СОДЕРЖАНИЕ

Введение 3
Глава 1. Исследование методов и средств формирования CAPAs на основе изменений репозитория кода 5
1.1. Современные подходы к формированию CAPA на основе анализа репозиториев кода 5
1.2. Системы управления CAPA и интеграция в процессы разработки 5
1.3. Методы анализа коммитов и обучение на данных репозиториев 6
1.3.1. Предсказание дефектных коммитов 6
1.3.2. Классификация коммитов по типам работ (Commit Classification) 7
1.4. Инструменты и системы автоматизации CAPA 8
1.5. Диаграмма вариантов использования 11
1.6. Выводы 13
Глава 2. Проектирование системы формирования CAPAs на основе изменений репозитория кода 13
2.1. Требования к системе 13
2.2. Нефункциональные требования 14
2.3. Выбор технологий и инструментов 15
2.4. Архитектура системы 16
2.4.1. Компонент сбора данных 16
2.4.2. Компонент статического анализа и формирования метрик 17
2.4.3. Компонент классификации 18
2.4.4. Компонент генерации рекомендаций 18
2.4.5. Веб-приложение визуализации на Dash 18
2.5. Диаграмма классов 19
2.6. Заключение 21
Глава 3. Реализация системы формирования CAPAs на основе изменений репозитория кода 22
3.1. Извлечение и обработка данных из GitHub 22
3.1.1. Аутентификация и инициализация подключения 22
3.1.2. Получение списка коммитов 24
3.1.3. Извлечение деталей коммита и подсчёт метрик 25
3.2. Интеграция модели CommitRiskModel 27
3.2.1. Постановка задачи и необходимость генерации псевдометок 27
3.2.2. Код генерации псевдометок 28
3.2.3. Выбор и обработка признаков 29

3.2.4. Обучение классификатора 29
3.2.5. Предсказание риска и вероятностей 30
3.2.6. Интерпретация модели — важность признаков 30
3.3. Реализация панели визуализации на фреймворке Dash 31
3.3.1. Структура и организация интерфейса 31
3.3.2. Пример построения гистограммы с использованием plotly.express 32
3.3.3. Динамическое обновление интерфейса и фильтрация данных 33
3.3.4. Генерация и отображение рекомендаций CAPA 34
3.4. Интеграция компонентов в единую систему 35
3.5. Выводы 36
Глава 4. Тестирование системы формирования CAPAs на основе изменений репозитория кода 36
4.1. Введение 36
4.2. Методика проведения тестирования 37
4.3. Сравнение моделей классификации 38
4.4. Модульное тестирование 41
4.5. Нагрузочное тестирование 43
4.5.1. Обзор и интерпретация результатов 44
4.5.2. Описание страницы рекомендаций по коммитам 48
4.6. Выводы 49
Заключение 51
Словарь терминов 76
Список использованных источников 77
Библиографический список 78

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

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 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 САРА 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 САРА

На практике для генерации САРА 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 САРА. Такой подход максимально автоматизирован и интегрируется через механизмы GitHub (было реализовано создание pull request с рекомендациями САРА). 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 инструментов формирования CAPA

Инструмент	**Подход**	**Открытый исходный код**	**CI/CD интеграция**	**Анализ истории коммитов**	**Статический анализ кода**	**Выдача CAPA**	**Детализация рекомендаций**
MasterControl CAPA	QMS платформа	Нет	Нет	Нет	Нет	Осуществляется вручную	Полный аудит
CommitGuru	JIT-Defect Prediction	Да	Нет	Да	Частично	Нет, только risk-scope	Только степень риска, вероятность дефекта
@0capa (TOM)	Бот-анализ репозитория	Да	Да	Да	Линтеры	Автоматическая Pull–Request	Текстовые шаблонные советы
Bugasura.io	CI-logs AI–inspector	Нет	Да	Нет	ESLint	Автоматическая Issue	Текстовые рекомендации при сбоях

Ниже — текстовое описание каждого из выбранных критериев сравнения инструментов формирования CAPA:

Подход. Определяет алгоритмическую или организационную основу, на которой построен инструмент. QMS-платформа (MasterControl CAPA) реализует классический жизненный цикл CAPA: сбор инцидента → расследование → план действий → проверка эффективности. JIT-Defect Prediction (CommitGuru) — модель, обученная на истории коммитов, предсказывает «риск» в момент фиксации изменений. Бот-анализ репозитория (@0capa/TOM) — GitHub-App, который на каждый пуш автоматически собирает метрики и создаёт PR с рекомендациями. CI-logs AI-inspector (Bugasura.io) — анализирует логи CI-пайплайна и на их основе формирует тикеты с советами.

Открытый исходный код. Показывает, можно ли самостоятельно посмотреть и модифицировать реализацию: Да (CommitGuru, @0capa) — инструменты с открытым исходным кодом, которые можно хостить и дорабатывать под свои нужды. Нет (MasterControl, Bugasura.io) — закрытые решения без публичного репозитория.

Интеграция в CI/CD. Наличие штатных плагинов/webhook’ов для автоматического запуска вместе с билдом или PR: Да (@0capa, Bugasura.io) — инструмент

подключается к GitHub Actions/Jenkins/GitLab CI и срабатывает на пуш/PR; Частичная (CommitGuru) — имеет API-доступ, но не поставляется в виде готового CI-плагина; Нет (MasterControl) — работает вне пайплайна, задачи заводятся вручную.

Анализ истории коммитов. Учитывает ли инструмент тренды и метрики прошлых коммитов, а не только текущее состояние кода: Да (CommitGuru, @0capa) — применяет JIT-модели или паттерны на всей истории изменений; Нет (MasterControl, Bugasura.io) — фокусируется только на одном коммите или логах сборки.

Статический анализ кода. Использует ли линтеры/SAST-сканеры для поиска дефектов: Линтеры (@0capa подключает pylint/ESLint/Checkstyle); ESLint (Bugasura.io анализирует только JS-файлы); Нет (MasterControl, CommitGuru) — не выполняют статический анализ.

Выдача CAPA. Как инструмент оформляет рекомендации разработчикам. Вручную (MasterControl) — CAPA создаётся пользователем в рамках QMS–процесса; Авто Pull-Request (@0capa) — бот сам открывает PR с файлом CapaRecommendations.md; Авто Issue (Bugasura.io) — генерирует тикет в GitHub Issues при падении сборки; Нет (CommitGuru) — выдаёт лишь числовой «risk–scope», без конкретных рекомендаций.

Детализация рекомендаций. Уровень подробностей и формат советов: Полный аудит (MasterControl) — многословные отчёты, шаблоны расследования, вложения; Вероятность дефекта (CommitGuru) — только процент или оценка риска, без пояснений; Шаблонные текстовые советы (@0capa) — заранее подготовленный набор рекомендаций; Текстовые рекомендации при сбоях (Bugasura.io) — описание ошибки из логов и указание на шаги CI.

1.5. Диаграмма вариантов использования

Для формализации функциональных требований к системе построим диаграмму вариантов использования. Она позволяет наглядно показать, какие внешние акторы взаимодействуют с системой и какие сервисы они должны вызывать. Основными акторами системы выступят разработчики и проектные менеджеры. Разработчик может использовать систему для просмотра результатов анализа в интерактивном дашборде, а также изучить автоматически выданные под коммиты рекомендации, полученные прямиков в репозиторий с помощью pull-request. Проектный менеджер может отслеживать общие метрики качества, такие как, на-

пример, средний объём изменений, число багфиксов, количество предупреждений, изучать тренды на дашборде, в резульате чего принимать стратегические решения по персоналу или изменению процесса разработки. Github Api - инициализирует весь конвейер: Извлекает историю коммитов и вычисляет базовые метрики (добавленные/удалённые строки, число затронутых файлов, интервалы между коммитами). Система запускает статический анализ кода (pylint, ESLint, Checkstyle и т.п.) и сохраняет предупреждения, объединяет все метрики в единый набор признаков и передаёт их модели машинного обучения. По результатам классификации формируются рекомендации для аномальных коммитов. GitHub Api создаёт в репозитории pull-request с файлом CapaRecommendations.md, содержащим эти рекомендации.

Таким образом, диаграмма Use Case будет содержать три «актора» (Разработчик, Менеджер, GitHub API) и 7 основных прецедентов: Извлечение истории коммитов, статический анализ кода, обучение модели, выявление аномалий, выдача рекомендаций, создание pull-request'а с рекомендациями, просмотр дашборда.

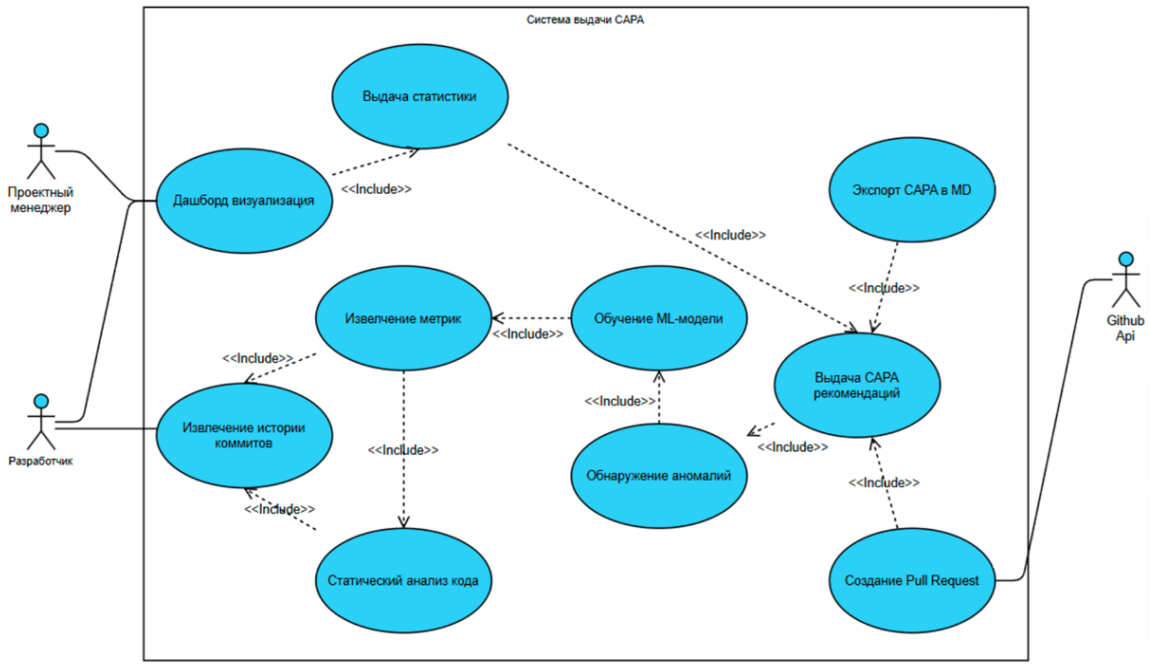


Рис.1.1. Use Case-диаграмма системы автоматического формирования CAPA

1.6. 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. 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.
- Отказоустойчивость: система должна корректно обрабатывать неуспешные запросы к API (повторять/логировать ошибки) и обеспечивать целостность данных.
- Интерпретируемость: интерфейс (дашборд) должен быть интуитивно понятным для пользователя и информативным. Информация на графиках должна быть исчерпывающей для понимания процесса разработки. Предложенные CAPA должны чётко определять найденную проблему.

– Безопасность: для взаимодействия с GitHub используются официальные API и безопасное храненилище токенов.
– Интеграция: Механизм выдачи CAPA для разработчиков оформляется в виде GitHub-Pull Request, что упростит 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

Проектируемая система реализована с учётом модульного, расширяемого и масштабируемого подхода. Архитектура разделена на несколько ключевых компонентов, которые взаимодействуют последовательно, обеспечивая надёжный сбор, обработку, анализ и визуализацию данных из репозиториев GitHub. Проектируемая система имеет модульную архитектуру с элементами сервисно-ориентированного подхода и частично реализует клиент-серверную модель.

Основная идея архитектуры — разделение задач по функциональным блокам, что обеспечивает:

- Модульность: каждый компонент реализует отдельную функцию, позволяя изменять или улучшать его независимо от остальных;
- Расширяемость: легко добавлять новые анализаторы, модели или визуализации без глобальных изменений;
- Производительность: локальное хранение и анализ кода минимизируют избыточные обращения к API GitHub и снижают задержки;
- Отказоустойчивость: чёткая обработка ошибок и проверка данных обеспечивают устойчивость работы при изменениях и сбоях.

2.4.1. Компонент сбора данных

Данный модуль отвечает за интеграцию с GitHub API и локальное клонирование репозитория с помощью библиотеки `GitPython`. Для каждого коммита собираются метаданные (SHA, автор, дата) и статистика изменений (число добавленных/удалённых строк, число изменённых файлов).

Особенностью реализации является словарь `mapping`, который связывает расширения файлов с набором соответствующих статических анализаторов (например, для `.py` — Pylint и Bandit, для `.js` — ESLint и др.). Это обеспечивает

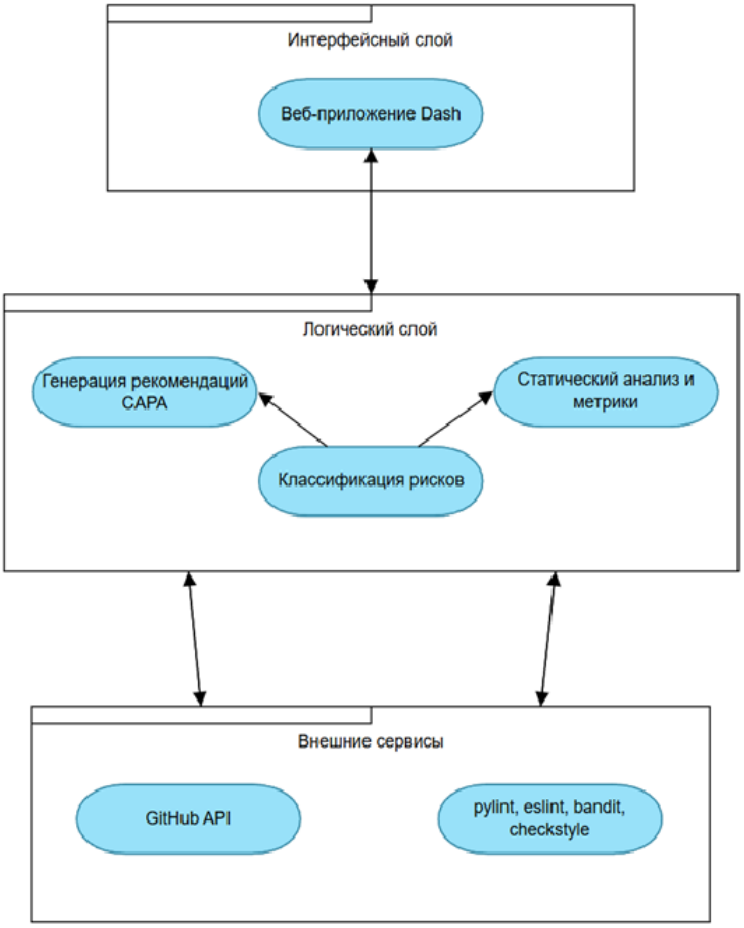


Рис.2.1. Архитектура системы анализа коммитов и формирования CAPA

гибкость и поддержку различных языков программирования без жёсткой привязки к конкретным инструментам.

Использование локального клона позволяет эффективно обращаться к содержимому файлов для глубокого анализа (например, запуск статических анализаторов, изучение диффов), а также существенно ускоряет повторные запуски системы, так как исключает необходимость повторного скачивания всей истории с GitHub. Такой подход уменьшает нагрузку на API GitHub и снижает вероятность сбоев из-за лимитов запросов.

2.4.2. Компонент статического анализа и формирования метрик

Полученные из GitHub данные обрабатываются с помощью внешних статических анализаторов — таких как `pylint`, `bandit`, `eslint`, `checkstyle` и др. — для оценки качества и безопасности кода. Количественные показатели (число

предупреждений, ошибок, сложность кода) объединяются с метриками изменений (объём изменений, частота обновлений файлов, сложность патчей, временные интервалы) в единый вектор признаков.

Архитектура позволяет легко расширять систему под новые языки программирования — для этого достаточно добавить соответствующий анализатор и подключить его к `GitHubRepoAnalyzer` через конфигурационный `mapping`.

2.4.3. Компонент классификации

Этот модуль реализует гибридный метод оценки риска коммитов, объединяющий кластеризацию и классификацию.

В отсутствие размеченных данных применяется кластеризация методом `KMeans` для формирования псевдометок, по которым далее обучается классификатор (например, `DeepForest`). Для нового коммита определяется вероятность аномальности или дефекта. Такой подход JIT-предсказания и непрерывного обучения повышает качество выявления рисковых изменений.

Архитектура компонента позволяет интегрировать любые модели, совместимые с `scikit-learn`, что обеспечивает гибкость в выборе алгоритмов.

2.4.4. Компонент генерации рекомендаций

На основе классификационных меток и анализа метрик для каждого выявленного аномального коммита формируется набор корректирующих и предупреждающих действий. Логика рекомендаций реализована в виде правил или дополнительного классификатора и может учитывать частые ошибки, объёмы изменений, недостаточное покрытие тестами и другие критерии.

Для удобства интеграции с рабочими процессами разработки система умеет автоматически создавать задачи pull request в GitHub через API, поддерживая цикл улучшения кода и контроля исправлений.

2.4.5. Веб-приложение визуализации на Dash

Интерактивный дашборд, реализованный с помощью `Dash` и `Plotly`, предоставляет удобный интерфейс для мониторинга ключевых метрик репозитория и рекомендаций CAPA. Визуализации включают гистограммы, тепловые карты, временные ряды и таблицы с возможностью фильтрации и выбора проектов.

Такой интерфейс значительно облегчает восприятие результатов анализа и принятие решений командой разработчиков.

2.5. Диаграмма классов

На рисунке 2.2 показана упрощённая UML-диаграмма основных классов и функций нашего приложения. Ниже дана её текстовая расшифровка.

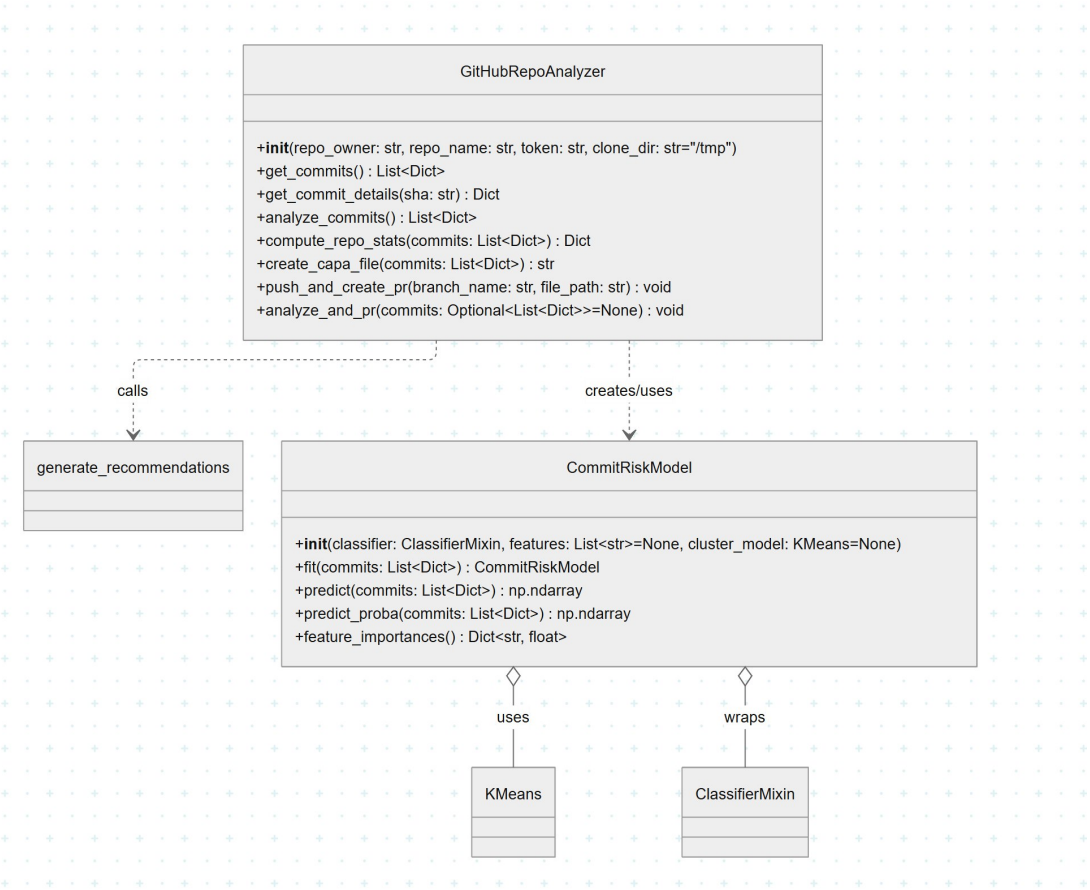


Рис.2.2. Диаграмма классов системы анализа коммитов и генерации CAPA

GitHubRepoAnalyzer

- init(repo_owner: str, repo_name: str, token: str, clone_dir: str="/tmp")
  - Клонирует (или открывает) локальный репозиторий и настраивает REST-клиент GitHub.
- get_commits(): List<Dict> — постранично получает историю коммитов через GitHub API.
- get_commit_details(sha: str): Dict — детальные сведения по одному SHA.
- analyze_commits(): List<Dict> — для каждого коммита:
  - делает `git checkout`,

    - собирает метрики (добавленные/удалённые строки, файлы, сложность, интервалы),
    - запускает анализаторы кода (pylint, eslint, checkstyle и пр.),
    - формирует словарь метрик и возвращает список таких словарей.
- compute_repo_stats(commits: List<Dict]): Dict — агрегирует статистики (среднее, стандартное отклонение, квантили) по всем коммитам.
- create_capa_file(commits: List<Dict]): str — генерирует Markdown-файл с рекомендациями CAPA.
- push_and_create_pr(branch_name: str, file_path: str): void — создаёт ветку, пушит изменения и открывает Pull Request.
- analyze_and_pr(commits: Optional<List<Dict»=None): void — объединяет анализ, модель и генерацию PR в единый конвейер.

CommitRiskModel

- init(classifier: ClassifierMixin, features: List<str]=None, cluster_model: KMeans=None) — сохраняет классификатор и модель кластеризации.
- fit(commits: List<Dict]): CommitRiskModel —
    - извлекает матрицу признаков из списка коммитов,
    - генерирует псевдолейблы через `KMeans`,
    - обучает переданный классификатор.
- predict(commits: List<Dict]): np.ndarray — возвращает предсказанные метки.
- predict_proba(commits: List<Dict]): np.ndarray — возвращает вероятность «аномальности».
- feature_importances(): Dict<str,float> — выдаёт важность каждого признака либо напрямую, либо через перестановочный анализ.

generate_recommendations

- Функция, принимающая одну запись коммита, вероятность риска, агрегированные статистики и важности признаков.
- Возвращает список текстовых рекомендаций CAPA на основе пороговых значений.

Взаимосвязи

- GitHubRepoAnalyzer вызывает `CommitRiskModel` в методе analyze_and_pr для обучения и предсказания риска.
- GitHubRepoAnalyzer вызывает функцию generate_recommendations для каждого коммита, передавая ей результаты модели и статистики репозитория.
- CommitRiskModel “wraps” любой классификатор, реализующий интерфейс `ClassifierMixin`, и “uses” `KMeans` для генерации псевдолейблов.

Таким образом, на диаграмме отражены ключевые компоненты нашего решения: извлечение и анализ коммитов, классификация риска и генерация рекомендаций CAPA, а также механизм автоматического создания Pull Request в репозитории.

2.6. 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 сопровождение. 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

В данной главе приведено подробное описание реализации системы, предназначенной для автоматического сбора, анализа и визуализации метрик коммитов из репозиториев GitHub с целью выявления потенциально проблемных изменений и генерации корректирующих и предупреждающих действий. Система состоит из нескольких ключевых компонентов: модуля сбора и обработки данных, модели классификации риска, модуля рекомендаций и веб-панели визуализации.

3.1. Извлечение и обработка данных из GitHub

Сбор данных является фундаментальным этапом системы. Для этого реализован класс `GitHubRepoAnalyzer`, отвечающий за подключение к API GitHub, локальное клонирование репозитория, анализ коммитов и вычисление множества метрик, необходимых для последующего моделирования.

3.1.1. Аутентификация и инициализация подключения

Для работы с GitHub API в конструкторе класса задаются параметры подключения и выполняется аутентификация с помощью персонального токена. В качестве примера рассмотрим фрагмент кода:

Листинг 3.1

Поля класса `GitHubRepoAnalyzer`

```
class GitHubRepoAnalyzer:
def __init__(self, repo_owner: str, repo_name: str, token:
    str, clone_dir: str = "/tmp"):
self.repo_owner = repo_owner
self.repo_name = repo_name
```

```
self.token = token
self.api_url = f"https://api.github.com/repos/{repo_owner}/{repo_name}"
self.headers = {"Authorization": f"token {token}"}

self.local_path = os.path.join(clone_dir, repo_name)
if not os.path.isdir(self.local_path):
clone_url = f"https://github.com/{repo_owner}/{repo_name}.git"
print(f"[INIT] Cloning repository {clone_url} into {self.local_path}")
Repo.clone_from(clone_url, self.local_path)
print(f"[INIT] Clone complete.")
else:
print(f"[INIT] Repository already cloned at {self.local_path}.")
self.repo = Repo(self.local_path)
print(f"[INIT] Repo object ready at {self.local_path}.")

self.complexity_re = re.compile(r"\b(if|for|while|switch|case)\b")
```

Здесь `self.headers` хранит заголовок авторизации для всех последующих HTTP-запросов к GitHub API, что позволяет безопасно получать данные без ограничений для неавторизованных пользователей. Также происходит локальное клонирование репозитория с использованием библиотеки `GitPython`. Использование локального клонирования позволяет эффективно управлять данными коммитов без необходимости организации отдельной базы данных. При первом запуске программа клонирует репозиторий в заданную директорию, после чего повторные запуски используют уже существующий локальный клон. Такой подход значительно ускоряет процесс анализа, так как исключает необходимость повторного скачивания всей истории изменений с удалённого сервера GitHub, что особенно важно для крупных проектов с большой историей коммитов. Более того, локальное хранение данных позволяет выполнять глубокий анализ исходного кода — например, запускать статические анализаторы и исследовать конкретные версии файлов в коммитах — без дополнительных сетевых задержек и ограничений API. Это снижает нагрузку на GitHub API, помогает избежать ограничений по количеству запросов и уменьшает зависимость от внешних сервисов. Таким образом, данное решение обеспечивает оптимальное соотношение между акту-

альностью данных и скоростью работы системы, обходясь при этом без сложной инфраструктуры хранения и ускоряя повторные запуски программы.

3.1.2. Получение списка коммитов

Для получения полной истории изменений в репозитории реализован метод `get_commits()`, который последовательно запрашивает страницы коммитов через GitHub API. Поскольку API возвращает данные порциями (по умолчанию до 100 элементов на страницу), метод использует механизм пагинации — он отправляет запросы, увеличивая номер страницы, пока не будет получена последняя страница с количеством коммитов меньше заданного лимита. Ниже приведён пример кода метода:

Листинг 3.2

Получение списка коммитов в методе `get_commits`

```
def get_commits(self) -> List[Dict]:
commits, page, per_page = [], 1, 100
while True:
print(f"[COMMITS] Requesting page {page}")
resp = requests.get(
f"{self.api_url}/commits",
headers=self.headers,
params={"page": page, "per_page": per_page},
)
data = resp.json()
if resp.status_code == 401:
raise RuntimeError("Bad credentials: check your GITHUB_TOKEN
    ")
if not isinstance(data, list):
print(f"[COMMITS] Unexpected response: {data}")
break
commits.extend(data)
if len(data) < per_page:
break
page += 1
print(f"[COMMITS] Total commits fetched: {len(commits)}")
return commits
```

Метод тщательно проверяет успешность каждого запроса — например, при ошибке аутентификации выбрасывается исключение с понятным сообщением. Кроме того, проверяется формат полученных данных, чтобы избежать сбоев при

неожиданном ответе API. Такая обработка ошибок повышает надёжность работы и удобство отладки.

Таким образом, метод обеспечивает полноту и корректность сбора данных, что особенно важно для больших репозиториев с тысячами коммитов, где одна страница API не может вместить всю историю. Механизм пагинации гарантирует, что все коммиты будут обработаны последовательно, без пропусков.

3.1.3. Извлечение деталей коммита и подсчёт метрик

Для каждого коммита дополнительно загружаются подробности, включая список изменённых файлов, их патчи и статистику. На основе этой информации вычисляются ключевые метрики:

- Количество добавленных и удалённых строк — суммируется по всем файлам коммита.
- Число изменённых файлов — количество файлов, затронутых изменениями.
- Интервал времени — разница во времени с предыдущим коммитом (в минутах), позволяющая оценить ритм работы.
- Оценка сложности — основана на подсчёте управляющих операторов в патчах (`if`, `for` и др.).
- Флаг багфикса — бинарная метка, указывающая на наличие в сообщении ключевых слов `fix`, `bug`, `error`.
- Результаты статического анализа — количество предупреждений и ошибок, выявленных инструментами `pylint`, `bandit`, `eslint` и `checkstyle`.

Пример кода анализа одного коммита:

Листинг 3.3

Анализ коммитов в методе `analyze_commits`

```
def analyze_commits(self) -> List[Dict]:
commits_data, file_count = [], {}
all_commits = self.get_commits()
all_commits.reverse()  # обработка в хронологическом порядке
prev_dt = None

for idx, c in enumerate(all_commits, 1):
sha = c["sha"]
det = self.get_commit_details(sha)
msg = det["commit"]["message"]
author = det["commit"]["author"]
```

```
name = author.get("name", "Unknown")
dt = datetime.strptime(author["date"], "%Y-%m-%dT%H:%M:%SZ")

files = det.get("files", [])
added = sum(f.get("additions", 0) for f in files)
deleted = sum(f.get("deletions", 0) for f in files)
hist = sum(file_count.get(f["filename"], 0) for f in files)
avg_hist = hist / len(files) if files else 0

comp = 0
for f in files:
for ln in f.get("patch", "").splitlines():
if ln.startswith("+") and not ln.startswith("+++") and self.
   complexity_re.search(ln):
comp += 1

delta = (dt - prev_dt).total_seconds() / 60 if prev_dt else
   None

metrics = {k: 0 for k in (
  "pylint_warnings","pylint_errors","bandit_issues",
  "eslint_warnings","eslint_errors","checkstyle_issues"
  )}
for f in files:
lang = self.detect_language(f["filename"])
full = os.path.join(self.local_path, f["filename"])
if lang == "python":
out = self.analyze_python_file(full)
elif lang == "javascript":
out = self.analyze_javascript_file(full)
elif lang == "java":
out = self.analyze_java_file(full)
else:
out = {}
for k,v in out.items():
metrics[k] += v

data = {
  "commit": sha,
  "author_name": name,
  "author_datetime": dt,
  "minutes_since_previous_commit": delta,
  "message": msg,
  "message_length": len(msg),
  "lines_added": added,
```

```
    "lines_deleted": deleted,
    "files_changed": len(files),
    "avg_file_history": avg_hist,
    "complexity_score": comp,
60  **metrics
  }
  commits_data.append(data)
  for f in files:
  file_count[f["filename"]] = file_count.get(f["filename"], 0)
      + 1
65 prev_dt = dt

  return commits_data
```

Данный подход позволяет запускать соответствующий статический анализатор для каждого файла в зависимости от его языка программирования. Это обеспечивает расширяемость системы и улучшает качество анализа за счёт использования специализированных инструментов для Python, JavaScript и Java.

3.2. Интеграция модели CommitRiskModel

Для автоматического выявления потенциально проблемных или «рисковых» коммитов в системе используется класс `CommitRiskModel`. Данная модель реализует гибридный подход, сочетающий алгоритмы кластеризации и классификации, что позволяет обучаться на неразмеченных данных и формировать предсказания риска для новых коммитов.

3.2.1. Постановка задачи и необходимость генерации псевдометок

В реальной задаче отсутствуют размеченные данные о том, какой коммит является проблемным, а какой — нормальным. Для обучения классификатора требуется либо вручную размеченный датасет, либо альтернативный способ получения меток. В качестве решения в `CommitRiskModel` реализован механизм генерации псевдометок (*pseudo-labels*) с помощью алгоритма кластеризации `KMeans`.

Идея такова: на основе вычисленных признаков коммитов формируется матрица признаков $X \in \mathbb{R}^{N \times F}$, где N — количество коммитов, F — число признаков. Затем алгоритм `KMeans` с числом кластеров $k = 2$ разбивает все коммиты на два кластера — один из которых интерпретируется как «нормальный», другой — как «аномальный» или «рисковый».

3.2.2. Код генерации псевдометок

Ниже приведён ключевой метод `_generate_pseudo_labels`, который реализует описанную логику. Комментарии в коде поясняют каждый шаг.

Листинг 3.4

Генерация псевдометок методом KMeans

```
def _generate_pseudo_labels(self, X: np.ndarray) -> np.
    ndarray:
# Выполняем кластеризацию методом KMeans на признаках коммит
    ов
labels = self.cluster_model.fit_predict(X)
centers = self.cluster_model.cluster_centers_

# Выбираем, какой кластер считать аномальным, сравнивая сред
    ние значения по первому признаку (lines_added)
if centers[0, 0] > centers[1, 0]:
mapping = {0: 1, 1: 0}  # Кластер 0 - аномальный, 1 - нормал
    ьный
else:
mapping = {0: 0, 1: 1}  # Кластер 1 - аномальный, 0 - нормал
    ьный

# Преобразуем метки кластеров в бинарные псевдометки {0,1}
pseudo_labels = np.vectorize(mapping.get)(labels)

return pseudo_labels
```

В этом методе:

- Метод `fit_predict` обучает `KMeans` и возвращает метки кластеров для каждого объекта.
- Центры кластеров `cluster_centers_` — это средние значения признаков для каждого кластера.
- Для определения «аномального» кластера используется правило: кластер с большим средним значением по первому признаку (число добавленных строк кода) считается более рискованным.
- Далее метки кластеров преобразуются в бинарные метки {0,1}, пригодные для обучения классификатора.

3.2.3. Выбор и обработка признаков

Класс `CommitRiskModel` по умолчанию использует следующий набор признаков, которые инкапсулируются в списке `features`:

Листинг 3.5

Список признаков модели

```
self.features = [
'lines_added',          # Число добавленных строк
'lines_deleted',        # Число удалённых строк
'files_changed',        # Количество изменённых файлов
'avg_file_history',     # Средняя частота изменений затронутых
    файлов
'message_length',       # Длина сообщения коммита
'has_bug_keyword',      # Флаг наличия ключевых слов багфикса
   в сообщении
'complexity_score'      # Оценка сложности изменения по структ
   уре патча
]
```

Признак `has_bug_keyword` является бинарным и определяется поиском ключевых слов в сообщении коммита, например, «fix», «bug», «error». Он важен, поскольку коммиты с такими словами чаще связаны с исправлением ошибок и потенциально имеют повышенный риск.

3.2.4. Обучение классификатора

После генерации псевдометок обучается классификатор — в текущей реализации используется `DeepForest` из библиотеки `deep-forest`.

Пример кода метода `fit`:

Листинг 3.6

Обучение модели CommitRiskModel

```
def fit(self, commits: List[Dict[str, Any]]):
# Извлечение матрицы признаков из списка коммитов
X = self._extract_X(commits)

# Генерация псевдометок с помощью KMeans
y = self._generate_pseudo_labels(X)

# Обучение классификатора на признаках и псевдометках
self.classifier.fit(X, y)
```

```
# Сохранение обученных данных для последующего использования
self._X, self._y = X, y
self._is_fitted = True
return self
```

Метод `_extract_X` преобразует список словарей с метриками в числовую матрицу по списку признаков `self.features`.

3.2.5. Предсказание риска и вероятностей

После обучения модель позволяет получать предсказания класса (рисковый или нормальный коммит) и вероятность риска. Это реализовано методами `predict` и `predict_proba`:

Листинг 3.7

Предсказание риска коммитов

```
def predict(self, commits: List[Dict[str, Any]]) -> np.
    ndarray:
assert self._is_fitted, "Model not fitted"
X = self._extract_X(commits)
return self.classifier.predict(X)

def predict_proba(self, commits: List[Dict[str, Any]]) -> np
    .ndarray:
assert self._is_fitted, "Model not fitted"
X = self._extract_X(commits)
# Возвращаем вероятность принадлежности к классу 1 (риск)
return self.classifier.predict_proba(X)[:, 1]
```

3.2.6. Интерпретация модели — важность признаков

Для повышения доверия к предсказаниям реализован метод `feature_importances()`, позволяющий оценить вклад каждого признака в принятие решения моделью.

Если классификатор поддерживает атрибут `feature_importances_`, он используется напрямую. В противном случае важности вычисляются через пермутационный метод:

Листинг 3.8

Вычисление важности признаков

```
def feature_importances(self) -> Dict[str, float]:
if hasattr(self.classifier, "feature_importances_"):
```

```
vals = self.classifier.feature_importances_
else:
result = permutation_importance(
self.classifier, self._X, self._y,
n_repeats=5, random_state=0, n_jobs=-1
)
vals = result.importances_mean
return dict(zip(self.features, vals))
```

На практике анализ важности показывает, что наибольший вклад в определение риска коммита вносят признаки объёма изменений (`lines_added`, `lines_deleted`) и наличие багфикс-ключевых слов (`has_bug_keyword`).

Таким образом, класс `CommitRiskModel` является ядром интеллектуальной подсистемы, позволяющей без разметки обучать модель, выявляющую потенциально проблемные коммиты. Это значительно упрощает автоматизацию мониторинга качества разработки и служит основой для формирования рекомендаций CAPA.

3.3. Реализация панели визуализации на фреймворке Dash

Для удобного представления результатов анализа коммитов и сформированных рекомендаций CAPA была разработана интерактивная веб-панель на базе Python-фреймворка `Dash`. Этот инструмент позволяет создавать адаптивные, масштабируемые и визуально привлекательные дашборды с богатым набором интерактивных графиков на основе библиотеки `Plotly`.

3.3.1. Структура и организация интерфейса

Интерфейс приложения разбит на несколько логически связанных вкладок, каждая из которых содержит соответствующую аналитику и визуализации:

- Общая статистика: гистограммы распределения ключевых метрик — количество добавленных и удалённых строк, изменённых файлов, оценки сложности коммитов. Эта вкладка служит обзором общего состояния репозитория и позволяет быстро оценить масштабы и характер изменений.
- Анализ риска: отображение важности признаков, распределение коммитов по классам риска, корреляция между риском и сложностью изменений. Здесь пользователь получает понимание, какие факторы влияют на вероятность проблемности коммита.

- Активность авторов: графики активности разработчиков и средний риск коммитов каждого автора. Позволяет выявлять наиболее активных и потенциально рискованных участников процесса.
- Карта риска файлов (File-Risk Map): визуализация взаимосвязи между частотой изменений файлов и их средним риском. Помогает выделять проблемные модули или компоненты.
- Временная шкала риска и предупреждений (Risk Timeline): динамическое отображение среднего риска и количества предупреждений по датам, что позволяет отслеживать тенденции в развитии проекта.
- Таблица коммитов с рекомендациями: интерактивная таблица с подробной информацией о каждом коммите, включающая сформированные системой рекомендации CAPA.
- Календарь активности: тепловая карта, показывающая распределение активности коммитов по дням недели и неделям.
- Качество кода по языкам: вкладки с метриками качества для основных используемых языков — Python, JavaScript, Java — на основе результатов статического анализа.

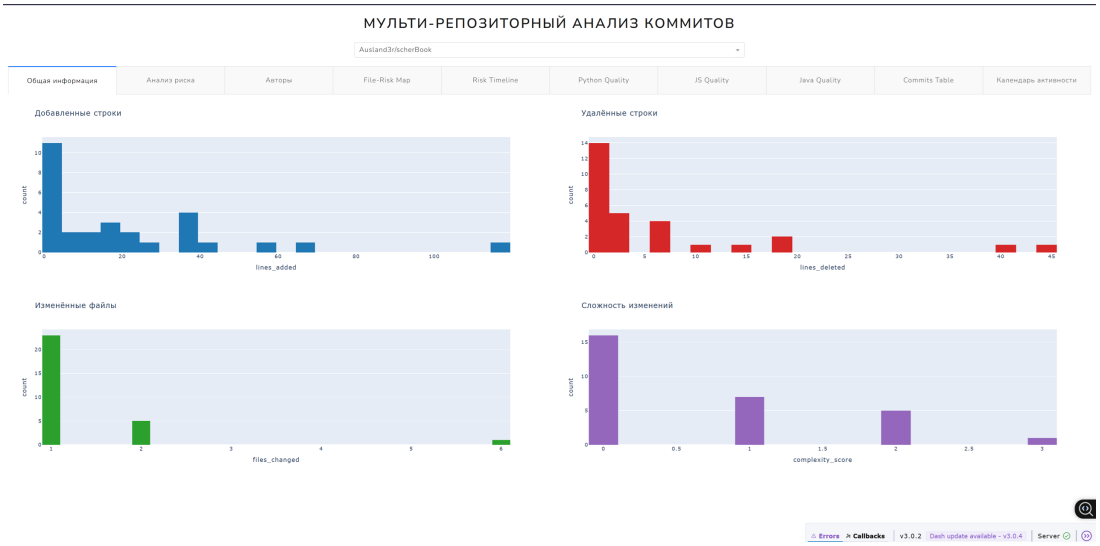


Рис.3.1. Пример страницы дашборда с общей статистикой по репозиторию

3.3.2. Пример построения гистограммы с использованием plotly.express

Для визуализации распределения числовых метрик широко используется компонент `dcc.Graph` в связке с `plotly.express`. Например, построение гистограммы по числу добавленных строк реализуется следующим образом:

Листинг 3.9

Построение гистограммы добавленных строк

```
dcc.Graph(
figure=px.histogram(
df, # DataFrame с данными коммитов
x = 'lines_added', #По оси X - число добавленных строк
nbins=30, # Количество корзин гистограммы
title='Добавленные строки',
color_discrete_sequence=['#1f77b4']  # Цвет столбцов (синий)
)
)
```

Такой подход позволяет быстро создавать красивые и информативные графики с минимальными усилиями.

3.3.3. Динамическое обновление интерфейса и фильтрация данных

Для обеспечения интерактивности и гибкости отображения данных в панели используются callback-функции Dash. Они реагируют на действия пользователя, такие как выбор репозитория, фильтрация по авторам или выбор временного диапазона, и динамически обновляют содержимое вкладок и графиков.

Ниже приведён пример callback-функции, которая обновляет вкладки с аналитикой при смене выбранного репозитория в выпадающем списке:

Листинг 3.10

Callback-функция обновления вкладок по выбранному репозиторию

```
@app.callback(
Output("tabs-container", "children"),
Input("repo-selector", "value")
)
def update_tabs(selected_repo):
if not selected_repo or selected_repo not in analyses:
return html.Div("Репозиторий не выбран или недоступен")

entry = analyses[selected_repo]
df = entry['df']
feat_imps = entry['feat_imps']
model = entry['model']

# Формирование вкладок с графиками и таблицами на основе выб
  ранного репозитория
tabs = [
create_summary_tab(df),
```

```
create_risk_analysis_tab(df, feat_imps, model.features),
create_authors_tab(df),
create_file_risk_map_tab(df),
create_risk_timeline_tab(df),
create_quality_tabs(df),
create_commits_table_tab(df),
create_activity_calendar_tab(df)
]
return dcc.Tabs(tabs)
```

В этом примере:

- `@app.callback` связывает функцию `update_tabs` с изменением значения в выпадающем списке с id `"repo-selector"`.
- Функция получает выбранный репозиторий, извлекает из предобработанных данных соответствующий набор аналитики.
- Возвращается обновлённый набор вкладок, которые отображаются в контейнере с id `"tabs-container"`.

Такой подход позволяет пользователю мгновенно переключаться между проектами и получать актуальную аналитику без перезагрузки страницы. Подобным образом можно реализовать и другие callback-функции для фильтрации по авторам, датам и т.д., обеспечивая гибкое и удобное взаимодействие с дашбордом.

3.3.4. Генерация и отображение рекомендаций CAPA

Для каждого коммита в системе формируются рекомендации корректирующих и предупреждающих действий на основе вычисленной вероятности риска и значений метрик. Логика генерации рекомендаций реализована в отдельном модуле `recommendations.py`, что обеспечивает модульность и упрощает расширение.

Пример функции генерации рекомендаций:

Листинг 3.11

Пример генерации рекомендаций CAPA

```
def generate_recommendations(commit, risk_proba, repo_stats,
    feature_importances):
recommendations = []
if risk_proba > 0.8:
recommendations.append("Очень высокий риск: провести углублё
   нное ревью.")
if commit['lines_added'] > 100:
recommendations.append("Большой объём изменений: рекомендует
   ся более тщательное тестирование.")
```

```
return recommendations
```

Рекомендации выводятся в таблице коммитов, что облегчает восприятие и принятие решений командой разработчиков.

3.4. Интеграция компонентов в единую систему

Вся система реализована как последовательный конвейер обработки данных, объединяющий сбор, анализ, генерацию рекомендаций и визуализацию:

A. Сбор и предварительная обработка — модуль `GitHubRepoAnalyzer` получает из GitHub историю коммитов, локально анализирует содержимое файлов и формирует набор признаков для каждого коммита.
B. Обучение и применение модели — класс `CommitRiskModel` обучается на полученных данных, используя псевдометки, после чего применяется для оценки риска новых коммитов.
C. Генерация рекомендаций — на основе результатов классификации формируются конкретные CAPA для каждого коммита, учитывая статистику репозитория и важность признаков.
D. Визуализация — все метрики, прогнозы и рекомендации выводятся в веб-интерфейсе Dash, обеспечивая пользователю удобный доступ к аналитике.
E. Автоматизация — реализован механизм периодического обновления данных, переобучения модели и актуализации интерфейса без участия пользователя.

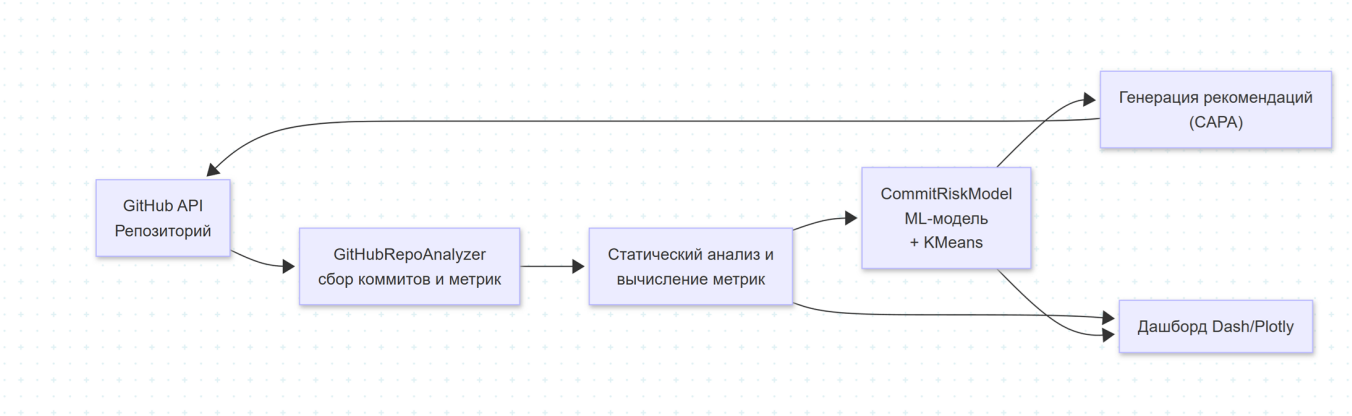


Рис.3.2. Пайплайн автоматического анализа коммитов GitHub

На диаграмме 3.2 представлена общая архитектура системы, где чётко прослеживается поток данных: от исходного кода в GitHub через модули анализа и обучения модели до визуализации и генерации CAPA.

3.5. Выводы

Реализованная система обеспечивает полный цикл автоматического мониторинга качества разработки на основе анализа коммитов, объединяя в себе сбор данных, интеллектуальную оценку риска и удобную визуализацию. Модульная архитектура позволяет легко расширять функциональность и адаптировать систему под различные проекты.

ГЛАВА 4. ТЕСТИРОВАНИЕ СИСТЕМЫ ФОРМИРОВАНИЯ CAPAS НА ОСНОВЕ ИЗМЕНЕНИЙ РЕПОЗИТОРИЯ КОДА

4.1. Введение

В ходе тестирования проверяется работоспособность разработанного инструментария и выполняются поставленные в работе задачи. Основными целями данного этапа являются:

- Проверка корректности функционирования пользовательского интерфейса и всех компонентов системы (сбора данных, анализа репозиториев, генерации рекомендаций).
- Оценка качества работы классификатора рисковых коммитов на реальных данных, полученных из активных репозиториев.
- Тестирование модели генерации CAPA (рекомендаций исправлений) для выявления ошибок и неточностей в алгоритме.
- Модульное и нагрузочное тестирование компонентов системы для выявления багов на уровне отдельных модулей и проверки масштабируемости на больших репозиториях.

Таким образом, экспериментальное тестирование охватывает проверку как функциональных характеристик (работа интерфейса, правильность алгоритмов), так и нефункциональных (надёжность, производительность) аспектов системы.

4.2. Методика проведения тестирования

Для проверки системы было выбрано несколько тестовых репозиториев с собственными разработками: `Tq`, `scherBook`, `NTO2024-2025`, а также репозитории предоставленные сторонними разработчиками и студентами, которые согласились поучаствовать в тестировании и апробации проекта: `urlagushka/polytech-labs`, `urlagushka/h8-pipeline`, `AlPakh/topotik-backend`, `AlPakh/topotik-frontend`, `Pacan4ik/tf-idf`, `Pacan4ik/tinkoff-course-spring2023`. Эти проекты содержат достаточно разнообразный код (на Java, Python, JavaScript, C++) и различную историю изменений, что обеспечивает репрезентативность данных.

Таблица 4.1

Описание репозиториев, использованных в тестировании

Репозиторий	**Язык(и)**	**Кол-во коммитов**	**Тип проекта**	**Краткое описание**
`urlagushka/polytech-labs`	C++, Java	82	Учебный проект	Набор лабораторных работ по программной инженерии; содержит решения на нескольких языках.
`urlagushka/h8-pipeline`	Python	110	ML/AI пайплайн	Инициализатор проекта на базе Hailo SDK; используется для компьютерного зрения.
`Ausland3r/Tq`	Python	40	Фреймворк для тестов	Небольшой собственный фреймворк на базе Pytest и Pydantic.
`Ausland3r/scherBook`	JavaScript	39	Веб-приложение	Клиентское приложение для платформы книгообмена; реализован основной UI.

Таблица 4.1

Репозиторий	**Язык(и)**	**Кол-во коммитов**	**Тип проекта**	**Краткое описание**
`Ausland3r/ NTO2024-2025`	Python	69	Учебный проект	Материалы к проекту "Умный город"для школьников; используется в рамках NTO.
`Pacan4ik/ tf-idf`	Python	36	Алгоритм обработки текста	Базовая реализация TF-IDF для анализа текстов на русском языке.
`Pacan4ik/ tinkoff-course-spring2023`	Java	188	Курсовой проект	Проект на Java Spring Boot с интеграцией Telegram-бота и веб–интерфейсом.
`AlPakh/ topotik-backend`	Python (FastAPI)	38	Серверная часть	REST API бэкенд с авторизацией, ORM–моделями и обработкой карт.
`AlPakh/ topotik-frontend`	JavaScript (Vue)	27	Клиентская часть	Vue-приложение с маршрутизацией, формами и подключением к API.
`jup-ag/ pyth-crosschain`	Python, Solidity	3692	Инфраструктура Web3	Форк проекта Jup-ag с доработками Pyth network: кроссчейн модуль для верификации транзакций.

4.3. Сравнение моделей классификации

Для оценки эффективности алгоритмов машинного обучения в задаче предсказания рисковых коммитов был разработан универсальный класс

`CommitRiskModel`. Он предоставляет единый интерфейс к различным классификаторам из `scikit-learn`, а также `deep-forest`. Класс реализует методы `fit()`, `predict()`, `predict_proba()`, `feature_importances()` и `evaluate_model()`.

Для сравнения были обучены три модели — RandomForest, XGBoost и DeepForest — на выборке коммитов с автоматически определёнными метками риска. Эксперименты проводились как на небольших проектах (в пределах сотни коммитов), так и на объёмном репозитории `pyth-crosschain` (3692 коммита).

Таблица 4.2

Сравнение моделей классификации на малом и большом репозитории

Модель	**Малый проект**				**Большой проект**			
	Precision	Recall	F1-score	ROC-AUC	Precision	Recall	F1-score	ROC-AUC
RandomForest	1.000	0.500	0.667	0.983	0.976	0.872	0.921	0.999
XGBoost	1.000	0.500	0.667	1.000	0.958	0.979	0.968	1.000
DeepForest	0.667	1.000	0.800	1.000	0.882	0.957	0.918	0.997

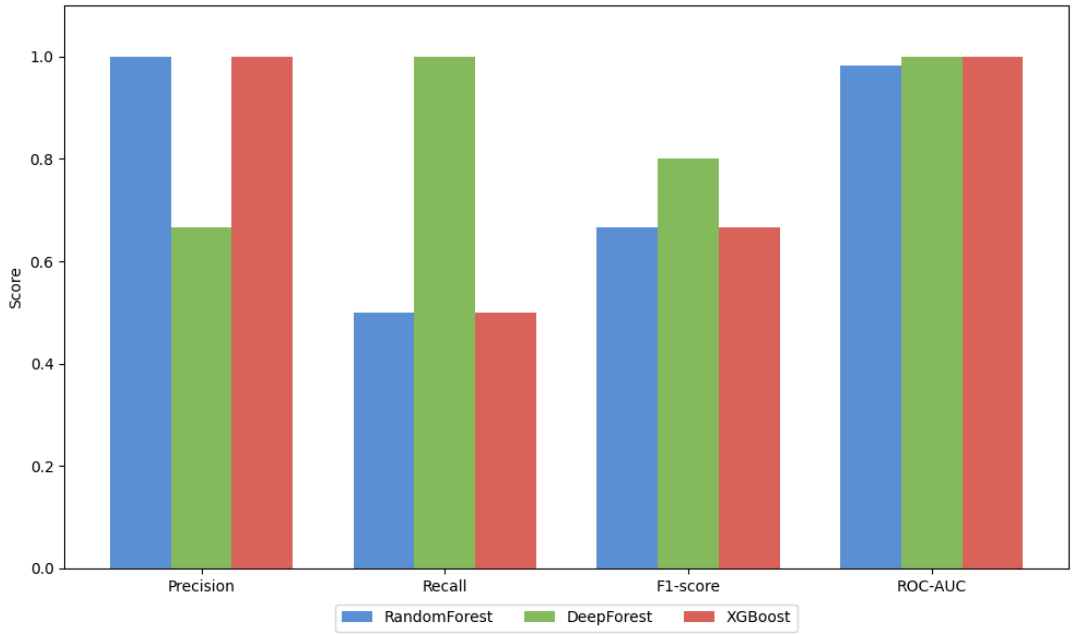


Рис.4.1. Результаты для разных моделей на малом проекте

На малом проекте DeepForest заметно превосходит другие модели по полноте (Recall = 1.0) и F1-score (0.800), что критично в условиях ограниченного количества обучающих примеров. RandomForest и XGBoost в таких условиях оказались переобученными и показали высокую точность, но пропустили половину рисковых коммитов. Это говорит о недостаточной чувствительности традиционных моделей при малом числе объектов.

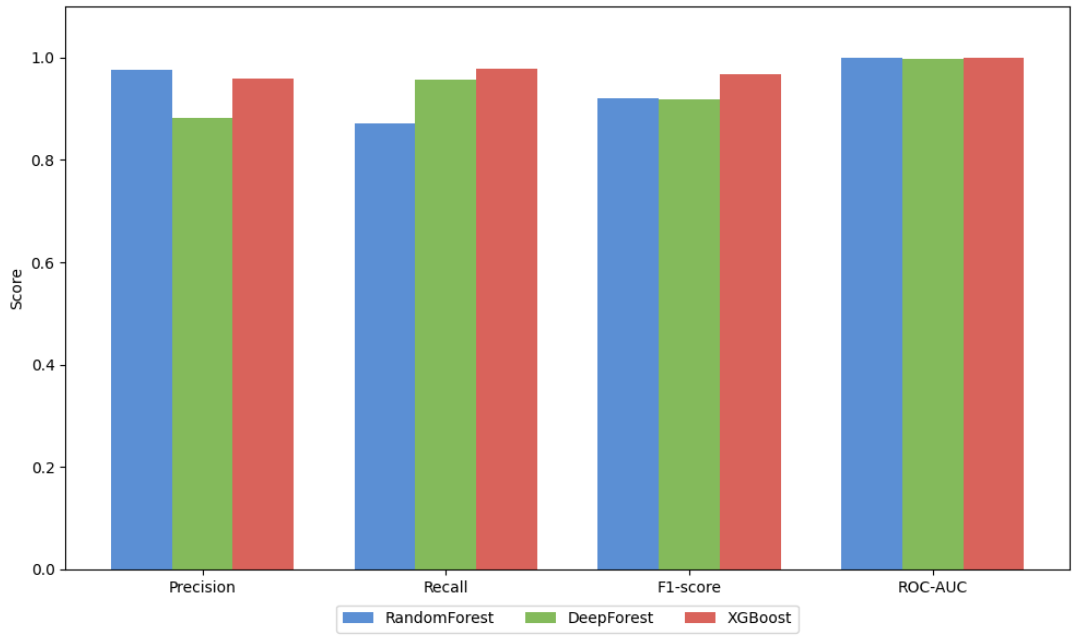


Рис.4.2. Результаты для разных моделей на большом проекте

На большом проекте (`pyth-crosschain`) ситуация меняется: XGBoost показывает лучшие результаты по F1-мере (0.968) и Recall (0.979), при этом сохраняя приемлемую точность (0.958). RandomForest также демонстрирует хорошее качество, особенно по метрике ROC-AUC (0.999). DeepForest остаётся конкурентоспособным, но немного уступает по F1-score и Precision, что можно объяснить большей сложностью структуры модели и возможным недообучением на шумных данных.

DeepForest оказывается особенно эффективной моделью в условиях малого количества данных, где критично не упустить ни одного рискового случая. Его архитектура каскадных слоёв обеспечивает высокую полноту даже на небольших выборках. Однако при наличии большого объёма данных более устойчивыми и точными оказываются градиентные модели, такие как XGBoost.

Таким образом:

- Для небольших проектов с ограниченной историей коммитов предпочтительнее использовать `DeepForest`, чтобы минимизировать пропущенные риски.
- Для крупных репозиториев рекомендуется `XGBoost` как наиболее сбалансированная и надёжная модель по метрикам качества.

В итоге можно с уверенностью сказать, что разработанный унифицированный класс `CommitRiskModel` позволяет гибко подключать и тестировать разные

алгоритмы без необходимости переписывать код, что делает систему легко масштабируемой и адаптируемой под разные сценарии.

4.4. Модульное тестирование

Для обеспечения качества и надёжности разработанного программного обеспечения выполнено модульное тестирование ключевых компонентов системы. В рамках тестирования было создано и выполнено 12 отдельных тестов, охватывающих основные сценарии работы и критичные граничные случаи. Основные направления тестирования и проверяемые аспекты включают:

- `ml_model.py`: Проверялись основные методы обучения и предсказания моделей машинного обучения. Тесты, такие как `test_model_extremes_and_balanced_cases` и `test_model_predict_proba_output`, гарантировали корректную работу метода `fit()`, проверяли способность модели обучаться на реальных и частично неполных данных, а также корректно обрабатывать ситуации с отсутствием или неполнотой входных данных. Валидация предсказаний включала проверку возвращаемых значений — скалярных вероятностей в диапазоне от 0 до 1. Данный набор тестов помогает своевременно обнаруживать ошибки, связанные с изменениями в логике обучения и предсказания, предотвращая ошибки при работе с моделями машинного обучения.
- `repository_analysis.py`:
  Проверялся процесс сбора и обработки статистики по коммитам из репозитория. В тестах, например, `test_repository_empty_and_corrupted` были учтены разные ситуации: работа с нормальным репозиторием с несколькими коммитами, а также с пустым репозиторием или с некорректными данными или отсутствующими файлами. Тесты гарантировали, что модуль не выдаёт ошибок при отсутствии данных, а возвращает корректные пустые структуры.
  Это снижает риск сбоев в работе системы при взаимодействии с нестандартными или пустыми репозиториями.
- `recommendations.py`:
  Проверялись функции генерации рекомендаций на основе анализа изменений в коммитах. Тест `test_recommendations_extreme_commit` моделирует ситуации с коммитами без добавленных строк кода, наличием уже

существующих рекомендаций. Проверялась корректность формируемых списков рекомендаций.

Это позволяет гарантировать, что рекомендации будут релевантными и не содержат дублирующей или ошибочной информации.

– `app.py`:

Этот модуль содержит ключевые функции, обеспечивающие загрузку и анализ данных из репозиториев, обучение модели машинного обучения и обновление табличного представления результатов. Тесты `test_load_and_analyze_repos`, `test_train_and_update_model` и `test_update_tabs` проверяют корректность выполнения основных операций.

Все тесты были автоматизированы с использованием фреймворка `pytest` и обеспечили покрытие ключевых функциональных частей системы более чем на 75%. Такой уровень тестового покрытия подтверждает надёжность и стабильность реализации, а также значительно упрощает сопровождение и дальнейшее развитие кода, обеспечивая своевременное выявление регрессий и ошибок.

Summary

12 tests took 690 ms.

(Un)check the boxes to filter the results.

☐ 0 Failed, ☑ 12 Passed, ☐ 0 Skipped, ☐ 0 Expected failures, ☐ 0 Unexpected passes, ☐ 0 Errors, ☐ 0 Reruns

Result	Test
Passed	test_system.py::test_model_extremes_and_balanced_cases
Passed	test_system.py::test_model_predict_proba_output
Passed	test_system.py::test_recommendations_extreme_commit
Passed	test_system.py::test_empty_commit_recommendation
Passed	test_system.py::test_model_handles_missing_fields_gracefully
Passed	test_system.py::test_model_predict_consistency
Passed	test_system.py::test_model_with_empty_input
Passed	test_system.py::test_model_with_invalid_input_types
Passed	test_system.py::test_recommendations_on_typical_and_bugfix_commits
Passed	test_system.py::test_recommendations_for_risk_bounds
Passed	test_system.py::test_load_and_analyze_repos
Passed	test_system.py::test_train_and_update_model_structure

Рис.4.3. Отчёт с результатом прогона автотестов в pytest-html

File ▲	statements	missing	excluded	coverage
app.py	129	76	0	41%
ml_model.py	88	5	0	94%
recommendations.py	55	0	0	100%
repository_analysis.py	247	73	0	70%
test_system.py	101	0	0	100%
Total	**620**	**154**	**0**	**75%**

Рис.4.4. Отчёт по покрытию автотестами в pytest-cov

4.5. Нагрузочное тестирование

Нагрузочное тестирование проводилось с целью оценки производительности системы при работе с крупными репозиториями. В качестве тестового примера был выбран репозиторий `jup-ag/pyth-crosschain` с более чем 3000 коммитов. Тестирование выполнялось на машине со следующими характеристиками: процессор AMD Ryzen 9 7900X, 16 ГБ оперативной памяти и SSD-накопитель.

- Общее время обработки полного репозитория составило 46 минут. Основная часть времени затрачивается на последовательный запуск статического анализа (Pylint, Checkstyle) для сотен файлов. Дополнительное время уходит на загрузку коммитов через API GitHub, особенно при большом количестве коммитов в репозитории.
- Наиболее ресурсоёмкой подсистемой оказался статический анализ: последовательный запуск линтеров на большом количестве файлов существенно увеличивает нагрузку на ресурсы компьютера. Максимальная загрузка CPU достигала 53%, а использование оперативной памяти — до 8 ГБ.

Для ускорения работы возможно применение кеширования результатов анализа и параллельной обработки файлов. Для избежания повторной загрузки данных с GitHub система сохраняет локальную копию репозитория. Поскольку API GitHub имеет ограничения по скорости и количеству запросов, локальный клон позволяет минимизировать обращения к API и работать с полной историей и файлами непосредственно на диске, что ускоряет повторный анализ уже загруженных репозиториев. Также чтобы не было необходимости постоянно запускать систему

для получения рекомендаций весь список рекомендаций сохраняется локально в md файл и отправляется в отдельную ветку в удаленном репозитории.

В целом система стабильно справлялась с обработкой крупного репозитория, обеспечивая корректные результаты и бесперебойную работу. Максимальная нагрузка приходилась на этап анализа качества кода. Полученные результаты подтверждают, что разработанные компоненты способны эффективно работать с реальными проектами значительных размеров.

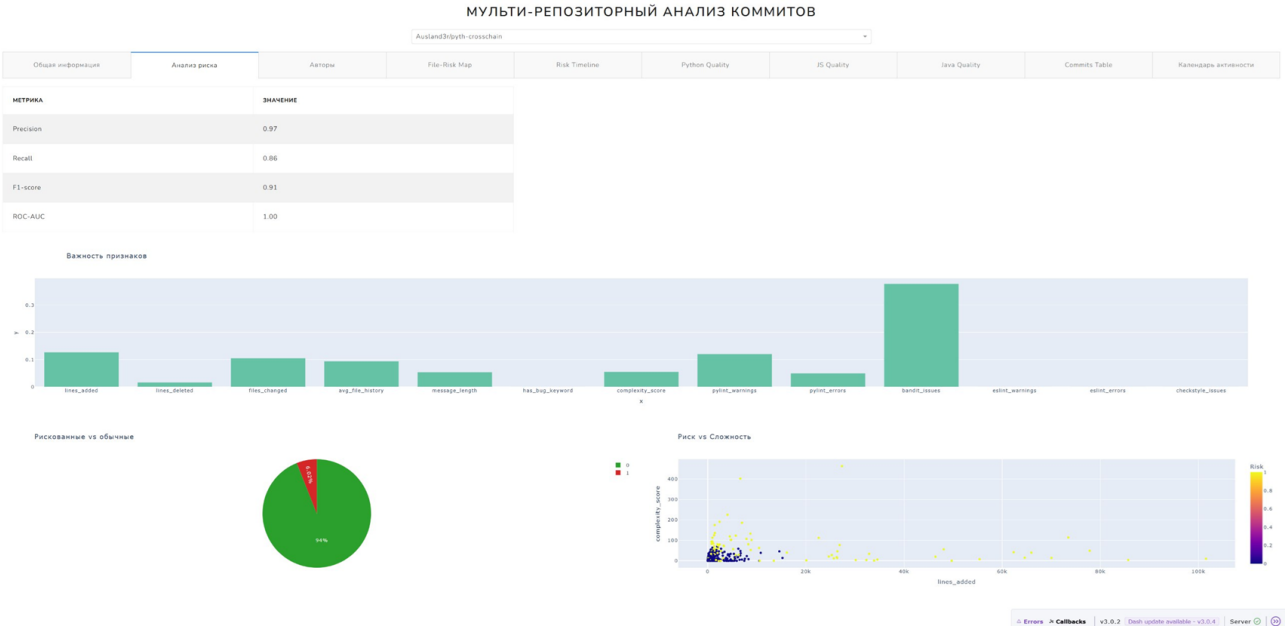


Рис.4.5. Запущенная система с репозиторием `jup-ag/pyth-crosschain`

4.5.1. Обзор и интерпретация результатов

Визуализация результатов анализа коммитов позволяет быстро выявлять тенденции в проекте. Рассмотрим пример репозитория `Ausland3r/NTO2024-2025`. На рисунках приведены разные аспекты анализа:

Рисунок 4.6 показывает гистограммы базовых метрик коммитов. Видно, что большинство коммитов содержит меньше 10 добавленных или удалённых строк, а также влияет не более чем на 5 файлов. Сложность изменений (четвёртый график) в большинстве случаев небольшая. Такие диаграммы позволяют визуально оценить, что значительная часть коммитов малых по размеру и сложности, что характерно для аккуратного ведения проекта.

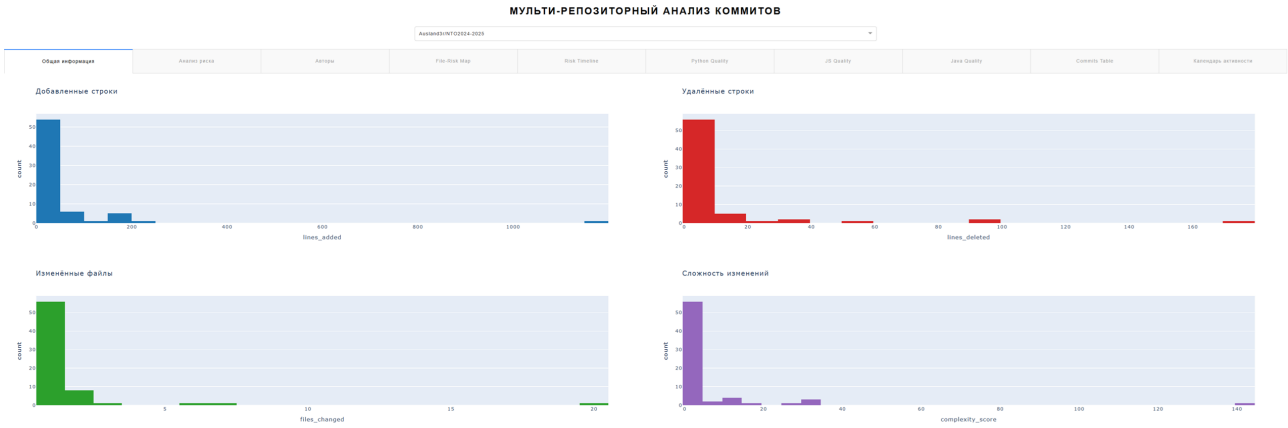


Рис.4.6. Распределение изменений в коммитах: добавленные и удалённые строки, количество изменённых файлов и сложность изменений для проекта `Auslandr/NTO2024-2025`.

На Рисунке 4.7 представлена информация об эффективности классификатора на этом репозитории. В таблице видим Precision=0.40, Recall=1.00, что соответствует метрикам модели на этих данных. Круговая диаграмма показывает, что около `11.8%` коммитов отмечены как рисковые (красным цветом). Справа виден график «Risk vs Complexity»: наблюдается тенденция, что коммиты с большей сложностью имеют более высокий риск (жёлтым - рисковые коммиты). Данный анализ помогает подтвердить, что алгоритм верно выделяет несколько потенциально проблемных коммитов (в основном с большой сложностью), и диаграммы наглядно демонстрируют распределение рисковых изменений.

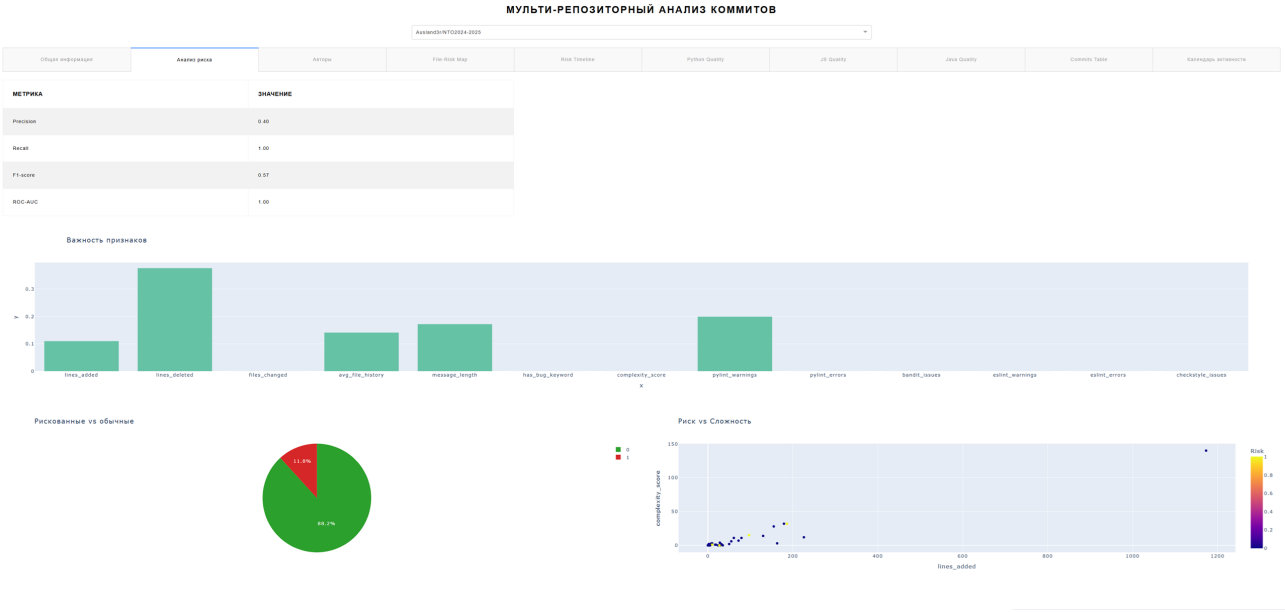


Рис.4.7. Метрики классификации и распределение рисковых коммитов для репозитория `Ausland3r/NTO2024-2025`. Таблица показывает качество модели (Precision, Recall, F1, ROC-AUC), диаграмма слева — долю рисковых коммитов (красным), справа — зависимость риска от сложности.

Рисунок 4.8 иллюстрирует вклад разных разработчиков. Слева видно, что основной объём коммитов внесли `Ausland3r` и `DenisovDmitrii` (по 30 коммитов каждый), остальные авторы — единичные вклады. Справа график показывает средний риск по автору: например, `Fliegende_Rehe` (средний риск 0.4) выделяется как относительный «рисковый» автор, хотя у него было меньше коммитов. Такая визуализация помогает в определении, кто из участников при текущем анализе вносит больше потенциально проблемных изменений.

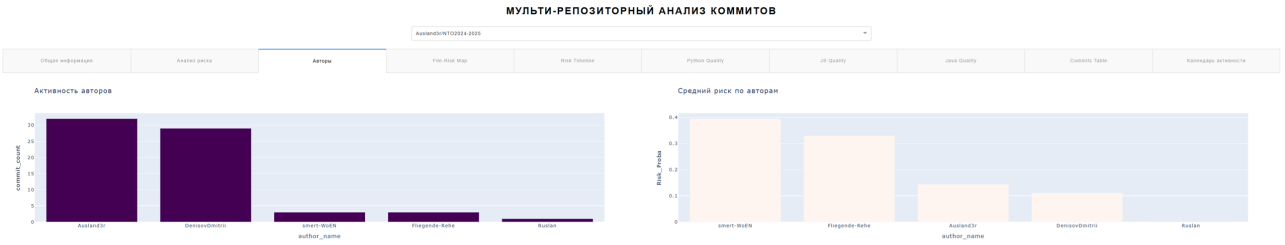


Рис.4.8. Активность авторов и средний риск на автора для проекта `Ausland3r/NTO2024-2025`. Слева — число коммитов на автора, справа — усреднённый риск.

На Рисунке 4.9 представлена файловая карта: по горизонтали — число изменений файла (`change_count`), по вертикали — средний риск изменений этого файла. Замечено, что файл `Task/task1.py` менялся 8 раз и имеет средний риск 0.38 (отмечен на графике). Большинство же файлов имеют низкий риск.

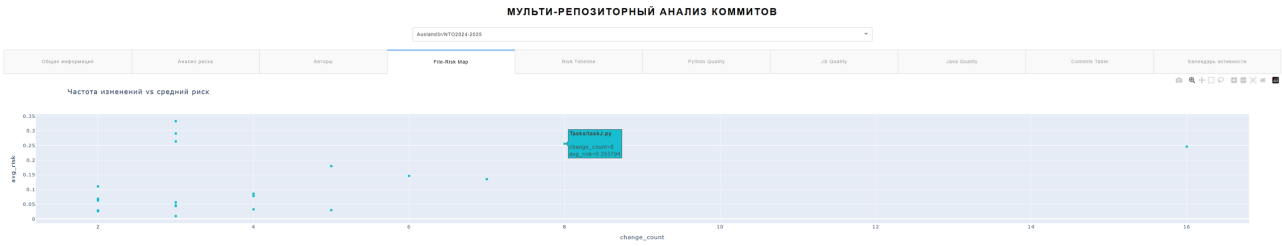


Рис.4.9. Файловая карта риска (частота изменений vs средний риск) для `Ausland3r/NTO2024-2025`. Точка `Task/taskJ.py` выделена как файл с 8 изменениями и средним риском 0.25.

Такая диаграмма позволяет выявлять «горячие точки» проекта — файлы, часто изменяющиеся и с высоким риском, требующие внимания.

Наконец, Рисунок 4.10 демонстрирует динамику проекта. По синей линии видно, что средний риск коммитов постепенно рос с ноября 2024 по декабрь 2024. В мае 2025 на проекте снова были внесены изменения. Оранжевые столбцы отображают количество предупреждений статического анализа во времени. Видно несколько пиков предупреждений в начале проекта; далее они стабилизировались на низком уровне. Такая временная диаграмма подчёркивает, как со временем изменялась стабильность проекта, и позволяет своевременно заметить всплески риска или предупреждений. В целом приведённые визуализации показывают, что проект велся относительно аккуратно.

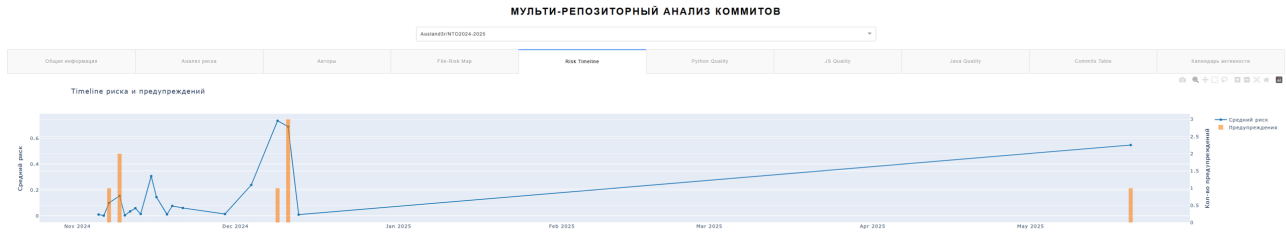


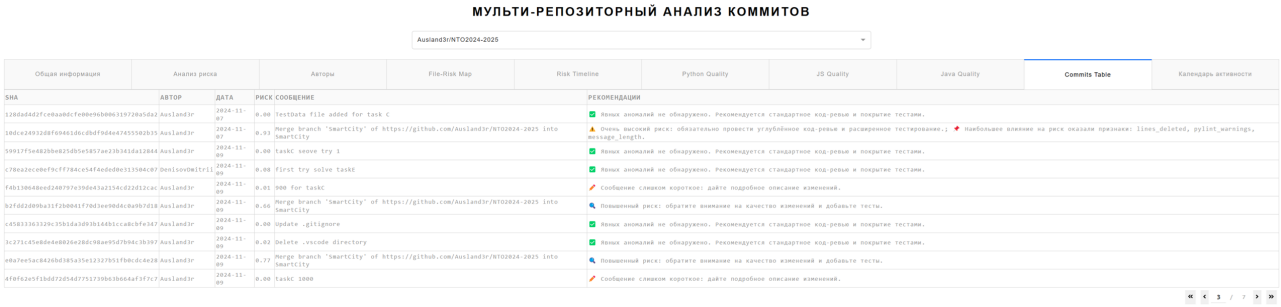
Рис.4.10. Временная шкала риска и предупреждений для `Ausland3r/NTO2024-2025`. Синяя линия — средний риск коммитов с течением времени, оранжевые столбцы — число предупреждений статического анализа.

Визуальные представления, такие как вышеперечисленные, существенно упрощают анализ состояния проекта: они помогают быстро выделить проблемные области (рисковые коммиты, ответственные авторы, модифицируемые файлы и т.д.), которые трудно заметить при простом чтении логов. Аналитическая панель с

такими графиками позволяет команде разработки эффективнее контролировать качество кода и прогнозировать потенциальные риски.

4.5.2. Описание страницы рекомендаций по коммитам

На данной странице отображаются рекомендации по каждому коммиту репозитория, сгенерированные на основе анализа риска и метрик изменений. Рекомендации призваны помочь разработчикам и ревьюерам быстро оценить потенциальные проблемы и принять соответствующие меры.



МУЛЬТИ-РЕПОЗИТОРНЫЙ АНАЛИЗ КОММИТОВ

Ausland3r/NTO2024-2025

Общая информация | Анализ риска | Авторы | File-Risk Map | Risk Timeline | Python Quality | JS Quality | Java Quality | Commits Table | Календарь активности

SHA	АВТОР	ДАТА	РИСК	СООБЩЕНИЕ	РЕКОМЕНДАЦИИ
128da4d2fce0aa00cfe00e96b0063197204a5da2	Ausland3r	2024-11-07	0.00	testOata file added for task C	✅ Явных аномалий не обнаружено. Рекомендуется стандартное код-ревью и покрытие тестами.
10dce24932d8f69461d6cd8df9d4e4745550 2b35	Ausland3r	2024-11-07	0.93	Merge branch 'SmartCity' of https://github.com/Ausland3r/NTO2024-2025 into SmartCity	⚠️ Очень высокий риск: обязательно провести углублённое код-ревью и расширенное тестирование.; 📌 Наибольшее влияние на риск оказали признаки: lines_deleted, pylint_warnings, message_length.
59917f5e482bbe825db5e5857ae23b361da12844	Ausland3r	2024-11-09	0.00	taskC seove try 1	✅ Явных аномалий не обнаружено. Рекомендуется стандартное код-ревью и покрытие тестами.
c78ea2ec0bef9cff784ce54f4e0ed0e313504c07	DenisovDmitri	2024-11-09	0.08	first try solve taskE	✅ Явных аномалий не обнаружено. Рекомендуется стандартное код-ревью и покрытие тестами.
f4b130648eed240797e39de43a2154cd22d12cac	Ausland3r	2024-11-09	0.01	900 for taskC	✏️ Сообщение слишком короткое: дайте подробное описание изменений.
b2fdd2d09ba31f2b0041f70d3ee90d4c0a9b7d18	Ausland3r	2024-11-09	0.66	Merge branch 'SmartCity' of https://github.com/Ausland3r/NTO2024-2025 into SmartCity	🔍 Повышенный риск: обратите внимание на качество изменений и добавьте тесты.
c45833363329c35b1da3d93b164b1cca8cbfe347	Ausland3r	2024-11-09	0.00	Update .gitignore	✅ Явных аномалий не обнаружено. Рекомендуется стандартное код-ревью и покрытие тестами.
1c271c45e8de4e8026e28dc98ae95d7b94c3b397	Ausland3r	2024-11-09	0.02	Delete .vscode directory	✅ Явных аномалий не обнаружено. Рекомендуется стандартное код-ревью и покрытие тестами.
e0a7ee5ac8426bd385a35e12327b51fb0cdc4e28	Ausland3r	2024-11-09	0.77	Merge branch 'SmartCity' of https://github.com/Ausland3r/NTO2024-2025 into SmartCity	🔍 Повышенный риск: обратите внимание на качество изменений и добавьте тесты.
4f0f62e5f1bdd72d54d7751739b63b664af3f7c7	Ausland3r	2024-11-09	0.00	taskC 1000	✏️ Сообщение слишком короткое: дайте подробное описание изменений.

« ‹ 3 / 7 › »

Рис.4.11. Таблица рисков и предупреждений для `Ausland3r/NTO2024-2025`.

Основные виды рекомендаций включают:

- Высокий риск (например, вероятность риска выше 0.8):
  - Рекомендуется углублённое код-ревью и расширенное тестирование.
  - При наличии — показаны наиболее значимые признаки, повлиявшие на оценку риска, чтобы понять причины высокой оценки.
- Повышенный риск (риск в диапазоне 0.5–0.8):
  - Совет обратить внимание на качество изменений и добавить тесты.
- Качество сообщения коммита:
  - Сообщения с длиной менее 15 символов получают рекомендацию расширить описание изменений.
  - Очень длинные сообщения (более 200 символов) предлагается структурировать или сократить.
- Объём изменений:
  - Если количество изменённых строк значительно превышает среднее по репозиторию (свыше среднего плюс два стандартных отклонения), рекомендуется разбивать изменения на более мелкие логические части.

- Специфические сигналы:
    - Коммиты с ключевыми словами, указывающими на исправление багов, сопровождаются рекомендацией проверить наличие регрессионных тестов и обновление документации.

Реализация рекомендаций основана на анализе различных метрик коммита, таких как длина сообщения, объём изменений, количество изменённых файлов, а также вероятности риска, вычисленные моделью. Это позволяет делать выводы не только на основе простых пороговых значений, но и учитывать специфику репозитория (через статистики по проекту) и значимость отдельных признаков риска.

Таким образом, представленные рекомендации реально соответствуют конкретному коммиту и его характеристикам, помогая в раннем выявлении потенциальных проблем и улучшении процесса код-ревью.

Пример рекомендаций по коммиту:

- Очень высокий риск: обязательно провести углублённое код-ревью и расширенное тестирование.
- Наибольшее влияние на риск оказали признаки: lines_deleted, pylint_warnings, message_length.
- Сообщение слишком короткое: дайте подробное описание изменений.
- Объём изменений (150) значительно превышает среднее (50.3). Разбейте коммит на более мелкие логические части.

Такая система рекомендаций повышает прозрачность оценки качества коммитов и способствует улучшению практик разработки в команде.

4.6. Выводы

В результате экспериментального тестирования подтверждена корректность реализации разработанной системы: интерфейс и функциональные блоки работают согласно заданию, а обученный классификатор рисковых коммитов показывает адекватные метрики на реальных данных. Успешно протестирована модель генерации рекомендаций CAPA: выявленные рекомендации соответствуют ожидаемым паттернам исправлений. Модульные тесты покрыли основные пути выполнения,

включая граничные сценарии, что свидетельствует о надёжности кода. Нагрузочные тесты показали, что система масштабируема — даже при анализе тысячи коммитов реакция остаётся предсказуемой, хотя оптимизации статического анализа целесообразны для ускорения. Сильными сторонами подхода являются комплексность анализа (объединение статики кода, анализа коммитов и визуализации) и высокая адаптивность модели (универсальный класс классификатора позволяет легко тестировать новые алгоритмы). Подробные визуализации дают мощный инструмент мониторинга проекта. Возможные улучшения: расширение набора признаков за счёт динамических метрик (например, метрики сборки или покрытия тестами), а также использование реальных размеченных данных для обучения классификатора вместо эвристической псевдоразметки. При дальнейшем развитии можно добавить автоматические рекомендации по приоритетам исправлений на основе выявленных рисков. В целом, проведённое экспериментальное тестирование подтвердило, что разработанный подход позволяет эффективно выявлять и анализировать рисковые изменения в репозиториях, обеспечивая корректность работы системы и демонстрируя перспективу для дальнейшего улучшения инструментов разработки.

ЗАКЛЮЧЕНИЕ

В ходе данной работы была разработана система автоматического анализа коммитов, направленная на выявление аномалий и формирование корректирующих и предупреждающих действий. 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.

ПРИЛОЖЕНИЕ

Листинг 4.1

repository_analysis.py

```
# repository_analysis.py

import os
import requests
import re
import json
import subprocess
from datetime import datetime
import pandas as pd
from git import Repo, GitCommandError
from typing import List, Dict, Optional

from xgboost import XGBClassifier

from ml_model import CommitRiskModel
from recommendations import generate_recommendations

LANGUAGE_ANALYZERS = {
  '.py': 'python',
  '.js': 'javascript',
  '.ts': 'javascript',
  '.java': 'java',
}

class GitHubRepoAnalyzer:
def __init__(
self,
repo_owner: str,
repo_name: str,
token: str,
clone_dir: str = "/tmp",
):
self.repo_owner = repo_owner
self.repo_name = repo_name
self.token = token
self.api_url = f"https://api.github.com/repos/{repo_owner}/{
   repo_name}"
self.headers = {"Authorization": f"token {token}"}

self.local_path = os.path.join(clone_dir, repo_name)
```

```
if not os.path.isdir(self.local_path):
clone_url = f"https://github.com/{repo_owner}/{repo_name}.
   git"
print(f"[INIT] Cloning repository {clone_url} into {self.
   local_path}")
Repo.clone_from(clone_url, self.local_path)
print(f"[INIT] Clone complete.")
else:
print(f"[INIT] Repository already cloned at {self.local_path
   }.")
self.repo = Repo(self.local_path)
print(f"[INIT] Repo object ready at {self.local_path}.")

self.complexity_re = re.compile(r"\b(if|for|while|switch|
   case)\b")

def get_commits(self) -> List[Dict]:
print("[COMMITS] Fetching commits via GitHub API")
commits, page, per_page = [], 1, 100
while True:
print(f"[COMMITS] Requesting page {page}")
resp = requests.get(
f"{self.api_url}/commits",
headers=self.headers,
params={"page": page, "per_page": per_page},
)
data = resp.json()
if resp.status_code == 401:
raise RuntimeError("Bad credentials: check your GITHUB_TOKEN
   ")
if not isinstance(data, list):
print(f"[COMMITS] Unexpected response: {data}")
break
commits.extend(data)
print(f"[COMMITS] Retrieved {len(data)} commits in page {
   page}.")
if len(data) < per_page:
print(f"[COMMITS] Less than {per_page} commits on page {page
   }, finishing.")
break
page += 1
print(f"[COMMITS] Total commits fetched: {len(commits)}")
return commits

def get_commit_details(self, sha: str) -> Dict:
```

```
print(f"[DETAILS] Fetching details for commit {sha}")
resp = requests.get(f"{self.api_url}/commits/{sha}", headers
   =self.headers)
return resp.json()

def detect_language(self, filename: str) -> str:
_, ext = os.path.splitext(filename.lower())
return LANGUAGE_ANALYZERS.get(ext, "")

def analyze_python_file(self, full_path: str) -> Dict[str,
   int]:
pyl_w = pyl_e = bandit = 0
try:
r = subprocess.run(
["pylint", full_path, "--output-format=json"],
stdout=subprocess.PIPE, stderr=subprocess.DEVNULL, text=True
)
msgs = json.loads(r.stdout or "[]")
for m in msgs:
if m.get("type") == "error":
pyl_e += 1
else:
pyl_w += 1
except Exception:
print(f"[ANALYZE][PY] Pylint failed on {full_path}")
try:
r = subprocess.run(
["bandit", "-f", "json", "-r", full_path],
stdout=subprocess.PIPE, stderr=subprocess.DEVNULL, text=True
)
js = json.loads(r.stdout or "{}")
bandit = len(js.get("results", []))
except Exception:
print(f"[ANALYZE][PY] Bandit failed on {full_path}")
return {"pylint_warnings": pyl_w, "pylint_errors": pyl_e, "
   bandit_issues": bandit}

def analyze_javascript_file(self, full_path: str) -> Dict[
   str,int]:
w = e = 0
try:
r = subprocess.run(
["eslint", full_path, "-f", "json"],
stdout=subprocess.PIPE, stderr=subprocess.DEVNULL, text=True
)
```

```
arr = json.loads(r.stdout or "[]")
for file_res in arr:
for msg in file_res.get("messages", []):
if msg.get("severity") == 2:
e += 1
else:
w += 1
except Exception:
print(f"[ANALYZE][JS] ESLint failed on {full_path}")
return {"eslint_warnings": w, "eslint_errors": e}

def analyze_java_file(self, full_path: str) -> Dict[str,int
    ]:
count = 0
try:
r = subprocess.run(
["checkstyle", "-f", "plain", full_path],
stdout=subprocess.PIPE, stderr=subprocess.DEVNULL, text=True
)
for ln in r.stdout.splitlines():
if "ERROR" in ln or "WARNING" in ln:
count += 1
except Exception:
print(f"[ANALYZE][JAVA] Checkstyle failed on {full_path}")
return {"checkstyle_issues": count}

def compute_repo_stats(self, commits: List[Dict]) -> Dict:
import pandas as pd
df = pd.DataFrame(commits)
stats = {}
for f in ['lines_added', 'lines_deleted', 'files_changed',
'avg_file_history', 'message_length', 'complexity_score']:
if f in df:
stats[f] = {
  'mean': df[f].mean(),
  'std': df[f].std(),
  'quantile_90': df[f].quantile(0.90),
  'quantile_95': df[f].quantile(0.95),
}
stats['author_stats'] = {a: {'median_lines_added': grp.
    median()}
  for a, grp in df.groupby('author_name')['lines_added']}
if 'minutes_since_previous_commit' in df:
stats['commit_interval'] = {'median': df['
    minutes_since_previous_commit'].median()}
```

```
return stats

def analyze_commits(self) -> List[Dict]:
print("[ANALYZE]Starting commit-by-commit analysis")
commits_data, file_count = [], {}

all_commits = self.get_commits()
all_commits.reverse()
prev_dt = None

for idx, c in enumerate(all_commits, 1):
sha = c["sha"]
print(f"[ANALYZE] ({idx}/{len(all_commits)}) Processing
   commit {sha}")
det = self.get_commit_details(sha)

try:
print(f"[GIT] Checking out {sha}")
self.repo.git.checkout(sha)
except GitCommandError:
print(f"[GIT] Cannot checkout {sha}, skipping FS analysis")

msg = det["commit"]["message"]
author = det["commit"]["author"]
name = author.get("name", "Unknown")
dt = datetime.strptime(author["date"], "%Y-%m-%dT%H:%M:%SZ")

files = det.get("files", [])
print(f"[ANALYZE]  {len(files)} files changed")

added = sum(f.get("additions", 0) for f in files)
deleted = sum(f.get("deletions", 0) for f in files)
hist = sum(file_count.get(f["filename"], 0) for f in files)
avg_hist = hist / len(files) if files else 0

comp = 0
for f in files:
for ln in f.get("patch", "").splitlines():
if ln.startswith("+") and not ln.startswith("+++") and self.
   complexity_re.search(ln):
comp += 1

delta = (dt - prev_dt).total_seconds() / 60 if prev_dt else
   None
```

```
metrics = {k: 0 for k in (
  "pylint_warnings","pylint_errors","bandit_issues",
  "eslint_warnings","eslint_errors","checkstyle_issues"
  )}
for f in files:
lang = self.detect_language(f["filename"])
full = os.path.join(self.local_path, f["filename"])
if lang:
print(f"[ANALYZE]Running {lang} analysis on {f['filename']}"
   )
if lang == "python":
out = self.analyze_python_file(full)
elif lang == "javascript":
out = self.analyze_javascript_file(full)
elif lang == "java":
out = self.analyze_java_file(full)
else:
out = {}
for k,v in out.items():
metrics[k] += v

data = {
  "commit": sha,
  "author_name": name,
  "author_datetime": dt,
  "minutes_since_previous_commit": delta,
  "message": msg,
  "message_length": len(msg),
  "lines_added": added,
  "lines_deleted": deleted,
  "files_changed": len(files),
  "avg_file_history": avg_hist,
  "complexity_score": comp,
  "file_list": [f["filename"] for f in files],
  **metrics
}
commits_data.append(data)

for f in files:
file_count[f["filename"]] = file_count.get(f["filename"], 0)
    + 1
prev_dt = dt

print(f"[ANALYZE] Completed analysis of {len(commits_data)}
   commits.")
```

```
return commits_data

def create_capa_file(self, commits: List[Dict]) -> str:
path = os.path.join(self.local_path, "CapaRecommendations.md
    ")
with open(path, "w", encoding="utf-8") as f:
f.write("CAPA Recommendations\n\n")
for c in commits:
if c.get("has_capa"):
for rec in c["capa_recommendations"]:
f.write(f"- {rec}\n")
f.write("\n")
return path

def push_and_create_pr(self, branch_name: str, file_path:
    str) -> None:
print(f"[PR] Fetching origin")
self.repo.git.fetch('origin')

base_branch = 'main'

if branch_name in self.repo.branches:
print(f"[PR] Branch {branch_name} already exists locally,
    checking out.")
self.repo.git.checkout(branch_name)
else:
print(f"[PR] Creating branch {branch_name} from origin/{
    base_branch}")
self.repo.git.checkout('-b', branch_name, f'origin/{
    base_branch}')

rel_path = os.path.relpath(file_path, self.local_path)
print(f"[PR] Adding file {rel_path}")
self.repo.index.add([rel_path])
print(f"[PR] Committing changes")
self.repo.index.commit("Add CAPA recommendations")

print(f"[PR] Pushing branch {branch_name}")
origin = self.repo.remote(name='origin')
origin.push(branch_name)

pr_data = {
  "title": "Add automated CAPA recommendations",
  "head": f"{self.repo_owner}:{branch_name}",
  "base": base_branch,
```

```
    "body": "This PR adds automatically generated corrective/
      preventive actions."
}
print(f"[PR] Opening PR via GitHub API")
response = requests.post(
f"{self.api_url}/pulls",
headers=self.headers,
json=pr_data
)
if response.status_code in (200, 201):
pr_url = response.json().get("html_url")
print(f"[PR] Pull request created: {pr_url}")
else:
print(f"[PR]Failed to create PR: {response.status_code} {
   response.text}")

def analyze_and_pr(self, commits: Optional[List[Dict]] =
   None) -> None:
if commits is None:
commits = self.analyze_commits()

if not commits:
print("No commits - пропускаем PR.")
return

model = CommitRiskModel(classifier=XGBClassifier(eval_metric
   ="logloss"))
model.fit(commits)
probs = model.predict_proba(commits)

repo_stats = self.compute_repo_stats(commits)

for commit, p in zip(commits, probs):
commit["Risk_Proba"] = float(p)
commit["has_capa"] = True
commit["capa_recommendations"] = generate_recommendations(
commit, p, repo_stats, model.feature_importances()
)

md_path = self.create_capa_file(commits)
branch = f"capa-{datetime.utcnow():%Y%m%d%H%M}"
self.push_and_create_pr(branch, md_path)
```

Листинг 4.2

ml_model.py

```python
# ml_models.py
from collections import Counter
from typing import List, Dict, Any, Optional
import numpy as np
from deepforest import CascadeForestClassifier
from sklearn.base import ClassifierMixin
from sklearn.cluster import KMeans
from sklearn.inspection import permutation_importance
from sklearn.metrics import precision_score, recall_score, f1_score, roc_auc_score
from sklearn.model_selection import train_test_split


class CommitRiskModel:
def __init__(
self,
classifier: ClassifierMixin,
features: Optional[List[str]] = None,
cluster_model: Optional[KMeans] = None
):
self.classifier = classifier
self.features = features or [
'lines_added', 'lines_deleted', 'files_changed',
'avg_file_history', 'message_length',
'has_bug_keyword', 'complexity_score'
]
self.cluster_model = cluster_model or KMeans(n_clusters=2, random_state=0, n_init=10)
self._is_fitted = False
self._X: Optional[np.ndarray] = None
self._y: Optional[np.ndarray] = None

def _extract_X(self, commits: List[Dict[str, Any]]) -> np.ndarray:
return np.array([[commit.get(f, 0) for f in self.features] for commit in commits])

def _generate_pseudo_labels(self, X: np.ndarray) -> np.ndarray:
labels = self.cluster_model.fit_predict(X)
centers = self.cluster_model.cluster_centers_

dist0 = np.linalg.norm(X - centers[0], axis=1)
dist1 = np.linalg.norm(X - centers[1], axis=1)
```

```
prob_cluster1 = dist0 / (dist0 + dist1 + 1e-8)

if centers[0, 0] > centers[1, 0]:
prob_risky = 1 - prob_cluster1
else:
prob_risky = prob_cluster1

threshold = 0.3
labels_soft = (prob_risky >= threshold).astype(int)

return labels_soft

def fit(self, commits: List[Dict[str, Any]]):
X = self._extract_X(commits)
y = self._generate_pseudo_labels(X)
self.classifier.fit(X, y)
self._X, self._y = X, y
self._is_fitted = True
return self

def predict(self, commits: List[Dict[str, Any]]) -> np.
   ndarray:
assert self._is_fitted, "Модель не обучена"
X = self._extract_X(commits)
return self.classifier.predict(X)

def predict_proba(self, commits: List[Dict[str, Any]]) -> np
   .ndarray:
assert self._is_fitted, "Модель не обучена"
X = self._extract_X(commits)
return self.classifier.predict_proba(X)[:, 1]

def feature_importances(self) -> Dict[str, float]:
if not self._is_fitted or self._X is None or self._y is None
   :
raise RuntimeError("Нужно вызвать .fit() перед
   feature_importances()")
if hasattr(self.classifier, "feature_importances_"):
vals = self.classifier.feature_importances_
else:
result = permutation_importance(
self.classifier, self._X, self._y,
n_repeats=5, random_state=0, n_jobs=-1
)
vals = result.importances_mean
```

```
return dict(zip(self.features, vals))

def evaluate_model(self, commits: List[Dict[str, Any]]) ->
   Dict[str, float]:
X = self._extract_X(commits)
y = self._generate_pseudo_labels(X)
print("[DEBUG] Метки (y) распределение:", Counter(y))

if len(set(y)) < 2:
print("[WARNING] В данных только один класс, метрики классиф
   икации не применимы.")
clf = self.classifier
clf.fit(X, y)
y_pred = clf.predict(X)
y_proba = clf.predict_proba(X)[:, 1] if hasattr(clf, "
   predict_proba") else np.zeros_like(y_pred,
dtype=float)
return {
  "precision": 0.0,
  "recall": 0.0,
  "f1_score": 0.0,
  "auc": 0.0
}

stratify_param = y if min(Counter(y).values()) > 1 else None
X_train, X_test, y_train, y_test = train_test_split(
X, y, test_size=0.2, random_state=42, stratify=
   stratify_param
)

if isinstance(self.classifier, CascadeForestClassifier):
clf = CascadeForestClassifier(random_state=42)
else:
from copy import deepcopy
clf = deepcopy(self.classifier)

clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
print("[DEBUG] y_pred распределение:", Counter(y_pred))

if hasattr(clf, "predict_proba"):
y_proba = clf.predict_proba(X_test)[:, 1]
print("[DEBUG] y_proba min/max:", y_proba.min(), y_proba.max
   ())
else:
```

```
y_proba = np.zeros_like(y_pred, dtype=float)

precision = precision_score(y_test, y_pred, zero_division=0)
recall = recall_score(y_test, y_pred, zero_division=0)
f1 = f1_score(y_test, y_pred, zero_division=0)
auc = roc_auc_score(y_test, y_proba) if len(set(y_test)) ==
    2 else 0.0

return {
  "precision": precision,
  "recall": recall,
  "f1_score": f1,
  "auc": auc
}
```

Листинг 4.3

app.py

```
# app.py

import os
from dotenv import load_dotenv
import dash
from dash import dcc, html, dash_table
from dash.dependencies import Input, Output
import dash_bootstrap_components as dbc
import plotly.express as px
import plotly.graph_objects as go
import pandas as pd
import numpy as np

from repository_analysis import GitHubRepoAnalyzer
from ml_model import CommitRiskModel
from xgboost import XGBClassifier
from deepforest import CascadeForestClassifier
from recommendations import generate_recommendations

# 1. Загрузка настроек
load_dotenv()
github_token = os.getenv('GITHUB_TOKEN')
repos = [r for r in os.getenv("GITHUB_REPOS", "").split(",")
    if r]

# 2. Сбор и анализ каждого репозитория
analyses = {}
```

```
for full_name in repos:
owner, name = full_name.split("/")
analyzer = GitHubRepoAnalyzer(owner, name, github_token)
commits = analyzer.analyze_commits()
analyzer.analyze_and_pr(commits)

# 3. Обучение модели на коммитах
#model = CommitRiskModel(XGBClassifier(eval_metric="logloss
   "))
model = CommitRiskModel(CascadeForestClassifier(random_state
   =42))
model.fit(commits)
metrics = model.evaluate_model(commits)

# 4. Подготовка DataFrame
df = pd.DataFrame(commits)
df['Risk_Proba'] = model.predict_proba(commits)
df['Risk'] = (df['Risk_Proba'] > 0.5).astype(int)

analyses[full_name] = {
  'df': df,
  'model': model,
  'feat_imps': model.feature_importances(),
  'metrics': metrics
}

# 5. Инициализация Dash-приложения
app = dash.Dash(__name__, external_stylesheets=[dbc.themes.
   LUX])

app.layout = dbc.Container(fluid=True, children=[
html.H1("Мульти-репозиторный анализ коммитов", className="
   text-center my-4"),
dcc.Dropdown(
id="repo-selector",
options=[{"label": r, "value": r} for r in repos],
value=repos[0] if repos else None,
clearable=False,
style={"width": "60%", "margin": "0 auto 20px auto"}
),
html.Div(id="tabs-container")
])

@app.callback(
Output("tabs-container", "children"),
```

```
Input("repo-selector", "value")
)
def update_tabs(selected_repo):
if not selected_repo or selected_repo not in analyses:
return html.Div("Репозиторий не выбран или недоступен")

entry = analyses[selected_repo]
df = entry['df'].copy()
feat_imps = entry['feat_imps']
metrics = entry.get('metrics', {})
metrics_table = dbc.Table([
html.Thead(html.Tr([html.Th("Метрика"), html.Th("Значение")
    ])),
html.Tbody([
html.Tr([html.Td("Precision"), html.Td(f"{metrics.get('
    precision', 0):.2f}")]),
html.Tr([html.Td("Recall"), html.Td(f"{metrics.get('recall',
     0):.2f}")]),
html.Tr([html.Td("F1-score"), html.Td(f"{metrics.get('
    f1_score', 0):.2f}")]),
html.Tr([html.Td("ROC-AUC"), html.Td(f"{metrics.get('auc',
    0):.2f}")]),
])
], bordered=True, striped=True, hover=True, style={"width":
    "40%", "marginTop": "20px"})
features = entry['model'].features

# Подстраховки
if 'author_name' not in df:
df['author_name'] = 'Unknown'
if 'has_bug_keyword' not in df:
df['has_bug_keyword'] = df['message'].str.contains(
r'\b(fix|bug|error)\b', case=False, regex=True, na=False
).astype(int)

# 6. Общая информация
tab_summary = dcc.Tab(label='Общая информация', children=[
dbc.Row([
dbc.Col(dcc.Graph(
figure=px.histogram(
df, x='lines_added', nbins=30,
title='Добавленные строки',
color_discrete_sequence=['#1f77b4']  # синяя
)
), md=6),
```

```
dbc.Col(dcc.Graph(
figure=px.histogram(
df, x='lines_deleted', nbins=30,
title='Удалённые строки',
color_discrete_sequence=['#d62728']  # красная
)
), md=6),
], className="g-4"),
dbc.Row([
dbc.Col(dcc.Graph(
figure=px.histogram(
df, x='files_changed', nbins=30,
title='Изменённые файлы',
color_discrete_sequence=['#2ca02c']  # зелёная
)
), md=6),
dbc.Col(dcc.Graph(
figure=px.histogram(
df, x='complexity_score', nbins=30,
title='Сложность изменений',
color_discrete_sequence=['#9467bd']  # фиолетовая
)
), md=6),
], className="g-4"),
])

# 7. Анализ риска
fi_vals = [feat_imps.get(f, 0) for f in features]
tab_risk = dcc.Tab(label='Анализ риска', children=[
metrics_table,
dbc.Row(dbc.Col(dcc.Graph(
figure=px.bar(
x=features, y=fi_vals,
title='Важность признаков',
color_discrete_sequence=px.colors.qualitative.Set2
)
)), className="g-4"),
dbc.Row([
dbc.Col(dcc.Graph(
figure=px.pie(
df, names='Risk', title='Рискованные vs обычные',
color_discrete_sequence=['#2ca02c', '#d62728']
)
), md=6),
dbc.Col(dcc.Graph(
```

```
figure=px.scatter(
df, x='lines_added', y='complexity_score',
color='Risk', title='Риск vs Сложность',
color_discrete_sequence=['#2ca02c', '#d62728']
)
), md=6),
], className="g-4"),
])

# 8. Авторы
author_activity = df['author_name'].value_counts().
   reset_index()
author_activity.columns = ['author_name', 'commit_count']
author_risk = df.groupby('author_name')['Risk_Proba'] \
.mean().reset_index() \
.sort_values('Risk_Proba', ascending=False)

tab_authors = dcc.Tab(label='Авторы', children=[
dbc.Row([
dbc.Col(dcc.Graph(
figure=px.bar(
author_activity, x='author_name', y='commit_count',
title='Активность авторов',
color_discrete_sequence=px.colors.sequential.Viridis
)
), md=6),
dbc.Col(dcc.Graph(
figure=px.bar(
author_risk, x='author_name', y='Risk_Proba',
title='Средний риск по авторам',
color_discrete_sequence=px.colors.sequential.Reds
)
), md=6),
], className="g-4"),
])

# 9. File-Risk Map
file_df = df.explode('file_list') if 'file_list' in df else
   pd.DataFrame()
if not file_df.empty:
fr = file_df.groupby('file_list').agg(
change_count=('file_list','size'),
avg_risk=('Risk_Proba','mean')
).reset_index()
else:
```

```
fr = pd.DataFrame(columns=['file_list','change_count','
    avg_risk'])
tab_file_risk = dcc.Tab(label='File-Risk Map', children=[
dbc.Row(dbc.Col(dcc.Graph(
figure=px.scatter(
fr, x='change_count', y='avg_risk',
hover_name='file_list',
title='Частота изменений vs средний риск',
color_discrete_sequence=['#17becf']  # бирюзовая
)
)), className="g-4")
])

# 10. Risk Timeline
df['commit_date'] = pd.to_datetime(df['author_datetime'],
    errors='coerce').dt.date
tl = df.sort_values('commit_date').groupby('commit_date').
    agg(
daily_risk=('Risk_Proba','mean'),
warnings=('Risk','sum')
).reset_index()
fig_tl = go.Figure([
go.Scatter(
x=tl['commit_date'], y=tl['daily_risk'],
mode='lines+markers', name='Средний риск',
line=dict(color='#1f77b4')
),
go.Bar(
x=tl['commit_date'], y=tl['warnings'],
name='Предупреждения', yaxis='y2', opacity=0.6,
marker_color='#ff7f0e'
)
])
fig_tl.update_layout(
title='Timeline риска и предупреждений',
yaxis=dict(title='Средний риск'),
yaxis2=dict(title='Кол-во предупреждений', overlaying='y',
    side='right')
)
tab_timeline = dcc.Tab(label='Risk Timeline', children=[
dbc.Row(dbc.Col(dcc.Graph(figure=fig_tl)), className="g-4")
])

# 11. Code Quality Tabs
quality_tabs = []
```

```
if {'pylint_warnings','pylint_errors','bandit_issues'} <=
   set(df.columns):
quality_tabs.append(dcc.Tab(label='Python Quality', children
   =[
dbc.Row([
dbc.Col(dcc.Graph(
figure=px.histogram(
df, x='pylint_warnings',
title='Pylint Warnings',
color_discrete_sequence=['#9467bd']
)
), md=6),
dbc.Col(dcc.Graph(
figure=px.scatter(
df, x='pylint_errors', y='bandit_issues',
title='Errors vs Security Issues',
color_discrete_sequence=['#8c564b']
)
), md=6),
], className="g-4")
]))
if {'eslint_warnings','eslint_errors'} <= set(df.columns):
quality_tabs.append(dcc.Tab(label='JS Quality', children=[
dbc.Row([
dbc.Col(dcc.Graph(
figure=px.histogram(
df, x='eslint_warnings',
title='ESLint Warnings',
color_discrete_sequence=['#e377c2']
)
), md=6),
dbc.Col(dcc.Graph(
figure=px.scatter(
df, x='eslint_errors', y='eslint_warnings',
title='Errors vs Warnings',
color_discrete_sequence=['#7f7f7f']
)
), md=6),
], className="g-4")
]))
if 'checkstyle_issues' in df.columns:
quality_tabs.append(dcc.Tab(label='Java Quality', children=[
dbc.Row(dbc.Col(dcc.Graph(
figure=px.histogram(
df, x='checkstyle_issues',
```

```
280 title='Checkstyle Issues',
    color_discrete_sequence=['#bcbd22']
    )
    )), className="g-4")
    ]))
285
    # 12. Commits Table
    df['Recommendations'] = df.apply(
    lambda row: generate_recommendations(row, row['Risk_Proba'],
        {}, feat_imps),
    axis=1
290 )
    df['Recommendations_Text'] = df['Recommendations'].apply(
       lambda recs: "; ".join(recs))
    tab_table = dcc.Tab(label='Commits Table', children=[
    dash_table.DataTable(
    columns=[
295 {"name":"SHA","id":"commit"},
    {"name":"Автор","id":"author_name"},
    {"name":"Дата","id":"commit_date"},
    {"name":"Риск","id":"Risk_Proba","type":"numeric","format":{
       "specifier":".2f"}},
    {"name":"Сообщение","id":"message"},
300 {"name":"Рекомендации","id":"Recommendations_Text"},
    ],
    data=df[['commit','author_name','commit_date','Risk_Proba','
       message','Recommendations_Text']]
    .to_dict('records'),
    page_size=10,
305 style_cell={'textAlign':'left','whiteSpace':'normal','height
       ':'auto'},
    style_header={'fontWeight':'bold'}
    )
    ])

310 # 13. Календарь активности
    all_dates = pd.date_range(df['commit_date'].min(), df['
       commit_date'].max(), freq='D')
    heat = pd.DataFrame({'date': all_dates})
    heat['count'] = heat['date'].map(df['commit_date'].
       value_counts()).fillna(0)
    heat['dow'] = heat['date'].dt.weekday
315 heat['week'] = ((heat['date'] - heat['date'].min()).dt.days
       // 7).astype(int)
    max_w = heat['week'].max() + 1
```

```
mat = np.zeros((7, max_w))
for _, r in heat.iterrows():
mat[int(r['dow']), int(r['week'])] = r['count']
cal_fig = go.Figure(data=go.Heatmap(
z=mat,
x=[f'Неделя {i+1}' for i in range(max_w)],
y=['Пн','Вт','Ср','Чт','Пт','Сб','Вс'],
colorscale='Greens', hoverongaps=False,
colorbar=dict(title='Коммитов/день')
))
cal_fig.update_layout(xaxis=dict(scaleanchor='y', showgrid=
   False),
yaxis=dict(showgrid=False))
tab_calendar = dcc.Tab(label='Календарь активности',
   children=[
dbc.Row(dbc.Col(dcc.Graph(figure=cal_fig)), className="g-4")
])

tabs = [
tab_summary,
tab_risk,
tab_authors,
tab_file_risk,
tab_timeline,
*quality_tabs,
tab_table,
tab_calendar
]
return dcc.Tabs(tabs)

if __name__ == '__main__':
app.run(debug=True)
```

Листинг 4.4

recommendationds.py

```
# recommendations.py
from typing import List

__all__ = ['generate_recommendations']

def generate_recommendations(commit: dict,
risk_proba: float,
repo_stats: dict,
feature_importances: dict) -> List[str]:
```

```
recommendations: List[str] = []

if risk_proba > 0.8:
recommendations.append(
"Очень высокий риск: обязательно провести углублённое код-ре
   вью и расширенное тестирование."
)
elif risk_proba > 0.5:
recommendations.append(
"Повышенный риск: обратите внимание на качество изменений и
   добавьте тесты."
)

msg_len = commit.get('message_length', 0)
if msg_len < 20:
recommendations.append(
"Сообщение слишком короткое: дайте подробное описание измене
   ний."
)
elif msg_len > 200:
recommendations.append(
"Очень длинное сообщение: разделите описание на ключевые пун
   кты или используйте более лаконичные формулировки."
)

if commit.get('has_bug_keyword', 0):
recommendations.append(
"Выявлен багфикс: убедитесь в наличии регрессионных тестов и
    обновлении документации."
)

lines_added = commit.get('lines_added', 0)
lines_deleted = commit.get('lines_deleted', 0)
total = lines_added + lines_deleted
stats_total = repo_stats.get('total_changes', {})
mean_total = stats_total.get('mean')
std_total = stats_total.get('std')
if mean_total and std_total and total > mean_total + 2 *
   std_total:
recommendations.append(
f"Объём изменений ({total}) значительно превышает среднее ({
   mean_total:.1f}). "
"Разбейте коммит на более мелкие логические части."
)
```

```
50  files_changed = commit.get('files_changed', 0)
    stats_files = repo_stats.get('files_changed', {})
    q95_files = stats_files.get('quantile_95')
    if q95_files and files_changed > q95_files:
    recommendations.append(
55  f"Затронуто слишком много файлов ({files_changed} > 95% кван
       тиль). Проверьте целостность изменений."
    )

    complexity = commit.get('complexity_score', 0)
    stats_complex = repo_stats.get('complexity_score', {})
60  q90_complex = stats_complex.get('quantile_90')
    if q90_complex and complexity > q90_complex:
    recommendations.append(
    f"Высокая сложность ({complexity} > 90% квантиль). "
    "Рассмотрите рефакторинг и дополнительное покрытие тестами."
65  )

    avg_hist = commit.get('avg_file_history', 0)
    stats_hist = repo_stats.get('avg_file_history', {})
    mean_hist = stats_hist.get('mean')
70  std_hist = stats_hist.get('std')
    if mean_hist and std_hist and avg_hist > mean_hist + 2 *
       std_hist:
    recommendations.append(
    "Возможно, стоит разделить функциональность."
    )
75
    interval = commit.get('minutes_since_previous_commit')
    stats_interval = repo_stats.get('commit_interval', {})
    median_int = stats_interval.get('median')
    if interval is not None and median_int:
80  if interval < 5:
    recommendations.append(
    "Очень быстрый коммит (<5 минут): убедитесь, что изменения з
       авершены и протестированы."
    )
    if interval > 2 * median_int:
85  recommendations.append(
    f"Промежуток {interval:.0f} мин более чем в 2 раза дольше ме
       дианы "
    f"({median_int:.0f} мин): проверьте актуальность ветки перед
        слиянием."
    )
```

```
90  author = commit.get('author_name', 'Unknown')
    author_stats = repo_stats.get('author_stats', {}).get(author
       , {})
    median_lines_author = author_stats.get('median_lines_added')
    if median_lines_author and lines_added > 2 *
       median_lines_author:
    recommendations.append(
95  f"Автор {author} внёс {lines_added} строк, что более чем в 2
        раза превышает "
    f"его медианные {median_lines_author} строк: дополнительная
       проверка кода."
    )

    if not recommendations:
100 recommendations.append(
    "Явных аномалий не обнаружено. Рекомендуется стандартное код
       -ревью и покрытие тестами."
    )

    return recommendations
```

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., Педрич В., Суччи Д. A meta-analytical comparison of Naive Bayes and Random Forest for software defect prediction // URL: https://link.springer.com/chapter/10.1007/978-3-031-35501-1_14#citeas

[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