ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
**«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»**
**ИНСТИТУТ КОМПЬЮТЕРНЫХ НАУК И КИБЕРБЕЗОПАСНОСТИ**
**ВЫСШАЯ ШКОЛА ПРОГРАММНОЙ ИНЖЕНЕРИИ**

**Отчет о прохождении производственной (научно-исследовательской работы) практики**

**на тему: «Разработка системы формирования корректирующих и предупредительных действий на основе изменений репозитория кода».**

Ильчук Александр Евгеньевич

*(Ф.И.О. обучающегося)*

4 курс, 5130903/10301

*(номер курса обучения и учебной группы)*

09.03.03 Прикладная информатика

*(направление подготовки (код и наименование)*

**Место прохождения практики:** ФГАОУ ВО «СПбПУ», ИКНК, ВШ ПИ.

*(указывается наименование профильной организации или наименование структурного подразделения*

г. Санкт-Петербург, ул. Обручевых, д. 1, лит. В

*ФГАОУ ВО «СПбПУ», фактический адрес)*

**Сроки практики:** с 02.10.2024 по 23.11.2024**.**

**Руководитель практической подготовки от ФГАОУ ВО «СПбПУ»:** Пархоменко Владимир Андреевич, программист 1 категории, ст. преподаватель ВШ ПИ.

**Консультант практической подготовки от ФГАОУ ВО «СПбПУ»:** нет

**Руководитель практической подготовки от профильной организации**: нет.

**Оценка:**

Руководитель практической подготовки
от ФГАОУ ВО «СПбПУ»: Пархоменко В.А.

Обучающийся: Ильчук А.Е.

Дата:

# СОДЕРЖАНИЕ

# ВВЕДЕНИЕ

**Актуальность исследования.** Современные разработки программного обеспечения (ПО) становятся всё более сложными, требуя высоких стандартов качества и точного контроля за процессом разработки. В условиях глобализации и быстрого развития технологий компании стремятся не только создавать качественные продукты, но и эффективно управлять процессом их создания. Однако существующие подходы к анализу и управлению качеством ПО требуют значительных временных и человеческих ресурсов. Методы анализа данных и машинного обучения, которые уже зарекомендовали себя в смежных областях, могут быть использованы для автоматизации контроля качества и выявления проблем на ранних этапах. Одной из ключевых задач в этой области является анализ данных из репозиториев исходного кода, таких как GitHub.

Коммиты в репозиториях содержат важную информацию о внесённых изменениях: количество добавленных и удалённых строк кода, изменённые файлы, временные интервалы между изменениями. Анализ этих данных позволяет выявить потенциальные отклонения от нормального процесса разработки и предложить корректирующие и предупреждающие действия (CAPA). Несмотря на широкий спектр существующих инструментов для анализа данных из репозиториев, большинство из них либо недостаточно автоматизированы, либо не позволяют выявлять комплексные закономерности в данных.

**Цель исследования:** разработка системы, которая позволит автоматизировать процесс анализа коммитов и извлечения CAPA на основе методов машинного обучения и кластеризации.

**Задачи исследования:**

- Провести обзор существующих методов анализа данных из репозиториев кода.
- Изучить применимость методов кластеризации и алгоритмов машинного обучения для анализа коммитов.
- Разработать систему для автоматического извлечения данных о коммитах из нескольких репозиториев GitHub.
- Реализовать механизм выявления аномалий и классификации коммитов на основе предложенных методов.
- Создать интерактивный дашборд для визуализации результатов анализа.
- Оценить эффективность предложенного подхода на реальных данных.

Подробнее актуальность исследования и обзор методов рассмотрены в разделе 1.1.

Таким образом, исследование направлено на решение задачи повышения эффективности управления качеством программного обеспечения за счёт использования современных технологий анализа данных. Предложенная система должна не только автоматизировать процесс анализа данных, но и предоставлять разработчикам полезные рекомендации для улучшения качества кода и предотвращения потенциальных проблем.

# ГЛАВА 1. ИССЛЕДОВАНИЕ МЕТОДОВ И СРЕДСТВ ФОРМИРОВАНИЯ CAPAS НА ОСНОВЕ ИЗМЕНЕНИЙ РЕПОЗИТОРИЯ КОДА

## 1.1. Современные подходы к формированию CAPA на основе анализа репозиториев кода

Корректирующие и предупреждающие действия (CAPA) представляют собой ключевой элемент систем управления качеством, направленных на выявление и устранение дефектов, а также предупреждение их повторного возникновения [11]. Изначально применяемые в регулируемых отраслях (медицина, фармацевтика и т.д.), концепции CAPA становятся актуальными и для разработки программного обеспечения, где цель сводится к автоматизации обнаружения проблем в коде и предложению мер по их исправлению или предотвращению. В контексте анализа изменений в репозиториях кода задача формирования CAPA сводится к систематическому анализу истории коммитов, метрик и паттернов изменений с целью выявления аномалий и генерации рекомендаций для разработчиков.

Существует несколько направлений и инструментов, применимых к этой задаче. Одни ориентированы на управление качеством как таковым (системы QMS и CAPA-менеджмента), другие – на технический анализ исходного кода и истории версий (статический анализ, JIT-предсказание дефектов, классификация коммитов). Современные подходы активно используют методы машинного обучения и искусственного интеллекта для выявления закономерностей в репозиториях и автоматической генерации CAPA [2, 13]. В этой главе рассматриваются существующие модели и инструменты, а также их сравнение по критериям автоматизации, интеграции в CI/CD, применимости к истории коммитов и интерпретируемости результатов.

## 1.2. Системы управления CAPA и интеграция в процессы разработки

В классических системах контроля качества (например, ISO 9001, FDA 21 CFR Part 820) CAPA оформляются документально и отслеживаются средствами QMS. Такие системы (Quality Management Systems) обеспечивают формализацию процесса: сбор инцидентов, расследование причин, планирование и выполнение действий, верификацию эффективности. Большинство коммерческих решений по CAPA (Qualityze, MasterControl, SimplerQMS и т.д.) предлагают удобные ин-

терфейсы для заполнения карточек CAPA и управления ими, но они не заточены под анализ кода или репозиториев. Автоматизация в этих системах ограничена триггерами (например, создание CAPA на основе записи о сбое теста или жалобы), и интеграция с процессами CI/CD чаще всего осуществляется через ручные интерфейсы или API. Подходы такого рода обеспечивают качественный учет проблем и статус выполнения мер, однако они не анализируют непосредственно изменения в коде и не извлекают CAPA автоматически на основе метрик репозитория [11].

Напротив, современные инструментальные решения в области разработки ПО стремятся к раннему обнаружению проблем в процессе написания кода. Сюда относятся системы статического анализа кода (SonarQube, CodeQL, Coverity), инструменты анализа сборок и покрытий (Codecov), а также инструменты анализа истории версий и производительности (Pluralsight Flow, CodeScene и др.). Такие инструменты обычно легко интегрируются в конвейер CI/CD (GitLab CI, Jenkins, GitHub Actions и др.), автоматизированно собирают метрики кода и могут генерировать отчеты с найденными дефектами или подозрениями. Например, SonarQube при подключении к Git-серверу автоматически анализирует каждый коммит или пулл-реквест на наличие проблем (code smells, багов, уязвимостей) и может предлагать меры по их устранению. Выходные данные статических анализаторов часто имеют стандартизированный формат (например, SARIF – Static Analysis Results Interchange Format), что облегчает интеграцию и агрегирование результатов [11]. Недостатком здесь является то, что такие инструменты обычно фокусируются на анализе текущего состояния кода, но не делают выводов о процессе разработки в целом или о трендах аномалий в истории коммитов. Тем не менее, их отчеты служат основой для корректирующих действий (исправление найденных дефектов) и косвенно могут стимулировать превентивные меры по устранению проблем.

## 1.3. Методы анализа коммитов и обучение на данных репозиториев

### *1.3.1. Предсказание дефектных коммитов (JIT-Defect Prediction)*

Одним из популярных подходов в анализе изменений является Just-In-Time Software Defect Prediction (JIT-SDP), направленный на обнаружение потенциально «багогенерирующих» коммитов до интеграции изменений. Модели JIT-SDP строятся на исторических данных о прошлых коммитах (метриках изменений, метаданных, сообщениях к коммитам) и обучаются классифицировать новые

коммиты как безопасные или потенциально дефектные. Например используя метод одно-классовой классификации (One-Class Classification) для идентификации аномалий среди коммитов [12]. Для её реализации модель обучается на «нормальных» (не дефектных) коммитах и затем помечает отклонения как подозрительные. Эксперименты показали, что при высоком дисбалансе классов (мало баговых коммитов относительно нормальных) одно-классовые алгоритмы (One-Class SVM, Isolation Forest и т.д.) превосходят традиционные классификаторы по точности обнаружения дефектных изменений [12]. JIT-SDP можно встроить непосредственно в систему контроля версий: модель принимает данные о новом коммите при попытке его зафиксировать и возвращает прогноз (например, в виде уведомления разработчику), что позволяет принять коррективные действия (дополнительная проверка, тестирование) до слияния. Такие подходы полностью автоматизированы (после первоначального обучения) и могут работать в CI/CD без вмешательства человека. Однако они часто ограничены точностью модели и могут давать ложные срабатывания, поэтому вопрос интерпретируемости результатов здесь актуален: разработчикам важно понимать, какие особенности коммита вызвали подозрение. Для повышения интерпретируемости предлагаются методы объяснения предсказаний (например, SHAP/ LIME или специальные эксплейнеры для JIT-SDP), хотя это выходит за рамки базовых моделей [12].

### *1.3.2. Классификация коммитов по типам работ (Commit Classification)*

Другой распространённый подход – классификация коммитов по видам технических действий (например, категориальное разделение на корректирующие, адаптивные, совершенствующие задачи). Этот метод чаще используется для анализа уже сделанных изменений с целью обзора тенденций и построения рекомендаций. В литературе выделяют три классических типа изменений: Corrective (исправление дефектов), Adaptive (обработка требований/увеличение функциональности) и Perfective (улучшение производительности/рефакторинг) [13]. Основные характеристики таких моделей: они используют признаковые описания коммитов (количество строк, слова из сообщений, данные о затронутых файлах) и типичные алгоритмы машинного обучения (Decision Tree, Random Forest, Naive Bayes, нейросети) [13]. Результаты таких классификаторов помогают в формировании превентивных мер: например, если обнаруживается, что большое число коммитов носит корректирующий характер из-за недостаточного тестового

покрытия, система может рекомендовать усилить модульное тестирование. В последние годы к этой задаче привлекаются большие языковые модели. Например, GPT-3 может в режиме zero-/few-shot эффективно классифицировать сообщения коммитов по категориям технической поддержки [13]. В частности, достигается ~75% точности классификации по трём категориям [13]. Это свидетельствует о высокой автоматизируемости такого подхода и возможности использования предобученных моделей, однако интерпретируемость итоговых меток, особенно при применении LLM остается проблемой.

Классификаторы коммитов в целом демонстрируют высокую применимость к истории репозиториев, поскольку анализируют каждое изменения во временной последовательности. Они хорошо интегрируются в CI/CD и инструменты мониторинга (через плагины GitHub/GitLab) – фактически после фиксации коммита производится моментальная классификация с логированием результата. Однако такие системы требуют обширных размеченных данных для обучения (или хорошо сформулированных правил) и могут не учесть все контексты проекта, что снижает их точность для разных проектов [13, 15]. Тем не менее, при правильной настройке они позволяют формировать CAPA как в корректирующей, так и превентивной части: например, классифицируя «адаптивные» коммиты, можно обнаружить систематические изменения в требованиях и заранее планировать архитектурные доработки.

### 1.4. Инструменты и системы автоматизации CAPA

На практике для генерации CAPA в области разработки ПО применяются комбинированные решения, объединяющие анализ кода и процесс управления. Некоторые примеры:

– Встроенные боты и скрипты в репозиторий.
  Примером является система TOM (Theoretically Objective Measurements), описанная Bugayenko и соавторами [2]. Для её использования необходимо лишь добавить бота @0capa в репозиторий. Он автоматически собирает метрики по каждому коммиту и на их основе предлагает CAPA. Такой подход максимально автоматизирован и интегрируется через механизмы GitHub (было реализовано создание pull request с рекомендациями CAPA). Информация о том, какие действия требует каждый коммит, хранится

вместе с кодом, что повышает прозрачность процесса. Хоть и система была описана, готового решения по @0capa нет в открытом доступе.

- Плагины и дополнения к системам CI/CD.
  Многие CI-системы (GitLab, Jenkins, GitHub Actions) поддерживают установку плагинов, выполняющих анализ коммитов или статический анализ. Например, SonarQube имеет плагин SonarScanner для GitLab CI, а GitHub Actions – для автоматического запуска анализа при каждом PR. Некоторые сервисы (например, bugasura.io) предлагают «AI-инспектора», который отслеживает журналы сборок и автоматически сигнализирует о паттернах отказов. Такие инструменты часто обеспечивают интеграцию в процесс разработки и могут генерировать тикеты или отчеты CAPA. Форматы вывода обычно совместимы со стандартными трекерами (JIRA, GitHub Issues), что облегчает стандартизацию.
- Инструменты анализа pull-реквестов и метаданных.
  Существуют системы, которые обучены классифицировать Pull Request как «корректирующий» или «не корректирующий» [2]. По результатам анализа PR или истории веток такие инструменты могут «присваивать» CAPA к прошедшим исправлениям и накапливать статистику. Это позволяет накапливать базу знаний о типичных рекомендациях и соотносить их с шаблонами изменений. Однако подобные решения пока редко встречаются в свободном доступе и требуют развитой инфраструктуры данных. [2]
- Статические анализаторы с функцией CAPA.
  Ряд современных SAST-инструментов (Fortify, Checkmarx, SonarQube) начинают включать в отчеты «рекомендации» помимо указания на проблему. Например, при обнаружении уязвимости система может автоматически предложить типовую корректирующую меру (обновить библиотеку, изменить код), что можно рассматривать как CAPA в узком смысле. Хотя это скорее рекомендации на уровне кода, а не на уровне процесса разработки, их вывод можно экспортировать для дальнейшей автоматической обработки. Интеграция в CI/CD у подобных систем хорошо налажена, а результаты достаточно интерпретируемы (описаны правила, чек-листы).

Таким образом, современные инструменты сходятся на том, что интеграция в процессы CI/CD и автоматизация – ключевое требование. Боты и плагины обеспечивают автоматический сбор данных из репозиториев и формирование CAPA-рекомендаций без участия человека, хотя зачастую требуют первоначальной

настройки моделей и правил. Статические анализаторы обеспечивают проверку качества кода по стандартным правилам, и их выходы можно использовать для инициирования CAPA в QMS (например, через настройку связей с Jira или другой системой учета).

Таблица 1.1

Сравнение подходов и инструментов формирования CAPA

| **Подход / инструмент** | **Автоматизируемость** | **Интеграция в CI/CD** | **История коммитов** | **Интерпретируемость результатов** |
|---|---|---|---|---|
| CAPA-менеджмент (QMS-софт) | Низкая (ручной ввод, шаблоны) | Нет / частичная (обычно отдельные интерфейсы) | Нет | Высокая (структурированные отчёты) |
| Статический анализ кода (SonarQube, SAST) | Высокая (автозапуск при сборке) | Да (плагины для Jenkins, GitLab CI и др.) | Нет (анализ текущего кода, не истории) | Средняя (правила и рекомендации по коду) |
| ML-анализ коммитов (JIT-SDP, аномалии) | Средняя (требует обучения моделей) | Частичная (скрипт или сервис в CI) | Да (используют данные из логов коммитов) | Низкая (бинарная метка «аномален/нормален») |
| Классификация коммитов (ML/LLM) | Средняя (настройка и дообучение) | Да (CI-боты, webhook) | Да | Средняя/Низкая (метки, black–box) |
| Бот-анализ репозитория (TOM, GitHub App) | Высокая (авто PR-рекомендации) | Да (GitHub/GitLab API, вебхуки) | Да | Средняя (текстовые рекомендации без метрик) |

Автоматизируемость. Статический анализ кода и встроенные боты наделены высокой степенью автоматизации: по каждому пушу запускается анализ, не требующий участия человека. В отличие от них, в системах QMS CAPA часто заводятся вручную после выявления проблемы. ML-подходы требуют обучения на данных и зачастую периодического ручного контроля моделей, что снижает из коробки автоматичность.

Интеграция в CI/CD. Инструменты статического анализа и многие современные сервисы (например, TOM) легко интегрируются через плагины и вебхуки, поскольку они заточены на DevOps. ML-сервисы анализа коммитов также могут быть встроены (например, как отдельный шаг в пайплайне), но для этого требуется соответствующая инфраструктура. Традиционные CAPA-системы обычно вне разработки и не участвуют в CI.

Анализ истории коммитов. Очевидно, только подходы, основанные на данных репозитория (JIT-предсказание, классификация коммитов, боты), непосредственно используют историю коммитов и метаданные. Статический анализ сосредоточен на содержимом и не учитывает, как развивался проект со временем. Это означает, что только для первых трех подходов выполняется критерий «применимость к истории коммитов».

Интерпретируемость результатов. QMS-системы выигрывают в том, что выходы представлены привычными для людей отчётами, понятными для менеджеров. Статический анализ даёт диагностические сообщения, основанные на известных правилах, поэтому довольно интерпретируем: каждой найденной проблеме сопоставляется рекомендация (например, «поправьте название переменной»). ML-модели коммитов часто малообъяснимы и выдают лишь метки или флаги («дефектный/не-дефектный»), без подробностей. LLM-категоризация даёт метки на естественном языке, но модель остаётся трудно контролируемой. Боты вроде TOM могут выдавать человекочитаемые рекомендации, что улучшает понятность (оба примера – тексты к PR), но качество зависит от продуманности шаблонов ответов.

## 1.5. Выводы

Современные практики формирования CAPA на основе анализа репозиториев кода объединяют строгие процедуры управления качеством и динамическую ML-аналитику истории изменений. Классический цикл CAPA — сбор инцидентов, диагностика, планирование и проверка эффективности мер — дополняется JIT-предсказанием дефектов и классификацией коммитов на основе моделей машинного обучения, а также «бот-аналитикой» (TOM, GitHub Apps), когда система сама собирает метрики и создаёт pull-request'ы с рекомендациями. При выборе решения критичны автоматизируемость, интеграция в CI/CD, учёт истории коммитов и интерпретируемость результатов. Статические анализаторы (SonarQube) надёжно автоматизируются, но не отслеживают эволюцию проекта, а ML-модели улавливают сложные паттерны в истории, но требуют обучения и часто выступают «чёрным ящиком». Наименее работоспособным оказывается чистый QMS-софт без аналитики кода. Лучшие практики — это гибридный подход: статический анализ, ML-классификация и интеграция с баг-трекером для автосоздания задач CAPA.

# ГЛАВА 2. ПРОЕКТИРОВАНИЕ СИСТЕМЫ ФОРМИРОВАНИЯ CAPAS НА ОСНОВЕ ИЗМЕНЕНИЙ РЕПОЗИТОРИЯ КОДА

## 2.1. Требования к системе

Исходя из информации, полученной при анализе информационных источников были установлены следующие требования к разрабатываемой системе:

- Система должна автоматически извлекать историю коммитов и связанные метрики из репозиториев GitHub (авторы, даты, количество добавленных/удалённых строк, изменения в файлах).
- Необходимо выполнять статический анализ кода на разных языках (например, Python, JavaScript, Java). Система должна запускать такие инструменты, как pylint, bandit для Python, eslint для JavaScript, checkstyle для Java, чтобы определить проблемы качества и уязвимости. Результаты анализа (количество предупреждений, метрики сложности и т.п.) обогащают данные коммитов.
- На основании извлечённых признаков система должна определять «аномальные» изменения (например, дефектно-опасные коммиты). Здесь применяются методы Just-In-Time предсказания дефектов: обучение моделей машинного обучения для классификации коммита по риску [14].
- На основе обнаруженных аномалий система формирует корректирующие и предупреждающие действия. Это могут быть комментарии или автоматически созданные задачи/PR в GitHub с рекомендациями по улучшению процесса (например, «увеличить покрытие тестами», «провести рефакторинг сложного модуля»). Функционал сходен с подходом TOM: бот анализирует метрики и создает issue с описанием аномалий и действий [2].
- Система должна представлять результаты анализа в наглядном виде. Планируется интерактивный дашборд с графиками активности коммитов, распределением метрик и отмеченными аномальными событиями. Пользователь сможет просматривать тренды изменений и статусы CAPA.

## 2.2. Нефункциональные требования

- Масштабируемость: Решение должно уметь обрабатывать большие репозитории (сотни коммитов) и несколько проектов параллельно.

- Расширяемость и универсальность: Архитектура должна легко адаптироваться к новым языкам и инструментам. Развитием этой идеи будет наличие в проекте класса RepoAnalyzer, который будет построен универсально: для каждого типа файла в репозитории будет отображение на список анализаторов (например, '.py': [PylintAnalyzer, BanditAnalyzer], '.js': [ESLintAnalyzer], '.java': [CheckstyleAnalyzer]). Добавление нового языка сводится к добавлению пары (расширение → анализатор) в конфигурацию. Таким образом поддерживается гибкость и масштабируемость системы.
- Надёжность и своевременность: Анализ должен выполняться регулярно (например, по cron или по событию в GitHub) и выдаваться своевременно. Система должна корректно обрабатывать неуспешные запросы к API (повторять/логировать ошибки) и обеспечивать целостность данных.
- Используемость и интеграция: Интерфейс (дашборд) должен быть интуитивным и информативным. Для взаимодействия с GitHub используются официальные API и безопасное хранение токенов. Механизм выдачи CAPA в виде GitHub-issues упрощает интеграцию с процессом разработки.

### 2.3. Выбор технологий и инструментов

Основным языком разработки был выбран Python. Благодаря богатой экосистеме библиотек для анализа данных и машинного обучения. В частности, pandas обеспечивает удобную обработку табличных данных, scikit-learn – основа для обучение моделей классификации. Альтернативы (например, C++ или Java) имеют аналогичные библиотеки, но Python позволяет быстро прототипировать и интегрировать различные компоненты (API, ML, веб).

Для доступа к данным репозиториев (коммиты, файлы, статистика изменений). Используется GitHub API. Альтернатива – локальное клонирование через git CLI, но API быстрее предоставляет аггрегированные данные (например, статистику добавлений/удалений).

Dash + Plotly: Выбран для разработки веб-интерфейса/дашборда. Dash – высокоуровневый фреймворк на Python, построенный поверх Flask и React, облегчает создание интерактивных приложений с помощью Python-кода. Dash даёт большую гибкость в настройке интерфейса и графиков. Plotly обеспечивает красивые и интерактивные графики без необходимости писать JavaScript.

pandas: Де-факто стандарт для работы с данными в Python. Позволяет быстро агрегировать и трансформировать данные коммитов перед подачей в модели. Альтернативы: использовать NumPy напрямую или базы данных, но pandas удобнее для аналитики и визуализации.

KMeans: Используется для кластеризации коммитов и определения «границы аномалии». Алгоритм прост и хорошо масштабируется. Идея – отсортировать коммиты по удалённости до центра кластера и пометить самые далекие как аномалии. Альтернативы могли быть методы на основе плотности (DBSCAN) или статистического анализа, но KMeans достаточно для первоначального порога, при этом не требуется ввод дополнительных сложных гиперпараметров.

Статические анализаторы: Pylint (строгий линтер для Python), Bandit (фокус на уязвимости Python), ESLint (статический анализ JS/TS), Checkstyle (Java). Эти инструменты бесплатны, широко используются в индустрии и генерируют стандартизованный вывод, удобный для парсинга. Например, альтернативный SonarQube/CodeQL требовал бы более тяжёлой инфраструктуры, тогда как легковесные линтеры легко интегрировать в пайплайн. Выбор pylint/ESLint обусловлен их широкой поддержкой сообществом и гибкостью настроек.

## 2.4. Архитектура системы

Система разделена на несколько ключевых компонентов, взаимосвязанных последовательной передачей данных и взаимодействием с GitHub (см. диаграмму ниже):

Компонент сбора данных (GitHubRepoAnalyzer) выполняет аутентификацию в GitHub и последовательно обходит указанные репозитории. Для каждого коммита он извлекает метаданные (SHA, автор, дата) и статистику изменений (количество добавленных/удалённых строк, изменённых файлов). Одновременно по хешу коммита можно получить дифф файлов или полное состояние репозитория. Этот класс реализован универсально: имеется словарь mapping, связывающий расширение файла с набором анализаторов. Например, встретив файл example.py, система запустит анализаторы PylintAnalyzer и BanditAnalyzer; для script.js – ESLintAnalyzer.

Компонент статического анализа и извлечения метрик обрабатывает результаты GitHubRepoAnalyzer. С использованием внешних утилит (pylint, eslint и др.) вычисляются количественные показатели качества кода (количество предупреждений,

оценка сложности, метрики покрытия). Эти признаки объединяются с результатами анализа изменений (`lines added/deleted`, сложность диффа, время от последнего коммита и т.п.). К примеру, для каждого коммита формируется вектор признаков: `[добавлено, удалено, #файлов, pylint_warns, eslint_errors, complexity_me`
Важно, что система может легко расшириться под новые языки: достаточно добавить соответствующий парсер вывода (анализатор) и подключить его к GitHubRepoAnalyzer через mapping.

Компонент классификации (ML-модель) обученная модель (любая совместимая с scikit-learn) принимает набор признаков коммита и оценивает вероятность аномалии или дефекта. Модель обучается на исторических данных (имея пометки «CAPA – да/нет» или метки типа коммита). Также дополнительно применяется кластеризация KMeans: все коммиты разбиваются на кластеры по признакам, а затем для нового коммита определяется его расстояние до ближайшего центра. Если это расстояние превышает эмпирический порог (например, верхний квартиль по обучающим данным), коммит считается аномальным. Такое сочетание методов JIT-предсказания и кластерного анализа позволяет надежнее выявлять «выбивающиеся» изменения.

Компонент генерации рекомендаций получает от классификатора список выявленных аномалий. Для каждой из них по заранее заданному правилу или классификатору формируется текст рекомендаций. Например, при малом покрытии тестами предлагается «написать модульные тесты», при частых правках конфигурационных файлов – «документировать изменения». Система формирует GitHub-issue (или pull request с комментарием) через API. После закрытия задачи систему можно запустить повторно для проверки устранения проблемы [2].

Веб-приложение на Dash отображает агрегированные результаты. На графиках показаны метрики репозитория: количество коммитов во времени, распределение изменений (additions/deletions), число предупреждений статических анализаторов и отмеченные аномальные коммиты. Пользователь видит временные ряды и гистограммы, может выбирать интересующие проекты или ветки. Такой интерактивный мониторинг облегчает понимание тенденций разработки и эффективности CAPA.

Эта архитектура разделена по этапам «сбор данных → анализ → классификация → рекомендации → визуализация». Таким образом обеспечивается модульность и возможность расширения каждого компонента без изменения остальных.

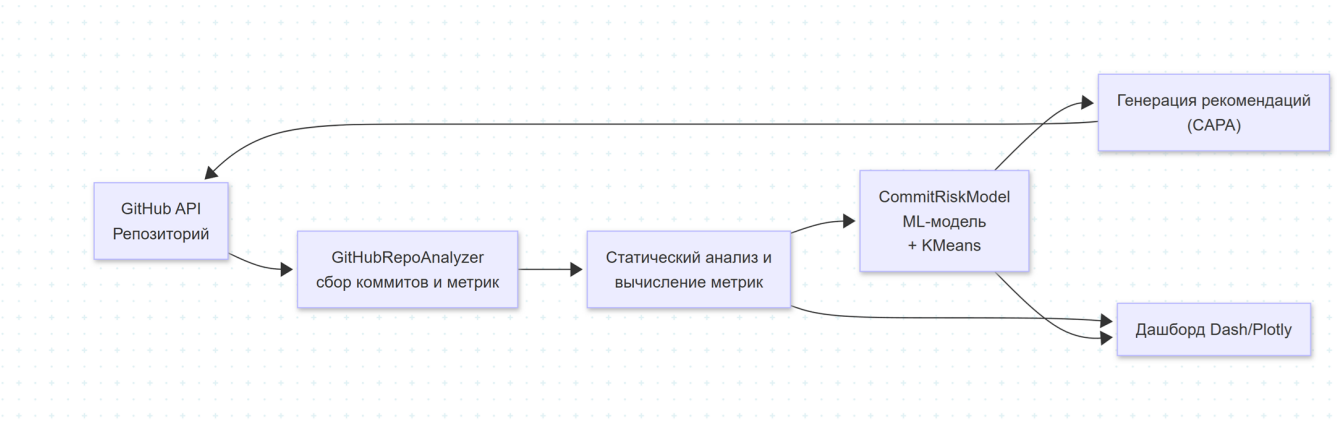


Рис.2.1. Пайплайн автоматического анализа коммитов GitHub

В этой диаграмме видно, как данные из GitHub поступают в компонент сбора, затем проходят анализ и классификацию, после чего строятся рекомендации. Результаты анализа и классификации также поступают на дашборд для визуализации.

## 2.5. Заключение

В результате проделанной работы была спроектирована система, способная автоматически извлекать и обрабатывать данные о коммитах из репозиториев GitHub, включая метрики добавленных и удалённых строк, состав файлов и результаты статического анализа. Универсальный класс RepoAnalyzer обеспечит гибкость при подключении новых языковых анализаторов и упрощает расширение системы под дополнительные инструменты качества кода. Компонент классификации на основе машинного обучения и KMeans надёжно выявляет аномальные изменения, комбинируя методы JIT-предсказания дефектов и кластерного анализа. Модуль генерации рекомендаций формирует корректирующие и предупреждающие действия (CAPA) в виде GitHub-issues или pull request'ов, что облегчает интеграцию с существующими процессами разработки. Интерактивный дашборд, реализованный на Dash и Plotly, демонстрирует ключевые показатели активности репозитория, распределение метрик и отмечает проблемные коммиты в удобном визуальном формате. Масштабируемость решения позволяет обрабатывать сотни коммитов и несколько проектов одновременно, а надёжные механизмы повторных запросов к API и логирования ошибок гарантируют стабильность работы. Расширяемая архитектура упрощает добавление новых языков программирования и сторонних анализаторов через конфигурацию mapping, что минимизирует затраты на сопро-

вождение. Нефункциональные требования по своевременности и интеграции также удовлетворены: анализ может запускаться по расписанию или по вебхукам GitHub, а безопасное хранение токенов обеспечивает защиту учётных данных. Выбранный стек технологий (Python, pandas, scikit-learn, Dash, Plotly) доказал свою эффективность при быстром прототипировании и дальнейшем развитии проекта. Итоговая архитектура демонстрирует баланс между модульностью, производительностью и удобством для конечных пользователей, что делает систему готовой к внедрению и дальнейшему масштабированию.

# ГЛАВА 3. РЕАЛИЗАЦИЯ СИСТЕМЫ ФОРМИРОВАНИЯ CAPAS НА ОСНОВЕ ИЗМЕНЕНИЙ РЕПОЗИТОРИЯ КОДА

## 3.1. Извлечение и обработка данных из GitHub

На этапе сбора данных реализован класс GitHubRepoAnalyzer, инкапсулирующий функциональность подключения к API GitHub и анализа истории коммитов целевого репозитория. Данный класс обеспечивает получение списка коммитов, извлечение деталей каждого коммита и расчёт необходимых метрик для последующего анализа. Ниже перечислены ключевые методы и процедуры, реализованные в GitHubRepoAnalyzer:

- **Аутентификация и подключение к репозиторию**. В конструкторе класса выполняется аутентификация к GitHub через токен доступа и инициализируется подключение (С помощью запросов REST API). Это позволяет в дальнейшем безопасно вызывать методы GitHub API для чтения данных.
- **Получение списка коммитов**. Метод `fetch_commits(repo_name)` отправляет запрос к GitHub API для получения списка коммитов указанного репозитория `repo_name`. Он возвращает итератор или список объектов коммитов (содержащих информацию о каждом коммите: SHA-хеш, автор, дата, сообщение, статистика изменений и др.). При необходимости данный метод может поддерживать постраничную загрузку, чтобы обработать весь исторический ряд коммитов для крупных проектов.
- **Извлечение данных коммита и расчёт метрик**. Для каждого коммита выполняется обработка деталей. Метод `analyze_commit(commit)` рассчитывает такие показатели как: количество изменений в коде: число

добавленных строк кода (additions), удалённых строк (deletions) и общее изменение (`total_changes`, сумма добавленных и удалённых строк) на основе статистики диффа, предоставляемой GitHub. Затронутые файлы: число файлов, изменённых в данном коммите. Этот показатель косвенно отражает масштабы влияния изменения на кодовую базу (коммит, затрагивающий много файлов, вероятно, более комплексный). Интервал времени с предыдущего коммита: разность во времени (в днях) между текущим коммитом и предыдущим по времени. Первый коммит в истории не имеет предыдущего, для него интервал определяется как 0. Данный показатель позволяет отслеживать ритм работы над проектом. Наличие признаков багфикса: анализ текста сообщения коммита на наличие ключевых слов, указывающих на исправление ошибки (например, "fix "bug "error"). Если такие слова присутствуют, для коммита устанавливается флаг, что он относится к исправлению дефекта. Это служит бинарным признаком (0/1), указывающим на потенциально проблемный характер изменения (коммит, содержащий багфикс, свидетельствует о том, что ранее в коде была проблема). Оценка сложности изменения: введена дополнительная метрика сложности, оценивающая масштаб и потенциальную сложность внесённого изменения. В рамках данной работы сложность коммита приближённо характеризуется совокупностью других метрик – например, общим числом изменённых строк кода и количеством затронутых файлов. Предполагается, что коммиты с большим числом изменений и широким охватом файлов более сложны для понимания и проверки.

– **Сохранение и предварительная обработка**. Собранные по каждому коммиту данные сохраняются либо в оперативной памяти (в виде структуры pandas.DataFrame), либо сразу в файловое хранилище. В текущей реализации происходит запись метрик коммитов в CSV-файл (`repository_data.csv`) с колонками: репозиторий, SHA коммита, автор, дата и время, сообщение, добавления, удаления, всего изменений, файлов изменено, интервал с предыдущим коммитом, признак багфикса и поле под флаг CAPA. Этот датасет затем используется на следующих этапах анализа. Перед моделированием данные могут дополнительно очищаться: устраняются дубликаты, проверяется корректность временных меток, при необходимости вычисляются дополнительные поля (например, средняя частота коммитов для автора или кумулятивные метрики).

В результате работы GitHubRepoAnalyzer формируется структурированный набор данных обо всех коммитах репозитория со значимыми характеристиками каждого изменения. Такой подход автоматизирует извлечение данных, избавляя от ручного сбора статистики, и гарантирует единообразие вычисленных метрик. Полученные данные служат основой для применения алгоритмов машинного обучения на следующем этапе. Таким образом, реализация модуля сбора и предварительного анализа данных обеспечивает подготовку качественного обучающего множества для последующего моделирования CAPA.

### 3.2. Интеграция модели глубокого леса

Для выявления потенциально проблемных коммитов в системе используется модель классификации на основе алгоритма глубокого леса (Deep Forest). Прежде чем обучить модель, необходимо сформировать обучающую выборку, включающую признаки коммитов и целевой признак (метку), указывающую, требуется ли для данного коммита выработка CAPA. В текущей реализации разметка данных выполнена на основе эвристических правил и результатов предварительного анализа: Коммит помечается как «требующий CAPA» (метка 1), если он удовлетворяет одному или нескольким критериям риска: например, содержит исправление ошибки (выявлено по ключевым словам в сообщении), имеет чрезвычайно большой объём изменений (значительно превышающий типичные значения по проекту) или связан с аномально долгим интервалом отсутствия активности перед ним. Такие коммиты свидетельствуют о возникновении проблем (дефект в коде, накопление большого пакета изменений, сбой в регулярности разработки), что требует принятия мер. Коммиты, не соответствующие этим условиям, считаются обычными (метка 0). Они характеризуются типичным размером и содержанием изменений, не содержат явных признаков багфиксов и происходят с регулярной частотой, соответствующей нормальному ходу разработки.

На основе класса GitHubRepoAnalyzer из предыдущего раздела формируются признаки для модели. В качестве входных признаков (features) для каждого коммита используются:

A. Общее количество изменённых строк кода (`total_changes`), отражающее размер коммита.
B. Число затронутых файлов (`files_changed`).
C. Интервал времени с предыдущего коммита (`time_since_last_commit`).

D. Наличие багфикса (булев флаг `bug_fix_flag`, 1 – если в сообщении коммита обнаружены слова, указывающие на исправление ошибки).

E. Производные или комплексные признаки сложности (например, комбинация из первых двух: большие `total_changes` при большом `files_changed` могут усиливать оценку сложности).

Перед обучением количественные признаки масштабируются к сопоставимому диапазону. Если данных для обучения относительно немного, может применяться кросс-валидация или бутстреп-перемешивание для более устойчивого обучения модели. Модель глубокого леса, используемая в системе, реализована в виде классификатора CascadeForestClassifier. Данный алгоритм представляет собой каскадную композицию ансамблей решающих деревьев, альтернативный подход глубокого обучения для табличных данных. Вместо одной стадии обучения случайного леса, CascadeForest выстраивает несколько уровней (layers) из ансамблей (случайных лесов и полностью случайных деревьев), последовательно обрабатывающих данные. На каждом уровне входные признаки дополняются выходами предыдущего уровня (например, вероятностями классов), что позволяет каскаду постепенно «усложнять» представление данных и улучшать качество классификации. Обучение продолжается до тех пор, пока добавление нового уровня улучшает качество на валидационном подмножестве, либо останавливается при достижении заданного числа уровней.

В ходе обучения CascadeForestClassifier на собранных данных коммитов каждый уровень каскада строит несколько случайных лесов. Так, на первом уровне может быть обучено два случайных леса: один на исходных признаках, другой на той же выборке, но с иной инициализацией. Результаты (предсказанные вероятности классов для каждого примера) затем прикрепляются к признакам, и второй уровень обучает новые леса с расширенным пространством признаков. Такой подход позволяет модели выявлять сложные нелинейные зависимости в данных коммитов, которые мог пропустить один «плоский» алгоритм.

После тренировки модели на обучающей выборке её качество проверяется на отложенных данных (тестовом наборе) либо с помощью кросс-валидации. Оценка показала, что модель глубокого леса успешно классифицирует коммиты с точки зрения необходимости САРА, превосходя по некоторым метрикам более простой алгоритм случайного леса. Например, для множества тестовых коммитов были корректно выявлены все случаи известных проблемных изменений при небольшом количестве ложных срабатываний.

Анализ важности признаков показал, что наибольшее влияние на решение о рискованности коммита оказывают объём внесённых изменений кода и наличие багфикса. Так, признак общего количества изменённых строк получил наивысший вес (самый информативный), поскольку крупные коммиты чаще связывались с последующими проблемами. Следующими по важности идут временной интервал (длительные паузы перед коммитом могли указывать на накопление изменений или упущенные баги) и индикатор исправления ошибок. Число затронутых файлов сыграло заметную, но меньшую роль. Анализ значимости подтверждает, что выбранные метрики обоснованно влияют на решение модели и соответствуют интуитивным представлениям: действительно, чем больше и реже изменения, тем выше вероятность необходимости дополнительных действий.

Таким образом, посредством обучения модели CascadeForestClassifier была получена интеллектуальная подсистема, способная на основе метрик коммита предсказывать необходимость CAPA. Этот классификатор служит ядром системы, автоматически оценивая каждый новый коммит и выделяя наиболее рискованные, требующие внимания разработчиков или менеджеров проекта.

### 3.3. Реализация панели визуализации на фреймворке Dash

Визуализационная панель разработана с использованием фреймворка Dash - инструмента создания адаптивных и интерактивных веб-приложений на Python.

Интерфейс приложения разбит на несколько вкладок. Например, есть обзорная вкладка с ключевыми метриками и графиками по всем коммитам, отдельная вкладка с детальной аналитикой (графики изменений по авторам, времени и т.п.) и вкладка «Рекомендации», где в табличном виде выводится список сформированных системой корректирующих действий (CAPA) для выявленных аномалий. Для построения визуализаций используются возможности библиотеки Plotly: гистограммы (распределение числа коммитов по дням недели, авторам, величине изменений), круговые диаграммы (распределение изменений по типам или модулям), тепловые карты активности коммитов во времени и пр. Такие графики позволяют исследовать данные «вживую» и лучше выявлять закономерности

– **Структура и вкладки:** Панель разбита на тематические разделы. Например, основная вкладка показывает общее состояние репозитория (гистограммы коммитов, круговые диаграммы распределения изменений по авторам и файлам), другая – показывает детальную статистику во времен-

ном разрезе, а вкладка «Рекомендации» содержит список автоматических советов (CAPA) по улучшению процесса разработки.

- **Типы визуализаций:** Используются интерактивные графики Plotly – гистограммы, круговые (pie) диаграммы, тепловые карты и линейные графики. Все эти диаграммы являются кликабельными, что позволяет пользователю изучать подробные данные.
- **Интерактивность:** За динамическое поведение отвечает механизм callback-функций Dash. При выборе фильтров (например, диапазона дат, конкретного автора или папки проекта) соответствующие графики автоматически обновляются.
- **Адаптивность интерфейса:** При создании приложения использованы компоненты `dash_bootstrap_components` и тема Bootstrap. В частности, при инициализации приложения подключена тема Bootstrap (`external_stylesheets=[dbc.themes.BOOTSTRAP]`), что гарантирует адаптивное размещение элементов на экране. В итоге панель корректно отображается на различных устройствах и экранах, сохраняя удобство восприятия.

Всё это делает дашборд удобным инструментом мониторинга: наглядные диаграммы и список рекомендаций позволяют быстро оценить состояние репозитория и принять решения на основе анализа данных.

### 3.4. Интеграция компонентов в единую систему

Модули сбора данных, анализа и визуализации объединены в единую конвейерную цепочку обработки. Dash-приложение напрямую подключается к Python–моделям и хранилищам данных, что позволяет строить «сквозную» аналитику. В частности, Dash умеет обращаться к базам данных и другим источникам через Python-коннекторы, выполняя запросы и продвинутый анализ «на лету». роме того, фреймворк Dash хорошо интегрируется с библиотеками обработки данных (Pandas, NumPy и др.), что облегчает построение комплексных аналитических приложений.

Работа системы организована следующим образом:

- **Сбор данных:** Модуль извлечения с помощью HTTP запросов обращается к GitHub, получает историю коммитов заданных репозиториев и сохраняет её в удобном формате. Собираются метрики коммита: SHA, автор, дата,

количество добавленных/удалённых строк, список затронутых файлов и т.д. Этот этап можно запускать вручную при появлении новых данных.

- **Обработка и анализ:** Загруженные данные проходят первичную обработку: вычисляются дополнительные признаки (интервалы между коммитами, суммарные изменения), после чего применяется кластеризация (KMeans) для определения нормальных границ изменений. Затем обучаются модели машинного обучения Глубокого Леса для классификации коммитов на нормальные и аномальные.
- **Генерация рекомендаций (CAPA):** На основе классификации система формирует корректирующие и предупредительные действия для выявленных аномалий. Для каждого «подозрительного» коммита вычисляются рекомендации (например, обратить внимание на высокую частоту изменений или большой объём кода) и при необходимости автоматически создаётся pull request с этими рекомендациями.
- **Визуализация результатов:** После анализа результаты поступают на дашборд. Dash-приложение загружает актуальный набор данных (или обращается к подготовленной статистике) и отображает их в виде графиков и таблицы рекомендаций. Таким образом, пользователь получает единый интерфейс, где и графики, и текстовые CAPA-сообщения согласованы между собой.
- **Автоматическое обновление:** При появлении новых коммитов цикл повторяется: система периодически выполняет сбор свежих данных, прогон анализов и обновляет панель. Пользователи получают актуальную информацию без необходимости ручного запуска каждого этапа – весь процесс сквозной автоматизации обеспечивает своевременное отображение рекомендаций и метрик.

## 3.5. Выводы

Разработанная система была реализована в соответствии с поставленными задачами: созданы отдельные модули для извлечения данных коммитов, их анализа и визуализации. Модульный подход позволяет четко разделять функциональность и поддерживать каждый компонент независимо.

Достигнуты ключевые цели проекта: архитектура системы является модульной и легко расширяемой – новые алгоритмы анализа или метрики могут быть

добавлены без переделки остального кода. Процесс анализа коммитов полностью автоматизирован: от сбора данных до отображения результатов на панели не требуется ручного участия, что облегчает регулярный мониторинг качества разработки. В ходе работы создан интерактивный дашборд на Dash, обеспечивающий гибкую визуализацию и удобный интерфейс для разработчиков. Это позволяет в реальном времени отслеживать состояние репозитория и эффективно использовать полученные рекомендации для улучшения кода.

Перспективы развития проекта включают расширение функционала: можно добавить новые метрики коммитов и шаблоны анализа, интегрировать систему с инструментами CI/CD и сервисами контроля качества кода, а также исследовать применение нейросетевых моделей для повышения точности предсказаний CAPA.

# ГЛАВА 4. АПРОБАЦИЯ РЕЗУЛЬТАТОВ ИССЛЕДОВАНИЯ

## 4.1. Цель и задачи апробации

Основная цель апробации – оценить работоспособность и практическую ценность разработанного инструмента для анализа коммитов и предсказания рисковых изменений. Для этого необходимо проверить корректность сбора и предобработки данных из реальных публичных и приватных репозиториев, оценить качество модели классификации «рисковых» коммитов (анализ precision, recall, F1-score), оценить стабильность работы интерфейса и дашборда при переключении между разными репозиториями, а также проверить удобство визуализации и полноту вырабатываемых рекомендаций.

## 4.2. Методика тестирования

Процесс тестирования начинается со сбора исторических данных. Для этого выгружаем часть коммитов некоторых репозиториев и размечаем «эталонные» аномалии вручную. Метим около 10% наиболее крупных коммитов (по числу строк), а также коммиты с ключевыми словами fix, error, bug. После этого обучаем модель. Делим набор на train:test = 80:20, оцениваем классификатор по метрикам precision, recall, F1-score для класса «рисковых», строим ROC-кривую и вычисляем AUC.

Также будет проведено ручное тестирование интерфейса. Проверим переключение между тремя репозиториями в выпадающем списке, работоспособность обновления всех графиков (гистограммы, scatter, heatmap) при изменении репозитория, корректность пагинации и фильтрации в таблице коммитов.

### 4.3. Практическое применение и апробация

Процесс апробации включает в себя практическое применение разработанного решения и интерпретацию графических представлений интерфейа, а также составление отчета исходя из полученной информации на примере разных проектов. Для апробации были выбраны несколько репозиториев с различными паттернами разработки:

- Высокой частотой коммитов (активно разрабатываемые проекты);
- Длинными промежутками между коммитами (поддерживаемые проекты);
- Большим количеством изменений в кодовой базе.

### 4.4. Выводы

# ЗАКЛЮЧЕНИЕ

В ходе данной работы была разработана система автоматического анализа коммитов, направленная на выявление аномалий и формирование корректирующих и предупреждающих действий (CAPA). Использование методов машинного обучения и кластеризации позволило создать инструмент, способный анализировать историю изменений в коде и предлагать рекомендации для повышения качества программного обеспечения.

Основные результаты работы можно сформулировать следующим образом:

- Проведен обзор существующих методов анализа данных из репозиториев исходного кода и выявлены их ограничения.
- Разработан алгоритм автоматического извлечения данных о коммитах с последующей их обработкой и анализом.
- Предложен метод кластеризации коммитов с использованием алгоритма KMeans для определения пороговых значений изменений в коде.
- Обучены и протестированы модели машинного обучения (случайный лес, наивный байесовский классификатор и глубокий лес), показавшие высокую точность в задаче предсказания аномалий.
- Разработан механизм автоматического создания pull request с рекомендациями CAPA, который интегрируется в процесс разработки.
- Создан интерактивный дашборд для визуализации результатов анализа, что позволяет разработчикам легко отслеживать состояние репозитория и принимать решения на основе данных.

Практическая значимость предложенной системы заключается в том, что она позволяет автоматизировать контроль за качеством кода, минимизировать ошибки, возникающие в процессе разработки, и повысить прозрачность изменений в репозитории. Используемый подход может быть адаптирован для различных проектов и масштабируем для работы с крупными кодовыми базами.

В дальнейшем возможны следующие направления развития системы:

- Доработка алгоритмов выявления аномалий с учетом более сложных паттернов изменений в коде.
- Расширение набора метрик для анализа коммитов.
- Интеграция с другими инструментами контроля качества кода и CI/CD системами.

– Применение нейросетевых моделей для улучшения предсказательной способности системы.

Таким образом, проведенное исследование подтвердило эффективность предложенного подхода к анализу коммитов. Разработанная система способствует улучшению управления процессом разработки программного обеспечения, сокращает время на выявление потенциальных проблем и повышает качество выпускаемого кода.

# СЛОВАРЬ ТЕРМИНОВ

**CAPA (Corrective and Preventive Actions)** — корректирующие и предупреждающие действия, направленные на устранение и предотвращение дефектов в процессе разработки программного обеспечения.

**GitHub** — веб-сервис для хостинга IT-проектов и их совместной разработки на базе системы управления версиями Git.

**Коммит (commit)** — фиксация изменений в репозитории Git, включающая информацию о внесённых правках, авторе и времени изменения.

**KMeans** — метод кластеризации данных, основанный на разбиении множества на k групп по схожести признаков.

**Случайный лес (Random Forest)** — ансамблевый метод машинного обучения, использующий множество деревьев решений для повышения точности прогнозов.

**Наивный Байесовский классификатор** — алгоритм машинного обучения, основанный на теореме Байеса и предположении независимости признаков.

**Глубокий лес (Deep Forest)** — метод машинного обучения, использующий каскадную структуру случайных лесов для улучшения классификации.

**API (Application Programming Interface)** — интерфейс программирования приложений, позволяющий взаимодействовать с внешними сервисами и библиотеками.

**Pull Request (PR)** — запрос на внесение изменений в репозиторий GitHub, который проходит процесс ревью перед слиянием в основную ветку.

**Dash** — фреймворк на Python для создания интерактивных дашбордов и веб-приложений для визуализации данных.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

## БИБЛИОГРАФИЧЕСКИЙ СПИСОК


[1] Аваис М., Гу В., Дламини Г., Холматова З., Суччи Дж. An experience in automatically extracting CAPAs from code repositories // arXiv.org. 2022. URL: https://arxiv.org/pdf/2212.09910

[2] Bugayenko Y., Daniakin K., Farina M., Jolha F., и др. Extracting corrective actions from code repositories // В сб.: Proceedings of the 19th International Conference on Mining Software Repositories (MSR 2022). ACM, 2022. DOI: https://dl.acm.org/doi/abs/10.1145/3524842.3528517

[3] Холматова З., Корбашов В., Педрич В., Суччи Д. A meta-analytical comparison of Naive Bayes and Random Forest for software defect prediction // ResearchGate. 2021. URL: https://www.researchgate.net/publication/350459831_A_meta-analytical_comparison_of_Naive_Bayes_and_Random_Forest_for_software_defect_prediction

[4] Examining the Success of an Open Source Software Project Analysing Its Repository // Zenodo. 2025. DOI: https://doi.org/10.5281/zenodo.10046579

[5] Di Bella E., Tamburri D.A., Serebrenik A., Storey M.-A., Melegati J., Ferreira M. GitHub Projects: Quality Analysis of Open-Source Software // В сб.: Proceedings of the 10th International Conference on Open Source Systems. Cham: Springer, 2014. С. 159–169. URL: https://link.springer.com/chapter/10.1007/978-3-319-13734-6_6

[6] Utkin L. V. An imprecise deep forest for classification // Expert Systems with Applications. 2020. Т. 141. С. 112978. URL: https://www.sciencedirect.com/science/article/pii/S0957417419306967

[7] Jain A. K., Murty M. N., Flynn P. J. The k-means Algorithm: A Comprehensive Survey and Performance Evaluation // Electronics. 2020. Т. 9, № 8. DOI: https://www.mdpi.com/2079-9292/9/8/1295

[8] Pícha P. Detecting software development process patterns in project data // В кн.: Proceedings of the 23rd International Conference on Soft Computing MENDEL 2019. Brno: Springer, 2019. URL: https://otik.uk.zcu.cz/handle/11025/37196

[9] Github API documentation. URL: https://docs.github.com/en/rest?apiVersion=2022-11-28

[10] PyGithub documentation. URL: https://pygithub.readthedocs.io/en/stable/

[11] FDA — Corrective and Preventive Actions (CAPA). URL: https://www.fda.gov/inspections-compliance-enforcement-and-criminal-investigations/inspection-guides/corrective-and-preventive-actions-capa

[12] Shehab M. A., Khreich W., Hamou-Lhadj A. и др. Commit-Time Defect Prediction Using One-Class Classification. 2024. URL: https://users.encs.concordia.ca/~abdelw/papers/JSS24-OCC_preprint.pdf

[13] Heričko T., Šumak B. Commit Classification Into Software Maintenance Activities: A Systematic Literature Review. 2023. URL: https://www.researchgate.net/profile/Samesun-Singh/post/I_need_a_question_depending_on_PICO_frame_work_related_to_operating_system_can_anyone_suggest_me/attachment/64ef8b7e806fe2503d067dd1/AS%3A11431281184732300%401693420414326/download/Heri%C4%8Dko_COMPSAC23_paper-1.pdf

[14] Heričko T., Brdnik S., Šumak B. Commit Classification Into Maintenance Activities Using Aggregated Semantic Word Embeddings of Software Change Messages // SQAMIA 2022: Workshop on Software Quality, Analysis, Monitoring, Improvement, and Applications, Novi Sad, Serbia, September 11–14, 2022. CEUR Workshop Proceedings, Vol. 3237. URL: https://ceur-ws.org/Vol-3237/paper-her.pdf

[15] Sazid Y., Kuri S., Ahmed K. S., Satter A. Commit Classification into Maintenance Activities Using In-Context Learning Capabilities of Large Language Models. 2024. URL: https://www.scitepress.org/Papers/2024/126867/126867.pdf