МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Университет «Дубна»

Институт системного анализа и управления

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

Тема: Методика сборки, тестирования и развертывания прикладного ПО для распределенной системы обработки данных эксперимента SPD на коллайдере NICA

Ф.И.О. студента Короткин Ринат Наильевич

Группа 6181 **Направление подготовки** 01.04.02 Прикладная математика и информатика

Направленность (профиль) образовательной программы Математическое моделирование и анализ данных

Выпускающая кафедра распределенных информационно-вычислительных систем

Руководитель работы _____ /доц. Прогулова Т.Б. /

Консультант (ы) _____ / Прокошин Ф.В. ___ /

_____ / Козловский А.А. __ /

_____ /_____/

Рецензент _____ / Олейник Д.А. _____/

Выпускная квалификационная работа допущена к защите «_____»_____20___г.
(дата)

Заведующий кафедрой _____ /_ Кореньков В.В. _/

г. Дубна

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Университет «Дубна»

Институт системного анализа и управления

УТВЕРЖДАЮ
Заведующий кафедрой

_____ /Кореньков В.В. /
(подпись) (Фамилия И. О.)

«_____» _____20___ г.

З а д а н и е
на выпускную квалификационную работу – магистерскую диссертацию

Тема: Методика сборки, тестирования и развертывания прикладного ПО для распределенной системы обработки данных эксперимента SPD на коллайдере NICA

Утверждена приказом № _____ от _____

ФИО студента Короткин Ринат Наильевич

Группа 6181 **Направление подготовки** 01.04.02 Прикладная математика и информатика

Направленность (профиль) образовательной программы Математическое моделирование и анализ данных

Выпускающая кафедра распределенных информационно-вычислительных систем

Дата выдачи задания «___» _____ 20___ г.

Дата завершения
выпускной квалификационной работы «___» _____ 20___ г.

г. Дубна

Исходные данные к работе

Документация по установке SPD, логические и физические схемы системы сбора сырых данных, техническое задание

Результаты работы:

1. Содержание пояснительной записки (перечень рассматриваемых вопросов)
Анализ существующего процесса разработки прикладного ПО для распределенной системы обработки данных эксперимента SPD. Построение моделей AS-IS, TO-BE. Изучение и обзор существующих на данный момент подходов, технических и программных средств реализации. Разработка технических средств, их настройка и методика их использования для автоматизации процесса разработки, тестирования и развертывания. Обсуждение результатов.
2. Перечень демонстрационных листов (материалов)
Презентация Power Point

Консультант(ы) _____ / Прокошин Ф.В. /
_____ / Козловский А.А. /
_____ /_____ /

Руководитель работы _____ /доц. Прогулова Т.Б./

Задание принял к исполнению _____
(дата)
_____
(подпись студента)

Я, Короткин Ринат Наильевич , ознакомлен(а) с требованием об обязательности проверки выпускной квалификационной работы на объем заимствования. Все прямые заимствования из печатных и электронных источников, а также из защищенных ранее выпускных квалификационных работ, научных докладов об основных результатах подготовленной научно-квалификационной работы (диссертации), кандидатских и докторских диссертаций, должны иметь в работе соответствующие ссылки.

Я ознакомлен(а) с Порядком проверки на объем заимствования и размещения в электронно-библиотечной системе текстов выпускных квалификационных работ и научных докладов обучающихся, согласно которому обнаружение тексте выпускной квалификационной работы заимствований, в том числе содержательных, неправомочных заимствований, является основанием для недопуска к защите выпускной квалификационной работы и отчисления из образовательной организации.

_____ /_____/
подпись *Фамилия И.О.*

Аннотация

В данной магистерской диссертации представлена методика автоматизации процессов сборки, тестирования и развертывания прикладного программного обеспечения для распределённой системы обработки данных эксперимента *SPD* на коллайдере *NICA*.

В ходе исследования проведён системный анализ существующего процесса разработки, выявлены его ключевые недостатки. На основе обзора современных практик и технологий разработана комплексная методика, обеспечивающая автоматизированное формирование среды разработки, сборку и тестирование программных модулей, а также публикацию и развертывание образов в контейнерных реестрах.

Практическая реализация методики осуществлена в рамках нескольких проектов коллаборации *SPD*, что позволило существенно сократить время подготовки к выпуску новых версий ПО, повысить качество и воспроизводимость программных продуктов.

Abstract

This master's thesis presents a method for automating the processes of building, testing, and deploying application software for the distributed data processing system of the SPD experiment at the NICA collider.

The study includes a systematic analysis of the existing development process, identifying its key shortcomings. Based on a review of modern practices and technologies, a comprehensive method was developed to enable automated creation of development environments, building and testing software modules, as well as publishing and deploying container images to registries.

The practical implementation of this method was carried out within several projects of the SPD collaboration, resulting in a significant reduction of preparation time for releasing new software versions, and an improvement in the quality and reproducibility of software products.

Оглавление

Введение

Основная цель коллаборации *SPD* [1] - проведение экспериментов по изучению спиновой структуры нуклонов и других процессов, зависящих от спина, путем измерений асимметрий в процессе Дрелла-Яна, процессах рождения J/Ψ частиц и прямых фотонов в столкновениях поляризованных протонов и дейтронов.

Коллаборация в рамках эксперимента насчитывает сотни физиков, инженеров, разработчиков и *IT*-специалистов из разных уголков мира. В задачи разработчиков и *IT*-специалистов входит реализация многоуровневого программного комплекса: от низкоуровневого программирования контроллеров и датчиков, до разработок баз данных и настройки систем оркестрации. Объемы выполняемой работы колоссальные, и, что немаловажно, проводимые исследования находятся на самом острие науки – задачи, решаемые в рамках эксперимента, часто являются нестандартными и уникальными.

Своевременная информационная и программно-техническая поддержка физической части эксперимента, команд разработчиков является актуальной задачей, ведь кроме непосредственного процесса написания кода, разработчики постоянно выполняют связанные с ним второстепенные задачи, которые занимают драгоценное время и силы. Важно активно внедрять современные методы оптимизации и автоматизации рутинных процессов, разрабатывать и развивать уникальные методики, в связи с уникальностью поставленных перед экспериментом целей, основываясь, в первую очередь, на уже имеющемся глобальном опыте в схожих кейсах.

Цель работы и поставленные в её рамках задачи направлены именно на обеспечение такой поддержки крупного физического эксперимента.

Описание проблемной ситуации

В процессе разработки ПО в ОИЯИ разработчик, помимо написания кода, сталкивается с необходимостью:

- Настраивать вручную свое виртуальное рабочее окружение, а также виртуальное окружение в конечной среде (физической машине, виртуальной машине, контейнере);
- Контролировать все системные зависимости, зависимости сторонних функциональных библиотек;
- Подготавливать тестовые данные и подбирать средства тестирования;
- Вручную осуществлять сборку, проверку работоспособности и развертывание ПО;
- Взаимодействовать с конечным пользователем ПО для согласования все тех же зависимостей.

Не менее важными факторами, усложняющими процесс, являются: малое число разработчиков, отсутствие единой методологии и единого набора инструментов для выполнения схожих или одинаковых задач, использование устаревших фреймворков и библиотек, недостаточный уровень коммуникации между отделами и лабораториями.

Актуальной задачей является снижение нагрузки на разработчика и ускорение процесса разработки.

Снизить нагрузку на разработчика можно путем уменьшения времени, которое он тратит на задачи, не относящиеся к написанию кода и подготовке тестовых данных для тестирования ПО: проверку всех зависимостей библиотек и среды разработки/конечной среды, сборку, тестирования и развертывания приложений.

Цель и основные требования к результату магистерской работы

Цель

Разработка программных средств, их настройка и методика их использования для автоматизации процесса разработки, тестирования и развертывания ПО для распределенной среды обработки данных эксперимента *SPD* на коллайдере *NICA* к июню 2025 года.

Основные требования

Основные функциональные требования, предъявляемые к разрабатываемой системе:

- Должна быть разработана и реализована методика автоматической сборки прикладного ПО на основе методологии *GitFlow* и средств *CI/CD*.

Настроенные технические средства должны позволять:

- Осуществлять авторизованную работу с репозиторием кода на основе соглашений сообщества разработчиков *SPD* [1].
- Автоматически собирать образ с приложением, проверять его работоспособность и отправлять в хранилище образов.

Соответствовать критериям:

- надежности;
- гибкости;
- функциональности.

К дополнительным критериям можно отнести:

- легкость применения;
- сопровождаемость.

Глава 1. Анализ существующего процесса разработки

Как уже было отмечено ранее, процесс разработки ПО в ОИЯИ нельзя назвать эффективным. На это влияют многие факторы. Для выявления источников неэффективности в первую очередь было проведение интервьюирования лаборантов, научных сотрудников и практикантов, активно задействованных в написании кода. Именно они сталкиваются с трудностями «ручного» выполнения многих задач.

Вынесенные на интервью вопросы:

- Что из себя представляет проект, в котором задействован разработчик, в чем состоит цель проекта, и как давно проект существует?
- Какие шаги выполняются разработчиком, когда поступает запрос на внесение изменений в программный код?
- Какие технологии при этом задействованы?
- Сколько по времени занимают все процессы, кроме непосредственного внесения изменения в программный код?
- Какие, по мнению разработчика, существуют проблемы в процессе разработки проекта?

Интервью было проведено с разработчиками:

- фреймворка *Gaudi* (*Sampo*);
- фреймворка *SPDRoot*;
- сервиса *SPD EventIndex*;
- сервиса *SPD Metadata DB*;
- сервиса *SPD Hardware DB*.

Выбор именно этих программных решений обусловлен по нескольким причинам:

1. На данный момент разработка и поддержка этих программных решений является актуальным для коллаборации.
2. Сервисы *EventIndex*, *Hardware DB* и фреймворк *Sampo* были спроектированы относительно недавно, и разработка над ними активно ведется в течение последнего года.
3. В работе над сервисами принимают участие молодые специалисты с свежим взглядом на разработку ПО, которым интересно и важно применять самые современные и актуальные средства автоматизации.
4. Сервисы будут введены в реальную эксплуатацию не скоро, потому что для эксперимента разработка оборудования и программной инфраструктуры, от которых зависит устройство этих сервисов, находится в начальной стадии. Из этого следует, что они будут активно дорабатываться и улучшаться с появлением уровня *hardware*, а значит процесс разработки важно научиться оптимизировать уже сейчас.
5. Фреймворк *SPDRoot*, *Gaudi* представляют из себя *legacy* код и разобщенные репозитории. Существует полноценная неразбериха в текущем состоянии

фреймворков, образы вообще отсутствуют или находятся в подвешенном состоянии. Процесс разработки недостаточно организован и плохо структурирован.

На основе проведенного интервьюирования и состояния репозиториев можно составить общую концептуальную последовательность действий, которую можно описать и представить в собственной нотации (см. рис. 1.1). Данная схема подходит, по сути, не только под конкретные фреймворки и сервисы, но и под обширный пласт процессов современной разработки с применением схожих технологий средств виртуализации, контейнеризации.

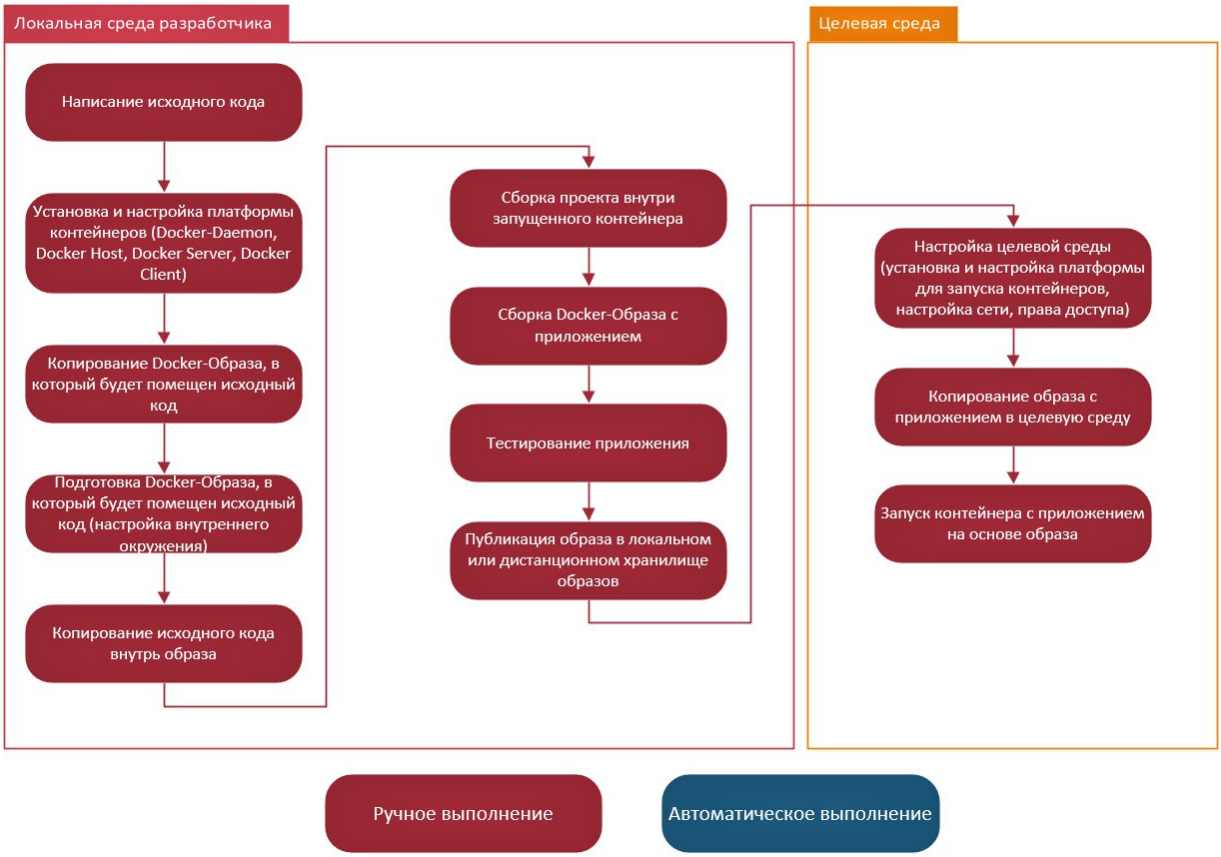

Рисунок 1.1. Общая схема последовательности шагов в процессе разработки, сборки, тестирования и развертывания приложения для представленных фреймворков и сервисов

Условно, можно разделить взаимодействие с приложением на 2 части: локальная среда разработчика и целевая среда:

- Локальная среда представляет из себя репозиторий и окружение любого устройства, виртуальной машины, где ведется разработка и состояние репозитория фиксируется периодически.
- Целевая среда – это физический сервер, виртуальная машина с определенным окружением: ОС, набором системных сервисов, переменных и тому подобного, где требуется приложение развернуть.

Как можно понять по схеме (см. рис. 1.1), действия при наихудшем раскладе на данный момент выполняются вручную. Все зависит от конкретного проекта и команды разработчиков, например для обоих фреймворков *SPDRoot* и *Gaudi* (на основе него сейчас создается *Sampo*, поэтому упоминаются одновременно и со скобками) – все действия выполняются разработчиками вручную.

Чтобы нагляднее и подробнее рассмотреть все стадии процесса разработки, была реализована модель *AS-IS*, в нотации *IDEF0*, с уровнем декомпозиции 2.

Модель AS-IS («Как-есть»)

Это модель существующего состояния (см. рис. 1.2). Она описывает существующие процессы, то, как они работают и взаимодействуют друг с другом здесь и сейчас. Эта модель позволяет оценить текущее состояние процессов.

Входными данными на данный момент являются:

- Начальный исходный код. Хранится локально или дистанционно.
- Сторонние функциональные библиотеки. Разработчик имеет их у себя локально, либо устанавливает из сторонних источников. Надежность этих источников разработчик проверяет сам, берет на себя ответственность и следит за правильностью выбора.
- Набор системных зависимостей.
- Базовый *Docker*-образ. Хранится локально или в хранилище образов.
- Тестовые данные.

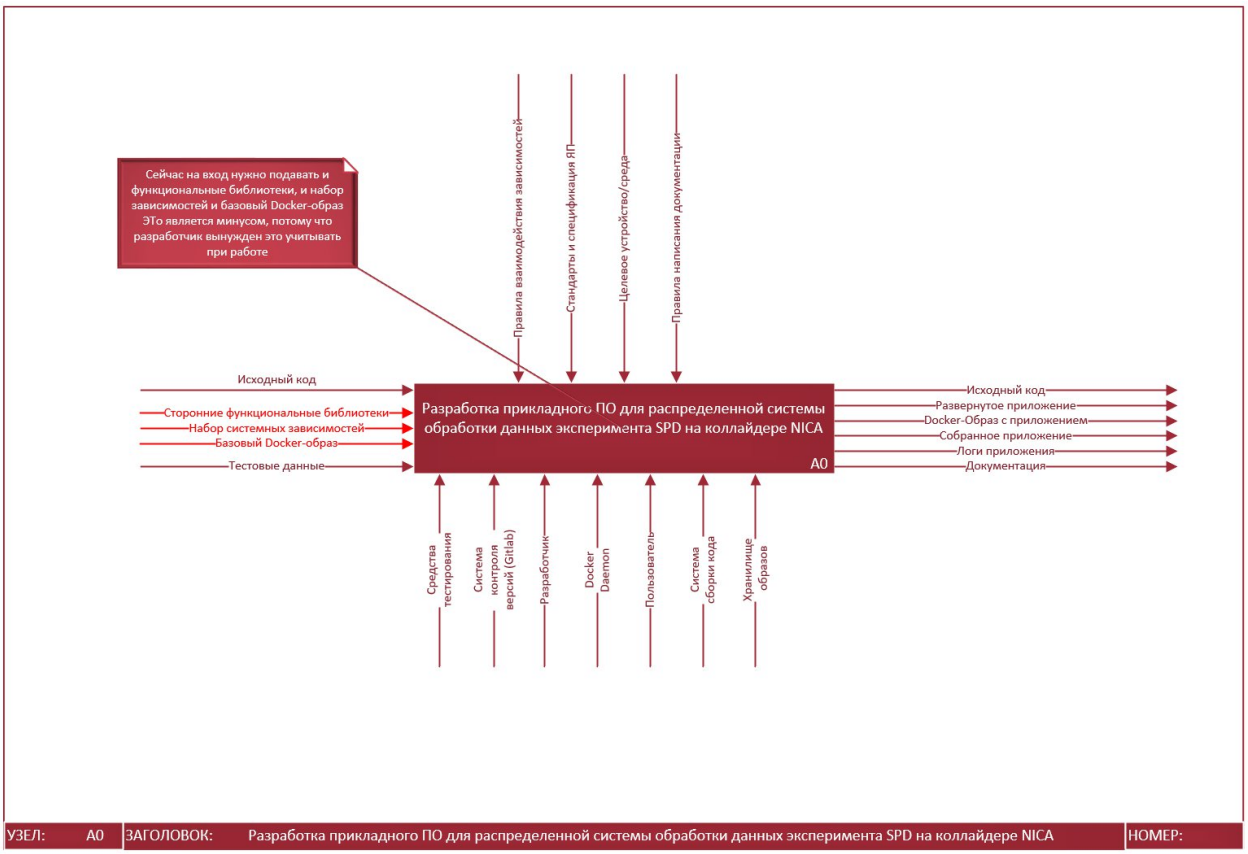

Рисунок 1.2. Модель *AS-IS* диаграмма в нотации *IDEF*0

На первом уровне декомпозиции процесса *A*0 (см. рис. 1.3) процесс состоит из 6 основных подпроцессов:

1. Разработка кода проекта. Рассматриваться не будет (в контексте *AS-IS*), потому как тема работы не предполагает вмешательство в непосредственное написание кода проекта (приложения).
2. Написание образа для разработки приложения.
3. Сборка проекта внутри контейнера.
4. Тестирование.
5. Публикация образа. Рассматриваться не будет, потому как процесс публикации образа приложения можно назвать тривиальным.
6. Развертывание.

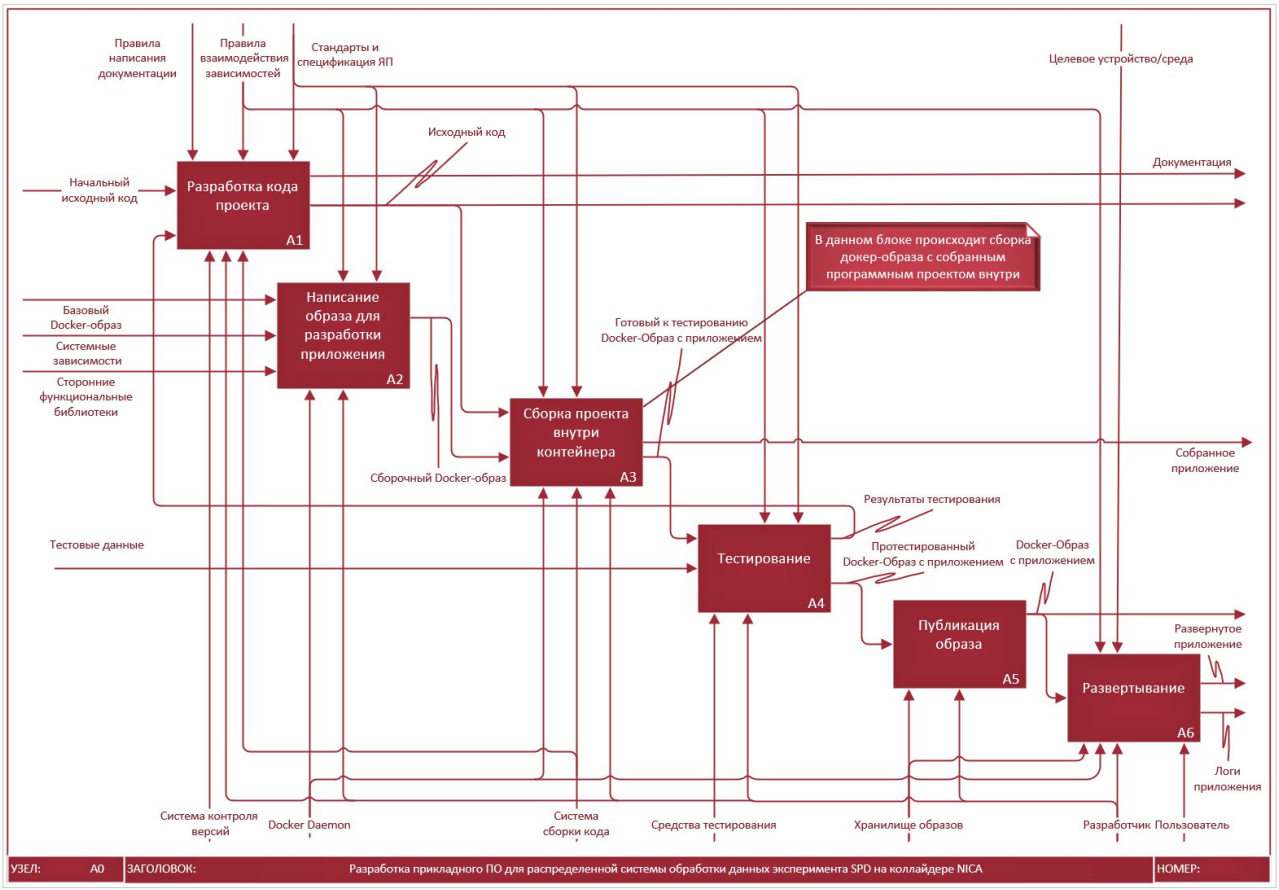

Рисунок 1.3. Декомпозиция процесса *A*0 «Разработка прикладного ПО для распределенной системы обработки данных эксперимента *SPD* на коллайдере *NICA*»

Процесс *A*2 (см. рис. 1.4) представляет из себя набор процессов, направленных на создание образа, в котором будет вестись разработка. Программист должен написать *Dockerfile* внимательно анализировать набор системных зависимостей, устанавливать функциональные библиотеки, а потом и собирать образ. На этот процесс разработчик отдает силы и время, которые мог бы потратить на написание кода приложения.

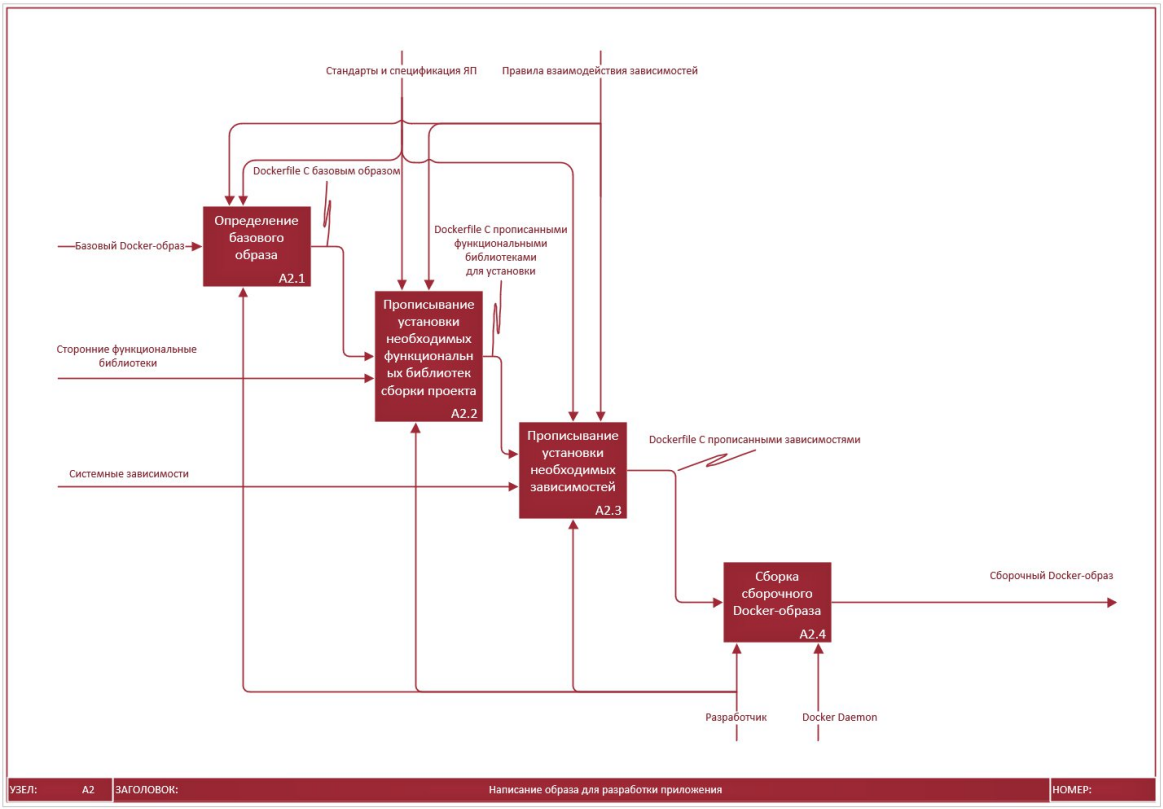

Рисунок 1.4. Декомпозиция процесса *A*2 «Написание образа для разработки приложения»

Далее, в процессе *A*3 (см. рис. 1.5), в поднятом, запущенном в системе контейнере, у себя в локальной или дистанционной виртуальной среде (виртуальной машине), программист копирует исходный код проекта, настраивает систему сборки приложения. По итогу получится собранное приложение и *Docker*-образ с приложением.

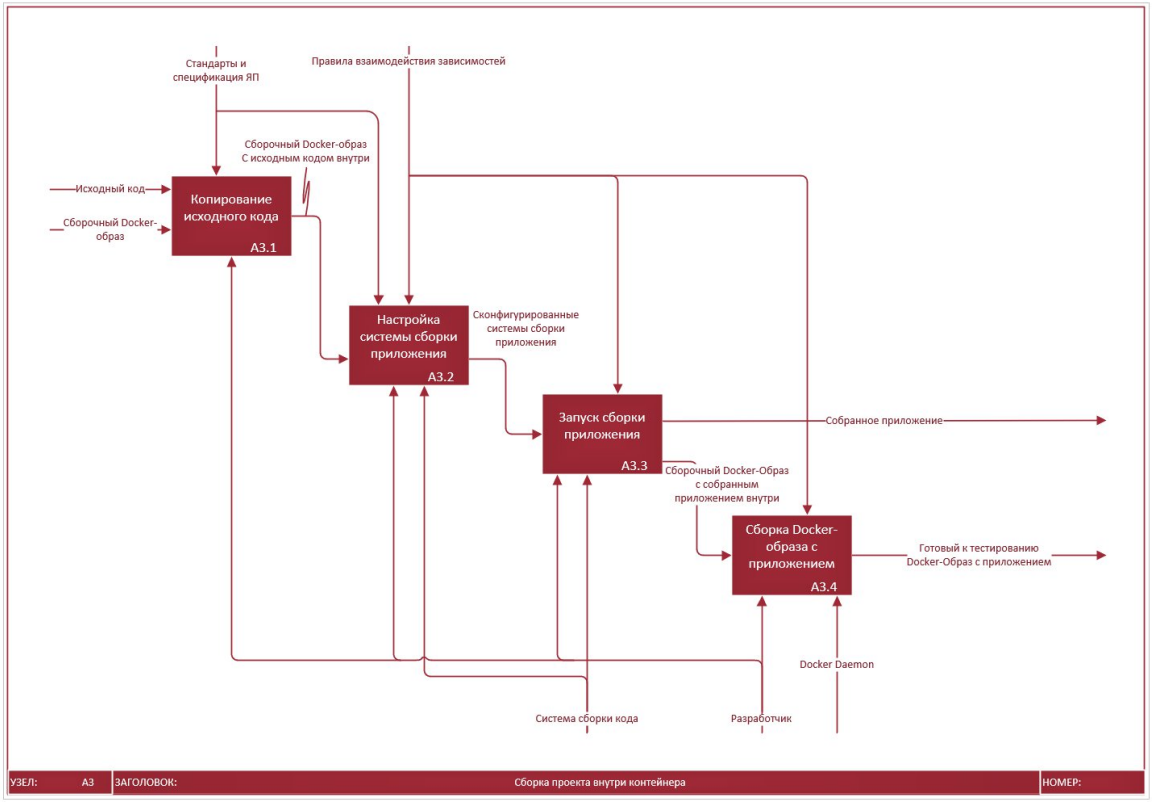

Рисунок 1.5. Декомпозиция процесса *A*3 «Сборка проекта внутри контейнера»

В процессе *A*4 (см. рис. 1.6) описывается разработка тестов различных видов и выполнение тестирования, всё опять же выполняется вручную силами разработчика.

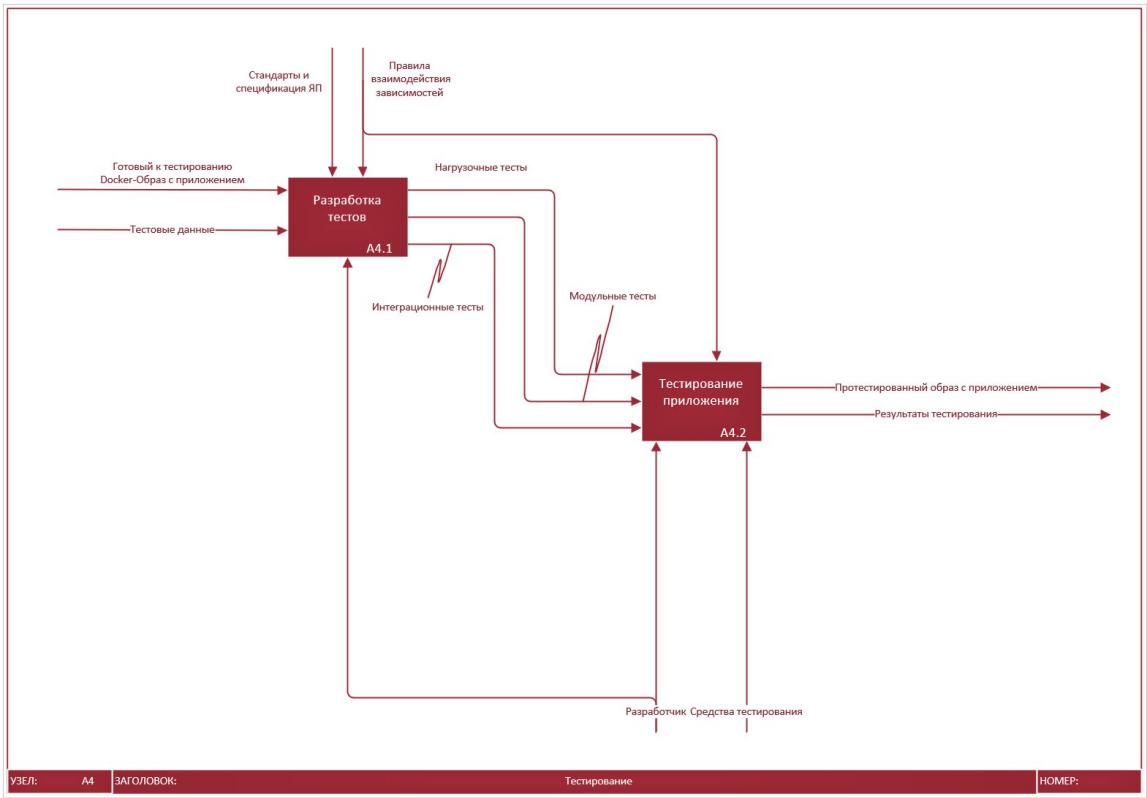

Рисунок 1.6. Декомпозиция процесса *A*4 «Тестирование»

В процессе *A*6 (см. рис. 1.7) описывается один из самых трудоемких этапов – развертывание контейнера с приложением в целевой среде. Изолированное окружение контейнера само по себе упрощает задачу развертывания, ведь все системные зависимости уже упакованы внутрь образа, однако остается подготовить саму целевую среду: установить менеджер контейнеров, выдать права доступа в системе и правильно настроить сеть. Например, для веб-приложения, которое будет развернуто на виртуальной машине, последний пункт – самый критичный, поэтому в подготовке к запуску участие принимает пользователь и разработчик, очень часто этот процесс затягивается – постоянно всплывают ошибки сети, портов или прав доступа, ведь тестирование проводилось средствами самого разработчика, предположить все тонкости для целевой среды возможности не имелось.

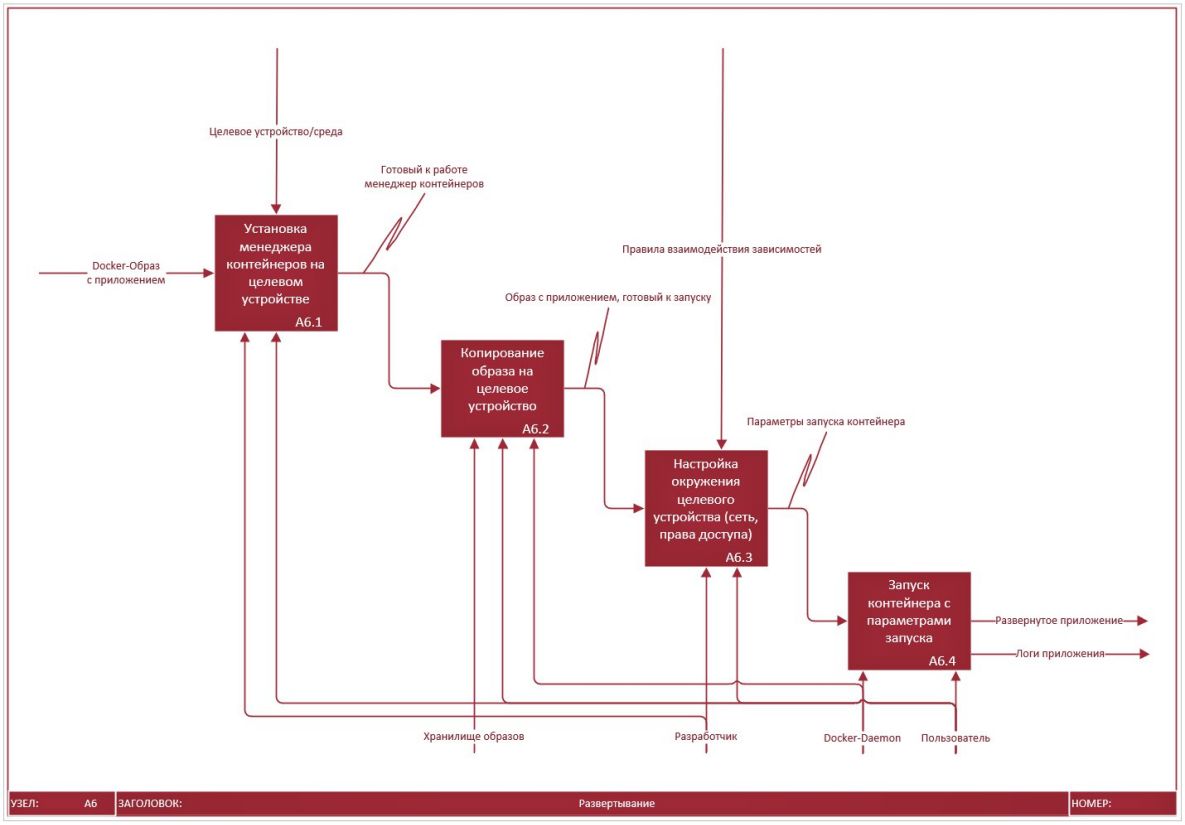

Рисунок 1.7. Декомпозиция процесса *A*6 «Развертывание»

Как уже было отмечено ранее, существующий процесс разработки нельзя назвать эффективным. На основе проанализированных данных и запроса руководства эксперимента было принято решение обратиться к опыту применения современных общепризнанных стандартов автоматизации процессов при создании ПО, изучить и разработать собственную методику их применения. Современные подходы и технические средства доказали свою эффективность в бизнес-проектах разного уровня и схожих по своим масштабам и задачам научных экспериментах [2].

Глава 2. Изучение современных подходов и средств реализации

Методология DevOps

Корнями *DevOps* уходит в *Agile* разработку, которая отличается от «каскадной», где применяется утвержденное еще до начала реализации проекта четко задокументированное ТЗ, разграничение по иерархии управления, строгого определенные этапы и выполнение их в четком порядке. *Agile* – это итеративный подход в разработке ПО, сфокусированный на плотном взаимодействии между разработчиками и обладающий высокой гибкостью по сравнению с линейным процессом [3].

DevOps [4] объединяет разработку (*Dev*) и операции (*Ops*) для объединения людей, процессов и технологий в планировании приложений, разработке, доставке и операциях. *DevOps* обеспечивает координацию и совместную работу между ранее разложенными ролями, такими как разработка, ИТ-операции, проектирование качества и безопасность. Центральным аспектом является непрерывность процессов – разработка превращается в бесконечный цикл, состоящий из: планирования, разработки, сборки, тестирования, выпуска, развертывания, эксплуатации, мониторинга. Оканчивается итерация – начинается новая. Важно понимать, что первые четыре шага традиционно относятся к командам *dev*, а вторые – к командам *ops*. Именно объединение этих двух отличающихся сущностей с помощью технических средств и принципов является ключевой задачей методологии.

Методология включает в себя набор *best practices* – «лучших практик», направленных на ускорение внедрения изменений и выпуска продукта, автоматизацию процессов сборки, тестирования и доставки, улучшение координации между командами разработчиков:

- непрерывная интеграция и непрерывная доставка;
- управление версиями;
- непрерывное тестирование;
- непрерывное наблюдение;
- инфраструктура как код;
- микросервисная архитектура приложений.

Применение *DevOps* требует от процесса разработки продукта определенных отношений в различных направлениях, таких как: гибкость, автоматизация, культура совместной работы, непрерывные измерения, обеспечение качества, устойчивость, обмен информацией и прозрачность [4]. Такие требования задают высокую планку для предприятий и команд. В текущих условиях реализация методологии *DevOps* в работе над ПО для эксперимента *SPD* в полной мере не представляется возможной, однако, применение некоторых практик и подходов с формулированием единой методики может способствовать оптимизации процесса разработки.

Практики контроля и управления версиями

В рамках разработки ПО важнейшим элементом является система контроля версий, способная объединить усилия при совместной разработке. На данный момент применение *Git* является стандартом в *IT*-индустрии [5].

Совместное решение любой поставленной задачи неизменно содержит в себе сложности коммуникации между людьми: различная глубина понимания контекста, субъективный взгляд каждого человека. В разработке примером такой сложности могут служить ведение репозиториев с кодом и присваивание версий программному продукту.

Например, к ведению репозиториев можно отнести: наименование репозиториев, коммитов и ветвей, ведение и расположение документации.

Важно, чтобы все разработчики придерживались единых подходов при работе над репозиториями, что позволит упростить реализацию проектов. Такие подходы в совокупности можно обозначить как практики контроля версий и управления ими, которые упрощают разработку программного обеспечения за счет улучшения отслеживания кода, совместной работы и соответствия требованиям к версиям [5].

Модели ветвления Git

Существует огромное множество подходов, которые описывают, как правильно вести репозитории с кодом и как фиксировать изменения. Такие подходы называются моделями ветвления. Основными, на данный момент, являются: *GitFlow* [6] и *Trunk-Based Development* (*TBD*) [7].

GitFlow — модель управления ветками кода, над моделью ветвления *Git,* которая включает в себя использование функциональных веток и несколько основных веток [6].

Ветка или *branch* – это среда в системе контроля версий, в которой можно реализовать новые идеи для проекта. Так как модель ветвления принято описывать как граф, то основные понятия, с которыми имеет дело разработчик, это – ветви. Когда требуется реализовать новый функционал для ПО, разработчик создает новую ветку на основе ранее созданных ветвей.

GitFlow имеет многочисленные, долгоживущие ветки и крупные фиксации или, по другому, коммиты. В рамках этой модели разработчики создают функциональную ветку и откладывают ее слияние с основной веткой до тех пор, пока функционал не будет завершен. Эти долгоживущие функциональные ветки требуют большего сотрудничества для слияния и имеют более высокий риск отклонения от ветви магистрали. Они также могут вводить противоречивые обновления.

В своей сути в модели разработки *GitFlow* присутствуют две основные постоянные ветки:

- ***main*** (стабильно работающая или «*production*» версия кода);
- ***develop*** (последняя или «*nightly*»-версия кода).

main — это основная ветка кода, продакшен версия кода, то есть все, что потребуется выпустить для реальных пользователей, собирают из этой ветки. Ветка может быть помечена тегами, в которых есть информация о версии, часто к тегу еще привязываются релизы. Версия формулируется на основе «Семантического версионирования». Любая новая задача, взятая в работу, начинается с этой ветки. Суть этой ветки хранить самую последнюю работоспособную версию кода.

develop — суть этой ветки хранить в себе самую последнюю кодовую базу проекта [6].

Trunk-Based Development (*TBD*) – модель, которая фокусируется на одной основной ветке, например *main* или *master*, в которую быстро и часто вносятся изменения. В рамках данной модели принято реализовывать *DevOp*s практики [7].

На реализацию нового функционала отводится отдельная, функциональная ветка, жизненный цикл которой всего лишь несколько дней. Слияние этой ветки с основной производится, даже если функционал не реализован до конца. Как только есть стабильная версия, она выпускается как релиз. Последующая реализация функционала продолжается с слиянием с основной веткой в новую функциональную, чтобы также прожить всего несколько дней [7].

Особенности процесса разработки РСОД *SPD*:

- Новые подходы и технологии внедряются медленно.
- Высокая параллельная нагрузка на команды вследствие небольшого состава разработчиков, что влечет за собой отсутствие достаточного количества тестов. Например, команда разработки сервиса *SPD Metadata DB* – 1 человек, а фреймворка *SPD Sampo* – 3 человека. Продукты могут иметь уязвимости, которые не обнаруживаются и не исправляются на протяжении месяцев или даже лет.
- Отсутствие четко сформулированной и отлаженной бизнес-модели подготовки программных продуктов, мало системности. Подходы к разработке могут быть едиными в рамках одной команды, максимум – отдела, но не в рамках сектора, лаборатории или эксперимента.

Вследствие всех этих особенностей складывается ситуация, в которой применение подходов с короткоживущими ветками репозиториев будет не актуально. Выпуск релизов для эксперимента сейчас медленный процесс, а применять сложный, требующий большой зависимости от уровня кадров и коммуникации между отделами *Agile* на данном этапе нецелесообразно. В рамках работы необходимо создать базовый фундамент, который позволит стандартизировать и оптимизировать некоторые процессы, ускорить выпуск релизов, создать понятную среду, без хаоса в репозиториях [7].

На основе особенностей процесса разработки принято решение придерживаться практики *GitFlow*, которая подходит для выпуска редких релизов и научных *Open-Source* проектов. Для данной практики несмотря на то, что общепринятым стандартом считается

применение *CI/CD* с *TBD*, также подойдут инструменты автоматизации развертывания и тестирования ПО. Для *CI/CD* нет строгих ограничений в применении.

Важным аспектом в выборе такой модели является большая зависимость *TBD* от автоматического всестороннего тестирования в момент фиксации изменений, что на данном этапе реализовать не представляется возможным.

Соглашение о коммитах

Стоит обратить внимание на проблемы, которые возникают у разработчиков, когда они сталкиваются с чужим описанием фиксации версии кода. Когда фиксация не содержит в своем комментарии достаточной информации о внесенных изменениях – это ухудшает понимание другими участниками проекта кода и замедляет поиск ошибок. Грамотное, четкое, контекстно-ориентированное описание коммита, основанное на единой методике, способствует повышению качества разработки [8].

Одним из таких подходов является применение «Соглашения о коммитах» (*Conventional Commits*), в рамках которого каждый коммит должен быть описан по определенной схеме, включающей определение типа коммита, контекста, описания, тела и сносок.

Спецификация «Соглашение о коммитах» — простое соглашение о том, как нужно писать сообщения коммитов. Оно описывает простой набор правил для создания понятной истории коммитов, а также позволяет проще разрабатывать инструменты автоматизации, основанные на истории коммитов. Данное соглашение совместимо с Семантическим Версионированием, описывая новые функции, исправления и изменения, нарушающие обратную совместимость в сообщениях коммитов [9].

Семантическое версионирование

Одна из существующих проблем в разработке – «ад зависимостей» (*dependency hell*). Заключается она в том, что чем больше растёт система и чем больше библиотек применяется в проекте, тем больше вероятность оказаться в «аде зависимостей».

Если спецификации зависимости слишком жесткие, вы находитесь в опасности блокирования выпуска новой версии (невозможность обновить пакет без необходимости выпуска новой версии каждой зависимой библиотеки). Если спецификация зависимостей слишком свободна, вас неизбежно настигнет версионное несоответствие (необоснованное предположение совместимости с будущими версиями

Для решения проблемы был разработан набор правил и требований, которые определяют, как назначаются и увеличиваются номера версий – «Семантическое версионирование (*Semantic Versioning*) [10].

Семантическое версионирование – это полезный инструмент, его важность обоснована и доказана опытным путем, в сравнении с версионированием, заданным

обособленными группами разработки, позволяет добиться стандартизации и используется повсеместно в современной разработки, особенно с общедоступным *API* [11; 12].

CI/CD

В процессе разработки ПО возникает потребность в его реактивной поддержке. У пользователей появляется желание получать новые версии с новым функционалом и исправлением багов чаще. Перед каждым релизом разработчики вынуждены проводить тестирование ПО, поэтому каждый выпуск релиза после добавления функциональности вызывает чрезмерные трудозатраты. Следовательно – разработчикам не выгодно делать релизы после добавления нового функционала или исправления, а пользователям этого хочется.

Цель *CI/CD* (*Continuous Integration / Continuous Development (Deployment)* ускорить обнаружение дефектов, повысить производительность и обеспечить более быстрые циклы выпуска: объединяет разработку, тестирование и развёртывание приложений [4].

CI/CD — это одна из *DevOps* практик, которая позволяет разработчикам чаще и надёжнее развёртывать изменения ПО, свести к минимуму ошибки, повысить темпы сборки и качество разрабатываемого продукта. Представляет собой комбинацию *continuous integration* и *continuous delivery*.

CI, или непрерывная интеграция — процесс постоянной разработки ПО с интеграцией в основную ветвь. Автоматически собирает софт, тестирует его и оповещает, если что-то идёт не так.

CD, или непрерывная доставка — процесс постоянной доставки ПО до потребителя. Обеспечивает разработку проекта небольшими частями и гарантирует, что он может быть отдан в релиз в любое время без дополнительных ручных проверок.

Пример реализации *CI/CD* на практике:

- Разработчик отправляет коммит в репозиторий, создаёт *merge request* через сайт, или ещё каким-либо образом явно или неявно запускает конвейер.
- Из конфигурации выбираются все задачи, условия которых позволяют их запустить в данном контексте.
- Задачи организуются в соответствии со своими этапами.
- Этапы по очереди выполняются — т.е. параллельно выполняются все задачи этого этапа.
- Если этап завершается неудачей (т.е. завершается неудачей хотя бы одна из задач этапа) — конвейер останавливается (почти всегда).
- Если все этапы завершены успешно, конвейер считается успешно прошедшим.

Основное преимущество *CI/CD* в том, что разработчику нужно только написать код, а остальные процессы по тестированию, сборке и доставке проходят автоматически. Технология позволяет свести к минимуму ручной труд при выполнении рутинных операций: сборки и выкладки новых версий кода.

За счёт автоматизации процесс *CI/CD* решает ряд важных проблем:

1. **Отсеивает некоторый процент ошибок на пути в продакшн**. Центральная часть любого *CI/CD*-конвейера — набор автоматизированных тестов. Автоматизация тестирования гарантирует, что тесты выполняются точно, а результаты — будут надёжны.
2. **Улучшает качество кода**. *CI/CD* автоматизирует запуск множества проверок: *unit*-тестирования, интеграционного тестирования и др. Это позволяет разработчикам заниматься программированием и кодом, а не тратить время на разбор инцидентов и поиск их причин.
3. **Устраняет разобщённость между разработкой и операционной деятельностью**. *CI/CD* — это единая точка коммуникации контроля разработки и интеграции ПО. Вы можете задействовать в создании продукта самых разных специалистов и добиться прозрачности процесса разработки и сотрудничества команд друг с другом.
4. **Повышает скорость внедрения нового функционала**. Вы можете настроить единообразный и простой интерфейс для всех команд разработчиков и организовать их одновременную работу над разными частями проекта. Каждая команда будет создавать и развёртывать службы, которыми она владеет, не влияя на другие команды и не нарушая их работу. Плюс, внедрение *CI/CD* влечёт повышение квалификации команды и создаёт среду, где каждый специалист занят своим делом.
5. **Позволяет быстро исправлять ошибки в ПО**. *CI/CD* хранит историю сборок за какой-то промежуток времени. При необходимости вы можете увидеть, когда было то или иное изменение, и откатиться назад, если что-то пошло не так. Чем раньше обнаружен баг, тем дешевле его исправление. Во-первых, потому что разработчик все ещё работает над той же частью программного кода, ему не нужно переключаться между контекстами, и он вносит правки буквально «на лету». Во-вторых, потому что ветки кода не успели размножиться на последующие релизы, и не придётся править один баг в нескольких местах.

Построение *CI/CD*-конвейера делает процесс разработки и выпуска ПО более компактным и эффективным. Пока компьютеры выполняют рутину, разработчики могут сосредоточиться на решении организационных задач и проведении экспериментов.

Правильно настроенный конвейер, основанный на правилах срабатывания способен существенно сократить время на сборку новой версии проекта [13], например, если был исправлен файл с документацией или комментариями к проекту.

Платформа системы контроля версий и жизненного цикла ПО

Gitlab

Для хранения и управления репозиториями ОИЯИ использует собственный развернутый на серверах Лаборатории Информационный Технологий экземпляр

платформы системы контроля версий и жизненного цикла ПО под названием *Gitlab*. Он считается стандартным решением для всех проектов института, вследствие чего, в рамках работы не предполагается обзор альтернатив.

GitLab — веб-инструмент для управления проектами и репозиториями, подходящий для реализации жизненного цикла *DevOps* с открытым исходным кодом, представляющий систему управления репозиториями кода для *Git* с собственной вики, системой отслеживания ошибок, *CI/CD* конвейером и другими функциями. Ключевыми особенностями *GitLab* являются непрерывность процесса разработки и взаимная интеграция различных элементов [14].

Отличительные черты *GitLab*:

- встроенные функции *CI/CD* (*Gitlab CI/CD*);
- автоматическое обнаружение секретов и тестирование безопасности;
- явные разрешения для ограничения слияния и отправки кода определенным пользователям, группам;
- бесплатные статические веб сайты с авто собираемой документацией в репозиториях *Git* (*Gitlab Pages*) и *Wiki* с документацией для каждого проекта.

К особенностям *Gitlab* также можно отнести его модульность, сочетание в себе большого числа полезных функций и сервисов: *Gitlab CI/CD, Gitlab Pages, Gitlab Runners, Gitlab Container Registry, Package Registry, Gitlab AI*. Платформа фокусируется на *DevOps* и *CI/CD*, в соответствии с этим фокусом на платформе представлена удобная интеграция с контейнерной средой *Docker* и платформой для автоматизации развёртывания и масштабирования контейнеризированных приложений *Kubernetes*.

Gitlab CI/CD

Gitlab CI/CD — это комплексная система, встроенная в *Gitlab*, которая автоматизирует процессы сборки, тестирования и развертывания программного обеспечения. *Gitlab CI/CD* управляет конфигурациями через *YAML*-файлы, находящиеся в репозитории проекта. Инструмент позволяет интегрироваться с кластерами *Kubernetes*. А еще поддерживает применение образов контейнеров для настройки окружения — это обеспечивает высокую гибкость и позволяет многократно использовать код [15].

Составные части *Gitlab CI/CD*:

- Конвейер (*pipeline*) – набор задач, организованных в этапы, в котором можно собрать, протестировать, упаковать код, развернуть готовую сборку в облачный сервис и так далее (см. рис. 2.1) Как будет рассмотрено далее, выполнением наборов задач – конвейеров (*pipelines*) *Gitlab CI/CD* занимаются исполнители (*Gitlab Runners*).
- Этап (*stage*) – единица организации конвейера, содержит как минимум 1 задачу.
- Задача (*job*) – единица работы в конвейере. Состоит из скрипта (обязательно), условий запуска, настроек публикации/кеширования артефактов и много другого.

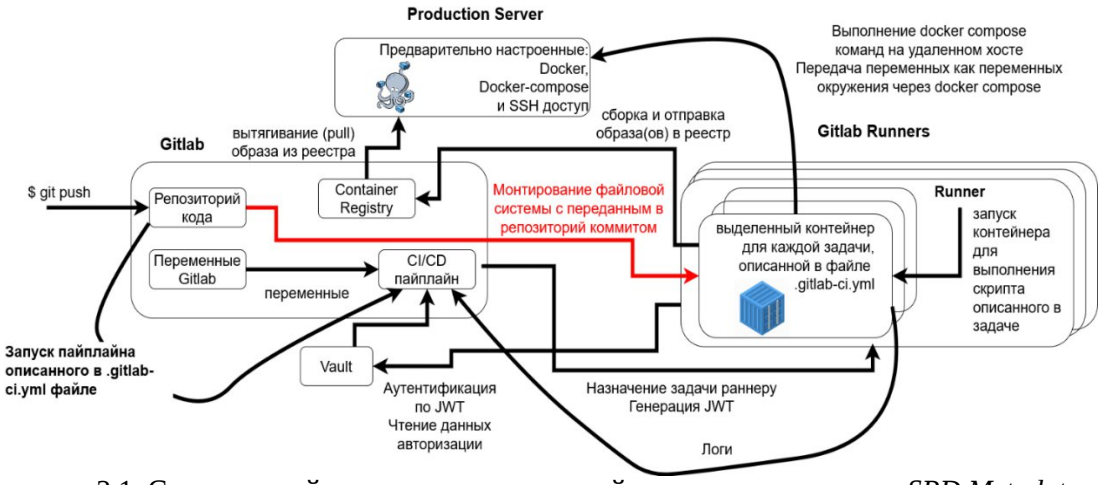

Рисунок 2.1. Схема конвейера для микросервисной архитектуры проекта *SPD Metadata* на текущий момент

При развертывании на сервере старые версии отключаются, удаляются и запускаются новые, обновленные, взятые из хранилища контейнеров.

Gitlab Runner

GitLab Runner или Исполнитель — это приложение-агент, который занимается выполнением инструкций из специального файла *gitlab-ci.yml* [14]. Его установка возможна на любую операционную платформу, будь то *Windows*, *MacOS* или *Linux*-подобные системы.

В возможности входит:

- выполнение задач, прописанных в конвейере;
- запуск задач совершенно различными способами: локально, в *Docker*-контейнерах, в различных облаках или через *ssh*-подключение к какому-либо серверу;
- выполнение задач параллельно с другими Исполнителями, а также обмен данными по средствам артефактов (*artifacts*).

Gitlab Container Registry

Запуск конвейера сопровождается сборкой образов, автоматическим присваиванием тега на основе ветки и совершённого коммита, отправкой этих образов в *Container Registry*.

Gitlab Container Registry — это безопасный приватный реестр для образов *Docker*. *Gitlab Container Registry* полностью интегрирован в *Gitlab* [14].

Базовые образы и образы приложений будут располагаться именно здесь. Существуют различные варианты хранилищ образов, однако, применение *Container Registry* как основного реестра образов приложений РСОД *SPD* обусловлено несколькими причинами:

- Полная интеграция *Container Registry* в *Gitlab* и использование этого реестра по умолчанию при работе с *Gitlab CI/CD*.
- Наличие всех необходимых для процесса разработки функций.
- Локальное размещение на серверах вычислительного комплекса ЛИТ ОИЯИ.

- Внешнеполитическая мировая ситуация и санкции. В любой момент компании могут запретить пользователям России получать образы из облачных реестров, например, таких как *DockerHub*.

Gitlab Pages

Для эффективной поддержки продукта, требуется документация, но, как правило, она устаревает быстрее всего в проектах. Поэтому необходимо средство для автоматической генерации документации и её актуализации для всех команд проекта. Для того, чтобы по исходному коду и комментариям в определенном формате сгенерировать документацию, мы используем *Sphinx*, а для того, чтобы поделиться документацией со всеми участниками проекта – используется *Gitlab Pages* [14], который размещает и обновляет сайт с документацией в рамках выполнения конвейера.

С помощью *Gitlab Pages* можно публиковать статические веб-сайты непосредственно из репозиториев *Gitlab*. К особенностям можно отнести:

- возможно использовать любой генератор статических сайтов (*SSG*) или обычный *HTML*;
- возможность создания веб-сайта для своих проектов, групп или конкретной учетной записи пользователя;
- размещение на собственном экземпляре *Gitlab* или официальном *Gitlab.com* бесплатно;
- подключение своих пользовательских доменов и сертификатов *TLS*.

Платформы контейнеров

Контейнеризация – это один из видов виртуализации. Её появление позволило изолировать приложения от среды, на которой эти приложения будут развернуты. Контейнеры имеют отличительные черты:

- легковесность;
- скорость развертывания;
- удобство настройки;
- меньшая изолированность.

Прямым образом сравнивать принципы построения контейнеризации и виртуальных машин – не совсем корректно, по своей сути, это новая ступень в развитии виртуализации.

В современной разработке применение контейнеров позволяет получить преимущество в сравнении с ВМ по скорости развертывания и уменьшению стоимости разработки: чем быстрее продукт пройдет тесты, тем меньше простои в разработке; чем быстрее продукт попадает на рынок, тем выше доходы и так далее.

Docker

Docker — это современный стандарт платформы контейнеров. Его применение можно обнаружить повсеместно. Кроссплатформенность, исчерпывающая по объемам документация. *Docker* — это мощный инструмент для разработки и развертывания

современных приложений, который обеспечивает непревзойденную гибкость, эффективность и масштабируемость [16].

Рассмотрение данной платформы в качестве основы для использования в рамках работы над проектами *SPD* обусловлено интеграцией *Docker* в *Gitlab*. Ранее отмечено – чтобы конвейер работал, нужна среда для выполнения. В качестве такой среды *Gitlab Runner* использует именно *Docker*-контейнер.

Apptainer

Данная платформа позиционируется авторами, как решение для научных исследований и высокопроизводительных вычислений. Имеет полную обратную совместимость с *Docker*.

Архитектурно *Apptainer* не использует «демона», запуск контейнеров происходит в непривилегированном режиме с наименьшими правами доступа [17].

Особенности *Apptainer*:

- Проверяемая воспроизводимость и безопасность, использование криптографических подписей, неизменяемый формат изображения контейнера и расшифровка в памяти.
- По умолчанию используется интеграция вместо изоляции. По умолчанию легко использовать графические процессоры, высокоскоростные сети, параллельные файловые системы в кластере или на сервере.
- Мобильность вычислений. Формат контейнера *SIF* с одним файлом прост в транспортировке и совместном использовании.
- Простая и эффективная модель безопасности. Внутри контейнера вы являетесь тем же пользователем, что и снаружи, и по умолчанию не можете получить дополнительные привилегии в хост-системе [17].

Одной из отличительных особенностей является разница в объемах образов, собранных на основе технологии *Docker* и *Apptainer*. Собранные файлы *Apptainer* занимают меньший объем пространства на дисковом устройстве в сравнении с *Docker*.

Отличительным особенностью также является отсутствие необходимости запускать контейнер с *root*-правами, как для *Docker*, а в самой среде контейнера возможна настройка нескольких пользователей.

В сравнении с *Docker* имеет довольно посредственную по объемам документацию, малое число релизов, узкую направленность, что приводит к сравнительно небольшому объему применений. Это приводит к недостатку информации в глобальной сети о различных *use*-кейсах, *best-practices* в применении этой платформы.

Apptainer предполагается использовать в качестве основной платформы для выполнения *online* короткоживущих изолированных задач на вычислительных комплексах ЛИТ. При выходе новой версии приложения *online-computing*, будут собираться образы *Docker* и *SIF*. *Docker*-образы будут хранится в *Gitlab Container Registry* каждого репозитория воспользоваться ими смогут пользователи на локальных машинах, а файлы

SIF – в *CVMFS*, откуда на их основе будут запускаться контейнеры на вычислительных серверах ЛИТ для разовых задач в рамках продакшн-использования.

Таким образом, в конечном итоге, система сборки, тестирования и развертывания должна поддерживать две версии образов: *Docker* и *Apptainer*. В рамках данной работы не предполагается реализовать внедрение *Apptainer*, однако, в дальнейших планах эта задача станет обязательной частью системы сборки, тестирования и развертывания.

Сторонние инструменты и сервисы

CVMFS

CVMFS (*CernVM File System*) — сервис распределения программного обеспечения от *CERN* [18]. Разработан для помощи в развертывании программ на распределённой вычислительной инфраструктуре, используемой для обработки данных.

Некоторые особенности *CVMFS:*

- реализована в виде файловой системы *POSIX* для чтения (модуль *FUSE)*;
- файлы и каталоги размещаются на стандартных веб-серверах и монтируются в универсальном пространстве имён */cvmfs*;
- использует только исходящие *HTTP*-соединения, что позволяет избежать проблем с брандмауэрами, характерных для других сетевых файловых систем;
- передаёт данные и метаданные по запросу и проверяет целостность данных с помощью криптографических хэшей;
- активно используется в научных исследованиях, во многих случаях заменяет менеджеры пакетов и общие области программного обеспечения на кластерных файловых системах.

CVMFS совместим с *Linux* и *macOS*, но не поддерживает использование нативной *Windows*. Хорошо работает с подсистемой *Windows для Linux (WSL)* версии 2 [18].

Сервис локально развернут на серверах вычислительной среды ЛИТ используется ОИЯИ как основа для хранения данных экспериментов, ПО, образов и других задач. Именно в *CVMFS* будет размещаться реестр *Apptainer*-образов ПО для эксперимента *SPD*.

Имеет поддержку монтирования файловой системы в/из контейнеров *Docker*, *Apptainer*, *Kubernetes*, *Podman*, а также поддержку хранения и распространения распакованных слоев и целиковых образов [19]. Функция хранения и запуска реализована для *Apptainer* и частично для *Docker*.

DockerHub

DockerHub – это облачная платформа для публикации, хранения и распространения *Docker*-образов [20]. Она позволяет разработчикам делиться контейнеризированными приложениями и сервисами.

Это самый большой в мире облачный реестр образов. Сервис создан разработчиками *Docker*. При написании *Dockerfile* этот реестр используется по умолчанию для загрузки базового образа и в него же отправляются собранные образы приложений [20].

Образы операционных систем, на которых основываются базовые образы для ПО *SPD* рассматриваемых в этой работе, загружаются именно из *DockerHub*.

На данный момент рассматривается возможность полностью отказаться от облачных зарубежных реестров, так как использование таких сервисов несет за собой определенные проблемы, рассмотренные выше.

Глава 3. Формулирование методики автоматизации процесса разработки

Первоочередной задачей является уменьшение нагрузки на разработчиков, чтобы они могли сконцентрироваться на основной своей задаче – написании кода и тестов, а побочные задачи были отданы в ведение других специалистов и программно-технических средств. Можно автоматизировать процессы, передав их в руки ПО-исполнителя, которое будет выполнять их в виртуальной среде, скорее всего, на отдельной виртуальной машине и в отдельных контейнерах. Контейнеры – это компонента работы. Ими управляет движок контейнеров, который взаимодействует с ОС. А вот ОС, может быть как на физическом сервере, так и внутри виртуальной машины. Таким образом можно реализовывать приложения сразу в контейнерах, чтобы не тратить время на настройку окружения. А также тестировать и запускать эти контейнеры мы тоже можем внутри контейнеров, созданных на основе образов целевой среды.

Такой подход реализует философия *DevOps* (акроним от англ. *development & operations*) – методология автоматизации технологических процессов сборки, настройки и развёртывания программного обеспечения. Методология предполагает активное взаимодействие специалистов по разработке со специалистами по информационно-технологическому обслуживанию и взаимную интеграцию их технологических процессов друг в друга для обеспечения высокого качества программного продукта [4].

В рамках этой методологии существует «непрерывная интеграция» (*Continuous Integration, CI*) и «непрерывная поставка» (*Continuous Delivery, CD*), представляющие собой культуру, набор принципов и практик, которые позволяют разработчикам чаще и надежнее развертывать изменения программного обеспечения [21].

Цель состоит в том, чтобы ускорить обнаружение дефектов, повысить производительность и обеспечить более быстрые циклы выпуска. Этот процесс отличается от традиционных методов, когда набор обновлений программного обеспечения интегрировался в один большой пакет перед развертыванием более новой версии.

Именно поэтому в модели (см. рис. 3.1) появляется новая сущность – «Средства *CI-CD*».

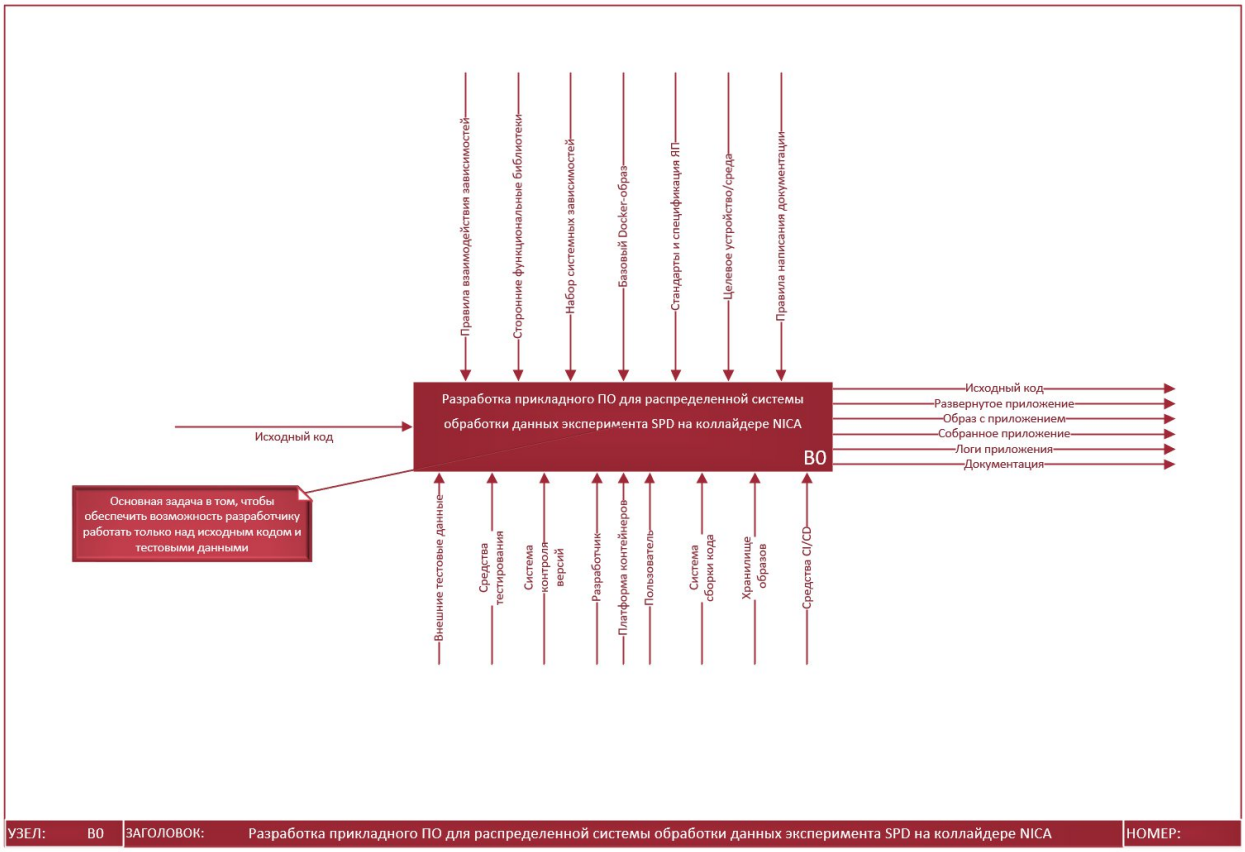

Рисунок 3.1. Модель *TO-BE* диаграмма в нотации *IDEF*0

На данный момент вся инфраструктура разработок для *SPD* базируется на *GitLab*, веб платформе, сочетающей в себе систему контроля версий, в которой интегрированы мощные инструменты для *CI/CD* «из коробки», а также налажена интеграция с системой контейнеризации *Docker*, системой оркестрации *Kubernetes* или отдельными средствами для *DevOps* разработок, например *Jenkins*. На данном этапе руководством лаборатории не вводились ограничения на использование систем контейнеризации.

Контейнеры обеспечивают достаточную виртуализацию для изолированного запуска программ:

- Изоляция: находящиеся в контейнере процессы изолированы друг от друга и от ОС. Каждый контейнер имеет частную файловую систему, поэтому конфликты зависимостей невозможны (при условии, что вы не злоупотребляете томами).
- Параллельность: существует возможность запускать несколько экземпляров одного и того же образа контейнера без конфликтов.
- Меньше нагрузки: поскольку нет необходимости загружать целую ОС, контейнеры куда более легковесны, чем виртуальные машины.
- Развёртывание без установки: установка контейнера — это просто загрузка и запуск образа. Установка не требуется.
- Контроль ресурсов: есть возможность установить ограничения на ресурсы процессора и памяти для контейнеров, чтобы они не оказывали критического воздействия на работу физического сервера.

Именно сущность «Средства *CI-CD*» предполагает под собой использование практик и ПО, нацеленных на автоматизацию рутинных, монотонных и время затратных действий (см. рис. 3.2).

Немаловажным фактором также стоит обозначить четкую регламентацию каждого отдельного базового *Docker*-образа, чтобы их созданием, настройкой и подбором занимался не разработчик, а другой ответственный сотрудник или группа, не относящиеся к конкретной команде разработки, а действующие на несколько проектов, например системный администратор или *DevOps* – инженер. В его задачи также будет входить менеджмент системных зависимостей, сторонних функциональных библиотек, конфигурация и администрирование менеджеров пакетов, хранилищ образов и самих образов для проектов.

Такой подход повысит безопасность, ускорит вывод новых версий приложений за счет использования верифицированного набора библиотек собранных и хранящихся локально на серверах лаборатории, разгрузит разработчиков, унифицирует подходы к созданию образов.

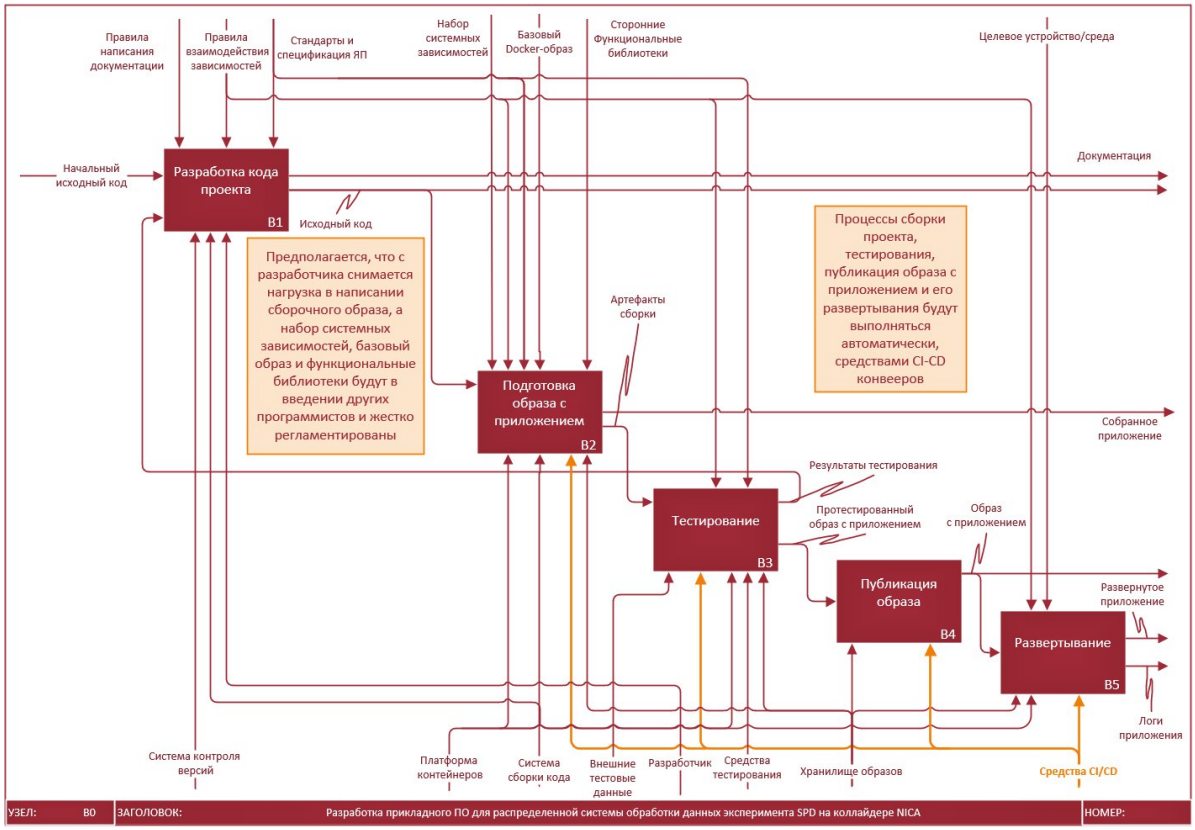

Рисунок 3.2. Декомпозиция *TO-BE* процесса *B*0 «Разработка прикладного ПО для распределенной системы обработки данных эксперимента *SPD* на коллайдере *NICA*»

Настройка и применение средств автоматизации зависит от того, по каким правилам вносятся изменения в код проекта – в зависимости от проекта, ветки, наличия или отсутствия *merge-request*'а, конкретных измененных файлов меняется и конфигурация средств *CI/CD*: присваиваемая версия приложения, тесты, права доступа, наличие или

отсутствие конкретных этапов конвейера и так далее. Вследствие чего появляется необходимость в сформулированной методике внесения изменений в код.

Внесение изменений в код

В код вносится изменение по нескольким причинам:

1) исправление ошибки (бага);
2) добавление нового функционала;
3) рефакторинг (изменение программного кода без изменения поведения системы). К рефакторингу отнесем написание документации.

Процесс внесения изменений тесно связан моделью ветвления, которую использует команда разработчиков. Как было отмечено ранее, *GitFlow* предполагает создание отдельной функциональной ветки для реализации в ее рамках нового функционала приложения на основе сформулированной технической задачи.

Формулированием такой задачи занимается лицо из числа управления командой разработки – лидер бригады разработки, в зависимости от конкретного проекта им могут являться разные должностные лица. В его же задачи входит: выбор ответственного участника команды, который создаст новую функциональную ветку и будет ее курировать; назначение ревьювера.

Ревьювер и разработчик совместными усилиями доводят состояние ветки кода до работоспособного при слиянии его в *develop*-ветку. В течение всего процесса деятельности, они пользуются встроенным в *Gitlab* функционалом *Merge-request, Review-request, Change-request.*

Merge-request запускается 1 раз, а вот *Review-request* может назначаться несколько раз. Ревьювер недоволен проделанной работой – код отправляется на исправление, снова запрос на ревью, снова проверка. Процесс цикличный.

Данные сущности вместе с системой и их деятельность описаны на представленной диаграмме деятельности (см. рис. 3.3). Такой способ описания процесса внесения изменений в код позволяет наглядно рассмотреть четкую последовательность действий и определенные выборы, совершаемые сущностями.

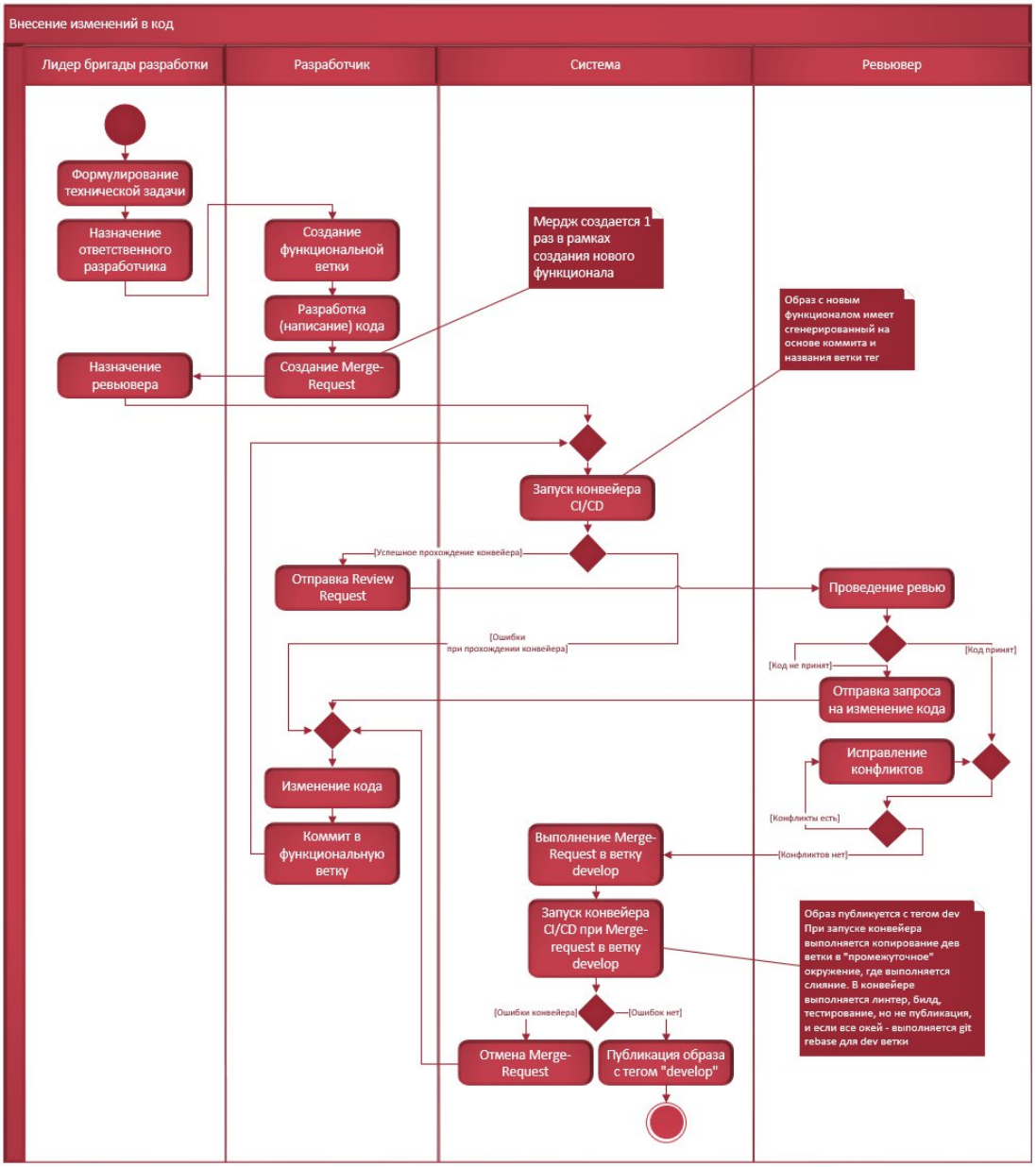

Рисунок 3.3. Диаграмма деятельности процесса внесения изменений в код

При описании всей методики с помощью диаграмм деятельности во всех последующих диаграммах в явном виде намеренно не добавлены узлы решения с проверкой на правильность выполнения каждого действия: этапа, задачи и шага конвейера *CI/CD*. Связано это с тем, что конвейер по умолчанию прерывается при появлении любой ошибки и на любом шаге своего выполнения. При каждом запуске конвейера в веб-интерфейсе *Gitlab* отображается ход его выполнения. В этот же интерфейс передается поток ввода/вывода каждого контейнера, в котором выполняется задача (*job*), благодаря чему представляется возможным анализировать ошибки и выполнять отладку.

Фиксация изменений в функциональную ветку или слияние одной ветки с другой провоцирует выполнение конвейера. В первую очередь выполняется этап сборки проекта.

Сборка проекта внутри контейнера

В рамках этого этапа средствами *CI/CD* подготавливается сборочное окружение в контейнеризированной среде, проект собирается внутри сборочного контейнера, а также запускаются тестовые команды (это может быть вызов какой-то функции или вывод строки в поток ввода/вывода) внутри контейнера, при их наличии (см. рис. 3.4).

По окончании данного этапа и каждой задаче конвейера на выходе остаются артефакты сборки, с которыми последует взаимодействие на следующих этапах конвейера.

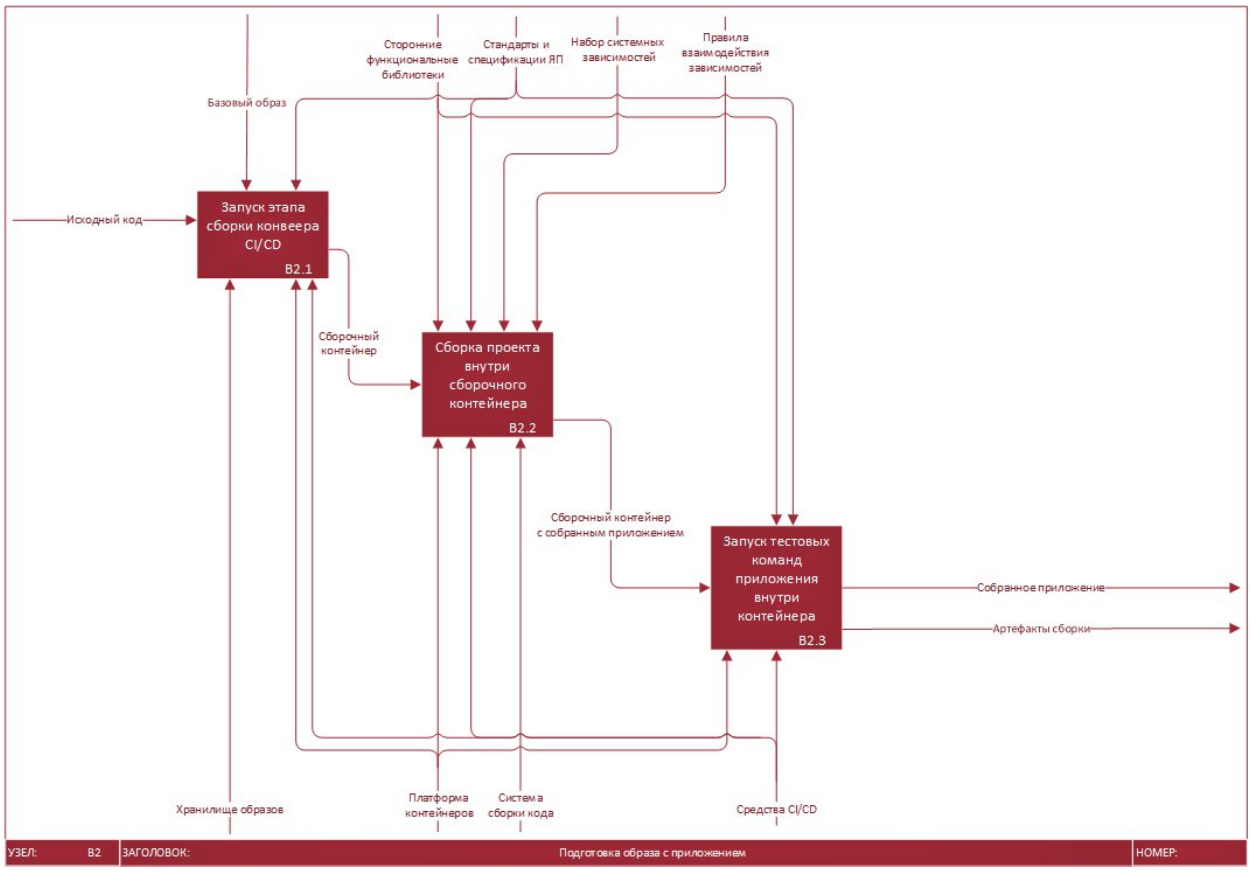

Рисунок 3.4. Декомпозиция процесса *B*2 «Подготовка образа с приложением»

Примечательным является процесс *B*2.2 «Сборка проекта внутри сборочного контейнера», в рамках которого существуют неочевидные требования к написанию конфигурации конвейера (см. рис. 3.5).

Выполнение происходит автоматически без участия разработчика, средствами *CI/CD* и в контейнеризированной среде. Запуск среды и копирование исходного кода в контейнер происходит в предыдущем процессе – *B*2.1.

Авторизация в хранилище образов выносится отдельным шагом и выполняется заранее – до выполнения основного скрипта, это позволяет сразу же определить доступность хранилища образов с *Docker* на узле исполнителя, а также получить доступ к ранее собранным образам проекта (если сборка производилась ранее), и права доступа разработчика, совершившего фиксацию.

Для выполнения сборки могут потребоваться специализированные переменные окружения, описанные в файле *gitlab-ci.yaml*, внешних модулях, или в переменных *CI/CD Gitlab*.

Данные авторизации для других сервисов, репозиториев не должны храниться в итоговой версии образа – это противоречит требованиям безопасности лаборатории, но на этапе сборки могут использоваться. Для решения этой проблемы применяется *Docker secret* – секреты в докер монтируются и существуют в окружении движка *Docker* только на время исполнения команды *docker build*, в конечный образ данные не перейдут (как гласит официальная документация – не «выпекутся»). Этот функционал встроен в *Docker* по умолчанию, не требует установки дополнительных пакетов или стороннего ПО, активируется при написании одной строчки в *Dockerfile*. Его функций хватает для достижения поставленной перед ним цели.

Docker поддерживает ускорение сборки нового образа с помощью кэша, состоящего из одинаковых слоев собранных ранее образов, при наличии такового. В команде сборки это прописывается отдельным флагом “*--cache-from* «ИМЯ_ОБРАЗА»”. Использование кэша позволяет сократить время.

Сборка приложения внутри образа осуществляется за счет *bash, sh, ps* скриптов. Если скриптов несколько, то важно написать так называемый «оберточный» скрипт, где будет присутствовать вызов всех имеющихся скриптов сборки приложения – такой подход существенно облегчает процесс отладки и поиска неисправностей в коде, решает проблему с видимостью переменных окружения. По успешному (в случае появлении какой-либо ошибки при выполнении скрипта, конвейер завершит свое выполнение со статусом «Неудачно») окончанию скриптов, образу будет присвоен тег. Тег формируется на основе динамических данных (в настоящий момент выбор единых данных находится в процессе обсуждения): имя ветки, *ID* коммита, *ID* задачи; и объявляется заранее как переменная конвейера по единому шаблону. Тег присвоен, теперь образ будет отправлен в реестр контейнеров, в котором на этом шаге *Docker* уже авторизован. Такой составной тег позволит наглядно отслеживать, в какой ветке были внесены изменений и какой коммит породил новый образ. Пользователю *Gitlab* будет доступна такая информация в графическом интерфейсе или при работе с терминалом, при условии достаточных прав доступа и авторизации в системе *git.jinr.ru.*

Все артефакты (*Gitlab artifacts*) отправляются в хранилище артефактов проекта конвейером автоматически. Этот шаг не требует дополнительной настройки и активирован по умолчанию администратором экземпляра *Gitlab*. В качестве артефактов выступают логи всего конвейера, документация, переменные, результаты сборки.

Рисунок 3.5. Диаграмма деятельности процесса *B2.2* «Сборка проекта внутри сборочного контейнера»

Если сборка прошла успешно, то необходимо перейти к проверке работоспособности основных модулей, а также устойчивости к нагрузкам (в том случае,

если приложение предполагается как *online*-решение, например, с постоянно развернутой БД). Важно понимать, что тестирование выявляет не только ошибки в коде проекта, но и ошибки, которые могут возникнуть в процессе эксплуатации в окружении конечного контейнера, из-за неправильно сконфигурированного процесса сборки. Именно поэтому тестирование происходит после появления образа с приложением.

Тестирование

Процесс тестирования (см. рис. 3.6) при использовании средств автоматизации заметно упрощается. *Unit* и интеграционные тесты можно провести в контейнерной среде, выгрузив с хранилища образов уже имеющийся образ приложения, запустив контейнер и вызвав необходимые скрипты и функции. Нагрузочное тестирование сопряжено с дополнительными сложностями – для тестирования необходимы сервисы, расположенные на других узлах сетевой инфраструктуры ОИЯИ или во внешней сети. Для их запуска, исполнитель конвейера *Gitlab* автоматически отправляет запрос и данные авторизации по заранее прописанному скрипту.

Запуск нового этапа – «Тестирование» никак не отличается от запусков других этапов конвейера. Для выполнения тестирования под каждый вид тестов запускается свой контейнер в отдельной задаче. Такой подход позволяет:

- избежать конфликтов зависимостей,
- настроить конфигурацию каждого отдельного вида тестирования или контейнера без оглядки на остальные тесты;
- упростить поиск ошибок и их исправление;
- сокращает время на проведение тестирования, ведь тесты запускаются параллельно и независимо, а не друг за другом.

Выполнение всех задач происходит параллельно и независимо друг от друга в отдельных контейнерных средах. Успешное окончание этапа тестирования возможно только после успешного выполнения всех задач (см. рис. 3.7).

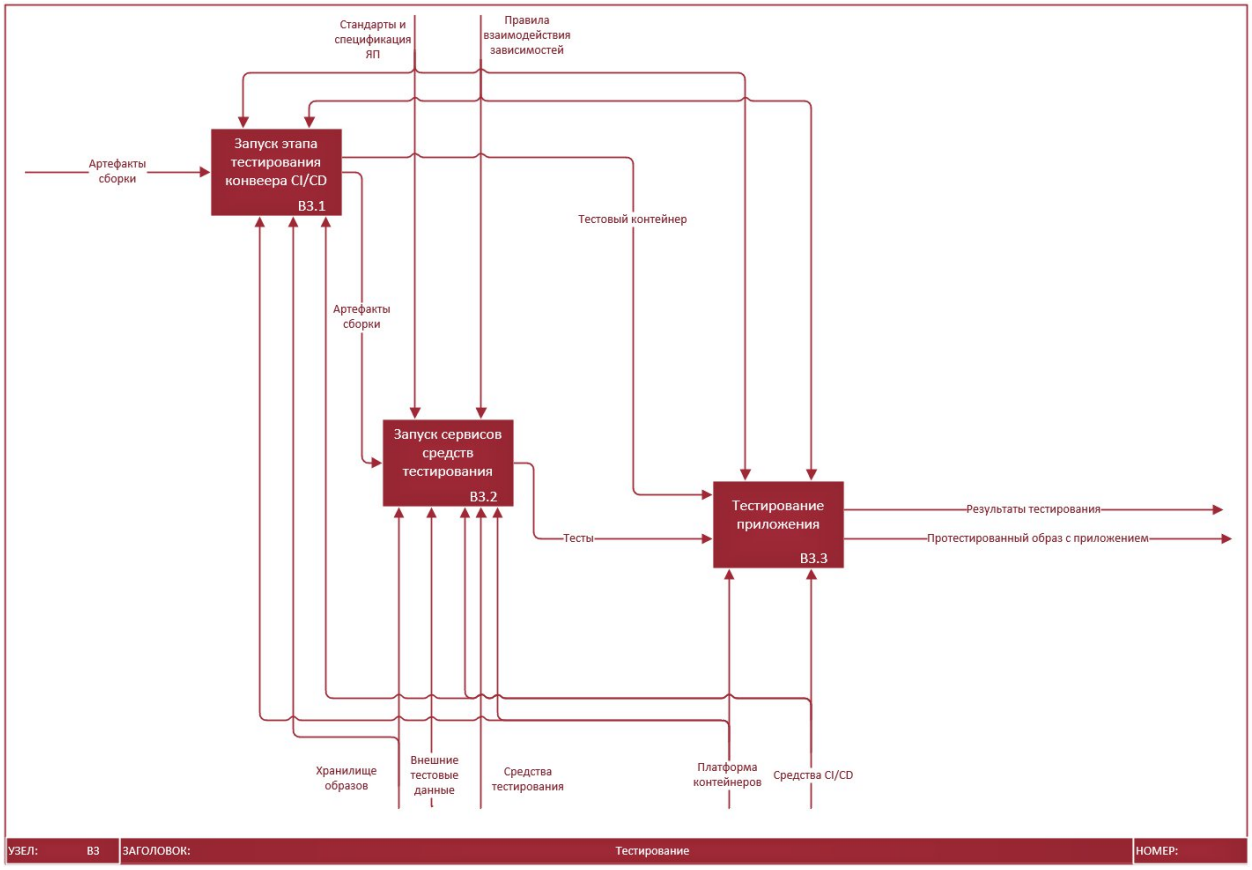

Рисунок 3.6. Декомпозиция процесса *B*3 «Тестирование»

В каждом тестовом контейнере исполнитель будет загружать образ приложения из реестра контейнеров, устанавливать (при необходимости) ПО для тестирования, запускать контейнер и выполнять только определенные скрипты или наборы команд на основе артефактов с предыдущего этапа.

Gitlab поддерживает интеграцию с наиболее часто использующимися средствами тестирования: *pytest*, *GoogleTest* и др. В интерфейсе содержится отдельный раздел с результатами запущенных в рамках конвейера тестов, в котором содержится расшифровка результатов и дополнительная информация по каждому выполненному скрипту. Чтобы воспользоваться функционалом, требуется в явном виде прописать в файле *gitlab-ci.yaml* в разделе артефактов путь до папки с результатами тестов внутри контейнера.

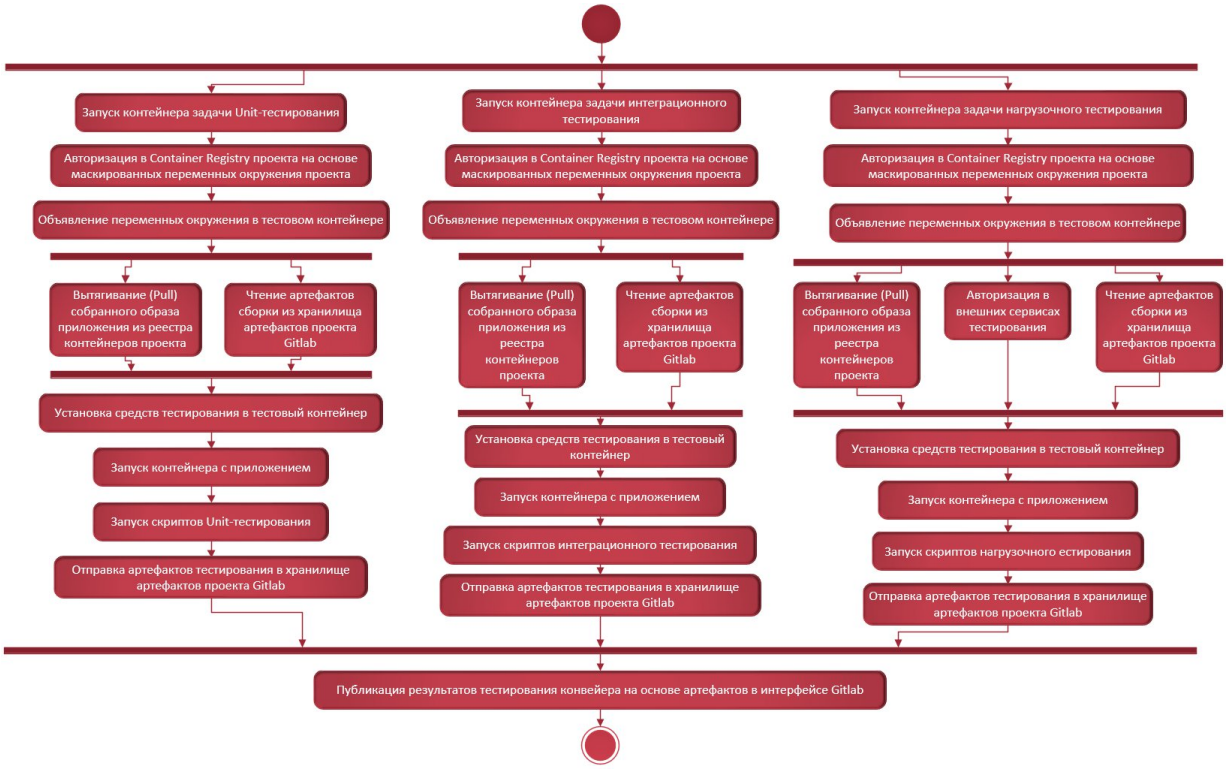

Рисунок 3.7. Диаграмма деятельности процесса *B*3.3 «Тестирование приложения»

Образ уже был опубликован в реестре контейнеров на этапе сборки, следующий за тестированием этап публикации, предполагает присваивание нового тега уже существующему образу в реестре контейнеров и конвертацию образа *Docker* в формат *SIF* для *Apptainer*.

Публикация образа

Этап публикации (см. рис. 3.8) не предполагается к выполнению после каждого совершенного коммита. Запуск этапа происходит при удачном завершении *Merge-request* в *develop* ветку. Под публикацией в данном случае предполагается размещение образа с определенным тегом, который используется разработчиками для последующей отладки всех происходящих за рамками конвейера *CI/CD* процессов. Именно единое наименование образа, который предполагается как доступный за пределами одного проекта, позволяет избежать проблем с постоянной сменой конфигурации сторонних сервисов, применяемых в разработке, или других средств автоматизации (сервисы оркестрации, развертывания, мониторинга).

Первым шагом выполняется объявление параметра «*git-strategy: none*», в таком случае, исполнитель не будет монтировать репозиторий кода в контейнерную среду. Нет смысла копировать исходный код проекта в контейнер, если никаких действий с кодом производится не будет, а все изменения касаются только уже имеющегося образа с приложением.

Следующим шагом выполняется установка *Apptainer*, с помощью которого будет произведена конвертация образа. Файл *SIF* будет отправлен в *CVMFS*, а образ с новым тегом – снова в реестр контейнеров.

Затем, авторизация в реестре контейнеров и *CVMFS*, объявление переменных, вытягивание уже существующего протестированного образа на основе переменной «ИМЯ_ОБРАЗА», о которой уже шла речь ранее и «тегирование». Новый тег присваивается уже на основе только имени ветки, потому как работа с новым функционалом на данном этапе разработки завершена. Далее запускается конвертация в *SIF* и отправка в *Container registry* и *CVMFS*. Такой подход соответствует советам по безопасности авторов *Gitlab*, где считается небезопасным использование тега *latest*, и позволяет избежать путаницы с уже имеющимися образами в реестре, ведь после успешного слияния функциональной ветки в *develop*, все предыдущие созданные образы в рамках работы над этой веткой остаются, а необходимость в нескольких отдельно обозначенных разными версиями того же функционала приложения отпадает.

Чтобы избежать хранения большого количества образов и их дубликатов в реестре контейнеров, предполагается удаление последнего успешно собранного и протестированного с тегом на основе переменной «ИМЯ_ОБРАЗА». Похожего результата можно добиться после настройки автоудаления образов из реестра, однако, в случае заморозки разработки какого-то проекта, потребуется дополнительно отслеживать настроенные параметры автоудаления, что не слишком разумно.

Рисунок 3.8. Диаграмма деятельности процесса *B*4 «Публикация образа»

Глава 4. Внедрение

В рамках работы разработанная методика, проанализированные программные средства активно внедряются и проходят апробацию в действующих проектах: *SPD Metadata Database, SPD Gaudi Framework, SPD Sampo Framework, SPDRoot Framework.*

SPD Metadata Database

Для обработки экспериментальных данных требуется широкий спектр вспомогательной информации. Для успешной работы установки требуется контроль большого числа параметров установки и систем обработки данных.

Именно для хранения этой вспомогательной информации разрабатывается база данных физических метаданных (*SPD Metadata Database*). Она будет развернута на серверах вычислительного комплекса ОИЯИ в постоянном онлайн режиме и будет содержать информацию о:

- наборах данных (датасетах), включая наследование и ссылки на конфигурации задач, которые создали эти датасеты;
- версии ПО, используемые при обработке данных;
- сечения и конфигурации, используемые для моделирования;
- информация о параметрах пучка (светимость, поляризация);
- настройки онлайн фильтра.

Большая часть информации будет поступать из других ИС эксперимента, анализа логов и конфигурационных файлов. Номенклатуру датасетов, параметров «*Run*», спецификации версий ПО и конфигураций МС еще предстоит разработать.

В процессе проектирования компонент ИС разработчиками был выбран подход создания микросервисной архитектуры, когда каждый сервис, представляет из себя небольшое приложение, связанное с остальными с помощью *API*-интерфейсов.

Каждый такой сервис был помещен разработчиками в *Docker*-контейнер со своим набором зависимостей и ресурсов.

В отличие от других рассмотренных в работе приложений, БД метаданных – это микросервисная архитектура, где будет существовать большое число экземпляров таблиц, представляющих из себя отдельные контейнеры. В связи с чем при дальнейшем внедрении средств автоматизации предполагается использовать системы оркестрации, балансировки нагрузки и мониторинга.

К настоящему моменту выполнено следующее:

- Внедрен подход в написании кода, описанный в рамках разработанной методики.
- Настроен реестр *Container Registry*.
- Реализован работающий конвейер на базе *Gitlab CI/CD*, включающий автоматическую:
    - сборку *Docke*r-образов микросервисов;

- выгрузку образов *Docker* в реестр *Container Registry* проекта;
- тестирование образов;
- развертывание собранных образов из реестра на выделенный сервер;
- публикацию документации на проектном домене экземпляра *git.jinr*.

- Поднят и актуализируется сайт с документацией на основе *Sphinx* на тестовом репозитории.

FairRoot Framework и SPDRoot Framework

Фреймворк *SPDRoot* реализован на базе *FairSoft FairRoot*, который в свою очередь реализован на базе фреймворка *ROOT*. Согласно официальному описанию фреймворка, *FairRoot* – это платформа моделирования, реконструкции и анализа, основанная на системе *ROOT*. Пользователь может создавать смоделированные данные и/или выполнять анализ с помощью той же платформы. Поддерживаются движки *Geant*3 и *Geant*4, однако пользовательский код, который создает имитационные данные, не зависит от конкретного движка Монте-Карло. Платформа предоставляет базовые классы, которые позволяют пользователям простым способом создавать свои детекторы и /или задачи анализа, а также предоставляет некоторые общие функциональные возможности, такие как визуализация треков. Кроме того, реализован интерфейс для чтения карт магнитного поля [22].

SpdRoot — это программный пакет для моделирования работы детектора *SPD* на основе фреймворка *FairRoot*. Он включает в себя генераторы событий (*Pythia*6, *Pythia*8, *FTF*), описание геометрии установки *SPD, Geant*4 для моделирования прохождения частиц через вещество, а также средства реконструкции и анализа физических процессов.

Базовый образ основывается на ПО, разработанном третьими лицами, соответственно, важно поддерживать состояние чужого кода в работоспособном состоянии и исправлять ошибки, возникающие при работе над *SPD Root*. Код был скопирован в отдельный репозиторий кода *Gitlab* ОИЯИ, исправлен в необходимых для работы *SPD Root* зафиксированных заранее версиях. Под разработку базового образа создан отдельный репозиторий.

Для функционирования *FairRoot* требуется наличие пакета ПО под названием *FairSoft*. Соответственно, для последующей разработки *SPDRoot*, разработанный и реализованный базовый образ «*fair-group-image*» содержит в себе пакет установленного ПО *FairSoft* и фреймворк *FairRoot*. Последующая разработка ведется на основе данного образа.

К настоящему моменту выполнено следующее:

- Созданы отдельные репозитории для исходного кода фреймворка *FairRoot* и базового образа на его основе.
- В репозиториях внедрен подход в написании кода, описанный в рамках разработанной методики.
- Настроено хранилище образов *Container Registry*.

- Реализованы работающие конвейеры *CI/CD*, включающие в себя автоматическую:
    - Сборку фреймворка *FairRoot* внутри базового образа *fair-group-image;*
    - Отправку собранного образа в хранилище *Container Registry* репозитория *fair-group-image*;
    - Сборку проекта *SPD Root* и образа на основе базового образа *fair-group-image;*
    - Отправку собранного образа в хранилище *Container Registry* репозитория *SPD Root* и в хранилище *CVMFS SPD Root.*

SPD Gaudi Framework и Sampo Framework

Gaudi – это объектно-ориентированный фреймворк, содержащий все необходимые интерфейсы и компоненты для написания фреймворков для экспериментов в области физики высоких энергий [23]. Первоначально *Gaudi* был разработан для внутренних нужд коллаборации *LHCb*, но вскоре после того, как к разработке присоединилась коллаборация *ATLAS*, стало ясно, что пакет можно легко трансформировать для любого другого эксперимента. Надежность упаковки подтверждается ее использованием в многочисленных совместных проектах по всему миру.

Sampo это программная платформа для обработки данных эксперимента *SPD* на базе фреймворка *Gaudi*. С технической точки зрения *Sampo* представляет собой набор модулей, управление которыми осуществляется ядром *Gaudi*. В то время как *Gaudi* является эксперименто-независимым, модули *Sampo* привязаны к эксперименту *SPD*.

К настоящему моменту выполнено следующее:

- Созданы отдельные репозитории для исходного кода фреймворка *Gaudi* и базового образа на его основе;
- В репозиториях внедрен подход в написании кода, описанный в рамках разработанной методики.
- Настроено хранилище образов *Container Registry*.
- Реализованы работающие конвейеры *CI/CD*, включающие в себя автоматическую:
    - Сборку фреймворка *Gaudi* и сборку базового образа на его основе*;*
    - Отправку собранного образа в хранилище *Container Registry* репозитория *Gaudi*;
    - Сборку проекта и образа *Sampo* на основе базового образа *Gaudi;*
    - Отправку собранного образа в хранилище *Container Registry* репозитория *Sampo* и в хранилище *CVMFS Sampo.*
- Разрабатывается и тестируется этап развертывания контейнера *Sampo* с помощью движка контейнеров *Apptainer* в среде *CVMFS*.

Анализ полученных результатов

Рассмотренные проекты имеют свои определенные особенности и различны по своей архитектуре. Когда дело доходит до приложения к проектам, то всегда стоит учитывать нюансы практической реализации. Цель работы предполагала не продумывание всех деталей, а проектирование общей методики без жесткой конкретики. Кроме того, методику планируется дополнять, расширять и корректировать в зависимости от полученного опыта, обратной связи разработчиков.

В среднем, по словам программистов, процесс подготовки к публикации (это все те действия, что не относятся к программной и аппаратной сборке приложения, сборке образа, запуска тестов и публикации, например – подготовка рабочего окружения, работа с графическим интерфейсом, написание команд, написание коммита и так далее) и публикация образа (это программные и аппаратные действия по непосредственной отправке в реестр) занимал 15-20 минут при выполнении всех действий вручную или с подключенными модулями, без учета времени сборки.

Настроенные программные средства позволили упростить подготовку до авторизации пользователя в *Gitlab* ОИЯИ и написания коммита. Сама же публикация образа осуществляется в удаленной среде – на серверах ОИЯИ, а не на устройстве разработчика, соответственно, высвобождаются занятые ресурсы, и он может приступить к работе над иными задачами без потери времени на ожидание. Таким образом, удалось сократить время на подготовку до, максимум, 5 минут без учета времени сборки.

Почему не учитывается время сборки приложения и образа? Это обусловлено тем, что ее скорость зависит от аппаратных ресурсов устройства, где она осуществляется, особенно, если речь идет про приложения объемом более 10 ГБ.

В случае с внедрением методики в процесс разработки фреймворка *Sampo* и БД *Metadata*, время выполнения всего конвейера *CI*/*CD*, включающего сборку и публикацию, и идентичных команд, выполненных ручным запуском на устройствах программистов, практически не отличается и составляет не более 5 минут (см. рис. 4.1) и не более 6 минут (см. рис. 4.2), соответственно. Оба этих проекта в собранном виде занимают не более 10 Гбайт: по 800 Мбайт занимает каждый микросервис-таблица БД *Metadata* (их 4 штуки) и ~9 Гбайт фреймворк *Sampo*.

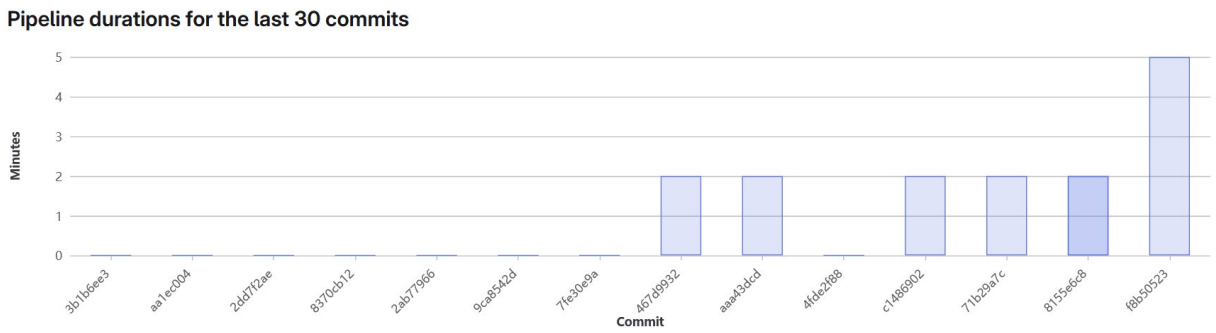

Рисунок 4.1. График времени выполнения последних 30 конвейеров для проекта *Sampo*, доступный в интерфейсе *Gitlab* ОИЯИ (*git.jinr.ru*)

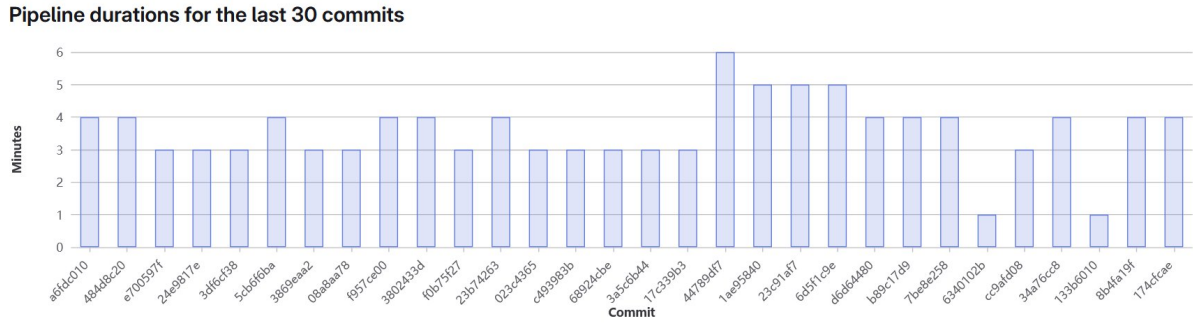

Рисунок 4.2. График времени выполнения последних 30 конвейеров для проекта *SPD Metadata Database*, доступный в интерфейсе *Gitlab* ОИЯИ (*git.jinr.ru*)

Иная ситуация наблюдается для базовых образов-фреймворков *Gaudi* и *FairRoot*. Сборка каждого из них занимает более 2 часов (см. рис. 4.3, 4.4). Такая длительность складывается из-за необходимости установки всех основных, дополнительных зависимостей, сторонних пакетов и библиотек, а также ограничения ресурсов каждого исполнителя конвейера на серверах ОИЯИ. Активной разработки над ними не ведется, сборка осуществляется единожды на основе фиксированной версии, которая будет корректироваться крайне редко. Таким образом, с применением методики каждый раз при коммите в репозитории проектов *Sampo* и *SPDRoot* экономится более 2 часов на подготовку всего окружения контейнеризированной среды, ведь она выполнена заранее.

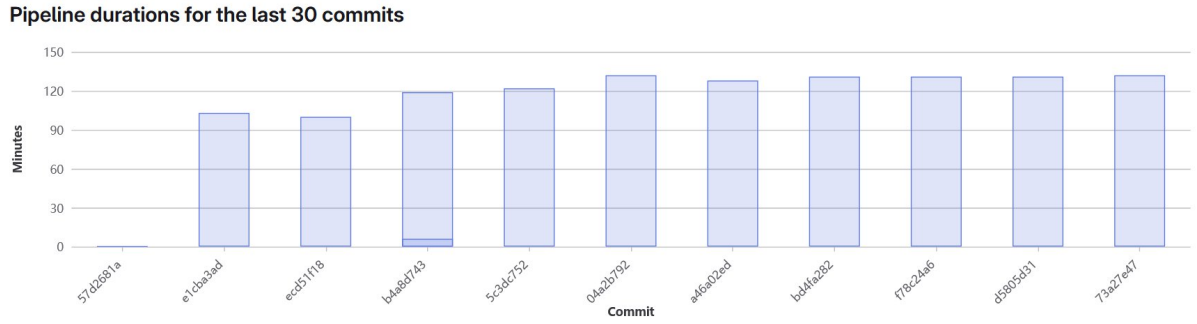

Рисунок 4.3. График времени выполнения последних 30 конвейеров для проекта *Gaudi*, доступный в интерфейсе *Gitlab* ОИЯИ (*git.jinr.ru*)

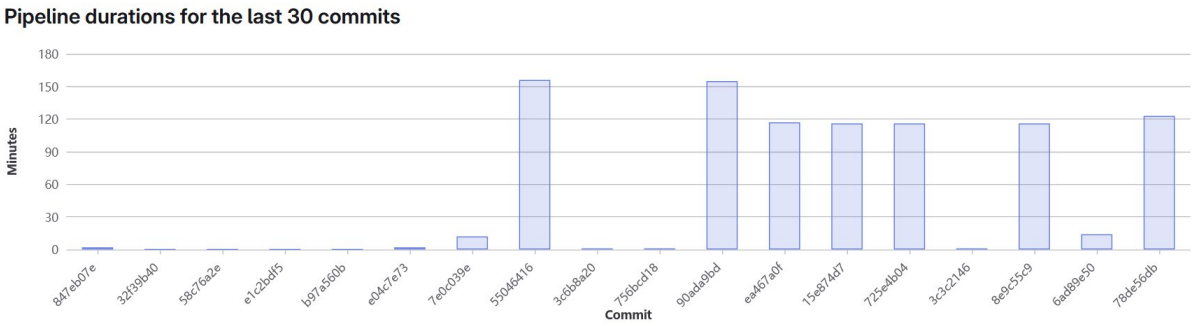

Рисунок 4.4. График времени выполнения последних 30 конвейеров для проекта *fair-group-image*, доступный в интерфейсе *Gitlab* ОИЯИ (*git.jinr.ru*)

Заключение

Работа была посвящена разработке программных средств автоматизации рутинных процессов и проектированию методики их использования. Спроектированная методика и настроенные программные средства позволяют сократить время на сборку, тестирование и публикацию образов приложений, способны выполнять поставленные перед ними задачи в полном объеме. Существующая конфигурация основана на международном опыте и может быть сохранена как стандартизированные шаблоны для применения в работе над иными проектами, будь то в рамках коллаборации *SPD* или в рамках других экспериментов ОИЯИ, так как они поддаются гибкой настройке под любой масштаб задач. Внесение изменений в описанные средства может быть осуществлено в любой момент при условии наличия доступа к системе контроля версий *Gitlab* ОИЯИ и соответствующих прав на редактирование. Методика сохраняет актуальность в условиях многообразия проектов за счет своей гибкости.

Результаты работы над внедрением разработанных средств автоматизации для проекта *SPD Metadata Database* были представлены и защищены на весенней «Школе по информационный технологиям ОИЯИ» 2024 года [24], а доклад был отмечен как один из лучших [25].

Материалы магистерской работы приняты для участия в 11 Международной Конференции «Распределенные вычисления и грид-технологии в науке и образовании» (*GRID'*2025) [26].

Проделанная работа имеет дальнейшее развитие, актуальным является продолжение внедрения методики и программных средств для коллаборации *SPD*, корректировка, в зависимости от поставленных задач, расширение области применения, обучение персонала лабораторий работе в рамках методики, в общем и целом, осуществление информационной, информационно-технической поддержки экспериментов. Разработанный программный код размещен в репозиториях *Gitlab* ОИЯИ [27].

Список сокращений и условных обозначений

1. Модель *AS IS* («Как есть) – это модель существующего состояния. Она описывает существующие процессы, то, как они работают и взаимодействуют друг с другом здесь и сейчас. Эта модель позволяет оценить текущее состояние процессов, будь то организационные бизнес-процессы или технологические системные.
2. Модель *TO BE* («Как должно быть») – это модель для описания процессов. Она помогает описать идеальное или улучшенное состояние системы. Чаще всего эту модель используют в связке с моделью *AS IS*. С помощью *AS IS* фиксируют, как устроена работа в системе. А в *TO BE* отражают, как устранить недостатки, оптимизировать процессы и внедрить улучшения.
3. Диаграмма деятельности (*Activity diagram*) – *UML*-диаграмма, на которой показаны действия, состояния которых описаны на диаграммах состояний. Под деятельностью (*activity*) понимается спецификация исполняемого поведения в виде координированного последовательного и параллельного выполнения подчинённых элементов — вложенных видов деятельности и отдельных действий, соединённых между собой потоками, которые идут от выходов одного узла ко входам другого. Отображает динамические аспекты поведения системы. Эта диаграмма представляет собой блок-схему, которая наглядно показывает, как поток управления переходит от одной деятельности к другой.
4. *Docker*-образ (*Docker Image*) – это неизменяемый файл, содержащий исходный код, библиотеки, зависимости, инструменты и другие файлы, необходимые для запуска приложения. Образ — это шаблон, на основе которого создается контейнер, существует отдельно и не может быть изменен. При запуске контейнерной среды внутри контейнера создается копия файловой системы (*docker* образа) для чтения и записи.
5. Контейнер *Docker* (*Docker Container*) – это виртуализированная среда выполнения, в которой пользователи могут изолировать приложения от хостовой системы. Эти контейнеры представляют собой компактные портативные хосты, в которых можно быстро и легко запустить приложение.
6. Базовый *Docker*-образ – это образ, на основе которого будет создан сборочный *Docker*-образ.
7. Сборочный *Docker*-образ – это образ, основанный на Базовом *Docker*-образе, в котором настроено окружение для разработки приложения (установлены необходимые зависимости, библиотеки) и в котором ведется разработка на данный момент.
8. *Docker*-образ приложения – это образ, основанный на сборочном *Docker*-образе в котором уже собрано приложение, на основе этого образа можно разворачивать контейнер с приложением.

9. Приложение (программный продукт) – это разрабатываемое ПО для распределенной вычислительной системы эксперимента *SPD*.
10. Платформа контейнеров — это набор программ, позволяющих взаимодействовать с образами и контейнерами: передавать или забирать из дистанционного хранилища, запускать и управлять ими.
11. *Singularity/Apptainer* – платформа контейнеров, одним из основных применений которой является внедрение контейнеров, воспроизводимость в мир научных и высокопроизводительных вычислений [17]. Является целевой платформой для ОИЯИ из-за своей направленности в научной области применения. Имеет полную совместимость с *Docker*-образами и контейнерами.
12. *Docker Daemon* – платформа контейнеров, позволяющих взаимодействовать с *Docker*-образами и контейнерами. Состоит из *Docker Engine*, *Docker Client, Docker Host, Docker Daemon*.
13. Исходный код – это разрабатываемый или разработанный код Приложения.
14. Сторонние функциональные библиотеки (библиотеки) – это набор готовых функций, классов и объектов для решения конкретных задач, написанный третьими лицами или самими разработчиками.
15. Системные зависимости – это зависимость между частями системы, когда систему рассматривают как единое целое, а также библиотеки, которые зависят от других системных библиотек во время компиляции. Например, *CUDA*, *FFMPEG*.
16. Тестовые данные – это наборы входных данных или информации, используемые для проверки корректности, производительности и надёжности программных систем.
17. Внутренние тестовые данные – это тестовые данные, определяемые на уровне исходного программного кода проекта.
18. Внешние тестовые данные – это тестовые данные, определяемые на уровне сторонних сервисов.
19. Целевое устройство/среда – это виртуальная и физическая среда и окружение целевого устройства, где планируется развертывание приложения.
20. РСОД – Распределённая Система Обработки Данных.
21. Система контроля версий (*Gitlab*) – это веб-платформа для управления проектами и репозиториями программного кода, работа которой основана на системе контроля версий (*Git*). Все дистанционные репозиторий для разрабатываемых приложений располагаются на экземпляре *Gitlab* ОИЯИ – *git.jinr.ru*.
22. *Dockerfile* – это текстовый документ, содержащий инструкции для генерации образа *Docker*. Он указывает *Docker*, как построить образ, который будет использоваться при создании контейнера.
23. Средства тестирования (ПО для тестирования) – вручную прописанные разработчиками приложения или сторонние готовые решения в виде функций,

скриптов или ПО для тестирования приложения и Docker-контейнера с приложением, а также физический или виртуально выделенный сервер, для проведения тестирования.

24. Системы сборки кода – это ПО для компиляции и сборки кода. Например, *CMake + Make*.
25. Хранилище образов – виртуальное или физическое хранилище данных формата *Docker* – образов.
26. Разработчик – лицо, осуществляющее разработку приложения.
27. Пользователь – системный администратор и/или другой разработчик
28. Лидер бригады разработки – это лицо, деятельностью которого является управление процессом разработки программного продукта и в зависимости от того, как организована разработка в конкретном случае может являться как старший программист, как менеджер, так и ведущий программист.
29. *Docker-secret* – технология *Docker*, позволяющая передать данные внутрь образа в виде переменных или файлов на момент сборки, не оставляя эти данные в собранном образе.
30. Чувствительные данные – это конфиденциальная информация, которая требует особой защиты из-за своей ценности и потенциальной уязвимости, например – данные авторизации.
31. Коммит (Фиксация) – команда в системе контроля версий, снимок проекта в определённый момент.
32. Исполнитель конвейеров – *Gitlab Runner*.
33. Merge-request – это запрос на слияние изменений из одной ветки в другую.
34. Ревью (Проверка качества кода) – процесс проверки, анализа кода более опытным программистом с целью выявления и исправления ошибок.
35. Review-request – запрос на ревью. В *Gitlab* существует специальный функционал, позволяющий отправлять запрос на обзор изменений в коде во время выполнения *Merge-Request.*
36. Ревьювер – опытный программист, назначенный лидером бригады разработки для выполнения ревью, им может быть дублер ведущего программиста, а может быть тестер (тестировщик) из отдела качества.
37. Контейнер задачи – контейнер *Docker*, в котором выполняется описанная в файле *gitlab-ci.yaml* задача (*job*) этапа (*stage*) конвейера *CI/CD*. Он существует до тех пор, пока выполняются инструкции в задаче, как только выполняется последняя инструкция – контейнер выключается и удаляется из памяти системы.
38. *Unit*-тестирование (Модульное тестирование) – это метод тестирования программного обеспечения, который направлен на проверку отдельных модулей или компонентов программы. Модуль в этом контексте представляет собой

небольшую, изолированную часть программы, такую как функция, метод, класс или объект, которая выполняет определённую задачу или функцию в приложении. Цель модульного тестирования — убеждение в том, что каждый модуль работает правильно и соответствует спецификациям.

39. Интеграционные тесты – это тесты программного обеспечения, цель которых – выявить дефекты во взаимодействии между различными частями системы, при выполнении интеграционного тестирования программные модули объединяются логически и тестируются как группа.
40. Нагрузочное тестирование – это метод тестирования программного обеспечения, который направлен на проверку устойчивости к нагрузкам: как приложение или сервер реагируют на большое количество запросов, обрабатывают данные и поддерживают стабильную работу.
41. Тестовый контейнер – контейнер *Docker* с приложением, тестовыми данными и средствами тестирования, в котором производится тестирование приложения.
42. Маскированные переменные окружения проекта – это переменные, объявленные в настройках *CI/CD* проекта *Gitlab*, значение которых скрыто от всех пользователей любого уровня доступа. Значение такой переменной может знать только пользователь, объявивший переменную, если запоминал/сохранял его где-то еще.

Список источников

1. International spin physics collaboration at the collider NICA: сайт / Объединенный Институт Ядерных Исследований. – Дубна. – Обновляется в течение суток. – URL: http://spd.jinr.ru (дата обращения: 05.03.2024). – Текст: электронный.
2. ATLAS Collaboration. Software and computing for Run 3 of the ATLAS experiment at the LHC / D. Hehir, J. Khubua, M. Lokajicek [et al.]. – Текст: непосредственный // European Physics Journal C. – 2025. – Vol. 85. – № 234. – DOI: 10.1140/epjc/s10052-024-13701-w.
3. Agile-манифест разработки программного обеспечения: [манифест гибкой разработки программного обеспечения]: [сайт]. – 2001 –. – URL: https://agilemanifesto.org/iso/ru/manifesto.html (дата обращения: 20.03.2024). – Текст: электронный.
4. **Luz, W. P.** Adopting DevOps in the real world: A theory, a model, and a case study /W. P. Luz, G. Pinto, R. Bonifacio. – Текст: электронный // Journal of Systems and Software. – 2019. – Vol. 157. – N. 110384. – DOI: 10.1016/j.jss.2019.07.083. – URL: https://www.researchgate.net/publication/334653014_Adopting_DevOps_in_the_Real_World_A_Theory_a_Model_and_a_Case_Study (дата обращения: 10.03.2024).
5. **Decan, A.** What Do Package Dependencies Tell Us About Semantic Versioning? / A. Decan, T. Mens. – Текст: непосредственный // IEEE Transactions on Software Engineering. – 2020. – Vol. 47. – № 6. – P. 1226-1240. – DOI: 10.1109/TSE.2019.2918315.
6. **Бузаков П.** GitFlow процесс / П. Бузаков. – Текст: электронный // Хабр: [веб-проект]. – 2023. – URL: https://habr.com/ru/articles/767424/ (дата обращения: 20.03.2024).
7. **Хомоненко, А. Д.** Использование продвинутых функций Git при разработке программного обеспечения / А. Д. Хомоненко, Е. Н. Каратаев. – Текст: электронный // Информационные технологии и телекоммуникации. – 2024. – № 2 (38). – С. 37–48. – DOI: 10.20295/2413-2527-2024-238-37-48. – URL: https://itt-pgups.ru/en/storage/viewWindow/161617 (дата обращения: 20.01.2025).
8. **Safwan, K. A.** Developers’ need for the rationale of code commits: An in-breadth and in-depth study / K. A. Safwan, M. Elarnaoty, F. Servant. – Текст: непосредственный // Journal of Systems and Software. – 2022. – Vol. 189 – № 4. – N. 111320. – DOI: 10.1016/j.jss.2022.111320.
9. Соглашение о коммитах: [Простое соглашение о том, как нужно писать сообщения коммитов]: [сайт]. – Обновляется в течение суток. – URL: https://www.conventionalcommits.org/ru/v1.0.0/ (дата обращения: 20.03.2024). – Текст: электронный.

10. Семантическое Версионирование 2.0.0: [спецификация]: [сайт] /авторство спецификации T. Preston-Werner. – 2013. – URL: https://semver.org/lang/ru/ (дата обращения: 20.03.2024). – Текст: электронный.
11. **Mahajan, A. A** Drift from Versioning to Semantic Versioning / A. Mahajan, P. Kaur. – Текст: непосредственный // Software Engineering and Technology. – 2015. – Vol. 7. – P. 187–190.
12. A Large-Scale Empirical Study on Semantic Versioning in Golang Ecosystem / L. Wenke, W. Feng, C. Fu, F. Zhou. – Текст: электронный // 38th IEEE/ACM International Conference on Automated Software Engineering (ASE): материалы 38-ой международной конференции (Люксембург, 11-15 сентября, 2023) – Люксембург, 2023. – ISSN 2643-1572. – DOI: 10.1109/ASE56229.2023.00140. – URL: https://arxiv.org/pdf/2309.02894 (дата обращения: 10.01.2025).
13. Which Commits Can Be CI Skipped? / R. Abdalkareem, S. Mujahid, E. Shihab, J. Rilling. – Текст: электронный // IEEE Transactions on Software Engineering. – 2019. – Vol. 99. – DOI: 10.1109/TSE.2019.2897300. – URL: https://www.researchgate.net/publication/330764420_Which_Commits_Can_Be_CI_Skipped (дата обращения: 15.09.2024).
14. Use GitLab: GitLab Docs: [сайт с документацией] / Gitlab Inc. – 2025 –. – Обновляется в течение суток. – URL: https://docs.gitlab.com/ee/user/ (дата обращения: 20.03.2024). – Текст: электронный.
15. **Иванчикова, К.** Руководство по *CI/CD* в GitLab для новичка / К. Иванчикова. – Текст: электронный // REG.RU: [сайт]. – 2024. – 26 июля. – URL: https://www.reg.ru/blog/rukovodstvo-po-ci-cd-v-gitlab-dlya-novichka/ (дата обращения: 20.01.2025).
16. **Narwal, R.** Implementation of DevOps using Git and Jenkins / R. Narwal, S. Goyal, K. Ahuja. – Текст: электронный // International Journal of Advanced Research in Science, Communication and Technology. – 2024. – Vol. 4 – P. 546-550. – ISSN 581-9429. – DOI: 10.48175/IJARSCT-17670. – URL: https://ijarsct.co.in/Paper17670.pdf (дата обращения: 11.02.2025).
17. Why use Apptainer? – Текст: электронный // Apptainer project: Apptainer User Guide: [сайт]. – URL: https://apptainer.org/docs/user/latest/introduction.html#why-use-apptainer (дата обращения: 20.03.2025).
18. CernVM-FS: Container Packaging and Transport Overview – Текст: электронный // Cern : CernVM File System Documentation : [сайт]. – URL: https://cvmfs.readthedocs.io/en/ stable/cpt-overview.html (дата обращения: 20.03.2025).
19. CernVM-FS: Containers – Текст: электронный // Cern : CernVM File System Documentation : [сайт]. – URL: https://cvmfs.readthedocs.io/en/stable/cpt-containers.html (дата обращения: 20.03.2025).

20. Docker Hub: [официальный сайт]. / Docker Inc. – 2025. – URL: https://hub.docker.com/ (дата обращения: 20.03.2024). – Текст: электронный.
21. Quality and productivity outcomes relating to continuous integration in GitHub / B.Vasilescu, Y.Yu, H.Wang [et al.]. – Текст: непосредственный // ESEC/FSE'15: Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: материалы конференции (Бергамо, 30 авг. - 4 сент., 2015) – Бергамо, 2015. – P. 805-816. – ISBN 9781450336758. – DOI 10.1145/2786805.2786850.
22. FairRoot: [project storage] // GitHub: [web platform]. – GitHub, Inc., 2025. – URL: https://github.com/FairRootGroup/FairRoot (дата обращения: 08.01.2025).
23. Gaudi Project documentation: [сайт с документацией] / Cern. – 2019. – URL: https://gaudi-framework.readthedocs.io. – Текст: электронный.
24. **Короткин, Р. Н.** Развертывание компонент ИС метаданных эксперимента SPD : Список докладов : [презентация : материалы Весенней школы по информационным технологиям ОИЯИ, Дубна, 15-16 апреля 2024 г.] / Р. Н. Короткин. – Текст: электронный // Весенняя школа по информационным технологиям: [сайт]. – 2024. – URL: https://indico.jinr.ru/event/4432/contributions/25914/ (дата обращения: 15.05.2025).
25. Лучшие доклады. – Текст: электронный // Весенняя школа по информационным технологиям: [сайт]. – 2024. – URL: https://indico.jinr.ru/event/4432/page/1994 (дата обращения: 15.05.2025).
26. 11th International Conference «Distributed Computing and Grid Technologies in Science and Education» (GRID'2025): [сайт]. – Дубна, 2025. – URL: https://indico.jinr.ru/event/5170/overview (дата обращения 15.05.2025). – Текст: электронный.
27. SPD Software: [project storage] // Gitlab ОИЯИ: [web platform]. – ОИЯИ, 2025. – URL: https://git.jinr.ru/spd/. – Режим доступа: для авториз. пользователей.