Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 22.Б10-мм

Разработка инструментов для портирования проектов и CI/CD из GitHub в GitLab

Позиев Алексей Владимирович

Отчёт по учебной практике
в форме «Производственное задание»

Научный руководитель:
старший преподаватель кафедры СП, к.т.н., Ю. В. Литвинов

Консультант:
технический руководитель команды разработки «Mindbox» М. А. Маркелов

Санкт-Петербург
2025

Оглавление

Введение

Активное использование систем контроля версий и платформ для совместной разработки программного обеспечения, таких как GitHub, GitLab, BitBucket, стало неотъемлемой частью современных процессов создания программных продуктов. Эти платформы позволяют хранить исходный код проектов, управлять задачами и организовывать процессы непрерывной интеграции и доставки (CI/CD). Однако изменение политических и экономических условий может привести к ситуации, когда использование коммерческих облачных решений, например, GitHub, становится неудобным или невозможным, что вынуждает компании переходить на самостоятельно развертываемые решения. В связи с этим команда ООО «Mindbox» столкнулась с задачей переноса всех репозиториев организации и их процессов CI/CD из GitHub в GitLab.

Сложность подобного переноса определяется не только различиями в интерфейсах программирования (API) двух систем, но и тем, что во многих репозиториях организации процессы CI/CD содержат множество индивидуальных доработок, отражающих специфику рабочих процессов, присущих командам, ответственным за них. В частности, в компании «Mindbox» трудность усугублялась необходимостью учета интеграций на уровне микросервисов и сторонних сервисов, неявно использующих репозитории GitHub, а также большим количеством проектов — более 800, каждый из которых может иметь уникальные настройки процессов CI/CD. При этом переход должен быть плавным для других разработчиков, без глобальных сбоев в процессе разработки.

Поскольку синтаксис и функциональные возможности, предоставляемые GitLab и GitHub для настройки процессов CI/CD, существенно отличаются, первоначальной задачей стало исследование возможности полноценного переноса этих процессов без потери функциональности и формирование понимания конечной структуры и способов реализации. Учитывая большое количество репозиториев в компании, возникла дополнительная потребность в возможности обобщения процессов CI/CD и их централизованного редактирования.

В связи с ранее упомянутым значительным количеством проектов следующим необходимым этапом является создание автоматизированного способа переноса репозиториев, реализуемого в виде консольного приложения, и написание документации, необходимой для быстрой адаптации новых разработчиков.

Кроме того, важной частью миграции является выявление и адаптация существующего кода, а также сервисов, зависимых от GitHub, для успешного перехода на новую систему контроля версий.

Таким образом, данная работа посвящена исследованию возможности и поиску оптимального способа перевода процессов CI/CD в рамках компании, а также разработке инструментов для переноса проектов и их процессов CI/CD из GitHub в GitLab.

1. Постановка задачи

Целью работы является исследование возможностей и поиск оптимальных способов перевода CI/CD в рамках компании, а также разработка инструментов для переноса проектов и их процессов CI/CD из GitHub в GitLab. Для достижения цели требовалось решить следующие задачи:

1. Провести анализ существующих решений и подходов для переноса процессов CI/CD между системами контроля версий.
2. Исследовать возможность полноценного перевода конфигураций GitHub Actions в GitLab CI/CD
    - Разработать обработчик конфигурационных файлов GitHub Actions.
    - Создать доменную модель, описывающую структуру конфигураций GitHub Actions, используемых в компании.
    - Провести анализ аналогов параметров и функциональностей GitHub Actions в GitLab CI/CD, а также определить обходные пути для параметров, не имеющих прямых аналогов.
3. Сформировать ключевые подходы создания конвйеров проектов.
4. Адаптировать существующий механизм автоматизации для работы с GitLab вместо GitHub.
5. Создать консольное приложение для автоматического переноса репозиториев из GitHub в GitLab.
6. Провести апробацию разработанных решений на реальных проектах компании.

2. Обзор

2.1. Обзор используемых технологий

- Bash [1] — командная оболочка для UNIX-подобных операционных систем. На ней написана значительная часть логики, используемой в процессах CI/CD.

- C# [2] — основной язык программирования в компании. В рамках задач текущий работы использовался для: мигратора репозиториев, утилиты уведомлений о деплое.

- Docker [3] — платформа для разработки, доставки и запуска приложений в контейнерах. Используется для установки различных утилит в рамках процессов CI/CD, а также для развертывания приложений с помощью Octopus Deploy.

- Octopus Deploy [8] — система автоматизации развертывания приложений, которая упрощает управление выпусками и обеспечивает их доставку в различные окружения.

- GitHub Actions [4] — платформа для автоматизации процессов непрерывной интеграции и доставки, позволяющая автоматизировать сборку, тестирование и развертывание приложений непосредственно в репозиториях GitHub.

- GitLab CI/CD [5] — встроенная в платформу GitLab система для автоматизации процессов непрерывной интеграции и развертывания.

2.2. Существующие решения

Поиск существующих решений проводился в целях нахождения инструмента для автоматического или полуавтоматического переноса конфигураций процессов CI/CD из GitHub Actions в GitLab CI/CD. В

результате анализа не было обнаружено готовых решений, соответствующих требованиям компании. Далее представлены ключевые причины и ограничения, объясняющие отсутствие подходящих инструментов.

2.2.1. Ограниченность стандартных инструментов самого сервиса

Встроенная механика импортирования GitLab позволяет переносить содержимое репозиториев, включая ветки, историю коммитов и частично настройки самих репозиториев, но имеет ряд ограничений:

- Не поддерживает автоматическое преобразование конфигураций CI/CD;
- Не сохраняет метаданные проектов (настройки доступа, правила слияния, способ слияния, опции обязательного успешного прохождения конвейера и так далее).

Это делает его применимым только для базового копирования кодовой базы, что не подходит для наших требований. Однако в дальнейшем его можно использовать как дополнительный инструмент.

2.2.2. Отсутствие сторонних решений

gh-actions-importer [13] — единственный публичный и поддерживаемый инструмент, работающий с GitHub и GitLab. Он способен частично транслировать конфигурации конвейеров, но, к сожалению, только в одном направлении (GitLab CI/CD $\rightarrow$ GitHub Actions), что не соответствует нашим требованиям. Кроме того, код логики инструмента недоступен, что делает его непригодным для адаптации под нужды компании.

2.2.3. Причины отсутствия инструментов

Поиск и анализ решений и историй о переездах между GitHub и GitLab позволил выделить три ключевые причины их возможного отсутствия:

1. Массовая миграция между GitHub и GitLab обычно вызвана внешними факторами: санкции, смена стратегии компании. Это не формирует устойчивого спроса на универсальные решения. При этом сам сервис GitLab не предоставляет инструмента для миграции CI/CD, что усложняет задачу.

2. Конфигурации CI/CD в компаниях часто существенно различаются и могут использовать разные подходы, механики и параметры, а также сторонние интеграции. Создание универсального инструмента, подходящего для нескольких корпораций, крайне затруднительно.

3. После успешного перехода компании не заинтересованы в поддержке инструмента в силу единоразовости перехода. Это подтверждается отсутствием открытых решений от других компаний.

3. Реализация

3.1. Реализация обработки конфигураций GitHub Actions и разработка доменной модели

Файлы конфигурации GitHub Actions представляют собой YAML-файлы, поэтому для их обработки использовалась библиотека YamlDotNet [11]. Однако в процессе работы возникла неожиданная проблема: некоторые параметры в GitHub Actions могут принимать значения разных типов. Например, параметр «on», обозначающий событие, на которое должен срабатывать процесс CI/CD, может быть как одиночным значением (например, строкой), так и списком значений или даже сложной структурой данных (например, словарем с дополнительными условиями).

Стандартный механизм библиотеки YamlDotNet не поддерживает такие сценарии, так как он требует строгой типизации для каждого поля. Для решения этой проблемы была использована библиотека OneOf [9], которая позволяет описать поле, способное принимать одно из нескольких возможных типов. Чтобы научить механизм преобразования данных (десериализатор) обрабатывать такие поля, был разработан и подключен специальный конвертер. Этот конвертер управляет процессом преобразования данных между YAML и объектами в коде. Однако у этого подхода есть ограничение: если в YAML-файле встречаются два значения одинакового типа (например, два одиночных значения или два списка), механизм не может однозначно определить, какой тип использовать. Это может привести к некорректному результату. На практике такие случаи не встречались, поэтому ограничение не повлияло на выполнение задачи.

Для создания доменной модели параметров конфигурации GitHub Actions, используемых в компании, была разработана небольшая консольная программа для локального клонирования и синхронизации репозиториев GitHub. Доменная модель разрабатывалась итерационно: при неудачной попытке десериализовать конфигурацию одного из репозито-

риев из-за неизвестного параметра модель расширялась. В результате была получена доменная модель, описывающая все используемые в компании параметры конфигурации GitHub Actions.

Используя полученную доменную модель, были выявлены параметры и функциональности GitHub Actions, для которых в GitLab отсутствуют аналоги или они существенно отличаются:

- Запуск процессов CI/CD по расписанию в формате cron. В GitHub это делается прямо в конфигурации, а в GitLab необходимо настраивать расписание либо через сетевые запросы к сервису, либо через пользовательский интерфейс.

- Матрицы позволяют запускать несколько аналогичных конвейеров с разными параметрами. В GitLab CI/CD отсутствует возможность создания динамических матриц, поддерживаются только статические. Однако для данной проблемы было найдено обходное решение, которое будет описано далее.

- Шаги с выполнением кода или вызовом действий (actions). В GitHub Actions шаги могут либо выполнять определенный код, либо вызывать отдельное, заранее написанное действие. Все шаги выполняются в одном контексте, что позволяет использовать результаты предыдущих шагов в текущих. Это упрощает многие сценарии, например, сборку проекта или образа с последующей загрузкой результата на удаленный сервер. В GitLab недавно появился аналог этой функциональности, но он пока недостаточно развит. Его ограничения, как и в случае с динамическими матрицами, будут рассмотрены далее.

В результате анализа проблемных мест GitLab и поиска обходных решений была подтверждена возможность выполнения миграции без потери функциональности.

3.2. Формирование ключевых подходов создания конвейеров

3.2.1. Недостатки шагов в GitLab

Основным фундаментальным отличием процессов конвейеров GitHub от GitLab является использование шагов вместо задач (jobs). Задачи, в отличие от шагов, не сохраняют контекст выполнения, что в некоторых ситуациях требует его передачи между задачами, что замедляет конвейер из-за дополнительных сетевых взаимодействий. Поэтому структуру шагов хотелось бы сохранить. В GitLab, как ранее было сказано, появился их аналог. Однако на данный момент он является экспериментальным и имеет ряд проблем:

- Это нововведение находится на стадии разработки, и некоторые параметры могут существенно измениться. В нашем случае это требует заморозки текущей версии исполнителя шагов, чтобы избежать неожиданных поломок. Однако при обновлении версии могут возникнуть ломающие изменения, что потребует значительной переработки кода;

- Не поддерживается условное выполнение шага. Шаг запустится, даже если предыдущие шаги завершились с ошибкой. Чтобы избежать этого, необходимо явно прописывать условия в коде шага, что непрактично для общих шагов, используемых в нескольких репозиториях. Было найдено решение в виде промежуточного шага, который принимает на вход данные о шаге, условие запуска и с помощью образа исполнителя шагов рекурсивно запускает или пропускает нужный шаг. Однако из-за рекурсивности это решение значительно замедляет работу при большом количестве вложенных условий;

- Отсутствие выражений, аналогичных GitHub, для написания базовых условных конструкций, включая методы, такие как поиск элемента в массиве. Это приводит к тому, что даже если шаг успеш-

но выполнился, необходимо вручную проверять условия внутри самого шага, что непрактично для переиспользуемых действий;

- Невозможно разрешить определенному шагу завершиться с ошибкой без остановки всего конвейера.

В связи с этими проблемами было решено отказаться от использования шагов в GitLab.

3.2.2. Замена динамических матриц

Решением проблемы стало использование нисходящих конвейеров (downstream pipelines). Этот механизм позволяет запускать дочерний конвейер из основного, передавая ему текст конфигурации. Таким образом, удалось воспроизвести динамическую матрицу в GitLab по следующему принципу: родительский конвейер получает динамические данные, например, путь к папке по её названию, запрашивает статическую конфигурацию дочернего конвейера у GitLab, после чего с помощью утилиты «envsubst» заменяет статические переменные на динамические данные и запускает дочерний конвейер с нужными параметрами.

3.2.3. Компоненты

В итоге основой конфигураций конвейеров были выбраны задачи (jobs). Как уже упоминалось, контекст между задачами не передается, а передача образов или собранных C# проектов через артефакты занимает слишком много времени. Поэтому все места, где требуется передача больших объемов данных, были переделаны в один большой скрипт.

Для всех задач есть набор часто используемых утилит, таких как jq, curl, git, которые необходимы почти в каждом конвейере. Они были добавлены в отдельный образ Docker, который используется в задачах по мере необходимости. Поскольку слои образов кешируются в исполнителях CI/CD, скачивание образов происходит быстрее, чем их ручная установка в каждой задаче.

Для переиспользования логики конвейеров, например, создания новой версии сервиса или библиотеки, используются компоненты. У компонента есть входные параметры, и вся их логика заключается в подстановке этих параметров в код компонента и копировании кода компонента в конфигурацию конвейера. По внутренней реализации компоненты делятся на два типа:

1. Явные — автономные компоненты, которые содержат полноценную логику и начинают работу сразу после подключения. Они предназначены для замены шаблонных файлов конфигурации конвейеров GitHub, которые можно было переиспользовать, задавая нужные параметры.

2. Неявные — компоненты, которые обычно содержат одну скрытую задачу (job), то есть при выполнении конвейера она не запускается, так как их название начинается с точки. Они используются как шаблонные задачи, а не полноценные конвейеры. Их параметры, включая переменные и правила вызова, можно переопределять с помощью параметра расширения (extends). Это позволяет использовать их гибко, так как они содержат основную логику, но вторичные параметры можно переопределить в месте их использования. Необходимость в них возникла из-за потребности в установке параметров задачи (job) на основе результатов предыдущей задачи, что невозможно с компонентами, так как они статические.

```yaml
spec:
  inputs:
    stage:
    release-number:
      default: "1.0.0"

---

.prepare-release-helper:
  variables:
    RELEASE_NUMBER: $[[ inputs.release-number ]]
  script:
    - |
      # creating release number

.create-gitlab-release:
  stage: $[[ inputs.stage ]]
  tags:
    - yandex
    - tiny
  image: registry.com/system/images/docker-dind:latest
  variables: !reference [.prepare-release-helper, variables]
  before_script:
    # копирует код скрипта из вспомогательной джобы
    - !reference [.prepare-release-helper, script]
  script:
    - |
      # creating release, and some write some variables to env file
  artifacts:
    reports:
      dotenv:
        - release_additional_outputs.env
```

Листинг 1: Пример неявного компонента

```
stages:
  - test-release

include:
  - component: link-to-component/create-gitlab-release.yml
    inputs:

create-gitlab-release:
  rules:
    - if: $CI_COMMIT_BRANCH == "main"
  extends:
    - .create-gitlab-release
  stage: test-release
```

Листинг 2: Пример использования неявного компонента

3.2.4. Принцип запуска конвейера

В отличие от GitHub, где конфигурация пайплайна может быть распределена по нескольким файлам, в GitLab начало работы конвейера определяется централизованно в файле «.gitlab-ci.yml». Чтобы сохранить структуру, привычную разработчикам (где для каждого триггера или функциональности используется отдельный файл конфигурации), было принято решение создавать аналогичные отдельные файлы, но вызывать их в GitLab с помощью механизма нисходящего пайплайна (downstream pipeline). Это означает, что в файле .gitlab-ci.yml для каждого события создается отдельная задача, которая запускает соответствующий конвейер при срабатывании триггера.

Данный подход, в отличие от простого подключения компонентов, позволяет избежать коллизий переменных и проблем с правилами запуска. Кроме того, он позволяет использовать группы ресурсов, чтобы предотвратить одновременный деплой нескольких версий. Для проверки эффективности этого подхода были проведены эксперименты по сравнению скорости работы двух подходов (централизованного и распределенного).

Разница в скорости оказалась незначительной или вовсе отсутствовала. Для проверки пригодности данной структуры был переведен рабочий микросервис с GitHub на GitLab (Рис. 1).

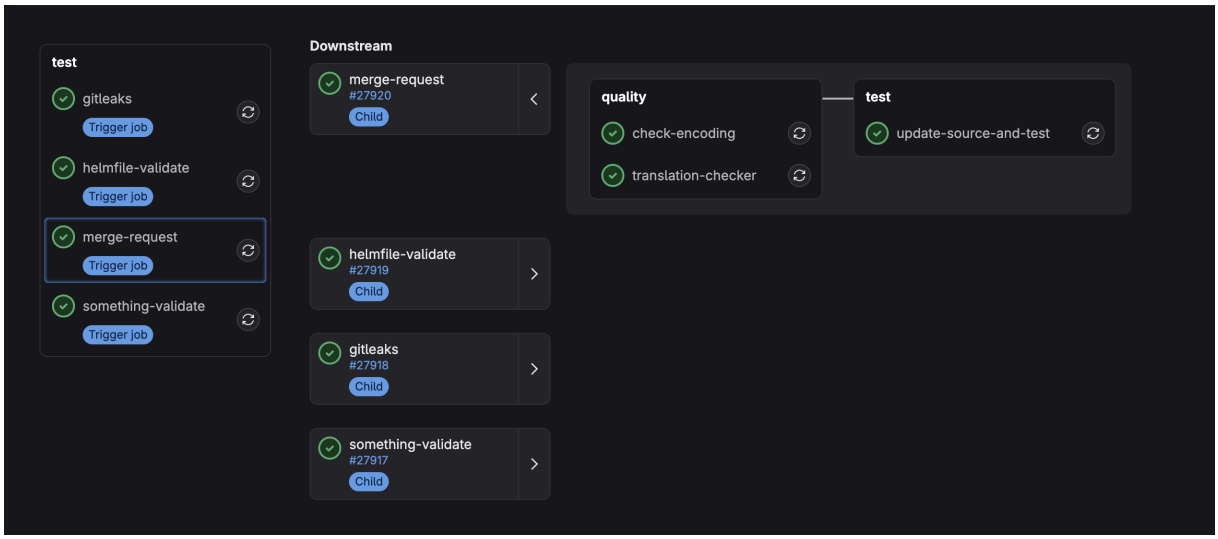

Рис. 1: Пример графика работы конвейера

3.2.5. Обеспечение правильной последовательности запуска конвейеров

Во время непосредственной работы с GitLab была найдена разница в последовательньсти выполнения конвейеров при новых изменений проектов. В GitHub выполнялись следующие правила:

- Для запросов на слияние (merge requests): если в нем появляется новое изменения, все предыдущие конвейеры, связанные со старой версией кода данного запроса на слияние должны быть прерваны;
- Для изменений главной ветки проекта: если появляется несколько новых изменений репозитория, уже запущенный конвейер должен завершить свою работу, а следующий конвейер должен быть запущен для последнего изменения. Остальные промежуточные конвейеры отменяются.

Для реализации этих правил в GitLab используются следующие механизмы:

- Параметр конфигурации конвейера автоматической отмены (auto_cancel) , которая, логично, позволяет автоматически отменять конвейеры при определенных условиях. Например, при появлении нового изменения в запросе на слияние все предыдущие конвейеры, связанные с этим запросом, автоматически прерываются;

- Настройка проекта режима обработки конвейеров (process_mode), доступная только через сетевой интерфейс приложения GitLab. Она определяет последовательность выполнения конвейеров в очереди, обеспечивая корректную обработку новых коммитов в главной ветке проекта;

- Использование групп ресурсов в конфигурации конвейеров, позволяющие ограничить количество одновременно выполняемых конвейеров одного вида при новых изменениях проекта;

- Отмена промежуточных конвейеров происходит засчет настройки репозитория в GitLab, позволяющей отменять все конвейры, созданные до последнего успешного конвейера.

Как итог, в GitLab получилось прийти к аналогичному GitHub поведению последовательности выполнения конвейеров.

3.3. Адаптация автоматизации, уведомляющей об успешном разворачивании новой версии кода в окружении

Сама автоматизация представляет собой шаг в процессе разворачивания проектов в Octopus Deploy, который добавляет комментарии в карточки задач и запросов на слияние кода из GitHub с уведомлением, что изменения, связанные с этими задачами и запросами, развернулись в определенном окружении.
Первоначально реализация представляла собой несколько сотен C# кода, расположенных в шаблоне шагов из Octopus Deploy, использую-

щий устаревший ScriptCS [10]. Данный код не был обнадежен ловлей исключений и различные его ошибки могли привести к провалу всего разворачивания проекта в окружение. А также возникала проблема, что не на всех рабочих машинах Octopus Deploy был установлен C#. Кроме проблем с существующим кодом требовалось добавить новую функциональность уведомлений карточек задач из Yandex Tracker [12] и запросов на слияния в GitLab.
В связи с проблемами выше были сделано следующее:

- Переписан существующий код, разбив его на несколько файлов и классов;
- Добавлена новая функциональность, связанная с GitLab и Yandex Tracker;
- Добавлена глобальная ловля исключений и все сетевые взаимодействия со сторонними сервисами были покрыты механизмами повторений попыток с экспоненциальным ростом ожидания между каждыми попытками для обеспечения стабильной работы шага;
- Код был помещен в образ Docker, который и используется в новом шаблон шага в Octopus Deploy.

После проделанной работы новый шаг был подключен во все 36 репозиториев отвечающих за клиентскую часть сервиса, а также в самый нагруженный по изменениям проект компании. Новых проблем выявлено не было.

3.4. Консольное приложение для автоматического переноса репозиториев из GitHub в GitLab

Консольное приложение, далее именуемое мигратором, представляет собой машину состояний с возможностью запуска нескольких экземпляров одновременно. Это необходимо, так как импорт репозитория в GitLab может занимать значительное время, и этот простой можно компенсировать параллельным запуском мигратора для нескольких репозиториев.

Архитектура приложения достаточно проста (Рис. 2), что соответствует его основной задаче: запускать определенный процессор, обрабатывать ошибки и фиксировать результаты работы.

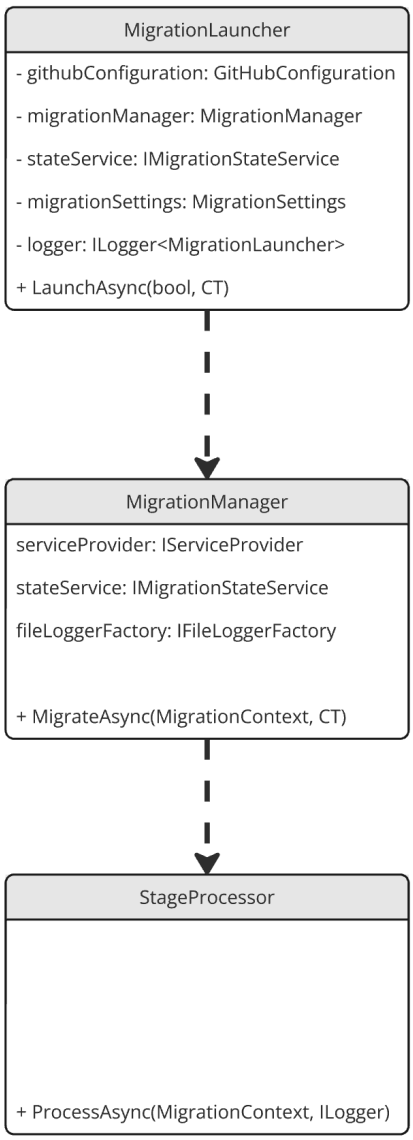

Рис. 2: Архитектура мигратора

Поскольку пользователь может в любой момент завершить работу консольного приложения, необходимо локальное хранилище для сохранения состояния миграции. В нем хранятся: текущее состояние, информация

о пользователе, выполняющем миграцию, системные данные, тип репозитория и другие параметры. В качестве хранилища был выбран YAML-файл, причина была в простоте и отсутствии необходимости более комплексных решений, например, отдельной базе данных. Сам файл содержит всю необходимую информацию о репозиториях (Рис. 3). Для создания и поддержания актуальности этого файла используется отдельная консольная команда, которая взаимодействует с сетевым интерфейсом сервиса GitHub для получения данных о репозиториях и сетевым интерфейсом сервиса Google Sheets для получения информации о принадлежности репозитория определенной команде организации. Актуализация данных происходит автоматически с помощью запланированной задачи, которая периодически запускает команду и фиксирует изменения в репозитории.

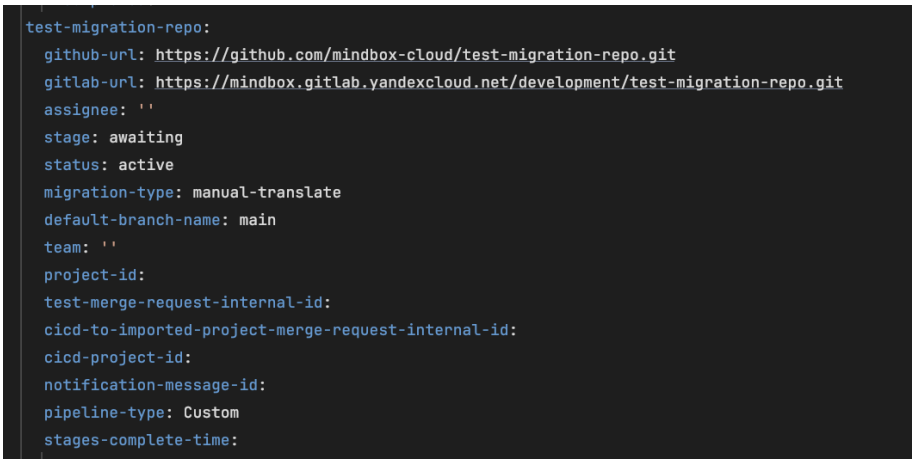

Рис. 3: Пример данных в YAML-файле для репозитория

Принцип работы мигратора следующий: *MigrationLauncher* выбирает репозитории, которые пользователь отметил для переноса, запрашивает данные из общего YAML-файла и создает соответствующее количество экземпляров *MigrationManager*, каждый из которых отвечает за миграцию одного репозитория. *MigrationManager* с помощью *IServiceProvider* находит *IStageProcessor*, соответствующий текущему этапу миграции, и запускает его. После выполнения этапа процессор возвращает следующий шаг или ошибку, которую необходимо обработать и записать.
Для удобства работы с промежуточными записями работы программы

была реализована обертка над *ILogger*, которая перенаправляет записи в отдельный файл. Это решение было принято, так как базовая библиотека записей от Microsoft не поддерживает запись в файл. Хотя GrayLog [6] также поддерживает записи в файл, было решено не подключать эту библиотеку из-за ее избыточности для данной задачи.
Всего в миграторе 30 шагов. Каждый шаг содержит идемпотентную логику, связанную с определенным аспектом переноса проекта с GitHub на GitLab. Это помогает тем, что если работа мигратора прерывается на середине какого-либо шага, то при следующем запуске сторонние эффекты предыдущих запусков не будут накапливаться, и результат будет таким же, как если бы прерывания не было. Это позволяет нескольким разработчикам, даже без глубокого знания GitLab и самого мигратора, быстро и качественно выполнять миграцию. В связи с этим каждый шаг достаточно атомарен и выполняет небольную логику. Каждый шаг реализован как сервис в единственном экземпляре, что позволяет использовать инъекцию зависимостей.
Шаги миграции разделены на три фазы:

1. Фаза переноса конфигураций конвейера в GitLab;
2. Проверка корректности конвейера на уровне запроса на слияние;
3. Полноценный импорт репозитория из GitHub с его архивацией и последующим слиянием перенесенной конфигурации конвейера.

Далее описаны общие шаги мигратора в каждой из фаз.

3.4.1. Фаза портирования конвейера в GitLab и проверка корректности на уровне запроса на слияние

Изначально вторая фаза включала проверку корректности конвейера как на уровне запроса на слияние, так и на уровне создания новой версии сервиса. Однако в текущей реализации проверка корректности конвейера выпуска (release) была перенесена в третью фазу, что сделало вторую фазу менее объемной. Поэтому здесь она рассматривается совместно с первой фазой.

1. Уведомление в мессенджер о начале портирования конвейера репозитория. Канал отправки зависит от указанной команды;

2. Локальное клонирование репозитория с префиксом *cicd и создание ветки для тестирования;

3. Замена источника пакетов NuGet [7] на GitLab в файле конфигурации и всех упоминаний GitHub в YAML и Docker файлах;

4. Опциональная трансляция *скелета* конвейера. Выполняется только при выборе определенного *MigrationType*;

5. Отображение подсказок в консоли, зависящих от типа репозитория. Например, для библиотек может быть предложено увеличить минорную версию, а для сервисов — заменить старое хранилище образов. Подсказки добавляются с помощью *KeyedSingleton*, что позволяет повторно использовать их в разных шагах без нарушения принципа избежаний дубликаций;

6. Автоматическое создание запросов на слияния из ветки для тестирования в основную ветку. Проверяется результат выполнения конвейера. Ожидание завершения конвейера реализовано через возвращение шагом самого себя, что позволяет мигратору периодически проверять статус;

7. При падении конвейера мигратор возвращается к шагу *merge_request_pipeline_changes_required*, чтобы пользователь мог исправить ошибки и повторить проверку. Для этого необходимо внести изменения в ветку для тестирования и перезапустить мигратор. Шаги с таким поведением помечены специальным атрибутом, что позволяет *MigrationManager* завершить работу при переходе на такой шаг.

3.4.2. Полноценный импорт репозитория из GitHub с архивацией и слиянием перенесенной конфигурации конвейера

1. Уведомление в мессенджер о начале импорта репозитория, а в конце работы и о завершении миграции. Канал отправки зависит от указанной команды;

2. Архивирование репозитория в GitHub для предотвращения рассинхронизации с GitLab;

3. Импорт репозитория из GitHub в GitLab и ожидание завершения процесса. Он выполняется с помощью ранеупомянутого встроенного инструмента GitLab ;

4. Шаг с подсказками, аналогичный предыдущему, но с другим набором рекомендаций;

5. Автоматическое обновление настроек проекта в Octopus Deploy, включая изменение различных переменных шагов, например, источник получаемого кода на встроенный вариант вместо GitHub;

6. Создание запроса на слияние в импортированном репозитории на основе ранее протестированной ветки. Поскольку конвейер уже был проверен, он сразу сливается с мигратором, и мигратор отслеживает результат выполнения конвейера выпуска или добавлений изменений в основную ветку;

7. При неудачном выполнении конвейера изменения откатываются, и мигратор возвращается к шагу *release_pipeline_changes_required*. Пользователь должен исправить ошибки в ветке для тестирования и перезапустить мигратор;

8. Перенос настроек, которые не поддерживаются встроенным импортом: настройки репозитория, например: политики слияния, защищенные ветки и множества правил (rulesets), которые отсутствуют в GitLab и требуют ручного сопоставления с доступными

настройками. Также выполняется настройка прав доступа для групп и отдельных пользователей.

9. Разархивация репозитория в GitHub и переключение его в режим чтения для всех пользователей. Необходимо, так как в компании есть автоматизация, основанная на Terraform модуле, которому необходимо, чтобы репозиторий не был заархивирован.

4. Апробация

Апробация выбранных подходов к созданию конфигураций конвейеров и инструмента, созданного для миграции проектов, была проведена в процессе перевода репозиториев компании. На момент 26 февраля 2025 года силами нескольких разработчиков успешно переведено 614 из 850 репозиториев, что составляет более 70% всех проектов компании. Доля задач, связанных с поддержкой работы мигратора, составляет менее 2% от общего числа задач, связанных с переходом на GitLab. Это свидетельствует о высокой надежности разработанного инструмента.

Апробация адаптированного под GitLab механизма уведомлений о развертывании в окружениях проводилась путем подключения соответствующего шага в процессы разворачивания всех микрофронтендов и самого нагруженного сервиса компании. В течение месяца после внедрения изменений проблем обнаружено не было.

Заключение

В рамках данной работы были выполнены следующие задачи:

1. Проведен анализ существующих решений и подходов для переноса процессов CI/CD между системами контроля версий.
2. Исследована возможность полноценного перевода конфигураций GitHub Actions в GitLab CI/CD.
    - Разработан обработчик конфигурационных файлов GitHub Actions.
    - Создана доменная модель, описывающая структуру конфигураций GitHub Actions, используемых в компании.
    - Проведен анализ аналогов параметров и функциональностей GitHub Actions в GitLab CI/CD, а также найдены обходные пути для параметров, не имеющих прямых аналогов.
3. Сформированы ключевые подходы для создания конвейеров проектов.
4. Адаптирован существующий механизм автоматизации для работы с GitLab вместо GitHub.
5. Создано консольное приложение для автоматического переноса репозиториев из GitHub в GitLab.
6. Проведена апробация разработанных решений на реальных проектах компании.

Список литературы

[1] Bash. — URL: https://www.gnu.org/software/bash/ (дата обращения: 12 2024 г.).

[2] C#. — URL: https://dotnet.microsoft.com/en-us/languages/csharp (дата обращения: 12 2024 г.).

[3] Docker. — URL: https://www.docker.com/ (дата обращения: 12 2024 г.).

[4] GitHub Actions. — URL: https://github.com/features/actions (дата обращения: 12 2024 г.).

[5] GitLab CI/CD. — URL: https://docs.gitlab.com/ee/ci/ (дата обращения: 12 2024 г.).

[6] GrayLog. — URL: https://graylog.org/ (дата обращения: 12 2024 г.).

[7] NuGet. — URL: https://www.nuget.org/PACKAGES (дата обращения: 3 февраля 2024 г.).

[8] Octopus Deploy. — URL: https://octopus.com/ (дата обращения: 12 2024 г.).

[9] OneOf. — URL: https://github.com/mcintyre321/OneOf (дата обращения: 12 2024 г.).

[10] ScriptCS. — URL: https://scriptcs.net (дата обращения: 12 2024 г.).

[11] YamlDotNet. — URL: https://aaubry.net/pages/yamldotnet.html (дата обращения: 12 2024 г.).

[12] Yandex Tracker. — URL: https://yandex.cloud/en-ru/services/tracker (дата обращения: 12 2024 г.).

[13] gh-actions-migrator. — URL: https://github.com/github/gh-actions-importer (дата обращения: 3 февраля 2024 г.).