, Logs, Administration (Users, Robot Accounts, Groups, Registries, Replications, Distributions, Labels, Project Quotas, Interrogation Services, Clean Up, Job Service Dashboard, Configuration). The main area displays the project 'te-tfs-build-cleaner-prod' under the 'Artifacts' tab. Above the table are buttons for 'SCAN', 'STOP SCAN', and 'ACTIONS', along with a 'COPY PULL COMMAND' field. The table lists artifacts with columns: Artifacts, Tags, Signed, Size, Vulnerabilities, and Labels. Two artifacts are shown, with the first one highlighted by a yellow box. A yellow arrow points from this box to a yellow callout box at the bottom right.

Artifacts	Tags	Signed	Size	Vulnerabilities	Labels
sha256:d0251004	952295	⊗	55.51MiB	Ⓢ 117 Total - 1 Fixable	
sha256:8e3f5fa5	949281	⊗	55.52MiB	Ⓢ 125 Total - 1 Fixable	

Крайняя версия артефакта, с тэгом номера сборки

Infrastructure as Code (IaC)



Проблема – окружение состоит из множества сервисов, необходимо иметь несколько окружений (dev \ uat \ qa \ stage). Часто у отдельного разработчика свое выделенное окружение. Иногда они ломаются. Требуется уметь быстро поднимать новое чистое окружение.

Решение – инфраструктура описана в коде и через CI\CD разворачивается.

Ключевые особенности:

- управление инфраструктурой через код, а не ручные процессы
- скорость, повторяемость, версионность, снижение ошибок

IaC – популярные инструменты Ansible

Ansible — безагентский инструмент для автоматизации конфигурирования серверов и ПО

- Создал Michael DeHaan, как альтернативу Chef и Puppet
- первый релиз 2012 году
- в 2015г куплен Red Hat
- название ansible позаимствовано из книг фэнтези (Игра Эндера, Планета Роканнон), ansible — устройство для мгновенной связи между мирами

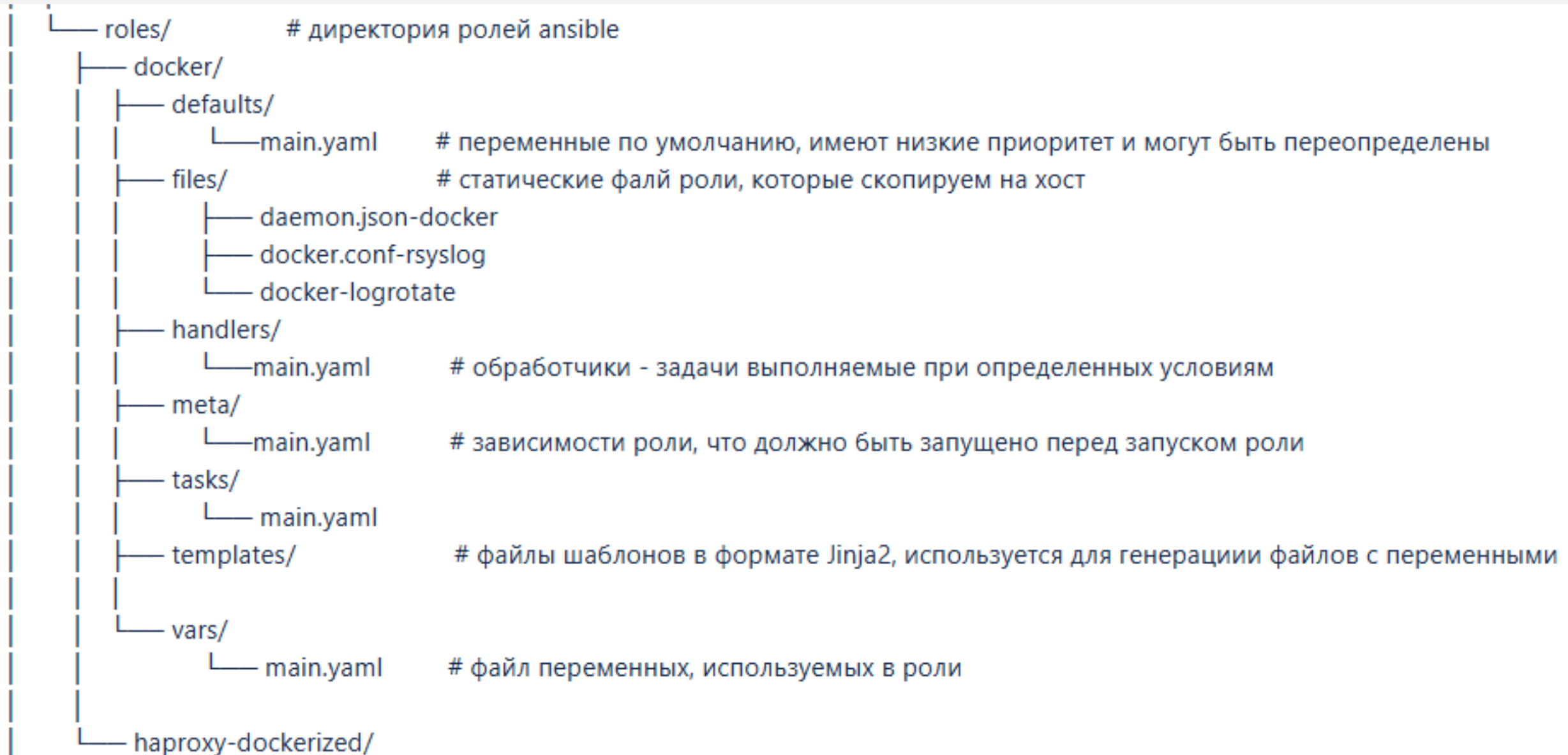
Ключевые особенности:

- YAML-синтаксис
- подключается по SSH (Linux), WinRM (Windows)
- agent less архитектура
- идемпотентность – независимо от текущего состояния, конфигурация приводится к целевой
- используется для настройки уже существующих VM и конфигурирования ПО
- основные компоненты - Playbooks, Inventory, Modules, Roles

laC Ansible – структура директории

```
|— .ansible/
|   |— files/           # общие статические файлы для ролей
|   |— templates/      # общие файлы-шаблоны для ролей
|   |— inventory/
|       |— dev/
|           |— group_vars/
|               |— vars.yaml # файл переменных для dev
|               |— hosts    # группа хостов для dev куда деплоим
|           |— prod/
|               |— group_vars/
|                   |— vars.yaml # файл переменных для prod
|                   |— hosts    # группа хостов для prod куда деплоим
|   |— playbook-haproxy-deploy.yaml # playbook, как обновляем конфиг HAProxy, обновление деплой
|   |— roles/           # директория ролей ansible
```

laC Ansible – структура директории



laC Ansible – структура директории

```
├── haproxy-dockerized/
│   ├── defaults/
│   │   └── main.yaml
│   ├── files/
│   │   ├── haproxy.cfg
│   │   ├── haproxy.conf-rsyslog
│   │   ├── haproxy-logrotate
│   │   └── lists/
│   │       ├── firehol_level1.netset
│   │       ├── firehol_level2.netset
│   │       ├── firehol_level3.netset
│   │       └── ....
│   │           ├── te_drop.netset
│   │           └── te_local.netset
│   ├── handlers/
│   │   └── main.yaml
│   ├── meta/
│   │   └── main.yaml
│   ├── tasks/
│   │   └── main.yaml
│   ├── templates/
│   │   ├── docker-compose.yaml.j2
│   │   ├── haproxy_stats_mail.sh.j2
│   │   └── update_firehol_lists.sh.j2
│   └── vars/
│       └── main.yaml
```

laC Ansible – структура playbook

main.yaml

Contents History Compare Blame

```
1 - name: Add Docker repository
2   yum_repository:
3     description: Docker CE Stable - x86_64
4     name: docker-ce-stable
5     baseurl: https://download.docker.com/linux/centos/{{ ansible\_facts\['distribution\_major\_version'\] }}/x86_64/stable
6     enabled: 'yes'
7     gpgcheck: 'yes'
8     gpgkey: https://download.docker.com/linux/rhel/gpg
9
10 - name: import docker gpg key
11   rpm_key:
12     state: present
13     key: https://download.docker.com/linux/rhel/gpg
14
15 - name: Prepare directory
16   file:
17     path: /opt/docker
18     state: directory
19     owner: root
20     group: root
21     mode: '0711'
22
23 - name: Create symlink
24   file:
25     src: /opt/docker
26     dest: /var/lib/docker
27     owner: root
28     group: root
29     state: link
30
31 - name: Create directory
32   file:
33     path: /etc/systemd/system/docker.service.d
34     state: directory
35     owner: root
36     group: root
```

IaC – популярные инструменты Terraform

Terraform — инструмент для оркестрации инфраструктуры (облако, сети, VM)

- создал Mitchell Hashimoto из компании HashiCorp
- первый релиз 2014 году
- отсылка к научной фантастике, термин «terraform» означает изменение климата планет для колонизации

Ключевые особенности:

- декларативный язык HCL (HashiCorp Configuration Language), смесь YAML и JSON
- концепция провайдеров/агент (providers, более 1000) для абстракции инфраструктуры
- главная идея - Write once, deploy anywhere
- стал стандартом для облачных сред, используется для создания любой сущности в облаке
- основные компоненты - Providers, Resources, State файлы, Plan/Apply

IaC – структура main.tf для cloud.ru

[Contents](#) [History](#) [Compare](#) [Blame](#)

```
1 terraform {
2   required_providers {
3     sbercloud = {
4       source = "sbercloud-terraform/sbercloud"
5     }
6   }
7 }
8
9 provider "sbercloud" {
10   region      = "ru-moscow-1"
11   access_key  = "██████████"
12   secret_key  = "████████████████████████████████████████"
13 }
14
15 # Create a VPC
16 resource "sbercloud_vpc" "myvpc" {
17   name = "vpc"
18   cidr = "192.168.0.0/16"
19 }
20
21 resource "sbercloud_vpc_subnet" "mysubnet" {
22   name          = "subnet"
23   cidr          = "192.168.0.0/16"
24   gateway_ip    = "192.168.0.1"
25
26   //dns is required for cce node installing
27   primary_dns   = "██████████"
28   secondary_dns = "8.8.8.8"
29   vpc_id        = sbercloud_vpc.myvpc.id
30 }
```

IaC – структура main.tf для gke

```
# провайдер
provider "google" {
  credentials = file("path/to/your/credentials.json")
  project    = "your-project-id"
  region     = "us-central1"
}

# VPC
resource "google_compute_network" "vpc" {
  name                = "my-vpc"
  auto_create_subnetworks = false
}

# подсеть
resource "google_compute_subnetwork" "subnet" {
  name            = "my-subnet"
  ip_cidr_range   = "10.0.0.0/16"
  network         = google_compute_network.vpc.self_link
  region          = "us-central1"
}

# GKE кластер
resource "google_container_cluster" "gke_cluster" {
  name     = "my-gke-cluster"
  location = "us-central1"

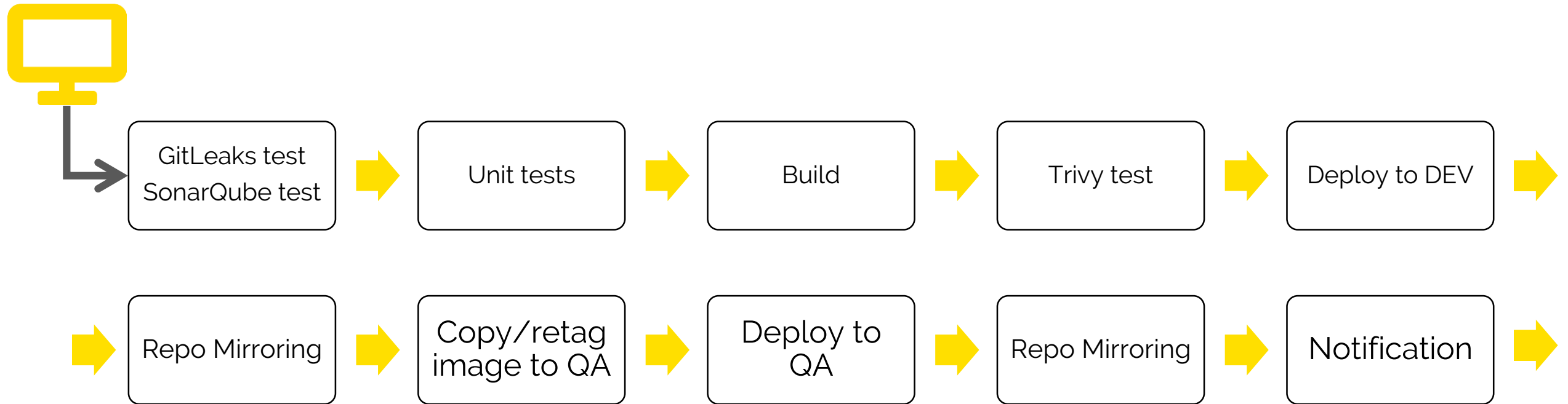
  remove_default_node_pool = true
  initial_node_count       = 1

  network    = google_compute_network.vpc.self_link
  subnetwork = google_compute_subnetwork.subnet.self_link
}
```

Сложная схема пайплайна

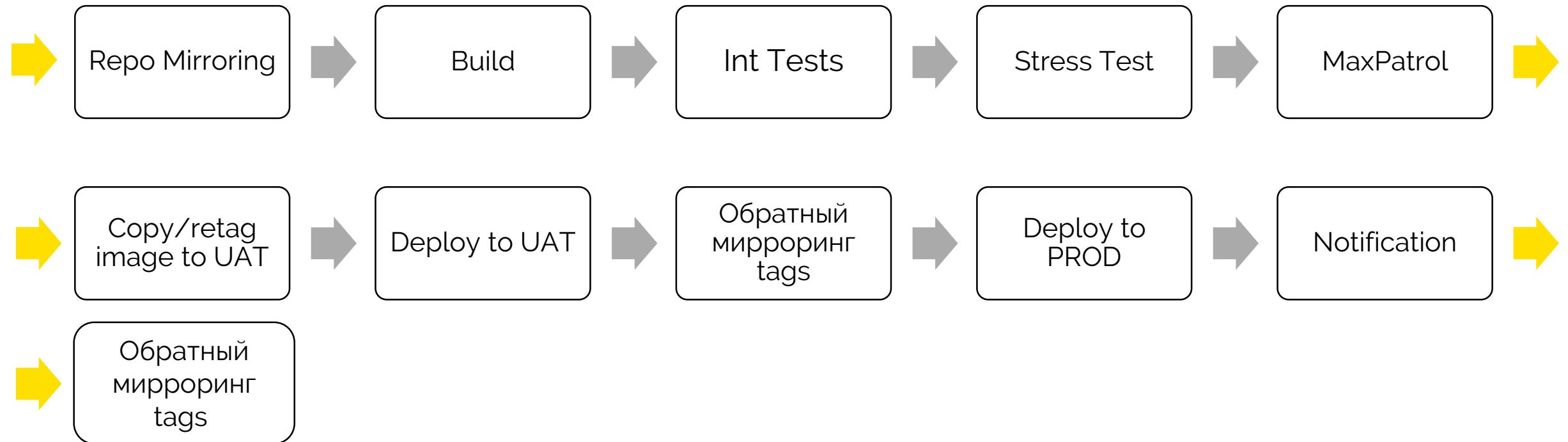


PipeLine на стороне Разработчика



Доступ только на уровне репозитория. Логика закладывается в пайплайны. Поддерживаются два набора пайплайнов для разных сторон. Один и тот же репозиторий миррорится от Разработчика Заказчику и обратно.

PipeLine на стороне Заказчика



После деплоя на UAT и PROD присходит обратный мирроринг тегов

Pipeline – как делать не надо

Сложный pipeline на 1700 строк – команде DevOps инженеров сложно поддерживать структуру кода для разных окружений и систем сборки. Использовать include и template для разбивки на блоки.

```
1701
1702 #####
1703 ### Ending
1704 #####
1705
1706 #stage prod_tag
1707 Prod Tag:
1708   image: $CI_REGISTRY_URL/$CI_REGISTRY_PATH/dind_jre8:$DIND_JRE8_VERSION
1709   only:
1710     refs:
1711       - tags
1712     variables:
1713 # поставил [REDACTED] вместо просто stable, пока не появился прод [REDACTED]
1714     - $CI_COMMIT_TAG =~ /^.[REDACTED]$/ && $CI_TARGET == [REDACTED]
1715   stage: prod_tag
1716   tags:
1717     - [REDACTED]
1718   script:
1719     - latest_tag=$(echo "$CI_COMMIT_TAG")
1720     - echo $latest_tag
1721     - tag_name="${latest_tag/stable/prod}"
1722     - echo $tag_name
1723     - git tag $tag_name
1724     - git push https://[REDACTED]:[REDACTED] --tags
1725   needs:
1726     - Deploy [REDACTED] Prod
1727     - [REDACTED] Deploy
1728   after_script:
1729     - echo "End CI"
1730
```

Pipeline – как делать не надо

```
41 # ----- Package -----
42
43 #
44 .copy-docker-images-to-prod:
45   <<: *epaas-restrictions
46   image: [REDACTED]/dind-helm-kubect1
47   stage: package
48   variables:
49     KUBE_VALUES: ${[REDACTED]_VALUES_PROD_1}
50     NEXUS_PATH_SRC: ${NEXUS_PATH_UAT}
51     NEXUS_PATH_DST: ${NEXUS_PATH_PROD}
52   script:
53     - export [REDACTED]_NEXUS_REPO_SRC=${[REDACTED]_NEXUS_SERVER}${NEXUS_PATH_SRC}
54     - export [REDACTED]_NEXUS_REPO_DST=${[REDACTED]_NEXUS_SERVER}${NEXUS_PATH_DST}
55     - [REDACTED]
56     - chmod +x ./kubernetes/retag_images.sh
57     - ./kubernetes/retag_images.sh
58     - echo "Finished!"
59   cache: {}
60   dependencies:
61     - define-version-epaas
62
63 # ----- Deploy -----
64
```

Используется личный образ из Docker Hub – вместо корпоративного и локального используется личный образ для сборки.

Pipeline – как делать не надо

```
# GitHub Actions (плохо!)
jobs:
  build:
    container: node:latest # Нет версии!
    steps: ...
```

Используется образ latest – может обновиться и упасть сборка, лучше фиксировать версию.

```
# Azure DevOps (плохо!)
steps:
- script: |
    npm install
    npm build
    npm test
    docker build .
    docker push
  displayName: "Build, test and deploy"
```

Монолитные этапы – лучше разделить на несколько stage.

Pipeline – как делать не надо

```
# GitLab CI (плохо!)
test:
  script:
    - echo "Создаю временные файлы..."
    - dd if=/dev/zero of=tempfile bs=1M count=1000 # 1 ГБ мусора!
```

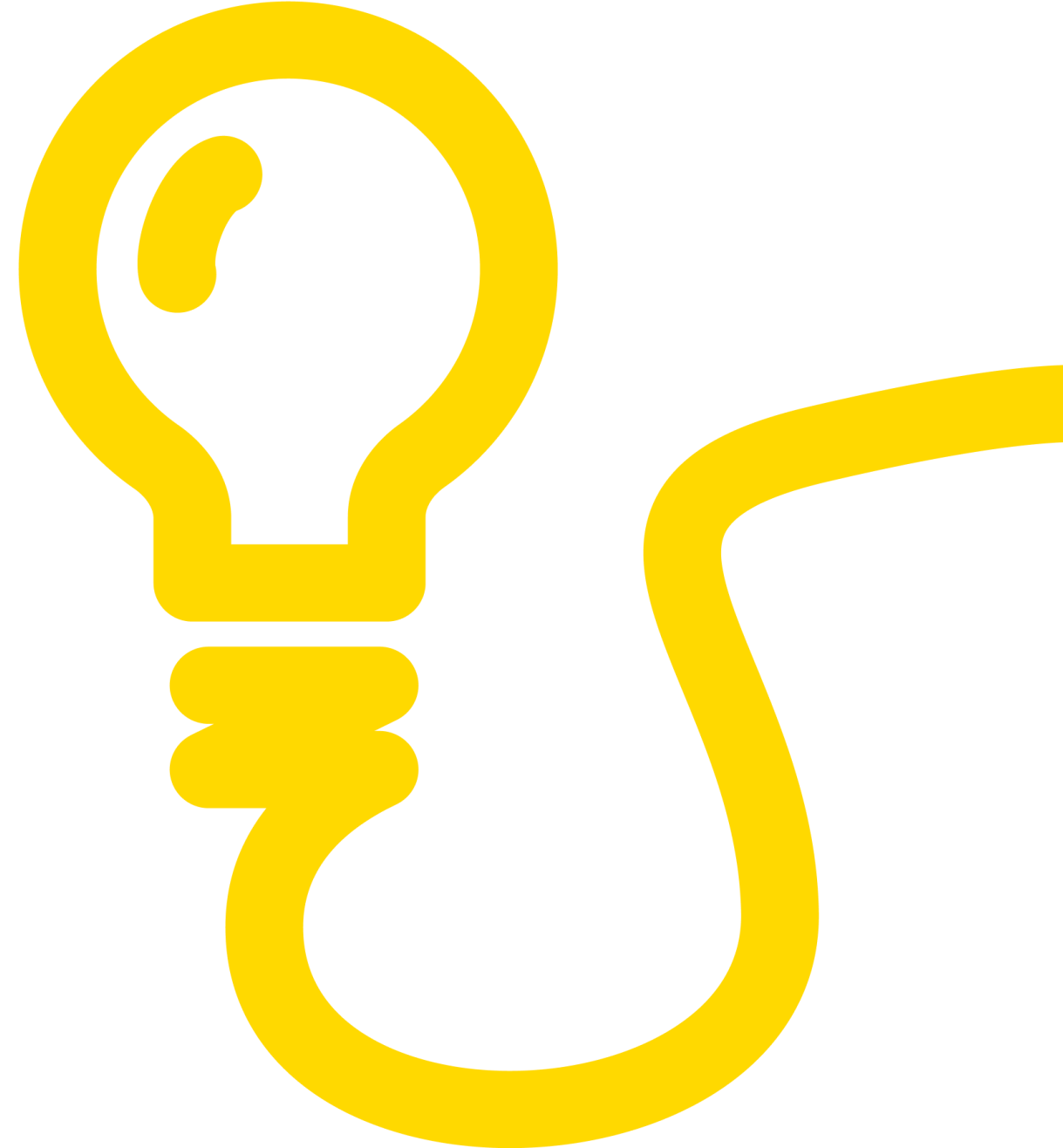
Не удаляются файлы после сборки – забивается место на runner, лучше добавить команду очистки.

```
# GitHub Actions (плохо!)
steps:
  - name: Deploy to AWS
    run: aws s3 cp . s3://my-bucket/ --access-key-id AKIAXXX --secret-access-key xyz123
```

Секреты в коде – лучше использовать через переменные.

Частые ошибки

1. Недостаточное тестирование (неполнота)
2. Отсутствие мониторинга
3. Игнорирование алертинга
4. Сложные pipeline
5. Игнорирование кэширования зависимостей
6. Игнорирование ошибок
7. Секреты в коде
8. Отсутствует взаимодействие Dev и Ops
9. Отсутствие документации
10. Отсутствие обратной связи от заказчика





Вопросы?



N*

**Спасибо
за внимание**

ФИТ Лекция N°15. 25'