Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ *Робототехники и комплексной автоматизации*

КАФЕДРА *Системы автоматизированного проектирования (РК-6)*

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К КУРСОВОМУ ПРОЕКТУ

НА ТЕМУ:

Проектирование микросервисной архитектуры и применение систем непрерывной интеграции для систематизации научно-технической документации

Студент РК6-82Б _____ **М.Т. Идрисов**
подпись, дата *фамилия, и.о.*

Руководитель курсового проекта _____ **А.П. Соколов**
подпись, дата *фамилия, и.о.*

Консультант _____ **А.Ю. Першин**
подпись, дата *фамилия, и.о.*

Москва, 2020 г.

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)» (МГТУ им. Н.Э. Баумана)

УТВЕРЖДАЮ

Заведующий кафедрой РК-6
(Индекс)

_____А.П. Карпенко_
(И.О.Фамилия)

«____» _____ 20___ г.

З А Д А Н И Е
на выполнение курсовой работы

по дисциплине __Технологии интернет_____

Студент группы __РК6-82Б_____

_____Идрисов Марат Тимурович_____
(Фамилия, имя, отчество)

Тема курсовой работы

Проектирование микросервисной архитектуры и применение систем непрерывной интеграции для систематизации научно-технической документации

Направленность КП (учебная, исследовательская, практическая, производственная, др.)

учебная

Источник тематики (кафедра, предприятие, НИР) _____

График выполнения КР: 25% к *3* нед., 50% к *10* нед., 75% к *15* нед., 100% к *17* нед.

Техническое задание: необходимо разработать подсистему автоматического построения обобщающего документа о ходе научно-образовательных работ по различным направлениям, функционирование подсистемы должно быть основано на разработке алгоритма интерпретации имен файлов, размещаемых в различных git-репозиториях.

Оформление курсовой работы:

Расчетно-пояснительная записка на _17__ листах формата А4.
Перечень графического (иллюстративного) материала (чертежи, плакаты, слайды и т.п.)

_____

Дата выдачи задания «18» февраля 2020 г.

Руководитель курсового проекта _____ ___А.П. Соколов_____
(Подпись, дата) (И.О.Фамилия)

Студент _____ ___М.Т. Идрисов_____
(Подпись, дата) (И.О.Фамилия)

Примечание: Задание оформляется в двух экземплярах: один выдается студенту, второй хранится на кафедре.

Аннотация

Работа посвящена проектированию микросервисной архитектуры и применению систем непрерывной интеграции к программному обеспечению, автоматизирующего процесс построения обобщающего документа о ходе научно-образовательных работ по различным направлениям. В работе описаны принципы микросервисной архитектуры, были определены критерии выбора базового контейнерного образа. В работе описан способ взаимодействия между основным и вспомогательными приложениями. Разработаны файлы конфигурации для размещения контейнеров на удаленном сервере, а также подготовлены конфигурационные файлы для внедрения приложений в цикл системы непрерывной интеграции.

Оглавление

1. Введение

Каждый успешный продукт приходит к состоянию, когда добавлять новые возможности в существующую кодовую базу становится тяжело настолько, что затраты на добавление новой функциональности превосходят все возможные выгоды от ее использования. Более того, при успешном развитии продукта компании постепенно увеличивают штат программистов. А это не только ускоряет темпы разрастания кодовой базы, но и повышает накладные расходы на администрирование.

С годами небольшое простое приложение превращается в чудовищный монолит. Точно так же некогда компактная команда разработчиков теперь состоит из нескольких команд, каждая из которых работает над конкретной функциональной областью. Приложение сильно разрастается, и разработка становится медленной и мучительной.

Основная проблема заключается в чрезмерной сложности приложения. Оно слишком большое для того, чтобы один разработчик мог его понять. В итоге исправление ошибок и реализация новых возможностей усложняются и занимают много времени. Разработчики не успевают выполнить работу в срок.

Усугубляет проблему то, что сложность, и так чрезмерная, обычно повышается экспоненциально. Если кодовая база плохо поддается пониманию, разработчик не сможет внести изменения подходящим образом. Каждое изменение усложняет код и делает его еще менее понятным.

Помимо борьбы с чрезмерной сложностью, разработчикам приходится иметь дело с замедлением ежедневных технических задач. Большое приложение перегружает и замедляет IDE. Сборка кода занимает много времени. Более того, из-за своей величины приложение долго запускается. В итоге затягивается цикл написания, сборки, запуска и тестирования кода, что плохо сказывается на продуктивности.

Еще одна причина того, почему изменения так долго доходят до промышленной среды, связана с длительным тестированием. Код настолько сложен, а эффект от внесенного изменения так неочевиден, а система требует ручного

тестирования. Кроме того, значительное время затрачивается на диагностику и исправление причин проваленных тестов. В итоге на завершение цикла тестирования требуется несколько дней.

Решением вышеуказанных проблем является переход от монолитной к микросервсиной архитектуре. Это стиль проектирования, когда большое приложение разбито на отдельные мини-приложения, реализующее узкоспециализированные функции, что значительно упрощает разработку и внедрение изменений в приложение. Также внедрение цикла непрерывной интеграции и развертывания существенно ускоряет процесс тестирования и развёртывания приложения на удаленном сервере.

Задача настоящего проекта заключалась во проектировании микросервисной архитектуры и внедрение цикла непрерывной интеграции для разработанной программной инфраструктуры автоматизации процесса обработки и сбора постоянно формируемой отчетной документации. Такая разработка позволит во много раз улучшить качество приложения, ускорить внедрение новой функциональности и исправление ошибок.

2. Основы микросервисной архитектуры

Современная архитектура ПО начала отходить от крупных монолитных приложений [1]. Теперь основное внимание в вопросах архитектуры уделяется достижению высокого уровня масштабируемости. «Разбивая монолит» на компоненты, инженерные организации предпринимали усилия по децентрализации управления изменениями, предоставляя командам больше контроля над тем, как функции вводятся в эксплуатацию. Повышая изолированность между компонентами, команды создателей ПО постепенно переходят к разработке распределенных систем, фокусируясь на написании менее крупных, более специализированных сервисов с независимыми циклами выпуска.

В приложениях, оптимизированных для выполнения в облаке, используется набор принципов, позволяющих командам более свободно оперировать способами ввода функций в эксплуатацию. По мере роста степени «распределенности»

приложений (в результате повышения степени изолированности, необходимой для предоставления большего контроля над ситуацией командам, владеющим приложением) возникает серьезная проблема, связанная с повышением вероятности сбоя при обмене данными между компонентами приложения. Неизбежным результатом является превращение приложений в сложные распределенные системы.

Архитектуры приложений, оптимизированных для работы в облачной среде, придают этим приложениям преимущества исключительно высокой масштабируемости, притом гарантируя их всеобщую доступность и высокий уровень производительности.

При использовании микросервисной архитектуры можно выделить главные преимущества ее использования [2]:

- модули можно легко заменить в любое время: акцент на простоту, независимость развёртывания и обновления каждого из микросервисов;
- модули организованы вокруг функций: микросервис по возможности выполняет только одну достаточно элементарную функцию;
- модули могут быть реализованы с использованием различных языков программирования, фреймворков, связующего программного обеспечения, выполняться в различных средах контейнеризации, виртуализации, под управлением различных операционных систем на различных аппаратных платформах: приоритет отдаётся в пользу наибольшей эффективности для каждой конкретной функции, нежели стандартизации средств разработки и исполнения;
- архитектура симметричная, а не иерархическая: зависимости между микросервисами одноранговые.

Философия микросервисов фактически «копирует» философию Unix [3], согласно которой каждая программа должна «делать что-то одно, и делать это хорошо» и взаимодействовать с другими программами простыми средствами: микросервисы минимальны и предназначаются для единственной функции. Основные изменения, в связи с этим, налагаются на организационную культуру, которая должна включать автоматизацию разработки и тестирования, а также

культуру проектирования, от которой требуется предусматривать «обход» прежних ошибок, исключение унаследованного кода.

Наиболее популярная среда для выполнения микросервисов — технология Docker [4]. В этом случае каждый из микросервисов изолируется в отдельный контейнер или небольшую группу контейнеров, доступную по сети другим микросервисам и внешним потребителям, и управляется средой оркестрации, обеспечивающей отказоустойчивость и балансировку нагрузки. Типовой практикой является включение в контур среды выполнения системы непрерывной интеграции [5].

3. Создание docker-образа приложения

Для создания docker-образа приложения, написанного на Python, требуется построить его поверх существующего образа — и есть много возможных вариантов. Существуют образы ОС, такие как Ubuntu и CentOS, а также существует множество различных вариантов python образов.

В результате анализа поставленной задачи были определены ряд критериев выбора базового образа:

1. **Стабильность.** Требуется, чтобы образ в долгосрочной перспективе содержал один и тот же базовый набор библиотек, структуру каталогов и инфраструктуру.

2. **Обновления безопасности:** требуется, чтобы базовый образ получал своевременные обновления безопасности для базовой операционной системы.

3. **Современные зависимости:** при создании сложного приложения существует зависимость от установленных в операционной системе библиотек и приложений. Требуется, чтобы нужные библиотеки имели самые новые и стабильные версии.

4. **Новая версия Python:** этот критерий не является основополагающим, т.к. требуемую версию Python можно установить самостоятельно, но заранее предустановленная версия Python экономит время и усилия на ее ручную установку.

5. **Небольшой размер образа:** при прочих равных условиях лучше иметь образ Docker меньшего размера, чем образ Docker большего размера.

Существуют четыре основные операционные системы, которые соответствуют вышеуказанным критериям.

– Ubuntu 18.04 (ubuntu:18.04 образ) был выпущен в апреле 2018 года, и, поскольку это релиз долгосрочной поддержки, он будет получать обновления безопасности до 2023 года [6];

– Ubuntu 20.04 (ubuntu:20.04 образ) был выпущен в конце апреля 2020 года, и, поскольку это релиз долгосрочной поддержки, он получит обновления безопасности до 2025 года [7];

– CentOS 8 (centos:8) был выпущен в 2019 году и будет иметь полные обновления до 2024 года и обновления до 2029 года [11];

– Debian 10 («Buster») был выпущен 6 июля 2019 года и будет поддерживаться до 2024 года [8].

Только Ubuntu 20.04 включает в себя последнюю версию Python.

Также существую «официальные python образы» [9], в которых уже установлены нужные версии Python (3.5, 3.6, 3.7, 3.8 бета и т.д.). Они также имею несколько вариацией:

- Debian Buster, с множеством установленных пакетов [10].
- slim вариант Debian Buster. В нем отсутствуют множество общих пакетов, поэтому сам образ намного меньше.

В результате анализа вышеуказанных docker-образов, было принято решение выбрать в качестве основы нашего приложения образ **python:3.8-slim-buster**. Он актуальнее, чем ubuntu:18.04, стабилен, не будет иметь изменений в библиотеках. Объем образа 60 МБ при загрузке и 180 МБ без сжатия [11], что говорит о его небольшом размере. Образ содержит последнюю версию Python и обладает всеми преимуществами Debian Buster.

Одной из задачей выпускной квалификационной работы является получение отчетов в формате TeX, с последующим преобразованием в PDF формат. Для решения этой задачи требуется дополнить базовый docker-образ

соответствующими утилитами пакетами для компиляции TeX файлов. Такой утилитой является latexmk и дополнительные пакеты с кириллическими шрифтами: texlive-lang-cyrillic и texlive-latex-recommended. *Dockerfile* с инструкциями для дополнения базового образа представлен в листинге 1.

Листинг 1. Dockerfile с пакетами для компиляции TeX файлов

```
FROM python:3.8-slim-buster
COPY /tmp /tmp

RUN apt-get update \
&& apt-get install c texlive-lang-cyrillic texlive-latex-recommended texlive-
pictures texlive-latex-extra -y \
&& cd "$_" \
&& latex hyphenat.ins \
&& mkdir -p /usr/share/texlive/texmf-dist/tex/latex/hyphenat \
&& mv hyphenat.sty /usr/share/texlive/texmf-dist/tex/latex/hyphenat
```

Базовый образ **python:3.8-slim-buster** состоит из стека слоев, которые доступны только для чтения (иммутабельны) (рисунок 1), а все изменения происходят в верхнем слое стека

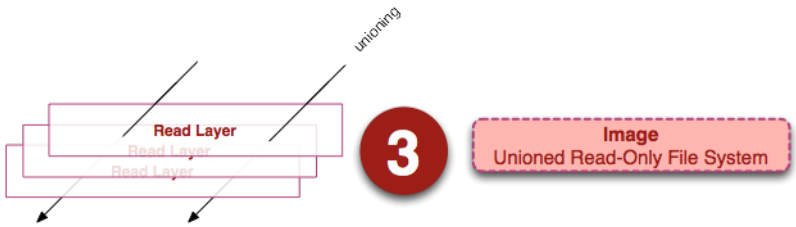

Рисунок 1 Общая структура базового Docker образа

Для дополнения базового образа требуется добавить верхний слой для записи сверху стека слоев (рисунок 2), записать изменения и преобразовать верхний слой в слой для чтения (рисунок 3).

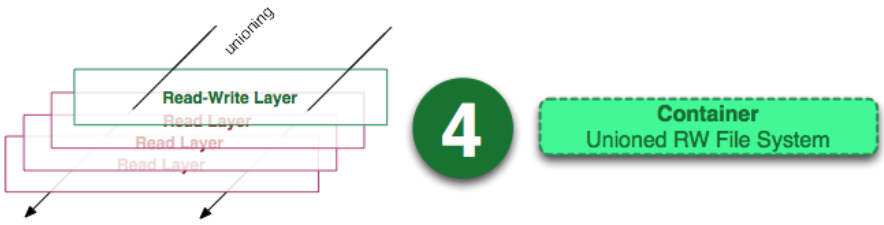

Рисунок 2 Добавление в стек верхнего слоя для записи

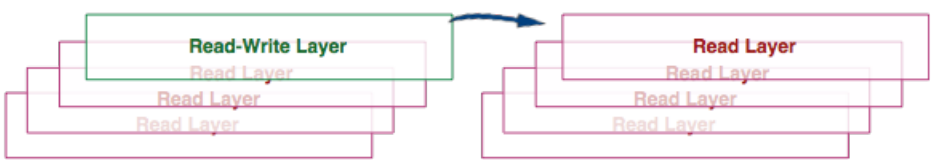

Рисунок 3 Преобразование верхнего слоя в слой для записи

Данную цепочку преобразований, выполняет команда **docker build**. Визуализация работы данной команды представлена на рисунке 4.

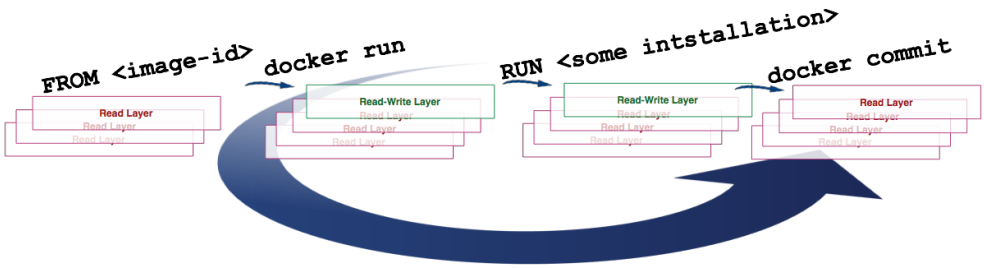

Рисунок 4 Цепочка преобразований, выполненная docker build

Команда **build** использует значение инструкции FROM из файла *Dockerfile* как базовый образ после чего: а) запускает контейнер (create и start); б) изменяет слой для записи; в) вызывает операцию commit.

Таким образом был получен базовый образ, содержащий утилиты для работы с TeX фалами, который был назван **python-latexmk.**

С помощью команды **docker pull** данный образ был размещен в Docker Hub – крупнейшую в мире библиотеку контейнерных образов [12], для возможности его использования в основном проекте.

4. Конфигурация основного и вспомогательных приложений

Помимо основного приложения систематизации научно-технической документации в проекте используется другие сервисы: Celery и RabbitMQ.

Celery — асинхронная очередь задач, основанная на распределенной передаче сообщений [13]. Она ориентирована на работу в реальном времени, но также поддерживает планирование задач. Единицы исполнения, называемые задачами, выполняются одновременно на одном или нескольких серверах. Задачи могут выполняться асинхронно (в фоновом режиме) или синхронно (в ожидании готовности).

В качестве брокера сообщений используется RabbitMQ — программный брокер сообщений на основе стандарта AMQP [14].

Для мониторинга и администрирования задач Celery используется веб-инструмент Flower [15].

Каждый из этих приложений размещён как отдельный docker-контейнер.

Для определения и запуска многоконтейнерных приложений используется инструмент compose [16]. Для работы с инструментом требуется выполнить предварительные шаги:

1) Определить базовоге приложение с помощью Dockerfile файла.

2) Определить сервисы, из которых состоит приложение, чтобы их можно было запускать вместе в изолированной среде. Требуется написать специальный YAML-файл под названием *docker-compose.yml*, в котором указана конфигурация для каждого сервиса.

3) Выполнить команду **docker-compose up**, которая запустит все приложения, указанные в *docker-compose.yml* файле.

В рамках текущей работы был определен следующий *docker-compose.yml* файл (листинг 2):

Листинг 2. docker-compose.yml файл для одновременного запуска нескольких контейнеров

```yaml
version: '3.1'
services:
  web:
    restart: always
    build:
      context: .
    env_file:
      - var.env
    volumes:
      - .:/opt/project
    command: ["web"]
    ports:
      - "9999:80"

  mq:
    hostname: mq
    image: rabbitmq:3-management
    ports:
      - "5672:5672"
      - "15672:15672"

  tasks:
    restart: always
    build:
      context: .
    links:
      - mq
    env_file:
      - var.env
    command: ['tasks']
```

```
  flower:
    restart: always
    build:
      context: .
    links:
      - mq
    env_file:
      - var.env
    volumes:
      - .:/opt/project
    ports:
      - "5555:5555"
    command: ["flower"]

  worker:
    restart: always
    build:
      context: .
    links:
      - mq
      - flower
    env_file:
      - var.env
    command: ['create_report']
```

Во время выполнения инструкции **command: [‘имя команды’],** инструмент compose извлекает из файла *docker-enrypoint.sh* соответствующие указанному имени команды и выполняет их в каждом контейнере соответственно. Написанный *docker-enrypoint.sh* файл представлен в листинге 3.

Листинг 3. docker-enrypoint.sh с именами и советующими им командами

```
#!/bin/bash
set -e

case "$1" in
web)
    python3 manage.py collectstatic --noinput
    python3 manage.py runserver 0.0.0.0:80
    ;;
tasks)
    exec celery worker -A reporter -l info --concurrency=1 -n reporter-crea-
tor_worker@%n
    ;;
flower)
    exec celery -A reporter flower --db=/flower/flower -l info
    ;;
create_report)
    exec python manage.py create_report 60
    ;;
*)
    exec "$@"
    ;;
esac
```

5. Внедрение приложения в цикл непрерывной интеграции

Для успешного автоматического тестирования и развертывания приложения и его компонент на удаленном сервере требуется инструмент, реализующий непрерывные методологии:

- Непрерывная интеграция (CI)
- Непрерывная доставка (CD)
- Непрерывное развертывание (CD)

Непрерывная интеграция работает путём добавления небольших фрагментов кода в кодовую базу проекта, размещённую в git-репозитории, и запуске конвейера скриптов для сборки, тестирования и проверки изменений кода перед их слиянием с основной веткой.

Непрерывная доставка и развёртывание состоят из следующего шага, разворачивая ваше приложение в промышленную эксплуатацию при каждом изменении.

Эти методологии позволяют отлавливать ошибки на ранних стадиях цикла разработки, гарантируя, что весь код, развернутый в производство, соответствует тем стандартам, которые были ранее установлены.

Требованиям для выпускной квалификационной работы является размещение отчетов в репозитории GitLab, который уже включает встроенный инструмент для разработки программного обеспечения с помощью непрерывных методологий – Gitlab CI [17].

Для автоматического тестирования и развертывания приложения был написан специальный файл-инструкция на языке YAML, содержащий выполняемые команды на каждом этапе цикла непрерывной интеграции.

Листинг 4. docker-enrypoint.sh с именами и советующими им командами

```
image: docker:latest

stages:
  - up

services:
```

```
  - docker:dind

before_script:
  - docker version
  - docker-compose version

deploy:
  variables:
    CI_DEBUG_TRACE: "true"
  only:
    - master
  stage: up
  script:
    - docker-compose down -v
    - docker-compose up --build -d
```

Указанный файл используется специальной утилитой GitLab Runner [18], которая используется для запуска заданий и отправки результатов в GitLab. В данном случает GitLab Runner во время изменений ветки master выполнит задание deploy. По команде **docker-compose down -v** уничтожатся все работающие контейнеры, далее по команде **docker-compose up --build -d** на основе *docker-compose.yml* файла конфигурации контейнеры переопределяться и запустятся в фоновом режиме.

6. Заключение

В текущем курсовом проекте спроектирована микросервисная архитектура и внедрена система непрерывной интеграции к программному обеспечению, автоматизирующего процесс построения обобщающего документа о ходе научно-образовательных работ по различным направлениям. Были описаны принципы микросервисной архитектуры, были определены критерии выбора базового контейнерного образа. Был описан способ взаимодействия между основным и вспомогательными приложениями. Разработаны файлы конфигурации для размещения контейнеров на удаленном сервере, а также подготовлены конфигурационные файлы для внедрения приложений в цикл системы непрерывной интеграции.

7. Список литературы

1. Осипенко, А.А. Переход от монолита к микросервисам [Электронный ресурс] / А.А. Осипенко. — Электрон. текстовые дан. — 2016. — Режим доступа: https://habr.com/ru/post/305826/
2. Microservices [Электронный ресурс] /. — Электрон. текстовые дан. — Режим доступа: https://en.wikipedia.org/wiki/Microservices
3. Basics of the Unix Philosophy [Электронный ресурс] /. — Электрон. текстовые дан. — Режим доступа: http://www.catb.org/esr/writings/taoup/html/ch01s06.html
4. Docker overview [Электронный ресурс] /. — Электрон. текстовые дан. — Режим доступа: https://docs.docker.com/get-started/overview/
5. Continuous Integration [Электронный ресурс] /. — Электрон. текстовые дан. — Режим доступа: http://wiki.c2.com/?ContinuousIntegration
6. Ubuntu 18.04.4 LTS (Bionic Beaver) [Электронный ресурс] /. — Электрон. текстовые дан. — Режим доступа: https://releases.ubuntu.com/18.04.4/
7. Ubuntu 20.04 LTS (Focal Fossa) [Электронный ресурс] /. — Электрон. текстовые дан. — Режим доступа: https://releases.ubuntu.com/20.04/
8. CentOS-8 (1911) Release Notes [Электронный ресурс] /. — Электрон. текстовые дан. — Режим доступа: https://wiki.centos.org/action/show/Manuals/ReleaseNotes/CentOS8.1911?action=show&redirect=Manuals%2FReleaseNotes%2FCentOSLinux8
9. Python - Docker Hub [Электронный ресурс] /. — Электрон. текстовые дан. — Режим доступа: https://hub.docker.com/_/python
10. Выпущен Debian 10 "buster" [Электронный ресурс] /. — Электрон. текстовые дан. — Режим доступа: https://www.debian.org/News/2019/20190706.ru.html
11. python:3.8-slim-buster [Электронный ресурс] /. — Электрон. текстовые дан. — Режим доступа: https://hub.docker.com/layers/python/library/python/3.8-slim-buster/images/sha256-

527bd4f643ae1885abeaa483cc675e2cee5b958612012d60ec10455ac5405270?context=explore

12. Docker Hub is the world's easiest way to create, manage, and deliver your teams' container applications [Электронный ресурс] /. — Электрон. текстовые дан. — Режим доступа: https://hub.docker.com/

13. Celery - Distributed Task Queue [Электронный ресурс] /. — Электрон. текстовые дан. — Режим доступа: https://docs.celeryproject.org/en/stable/index.html

14. RabbitMQ is the most widely deployed open source message broker [Электронный ресурс] /. — Электрон. текстовые дан. — Режим доступа: https://www.rabbitmq.com/documentation.html

15. Flower - Celery monitoring tool [Электронный ресурс] /. — Электрон. текстовые дан. — Режим доступа: https://flower.readthedocs.io/en/latest/

16. Overview of Docker Compose [Электронный ресурс] /. — Электрон. текстовые дан. — Режим доступа: https://docs.docker.com/compose/

17. GitLab CI/CD [Электронный ресурс] /. — Электрон. текстовые дан. — Режим доступа: https://docs.gitlab.com/ee/ci/

18. GitLab Runner Docs [Электронный ресурс] /. — Электрон. текстовые дан. — Режим доступа: https://docs.gitlab.com/runner/