Конкурс исследовательских и проектных работ школьников «Высший пилотаж»

Влиянии CI/CD (Continuous Integration, Continuous Delivery — непрерывная интеграция и доставка) методологии на разработку программного обеспечения.

Исследовательская работа
Направление «Computer science»

Автор: ***Шарабарин Михаил,*** Лингвистическая Гимназия "БЭСТ",
11 класс физико-географического направления,
Казахстан г. Петропавловск,

г. Петропавловск

ОГЛАВЛЕНИЕ

Ключевые слова

CI/CD, pipeline, Continuous Deployment, Continuous Integration, Continuous Delivery, Agile, GitLab, Jenkins, SDLC, модели разработки ПО, методологии.

ВВЕДЕНИЕ

В реальной жизни деплой приложения для разработчика становится очень проблематичной задачей, особенно для начинающих разработчиков, которые не имеют никакого отношения к администрированию операционных систем.
Раньше разработчики не задумывались о необходимости создания инструментов для автоматизации, потому что потоки данных были не очень большими и производительность серверов была не столь высока. Разработчики просто копировали все файлы со своего компьютера на удаленный сервер по FTP (англ. **File Transfer Protocol** — протокол передачи файлов по сети), но перед этим им приходилось как-то обмениваться кодом со всей командой, ждать пока тестировщики все проверят и заказчик одобрит проект, а в случае неудачи все начинается с начала. Это было весьма неудобно и времязатратно, поэтому были созданы инструменты для автоматизации этих рутинных задач. Теперь разработчикам остается только писать код, заливать их в свою ветку и ждать результатов.
Это послужило началом для создания инструментов автоматизации ПО
Автоматизация разработки нужна для автоматического развертывания, тестирования и деплоя приложения. То есть каждый этап разработки кроме написания кода в идеале должен происходить автоматически.

Сегодня мы рассмотрим методологию CI/CD, как прототип SDLC (System/**Software Development Life Cycle** - Жизненный цикл разработки ПО). Смотрите Рис. 1

Рис. 1 SDLC - Жизненный цикл разработки ПО

I. Плюсы и минусы использования автоматизации ПО

Автоматическое развертывание имеет множество **плюсов**:

1. разработчики могут сконцентрироваться на написании кода и не отвлекаться на сборку и развертывание проекта
2. все этапы разработки кроме написания кода происходят автоматически
3. отпадает необходимость в большом количестве devops-ов.
4. время до деплоя значительно сокращается
5. появляется версионирование[1]
6. быстрый откат, в случае неполадок
7. меньший шанс прохождения ошибки в продакшн[2]
8. возможность настройки тестового и реального сервера одновременно, без надобности создания отдельных конфигов (не считая переменных среды)
9. быстрая интеграция сторонних библиотек в проект
10. динамическое проведение тестов при сборке

[1] Версионирование - нумерация версий программного обеспечения, процесс присвоения уникальных имен версий

[2] Продакшн — сервер, который работает для конечных пользователей.

11. можно настроить лишь один раз и будет работать всегда

Минусы:

1. переход на автоматизацию занимание довольно много времени
2. необходимость переобучения сотрудников
3. зависимость от стороннего софта

Одним из самых популярных подходов к автоматизации является ci-cd. Данных подход является подмножеством agile архитектуры и широко используется всеми большими компаниями мира, таким как google, AWS и vkontakte.

II. Подготовка к пониманию темы

Для понимания работы CI/CD мы должны рассмотреть основные модели разработки ПО (далее “Модель”) и методологии, которые используются в разработки ПО. В будущем, это даст нам возможность сравнить эти модели между собой и выделить из них наиболее подходящую под конкретную ситуацию. В современной разработке выбор модели - очень важный этап ведения бизнеса. Однако стоит помнить, что модели рассматриваются больше как образ мышления, а не структура создания приложения.

a. МОДЕЛИ разработки ПО:

1) Основой для заложения всех моделей является **Водопадная модель** разработки ПО У. У. Ройсома 1970 года. Суть этой модели в том, что каждый этап разработки начинается только после завершения предыдущего . Она была популярна почти до 2000 годов,

2) **Гибкая методика разработки** (англ. *agile software development, agile-разработка*) — обобщающий термин для целого ряда подходов и практик. Она пришла на смену Водопадной модели в начале 21 века и занимает лидирующие позиции по сей день. Суть методики: данная методика подразумевает постоянное взаимодействие заказчика с группой разработчика. Самая главная цель - выпустить минимально жизнеспособный продукт. Этот подход очень эффективен и продуктивен по сравнению с другими и часто используется в компаниях-гигантах.

3) **RAD-модель** — разновидность инкрементной модели. Эта модель подразумевает максимально быструю разработку ПО. Обычно проект состоит из независимых модулей, которые разрабатываются и внедряются параллельно. Однако каждый этап разработки

подразумевает использование состояния прошлого этапа. Сборка приложения должна проводиться в автоматическом режиме.
4) **Итеративная или итерационная модель**. Подходит для очень длительных и больших проектов. Важной частью этой модели является четкое понимание конечной цели. Т.е. не обязательно сразу давать готовый продукт, а можно представлять его маленькими функциональными частями, но при этом стремиться к достижению поставленной цели и усовершенствовать продукт с каждым новым релизом.
5) **«Incremental Model» (инкрементная модель)** подразумевает постепенное внедрение модулей. Эта модель очень похожа на модель водопада, но проект не является одним целым, а состоит из независимых модулей, дополняющих базовый функционал.

Мы перечислили основные и самые популярные модели ПО. Помимо перечисленных существует еще огромное количество моделей разработки, таких как **спиральная**, **v-образная**, однако они не нашли широкого применения и считаются специфичными.

b. МЕТОДОЛОГИИ :

Если модель описывает, какие стадии жизненного цикла оно проходит и что происходит на каждой из них, то методологии описывают методы управления разработкой.

Самый вариативной моделью является **agile-mode**l. Она подразделяется на такие методологии как:

1) **scrum** - методология которая подразумевает поэтапное выполнение задач с помощью “спринтов”. Спринт - это какой-то промежуток времени, за который команда должна справиться с поставленной задачей. В свою очередь, команда состоит из разработчиков, дизайнеров, тестировщиков и других людей необходимых для полной независимости. Таких команд обычно несколько и каждая занимается своей задачей. Заказчик или scrum-мастер создает список задач (бэклог), в котором создает и расставляет приоритеты для задач. Когда спринт заканчивает команда переходит к выполнению следующей задачи, независимо от результата. Это помогает выпустить полностью функциональный продукт в указанный срок. Самым важным показателем является скорость работы команды. В ходе спринта команды стремятся избегать изменений в прогнозах спринта: изменения приведут к неверным выводам относительно оценки задач

2) **kanban** - это методология подразумевающая выполнение задач точно в срок. Данный подход является очень популярным и интересным. Вся методология завязана на доске задач (бэклоге). Бэклог - это место где заказчик или team-lead размещает задачи для выполнения и цель каждого разработчика уменьшить количество этих задач. Самым важным показателем является продолжительность цикла. Стоит отметить, что изменения к требованиям продукта могут происходить в любое время и это никак не повлияет на разработку.
3) **CI/CD** - это комбинация непрерывной интеграции и непрерывного развертывания программного обеспечения в процессе разработки. Подразумевает полностью автоматизированный процесс разработки. Методология строится на доверии между заказчиком и командой разработки. Основное правило методологии: делать изменения кода как можно чаще. Сегодня мы остановимся на этой методологии, потому что к моменту написания она является самой популярной, удобной и эффективной для ведения бизнеса.

III. Что такое CI/CD и его актуальность

Для начала выясним что такое ci и cd и для кого оно нужно:

Как уже было сказано, данная методология относится к **agile** модели и значит наследует её образ мышления. Это означает что проект может меняться в процессе разработки и быстро адаптироваться к новым условиям.

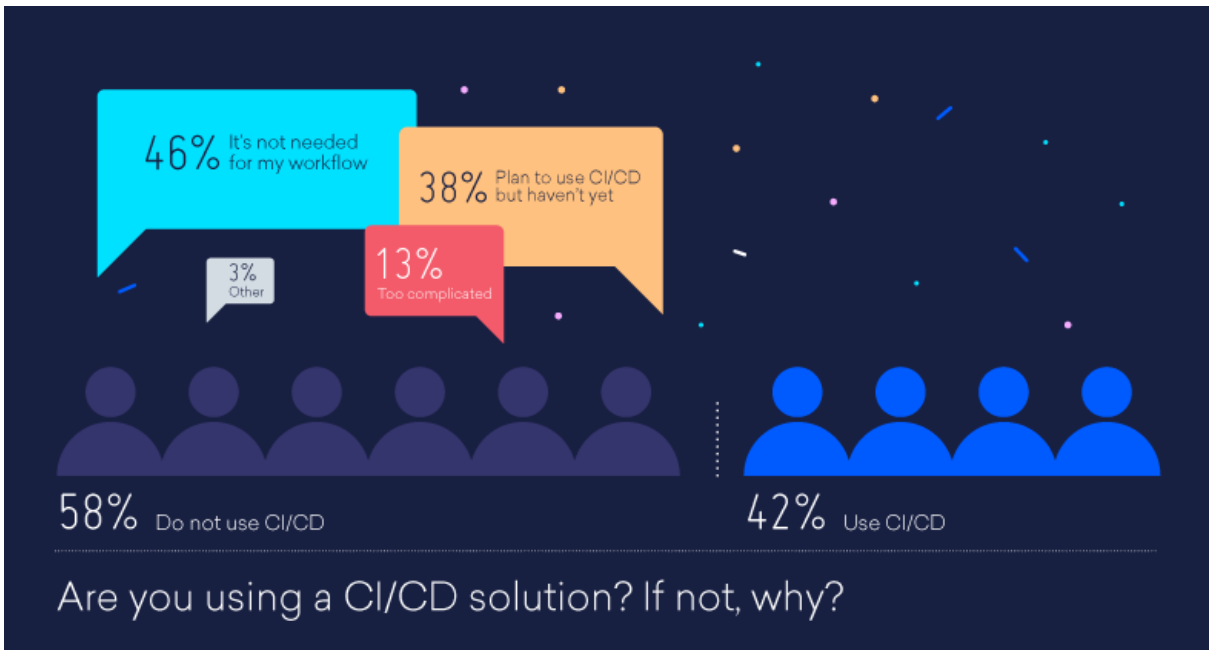

Рис.2 Статистика использования CI/CD

В декабре 2017 года компания Digital Ocean провела опрос Currents среди разработчиков (и не только) по использованию ci cd .Около 58 % не используют CI/CD. Среди них 46 % считают, что потребности в CI/CD для их рабочего процесса нет (т.е. около 27 % от общего числа участников опроса), а 38 % — собираются внедрить CI/CD.

По результатам опроса было выявлено что самым популярным инструментом ci cd является Jenkins 44%, второе место занимает GitLab 39% и третье Travis 26% .

IV. Популярные инструменты CI/CD

При выборе инструмента CI/CD компании часто обращают выбор на скорость разработки и бюджет. Для примера сравним два самым популярных инструмента **Jenkins** и **GitLab CI/CD** и выявим для них сильные и слабые стороны:

Jenkins:

Во-первых Jenkins это спонсорский проект и его нужно разворачивать самостоятельно, это дает более гибкую настройку во время установки. У него можно выделить несколько сильных сторон:

- развитая система управления учетными данными
- огромное количество плагинов
- интеграция со сторонними сервисами
- поддержка разных языков программирования
- Настройка ранеров очень гибкая и довольно простая.

Минусы:

- при использовании плагинов, часто приходится изучать совершенно новую документацию, нет централизованных команд для выполнения задач
- настройка занимает много времени

GitLab CI/CD:

Это очень популярный инструмент разработки. Он поставляется вместе с gitLab, следовательно его не нужно устанавливать отдельно. Рассмотрим плюсы использования:

- возможность параллельного выполнения ранеров
- быстрая интеграция со сторонними сервисами

- надежная система безопасности
- полный контроль над gitHub репозиторием
- простота настройки (имеется графический интерфейс)

Минусы:

- Нельзя протестировать результаты объединения веток до их фактического объединения
- При описании стадий CI/CD-конвейера в них пока нельзя выделять отдельные этапы

Подводя итог, можно выделить несколько критериев для выбора инструмента CI/CD:

Когда стоит выбрать **GitLab CI/CD**:

- Если проект небольшой. Такой проект будет быстрее и удобнее настроить
- Если вам нужно выполнять несколько ранеров в одновременно
- Если вы в большей степени зависите от своего gitHub репозитория
- Если вам нужно готовое решение без самостоятельной установки

Когда стоит выбрать **Jenkins**:

- Вам нужна простота в развертывании кода
- Вы нуждаетесь в большом количестве плагинов
- У вас сложный проект состоящий из множества частей

Стоит отметить, что оба инструмента поддерживают javascript, контроль качества кода, создание ci-cd конвейеров.

V. Структура CI/CD

a. Этапы разработки программного обеспечения:

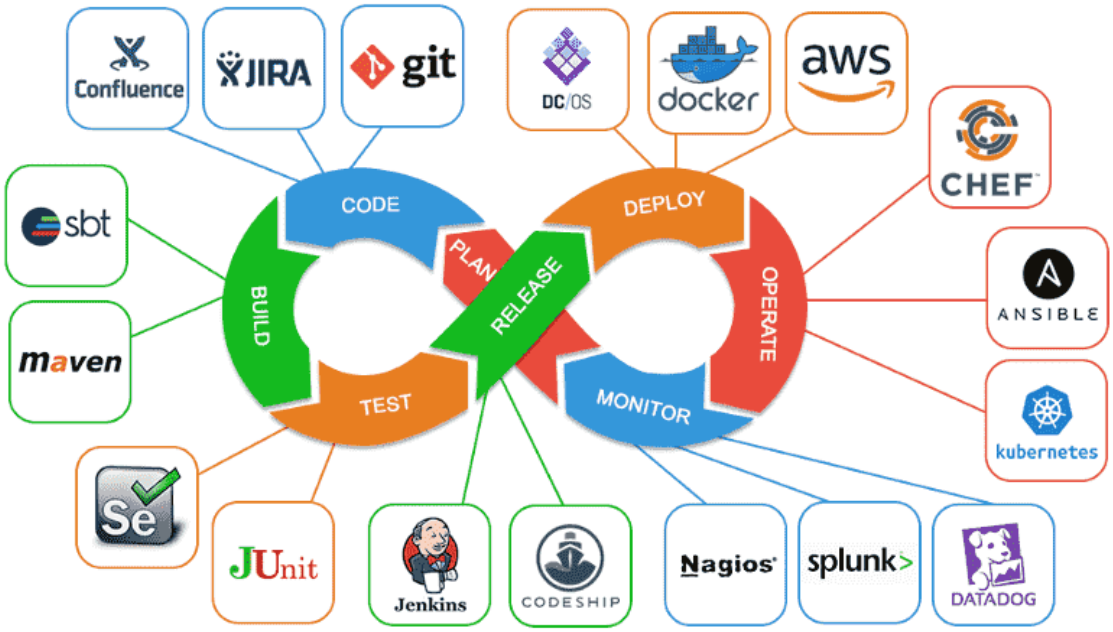

Рис. 3 CI/CD цикл

Структура CI/CD является циклом (см Рис 3), который обычно изображают в виде петли, означающей бесконечно повторяющиеся операции. Она является прототипом SDLS, о котором говорилось в начале.

1. **План**: на этом этапе зарождается идея проекта и ее главная цель, что заказчик хочет видеть в проекте, а так же обсуждается бюджет и процесс разработки
2. **Код**: Разработчики приступают к написанию кода и постепенному внедрению его на платформу (например gitHub, JIra)
3. **Сборка**: Это процесс автоматизации сборки проектов на основе описания их структуры в файлах. (например: mave, webpack)
4. **Тест**: самая важная часть, в которой проверяется качество, стиль кода, а так же ошибки в нем. Существует много инструментов тестирования (JTEST, Se)
5. **Релиз**: Обсуждение выполненных задач с заказчиком и командой, подключение всех метрик и сторонних сервисов. Подготовка к выпуску проекта в продакшн
6. **Развертывание**: выгрузка проекта на сервер (docker, aws)
7. **Управление**: помогает соединить все части проекта в одно целое и дает возможность управлять всем одновременно (Kubernetes, Chef)
8. **Мониторинг**: выявление влияния нового релиза на бизнес. Сбор статистики по посещаемости (в случае если это web-сервис), улучшению пользовательского опыта и другим параметрам, а также анализ этих данных. (Яндекс метрики, splunk)

После появления новых задач, команда программистов снова начинает писать код и процесс повторяется.

b. Процессы разработки программного обеспечения

CI (непрерывная интеграция) - подразумевает постоянное обновление кода, т.е. разработка ведется динамически. Максимальная эффективность этого подхода будет достигаться, когда разработчики будут чаще делать изменения своего кода и сливать их с другими ветками, минимум 1 раз в день. Это очень удобно когда в команде много задач, и проект очень большой. Если рассматривать непрерывную интеграцию как отдельную часть, то она будет выглядеть следующим образом (смотреть Рис.4):

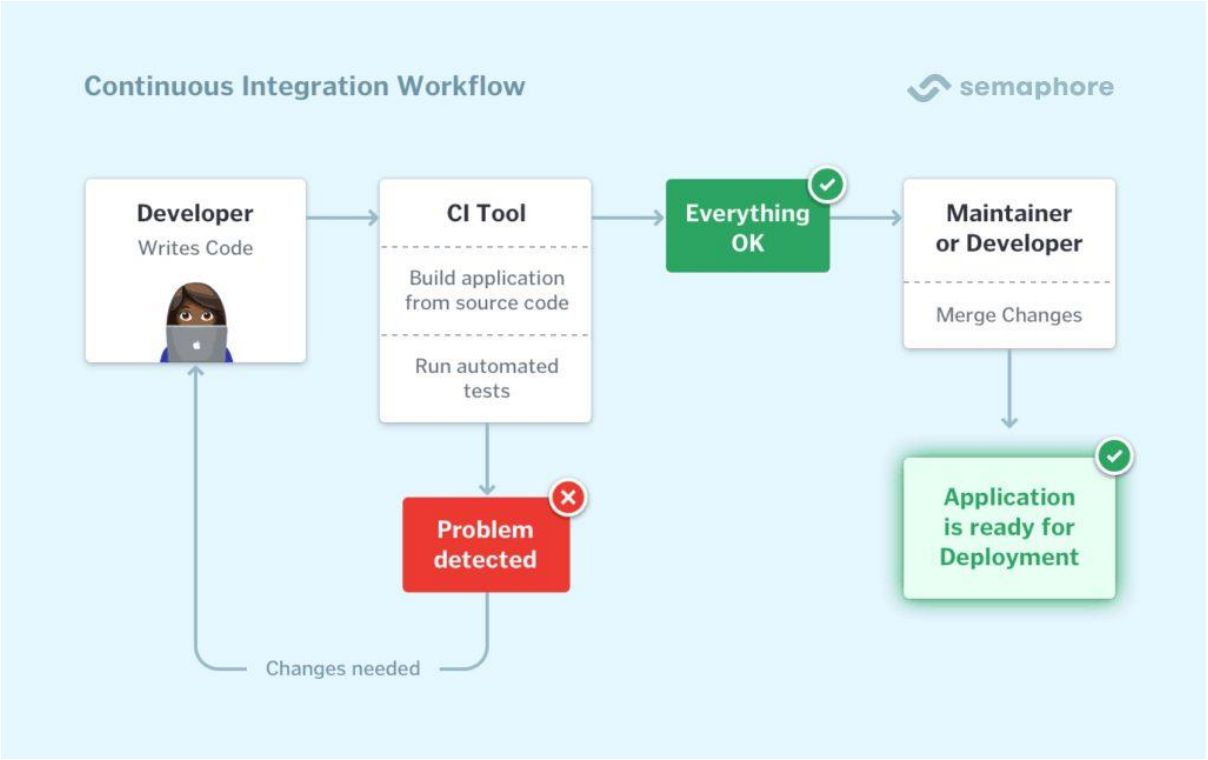

Рис. 4 Continuous Integration рабочий процесс

Сначала разработчики пишут код, потом заливают его куда то, например в свою ветку gitHub. Там идет автоматическая сборка проекта из исходного кода, далее идут автоматические тесты и если все эти операции прошли успешно, тот проект передается дальше на развертывание, смерживание[3] с основной веткой, программистам или куда-угодно. На этом этап непрерывной

[3] смерживание (от слова merge) - слияние веток кода в системах управления версиями

интеграции заканчивается. Однако в случае ошибки при сборке или тестах, код не будет никуда отправляться и разработчик будет заниматься отладкой, а после процесс будет повторяться.

CD: На самом деле под **CD** могут подразумевать либо **continuous delivery**, либо **continuous deployment**.

Continuous delivery (непрерывная доставка) - описывает процесс от начала разработки до успешного проведения тестов. Следует сразу после **CI**. Этот этап включает развертывание результатов на тестовый сервер (если такой имеется). К этому этапу весь проект должен быть готов к публикации. Отличительной особенностью этого этапа, является то, что развертывание проекта на продакшн должно происходить только после подтверждения человеком. Т.е. весь этап разработки должен быть автоматическим, кроме начала выгрузки на сервер. Для примера мы можем рассмотреть пример созданный с помощью gitHub actions (см Рис. 5):

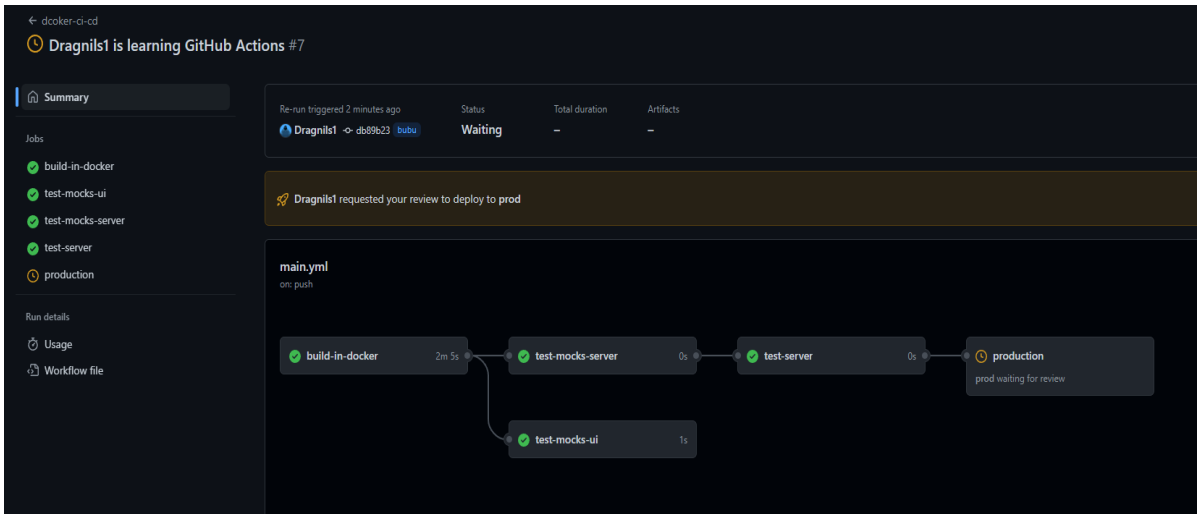

Рис. 5 GitHub Actions pipeline

Так будет выглядеть конвейер в приложении, которое мы создадим позже. Он дает наглядное представление о структуре проекта и этапах разработки. После подтверждения начнется автоматический процесс деплоя на сервер.

Continuous deployment (непрерывное развертывание) - тоже самое, что и **continuous delivery**, но только весь процесс разработки происходит автоматически, включая развертывание на сервер.

Каждая компания стремится к полной автоматизации всех процессов разработки, но это требует очень больших материальных вложений в разработку и других факторов:

1. тестирование должно покрывать более 90 процентов кода
2. поддержка версионирования
3. четкая организация работы в команде
4. минимум один высококвалифицированный разработчик на команду
5. очень быстрая и отзывчивая работа разработчиков.

После достижения этих целей вы получите:

1. заметное снижение затрат на тестирование. **CI** сервер теперь сможет быстро выполнять огромное количество тестов. (например около 100-200 в секунду)
2. возможность чаще выпускать релизы, тем самым уменьшая конкуренцию
3. исключение человеческого фактора и снижение рисков возможных багов
4. быстрая отзывчивость под интересы клиентов (в течении недели, а не года как это обычно бывает)
5. снижается нагрузка на DevOps-ов

c. Обобщение полученных знаний за 4 главу

Установим связь между этапами и процессами разработки с помощью Рис. 6 :

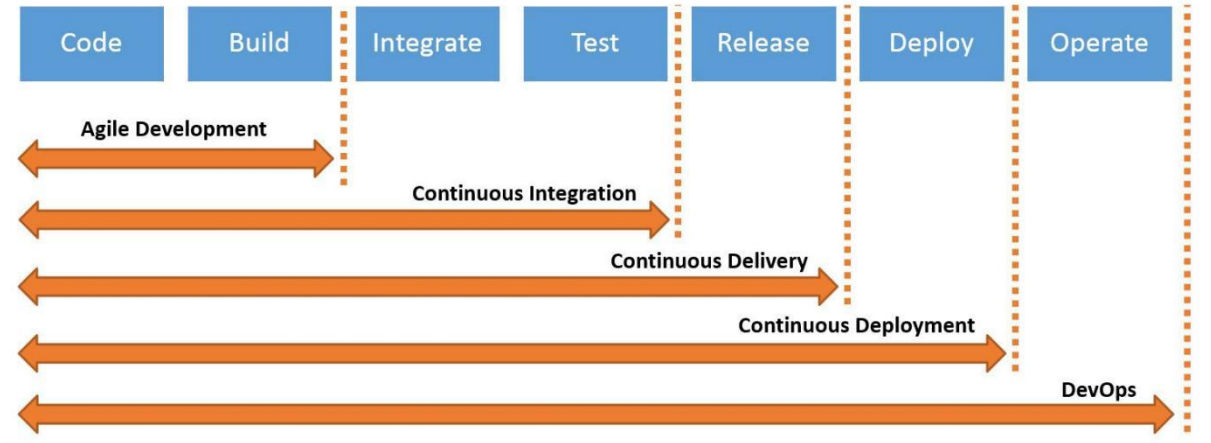

Рис. 6

Таким образом мы смогли объединить этапы и процессы разработки ПО и понять структуру **CI/CD методологии**.

VI. CI/CD pipeline

CI/CD pipeline - это конвейер непрерывной интеграции и развертывания в котором описан порядок действий реализации проекта (см. пример на Рис. 7).

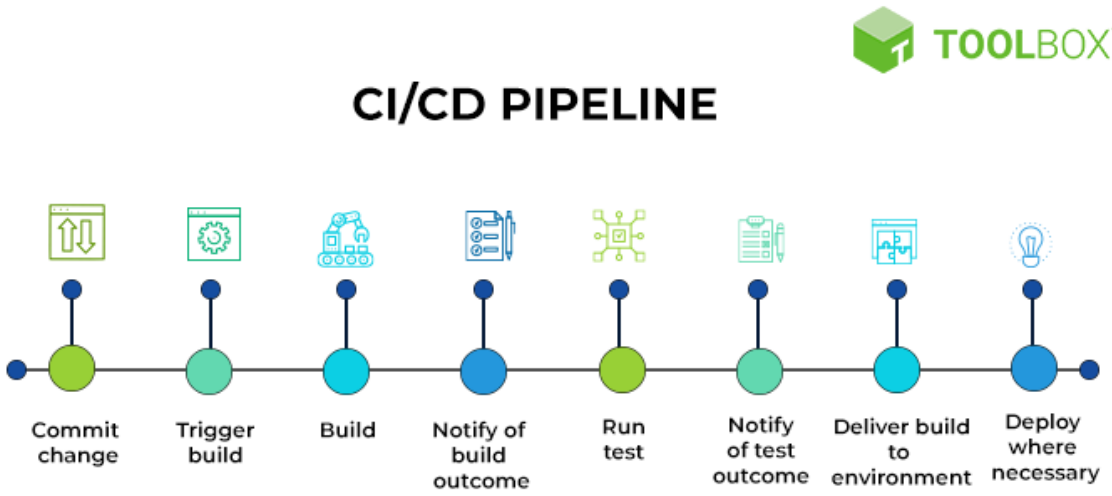

Рис. 7

Он автоматизируют процессы в жизненном цикле разработки программного обеспечения (**SDLC**), осуществляя плавную интеграцию и поставку новых функций. Мы говорили об этом во “вступлении”. **Pipeline** дает наглядное представление как работает проект. Он хорошо подходит для разделения этапов на еще более мелкие подзадачи. Мы уже видели примеры **CI/CD pipeline**, когда описывали структуру **ci-cd методологии** (Рис. 3), а также при описании процесса непрерывной доставки (Рис. 5).

VII. Сравнение CI/CD с другими методологиями

Agile архитектура уже давно превзошла старую каскадную модель и показала, что имеет весомое место в современном мире (см Рис 8).

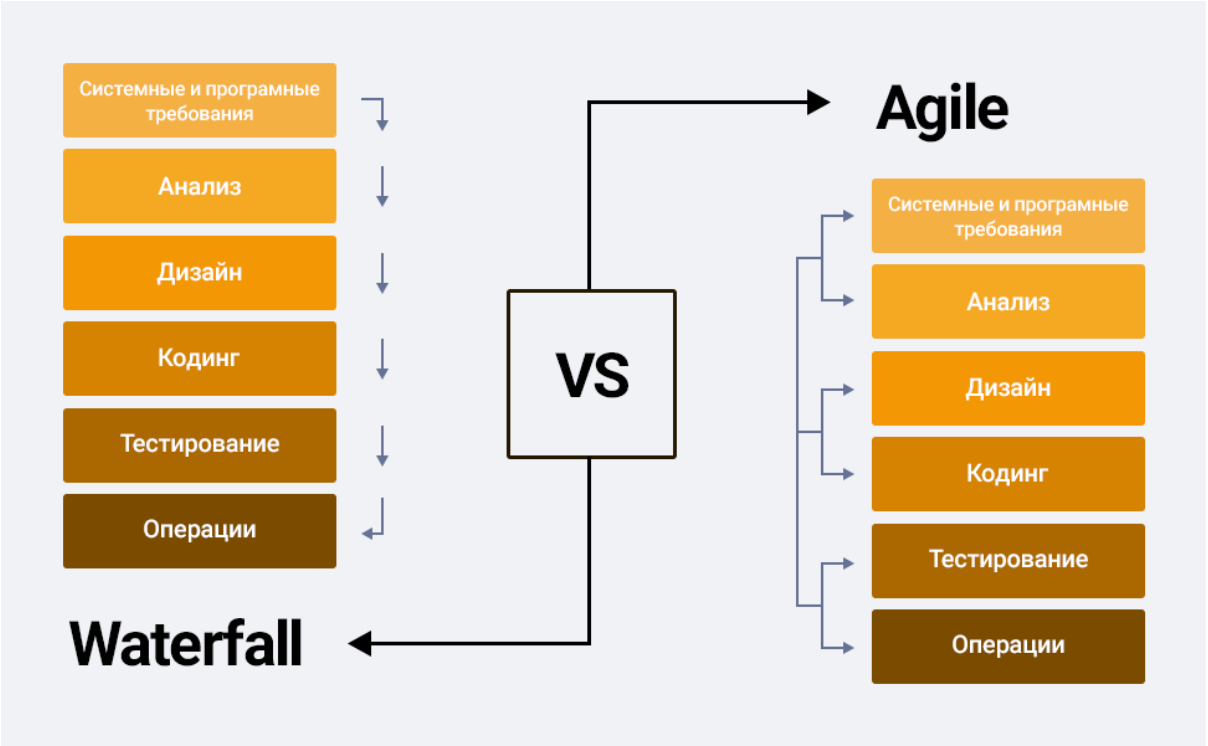

Рис. 8

По опросу **tadviser**[4] к 2021, около 60% компаний используют **agile** в своих проектах. Не смотря на это, требования к проекту постоянно меняются и разработчикам важно быстро реагировать на эти изменения (см Рис 9).

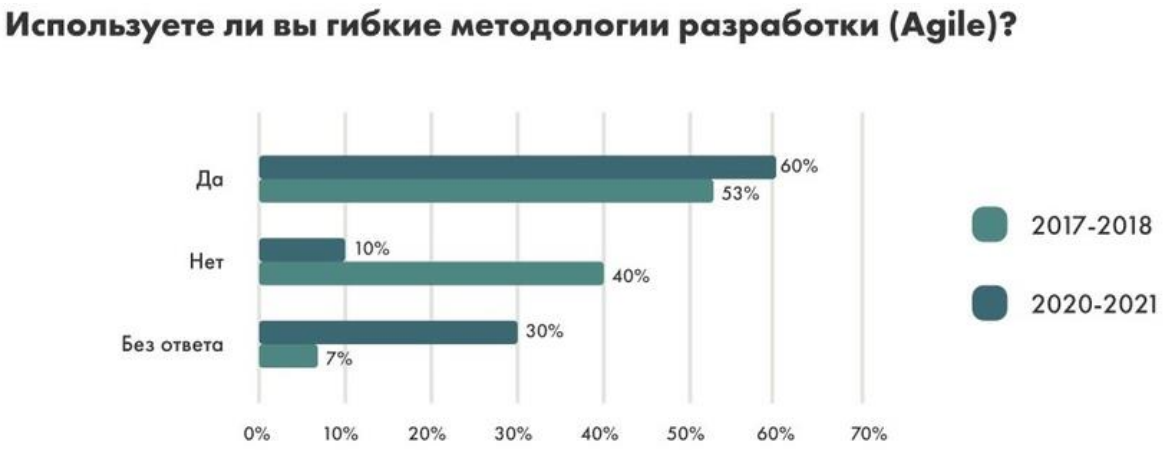

Рис. 9

Опрос проводился от малых до больших по количеству работников компаний. Следовательно можно сделать вывод, что **CI/CD** подходит как для больший компаний-гигантов, так и для начинающих стартапов. Численность работников компании вы можете видеть на рисунке представленном ниже (см Рис 10):

[4] tadvisor - портал выбора технологий и поставщиков

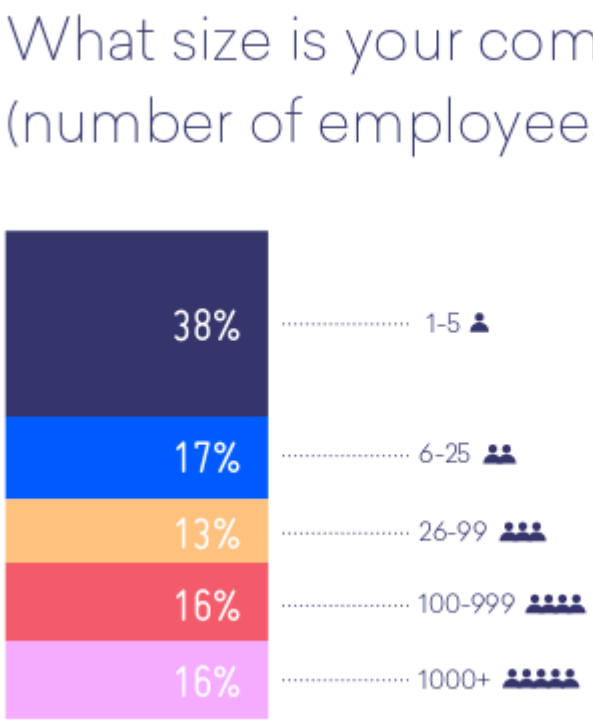

Рис. 10

Стоит отметить, что некоторые разработчики не выделяют **CI/CD** как отдельную методологию **agile**, а считают её практикой **dev-ops**.

По сравнению с другими методологиями, **CI/CD** имеет следующие преимущества:

1. **Конвейеры CI/CD**: автоматизация снижает затраты
2. Обеспечивает прекрасное качество кода
3. Улучшенное среднее время до разрешения (**MTTR**)
4. Сокращение некритических дефектов в бэклоге
5. Локализация отказов

VIII. Пример создания тестового CI/CD приложения

Примечания к проекту:

*Исходный код проекта вы можете посмотреть на моем **gitHub:** https://github.com/Dragnils1/ci-cd-docker-actions*

В данном примере не будут объясняться подробности написания кода, акцент будет сделан на ***gitHub actions****, для наглядного примера* ***CI/CD pipeline***

Этапы описания будут соответствовать ***CI/CD конвейеру****.*

1) План

Стек приложения: react + nodeJs + docker + github actions.

Описание: Данное приложение возвращает рандомное число с сервера при увеличении/уменьшении счетчика

Цель: показать процесс прохождения **CI/CD pipeline**

Структура проекта (См. Рис. 11):

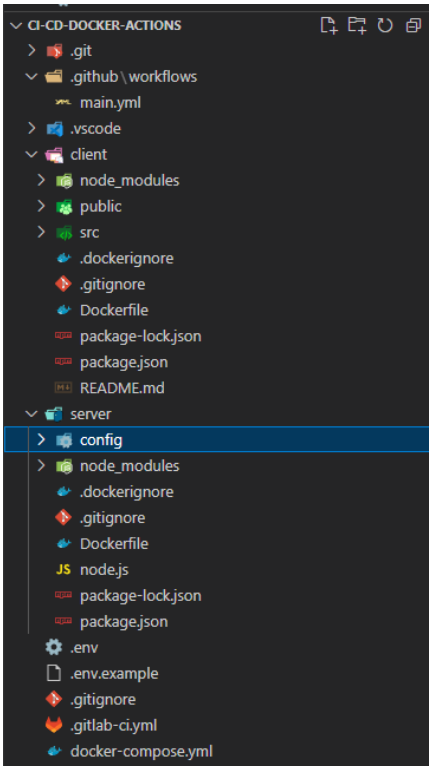

Рис. 11

2) Код:

Здесь мы подробно рассмотрим только создание **CI/CD pipeline** с помощью actions.yml файла.

1. Напишем моковый сервер, клиент
2. Все действия будут наглядно показаны в этапе сборки. Создадим main.yml в workflow директории. Определим этапы работы нашего приложения (jobs). Jobs указывает стадии нашего проекта. Для краткости определим в них сборку, тестирование, выгрузку на тестовый сервер и ожидание выгрузки на продакшн сервер. Запишем зависимости этапов: т.е. тестирование будет дожидаться выполнения сборки с помощью тега needs и так далее по цепочке. Настроим чтобы сборка начиналась при каждом push-е в bubu ветку. Для наглядности запросим подтверждение

перед загрузкой на сервер с помощью переменной среды 'prod', это называется **continuous delivery**, потому что процесс разработки не полностью автоматически, а требует минимального вмешательства человека (см Рис. 12).

```yaml
# action.yml

name: dcoker-ci-cd
run-name: ${{ github.actor }} is learning GitHub Actions
on:
  push:
    branches: [ "bubu" ]
jobs:

  build-in-docker:
    runs-on: ubuntu-latest
    env:
      MYSQLDB_USER: 'root'
      MYSQLDB_DATABASE: 'db'
      MYSQLDB_ROOT_PASSWORD: ${{secrets.MYSQLDB_ROOT_PASSWORD}}
      MYSQLDB_LOCAL_PORT: '3307'
      MYSQLDB_DOCKER_PORT: '3306'
      NODE_LOCAL_PORT: '8080'
      NODE_DOCKER_PORT: '8080'
    steps:
      - uses: actions/checkout@v3
      - uses: isbang/compose-action@v1.4.1
        with:
          compose-file: "./docker-compose.yml"

  test-mocks-ui:
    needs: build-in-docker
    runs-on: ubuntu-latest
    steps:
      - name: add some mock tests
        run: echo '\033[32m All test on client are success \033[0m'

  test-mocks-server:
    needs: build-in-docker
    runs-on: ubuntu-latest
    steps:
      - name: add some mock tests
        run: echo '\033[32m All test on server are success \033[0m'

  test-server:
    needs: test-mocks-server
    runs-on: ubuntu-latest
    steps:
      - name: deploy on test server
        run: |
          echo '\033[32m deployment on mock server success \033[0m'
          echo 'wait confirmation'

  production:
    needs: test-server
    runs-on: ubuntu-latest
    timeout-minutes: 5
    environment: 'prod'
    steps:
      - name: deployment
        run: echo '\033[32m deployment success \033[0m'
```

Рис. 12

3. Для удобства поместим это все в docker контейнер
4. Создадим dockerfile на клиенте и сервере.
5. Объединим все в docker-compose и добавим СУБД (Система управления базами данных) - mysql. Здесь же добавим тома и пробросим порты.

В итоге контейнер должен выглядеть примерно так (см Рис. 13):

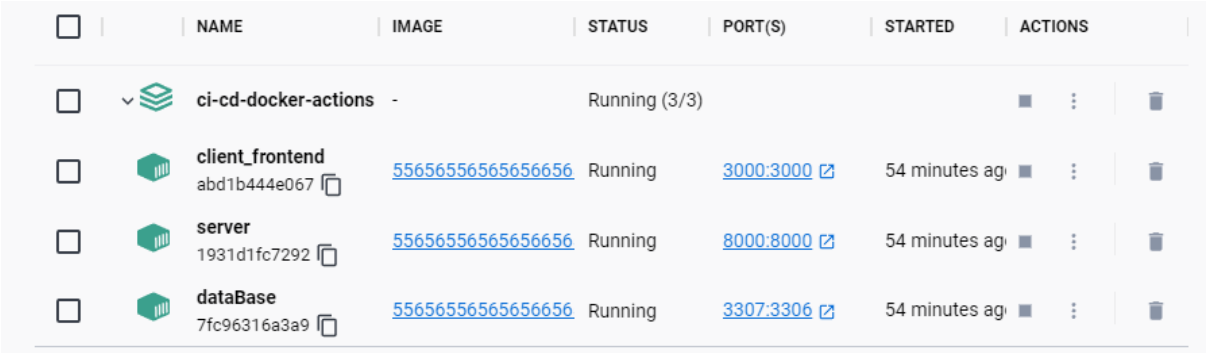

Рис. 13

6. Добавим images в Docker Hub: https://hub.docker.com/repository/docker/5565655656565665655/ci-cd (в данном руководстве мы сделаем это механически на этапе разработки кода для облегчения сложности)

7. Зальем код на gitHub: https://github.com/Dragnils1/ci-cd-docker-actions

3,4) Этап сборки и тестирования будет проходить автоматически поэтому мы понаблюдаем, как это работает в **gitHub actions**

1. На рисунке мы видим **CI/CD pipeline**, созданный с помощью **gitHub Actions**.

2. Запустим сборку проекта, это самая продолжительная часть конвейера. (см Рис. 14)

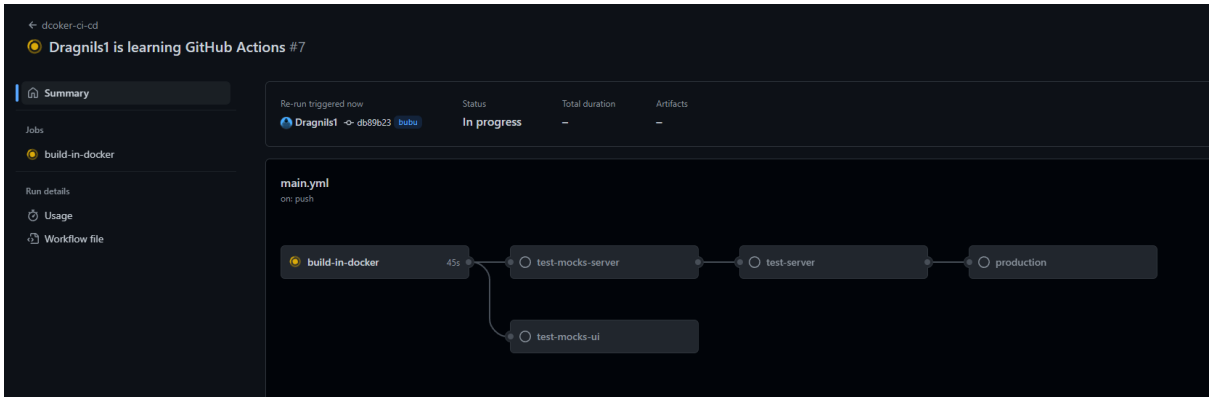

Рис. 14

3. После выполнения сборки запускаются тесты, причем ui-mock-test и server-mock-test выполняются параллельно. Далее идет выгрузка проекта на тестовый сервер, не дожидаясь окончания UI тестов. Это было сделано для примера, чтобы показать как может работать конвейер. (см. Рис. 15)

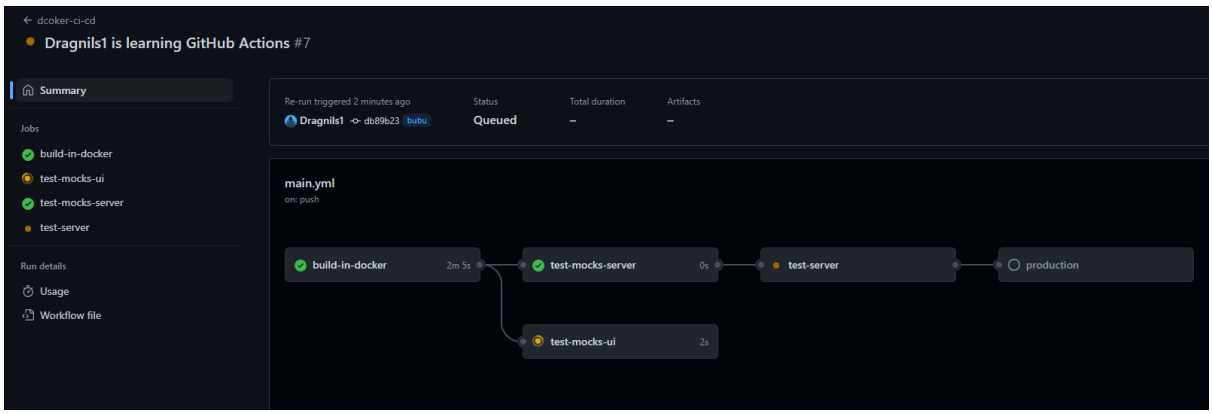

Рис. 15

5) Этап релиза наступает когда продукт выложен на тестовый сервер и у группы тестировщиков вместе с заказчиком есть возможность проверить работоспособность проекта и найти возможные баги. (см. Рис. 16)

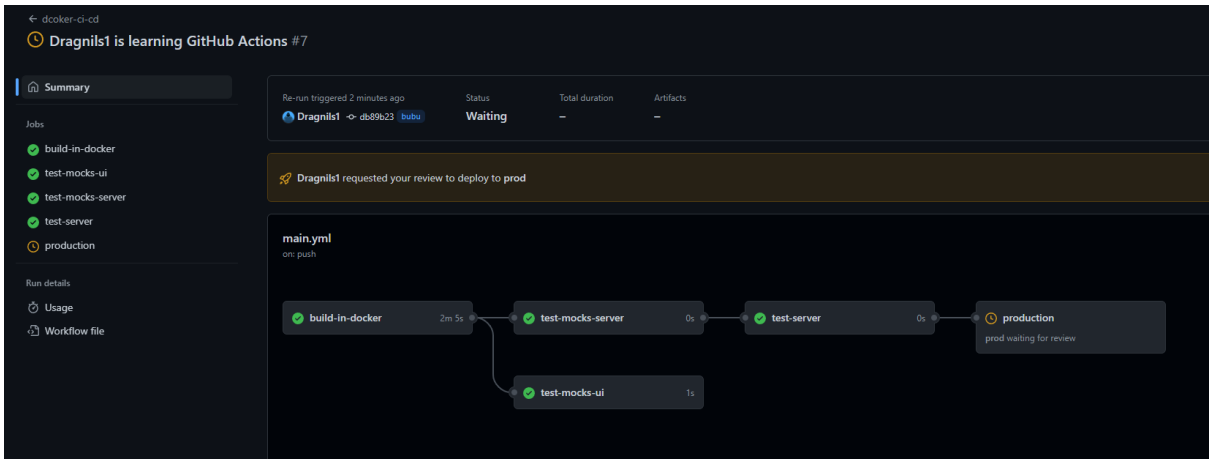

Рис.16

Теперь когда мы убедились что все тесты были пройдены, мы можем применить изменения (см. Рис. 17)

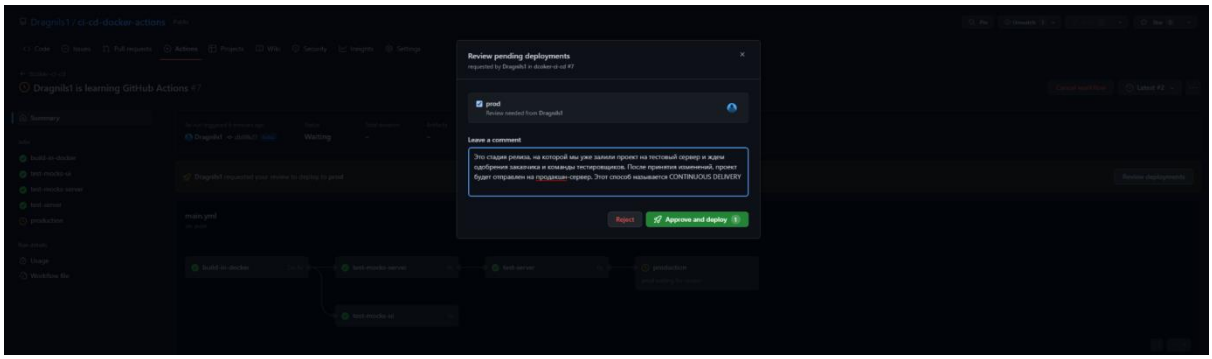

Рис. 17

6) этап деплоя мы сымитировали ручной загрузкой проекта в docker-hub

7) управление проекта происходило с помощью docker-compose на этапе написания кода. Он объединяет все образы (images) в один контейнер и задает конфигурацию проекта. В крупных проектах для этого в основном используется kubernetes.
8) Мониторинг - мы пропустим данный этап, по понятным причинам.

В итоге мы имеем полностью работающее приложение, включающее все этапы конвейера (см. Рис. 18)

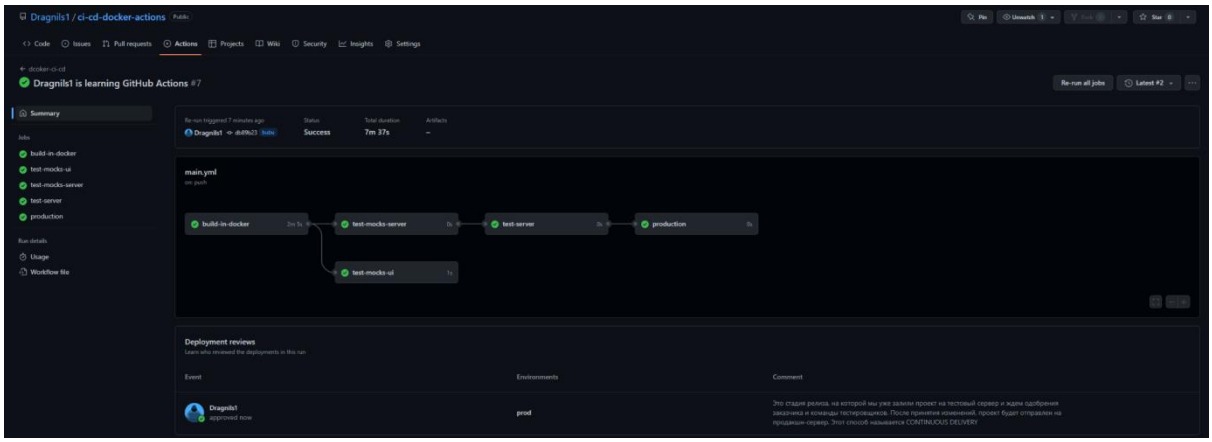

Рис. 18

Быстро рассмотрим развертывание того же приложение в **gitLab**, только уже в комбинации с **continuous deployment**: https://gitlab.com/Dragnils1/ci-cd-docker-actions

Вместо сборки проекта в докер-контейнере мы будем эмулировать его загрузку.
Для публикации проекта на **gitLab** мы можем использовать тот же репозиторий gitHub, только нам нужно написать файл .gitlab-ci.yml в котором определены стадии сборки проекта (stages).(см. Рис 19)

```yaml
stages:          # List of stages for jobs, and their order of execution
  - build
  - test
  - deploy

build-job:       # This job runs in the build stage, which runs first.
  stage: build
  script:
    - echo "Compiling the code..."
    - echo "It is only test code, beside free repo uses"
    - echo "Compile complete."

unit-test-job:   # This job runs in the test stage.
  stage: test    # It only starts when the job in the build stage completes successfully.
  script:
    - echo "Running unit tests... This will take about 60 seconds."
    - echo "Code coverage is 90%"

lint-test-job:   # This job also runs in the test stage.
  stage: test    # It can run at the same time as unit-test-job (in parallel).
  script:
    - echo "Linting code... This will take about 10 seconds."
    - sleep 10
    - echo "No lint issues found."

deploy-job:      # This job runs in the deploy stage.
  stage: deploy  # It only runs when *both* jobs in the test stage complete successfully.
  environment: production
  script:
    - echo "Deploying application..."
    - echo "Application successfully deployed."
```

Рис. 19

Настройка проекта очень проста и занимает в разы меньше времени чем другие инструменты **CI/CD**. После выполнения конвейера, мы что все прошло успешно и автоматически выложено на сервер, это называется **continuous deployment**. (см. Рис. 20)

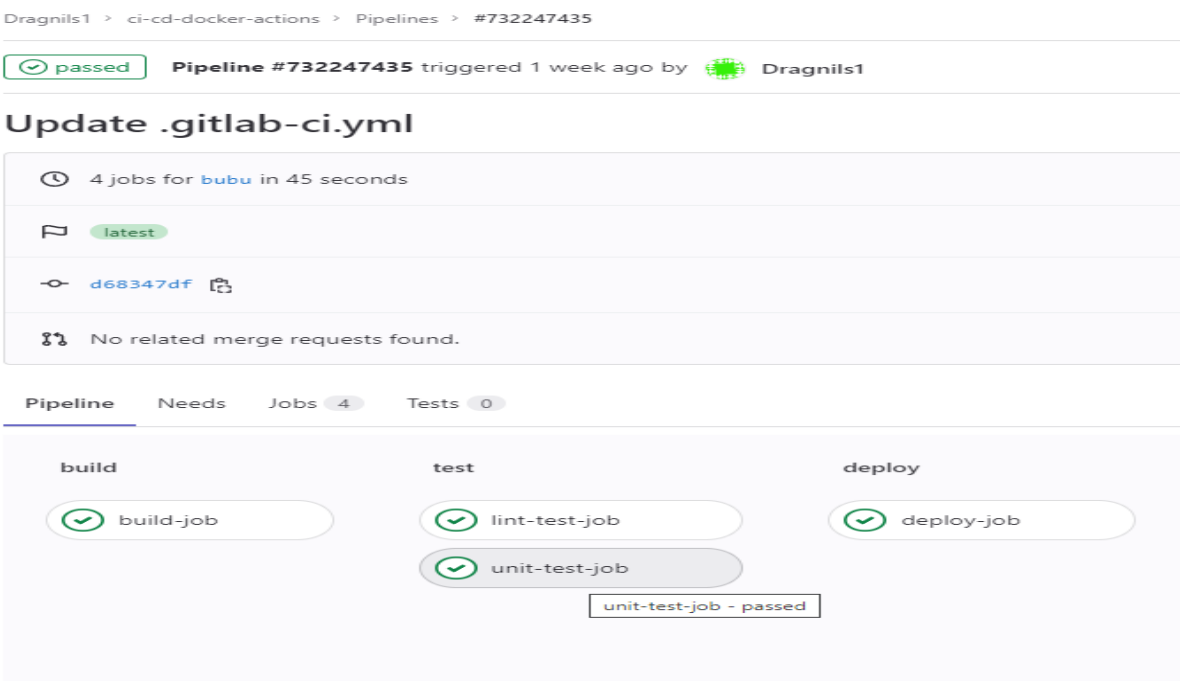

Рис. 20

ЗАКЛЮЧЕНИЕ

Исследование помогает компаниям выбрать подходящую модель разработки ПО. Определиться с методологий и выбрать инструменты для полного цикла разработки ПО. Это дает общее понимание для начинающих разработчиков, как работает **CI/CD pipeline**, что это такое и для чего он нужен

СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ

1. А. А. Вичугова. ПРИКЛАДНАЯ ИНФОРМАТИКА [Электронный ресурс]. — URL: https://earchive.tpu.ru/bitstream/11683/37406/1/reprint-nw-15418.pdf
2. Wikipedia, [Электронный ресурс]. — URL: https://ru.wikipedia.org/wiki/%D0%93%D0%B8%D0%B1%D0%BA%D0%B0%D1%8F_%D0%BC%D0%B5%D1%82%D0%BE%D0%B4%D0%BE%D0%BB%D0%BE%D0%B3%D0%B8%D1%8F_%D1%80%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B8
3. Комания Edison, [Электронный ресурс]. — URL: https://habr.com/ru/company/edison/blog/269789/
4. Wikipedia, [Электронный ресурс]. — URL: https://ru.wikipedia.org/wiki/CI/CD
5. Стен Питтен, Atlassian - [Электронный ресурс]. — URL: https://www.atlassian.com/ru/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment
6. Digitalocean [Электронный ресурс]. — URL: https://assets.digitalocean.com/currents-report/DigitalOcean-Currents-Q4-2017.pdf
7. Компания «Флант» — специалисты по Kubernetes и DevOps [Электронный ресурс]. — URL: https://habr.com/ru/company/flant/blog/346418/