Московский физико-технический институт
(национальный исследовательский университет)

На правах рукописи

Качанов Владимир Владимирович

Методы и программные средства автоматического рецензирования исходного кода

2.3.5 — Математическое и программное обеспечение вычислительных систем, комплексов и компьютерных сетей

Диссертация на соискание ученой степени
кандидата технических наук

Научный руководитель:
д.ф.-м.н. проф.
Цурков Владимир Иванович

Москва — 2025

Оглавление

Введение

Актуальность темы. В условиях современной цифровой трансформации и динамичного развития информационных технологий обеспечение высокого качества программных продуктов становится стратегически важной задачей. Кодовая база, которая является основой функционирования программного обеспечения, требует постоянного мониторинга и совершенствования для минимизации риска возникновения критических ошибок. В этом контексте методы и программные средства автоматического рецензирования исходного кода, основанные на расширяемом наборе детекторов нежелательного или дефектного кода, представляют собой перспективное направление.

Рецензирование кода традиционно рассматривается как неотъемлемая часть жизненного цикла разработки программного обеспечения наряду с тестированием, изменением кода и его рефакторингом. Этап анализа кода, проводимый после его непосредственного написания, позволяет выявить скрытые дефекты и неформальные ошибки, которые могут быть не обнаружены в рамках стандартных систем непрерывной интеграции и автоматизированного тестирования. Такой подход обеспечивает более глубокую и всестороннюю оценку кода, способствуя повышению его качества и уменьшению вероятности возникновения ошибок в будущем. Кроме того, повышается читаемость и структурированность кода: он становится понятен не только одному специалисту, но и всей команде, что значительно упрощает и ускоряет дальнейшую разработку и поддержку проекта. Процесс рецензирования кода оказывает многогранное положительное влияние на качество конечного продукта и продуктивность команды разработчиков.

Однако проведение рецензирования кода вручную сопряжено с рядом существенных проблем. Ручной анализ требует значительных временных затрат и зачастую зависит от субъективной оценки отдельных участников команды, что может приводить к неравномерности качества рецензий. Кроме того, человеческий фактор нередко является источником пропуска небольших, но важных дефектов, особенно в условиях работы с большими объемами кода. В этом контексте внедрение автоматических средств рецензирования способно существенно повысить эффективность данного процесса, обеспечивая систематический и непрерывный контроль качества. Автоматизация помогает стандартизировать

процедуру проверки, снизить вероятность возникновения ошибок, а также разгрузить специалистов, позволяя им сосредоточиться на более сложных и креативных аспектах разработки программного обеспечения.

Особое внимание в современных исследованиях уделяется разработке инструментов для детектирования признаков технического долга и антипаттернов исходного кода («запахи кода», англ. code smells). Это понятие, введенное Кентом Бэком и популяризированное Мартином Фаулером, описывает характеристики кода, которые затрудняют его поддержку и развитие. К таким признакам относятся длинные методы, сложные условия, классы с избыточными обязанностями или чрезмерная связанность между модулями. Автоматизированные инструменты, способные выявлять существующие антипаттерны и прогнозировать потенциальные проблемные места, значительно повышают поддерживаемость системы и улучшают коммуникацию внутри команды.

Уже более десяти лет ведутся активные исследования и разработки в области автоматизированных систем рецензирования исходного кода, в которых приняли участие как академические исследователи, так и промышленные компании. Среди наиболее заметных инициатив следует отметить такие инструменты, как SonarQube, Sourcery и Semgrep, интегрируемые в системы непрерывной интеграции и разработки (CI/CD) и обеспечивающие автоматическое комментирование запросов на включение изменений (pull requests) с предоставлением рекомендаций. В последние годы акцент сместился в сторону облачных решений, таких как GitHub CodeQL, Snyk Code и AWS CodeGuru Reviewer, которые применяют методы анализа данных и машинного обучения для повышения точности анализа и сокращения доли ложных срабатываний. Однако существующие инструменты в значительной степени ориентированы на статический анализ или эвристические правила. Пилотные внедрения систем на основе больших языковых моделей в настоящее время осуществляются в ряде крупных технологических компаний, включая Google, ByteDance и Microsoft.

Таким образом, актуальной является задача разработки методов анализа исходного кода и детектирования не только стандартных ошибок и уязвимостей, которые обнаруживаются методами статического и динамического анализа, но и менее формализованных дефектов, а также инструмента, который интегрирует в себе автоматическое рецензирование исходного кода с пояснением и фильтрацией обнаруженных антипаттернов. При этом требуется обеспечить

достаточную точность и полноту для практического применения. Разработка таких методов и итогового инструмента не только повысит качество ПО, но и облегчит дальнейшую его поддержку. Также исследования в этом направлении способствуют развитию теоретической базы, на которой будут строиться дальнейшие инновационные решения в области обеспечения качества программного обеспечения.

Целью данной работы является развитие методов обнаружения признаков технического долга, заданных на основе примеров исходного кода, и построение системы автоматического рецензирования кода, которая интегрирует в себя реализованные методы. Для достижения поставленной цели были сформулированы и решены следующие **задачи**:

1. Разработать и реализовать детекторы признаков технического долга в исходном коде. Собрать наборы размеченных примеров антипаттернов кода по исправлениям непосредственных разработчиков проектов.
2. Проанализировать существующие подходы к автоматическому рецензированию исходного кода. Провести обзор тематик рецензий различными авторами. Разработать методы определения полезных и применимых замечаний.
3. Разработать и реализовать инструмент автоматического рецензирования на основе наиболее перспективных подходов.
4. Интегрировать разработанные и реализованные детекторы в единую систему рецензирования.

Методы исследования. Для решения задач, поставленных в диссертационной работе, использовались методы машинного обучения, глубокого обучения и программной инженерии.

Основные положения, выносимые на защиту, и научная новизна.

1. Метод сбора размеченных примеров и набор данных различных типов дефектов кода, извлеченный из истории разработки проектов;
2. Методы детектирования антипаттернов исходного кода, основанный на анализе исходного кода и методах машинного обучения;
3. Методы автоматического рецензирования исходного кода, основанные на применении языковых моделей;

4. Программный комплекс, интегрирующий работу детекторов антипаттернов исходного кода в систему автоматического рецензирования.

Научная новизна.

1. Сформирован репрезентативный набор данных, охватывающий различные типы ошибок, выявленные в процессе эволюции реальных программных проектов, что создаёт основу для обучения и валидации интеллектуальных моделей анализа кода;
2. Разработаны методы интеллектуального детектирования неформальных дефектов программного кода, основанные на синтезе статического анализа исходного текста и алгоритмов машинного обучения;
3. Проведено сравнительное исследование эффективности методов повышения качества генерации больших языковых моделей в задаче рецензирования исходного кода;
4. Выполнено оригинальное исследование, результатом которого стало проектирование и реализация программного комплекса, интегрирующего разработанные детекторы антипаттернов в единую систему автоматического рецензирования кода. Архитектура комплекса обеспечивает модульность, расширяемость и совместимость с существующими инструментами разработки.

Теоретическая значимость. Исследование вносит вклад в развитие научной базы автоматизированного анализа исходного кода и рецензирования изменений, вносимых разработчиками. В работе предложены новые методы интеллектуального анализа и сформирован репрезентативный набор данных, что расширяет спектр задач, решаемых традиционным статическим и динамическим анализом и дает основу для обучения и тестирования моделей, ориентированных на практику индустриальной разработки. Проведённое исследование помогает формализовать подходы к фильтрации, объяснению и структурированию замечаний в задаче рецензирования кода, что открывает перспективы для последующих работ в области использования ИИ для обеспечения качества ПО. Таким образом, работа способствует развитию теоретических представлений о сочетании статического анализа и методов машинного обучения.

Практическая значимость. Разработанный программный комплекс позволяет повысить эффективность процесса рецензирования кода, уменьшить зависимость от субъективных факторов и снизить временные затраты на ручной анализ. Автоматизация проверки способствует выявлению как формальных, так и неформальных дефектов, улучшает читаемость и структурированность кода, облегчает его поддержку и ускоряет интеграцию новых участников команды. Применение системы в промышленной среде даёт возможность стандартизировать практики контроля качества и интегрировать их в существующие системы непрерывной разработки и внедрения. Тестирование показало, что более половины предлагаемых исправлений и замечаний к изменениям в исходном коде отмечаются разработчиками как приемлемые или применимые. Имеется акт о внедрении результатов исследования.

Достоверность полученных результатов обеспечивается:

- корректностью выбора и применения используемых алгоритмов, методов и моделей для сбора, анализа и обработки данных;
- результатами апробации реализованных методов на внутреннем тестировании программного комплекса;
- согласованностью полученных результатов с результатами, полученными другими авторами.

Апробация работы. Результаты работы докладывалисьна следующих конференциях:

1. “Технический долг в жизненном цикле разработки ПО: запахи кода”, Качанов В.В., Ермаков М., Панкратенко Г., Спиридонов А., Волков А.С., Марков С.И. ISPRAS Open 2021, секция «Технологии анализа, моделирования и трансформации программ», Москва.
2. “Извлечение именованных сущностей из рецензий к исходному коду”, Качанов В.В, Хитрова А.С., Марков С.И., ISPRAS Open 2023, секция «Управление данными и информационные системы», Москва.
3. “Создание набора данных для комбинированной классификации рецензий исходного кода”, Петрова П.А., Марков С.И., Качанов В.В., ИОИ-2024

Публикации по теме диссертации. Основные результаты по теме диссертации изложены в 6 печатных изданиях, 5 из которых изданы в журналах,

рекомендованных ВАК, 1 — в тезисах докладов. Получено 1 свидетельство о регистрации программы для ЭВМ.

Личный вклад в работах с соавтоарми заключается в следующем: [1] — все приведенные результаты получены самостоятельно. [2, 3, 4, 5] — разработка и модификация моделей машинного обучения, сбор и разметка результатов, проведение вычислительных экспериментов и анализ результатов. [6] — разработка и оптимизация алгоритмов трансформации исходного кода, анализ результатов.

Структура и объем работы. Диссертация состоит из оглавления, введения, четырех глав, заключения, списка иллюстраций, списка таблиц и списка литературы из 129 наименований. Основной текст занимает 134 страницы.

Краткое содержание работы по главам.

В первой главе приводится обзор работа, которые имеют отношение к теме диссертации. Проведена систематизация основных понятий и терминологии в области технического долга и методов его идентификации. Полученные результаты подтверждают актуальность разработки методов детектирования антипаттернов с повышенной точностью и обосновывают необходимость создания гибридных подходов. Осуществлено комплексное исследование современного состояния автоматического рецензирования исходного кода и детектирования антипаттернов, в результате которого были выделены два принципиально различных подхода.

Во второй главе приводится описание разработки методов детектирования антипаттернов кода с использованием машинного обучения. Создан алгоритм автоматического сбора данных из истории разработки проектов с открытым исходным кодом. Получен набора из 5912 примеров антипаттернов и их исправлений от разработчиков. Обученные модели показали улучшение F1-меры на 46-78% для разных типов антипаттернов.

В третьей главе описывается исследование и разработка методов автоматического рецензирования кода. Проанализированы подходы поиска исторических рецензий и генерации комментариев языковыми моделями. Выявлены ограничения существующих методов в виде низкого качества данных и несоответствия используемых метрик оценки качества ручной разметке. Предложены модификации алгоритмов по поиску похожего кода. Решена задача классифика-

ции рецензий к исходному коду, создан размеченный набор данных. Разработан метод рецензирования исходного кода на основе больших языковых моделей с методами улучшения генерации. Показана возможность интеграции разработанных методов в единую систему рецензирования.

В четвертой главе излагается оценка эффективности систем автоматического рецензирования кода на основе больших языковых моделей (LLM). Результаты показывают двукратное превосходство LLM над традиционными методами по доле приемлемых комментариев. Выявлена зависимость качества от объема и тематической релевантности данных. Оптимизация инструкций и фильтрация повысили долю применимых рецензий до 38.8%. Практическая валидация подтвердила эффективность подхода (MRR=0.915).

В заключении приводится краткое описание результатов диссертационного исследования.

Глава 1
Обзор существующих методов автоматического рецензирвоания и детектирования антипаттернов исходного кода

1.1. Особенности терминологии

Рецензирование исходного кода (англ. source code review) является неотъемлемой частью жизненного цикла разработки программного обеспечения[7]. Под рецензией подразумевается короткое сообщение, содержащее рекомендации по исправлению исходного кода автора. Рецензирование выполняется не тем человеком, который разрабатывал изменения, а другим членом команды. Результатом является обратная связь по проведенным изменениям: указания на недостатки программы, которые надо изменить перед сохранением, либо согласование внесенных изменений и внесение их в код программы.

Такой процесс привносит множество полезных характеристик, таких как улучшение качества исходного кода, обмен знаниями, улучшение взаимодействия в команде повышение безопасности разрабатываемого программного обеспечения, улучшение качества тестирования, способствует снижению технического долга.

Исследования показывают, что рецензирование кода значительно снижает количество дефектов в программном обеспечении (ПО). Хотя в исследовании [8] утверждается, что в отличие от формальных проверок кода, которые позволяют обнаружить 60-65% скрытых дефектов, неформальные проверки, такие как рецензирование, обнаруживают менее 50% ошибок, авторы книги [9] утверждают, что рецензирование помогает найти столько же ошибок, как и формальные проверки исходного кода.

В области обмена знаниями польза рецензирования очевидна, так как разработчики знакомятся с участками исходного кода, который они не писали, но вынуждены его понять, чтобы написать рецензию. Таким образом повышается так называемый bus factor — некий неформальный признак, обозначающий сколько разработчиков нужно забрать из проекта, чтобы дальнейшая разработка была невозможна по причине недостатка знаний у оставшихся участников над каким-либо модулем проекта [10]. Также рецензирование кода является способом обучения новых коллег путем, непосредственно, указания на их ошибки и неточности в их коммитах, либо, наоборот — новички, когда сами рецензи-

руют чужой код узнают для себя что-то новое. Кроме того рецензирование на ряду с линтерами способствуют стандартизации оформления исходного кода, что повышает читаемость и поддерживаемость.

В работе [11] исследован вопрос обнаружения ошибок в исходном коде связанных с безопасностью. Так на примере проекта с открытым исходным кодом Chromium OS авторы статьи нашли 516 примеров рецензий, которые указывали на security CVE defect. Также в данном исследовании показано, что при совместном рецензировании и общении между двумя разработчиакми повышается верятность найти уязвимость.

Также рецензирование кода позволяет на более ранних стадиях обнаруживать потенциальные будущие проблемы тем самым заранее снижая технический долг, который накапливается в ходе разработки программного продукта.

Рецензирование исходного кода — это процесс систематической проверки и оценки исходного кода программ одним или несколькими сотрудниками (рецензентами), не являющимися его автором, с целью выявления дефектов, уязвимостей, нарушения стандартов, повышения читаемости, сопровождающаяся обратной связью до слияния изменений в основную ветку кода.

Автоматическое рецензирование исходного кода — это процесс анализа исходного кода с помощью специализированных инструментов (статический анализ, линтеры и пр.), которые автоматически проверяют соответствие кода заранее заданным правилам качества, стиля и безопасности, с быстродействующим выявлением ошибок и нарушений без участия человека.

Антипаттерны кода (англ. code smells) — поведенческие или структурные характеристики в исходном коде, которые не являются напрямую ошибками, но указывают на потенциальные проблемы в архитектуре или дизайне системы, снижают сопровождаемость и облегчают накопление технического долга; термин введён Кентом Беком и популяризирован Мартином Фаулером.

Технический долг (англ. technical debt) — концепция, обозначающая будущие затраты во времени и ресурсах, возникающие вследствие выбора быстрых, необдуманных решений вместо более качественных и устойчивых, но дорогостоящих подходов. Можно провести аналогию классическому долгу — это деньги, которые берутся на разработку новой функциональности (расширение своего производства), а процесс рефакторинга (приведение вашего исходного кода ПО в должный вид, который не приносит для пользователей ничего нового, но за-

нимает время) — выплаты по этому долгу.

1.2. Анализ существующих методов автоматизированного рецензирования и генерации комментариев

Рассмотрим существующие подходы к решению задачи автоматического рецензирования. Существуют два основных направления методов: подбор ранее написанных рецензий для новых контекстов на основе интеллектуального анализа и сравнения, и генерация комментариев с помощью языковых моделей различной степени сложности.

Одним из ярких представителей первой группы является CommentFinder [12]. В основе его работы [12] лежит предположение, что похожие изменения кода часто получают похожие комментарии. В отличие от методов, основанных на глубоком обучении (DL), CommentFinder фокусируется на простоте и эффективности, устраняя необходимость в тяжеловесном обучении модели. Авторы работы утверждают о конкурентоспособных результатов. Подход работает в три основных этапа: векторизация, поиск и рекомендация.

На первом шаге CommentFinder преобразует код в векторные представления с помощью алгоритма Мешок Слов (англ. Bag-of-Words). Каждый измененный метод в наборе данных токенизируется в последовательность токенов кода, которые затем преобразуются в частотные векторы. Часто встречающиеся токены, ключевые слова, отфильтровываются для уменьшения шума и размерности, обеспечивая более целенаправленное сравнение. Такая токенизация сохраняет синтаксис и семантику кода.

Далее система извлекает похожие изменения кода, вычисляя косинусное расстояние между векторным представлением данного метода и массивом тех, что находятся в обучающем наборе. Эта метрика расстояния измеряет лексическое сходство, позволяя CommentFinder относительн быстро идентифицировать N наиболее близких совпадений из большого корпуса пар код-комментарий. Затем эти N кандидатов дополнительно уточняются с помощью Gestalt Pattern Matching (GPM) [13], метода текстового сходства, который учитывает порядок токенов кода для определения более точных совпадений.

Наконец, CommentFinder ранжирует кандидатов на основе оценок GPM и извлекает связанные комментарии обзора в качестве рекомендаций. Объединяя подход основанный на лексическом сравнении с сопоставлением частотных

векторов, CommentFinder балансирует точность и вычислительную эффективность. Такой подход избегает ресурсоемких требований моделей глубокого обучения, предоставляя релевантные предложения по улучшению исходного кода, что делает его более доступным для конфигураций с ограниченными ресурсами.

Работ применяющих в той или иной степени языковые модели значительно больше, так как задача автоматического рецензирвоания является по сути задачей генерации текстов. В качестве примеров работ по генерации комментариев по изменениям кода на основе языковых моделей рассматривались программные комплексы AUGER[14] и CodeReviewer [15].

AUGER — это модель, разработанная для автоматизации генерации комментариев к обзору кода с использованием предварительно обученных моделей T5. Система направлена на устранение неэффективности и ограничений, которые присущи написанным вручную рецензиям кода, путем синтеза человекоподобных комментариев к обзору, которые являются как точными, так и релевантными. Процесс начинается с подготовки данных, где изменения кода и их соответствующие комментарии к обзорам собираются из открытых репозиториев. Эти данные проходят тщательную очистку, включая удаление дубликатов, неактуальных обзоров и шумного содержания, такого как слишком короткие или чрезмерно длинные комментарии. Дополнительные предварительные обработки, такие как токенизация и семантическое увеличение, применяются для улучшения качества набора данных.

Далее вводится специальная разметка строк коммита для идентификации и выделения конкретных участков, которые были изменены в процессе обзора. Этот шаг обеспечивает более тонкий акцент на критических фрагментах кода путем вставки специальных тегов в строки, на которые нацелены комментарии. Тегированные блоки кода затем поступают на стадию кросс-предварительного обучения, где модель T5, предварительно обученная на коде Java, дополнительно обучается для изучения контекстных связей между программным кодом и комментариями к обзорам на естественном языке. Для обучения моедли на би-модальных данных используется метод маскированного языкового моделирования (MLM), позволяя модели эффективно обрабатывать структуры кода и семантику рецензий.

Затем стадия генерации комментариев дообучает предварительно обученную модель для конкретной задачи создания комментариев к коммитам. Модель оптимизируется для отображения тегированного участка исходного кода в полезные и точные комментарии. AUGER объединяет преимущества переноса обучения (англ. transfer learning) и дообученого (англ. fine-tuned) тегирования, позволяя ему эффективно генерировать комментарии и адаптироваться к изменяющимся сценариям кода. Этот подход существенно улучшает как эффективность, так и масштабируемость процесса кодовых обзоров[14].

CodeReviewer. Подход CodeReviewer [15] представляет собой предварительно обученную модель на основе трансформера, разработанную для автоматизации различных аспектов процесса проверки кода путем понимания изменений кода и создания содержательных комментариев к обзору. В отличие от традиционных моделей глубокого обучения, обучаемых исключительно на исходном коде, CodeReviewer специально предварительно обучен на различиях кода (англ. diff), что позволяет ему фиксировать структурные и контекстные изменения между различными версиями кода. Для достижения этого модель использует архитектуру кодировщика-декодера, похожую на Text-to-Text Transfer Transformer (T5) [16], включающую четыре тщательно разработанные задачи предварительного обучения: прогнозирование тегов различий (англ. diff tag prediction), шумоподавление различий кода (англ. code diffs denoising), шумоподавление комментариев обзора (англ. denoising review comments) и генерация комментариев обзора (англ. review comment generation) [16].

Прогнозирование тегов различий позволяет CodeReviewer понимать структуру различий кода, обучая его распознавать измененные, добавленные и удаленные строки в изменении кода. Этот шаг гарантирует, что модель эффективно различает старый и новый код, что является важным аспектом проверки изменений. Задача шумоподавление различий кода (code diffs denoising) помогает модели изучать закономерности изменений кода путем реконструкции замаскированных или поврежденных строк, улучшая ее способность распознавать распространенные ошибки кодирования и логические несоответствия. Аналогичным образом, задача шумоподавления комментариев (review comment denoising) повышает способность модели понимать написанные человеком комментарии путем реконструкции частично замаскированной обратной связи. На-

конец, задача генерации комментариев обзора (review comment generation) обучает CodeReviewer генерировать рецензии на основе изменений кода, сокращая разрыв между языками программирования и естественным языком [15].

Благодаря предварительному обучению на крупномасштабном многоязычном наборе данных pull requests GitHub, охватывающем девять языков программирования, CodeReviewer изучает широкое понимание практик рецензирования кода. При тонкой настройке для определенных задач, таких как генерация рецензий или уточнение кода, по утверждениям авторов CodeReviewer значительно превосходит существующие модели глубокого обучения, демонстрируя свою эффективность в автоматизации действий по рецензированию кода.

Большие языковые модели. Более поздние работы используют действительно большие языковые предобученные модели с числом параметров более 7 миллиардов. Такие модели способны без дополнительного дообучения генерировать достаточно связный и осмысленный текст, качество которого в большой степени зависит от качества написанной инструкции и предоставленного контекста.

Авторы работы [17] описывают процесс генерации комментариев при помощи LLM, а также применение подхода модели-судьи (LLM-as-a-Judge) для автоматической оценки качества рецензий и сравнения нескольких. Авторы использовали подходы, называемые «few-shot» и «reasoning» [18]. Также в открытый доступ предоставлен исходный код и набор данных, на котором проводилась оценка качества. В публикации было введено 10 параметров качества рецензий по которым происходит как оценка, так и написание: Readability, Relevance, Explanation Clarity, Problem Identification, Actionability, Completeness, Specificity, Contextual Adequacy, Brevity. Основной единицей с которой работает разработанный подход являются методы и функции.

В работе [19] описывается промышленная система для автоматизированного code review в компании ByteDance — BitsAI-CR. Выделяются 4 базовые категорий рецензий, на которые можно разбить основные полезные комментарии в рецензиях. В качестве контекста для модели используют измененный метод в унифицированном формате. Для оценки эффективности в первую очередь используют метрику точности предложенных комментариев, так как для практического применения системы важно избежать предоставления пользователю

большого количества низкокачественных комментариев, даже если это происходит в ущерб полноте. В качестве примера открытой модели в исследовании используется большая языковая модель Qwen/Qwen2.5-Coder-32B-Instruct [20].

1.3. Основные понятия в области детектирования признаков технического долга

Данная секция посвящена описанию проблемы запахов кода (англ. code smells) и предлагаемых алгоритмов и методов их обнаружения.

Одна из основных задач разработчика программного обеспечения (ПО) заключается в развитии продукта. Процесс постоянного внесения новой функциональности не может продолжаться вечно, так как из-за нарастающей сложности программного продукта и сжатых сроков качество написанного исходного кода (текста программы) ухудшается. Иногда настолько, что дальнейшее расширение функциональности становится невозможным. В литературе принято такое явление называть «техническим долгом» [21]. Можно провести параллель с обычным. Технический долг — это деньги, которые вы берете на разработку новой функциональности (расширение своего производства), а процесс рефакторинга (приведение вашего исходного кода ПО в должный вид, который не приносит для пользователей ничего нового, но занимает какое-то время) — выплаты по этому долгу. Соответственно чем чаще вы гонитесь за быстрым развитием функциональности, не погашая долг, тем больше проценты по нему придется платить, а значит, более долгий и значительный рефакторинг придется совершать. Стоит также заметить, что технический долг не всегда оказывается чем-то безусловно плохим, иногда приоритет быстрого решения проблемы над надежным необходим для продвижения проекта. Одни из ярких представителей симптомов технического долга называются «запахи кода»[21, 3] — антипаттерны программирования.

Термин запахи кода был предложен Кентом Бэком (Kent Beck) и популяризован Мартином Фаулером (Martin Fowler) [21] в контексте проведения рефакторинга кода — модификации с целью устранения архитектурных недостатков. Каждая разновидность антипаттернов кода описывает определенный набор особенностей, которые в той или иной степени затрудняют работу с кодом. Это индикаторы технической задолженности, которые усложняют сопровождение и развитие программных систем. Стоит понимать, что такие антипаттерны —

это вопрос поддерживаемости. Это не те дефекты, которые приведут к сбою кода на производстве или откроют дверь для злоумышленников. Антипаттерны кода — это о вещах, которые могут заставить вас или ваших коллег колебаться или споткнуться в следующий раз, когда код будет меняться или улучшаться. Примеры таких антипаттернов кода включают длинные методы, сложные условия, классы с несколькими обязанностями или избыточную связанность между модулями.

Антипаттерны исходного кода делятся на три основных категории:

- Implementation smells: локализованы в пределах одного метода (например, сложные методы, длинные списки параметров).
- Design smells: затрагивают классы или их взаимодействия (например, «божественный класс» или многозадачная абстракция).
- Architecture smells: имеют влияние на уровень архитектуры системы, такие как концентрированная функциональность или рассеянная логика.

Также на примере выделяемых антипаттернов можно рассмотреть уровни абстракции, которые они затрагивают:

- «Божественный класс» (God Class) — компонент, который реализует большое количество разноплановой функциональности. Исправление кода для устранения такого дефекта предполагает разбиение функциональности по принципу «разделяй и властвуй».
- «Завистливый метод» (Feature Envy) — метод, который обращается к полям других классов больше, чем к полям собственного класса. Такие ситуации могут свидетельствовать о том, что инкапсуляция данных применяется некорректно, что затрудняет написание тестов и требует создания лишних объектов. Исправление кода в данной ситуации предполагает перенос функции в более подходящий класс.
- «Стрельба дробью» (Shotgun Surgery) описывает ситуацию, в которой изменение класса приводит к необходимости делать незначительные изменения в большом количестве других классов. Причинами подобной архитектуры может быть выделение слишком узкоспециализированных сущностей, с которыми производятся одни и те же действия. Для исправления подобной ситуации стоит перенести в один класс все совместно изменяемые методы и поля

- «Сложный метод» (Complex Method) — метод с излишне сложной реализацией (с точки зрения графов потока управления и данных). Понимание и изменение подобных методов потребует больших ресурсов, а реализация тестов может не покрывать все граничные случаи или иметь свою сложную структуру, которая потребует собственного тестирования. Исправление подобного метода предполагает вынесение некоторой части функциональности в отдельный метод.
- «Длинный метод» (Long Method) — метод, реализация которого включает слишком большой объем кода (в пересчёте на строки, инструкции и т.д.). Исправление аналогично сложному методу.
- «Длинный список параметров» (Long Parameter List) описывает проблему методов и функций, имеющих слишком много формальных параметров. Подобная ситуация может возникать, например, если в метод передают много полей некоторого класса, тогда лучшим решением было бы передать объект класса целиком и получать нужные поля уже внутри метода.

Исходя из описания и примеров можно сделать вывод о неформальности определения а также о достаточно размытых и субъективных границах какую сущность можно назвать «дефектной», а какую — нет. Антипаттерны зависят от контекста проекта: одни и те же паттерны в коде могут быть дефектными в одном случае и нормальными конструкциями в другом. Например, метод с высокой цикломатической сложностью может быть приемлемым, если его структура проста и хорошо читаема. Также они могут проявляться по-разному в зависимости от используемого языка программирования.

Итого, антипаттерны исходного кода представляют собой «сигналы» кода о возможных проблемах качества, которые требуют устранения для улучшения надежности, читаемости и поддерживаемости системы.

1.4. Обзор методов и алгоритмов детектирования антипаттернов

Автоматическое детектирование антипаттернов кода представляет собой сложную задачу, поскольку требует анализа исходного кода для выявления структурных и семантических проблем.

Кроме описанной выше чуствительности к контексту проекта и зависимости от используемого языка программирования есть и другие сложности в автоматическом дектектировании. Как видно, описания антипаттернов кода опира-

ются на крайне общие понятия структуры и семантики исходного кода. Обнаружение требует ручного анализа, проводимого экспертом, или использования косвенных характеристик, которые так или иначе отражают «суть проблемы». В качестве примера рассмотрим божественный класс и попытаемся предположить, по каким характеристикам можно было бы его идентифицировать. По неформальному определению божественный класс реализует большой объем независимой функциональности. С практической точки зрения это как минимум должно означать, что класс определяет большое количество публичных методов. Классы, количество методов которых ниже заданного порога, маловероятно можно будет назвать божественными. Независимость реализуемой функциональности можно определить следующим образом: если класс определяет некоторое количество внутренних полей, а два его метода осуществляют доступ к непересекающимся подмножествам этих полей, то эти методы никак не связаны. Чем меньше степень связности методов класса при большом их количестве, тем более несогласованным является класс. Процесс подсчёта подобных характеристик может быть довольно сложным на практике. Так, например, «геттеры» и «сеттеры» — методы доступа к переменным — вряд ли имеет смысл учитывать при подсчёте связности, потому что обычно они не должны оперировать больше, чем одним полем класса.

В отличие от неформальных определений, подобные характеристики поддаются численному описанию. В контексте области анализа кода подобные числовые характеристики называются термином метрики [22]. Подсчёт метрик может быть легко автоматизирован на основе инструмента, который осуществляет разбор исходного кода. Так, количество методов класса (NMD — Number of Methods Declared) и степень несогласованности методов (LCOM — Lack of Cohesion of Methods) — метрики, которые часто входят в набор поддерживаемых инструментами анализа кода.

Подход основанный на метриках является базовым в различных иструментах анализа кода связанных с признаками технического долга (PMD, SonarQube, JDeodorant, SpotBugs). Однако такие заранее определенные правила часто не покрывают весь спектр антипаттернов кода. Также созданные эвристики могут быть слишком жесткими или наоборот слишком общими, что приводит к ложноположительным или ложноотрицательным результатам.

1.4.1. Методы основанные на метриках кода

В данной секции будет описано исследование набора свободно распространяемых инструментов автоматического поиска антипаттернов кода. Для предварительной оценки были рассмотрены следующие: DesigniteJava, PMD, SonarQube, JDeodorant, SpotBugs, SAD, iPlasma, inFusion, JSpIRIT, DECOR.

DECOR, JSpIRIT, inFusion, iPlasma были отброшены, так как они не были найдены в свободном доступе как полноценные инструменты или дополнения к IDE, либо они больше не поддерживаются. SAD и JDeodorant являются дополнениями к IDE Eclipse, однако в рамках исследования их не удалось настроить для выбранных проектов. Список дефектов, поддерживаемых в SpotBugs, не включает в себя антипаттерны кода, обнаруживаемые другими инструментами. В итоге, для дальнейшего исследования остались 3 инструмента: DesigniteJava, PMD, SonarQube.

PMD [23] — статический анализатор кода, который ищет стандартные ошибки программирования. Этот инструмент поддерживает 8 языков программирования, в том числе Java. В PMD есть некоторый список встроенных проверок кода (около 400 для Java), но также есть возможность их изменения и добавления собственных. Среди встроенных есть детекторы CognitiveComplexity (сложный метод), LongVariable (длинный идентификатор), GodClass (божественный класс), ExcessiveMethodLength (длинный метод).

DesigniteJava [24] – автономный инструмент оценки качества кода, написанного на Java. Версия, распространяемая в свободном доступе, ищет 17 антипаттернов проектирования и 10 антипаттернов реализации кода, в том числе LongMethod (длинный метод), ComplexMethod (сложный метод) и LongIdentifier (длинный идентификатор).

SonarQube [25] — инструмент автоматической проверки кода для обнаружения ошибок, уязвимостей и антипаттернов кода. Инструмент поддерживает 27 языков программирования. Для Java написано 621 правило, среди которых 153 проверки ошибок и 396 проверок антипаттернов кода, в том числе CognitiveComplexity (сложный метод). Чтобы найти антипаттерны кода, все три инструмента вычисляют метрики и сравнивают эти значения с некоторыми пороговыми. Если порог пересечён, отмечается нарушение.

Сравнение результатов. Все выбранные инструменты поиска антипаттернов кода рассматривают Java как целевой язык программирования, в связи с чем было принято решение использовать в рамках исследования набор из 9 популярных проектов с открытым исходным кодом на этом языке, а именно: ant[1], drill[2], error-prone[3], ExoPlayer[4], giraph[5], gson[6], guava[7], hive[8], truth[9].

В таблице 1.1 приведены численные характеристики указанных проектов, показывающие объём исходного кода, количество объектов для анализа. Наличие антипаттернов кода присуще, также, и другим языкам программирования (C++, C#, Python). Однако для этих языков не было найдено достаточное количество инструментов для рассмотрения.

Проект	Количество строк кода	Количество пакетов	Количество классов	Количество методов
ant	201k	93	1803	16844
drill	675k	389	6998	79465
error-prone	113k	61	3342	16618
ExoPlayer	238k	88	2038	21592
giraph	112k	134	1750	11125
gson	29k	23	489	3005
guava	548k	31	6840	75049
hive	1412k	590	12661	129015
truth	32k	6	305	4873

Таблица 1.1: Характеристики рассматриваемых проектов

Каждый из выбранных инструментов был запущен на всём наборе проектов. В табл. 1.2 отображено количество нарушений, которые были найдены инструментами накаждом из проектов. Для проектов ant, ExoPlayer, giraph, guava и

[1]https://github.com/apache/ant.git, f61ac1296
[2]https://github.com/apache/drill.git, 85d270f3
[3]https://github.com/google/error-prone.git, 54fa3f38
[4]https://github.com/google/ExoPlayer.git, 47b82cdc
[5]https://github.com/apache/giraph.git, 2c63aa23
[6]https://github.com/google/gson.git, f319c1b8
[7]https://github.com/google/guava.git, 3dfd7076
[8]https://github.com/apache/hive.git, de0a7ecb
[9]https://github.com/google/truth.git, eaddc49f

hive не удалось настроить SonarQube, поэтому отчеты по этим проектам отсутствуют.Из всех полученных данных были отфильтрованы только те записи, что относятся к 3 видам антипаттернов: длинный метод, божественный класс, сложный метод. Однако даже такой базовый список дефектов ищут не все инструменты.

Проект	**PMD**	**Designite**	**SonarQube**
ant	37261	5954	-
drill	201483	49644	15773
error-prone	54890	11390	2489
ExoPlayer	81847	24769	-
giraph	25436	6698	-
gson	9347	1928	385
guava	223384	52193	-
hive	666423	108106	-
truth	14890	5601	642

Таблица 1.2: Общее количество срабатываний инструментов на проектах

Далее на основе отчётов была проведена оценка корреляции результатов работы инструментов. Отсутствие общего стандарта описания ошибок и формирования отчетов усложняет процесс совмещения выявленных нарушений на объектах проекта по файлу/классу/методу.

На рис. 1.1 и рис. 1.2 изображены диаграммы Венна, показывающие все пересечения отчётов детекторов дефектов кода от различных инструментов. Сложные методы ищут все выбранные инструменты, поэтому приведена статистика только для запусков на общих проектах (на которых удалось запустить SonarQube). На рис. 1.1 видно, что значительная часть записей общая для всех инструментов (374 метода из 1132-х отмеченных хотя бы одним инструментом). DesigniteJava определяет как сложные ещё 422 метода, на которые другие инструменты не отреагировали. Длинные методы SonarQube не определяет, поэтому статистка бралась по всем проектам. Отчет PMD практически включает в себя отчет DesigniteJava, но также содержит ещё значительное количество записей о других методах. Божественный класс определяет только PMD, поэтому диаграмма Венна для этого антипаттерна кода не приведена.

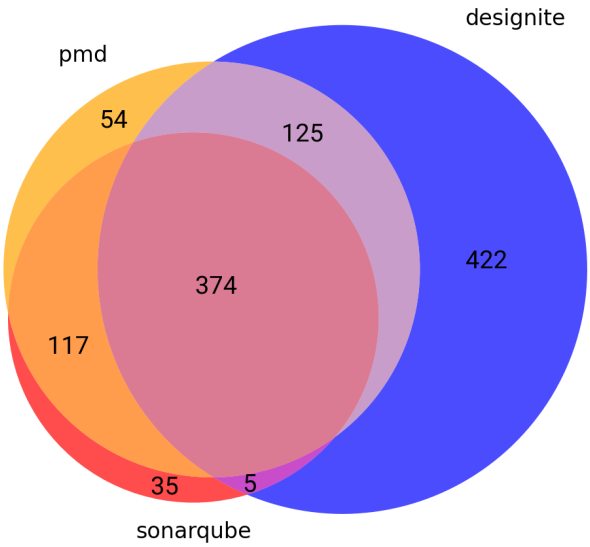

Рис. 1.1: Диаграмма Венна для «сложного метода»

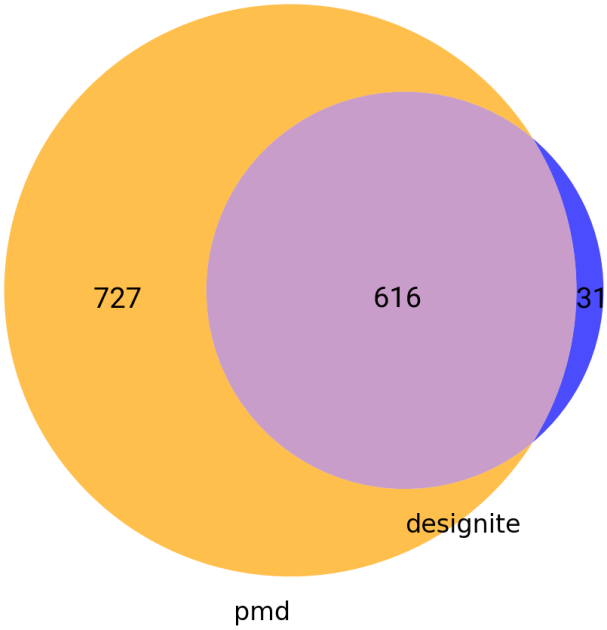

Рис. 1.2: Диаграмма Венна для «длинного метода»

Сравнение результатов. Для оценки точности обнаружения антипаттернов кода часть предупреждений была исследована вручную. Для каждого типа предупреждений была сформирована случайная выборка из 100 элементов (исключая автоматически сгенерированный и тестовый код), общих для всех инструментов. В рамках ручного анализа каждому предупреждению была назначена одна из трёх оценок — истинное срабатывание, ложное срабатывание и истинное срабатывание, не требующее исправления (англ. won't fix). В

табл. 1.3 представлены результаты анализа. Длинный метод как наиболее «понятный» дефект, обнаружение которого зачастую связано с количеством строк, имеет наивысший уровень истинных срабатываний, близкий к 90%. С другой стороны, при беглом осмотре части файлов проекта truth было обнаружено много методов, которые эксперты определили как «длинный», но инструменты их не обнаружили. Божественный класс и сложный метод показывают крайне низкий уровень истинных срабатываний (10% и 20% соответственно). Так как процесс оценки дефектов занимает определенное время, подобную статистику можно считать неприемлемой на практике.

Оценка эксперта	Божественный класс	Сложный метод	Длинный метод
Истинное срабатывание	10%	20%	90%
Истинное срабатывание, не требует исправления	21%	25%	5%
Ложное срабатывание	69%	55%	5%

Таблица 1.3: Статистика ложных срабатываний

Для демонстрации проблем, связанных с необходимостью выбора пороговых значений, рассмотрим следующую зависимость, которая отображает процент объектов, со значением метрики больше некоторого заданного. На рис. 1.3, рис. 1.4 изображены такие графики для метрик ATFD и WMC [26] - базовых метрик для определения божественного класса. ATFD (доступ к сторонним данным) — это количество атрибутов из несвязанных классов, доступ к которым осуществляется напрямую или путем вызова методов доступа. WMC (взвешенные методы класса) — сумма всех сложностей методов в классе. Вертикальной линией отмечены пороговые значения, взятые из документации PMD10 . Как видно, при одинаковом пороговом значении процент объектов, превышающих этот порог, а значит определяемых как божественный класс — сильно отличаются, по метрике ATFD 43.75% классов ant, тогда как в error-prone — 7%. Для метрики WMC при установленном пороговом значении error-prone имеет около 3% классов-нарушителей, тогда как truth — чуть больше 17%. Подобное поведение показывает, что пороговые значения необходимо подбирать индивидуально под проект. Рассмотрим также аналогичный график для метрики Cognitive

Complexity (некоторая вариация цикломатической сложности, разработанная в SonarQube) рис. 1.5 — по нему видно, что в проекте error-prone в целом значение этой метрики ниже, чем у drill на 5%, что свидетельствует о различных стилях оформления кода в этой паре проектов.

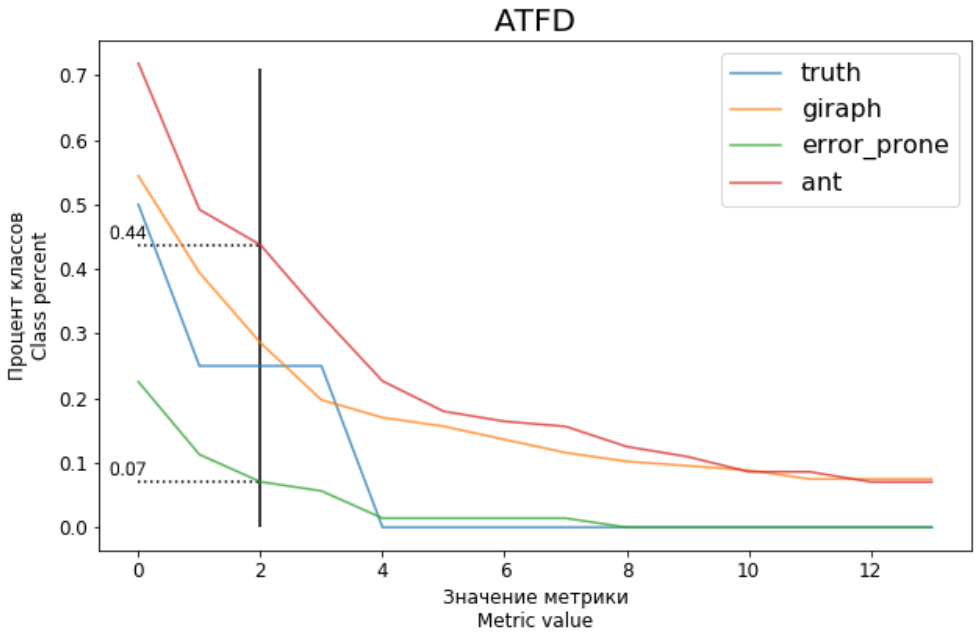

Рис. 1.3: График распределения классов по метрике ATFD

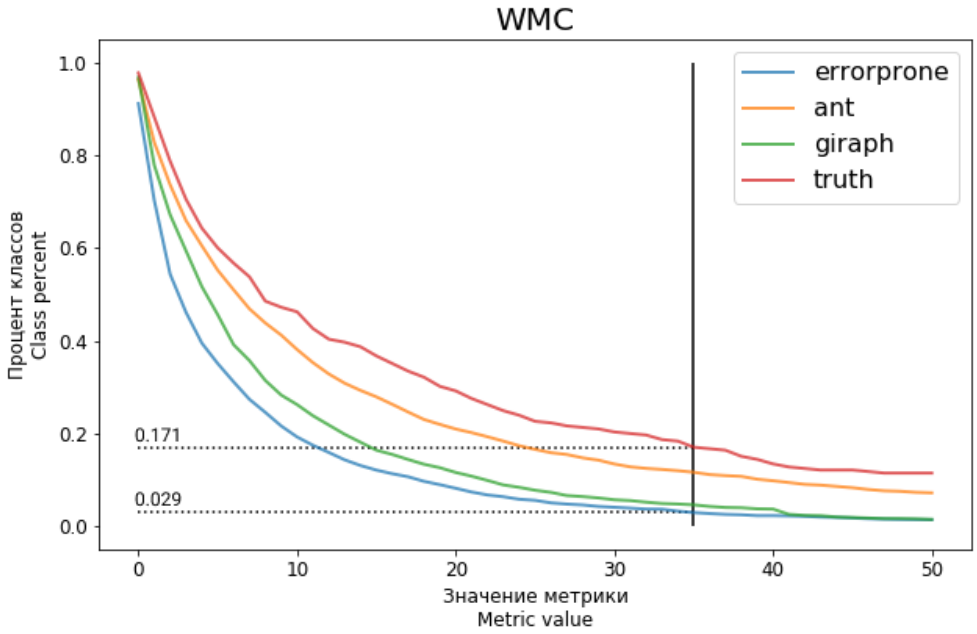

Рис. 1.4: График распределения классов по метрике WMC

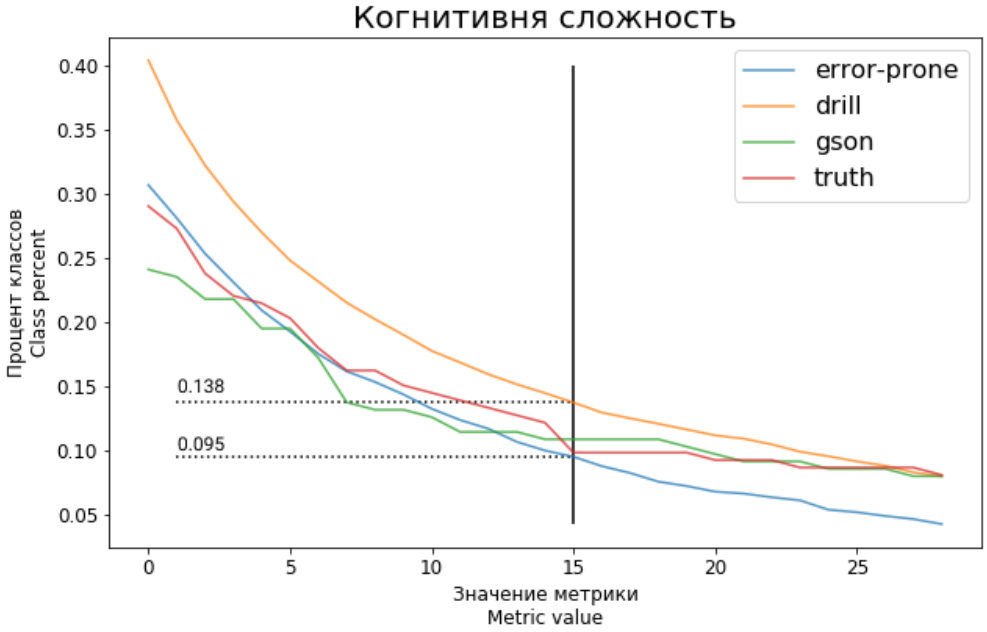

Рис. 1.5: График распределения классов по метрике когнитивной сложности

Таким образом существующие инструменты при стандартной настройке довольно шумные: PMD выдает предупреждения на 18 — 48% (ant и error-prone) строчек кода проекта (отношение числа репортов к количеству строк кода), Designite чуть сдержаннее — 3-17.5% (ant и truth).

Процент истинных срабатываний неоднородный и зависит от искомого дефекта: Для божественных классов и сложных методов довольно низкий (10% и 20% соответственно), для длинных методов — 92%, но это не делает данный детектор хорошим, так как полнота его отчёта низкая.

Также стоит отметить, что различные инструменты обращают внимание на различные недостатки в кодовой базе. Так, например, 374 из 1132-х общих сложных методов и 616 из 1374 общих длинных методов. Но также имеется и большое количество замечаний, которые пишут одни инструменты, но не другие (рис. 1.1, рис. 1.2).

1.4.2. Примеры внедрения подобных систем

Согласно исследованиям, внедрение систем автоматизированного рецензирования кода на основе искусственного интеллекта позволяет сократить время проведения проверки с 30 до 10 минут, что приводит к высвобождению до 20% рабочего времени разработчиков для решения более творческих и стратегиче-

ских задач. Этот эффект масштабируется: для крупной организации это эквивалентно экономии сотен тысяч человеко-часов ежегодно, как в случае с Google. Как отмечают эксперты Института системных наук IBM, стоимость устранения дефекта, обнаруженного на этапе эксплуатации проекта, приблизительно в шесть раз превышает затраты на его исправление в процессе разработки. В данном контексте автоматическое рецензирование кода приобретает особую значимость, поскольку обеспечивает выявление дефектов, включая критические уязвимости безопасности (SQL-инъекции, утечки ресурсов) и проблемы архитектуры, на самых ранних стадиях жизненного цикла программного обеспечения, что напрямую сокращает финансовые потери.

Опыт внедрения подобных систем в крупнейших технологических компаниях подтверждает их эффективность и окупаемость. Например, корпорация Microsoft обрабатывает с их помощью более 600 000 запросов на включение изменений (pull request) ежемесячно, что позволило сократить медианное время внедрения правок на 10-20% в рамках 5000 репозиториев. Компания ByteDance сообщает об обслуживании 12 000 активных пользователей в неделю с помощью своей системы BitsAI-CR. Точность предлагаемых системой предложений достигает 75%, что, наряду с низким процентом ложных срабатываний, является ключевым фактором, поддерживающим доверие и принятие со стороны разработчиков (74,5% положительных отзывов). Важно отметить, что успех внедрения, как показывают кейсы, напрямую зависит от способности системы минимизировать «шум» — при превышении порога в 30% ложных срабатываний разработчики склонны игнорировать ее рекомендации.

Таким образом, внедрение систем автоматического рецензирования кода демонстрирует значительные практические и экономические преимущества для IT-компаний и бизнеса в целом. Экономический эффект проявляется не только в прямом сокращении затрат на верификацию кода и последующее исправление ошибок, но и в масштабируемости процессов, позволяющей небольшим командам управлять огромными объемами кода без пропорционального роста штата. Практическая польза заключается в комплексном повышении общей эффективности, качества и безопасности процесса разработки, обеспечении соответствия стандартам (SOC 2, ISO 27001) и создании самообучающейся экосистемы, которая непрерывно совершенствуется. Для достижения максимального эффекта внедрение должно быть поэтапным, начинаясь с пилотных проектов,

и сопровождаться обучением команды и интеграцией в существующие CI/CD-конвейеры.

1.5. Выводы исследования

В первой главе проведено комплексное исследование современного состояния области автоматического рецензирования исходного кода и детектирования антипаттернов. Проведенный анализ позволил выделить два принципиально различных подхода к автоматическому рецензированию: системы, основанные на повторном использовании существующих рецензий, и решения, использующие генерацию новых рецензий с помощью языковых моделей. Предлагается провести сравнительный анализ существующих подходов и оценить применимость их предложений.

Сравнительный анализ существующих инструментов (PMD, SonarQube, DesigniteJava) детектирования антипаттернов выявил существенные расхождения в результатах их работы. Полученные результаты подтверждают актуальность разработки новых методов детектирования антипаттернов с повышенной точностью и обосновывают необходимость дальнейших исследований. Для достижения цели предлагается использовать гибридный подход, сочетающих численные метрики исходного кода с моделями машинного обучения для повышения точности детектирования.

Глава 2
Разработка детекторов антипаттернов

2.1. Анализ методов детектирования антипаттернов исходного кода

Как уже было описано выше антипвттерны кода могут охватывать различные уровни абстракции разработки ПО. Так Божественный Класс (англ. God Class) — это некоторый объект в исходном коде программы, который управляет слишком многими другими объектами в системе, превратившись в класс, который делает все. Например, класс Customer, который содержит в себе имя и фамилию человека, его адрес проживания, страну, город и почтовый код. Кроме простого хранения информации, необходимы ещё методы взаимодействия с данными полями и, возможно, некоторые служебные функции. С одной стороны, в этом нет ничего критического, однако намного удобнее иметь отдельный класс Address, который хранит внутри себя всю информацию об адресе и методы работы с ней, а в Customer держать только ссылку на объект класса Address. Более масштабный и неопределенный пример антипаттерна программирования — это Стрельба дробью (англ. Shotgun Surgery), который описывает ситуацию, когда при изменении незначительного поведения вашей программы в одном месте приходится делать дополнительные изменения в большом количестве других классов. Этот антипаттерн кода не ограничивается сущностью одного класса, для его обнаружения необходимо следить за историей изменения проекта и искать ситуации подобного множественного исправления текста программы. Кажется, что одними из наиболее простых для обнаружения и исправления дефектов кода являются Длинный Метод (англ. Long Method) и Сложный Метод (англ. Complex Method). Названия данных антипаттернов программирования говорят сами за себя, это слишком разросшиеся в «длину» или «ширину» функции. Сложность их обнаружения заключается в том, что иногда такая реализация необходима и иным способом ее не сделать. Например методы, содержащие большое количество switch/case конструкций либо большое количество схожих операций преобразования чисел/строк, которые проблематично вынести в подфункции. Также понятие «длинности» или «сложности» зависит от конкретного проекта, языка программирования, команды разработчиков. Исследования в рамках данной области ставят перед собой цель наиболее точно и без пропусков определять наличие возможных дефектов кода.

Инструменты поиска элементов признаков технического долга являются актуальным предметом исследования уже более 20-ти лет[27]. В последнее время для улучшения качества опеределения дефектов используются методы машинного обучения [28, 29, 30]. Для обучения этих моделей требуются наборы размеченных данных. В области создания таких наборов данных есть два направления: ручная [31, 32, 33] и автоматическая [34, 30] разметки. Наиболее актуальный представитель первой группы работ [33] содержит набор данных где собрано большое количество ручных разметок примеров, большая часть из которых оказались размечены как отрицательные примеры (без какого либо code smell). Представители второй группы используют готовый инструмент поиска дефектов исходного кода для обнаружения code smells на большом наборе проектов, анализируя последнюю версию проекта.

Основная проблема такого метода сбора примеров заключается в том, что исследователи/детекторы/оракулы, которые размечают положительные случаи, не являются разработчиками данного ПО, соответственно совершенно не знакомы с его строением и идеями архитектора ПО. Длинные и Сложные Методы вне зависимости от замыслов разработчиков —- плохая практика. Другое дело Завистливые Методы (англ. Feature Envy) и Божественные Классы: такие случаи технического долга сложно определить без детального анализа структуры проекта. Есть инструменты, которые принимают на вход полноценный проект, анализируют связи между классами/методами/переменными и меняют структуру, решая задачу оптимизации зависимостей между сущностями [35], предлагая свой вид проекта. У такого подхода основным недостатком является сложность проведения генерального рефакторинга, так как в автоматическом режиме проводить столь масштабные изменения рискованно

Ещё один подход к созданию набора данных с исправлениями исходного кода представлен в ManySStuBs4J Dataset[36]. В работе авторы собирают исправления реальных разработчкиков, которые те посчитали необходимым сделать. Однако исправления, которые ищут в статье являются однострочными: исправление пременной, численной константы, имени метода. Такой подход был применен к поиску примеров антипаттернов кода. А именно, поиск в истории изменения проектов с открытым исходным кодов тех исправлений, в которых разработчики исправляли имеющиеся дефекты своего проекта.

2.1.1. Существующие подходы

Авторы статьи [37] одними из первых опубликовали открытые наборы данных с примерами антипаттернов кода, представлили 243 примера дефектов пяти типов. Однако на момент исследования (февраль 2023г.) результаты недоступны. В работах [32, 38] говорится о 17350 и 40888 примерах 13 различных типов проявлений технического долга. Однако сами наборы данных закрыты, и получить к ним доступ не удалось.

Открытый набор данных приведен в [31], здесь рассмотрен набор из 1986 примеров для четырех типов антипаттернов кода. Недостатком можно считать то, что примеры взяты из набора проектов Qualitas Corpus [1], последняя дата обновления которого была в 2012 г., что означает меньшую актуальность кодовой базы.

Более новой публикацией подобного плана является [33], в которой авторы попросили 20 специалистов, имеющих различный опыт в разработке ПО, оценить некоторое количество примеров исходного кода на наличие в них симптомов технического долга. Итоговый набор данных содержит 14739 пометок для 4770 фрагментов кода. Безусловным плюсом данной работы можно отметить открытый доступ к самой разметке и опросам, на которые отвечали специалисты. К недостаткам стоит отнести довольно небольшое количество примеров, которые специалисты разметили как фрагменты, содержащие тот или иной дефект (1504 пометки «critical» или «major» для 795 различных примеров на четырех типах дефектов).

Описанные выше наборы данных объединяет ручная разметка примеров некоторым количеством проверяющих, что само по себе очень хорошо влияет на качество набора данных. Недостаток такого подхода следующий: этот процессе занимает значительное количество ресурсов опытных разработчиков, соответственно не может быть масштабируем на большие наборы данных в десятки тысяч примеров.

Другой подход используется в [30, 34]. Для получения наборов данных в этих работах применяются существующие инструменты поиска проявлений технического долга на большом количестве проектов с открытым исходным кодом. В работе [34] проводился анализ 33 проектов при помощи инструмента

[1] http://qualitascorpus.com/docs/history/20120401.html

SonarQube[2], в [30] было проанализировано 1,072 C# и 100 Java проектов инструментами Designite[3] и DesigniteJava[4] соответственно. Эти публикации содержат большое количество размеченных фрагментов кода, однако они получены некоторым инструментом, а не разработчиками. Разметку, составленную при помощи DesigniteJava, будем использовать в дальнейшем, так как сам инструмент имеет открытый исходный код и мы имеем доступ непосредственно к алгоритму детекторов признаков технического долга.

Основными ограничениями предложенных выше наборов данных являются две крайности. С одной стороны, их трудозатратность для создания и масштабирования при высокой точности разметки. С другой — вызывающее сомнения качество данных при практически неограниченном количестве примеров.

2.2. Построение набора данных

Инструменты поиска элементов технического долга являются актуальным предметом исследования уже более 20 лет [27]. В последнее время для улучшения качества определения таких дефектов используются методы машинного обучения [29],[28], [30]. В основе подобных методов стоит задача обучения с учителем на наборе размеченных данных. В области создания таких наборов данных есть два направления: ручная [33], [39], [31] и автоматическая [30], [34] разметки. Наиболее актуальный представитель первой группы работ [33] содержит набор данных, где собрано большое количество ручных разметок примеров. Большинство из них оказались размечены как отрицательные примеры (без какого-либо дефекта). Представители второй группы используют готовый инструмент поиска дефектов исходного кода для обнаружения антипаттернов на большом наборе, анализируя текущие версии проектов, а не историю изменений.

Основная проблема такого метода сбора примеров заключается в том, что исследователи/детекторы/оракулы, которые размечают положительные случаи, не являются разработчиками данного ПО, соответственно совершенно не знакомы с его строением и идеями архитектора ПО. Длинные и Сложные Методы вне зависимости от замыслов разработчиков – плохая практика. Другое

[2]https://www.sonarsource.com/ .

[3]https://www.designite-tools.com/ .

[4]https://github.com/tushartushar/DesigniteJava/ .

дело Завистливые Методы (англ. Feature Envy) и Божественные Классы: такие случаи технического долга сложно определить без детального анализа структуры проекта. Есть инструменты, которые принимают на вход полноценный проект, анализируют связи между классами/методами/переменными и меняют структуру, решая задачу оптимизации зависимостей между сущностями [35], предлагая свой вид проекта. У такого подхода основным недостатком является сложность проведения генерального рефакторинга, так как в автоматическом режиме проводить столь масштабные изменения рискованно.

Еще один подход к созданию набора данных с исправлениями исходного кода представлен в ManySStuBs4J Dataset [36]. В этой публикации авторы собирают исправления реальных разработчиков, которые те посчитали необходимым сделать. Однако исправления, которые ищут, являются однострочными: замена переменной, численной константы, имени метода. Такой подход мы хотим применить к поиску примеров антипаттернов кода. А именно, будем искать в истории изменения проектов с открытым исходным кодов те исправления, в которых разработчики исправляли имеющиеся дефекты своего проекта.

Рассмотрим схему получения набора данных, который доступен публично на Zenodo [40].

Выбор проектов с историей изменений. Для дальнейшего анализа было загружено топ-100 проектов с Github по количеству положительных отметок, основным языком которых является Java. Полный список и версии проектов можно найти в отдельном файле *projects.csv* вместе с набором данных. Большинство исследуемых проектов находятся в стадии активной разработки, 30% проанализированных изменений исходного кода приходятся на последние три года.

2.2.1. Инструмент сбора примеров

Инструмент анализирует каждый проект по очереди, проходя всю историю его изменений. На каждом шаге при помощи созданных детекторов антипаттернов кода происходит анализ изменившихся в коммите файлов. Если анализ показал, что в коммите есть случаи исправления определенного дефекта, то каждый такой пример попадает в итоговый список своего детектора.

Определение исправлений кода. Будем считать парой ДО и ПОСЛЕ фрагменты кода перед исправлением, вносимым разработчиком, и после него соответственно. Детекторы ищут дефекты в версии файлов ДО и в версии файлов ПОСЛЕ. Затем отчеты этих двух запусков сравниваются. Если в файле пропал дефект в каком-то методе или классе, то этот случай фиксируется и происходит дополнительная проверка двух фрагментов.

Детекторы антипаттернов кода. В текущем эксперименте использовалось три детектора: Божественный Класс (God Class), Сложный Метод (Complex Method), Длинный Метод (Long Method).

Тип дефекта	Набор данных	Положительных	Отрицательных
Сложный Метод	DLS	22000	22000
Длинный Метод	MLCQ	140	2119
Божественный Класс	MLCQ	246	1719

Таблица 2.1: Описание обучающей выборки базовых моделей

Для обучения базовых моделей применялись наборы данных MLCQ[33] и DLS[30], численные характеристики которых указаны в табл. 2.1.

В силу ограниченности количества примеров для Длинных Методов и Божественных Классов был использован подход, основанный на подсчете метрик по коду и модель машинного обучения Случайный Лес (Random Forest) для бинарной классификации фрагментов кода. Для Длинных Методов применялись LOC, NCLOC, MCC, TokenCount; для Божественных Классов — LCOM5, NAD, NMD[3]. Для работы со Сложными Методами была реализована сверточная нейронная сеть (CNN-2D), описанная в[30].

Постфильтры. В ходе валидации результатов сбора примеров стало ясно, что некоторые из исправлений, которые мы находили, не были нацелены на исправление дефектов, связанных с возможными дефектами кода. В «положительные» примеры исправления Длинных Методов попадали ситуации, в которых разработчик вносил незначительные стилистические исправления в свой код. В противоположной ситуации весь Длинный Метод переносился, оставляя в старом только вызов нового метода.

Для фильтрации таких примеров были написаны проверки на основе количественных метрик вносимых изменений. Итоговые фильтры получились сле-

дующими (срабатывание хотя бы одного из условий исключает данное исправление).

Длинный Метод:

- тело метода ПОСЛЕ меньше пяти строк;
- разница длин методов ДО и ПОСЛЕ меньше 12;
- относительное уменьшение длины метода меньше 0.2;
- нет достаточно длинного сплошного участка удаления кода из фрагмента ДО с заменой на небольшой участок в ПОСЛЕ.

Сложный Метод:

- тело метода ПОСЛЕ меньше пяти строк;
- относительное уменьшение сложности (по McCabe's Cyclomatic Complexity[5]) фрагмента меньше 0.25.

Божественный Класс:

- разница между количеством удаленных и добавленных методов меньше трех.

Все пороговые значения были получены экспериментально исходя из тестовой выборки объемом $\sim 5\%$. Полученные детекторы и дополнительные фильтрации были использованы в анализе истории изменений 100 проектов.

Псевдокод разработанного метода сбора данных представлен на листинге 1.

Листинг 2.1: Псевдокод алгоритма автоматического сбора примеров дефектов кода с исправлениями

```
FUNCTION collect_dataset(repositories)
    dataset = []
    FOR each repo IN repositories:
        commits = get_commit_history(repo)
        FOR each commit IN commits:
            changed_files = get_changed_files(commit)
            FOR each file IN changed_files:
                old_code = get_old_version(file, commit)
                new_code = get_new_version(file, commit)

                FOR each detector_type IN detectors:
```

[5]https://radon.readthedocs.io/en/latest/intro.html#cyclomatic-complexity/ .

```
                check_and_add_example(
                    detector_type,
                    old_code,
                    new_code,
                    commit,
                    dataset
                )

    RETURN dataset
END FUNCTION

FUNCTION check_and_add_example(detector_type, old_code,
    new_code,
    commit,
    dataset
)
    detector = detectors[detector_type]
    old_contains_defect = detector.detect_fix(old_code)
    new_contains_defect = detector.detect_fix(new_code)

    IF old_contains_defect AND NOT new_contains_defect:
        example = create_example(old_code, new_code, commit)
        IF heuristics_filter.check(old_code, new_code):
            dataset.add(example)

END FUNCTION
```

2.2.2. Валидация полученных примеров

Для более удобного просмотра и валидации примеры отправлялись в LabelStudio[6] — инструмент для совместной аннотации примеров. Вручную было размечено 132 примера Длинных Методов, 164 примера Сложных Методов и 32 примера Божественных Классов. Аннотацию полученных примеров (табл. 2.2)

[6]https://github.com/heartexlabs/label-studio/ .

проводили два опытных разработчика. Коэффициент согласия составил 92%.

Тип дефекта	Всего	Положительных, (%)	Отрицательных
Сложный Метод	164	106 (65%)	58
Длинный Метод	132	97 (73%)	35
Божественный Класс	32	24 (75%)	8

Таблица 2.2: Аннотация полученных примеров

Не очень высокий уровень положительных разметок, однако, не означает, что остальные пары не имеют дефекта в коде ДО, разметка проводилась с целью оценки пар с исправлениями дефектов. В остальных $\sim 25 - 35\%$ пар изменения не были явно направлены на рефакторинг или исправления искомого дефекта.

2.2.3. Характеристики данных

Ниже представлены численные характеристики собранных данных и описание предоставляемого архива.

Основные численные характеристики приведены в табл. 2.3. При анализе 100 проектов было получено около 6000 пар фрагментов исходного кода (с дефектом и без него).

Характеристика	Значение
Общее число примеров	5912
Сложных Методов	2967
Длинных Методов	2517
Божественных Классов	428
Проанализировано проектов	100
Проанализировано коммитов	788485

Таблица 2.3: Общие характеристики полученных данных

Структура данных. Основная директория архива dataset содержит три поддиректории по каждому типу антипаттернов: ComplexMethod, GodClass,

LongMethod, в каждой из которых хранится список директорий с примерами before.java и after.java. Также на верхнем уровне находятся файлы с описанием примеров: ComplexMethod.csv, GodClass.csv, LongMethod.csv. Каждая запись о примере содержит следующую информацию:

- `project` – имя проекта, из которого взят пример,
- `path` – относительный путь к файлу в директории репозитория,
- `entity` – идентификатор интересующей нас сущности в формате ИмяКласса:ИмяМетода(типы аргументов),
- `url` – гиперссылка на коммит,
- `before file` – путь к файлу, содержащему метод/класс до изменения, относительно директории dataset,
- `after file` – путь к файлу, содержащему метод/класс после изменения, относительно директории dataset.

Также в корневой директории dataset содержится файл projects.csv, описывающий использованные для анализа проекты.

2.2.4. Апробация данных

Полученный набор данных можно использовать для обучения новых моделей машинного обучения для создания более точных детекторов технического долга. Из этих пар примеры до исправлений брались как код с дефектом, а примеры после исправлений — как код без дефекта.

Количество примеров Длинных Методов стало значительно больше, чем в наборе данных MLCQ, соответственно, появилась возможность обучить более сложную модель, а именно LSTM, описанную в [30].

В данном эксперименте сравниваем модели, обученные на новых данных с предыдущими результатами. В качестве валидационных данных применялись размеченные вручную примеры из проектов представленных выше в таблице 1.1: ant, drill, error-prone, giraph, hive, truth.

При разметке этого валидационного набора данных использовались три пометки: истинные срабатывания (TP), ложные срабатывания (FP) и истинные срабатывания, не требующие исправления (WF). Применение пометки «не требующие исправления» вызвано тем, что в данном случае «дефект» мог вноситься осознанно и специально, либо исправление такого участка кода потре-

бует значительных затрат ресурсов. Оценивались точность (PR), полнота (RE) и F1-мера (F1).

Модель	F1	PR	RE	WF	TP	FP
Длинный Метод						
Базовая	0.277	0.163	0.928	116	39	84
Новая	0.493	0.346	0.857	47	36	21
Сложный Метод						
Базовая	0.518	0.35	1.0	36	70	94
Новая	0.757	0.626	0.957	23	67	17
Божественный Класс						
Базовая	0.092	0.05	0.625	68	5	27
Новая	0.141	0.079	0.625	37	5	21

Таблица 2.4: Результаты валидации обученных моделей

Как видно из табл. 2.4, результаты моделей по всем типам дефектов улучшились. Для Длинных Методов за счет перехода на более сложную модель удалось получить значительное улучшение точности при незначительной потере полноты.

Таким образом показано, что данный набор данных можно использовать для обучения новых более точных детекторов дефектов исходного кода.

2.2.5. Текущие ограничения подхода

Очевидно, что не все участки исправлений данных типов дефектов кода на рассматриваемых проектах были обнаружены. С другой стороны, применение моделей, основанных на методах машинного обучения, и ручные эвристики могут вносить и ложноположительные срабатывания, однако такие примеры, скорее, не являются исправлениями дефекта. Метод/класс так или иначе стал меньше и проще для понимания, однако, возможно, не настолько, насколько это можно было сделать, если бы рефакторинг проводился целенаправленно.

Кроме того, встречается рефакторинг вынесения значительного участка кода из метода/класса, который нельзя считать хорошим, если он был выделен целиком и стал новым примером дефекта кода.

Также в работе не были рассмотрены ситуации с переименованием файлов в коммитах, учитывались только изменения в рамках одного файла. Данный набор был собран исключительно на проектах с Java. Применять такой подход возможно и для других языков программирования, однако это требует наличия относительно неплохих базовых детекторов.

2.3. Разработка детектора

В ходе исследований был разработан прототип инструмента обнаружения длинных методов. В данной секции мы рассмотрим его устройство, а также как к нему может быть применена система фильтрации срабатываний, использующая методы машинного обучения.

В качестве алгоритма классификации нами была взята модель произвольного леса деревьев (англ. Random Forest) из библиотеки sklearn[7]. В качестве входа для модели предоставлялись метрики исходного кода, подсчитанные на методах. Данные для обучения были получены из набора MLCQ, содержащий 2430 уникальных примеров методов. Изначально авторы разделили методы на 4 категории по степени уверенности наличия в примере длинного метода. Далее эта разметка была переведена в бинарный вид наличия или отсутствия дефекта кода. А также нами был реализован прототип для сбора метрик исходного кода: LOC (количество строк кода), NCLOC (количество строк кода без комментариев и пустых строк), MCC (цикломатическая сложность Маккаби), TokenCount (количество токенов).

2.3.1. Фильтрация результатов

После обнаружения моделью длинного метода, при помощи анализа графа потока данных, для каждого блока кода (в случае Java — фрагмент кода, ограниченный фигурными скобками) определяются зависимости переменных внутри этого блока от переменных, находящихся во вне. Далее выбирается блок, в котором зависимых переменных меньше, чем в остальных. Такой блок мы считаем кандидатом для вынесения в отдельную функцию и производим соответствующую трансформацию, а именно создается отдельный файл, класс и метод для этого блока, а в исходном коде на его место вставляется вызов созданного ме-

[7]https://scikit-learn.org .

тода. Далее проводится валидация описанной выше модификации, состоящая из двух шагов.

На первом шаге уже упомянутая модель запускается на обоих участках кода и проверяет, что среди них нет длинных методов. Если данное условие не выполняется, операция выделения откатывается, а предупреждения о том, что исходные метод является длинным, отфильтровывается. На втором шаге те же участки кода, что и в первом шаге, обрабатываются через модель code2vec[41]. Данная модель по участку кода формирует внутреннее представление, описывающее его семантику, и далее по этому представлению присваивает несколько имён каждому методу. Если для двух участков кода присвоенные их методам имена не пересекаются, мы считаем такие участки семантически различимыми. В случае если выделенный блок текста, помещенный в новый метод нового класса, семантически различим с исходным длинным методом, из которых он был изъят, мы считаем такую трансформацию успешной и, тем самым, подтверждаем, что исходный метод был длинным. В отличном случае, как и в шаге 1, данная трансформация откатывается, а предупреждение о том, что метод является длинным, отфильтровывается. Полная схема отображена на рисунке 2.1.

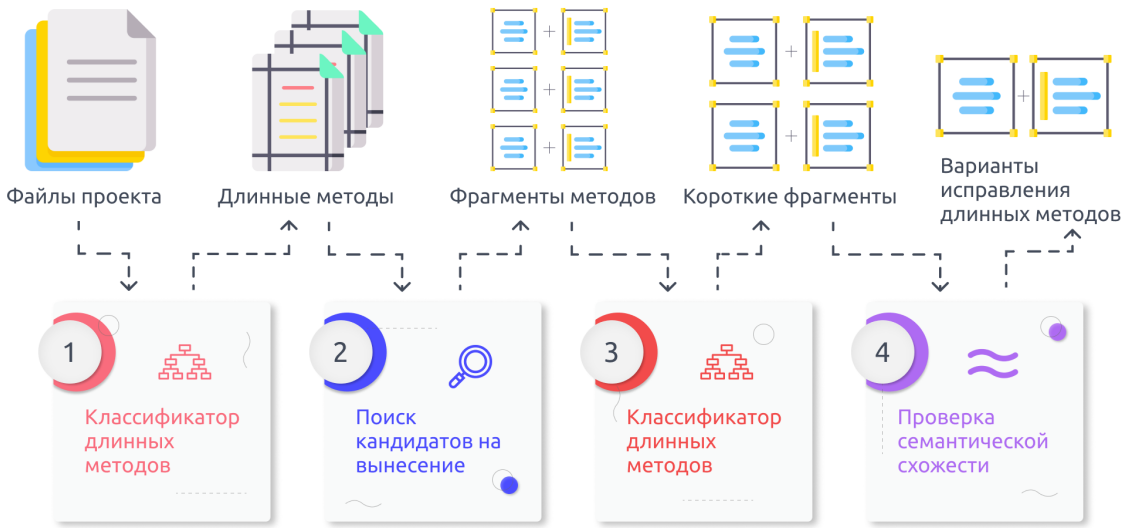

Рис. 2.1: Схема работы детектора длинных методов с дополнительной фильтрацией

Детектор длинных методов, реализованный подобным образом, имеет несколько особенностей.

Во-первых, применение методов машинного обучения должно помочь использованию подобного инструмента на различных кодовых базах без дополнительной настройки.

Во-вторых, так как разработчик, оценивая предупреждения рассматривает возможные варианты их исправления, шаги 1 и 2 фильтрации отбрасывают как раз то, что было бы оценено человеком как ложные срабатывания или истинные срабатывания, не требующие исправления. Для оценки эффективности фильтрации предупреждений о длинных методах мы запустили прототип классификатора на проектах из части 3.2. На результатах классификации была запущена 2-х шаговая фильтрация, описанная выше.

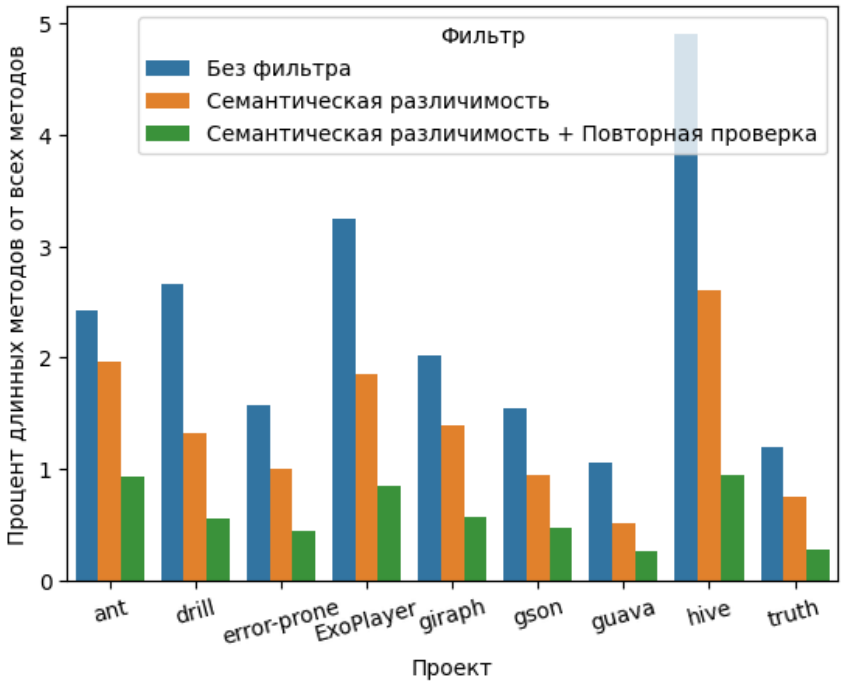

Рис. 2.2: Эффективность фильтрации длинных методов в зависимости от алгоритма

На рис. 2.2 приведены результаты этой фильтрации. При использовании алгоритма, основанного на семантической различимости выносимого блока от его контекста, прототип отфильтровал 19-50% (проекты ant и guava) предупреждений о длинных методах. Если же помимо указанного алгоритма дополнительно фильтровать те методы, которые после предложенной прототипом трансформации остаются длинными методами, то удастся снизить общее количество предупреждений на 60-80% (проекты ant и hive) от изначального числа.

2.3.2. Оценка качества разработанных методов фильтрации

Из всех полученных базовым классификатором предупреждений были выбраны случайным образом по 30 примеров из каждого проекта. Далее три эксперта разметили эти примеры на три вышеупомянутые категории: истинное срабатывание, ложное срабатывание и срабатывание, не требующее исправления. Для оценки согласованности разметки была использована каппа-статистика Коэна. Коэффициенты парной согласованности:0.888, 0.826, 0.848. Общий коэффициент согласованности определен как среднее значение парных статистик —- 0.85. К разметке были применены описанные выше фильтры: семантическая различимость и семантическая различимость с повторной проверкой классификатором.

Разметка экспертов	До фильтрации, количество (%)	После фильтрации, количество,(%)
Не является антипаттерном	125 (47,4%)	**50 (32,8%)**
Не требует исправления	83 (31,4%)	51 (33,5%)
Длинный метод	56 (21,2%)	**51 (33,5%)**
Всего	264	152

Таблица 2.5: Результаты фильтрации антипаттернов

В табл. 2.5 приведены результаты применения фильтров к размеченным данным. В первом столбце указаны типы меток. Во втором и третьем столбцах представлена статистика распределения классов для исходного классификатора. Видно, что к истинным срабатываниям отнесли 56 (21,2% от всех) предупреждений. В третьем столбце показаны данные по результатам после применения фильтра, основанного на выделении возможного кандидата и сравнении его семантики с базовым методом. Видно, что количество ложных срабатываний значительно снизилась с 47,4% до 32,8%, процент примеров, не требующих исправления почти не изменился, а процент истинных срабатываний 33,5% от общего итогового числа. Дополнительная проверка того, остается ли метод длинным, устраняет более 75% ложных срабатываний и тех, что не требуют исправления,

но также уменьшает и количество истинных примеров. Такое поведение можно объяснить тем, что алгоритм пытается извлечь в отдельный метод только один блок операций. В случае очень крупного метода такого исправления может быть недостаточно, чтобы полученный метод больше не считался длинным. Исследование других предикатов фильтрации и извлечения нескольких участков кода выходят за рамки данной работы.

2.4. Выводы

Во второй главе проведено комплексное исследование, посвящённое разработке и валидации методов автоматического детектирования антипаттернов исходного кода с применением методов машинного обучения. Разработан и реализован алгоритм автоматического сбора примеров антипаттернов кода из истории разработки проектов с открытым исходным кодом на языке программирования Java. Созданный в результате набор данных, включающий 5912 размеченных примеров трёх категорий антипаттернов («Длинные методы», «Сложные методы», «Божественный класс»), представляет значительную ценность для научного сообщества и опубликован в открытом доступе.

Проведённое сравнительное тестирование продемонстрировало существенное улучшение качества детектирования антипаттернов при использовании моделей, обученных на новом наборе данных. Наибольший прирост эффективности показали детекторы для «Длинных методов» (улучшение F1-меры на 78%) и «Сложных методов» (улучшение F1-меры на 46%). Важным результатом работы стало создание не только системы детектирования, но и механизма генерации предложений по исправлению выявленных антипаттернов, что повышает практическую ценность разработки для использования в реальных процессах разработки программного обеспечения.

Полученные результаты подтверждают эффективность предложенного подхода к автоматическому созданию обучающих наборов данных и перспективность использования комбинированных методов машинного обучения для решения задач статического анализа кода. Разработанные решения могут быть интегрированы в современные инструменты непрерывной интеграции и использованы для повышения качества программного обеспечения.

Глава 3
Методика автоматического рецензирования исходного кода

В процессе разработки программного обеспечения код проходит через множество итераций, включающих внесение новых изменений, исправление ошибок и рефакторинг. В условиях командной разработки важную роль играет механизм рецензирования кода (англ. code review), который позволяет выявлять потенциальные проблемы, улучшать читаемость и обеспечивать соответствие стилевым и архитектурным стандартам. Однако процесс ручного рецензирования требует значительных временных затрат со стороны разработчиков и может быть субъективным.

Общая схема рецензирования представлена на рисунке 3.1.

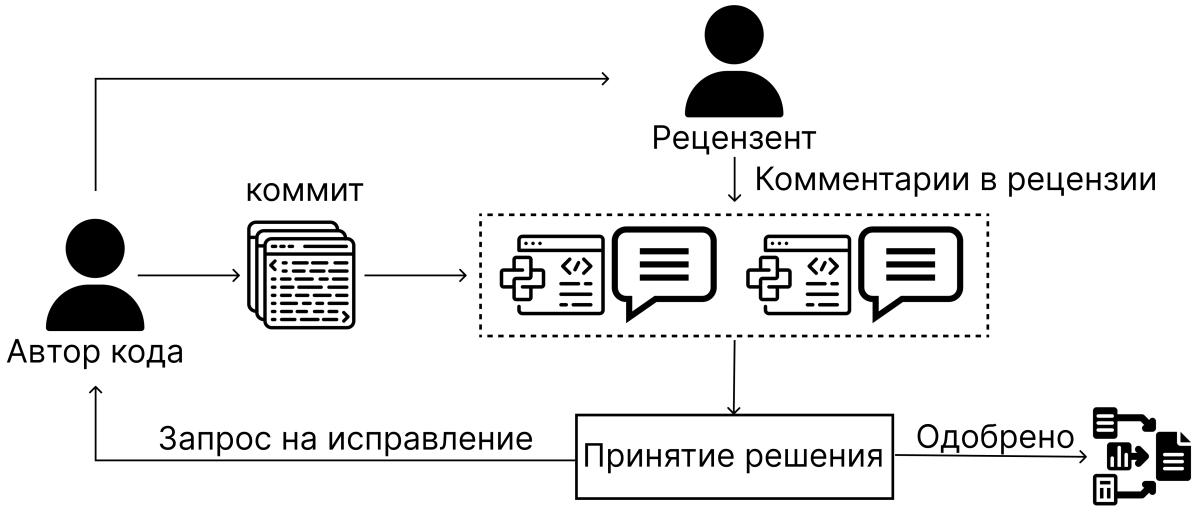

Рис. 3.1: Общая схема рецензирования

Автоматизированная генерация комментариев к изменениям кода на основе исторических данных рецензирования представляет собой задачу повышения эффективности процессов анализа кода. Такой инструмент должен анализировать различия между версиями кода, выявлять значимые изменения и формировать осмысленные текстовые комментарии, аналогичные тем, которые оставляют эксперты.

Пусть:

- $\mathcal{C}_{\text{prev}}$ — исходное состояние кода до изменения,
- $\mathcal{C}_{\text{new}}$ — состояние кода после изменения,

- $\mathcal{D}$ — разница между $\mathcal{C}_{\text{prev}}$ и $\mathcal{C}_{\text{new}}$, представленная в унифицированном формате,
- $\mathcal{H} = \{(\mathcal{C}_{\text{prev}}^{(i)}, \mathcal{C}_{\text{new}}^{(i)}, \mathcal{D}^{(i)}, \mathcal{R}^{(i)})\}_{i=1}^{N}$ — исторический набор данных, содержащий N примеров изменений кода и соответствующих им рецензий $\mathcal{R}^{(i)}$.

Требуется построить модель $\mathcal{M}$, которая для произвольного изменения кода, заданного кортежем $(\mathcal{C}_{\text{prev}}, \mathcal{C}_{\text{new}}, \mathcal{D})$, генерирует релевантный, точный и применимый текстовый комментарий $\mathcal{R}$:

$$\mathcal{R} = \mathcal{M}(\mathcal{C}_{\text{prev}}, \mathcal{C}_{\text{new}}, \mathcal{D}; \Theta, \mathcal{H})$$

где Θ — параметры модели.

Для успешного внедрения в рабочий процесс автоматизированная система генерации комментариев должна удовлетворять нескольким базовым требованиям. Генерируемые тексты должны быть грамматически правильными, осмысленными и точно отражать суть модификаций кода, без включения лишней информации. Ключевым фактором является их практическая направленность: комментарии должны давать разработчику четкий совет — например, указание на необходимость изменения алгоритма или добавления проверок. Кроме того, время генерации должно оставаться приемлемым для интеграции в процесс рецензирования.

3.1. Анализ и валидация существующих генераторов рецензий к коду

3.1.1. Анализ и разработка генератора комментариев с использованием исторических данных

Как уже было описано в первой главе — основной работой в области автоматического рецензирования с переиспользованием исторических данных является CommetnFinder [12]. Общая схема его работы заключается в векторизации входящих данных об изменениях в коде, поиск схожего кода в своей исторической базе и рекомендация комментариев, написанных к этим похожим участкам кода для нового случая.

Целью оценки авторов CommentFinder была проверка его эффективности и сравнение с современными подходами на основе глубокого обучения. В исследовании использовался тот же набор данных, что и в предыдущем методе

на основе DL[42], чтобы обеспечить справедливое сравнение. Этот набор данных состоял из 151 019 пар код-комментарий, полученных из 11 289 проектов с открытым исходным кодом, включая репозитории GitHub и Gerrit. Каждая пара содержала измененный метод и соответствующий ему комментарий к обзору кода, написанный человеком. Набор данных был тщательно отобран, чтобы исключить нерелевантные, дублированные или сгенерированные ботами комментарии, что обеспечило качество оценки.

Перед началом работы авторами было поставлено три исследовательских вопроса:

- Насколько эффективен CommentFinder в рекомендации точных комментариев участку кода по сравнению с подходом на основе DL?
- Насколько эффективен CommentFinder с точки зрения времени вычислений?
- Как различные методы похожести влияют на эффективность и производительность CommentFinder?

Для измерения производительности оценка использовала две метрики: точные предсказания, которые подсчитывали случаи, когда рекомендуемый комментарий полностью соответствовал истинному, и результат метрики BLEU [43], которые количественно определяли текстовое сходство между рекомендуемым и истинными комментариями. Оба показателя были протестированы на разном количестве кандидатов на рекомендации (1, 3, 5 и 10).

В своей работе авторы отмечают превосходство CommentFinder как в точности, так и в эффективности. Реализованный подход достиг на 32%-51% больше идеальных предсказаний (6.16% для топ-10 рецензий), чем подход на основе DL, продемонстрировав свою способность предоставлять качественные комментарии по участкам кода. Кроме того, он превзошел предыдущий метод по баллам BLEU на 3%-28%, что указывает на более высокое текстовое сходство с комментариями, предоставленными рецензентами [12].

С точки зрения эффективности CommentFinder предложил свои рекомендации всего за 4 минуты по сравнению со 194 минутами, требуемыми модели глубокого обучения. Анализ методов схожести показал, что выбор косинусного расстояния и Gestalt Pattern Matching оказались наиболее экономически эффективной комбинацией. Эти результаты подчеркивают практические преимущества CommentFinder для масштабируемых и ресурсоэффективных процессов проверки кода.

Ручная валидация. При дальнейшей работе с авторскими данными и открытой реализаций были обнаружены проблемы реализации и идеи подхода:

- В реализации скрипта сравнения некорректно была использована метрика BLEU. Так единицей сравнения были не токены, а отдельные символы, таким образом высокие результаты получались при совпадении слов или их частей, что не отражает общий смысл комментария и не позволяет говорить о семантической схожести рекомендуемой и истинной рецензий.
- При анализе результатов точных попаданий было обнаружено, что подавляющее большинство совпадающих комментариев состояли из одного или двух слов («suggestion», «rename», «remove that»).
- В качестве векторного представления используется мешок слов (англ. bag-of-words) для исходного текста. Такой подход очень тяжелый вычислительно для большого разнообразия слов в итоговом словаре. Обработка данных очень простая, отсутствует токенизация в случае CamelCase или snake_case, обрабатывается пунктуация. Также используются спорные участки, такие как константы и DateTime, которые загрязняют векторное представление.
- Такой подход приводит к огромным векторам в 500000 элементов на каждое слово с одной единицей и нулями в остальных позициях. Такая ситуация приводит к проблемам «проклятия размерности» и проблемам производительности.
- Текущая функциональность «поиска похожести» основана на вычислении матрицы косинусного сходства в памяти. Несмотря на то, что она работает с разреженными входными матрицами, на поиск комментариев требуется 20 ГБ свободной RAM.

3.1.2. Модификация подхода

Для получения новых данных, обучения и проверки новых моделей был реализован сборщик/краулер (англ. crawler), который взаимодействует с API платформ, извлекая комментарии к коду, оставленные в процессе рецензирования. Для проектов на GitHub собираются комментарии к Pull Requests, которые были слиты, в то время как для проектов на Gerrit собираются комментарии к ревизиям коммитов.

Каждый собранный комментарий содержит следующие метаданные:

- текст сообщения (message),
- дата и время его оставления (timestamp),
- относительный путь к файлу в проекте (file_name),
- положение интересующего участка кода в файле (range, представленное в формате start_line:start_column-end_line:end_column),
- текст участка кода, на который был оставлен комментарий (attention_area),
- HTML и API ссылки на комментарий (для GitHub),
- список булевых значений по эвристическим фильтрам (filters),
- язык программирования, определяемый по расширению файла (language).

Для повышения качества данных и исключения неинформативных комментариев были разработаны и применены эвристические фильтры. Основной фильтр, comment_message, предназначен для исключения комментариев, не несущих ценной информации, таких как «LGTM but see comments», «Ok, thanks», «ditto for others», «Ack». Эти комментарии не являются мотивационными и не требуют детального рассмотрения. Фильтр реализован на основе эвристически подобранных наборов правил и регулярных выражений. Результаты вручную собранных тестов показали, что текущая версия фильтра прошла 343 теста из 419.

Дополнительный фильтр, patch_set, проверяет, были ли последующие изменения в методе, который комментировался. Для Gerrit это проверяется на последней ревизии коммита, которую одобрили, а для GitHub — сравнивается с итоговым коммитом, который был замержен. Этот фильтр также направлен на выявление мотивационных комментариев, которые указывают на необходимые изменения.

Третий фильтр, first_in_thread, проверяет, является ли сообщение первым в последовательности сообщений или треде. Зачастую мотивационными являются именно первые сообщения, а не последующие в дискуссии. Хотя можно встретить объемные и основательные пояснения, которые являются ответами на предыдущие комментарии — процент таких полезных сообщений довольно мал.

Собранный датасета из репозитория Android за период с 01.01.2020 по 31.12.2022, состоит из 154928 рецензий. Из них 17671 пример не прошел фильтр comment_message, 24654 примера не прошли фильтр patch_set, 23132 примера

не прошли фильтр first_in_thread, а 51966 примеров не прошли хотя бы один из фильтров. Таким образом применениим фильтров удалось очистить первоначальны набор данных на 33% от предположительно «грязных» и бесполезных данных. Эта статистика свидетельствуют о важности применения эвристических фильтров для улучшения качества собранных данных, что критично для последующего обучения и проверки моделей.

Процесс формирования датасета является одной из ключевых задач в контексте разработки инструментов анализа исходного кода. Полнота и разнообразие данных в обучающем наборе непосредственно определяют степень охвата реальных сценариев использования, а также качество предложенных инструментом комментариев.

На текущем этапе осуществлялся сбор комментариев из следующих источников: Tizen Gerrit (после применения фильтров / общее число) (40832/109326), Android Gerrit (112196/171164), Top-100 C/C++ (94422/139079), Top-100 Java (79687/118184) и Top-250 Python проектов с Github.

В ходе исследований была подтверждена гипотеза о целесообразности раздельного использования примеров для различных языков программирования. Данный подход позволяет существенно сократить размер векторных представлений и оптимизировать временные затраты на выполнение запросов к базе данных.

Полученные в процессе работы краулера файлы, а также области кода (attention_area), к которым были оставлены комментарии, подвергались предварительной обработке с целью их последующего индексирования и эффективного поиска. В качестве структуры данных для хранения информации использовался предложенный в CommentFinder InMemoryIndex с применением метода CountVectorizer.

Для повышения качества векторизации CountVectorizer требовалась корректная токенизация исходного кода. Для решения данной задачи была выбрана open-source библиотека tree-sitter, обеспечивающая синтаксический разбор кода, определение метода как базовой единицы анализа, а также разбиение на токены. Полученные последовательности токенов затем передавались в CountVectorizer для построения векторных представлений. Процесс инференса включал аналогичные этапы: разбор исходного кода, токенизацию и преобразование в векторное пространство. Для измерения степени схожести между

векторами в рамках CommentFinder использовался показатель cosine_distance, дополненный алгоритмом Gestalt Pattern Matching для оценки текстовой близости методов.

Улучшение алгоритмов и метода сопоставления участков исходного кода.

- **Оптимизация временных характеристик.** В ходе тестирования производительности было установлено, что использование SequenceMatcher на уровне токенов, а не строк, позволило сократить время обработки с 1280 секунд до 60 секунд. Дополнительно был реализован механизм применения SequenceMatcher исключительно к области attention_area в обучающем наборе, что обеспечило дополнительное ускорение обработки данных.
- **Повышение точности сопоставления методов.** Для оценки качества идентификации схожих методов и их фрагментов были разработаны многочисленные тесты, включающие как синтетические, так и реальные примеры из промышленных проектов. По результатам тестирования было выявлено, что вместо cosine_distance более эффективным показателем схожести является F1-Score, в котором количество встреченных токенов заменяется бинарным признаком их присутствия в методе.
- **Выделение значимости отдельных токенов.** В процессе анализа было обнаружено, что различные типы токенов обладают неодинаковой значимостью. В связи с этим была разработана методика идентификации локальных и глобальных переменных, а также вызовов функций и методов классов. В результате экспериментов, включавших подбор гиперпараметров с использованием GridSearch, было установлено, что имена локальных переменных должны оцениваться в 1,5 раза выше, чем остальные токены. В настоящее время данный механизм реализован для языка Java.
- **Анализ области внимания.** Ведутся исследования по использованию в обучающей базе не полных методов, а только их фрагментов, относящихся к области внимания. Первые тесты не продемонстрировали значительного прироста качества, однако исследования продолжаются.
- **Оптимизация сравнения длинных методов (Diff-window).** Разработан подход для улучшенного сравнения длинных методов и выявления

схожих фрагментов кода. Префикс и суффикс определяются в зависимости от плотности идентичных последовательностей токенов в исходном коде. Для первичной фильтрации используется пороговое значение длины, позволяющее отсечь короткие соответствия, состоящие преимущественно из знаков пунктуации. Далее применяется алгоритм Чаувена (Chauvene Algorithm) для выявления выбросов и последовательного удаления лишних токенов в начале и конце последовательности. Дополнительно была внедрена проверка пересечения сохранённой области внимания с ref_diff_area, что предотвращает случаи, когда релевантный фрагмент кода обнаружен, но комментарий был оставлен к другому участку исходного кода.

Разработанные и протестированные подходы позволили значительно повысить качество идентификации схожих методов, а также оптимизировать временные характеристики поиска. В таблице 3.1 представлены результаты собранных вручную тестов, состоящих из троек участков исходного кода: референса (нового, пришедшего на ревью кода) и двух кандидатов (находящихся в обучающем наборе). Цель теста — проверить, что алгоритм поиска похожего кода выберет действительно более похожего кандидата.

	Пройдено	Провалено
Исходный	41	42
+ бинаризация	53	30
+ F1-score	70	13
+ веса токенов	76	7

Таблица 3.1: Результаты тестирования после внесения улучшений

Для оценки эффективности разработанного инструмента была использована методология, описанная в работе CommentFinder. В рамках данной методики для каждого примера из отложенной выборки производится поиск наиболее соответствующего участка кода, после чего проводится сравнение предложенного и реального комментариев с использованием двух метрик: PerfectPrediction и BLEU-4[43]. Анализ проводится на нескольких уровнях: среди Top-N (N = 1, 3, 5, 10) наиболее схожих фрагментов кода выбирается тот, чей комментарий демонстрирует наибольшее значение BLEU, после чего вычисляется среднее значение данной метрики по всей тестовой выборке.

В исследовании CommentFinder представлены значения метрики BLEU-4, полученные в ходе экспериментов: Top-1 — 0.124, Top-3 — 0.183, Top-5 — 0.209, Top-10 — 0.238. Однако анализ показал, что данные показатели были получены при расчете BLEU на символьном уровне, в то время как документация метода предполагает его применение к словам или токенам. При использовании BLEU в соответствии с официальной спецификацией результаты оказываются значительно ниже. В таблице 3.2 представлены результаты валидации инструмента CommentFinder и дополненного инструмента с включением всех описанных выше доработок, отмеченное как MLCF. Валидация проводилась с помощью описанной выше процедуры на наборах данных собранных краулером из различных источников и сгруппированных по языкам программираовния.

	MLCF Total	CommentFinder	Random	Random/CF	Random/MLCF
Regression test cBLEU					
Top1	0.0809	0.083	0.0755	91%	93%
Top3	0.1364	0.14	0.1263	90%	93%
Top5	0.1608	0.164	0.1501	92%	93%
Top10	0.1924	0.195	0.1781	91%	93%
Regression test wBLEU					
Top1	0.0043	0.0045	0.0034	76%	79%
Top3	0.0100	0.0098	0.0076	78%	76%
Top5	0.0139	0.0132	0.0106	80%	76%
Top10	0.0200	0.0189	0.0159	84%	79%

Таблица 3.2: Результаты регрессионного тестирования cBLEU и wBLEU

Анализ продемонстрировал, что по метрике cBLEU (символьное представление) модель CommentFinder незначительно превосходит MLCF. Однако при переходе к метрике wBLEU (основанной на словах) модель MLCF демонстрирует более высокие показатели. В качестве контрольного эксперимента была проведена серия тестов, в рамках которых примеры из обучающей выборки отбирались случайным образом. Результаты данного эксперимента отражены в колонке Random. По метрике cBLEU случайный выбор фрагментов демонстрирует на 10% худшие результаты по сравнению с CommentFinder. В случае wBLEU разница более значительна: качество случайного выбора оказывается

на 16-24% ниже, чем у CommentFinder, и на 21-24% хуже по сравнению с MLCF. Таким образом, можно заключить, что метрика wBLEU является более репрезентативной для оценки качества модели в рамках данной методологии.

По итогу детального анализа CommentFinder и попыток улучшить описанный алгоритм, а также анализируя полученные результаты можно сделать вывод о несостоятельности предложенного подхода и довольно низких результатах для применения подобного инструмента с целью автоматического ревью.

3.1.3. Анализ существующих генераторов рецензий к изменениям кода на основе языковых моделей

В качестве примеров работ по генерации комментариев по изменениям кода на основе больших языковых моделей рассматривались программные комплексы AUGER[14] и CodeReviewer [15]. Оба подхода применяют дообучение языковых моделей с архитектурой T5. Различие составляют задачи дообучения, наборы данных и предварительная подготовка входных данных. Рассмотрим детальнее экспериментальную часть обоих работ.

AUGER. Экспериментальная установка для оценки AUGER была направлена на измерение его производительности в генерации комментариев к обзорам по сравнению с базовыми моделями. Набор данных состоял из 79 344 экземпляров обзоров, собранных из 11 известных открытых репозиториев Java-кода, таких как Apache Kafka, Elasticsearch и Flink, обеспечивая разнообразную и представительную выборку коммитов. После предварительной обработки и фильтрации шумных или неактуальных данных, окончательный набор данных был разделен на 80% для обучения, 10% для валидации и 10% для тестирования. Каждый экземпляр состоял из тройки изменений кода, тегов строк, связанных с ними, и соответствующих комментариев. Базовые модели включали LSTM [44], CopyNET, CodeBERT [45] и вариант T5 [16], используемый для генерации комментариев к коду в предыдущих исследованиях. Метрики, такие как ROUGE-1, ROUGE-L и Perfect Prediction, использовались для оценки лексического перекрытия и точности точного совпадения между сгенерированными и фактическими комментариями. Эксперименты проводились на видеокартах NVIDIA RTX 3090, предварительная подготовка AUGER заняла 12 часов, а дополнительная настройка — 5 часов.

Результаты авторских экспериментов показали, что AUGER превосходит все базовые модели по нескольким метрикам. В терминах ROUGE-L AUGER достиг значения 22,97%, превысив лучшие результаты базовой модели на 37,38%. Модель также достигла скорости Perfect Prediction 4,32%, в 14 раз больше, чем у некоторых базовых моделей, несмотря на строгую требовательность к точным совпадениям [14]. Было показано, что использование AUGER маркировки строк коммита и кросс-предварительной подготовки значительно улучшает производительность, сосредотачиваясь на актуальных фрагментах кода и улучшая понимание контекста. Кроме того, AUGER генерировал комментарии к обзорам за 20 секунд в среднем на экземпляр, подчеркивая его эффективность по сравнению с ручными процессами обзора. Результаты подтвердили, что AUGER эффективно синтезирует значимые комментарии к обзорам, предлагая существенные улучшения точности и скорости по сравнению с традиционными и методами глубокого обучения.

CodeReviewer. Экспериментальная установка для CodeReviewer была разработана для оценки его эффективности в автоматизации задач по обзору кода и для сравнения его производительности с современными моделями. Набор данных, используемый для обучения и оценки, был создан из коллекции pull requests с GitHub, охватывающих девять популярных языков программирования, включая Java, Python, C++ и JavaScript. Набор данных был разделен на обучающий, проверочный и тестовый наборы на уровне проекта, чтобы предотвратить утечку данных. Оценивались три ключевые задачи: оценка качества изменения кода, генерация комментариев к обзору и уточнение кода. Базовые модели включали Transformer, T5 и CodeT5 [46], каждая из которых была настроена на одном и том же наборе данных для справедливого сравнения. Метрики оценки различались в зависимости от задачи, при этом точность, точность, отзыв и оценка F1 использовались для оценки качества, оценки BLEU [43] и человеческая оценка для генерации комментариев к обзору, а также оценки BLEU и точного соответствия (EM) для уточнения кода. Результаты показали, что CodeReviewer значительно превзошел все базовые модели по всем задачам. В задаче оценки качества изменения кода (англ. code change quality) CodeReviewer достиг оценки F1 71,53%, превзойдя CodeT5 на 7%. Для генерации комментариев к обзорам (англ. review generation) CodeReviewer получил

оценку BLEU 5,32, которая, хотя и является численно низкой из-за присущего комментариям к обзорам разнообразия, была подтверждена человеческой оценкой, где она набрала на 20% больше базовых показателей по информативности и релевантности 3,6 против 3,03 из маскимальных 5. В задаче уточнения кода (англ. code refinement) она достигла точного уровня соответствия (англ. exact match) 30,32%, значительно превзойдя CodeT5 (24,41%) [15]. Дальнейшие исследования абляции (англ. ablation study) подтвердили, что каждая задача предварительного обучения вносит вклад в общую производительность модели, при этом задачи прогнозирования тегов различий и шумоподавления играют решающую роль. Кроме того, многоязычное обучение улучшило производительность CodeReviewer на отдельных языках, доказав преимущества крупномасштабного кросс-языкового обучения.

Ручная валидация. AUGER поддерживает только язык Java, работает на уровне функций с внедренным тегом review_tag для обозначения начала местоположения обзора комментариев. Местоположения для этих тегов review_tag вычисляются геометрически. Он также выполняет специфическую предварительную обработку исходного кода, например, подслоговую токенизацию для имен методов, и предварительную обработку сообщений обзора с использованием NLTK, например, лемматизацию и т. д. Этот подход может быть более удобным, но он может быть более шумным.

С другой стороны, модели CodeReviewer работают на более тонком уровне. В качестве входных данных используются фрагменты изменений кода в унифицированном формате с внедренными специальными токенами add, delete и keep для обозначения добавленных, удаленных и сохраненных строк соответственно. Это позволяет CodeReviewer лучше интегрировать семантику изменений и связать ее с конкретными комментариями обзора. Также выполняется специальная предварительная обработка токенов исходного кода. CodeReviewer был обучен и оценен на нескольких разных языках программирования, включая Java, Python и Go. Этот подход может быть немного сложнее интегрировать, потому что требует правильного извлечения фрагментов.

При дальнейшей работе с авторскими данными и открытой реализаций было обнаружены проблемы реализации и идеи подхода:

- При ручном анализе нескольких сотен примеров из тестового набора предложенных данных было установлено, что полезные комментарии гене-

рируются очень редко. Положительным аспектом можно отметить качественный английский язык.

- Генерируются странные предложения к изменению исходного кода на тот же самый.
- Присутствуют галюцинации модели с генераций повторяющихся сообщений «this is bad. this is bad. this is bad ....».
- Генерируются несодержательные сообщения с вопросам «Why this change?» или чем-то похожим без конкретных предложений по улучшению исходного кода.
- в датасете присутствуют следы аугментации методом дублирования ситуаций с минимальными изменениями, а зачастую и без изменений кода и рецензии.
- «Грязные» комментарии по типу «Done», «understood», «thanks», «oh my god» также содержатся в наборе обучающей и тестовой выборок.
- комментарии, слишком специфичные для проекта или касающиеся слишком большого контекста, например, «it should be done in the ‘foo‘ method» или «move it to bar method from FooBuilder class»

Можно заключить, что исходный обучающий набор данных содержит значительное количество комментариев, не релевантных для задач автоматизированного ревью кода, по крайней мере, в рамках субъективной оценки. Подобные образцы «загрязняют» модель, способствуя усвоению ею бесполезных шаблонов и приводя к генерации комментариев, чрезмерно специфичных для конкретного проекта.

3.2. Структурирование и фильтрация комментариев

Как следует из предыдущих разделов, ключевой проблемой для всех рассматриваемых подходов является формирование достаточного объёма качественных данных. С одной стороны, для обучения языковых моделей или для подхода, основанного на выборе наиболее подходящих исторических комментариев, требуется максимально представительная выборка рецензий и связанных с ними фрагментов исходного кода. С другой стороны необходимо собирать мксимально релевантные данные, подходящие для разрабатываемоего проекта, чтобы учесть некоторые проектно-зависимые особенности. Также необходимо иметь

возможность фильтровать комментарии и собирать из общего числа только полезные, которые несут некоторую полезную и осмысленную нагрузку, помогают улучшить разрабатываемое ПО.

Для решения этой задачи был использован сборщик комментариев к ревью из GitHub Pull Requests и Gerrit Review System, реализованный ранее для задач классификации рецензий. Для последующего обучения пригодны не все собранные комментарии, а лишь те, которые непосредственно относятся к изменяемому коду и могут быть классифицированы как полезные. Наряду с анализом текста рецензии, логично применить фильтрацию по её «практической применимости» — то есть проверить, повлёк ли оставленный комментарий фактические изменения в коде. Таким образом можно избавиться от так называемых non-actionble комментариев вне зависимости от «хорошести» написанного текста.

Инструмент AUGER распространяется вместе с открытым набором данных, в котором при детальном анализе была обнаружена некачественная аугментация примеров, приведшая к значительному количеству дубликатов как кода, так и комментариев. Фильтрация таких примеров сократила размер обучающей выборки примерно в десять раз — с приблизительно 80 тысяч до 8 тысяч примеров. Затем были применены эвристические фильтры, предназначенные для идентификации бесполезных комментариев. Набор эвристик включал проверку на наличие фраз «yes we can/may», «ignore this», комментарии, состоящие исключительно из цифр или эмодзи, а также короткие рецензии, чтобы исключить неинформативные обсуждения, благодарности и выражения согласия. В результате объём выборки из набора AUGER сократился дополнительно на 5%. При применении тех же фильтров к неочищенному корпусу комментариев из открытых проектов доля отфильтрованного материала составила около 25%, что может свидетельствовать о проведённой авторами AUGER предварительной фильтрации.

Далее были разработаны 3 вида включающих (англ. inclusive) фильтра.

Первым была попытка аналогичного эвристического подхода с выбором шаблонов рецензий, которые хотелось бы увидеть в итоговом наборе. Этот подход не дал особых результатов, так как подбор подобных шаблонов сильно зависит от целевого языка программирования или проекта, например в проектах на C++ было значительное количество рецензий, затрагивающих тему идиом

программирования, тогда как в проектах на Python — нет.

Второй подход основывался на анализе ответов на рецензии. Было эмпирически замечено, что если ответ содержит прямое согласие с комментарием (например, «отличная идея», «согласен», «абсолютно»), то исходный комментарий с высокой вероятностью можно классифицировать как полезный. На основе этого наблюдения был реализован дополнительный эвристический механизм, помечающий комментарии, получившие одобрительные ответы.

Для генерации новых идей включающих фильтров было проведено тематическое моделирование рецензий. Для решения задачи была использована библиотека BERTopic[47] — это библиотека на Python для тематического моделирования, использующая современные методы обработки естественного языка и машинного обучения. В отличие от классических подходов, таких как Латентное размещение Дирихле (англ. LDA), она применяет трансформеры (BERT, RoBERTa, DistilBERT и др.) для векторизации текстов, что позволяет лучше выявлять смысловые связи. Основной процесс работы BERTopic включает несколько этапов:

- Векторизация текстов с использованием моделей-трансформеров.
- Снижение размерности полученных эмбеддингов с помощью алгоритма UMAP [48] для упрощения последующей кластеризации.
- Кластеризация методом HDBSCAN [49], который выделяет плотные группы точек данных и устойчив к выбросам.
- Интерпретация тематик кластеров с помощью модифицированного метода c-TF-IDF[1], выделяющего наиболее релевантные термины для каждого кластера.

Для обеспечения качества моделирования был проведён обширный цикл экспериментов с подбором гиперпараметров на каждом из этапов с последующей сравнительной оценкой интерпретируемости и согласованности извлечённых тем.

Используемому алгоритму удалось в значительной мере кластеризовать некоторые типы комментариев из общей массы (комментарии для пропущенной проверки на null, правила форматирования и т. д.), что облегчает исследование и определение групп и тематик частоиспользуемых комментариев. Однако

[1]https://maartengr.github.io/BERTopic/api/ctfidf.html

значительное количество комментариев остается неразмеченными на этапе кластеризации. Например: на небольшом наборе из 16 тыс. комментариев 7 тыс. комментариев помечены как выбросы.

Было проверено несколько гипотез для этого и выделено основную причину: в наборе данных есть шумный примеры и слабо разделяемые вложения (англ. embedding). В качестве преобразования в векторное предствление были опробованы алгоритмы Bag-of-words [50], sentence-bert [51] и эмбеддинги CodeT5 [46], но все они приводят к схожим результатам. Также были проведены эксперименты с разбиением комментариев на несколько предложений (когда это уместно) с использованием процедур определения границ предложений с помощью библиотеки Spacy, но это улучшило ситуацию незначительно. Такое поведение означает, что когда при уменьшении размерность векторных представлений, они не образуют легко различимых кластеров.

После проведения ручного анализа тематических кластеров и отдельных комментариев был сделан вывод о низкой эффективности подхода с ручными включающими фильтрами. Основная причина этого заключается в трудности формализации хорошо переиспользуемых шаблонов в виде правил из-за большого количества аномалий, не относящихся к выделенным темам (порядка тысяч экземпляров). Кроме того, эффективность включающих фильтров также сильно зависит от целевого языка программирования и специфики проекта. Например, для проекта на C++ могут быть разработаны фильтры, направленные на выявление типичных для этого языка проблем, таких как:

- rvalue/move/copy;
- ptr/pointer и ref/reference related (включая null, none);
- mutex/threading/locking/multiprocessing/parallel/sync/race;
- bit related operations;
- exception handling;
- overflow/underflow;
- casts;
- API;
- namespace;

При дальнейшем исследовании вопроса об отборе «полезных» комментариев был рассмотрен вариант разарботки интеллектуального фильтра на основе

техник few-shot learning. Это подход в машинном обучении, позволяющий моделям эффективно обучаться на ограниченном количестве примеров, обычно от одного до десяти примеров на класс. В отличие от традиционных методов, требующих больших объемов данных для достижения высокой производительности, few-shot learning стремится к обобщению на основе очень небольшого числа примеров. Это особенно ценно в ситуациях, где сбор данных затруднен или невозможен.[2] В такой постановке задачи у нас есть большой непомеченный набор комментариев, размеченная тестовая часть данных и достаточно небольшой размеченный фрагмент данных для обучения. Специализированные методы обучения на таком небольшом размеченном подмножестве позволяют достичь высокой точности классификации без значительных затрат на ручную разметку.

В ходе экспериментов были использованы 64 примера в качестве маленького размеченного набора для обучения и 464 примеров комментариев для разметки. Ручная доразметка проводилась батчами (англ. batch) по 16 примера. Базовой моделью была использована sentence-transformers_all-MiniLM-L6-v2 [51]. Для валидации использовались 500 размеченных вручную примеров. Итоговые метрики обучения и оценки результатов представлены в таблице 3.3.

Таблица 3.3: Результаты валидации алгоритма активного обучения

Метрика	**обучение**	**валидация**
Точность	0.981	0.741
Полнота	0.975	0.727
F1-macro	0.979	0.723

Далее были собраны комментарии с pull requests с топ-100 репозиториев для 3 языков программирования: Java, C/C++, Python. В итоге применения описанных исключающих и включающих эвристик и методов кластеризации были получены новые очищенные наборы данных. Количественные статистики представлены в таблице 3.4. Таким образом было получено порядка 100_000 новых потенциально чистых и качественных комментариев для дальнейшего обучения и валидации моделей. Во время анализа полученных данных были обнаружены некоторые недостатки, а именно: всё ещё присутствует значительное количе-

[2]https://www.dremio.com/wiki/few-shot-learning/

ство шумных комментариев (в духе «I am wondering if we should do this», «I don't think we should do it this way»), слишком длинные комментарии, которые могут охватывать несколько тем обсуждения и предложения.

Таблица 3.4: Результаты сбора и фильтрации рецензий

Метрика	**C++**	**Java**	**Python**
Общее #собранных комментариев	218676	83141	182690
#комментариев после эвристической фильтрации	51861	20410	52141
#комментариев после фильтрации эвристиками и МО	29773	11851	30405

3.3. Задача классификации рецензий

Для решения задачи классификации и в последствии выделения полезных комментариев требуется достаточно большой набор размеченных данных. В работах [52, 53, 54] проводились исследования на похожие темы по сбору комментариев и рецензий с целью их классификации.

Как уже было сказано выше, существует значительное количество реальных данных в открытом доступе с рецензиями разработчиков различного открытого ПО. Однако из всех данных более 75 % этих комментариев не относятся непосредственно к исходному коду, а описывают возможность развития и обслуживания программного обеспечения. С целью разделения всех комментариев на полезные и бесполезные, условно, конечно, имеет смысл научиться классифицировать рецензии на несколько групп, которые можно в дальнейшем иерархически объединять в нужные кластеры в зависимости от необходимости. Например выделять комментарии об указании на ошибки в коде, или о предложениях к рефакторингу, либо о глобальных архитектурных решениях. В этой секции описана проблема и решение задачи классификации рецензий к исходному коду. В исследовании были опробованы различные методы для проведения классификации и выявлен лучший среди них. Для этого были использованы различные методы для проведения классификации и выявили лучший. Для этого мы использовали различные виды векторных представлений (англ. embedding), среди них: CountVectorizer [55], TfidfVectorizer — из библиотеки Sklearn [56], а также более сложные: Word2Vec [57] и FastText [50]. Помимо

эмбеддингов были исследованы различные классификаторы, среди них: классические — Decision Tree [58], LogReg [59], Ridge [60], SVM [61]; ансамблевые — Random Forest[62], Gradient-Boosted Decision Trees [63], используемые также из библиотеки Sklearn [56]. Отдельно были рассмотрены разновидности модели-трансформера — BERT [64]. Стоит отметить, что на классификацию может влиять предварительная обработка текста — препроцессинг (англ. preprocessing). Особенности области, связанной с рецензированием кода, заключаются в том, что комментарии часто содержат англоязычные термины, а также могут включать в себя имена переменных, использующих различные стили написания, такие как CamelCase или snake_case. В этом контексте важным аспектом является токенизация, поскольку имена переменных могут состоять из нескольких слов. Комментарий может содержать аббревиатуры и опечатки, для которых тоже нужна отдельная обработка. Одним из ключевых отличий комментариев code review от обычных текстов является наличие фрагментов исходного кода, что также оказывает влияние на препроцессинг.

3.3.1. Существующие решения

Первым шагом исследования являлось изучение существующих работ на тему классификации рецензий исходного кода. При изучении нам встретились работы, включающие как бинарную, так и многоклассовую классификации. Также, были встречены различные методы препроцессинга, эмбеддинги и классификаторы.

Основополагающей статьей по данной теме является работа, опубликованная в 2015 году компанией Microsoft [65]. В своей работе авторы исследуют рецензии на полезность. Целью является показать, какие комментарии действительно эффективны и как их писать. Для этого была реализована модель бинарной классификации, которая разделяет комментарии на полезные — useful и не очень полезные — not useful. Сначала авторы статьи провели множество интервью с разработчиками, чтобы понять какие комментарии стоит считать полезными, а какие бесполезными. Входными параметрами для классификатора по итогам опроса являются: триггеры изменения — был ли изменен код после данной рецензии; классификатор, основанный на ключевых словах (был получен с помощью Naive Bayes algorithm); общее настроение комментария. Далее на основе этих признаков авторы строят дерево решений с помощью метода

машинного обучения — decision tree. Авторы используют стандартный препроцессинг, включающий в себя удаление пробелов, знаков препинания и цифр, преобразование слов в нижний регистр, приведение слов к начальной форме и удаление стоп-слов. Стоит отметить, что размер датасета для обучения модели — 844 комментария, а получившийся score: precision - 0.89, recall - 0.85, classification error - 0.17.

В работе [52] авторы разделяют рецензии на 11 классов, а размеченный датасет состоит из 5645 комментариев. Классификация в этой работе производится с помощью алгоритма TSHC, который включает в себя две стадии. В первой стадии проводится препроцессинг текста, среди особенностей которого можно отметить замену исходного кода и гиперссылок на слова-индикаторы, так как эти компоненты сами по себе не несут смысла для классификации. Также, авторы оставляют стоп-слова, так как они предполагают, что такие слова и аббревиатуры могут влиять на то, к какой категории относится короткий текст. Далее на этой стадии составляются вектора комментариев с помощью TF-IDF, а также проводится классификация методом машинного обучения — SVM [61]. Один комментарий к обзору часто затрагивает несколько тем. Следовательно, одной из целей TSHC является выполнение классификации по нескольким меткам. Для этого авторы создают текстовые классификаторы для каждой категории по стратегии «один против всех». Каждый классификатор применяется и прогнозирует возможность принадлежности этого комментария к соответствующей категории. Далее эти признаки передаются во второй этап, где в дополнение к уже выбранным признакам добавляются новые: длина комментария, его тип (детальный или глобальный), тип автора рецензии (является он автором проекта или нет) и множество других. Позже эти признаки используются для формирования нового вектора признаков. Наконец, каждый комментарий, обработанный TSHC, помечается как минимум одной меткой класса. В итоге исследования авторы проводят оценку выполненного алгоритма с помощью F-меры

Также была изучена работа 2020 года [66], где автор использовал датасет из 2000 комментариев, размеченных на 13 классов: readability, naming, documentation и другие. На первом этапе выполняется препроцессинг текста, при котором сохраняются стоп-слова, не применяется стемминг, а символы форматирования удаляются. Затем с помощью CountVectorizer [55] короткие тексты

преобразуются в векторное представление. Автор представляет нам две версии модели. Первая использует CountVectorizer и SGDClassifier с параметром loss = ‘hinge‘, а во второй еще добавляется TfidfTransformer . При тренировке модели нами был найден недостаток — автор тренировал свою модель на всей выборке, которая состояла из тренировочной и тестовой, и проверял на тестовой, что привело к завышенному результату — F1-macro = 0.94, приблизительно одинаковый в двух версиях, с небольшим преимуществом в первой. При обучении модели только на тренировочной части выборки F1-macro получился 0.49.

В опубликованной в 2022 году работе [53] авторы разработали инструмент на основе BERT — CommentBERT для классификации коротких рецензий, а также оценили этот метод, применив его к трем проектам с открытым исходным кодом: Mono Framework, Wireshark и Open Network. В их исследовании комментарий может принадлежать нескольким классам, они не ставят цель подобрать наилучший класс. Всего авторы разделили датасет на 12 классов. Из особенностей препроцессинга авторы применяли токенизатор WordPiece, который использует фиксированный словарь для обозначения слов в тексте, а неизвестное слово может быть разделено на подслова, которые присутствуют в словаре. Входные данные преобразуются путем прохождения через 12 уровней кодировщика BERT. Точность обученных авторами моделей варьируется от 0.84 до 0.99 (среднее значение = 0.94). Такие высокие опубликованные данные могут быть связаны с отсутствием фильтрации дубликатов при подготовке набора данных. При этом одинаковые сообщения могли использоваться как в обучающей, так и в тестовой частях.

В пятой работе [54] авторы используют датасет из 1828 комментариев, которые размечены на 5 категорий. Авторы ссылаются на работу, в которой этот же датасет размечен на 18 классов, но они решили объединить их в более высокие категории: functional, refactoring, documentation, discussion, false positive. Авторы опробовали два подхода для задачи токенизации и векторизации: использовать CodeBERT [45] для исходного кода и BERT [64] для рецензий; использовать CodeBERT и для исходного кода, и для рецензий. Было выяснено, что CodeBERT+CodeBERT подходит лучше. Далее каждый исходный код и комментарий передаются в свой уровень LSTM, который состоит из 50 ячеек. Затем объединенный вектор передается через полносвязный слой с функцией активации софтмакс (англ. softmax). Выходной вектор определяет итоговый

класс. Итоговое значение метрики model accuracy получилось 0.593.

В работе [67] авторы исследуют короткие рецензии кода на токсичность — комментарии, содержащие в себе неприемлемое поведение, из-за которых работоспособность и сплоченность команды ухудшается. Таким образом, данная классификация помогла бы сделать рецензии более эффективными и менее токсичными. Для такой классификации авторы комбинируют различные эмбеддинги и классификаторы. В качестве эмбеддингов используются: TfIdf, Word2vec, GloVe, FastText. В качестве классификаторов: классические и ансамблевые методы (Decision Tree, Logistic Regression, SVM, Random Forest, Gradient-Boosted Decision Trees) и глубокие нейронные сети (LSTM, BiLSTM, GRU, DPCNN). Также была рассмотрена модель трансформера BERT. Из особенностей препроцессинга авторы использовали удаление ссылок, сокращений, незначащих слов, лишних пробелов, специальных символов, повторений букв, замену символов в словах на буквы. Наиболее эффективной комбинацией оказалась модель BERT с дополнительной предварительной обработкой — удалением слов, которые никак не влияют на оценку токсичности комментария. Наилучшая комбинация обеспечивает F1-score = 0.889 и accuracy = 0.958.

3.3.2. Наборы данных

Для обучения моделей был собран собственный датасет, процесс создания которого включал несколько этапов. На первом этапе были объединены открытые наборы данных (около семи тысяч комментариев) из ранее рассмотренных решений. На втором этапе вручную было размечено 3200 комментариев.

В качестве базовых решений были доступны четыре размеченных датасета с которыми можно подробнее ознакомиться в таблице 3.5. Но в каждой работе своя классификация, для объединения надо было совместить классы из статей и на этом построить новую классификацию. Новые классы мы составляли так, чтобы охватить максимальное количество комментариев из статей в одном классе. Слишком большое количество классов в новой классификации негативно повлияло бы на итоговый результат. Поэтому мы решили также новые классы объединить в группы. Таким образом у нас получилась иерархическая классификация.

В первом доступном датасете также была иерархическая классификация. Классы разделились на 4 основные группы: Correctness — все о том, что тре-

Назавние статьи	Размер датасета	Количество классов	Иерархическая классификация
Automatic Classification of Review Comments in Pull-based Development Model [52]	5645	11	Да
Dissecting GitHub Code Reviews: A Text Classification Experiment [66]	2000	13	Нет
Automated Code Review Comment Classification to Improve Modern Code Reviews [53]	2672	12	Нет
Towards Automated Classification of Code Review Feedback to Support Analytics [54]	2828	17	Да

Таблица 3.5: Существующие наборы данных

буется исправить в коде, Decision — в комментариях этого класса выносится решение о дальнейшем статусе программы, Management — про управление рецензированием, Interaction — реакции, на комментарии, не несущие смысла для исправления кода. Классификация из статьи [52] распределилась в новую классификацию следующим образом. Некоторые классы из старой и новой классификации сопоставлялись один-в-один без изменений: Style, Questioning, Convention, Test (Testing), Functionality, Roadmap. Большая часть комментариев в датасете относится к обсуждениям и вынесению какого-либо решения или это ответ автора рецензируемого кода, такие комментарии мы определили в класс Response, в старой классификации это были классы — Response, Approval, Disagreeing, Encouragement, Diversion. Следующий датасет был опубликован в статье [66], описание классов здесь наиболее приближено к тому, что получилось у нас. Некоторые классы сопоставлялись один-в-один: Readability → Style; Naming → Naming; Testing → Testing; CodeOrganization / Refactoring → Refactoring; Error / ResourceHandling → Error; Documentation → Documentation; Other → Other. Другие были объединены: HighLevelClassSemantics / Design + HighLevelMethodSemantics / Design = Design; ControlStructures / ProgramFlow

+ Visibility / Access = Functionality; Concurrency + Efficiency / Optimization = Optimization.

Третий датасет, который нам доступен, был выложен со статьей [53], здесь, как и во второй статье представлена классификация по многим меткам (англ. multi-label classification). Из особенностей можно отметить отдельный класс для решения проблем с данными и еще один класс комментариев про совместимость кода с другими системами, в остальном классификация похожа на описанную выше. Здесь основную часть классов также удалось перевести один-в-один: code naming → Naming; config commit patch review → Convention; code design → Design; code api → Roadmap; code doc → Documentation; code io → Input/Output. Остальные классы объединились: code style + rule def = Style; code logic + code data = Functionality; config building installing + compatibility = Support.

В четвертом датасете из статьи [54] 17 классов были разделены на 5 групп: Functional — все, что связано с правильной работой и оптимизацией кода, Refactoring — рефакторинг, альтернативные решения и стиль написания кода, Documentation — комментарии о проблемах с документацией, Discussion — обсуждения решений проблем, False Positive — комментарии о проблеме, которая на самом деле не является таковой. Тут все проходило по тому же принципу: Visual Representation → Style; Naming Convention → Naming; Question → Questioning; False Positive → Response; Praise → Convention; Logical → Functionality; Timing → Optimization; Validation → Error; Documentation → Documentation; Support → Support; Alternate Output → InputOutput; Larger defect → Roadmap; Design discussion + Solution Approach + Interface = Design; Organization of Code + Resource = Refactoring.

Если в целом говорить о принципах объединения датасетов, мы старались объединить самые близкие классы из разных классификаций между собой при помощи их описаний из статей. Все что связано с удобочитаемостью (англ. readability), именованием переменных и в целом с тем, как будет выглядеть код, мы помещали в группу CodeStyle. Если речь шла о разработке, оптимизации, отладке, ошибках, дефектах, функциональности и структуре кода, то это относилось в раздел Development. Если же в комментариях люди обсуждали что-то, не указывая на конкретную проблему, либо выносили какой-либо вердикт это относилось к группе Discussion. В раздел User мы отнесли классы, в которых

как-либо затрагивается пользователь: поддержка использования кода, вывод ошибок на экран, документация. Таким образом мы объяли максимальное количество комментариев, остальные отнесли в группу Other. Итоговое описание групп и классов отображено в таблицах 3.8 и 3.9.

Второй этап — разметка комментариев вручную. После составления предварительных классов несколькими участниками команды было размечено 200 рецензиий в Label Studio. Каждый комментарий оценивался несколькими экспертами, после чего спорные случаи обсуждались, и принималось коллективное решение по выбору наиболее подходящего класса для каждого комментария. Этот процесс позволил оценить адекватность предложенной классификации, которая в итоге была признана удовлетворительной и включала 16 классов, объединенных в 5 групп. Далее каждый участник взял себе часть комментариев (300-500 штук) и разметил по согласованной классификации. В сумме получилось 3200, размеченных вручную, комментариев. Итоговый размер датасета составляет 10045 комментариев. На рисунке 1 изображена круговая диаграмма с распределением классов и групп в итоговом наборе данных. Датасет получился несбалансированный, мы учитывали это при выборе метрики для измерения качества полученных результатов. Созданный набор данных был опубликован в открытом доступе [68].

3.3.3. Исследование и реализация методов классификации

Для исследования задачи классификации комментариев были выбраны и сравнены различные модели, состоящие из эмбеддинга и классификатора. В качестве эмбеддингов были рассмотрены CountVectorizer [55], TfidfVectorizer — из библиотеки Sklearn [56], Word2Vec [57], FastText [50]. Для классификации рассматривались четыре классических классификатора — Decision Tree [57], LogReg [59], Ridge [60], SVM [61] и два ансамблевых — Random Forest [62], Gradient-Boosted Decision Trees [63], используемые также из библиотеки Sklearn [56]. Отдельно был рассмотрен трансформер BERT [64]. Подробнее о каждом классификаторе.

Decision Tree — алгоритм, в котором набор данных непрерывно разбивается в соответствии с определенным параметром. Он имеет две сущности, а именно узлы принятия решений и листья. Листья — это решения или конечные результаты. А узлы принятия решений — это места, где данные разбиваются на два

или более подузлов [57].

LogReg — основная идея логистической регрессии заключается в том, что пространство исходных значений может быть разделено линейной границей на соответствующие классы [59].

Ridge — это классификатор, основанный на гребневой регрессии. Он использует L2-регуляризацию, что помогает бороться с переобучением [60].

SVM — после отображения входных векторов в многомерное нелинейное пространство признаков SVM пытается определить наилучшую гиперплоскость для разбиения данных на n-классы, где n — количество возможных результатов [61].

Random Forest (RF) — это метод, основанный на ансамбле, объединяющим результаты, полученные с помощью нескольких деревьев решений [62].

Gradient-Boosted Decision Trees (GBT) — подобно RF, GBT также является ансамблевым методом, использующим деревья решений. Однако GBT создает деревья решений последовательно, так что каждое новое дерево может исправлять ошибки предыдущего, и объединяет результаты, используя подход «boosting» [63].

Перейдем к эмбеддингам, которые мы также использовали.

CountVectorizer — считает встречаемость слов в документе. Под документом может подразумеваться предложение, абзац, пост или комментарий [55].

TfidfVectorizer — усложненная версия CountVectorizer, заполняет вектора весами, которые показывают важность слов в тексте.

Word2Vec — алгоритм, основанный на двух моделях нейронной сети, Continuous Bag-of-Words (CBOW) и Skip-gram. CBOW предсказывает целевое слово на основе его контекста, в то время как skip-gram использует текущее слово для прогнозирования окружающего контекста. [57]

FastText — алгоритм, который строит вектор слова на основе векторов подстрок символов, содержащихся в словарном запасе модели. [50]

BERT — контекстно-зависимая модель. В отличие от контекстно-свободных встраиваний (например, word2vec, GloVe и FastText), где каждое слово имеет фиксированное представление независимо от контекста, в котором оно появляется, контекстуализированное встраивание создает представления слов, которые динамически информируются окружающими их словами. Архитектура включает в себя кодировщик, который принимает входные данные и генерирует

векторное представление более высокой размерности. Для задач классификации выходные данные кодировщиков используются для обучения. [64]

В дополнение нам надо было рассмотреть препроцессинг и токенизацию. Учитывая особенность нашей области исследования — рецензирование кода, здесь не подходит стандартные препроцессинг и токенизация, так как в комментариях code review встречаются отрывки кода и именованные сущности, которые надо обрабатывать отдельно.

На основе анализа существующих решений была выбрана метрика для проверки моделей — F1-macro. Данная метрика используется для оценки модели классификации при несбалансированных классах, она вычисляется как среднее значение F1-score по всем классам без учета их частоты в выборке.

Также, мы оценили, какая модель отработала лучше всего в соотношении количество затраченных ресурсов/качество результата.

3.3.4. Эксперименты и результаты

Все эксперименты были проведены на языке Python с использованием библиотек: Sklearn — библиотека с классическими моделями машинного обучения [56]; Transformers — библиотека, содержащая множество моделей глубокого обучения [69]; TensorFlow — библиотека от Google для решения задач построения и тренировки нейронной сети [70]; также FastText [50] и Word2Vec [57] для одноименных моделей. Измерения проводились на рабочей станции с операционной системой Ubuntu Server 20.04 LTS, процессором Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz, 32GB RAM, и графическим ускорителем NVIDIA TITAN Xp с 12GB памяти.

Было исследовано несколько препроцессингов. Во-первых, это добавление пробелов между скобочными выражениями, чтобы при токенизации в словарь добавлялось слово без приписанных к нему скобок. Во-вторых, это анонимизация, мы заменяли имена, url-ссылки и пути к файлам на токены PERSON, LINK и PATH соответственно. Это позволяло исключить влияние конкретных имен, ссылок и путей на предсказания модели, предотвращая ее смещение из-за часто встречающихся неключевых слов. В-третьих, это работа с кодом — разбиение на отдельные токены названия переменных CamelCase и snake_case. В-четвертых, применялось выделение именованных сущностей (NER) для идентификации и маркировки ключевых элементов текста. Наконец, проводилась очистка текста,

включающая удаление эмодзи и стоп-слов для улучшения качества данных и повышения точности модели.

Для каждого классификатора и набора данных был также выполнен подбор гиперпараметров. Для классических классификаторов мы использовали GridSearch из библиотеки Sklearn [56] — инструмент для автоматического подбора параметров для моделей машинного обучения. Для того чтобы добиться более высоких и стабильных метрик также применялась перекрестная проверка (англ. KFold Cross validation). Также при подборе гиперпараметров проводился мониторинг потребляемых ресурсов и переобучаемости модели. Разберем на примере комбинации FastText + Random Forest: мы хотели бы получить прирост F1-macro, для этого можно выставить параметры для Random Forest — n_estimators = 200, отвечающий за количество решающих деревьев и max_depth = 250, отвечающий за глубину деревьев, мы получили +1 пункт в F1-macro, но затратили в 4-5 раз больше ресурсов чем при значениях 50 и 140 для n_estimators и max_depth соответственно. Для FastText можно изменять количество эпох, но из-за желания получить лучшие результаты, мы можем прийти к переобучению модели, если выставим количество эпох больше 300.

С помощью метода GridSearch были подобраны оптимальные параметры для вышеперечисленных эмбеддингов и классификаторов. Результаты отображены в таблице 3.6, где Classifier — выбранная модель классификации, base — ее результат без какого-либо препроцессинга, pre-tokenize — результат модели с предварительной токенизацией (добавление пробелов в скобочных выражениях), anominize — анонимизация имен/ссылок/путей к файлам, split-idents — разделение на слова CamelCase и snake_case имен переменных, all — использование всех видов препроцессинга. Как видно из таблицы 3.6, различные виды препроцессинга практически не оказывали влияние на результаты данных моделей.

На рисунке 3.2 показана зависимость результатов от времени работы программы, время представлено логарифмической шкалой. Можно увидеть, что классификаторы разделились на три подгруппы, как и ожидалось: классические работают быстро, но их результаты самые низкие; FastText [50] и Word2Vec [57] имеют среднее время работы и немного лучшие результаты; и третья группа с лучшими результатами — разновидности BERT [64]. Также результаты отображены в таблице 3.7.

Таблица 3.6: Итоговая классификация

Classifier	**base**	**pre-tokenize**	**anominize**	**split-idents**	**all**
DecisionTree	0.435	0.436	0.438	0.438	0.436
LogReg	0.540	0.539	0.539	0.538	0.539
Ridge	0.506	0.506	0.503	0.491	0.490
SVM	0.277	0.276	0.298	0.285	0.306
RandomForest	0.461	0.462	0.467	0.461	0.467
GBT	0.457	0.456	0.461	0.452	0.457
fasttext	0.579				

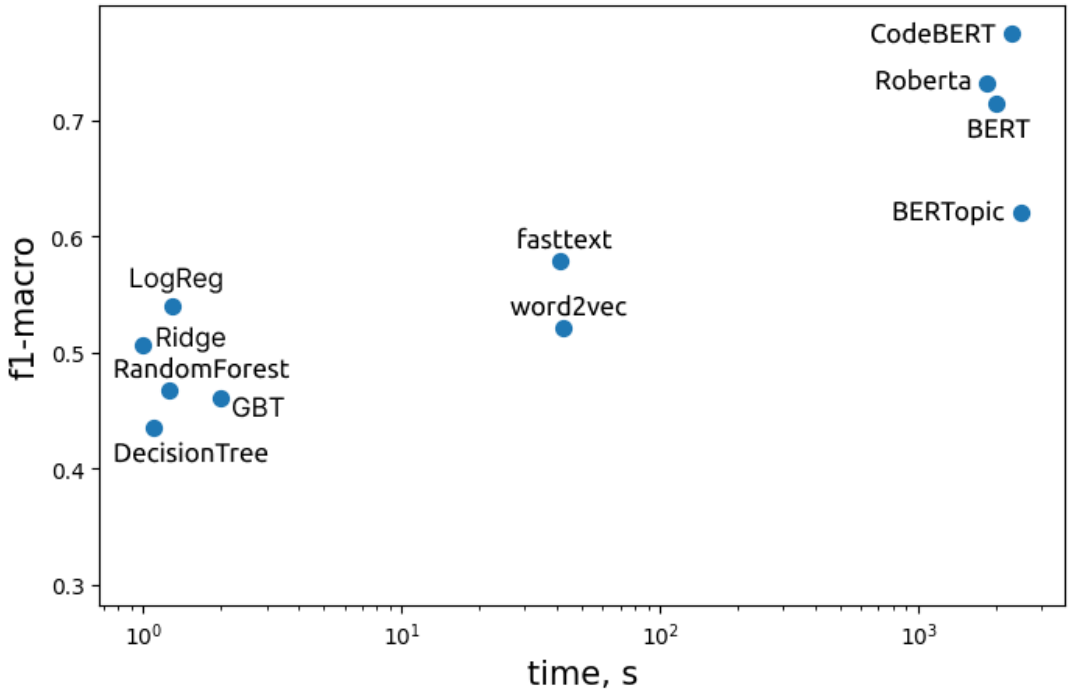

Рис. 3.2: Зависимость f1-macro от времени работы

Таблица 3.7: Результаты работы классических классификаторов

Classifier	**word2vec**	**fasttext**	**bertopic**	**BERT**	**Roberta**	**CodeBert**
F1-macro	0.521	0.579	0.621	0.714	0.732	0.775

Таблица 3.8: Описание групп и классов, часть 1

Группа	**Класс**	**Аннотация**
Code style	Style	Стиль программирования: удобочитаемость, компоновка кода, проблемы с отступами, и другие общие соглашения при написании исходного кода.
Code style	Style	Стиль программирования: удобочитаемость, компоновка кода, проблемы с отступами, и другие общие соглашения при написании исходного кода.
	Naming	Единый стиль и удобство именования переменных, методов, классов, каждое имя несет в себе смысл.
Discussion	Questioning	Вопросы к автору кода. Просит разъяснение кода либо примеры использования.
	Response	Назначает других рецензентов, пишет благодарности, соглашается с другими, дополняет мнения разработчика, устанавливает какое-либо решение.
	Convention	Обсуждение процесса разработки ПО.
	Testing	Запрашивает тесты для проверки функционала кода.

3.3.5. Результаты

В данной работе были исследованы методы классификации комментариев исходного кода программ. Реализованы модели, состоящие из комбинаций нескольких эмбеддингов и большого количества классификаторов. Отдельно было исследовано влияние препроцессингов на результаты различных моделей. Также был поставлен ряд экспериментов для подбора гиперпараметров. Исследования проводились на объединенном наборе данных, состоящем из около семи тысяч комментариев из работ других авторов и 3200 комментариев, размеченных вручную нашей командой, общим объемом в 10045 рецензий [68]. Для разметки была использована иерархическая классификация, комментарии бы-

Таблица 3.9: Описание групп и классов, часть 2

Группа	**Класс**	**Аннотация**
Development	Design	Архитектура и проектирование кода, контроль структуры программы.
	Refactoring	Логическая структура, создание объектов, логические ошибки.
	Functionality	Правильно ли реализована задача, выявляет дефекты работы кода.
	Roadmap	Как должна в дальнейшем развиваться программа.
	Optimization	Оптимизация кода, параллелизм, синхронизация, отладка программы.
	Error	Выявляет проблемы обработки исключений и ошибок.
User	Documentation	Проблемы с документацией либо с комментариями в исходном коде.
	Support	Совместимость с другими системами, системы поддержки.
	Input/Output	Ввод/вывод в графическом интерфейсе пользователя, проблемы с всплывающими окнами.
Other	Other	Комментарии не несущие смысловой нагрузки без контекста.

ли разделены на 16 классов, а затем объединены в 5 групп.

Модели, использующие классические векторные представления — CountVectorizer и TfidfVectorizer уступили в качестве моделям, использующим Word2Vec и FastText, различие наблюдалось приблизительно в 10 пунктов F1-macro. В то же время модели, основанные на трансформере BERT показали лучшие результаты. Таким образом, можно сделать вывод, что задача классификации комментариев исходного кода решаема и получены приемлемые результаты. Модель, основанная на BERT, получила результат равный 0.775 F1-macro при классификации на 5 групп.

3.4. Метрика схожести комментариев

Одной из ключевых проблем существующих решений является сложность оценки семантической похожести не только фрагментов кода, но и текстовых рецензий. Это порождает задачи типа «определить, какой из двух комментариев более схож с заданным эталонным» или «оценить степень семантической близости между двумя комментариями». Помимо очевидных технических сложностей, возникает проблема определения критериев похожести: в каком аспекте — синтаксическом, стилистическом или семантическом — следует оценивать схожесть текстовых рецензий?

```
1    1.
2    Ref: 'Nitpick: Please use nullptr instead of NULL in C++ code'
3    Cand_1: 'C++, use `nullptr`, not `NULL`'
4    Cand_2: 'We use raw C, so no nullptr, only NULL'
5
6    2.
7    Ref: 'I think return after here directly and can remove \"else.'
8    Cand_1: 'remove else after return'
9    Cand_2: 'Did you intend return after 1 iteration and remove layerId?'
10
11   3.
12   Ref: 'maybe we can extract this as a sub-function. (or function pointer to
     make a branching only once)'
13   Cand_1: 'this code can be extracted to a separate function to avoid branch'
14   Cand_2: 'maybe we can extract as a member function, not a static function.'
15
16   4.
17   Ref: 'Can we test that `self.connection is None` here instead ?'
18   Cand_1: 'Could we not look if `self.connection` is not None instead of
     having a counter ?'
19   Cand_2: 'it doesn't matter if `self.connection` is not None here because we
      assign it further'
```

Листинг 3.1: Примеры троек рецензий

В первом примере Листинга 3.1 кандидаты 1 и 2 семантически противоположны (использовать `nullptr` или использовать `NULL`), но имеют высокую текстовую схожесть по метрикам. Во втором — можно также отметить, что длина сообщения не является признаком схожести и может только мешать при вычислении метрики схожести.

Отдельного внимания заслуживает вопрос адекватности метрических оце-

нок. В частности, метрика BLEU, широко применяемая в существующих работах, демонстрирует существенные ограничения. Например, для пары комментариев:

- this doesn't look right, we expect a boolean value here [link]
- This function needs to return initiator_target_map to remove the zone. See example below: [link]

Показатель BLEU составляет 0.37, что превышает среднее значение по валидационной выборке в 20 раз, несмотря на полное семантическое различие комментариев. Данный артефакт обусловлен наличием протяженных общих фрагментов текста (URL-адресов). В отличие от BLEU, метрика косинусного сходства демонстрирует на данном примере значения, сопоставимые со средними по выборке, что подтверждает ее более адекватную чувствительность к семантическим аспектам.

Для выбора оптимальной метрики подобия был разработан тестовый набор, состоящий из 105 троек реальных рецензий к открытому программному обеспечению. Каждая тройка включала эталонный комментарий и два кандидата, один из которых априори считался более близким к эталону, а второй — менее близким. Данные были размечены вручную. В ходе эксперимента оценивалась эффективность ряда классических метрик (BLEU, ROUGE, METEOR, chrF), а также методов, основанных на векторных представлениях текста, генерируемых моделями глубокого обучения (с использованием bert_score и косинусного расстояния). Дополнительно исследовалось влияние процедур предобработки текста на качество оценки.

- BLEU [43] оценивает n-gram precision между сгенерированным текстом и референсными вариантами, включая штраф за излишнюю краткость. Метрика применялась в работах [42] и CodeReviewer [15]. В эксперименте представлены результаты с удалением стоп-слов и без.
- ROUGE-L F-measure — метрика, ориентированная на полноту (recall), изначально предложенная для задач суммаризации и машинного перевода. Используется реализация, основанная на longest common subsequence (LCS) на уровне предложений.
- chrF [71] — метрика для оценки машинного перевода, представляющая собой F-меру на основе символьных n-грамм.

- chrF++ [72] — вариант chrF, использующий n-граммы на уровне слов, демонстрирует более высокую корреляцию с человеческой оценкой.
- BERTScore [73] — используемая мера F1, отражающая семантическое сходство между эталонной и сгенерированной рецензией.

```

Origin: Is this function necessary? Unless you are going for very explicit and
   clear code, the same could be acheived by ‘int(x + 0.5)‘, maybe enclosed in
    a lambda.
Processed: [’function’, ’necessary’, ’unless’, ’go’, ’explicit’, ’clear’, ’code
   ’, ’could’, ’acheived’, ’int’, ’(’, ’x’, ’+’, ’0’, ’.’, ’5’, ’)’, ’enclose
   ’, ’lambda’]

Origin: this breaks down on marker/generator reuse
Processed: [’break’, ’marker’, ’/’, ’generator’, ’reuse’]

Origin: This is posted to the handler thread in the next CL, but should
   probably be done directly in this CL
Processed: [’post’, ’handler’, ’thread’, ’next’, ’cl’, ’probably’, ’do’, ’
   directly’, ’cl’]
```

Листинг 3.2: Примеры работы предварительной обработки рецензий

Префикс p_ в названии метрики означает, что входной текст был предварительно обработан с помощью функции ‘str_preproc(x)‘‘. Данная функция включает в себя токенизацию, приведение к нижнему регистру, стемминг, лемматизацию, а также удаление стоп-слов и части пунктуации, за исключением символов, находящихся внутри фрагментов кода. В листинге 3.2 представлены примеры работы реализованного метода.

Стоит отметить, что разработанный и реализованный препроцессинг почти всегда улучшает результаты метрики сравнения сообщений. Как видно из результатов, представленных в таблицах 3.10, 3.11, 3.12, наилучшим решением является использование BertScore с эмбедингами от модели microsoft/deberta-xlarge-mnli [74] и предварительным процессированием, при котором правильно определяется 83 пара из 105. Несколько хуже с результатом в 82 корректно определенных пар отработала метрика косинусного расстояния с моделью codet5p-110m-embedding [46] и также с предварительным процессированием сообщения. Среди классических решений лучшим оказался chrF++ с препроцессингом, онако он значительно отстает от метрик с использованием эмбеддингов. Однако microsoft/deberta-xlarge-mnli состоит из 750M параметров, тогда

Таблица 3.10: Результаты классических метрик

Classifier	**#правильных**
cBLEU	60
p_cBLEU	67
wBLEU	52
p_wBLEU	66
Rouge4	65
p_Rouge4	69
Meteor	68
chrF	64
p_chrF	70
chrF++	65
p_chrF++	**73**

Таблица 3.11: Результаты BertScore

Classifier	**#правильных**
roberta-large	73
p_roberta-large	74
t5-large	75
p_t5-large	78
roberta-large-mnli	80
p_roberta-large-mnli	77
microsoft/deberta-large-mnli	78
p_microsoft/deberta-large-mnli	77
microsoft/deberta-xlarge-mnli	72
p_microsoft/deberta-xlarge-mnli	**83**

как codet5p-110m-embedding — из 110M и к тому же BertScore работает сильно медленнее, чем расчет косинусного расстояния. В итоге время работы первого подхода в 6 раз больше, чем у второго, поэтому было принято решение в дальнейшем использовать ‘косинусного расстояния с моделью codet5p-110m-embedding и также с предварительным процессированием в качестве метрики

Таблица 3.12: Результаты косинусного расстояния

Classifier	**#правильных**
codet5p-110m-embedding	76
p_codet5p-110m-embedding	**82**
codet5p-220m-bimodal	73
p_codet5p-220m-bimodal	78
codet5p-770m	61
p_codet5p-770m	68

схожести сообщений.

3.5. Разработка системы на базе большой языковой модели

Как было описано выше, последние разработки в области автоматического рецензирования применяют большие языковые модели в качестве генератора комментариев. Наиболее яркими примерами подобных систем являются BitsAI-CR [19] от ByteDance и DeepCRCEval [17], описанные в первой главе. Однако существует ещё ряд работ, которые описывают различные подходы к решению возникающих трудностей на пути к итоговой цели.

В работе [75] подчеркивается критическая важность качества обучающих данных для эффективности моделей генерации комментариев к коду, предлагаются методы их очистки. Авторы фокусируются на устранении «шумных» (малоинформативных) комментариев из открытых наборов данных, используя LLM для фильтрации и сохранения только качественных рецензий. Показано, что модели, обученные на очищенных данных, демонстрируют улучшение (на 12-13% по метрике BLEU) в сходстве с эталонными человеческими комментариями. Авторы работы [76] предлагают подход DesiView. Он использует большие языковые модели, чтобы выделять рецензии, указывающие на реальные дефекты, классифицируя комментарии и обучая модель только на полезных данных. Такой подход позволяет генерировать более точные и целенаправленные комментарии, сфокусированные на существенных проблемах кода. Обе работы эмпирически доказывают, что предварительная обработка и строгий отбор тренировочных данных являются ключевыми факторами для повышения качества вывода генеративных систем в области автоматизированного рецензирования

кода.

3.5.1. Предлагаемые методы

Базовая инструкция. В работе [17] предложена инструкция к модели, генерирующей рецензии к исходному коду, названная LLM-Reviewer, которой в роли контекста исходного кода подается унифицированный формат измененного метода. В инструкции применяется метод, заключающийся в добавлении к инструкции небольшого числа примеров работы из «обучающего» набора, однако эти примеры подбираются случайным образом (few-shot learning). К недостаткам подхода можно отнести отсутствие структурированного вывода [14] работы модели, что осложняет понимание и выделение конкретных замечаний. Так, модель не указывает к какой конкретно строке относится предложение по исправлению, что не удобно пользователю, особенно, когда комментариев несколько, а целевой метод — достаточно длинный. Также в отсутствие стандартизированного вывода модель не всегда предлагает исправленный вариант кода. К минусам можно также отнести отсутствие контекста, хотя бы уровня класса или файла, что приводит к генерации замечаний такого вида «Ensure that all necessary modules (‘numpy‘, ‘matplotlib.pyplot‘, ‘sklearn.metrics‘) are imported at the beginning of the file. This is crucial for the code to run without errors.» или «The function ‘set_seed(seed)‘ is called but not defined within the snippet. Ensure that ‘set_seed(seed)‘ is defined elsewhere in the codebase or import it from an appropriate module.». Такие комментарии явно указывают, что ограничение контекста модели только одним методом повышает количество ошибочных рецензий.

Структурированный вывод. Структурированный вывод (structured output) — это технология, позволяющая большим языковым моделям генерировать выходные данные в строго заданном формате, таком как JSON, XML или таблицы, вместо произвольного текста. Такой метод обеспечивает предсказуемость, машинную читаемость и упрощает интеграцию с внешними системами (например, с базами данных или с прикладными системами) [77, 78]. Указание структуры в запросе с конкретным примером далеко не всегда гарантирует, что итоговый вывод модели будет таковым или в целом корректным форматом данных, например, словарем в языке Python. Применение структурированного

вывода значительно повышает надежность вывода модели в формате данных JSON или других сложных структур в ответе модели.

В библиотеке LangChain [79] представлены методы интеграции структурированного вывода, реализованные на базе библиотеки Pydantic [80]. Проверка вывода исключает пропущенные поля и неверные типы данных. Ограничение формата снижает риск генерации некорректной информации. Использование описанного подхода может привести и к нежелательным эффектам, например, к снижению качества генерации, так как строгие форматы могут ухудшать качество генерации, увеличивать время работы и обработки ответа модели. К тому же не все модели поддерживают и хорошо работают с режимом структурированного вывода. Для решаемой задачи определен и добавлен следующий структурированный вывод для модели — каждая рецензия должна состоять из комментария, предложенного исправленного кода, уровня важности (от 1 до 5) и номера строки, к которой относится комментарий.

В листинге 3.3 показан вариант инструкций для рецензирования кода.

```
CODE_REVIEW_PROMPT = """
Please review the following code:
```
{code_snippet}
```
Consider:
1. Code quality and adherence to best practices
2. Potential bugs or edge cases
3. Performance optimizations
4. Readability and maintainability
Suggest up to {comments_limit} improvements with reference to line numbers.
Rate the necessity of each suggetion from 1 (optional) to 5 (mandatory).
Give suggetion ONLY IF it is need to change source code and provide line
reference.
If suggestion is not actionable or almost generic - DO NOT give such suggestion
    .
NO NEED TO WRITE PRAISE. Be concise, give simple output.
Please write comments in markdown format.
Please reply in the following format: {schema}
"""
```

Листинг 3.3: Базовый вариант иструкций для автоматического рецензирования

Расширение контекста. Для улучшения восприятия моделью исходного кода и внесенных изменений было предложено расширить контекст с уровня конкретного метода до уровня файла. Так, в случае добавления в коммите нового файла следует подавать модели его целиком и рецензии будут писаться ко всему файлу, а не к отдельным методам. Если файл был изменен, то для полностью новых методов подавать в модель этот метод и все изменения в файле в формате unidiff. Для измененных методов предлагается подавать на вход модели все в соответствующем файле. Чтобы дать модели возможность точно указать место, к которому относится замечание, к целевым методам добавляется нумерация строк. Для изменений нумеруются все не удаленные строки, например как показано на листинге 3.4.

```
111: def sum( x: int, y: int):
-      print(f'{x} + {y} = {x + y}')
112: +     logging.info(f'{x} + {y} = {x + y}')
113:       return x + y
```

Листинг 3.4: Примеры нумерации строк измененного метода

Внедрение указанных усовершенствований привело к значительному сокращению абсолютно не релевантных комментариев. Тем не менее остаточные проблемы сохраняют свою актуальность, включая как критичные случаи, так и избыточные рекомендации, не оптимальные в данном контексте.

К числу наиболее распространённых групп относятся:

- Предложения по улучшению, уже реализованные в анализируемом коде, например, добавить работу с контекстным менеджером with open(...), при явном его наличии;
- Предложения добавить избыточные проверки, например, проверки наличия ключей в словарях там, где это гарантировано контекстом;
- Предложения вставить каждый вызов метода в операторы перехвата исключений (try-except), игнорируя принцип разумной достаточности;
- Предложения, противоречащие проектным практикам (например, замена метода ‘print’ на метод ‘logging’ в проектах, где такая библиотека логирования не используется);
- Ложное определение дублирования с последующими рекомендациями по «оптимизации повторяющихся вызовов»;

- Генерация комментариев с неявной или неочевидной мотивацией, затрудняющей понимание разработчиком цели предложенных изменений;
- Рекомендации по изменению устоявшихся практик проекта, не подлежащих рефакторингу по соображениям согласованности или усилий.

Уточнение инструкций. В работе BitsAI [19] приводится таксономия правил рецензирования кода, в который входят 4 большие группы: Security Vulnerability, Code Defect, Performance Issue и Maintanability and Readability.

В ходе обсуждения внутри команды данные группы были пересмотрены, учитывая исследовательский характер проектов, приводящий к более низким требованиям к надежности и безотказности написанного программного кода. В качестве целевого языка программирования был выбран язык Python. Итоговый список больших групп комментариев, которые хотелось бы видеть от системы автоматического рецензирования, следующий:

- **Programming Best Practices** — применение лучших практик программирования, оценка удобства поддержки, четкости наименований и соответствия PEP 8, с замечаниями по поводу отсутствия документации, только если это затрудняет понимание кода.
- **Bug Fixes** — выявление потенциальных ошибок или улучшение обработки исключений.
- **Optimization** — выявление избыточных или неэффективных вычислений, которые оказывают измеримое влияние.
- **Modern Python Features** — рекомендации по внедрению новых структур и особенностей языка Python.

Каждая из указанных категорий была реализована в качестве опционального поля в рамках структурированного формата вывода, генерируемого моделью. Дополнительно были интегрированы требования, направленные на предотвращение формирования ложных выводов в ситуациях, требующих межпроцедурного анализа или доступа к внешнему контексту, отсутствующему в текущих входных данных модели. Также были формализованы требования, предписывающие внутреннюю проверку предлагаемых изменений (например, целесообразность устранения предполагаемых избыточных вызовов) и фильтрацию гипотетических рекомендаций, не подкрепленных достаточными основаниями. Вся

совокупность данных описанных замечаний и ограничивающих предписаний была интегрирована в системный запрос модели.

Реализация ограничивающих механизмов привела к существенному снижению объема генерируемых замечаний и комментариев. Данное сокращение, при отсутствии способов обогащения контекста, закономерно сужает область применимости автоматизированного рецензента.

В листинге 3.5 представлена инструкция, в которой улучшены требования к рецензированию, четко определены типы недостатков кода, которые модель должны отслеживать.

```
CODE_REVIEW_PROMPT = """
Please review the following code:
```
{code_snippet}
```

Requirements:
- Provide up to {comments_limit} concise, actionable improvement suggestions with specific line number references.
- Suggestions must target actual issues in the code (e.g. architecture, readability, maintainability, bugs, or efficiency), and should not be generic or already addressed in the code.
- Only suggest changes that require modifications in the source code. Do not include suggestions for trivial docstring additions or style improvements when the code is already clear and concise.
- If a suggestion appears redundant (for example, 'use f-strings' when f-strings are already in use), omit it.
- Review and comment on any architectural issues that could affect the overall design or organization of the codebase.
- Do not provide compliments or generic praise.

Consider:
- **Programming Best Practices:**
    1. Evaluate readability, maintainability, clear naming, and adherence to PEP 8.
    2. Only note missing documentation if the lack actually impedes understanding.

- **Bug Fixes:**
    1. Identify potential bugs or exception handling improvements.

- **Optimization:**
    1. Look for redundant computations or inefficiencies that have a measurable impact.

```

```
- **Modern Python Features:**
    1. Recommend only if the feature enhances clarity or performance, ensuring
   the current code does not already implement it.

Return output in the following schema:
{schema}
"""
```

Листинг 3.5: Расширенный вариант иструкций для автоматического рецензирования

Добавление примеров в инструкцию. Для начала было решено зафиксировать 2 примера с хорошими рецензиями из существующей истории разработки. В первом примере указывалось на использование непонятных идентификаторов имен переменных, слишком общий тип исключений, отсутствие применения библиотечной функции tempfile для создания временного файла. Второй пример содержал советы по изменению цикла обработки графа, использованию конструкции 'if cond: return' для сокращения дальнейшей вложенности кода. Добавление примеров расширило спектр того, о чем пишет модель, но качество новых рецензий оставляло желать лучшего.

В системах генерации ответа с учетом дополнительно найденной релевантной информации (retrieval-augmented generation) [81] выполняется подбор нескольких примеров, наиболее похожих на заданный, по заранее определенной мере, например, косинусному расстоянию между векторными представлениями. Для решения этой задачи используем ранее разработанный и реализованный метод, описанный в 3.1.2. — доработанный вариант поискового алгоритма из [12].

Кроме того, важно иметь относительно чистый набор рецензий, которые будут подаваться в качестве примеров [75], такие как «Done» или «что это такое?» могут только ухудшить качество итоговой генерации. Для решения этой задачи использовался классификатор комментариев к исходному коду [5]. В данном исследовании была дообучена языковая модель на базе архитектуры CodeBERT, которая классифицирует рецензии на одну из 5 категорий: Development, Code Style, Discussion, User, Other. Для дальнейшей работы будут использованы только комментарии группы Development.

Дополнительная фильтрация. В работе [17] также описана инструкция для автоматической оценки качества сгенерированных рецензий. Выбранный авторам подход носит название «большая языковая модель как судья» — LLM-as-a-Judge, он представляет собой метод автоматической оценки текстовых ответов, сгенерированных большими языковыми моделями, с использованием другой языковой модели в качестве судьи. Этот подход позволяет имитировать человеческую оценку, фокусируясь на заданных критериях качества, таких как релевантность, точность, стиль или соответствие контексту [82]. В исследовании [83] авторы систематизировали метод, продемонстрировав, что LLM-судьи (например, GPT-4) достигают согласия с человеческими оценками в 85% случаев, что выше, чем уровень согласия между людьми (81%). Выделяют 3 подхода к оценке:

- Попарное сравнение (Pairwise Comparison): LLM выбирает лучший ответ из двух вариантов, например, для сравнения разных версий модели или запросов.
- Прямая оценка (Direct Scoring): LLM присваивает баллы или метки (например, 1-5) на основе критериев: корректность, отсутствие предвзятости, соответствие тону.
- Оценка с контекстом (Reference-Based): LLM проверяет соответствие ответа исходному документу, что полезно для выявления галлюцинаций.

Примером прямой оценки является такой запрос: «Оцените ответ чат-бота по шкале от 1 до 5, где 1 — полностью нерелевантный, 5 — идеально соответствует вопросу. Объяснения не требуются».

Почему это вообще работает, ведь модель не изменилась и какого-то дополнительного контекста, и новых знаний модели подать нельзя, иначе логичнее было бы его передать сразу в задачу генерации? Основным аргументом является то, что задача классификации (например, «содержит ли текст предвзятость?») требует меньших вычислительных ресурсов и проще для LLM, чем генерация связного ответа.

В работе [17] используются два варианта подхода «модель как судья», первый из них — это прямая оценка по 10 категориям, которые авторы выделили и для самой генерации рецензий. По каждой из этих категорий модели предлагается выставить оценку от 1 до 10 и сделать по ним итоговый вывод. Второй вариант оценщика, предложенный авторами этой работы, это модель, которая ранжирует несколько ответов, основываясь на все тех же 10 категориях.

Применение представленной инструкции для использования на своих данных показало очень низкий коэффициент согласия с ручной разметкой (коэффициент каппа Коэна меньше 0.01). Тем не менее даже такой запрос отсеивал больше плохих/бесполезных комментариев, чем хороших. Отсюда возникло желание улучшить инструкцию этого инструмента, но использовать его как дополнительную проверку предложений модели и фильтрации ошибочных рецензий. Для этого был проведен ряд экспериментов по улучшению запроса.

Первоначально, проведённый анализ признаков оценки привёл к сокращению исходного набора из 10 метрик до 5 ключевых категорий: Understanding (Понимание), Relevance (Релевантность), Actionability (Выполнимость), Accuracy (Правильность), Clarity (Ясность). Данное решение было обусловлено тем, что исключённые признаки показались не столь важными и скорее отсеивающими очень плохие по структуре комментарии, которые системы автоматического рецензирования на базе языковой модели пишут редко, при этом их наличие вносило свой положительный вклад в некачественные рецензии и негативно влияло на точность оценки общего качества.

Последующим шагом стал переход от 10-балльной оценочной шкалы к трехуровневой категоризации (poor, medium, good). Это изменение было мотивировано высокой субъективной сложностью для аннотаторов в дифференциации близких численных оценок (например, разграничение баллов 5, 6, 7 по метрике Actionability) и эмпирически подтверждённым превосходством LLM в классификации текстовых меток над числовыми прогнозами. Как демонстрируется в работе [83], использование семантически прозрачных текстовых дескрипторов повышает точность классификации LLM на 15-20%.

Далее, в ходе итеративного процесса валидации (включающего анализ ошибок модели на репрезентативных примерах) были сформулированы 5 групп нормативных требований к качеству комментариев и уточнены правила агрегации частных оценок в итоговую категорию. Наконец, для обеспечения контекстного понимания, в системный запрос были интегрированы 3 примера (фрагменты кода с сопутствующими рецензиями), сопровождаемые развёрнутыми пояснениями применения установленных правил оценки к данным случаям. Это обеспечило явную демонстрацию ожидаемого формата и логики оценки модели.

Согласованность между моделью с итоговым запросом и ручной разметкой по метрике каппа Коэна достигла 0.14, что всё ещё является недостаточным для

использования подхода на практике, но значительно выше первоначального. Основной его пользой для практического применения является пост-фильтрация рецензий. Для каждого комментария моделью была выставлена оценка: ошибочные, средние и хорошие (wrong, medium, good). Далее возможны два варианта работы: мягкая — когда убираются только ошибочные комментарии и, соответственно, остаются средние и хорошие; жесткая — когда остаются только хорошие комментарии, а все остальные отфильтровываются. Все использованные и разработанные инструкции опубликованы и доступны по в репозитории [84].

Разработанные в главе 2 детекторы антипаттернов исходного кода органично встраиваются в инструмент рецензирования, поскольку все необходимые для их работы входные данные уже используются алгоритмом рецензирования. Выходные данные детекторов, а именно — выявленные паттерны нарушений с указанием их типа и местоположения в коде, поступают в языковую модель в виде структурированного контекста. Это позволяет LLM генерировать а конкретные, предметные комментарии, напрямую связанные с обнаруженными проблемами. Модель получает возможность обосновывать свои замечания, ссылаясь на конкретные антипаттерны, и предлагать более точные и релевантные способы их устранения, что значительно повышает практическую ценность и действенность автоматически сгенерированной рецензии. Таким образом, интеграция преобразует сырые результаты анализа в экспертные заключения на естественном языке, усиливая как информативность выводов системы, так и их восприятие разработчиком. Общая архитектура разработанного программного комплекса автоматического рецензирования исходного кода, интегрирующего работу детекторов антипаттернов представлена на рисунке 3.3.

3.6. Дообучение языковой модели

Для адаптации базовой языковой модели к задаче генерации рецензий кода был проведен сравнительный анализ методов дообучения на данных откликов реальных пользователей на сгенерированные рецензии.

Были рассмотрены следующие основные подходы к дообучению языковых моделей:

- **Контролируемое дообучение (SFT, Supervised Fine-Tuning)**: Классический подход, предполагающий прямое дообучение модели на раз-

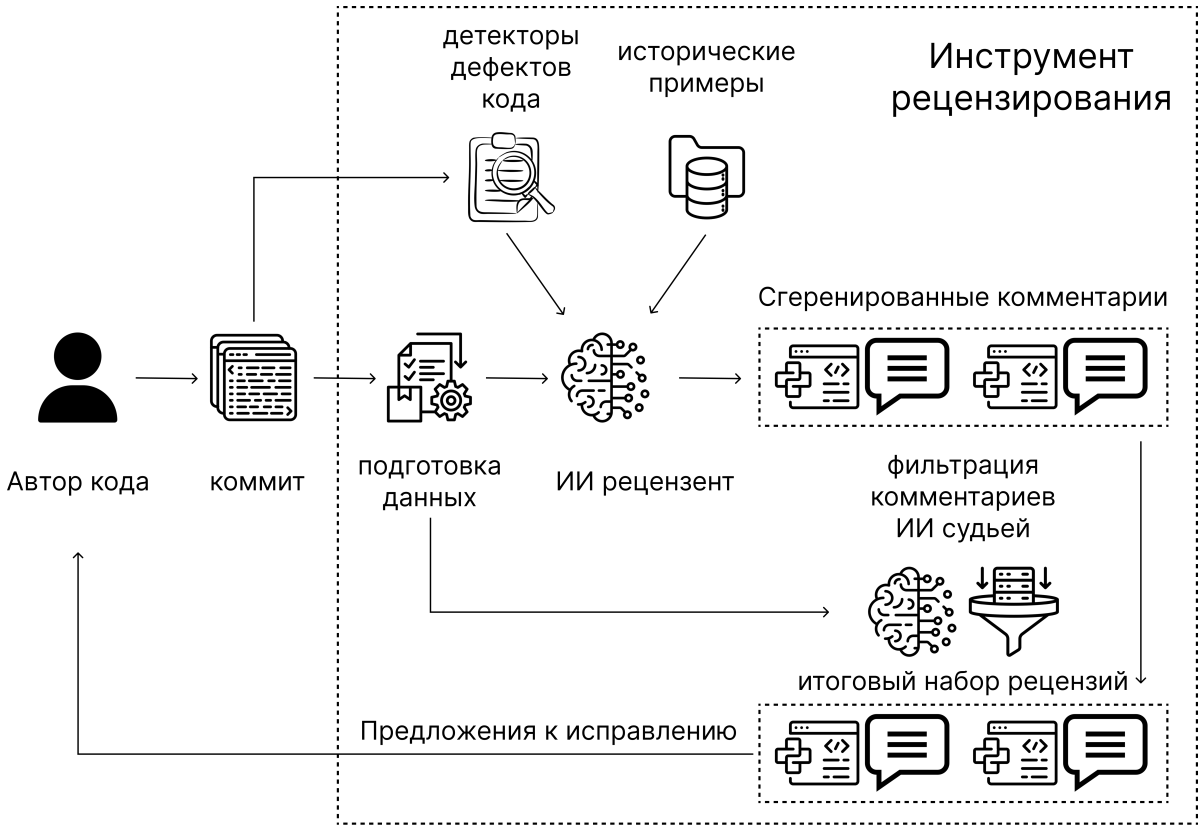

Рис. 3.3: Схема работы системы автоматического рецензирования

меченных данных вида «вход-выход». Метод минимизирует перекрестную энтропию между предсказаниями модели и эталонными ответами:

$$\mathcal{L}_{\text{SFT}}(\Theta) = -\Sigma_{t=1}^{T} \left[\log \pi_\theta(y_t|y_{<t})\right]$$

где y_t — это целевой токен в момент времени t и модель обучается предсказывать следующий токен по предыдущим. Хотя SFT демонстрирует хорошие результаты на больших наборах данных, при размере выборки менее 1000 примеров часто наблюдается переобучение и неудовлетворительная генерализация, что делает данный метод неподходящим для наших условий.

- **Обучение с подкреплением на основе человеческих предпочтений (RLHF)**[85]: Трехэтапный метод, ставший стандартом для выравнивания больших языковых моделей. Процесс включает: (1) начальное SFT-дообучение, (2) обучение модели вознаграждения на парных сравнениях, (3) оптимизацию политики модели с помощью алгоритма PPO. Несмотря на доказанную эффективность, метод требует значительных вычисли-

тельных ресурсов и наличия парных сравнений вида (y_w, y_l) для каждого промпта x, где y_w — выигравший, а y_l — проигравший ответ.

- **Прямая оптимизация предпочтений (DPO, Direct Preference Optimization)**[86]: Инновационный подход, позволяющий избежать сложного этапа RL-оптимизации. DPO непосредственно оптимизирует политику модели, используя аналитическое соответствие между функциями вознаграждения и оптимальными политиками. Функция потерь DPO имеет вид:

$$\mathcal{L}_{\text{DPO}} = -\mathbb{E}_{(x,y_w,y_l)\sim\mathcal{D}}\left[\log\sigma\left(\beta\log\frac{\pi_\theta(y_w|x)}{\pi_{\text{ref}}(y_w|x)} - \beta\log\frac{\pi_\theta(y_l|x)}{\pi_{\text{ref}}(y_l|x)}\right)\right]$$

где π_{ref} — эталонная модель, β — гиперпараметр температуры. Метод эффективен, но по-прежнему требует парных сравнений для каждого промпта.

- **Оптимизация Канемана-Тверского (KTO, Kahneman-Tversky Optimization)**[87]: Метод, основанный на поведенческой экономике, который использует непарные данные с бинарными метками. В отличие от DPO и RLHF, KTO не требует парных сравнений, а работает с бинарными оценками «желательно»/«нежелательно» для каждого отдельного ответа. Это делает метод идеально подходящим для нашей ситуации, где каждый пример рецензии имеет независимую оценку качества.

Учитывая исходно тернарный характер разметки (применимые/средние/плохие отклики) и ограниченный объем данных, в качестве основного метода был выбран KTO. Для приведения разметки к бинарному виду, необходимому для KTO, применимые и средние оценки были объединены в категорию «желательных» ответов, в то время как плохие отклики составили категорию «нежелательных». Такой подход демонстрирует устойчивую работу на небольших выборках и не требует парных сравнений, в отличие от DPO и RLHF.

3.6.1. Метод KTO

Метод KTO оптимизирует политику модели π_θ таким образом, чтобы увеличить вероятность генерации «желаемых» (desirable) ответов и уменьшить вероятность «нежелаемых» (undesirable). Функция потерь KTO формализует эту

интуицию через призму теории перспектив Канемана-Тверского, которая описывает как люди на самом деле принимают решения в условиях неопределенности.

Функция потерь KTO определяется следующим образом:

$$\mathcal{L}_{\text{KTO}}(\theta) = \mathbb{E}_{x,y\sim\mathcal{D}}\left[\lambda_y - v(x,y)\right]$$

где функция $v(x,y)$ задается как:

$$v(x,y) = \begin{cases} \sigma\lambda_D\left(\beta\left(\log\frac{\pi_\theta(y|x)}{\pi_{\text{ref}}(y|x)} - \mathbb{E}_{x\sim\mathcal{D}}\left[\log\frac{\pi_\theta(y|x)}{\pi_{\text{ref}}(y|x)}\right]\right)\right), & \text{если } y\sim\mathcal{D}_x^D \\ \sigma\lambda_U\left(\beta\left(\mathbb{E}_{x\sim\mathcal{D}}\left[\log\frac{\pi_\theta(y|x)}{\pi_{\text{ref}}(y|x)}\right] - \log\frac{\pi_\theta(y|x)}{\pi_{\text{ref}}(y|x)}\right)\right), & \text{если } y\sim\mathcal{D}_x^U \end{cases}$$

Здесь:

- π_{ref} — исходная модель до дообучения
- $\mathcal{D}_x^D$ и $\mathcal{D}_x^U$ — распределения желательных и нежелательных ответов
- β — параметр, регулирующий отклонение от эталонной модели
- λ_D, λ_U — весовые коэффициенты для желательных и нежелательных примеров
- σ — сигмоидная функция активации

3.6.2. Экспериментальные наблюдения

В ходе экспериментов было обнаружено критическое влияние баланса классов в обучающей выборке на качество дообученной модели. При использовании исходного соотношения положительных и отрицательных примеров наблюдалась значительная деградация качества генерации — модель генерировала менее содержательные рецензии.

Наилучшие результаты были достигнуты при сбалансированном наборе данных, где количество положительных и отрицательных примеров было приблизительно равным. Это ограничило итоговый объем обучающей выборки 300 примерами (150 положительных и 150 случайно выбранных отрицательных), однако позволило сохранить качество генерации при значительном улучшении релевантности рецензий.

Для сравнения также было проведено дообучение методом SFT на той же выборке, которое подтвердило теоретические ожидания: при ограниченном объеме данных SFT показало неудовлетворительные результаты с признаками переобучения и низкой способностью к обобщению на новые примеры.

3.7. Интеграция с системами непрерывной разработки

В данной секции рассматриваются практические аспекты интеграции систем автоматического рецензирования кода в современные CI/CD конвейеры. Исследуются подходы к реализации для дыух популярных платформ: GitHub и Gerrit. Особое внимание уделяется техническим деталям настройки и реализации автоматизированной проверки кода с использованием соответствующих API и инструментов.

Автоматическое рецензирование в GitHub с использованием GitHub Actions. GitHub Actions предоставляет гибкую платформу автоматизации, которую можно использовать для реализации автоматического рецензирования кода. Ключевыми особенностями являются событийные триггеры, доступ к содержимому кода, интеграция с проверками и безопасное хранение конфиденциальных данных. В качестве универсального формата, который далее можнет быть преобразован в необходимый формат для всех платформ предлагается использовать библиотеку reviewdog[3].

```
name: reviewdog
on: [pull_request]

jobs:
  lint:
    runs-on: ubuntu-latest
    permissions:
      pull-requests: write
    steps:
      - uses: actions/checkout@v3

      - name: Install reviewdog
        run: |
          curl -sfL https://raw.githubusercontent.com/reviewdog/reviewdog/master/install.sh \
            | sh -s -- -b /usr/local/bin
```

[3]https://github.com/reviewdog/reviewdog

```
16
17
18      - name: Run analyzer with reviewdog
19        run: |
20          http://localhost:8382/review?source=github&url=https://github.com/\
21            ${{ github.repository }}.git&change_id=\
22            ${{ github.event.pull_request.number }}&message_format=reviewdog \
23            | reviewdog -f=rdjson \
24                -name="my-analyzer" \
25                -reporter=github-pr-review \
26                -level=warning
27        env:
28          REVIEWDOG_GITHUB_API_TOKEN: ${{ secrets.GITHUB_TOKEN }}
```

Листинг 3.6: Пример workflow для автоматического рецензирования в GitHub

Автоматическое рецензирование в Gerrit с использованием robot comments. Gerrit использует иной workflow по сравнению с GitHub, основанный на обработке изменений (changesets), системе оценок и роботизированных комментариев (robot comments). Для автоматического рецензирования в Gerrit обычно используется Jenkins с плагином Gerrit Trigger.

```
1  import requests
2  import json
3
4  def post_robot_comment(change_id, revision_id, comment):
5      auth = requests.auth.HTTPDigestAuth('username', 'password')
6      url = f'https://gerrit-server.com/a/changes/{change_id}/revisions/{
       revision_id}/comments'
7
8      data = {
9          'message': comment,
10         'robot_id': 'my-static-analyzer',
11         'robot_run_id': 'run-123',
12         'url': 'https://ci-server.com/job/123/'
13     }
14
15     response = requests.post(url, auth=auth, json=data)
16     return response.status_code == 200
```

Листинг 3.7: Пример использования API Gerrit для отправки роботизированных комментариев

Интеграция систем автоматического рецензирования в CI/CD pipelines значительно улучшает качество кода и ускоряет процесс разработки. Каждая из

рассмотренных платформ — GitHub и Gerrit — предлагает уникальные возможности для реализации такого подхода, основанные на их архитектурных особенностях и экосистемах.

Техническая реализация варьируется от использования готовых инструментов в GitHub Actions до более низкоуровневого подхода с использованием REST API в Gerrit. Выбор конкретного решения зависит от технических требований проекта и используемой платформы управления версиями.

3.8. Выводы

Третья глава посвящена комплексному исследованию и разработке методов автоматического рецензирования исходного кода. В рамках проведенной работы был выполнен анализ и практическая апробация двух ключевых подходов: поиска готовых рецензий в исторических данных и генерации текстовых комментариев с использованием языковых моделей (AUGER, CodeReviewer).

В результате исследования выявлены ограничения существующих методов, включая низкое качество обучающих данных, присутствие неинформативных рецензий, а также низкое соответствие стандартных текстовых метрик (BLEU, ROUGE) для оценки смысловой близости комментариев человеческой разметке. Для решения этих проблем предложен и реализован ряд модификаций: оптимизация алгоритма поиска схожих участков кода, взвешенный учет типов токенов, разработка эвристических и машинно-обучаемых фильтров для очистки набора данных.

Значимым научно-практическим результатом является создание и публикация размеченного набора данных для классификации рецензий, а также разработка иерархической таксономии категорий комментариев. Сравнительный эксперимент показал превосходство моделей на архитектуре BERT (F1-macro = 0.775) для задачи классификации.

В рамках задачи оценки семантической схожести комментариев наилучшие результаты продемонстрировала мера, основанная на библиотеке BertScore, однако ввиду ее вычислительной сложности для практического применения рекомендован компромиссный вариант на основе эмбеддингов CodeT5.

Завершающим этапом стала разработка архитектуры системы автоматического рецензирования на основе Large Language Models с интеграцией детекторов антипаттернов. Предложены и протестированы методы улучшения качества

генерации: структурированный вывод, расширение контекста, few-shot learning с семантическим поиском примеров и пост-обработка с использованием подхода модели-судьи.

Глава 4
Экспериментальное исследование и оценка эффективности разработанных методов

4.1. Описание тестового окружения

Для оценки разработанных моделей автоматическим регрессионным тестированием был собран набор данных и разделен определенным образом на валидационную выборку и несколько видов обучающих. В качестве источника рецензий были использованы несколько открытых Gerrit репозиториев: opendev[1], openbmc[2], chromium[3], android[4]. Далее было произведено деление на обучающую и валидационную выборки путем отделения рецензий по времени: более старые ушли в обучающую, более новые — в валидационную. Также было разделение на проекты на два непересекающихся множества — один для обучения, второй для валидации. Таким образом было проведено исследование о полезности иметь исторический набоор данных для целевого проекта или достаточно иметь большой исторический набор рецензий из других открытых источников. Также с целью исследования эффективности разработанного классификатора рецензий на 5 групп, примеры в валидационную выборку попадали только после классификации с полученной отметкой Development, как целевой для рецензирования на уровне методов и файлов. Итоговая валидационная выборка имела 15000 рецензий к 3 языкам программирования: Python, Java, C++, по 5000 к каждому. Обучающая выборка составила 300000 примеров, по 100000 на каждый язык программирования. Далее была проведена классификация рецензий на 5 групп и отделен набор примеров с меткой Development. Так же были выделены выборки меньшего размера для оценки зависимости качества предсказания рецензий от размера обучающей выборки: 150000, 90000 и 45000 примеров.

В таблице 4.1 представлена более детальная информация о численных характеристиках полученных выборок.

В качестве целевых метрик выбраны классический BLEU4[43], использован-

[1]https://review.opendev.org/

[2]https://gerrit.openbmc.org/

[3]https://chromium-review.googlesource.com/

[4]https://android-review.googlesource.com/

	Общее число примеров	# примеров Development
Обучение 1	300000	179346
Обучение 2	150000	87717
Обучение 3	90000	58953
Обучение 4	45000	28491
Валидация	15000	15000

Таблица 4.1: Численные характеристики выборок

ный в большем количестве референсных статей и оптимальный вариант, выбранный в предыдущей главе исследования — метрика косинусного расстояния с моделью codet5p-110m-embedding [46] и также с предварительным процессированием сообщения.

Таблица 4.2 содержит информацию о результатах валидации реализованного инструмента на базе CommentFinder. Из полученных результатов можно сделать вывод о росте результатов при увеличении выборки с 0,2422 до 0,2786 (на 15%) на cosine_sim метрике для топ-1 и с 0,37 до 0,531 (на 43%) на метрике BLEU4 для топ-1. Для топ-10 BLEU4 вырос с 0,0156 до 0,0204 (на 30,7%), а cosine_sim — с 0,3898 до 0,4286 (на 10%).Также можно заметить, что почти во всех случаях метрики при обучении на отфильтрованной выборке показывают большие результаты, чем на общей, хотя и имеют меньшее количество данных. Немаловажным пунктом является время работы инструмента, если для MLCF eval занимает около 3 минут, то для подходов на базе AUGER и CodeReviewer — около 90 минут на то же количество примеров.

Таблицы 4.3, 4.4 содержат аналогичные результаты с результатами валидации адаптированных инструментов AUGER и CodeReviewer.

Был проведен численный эксперимент, направленный на эмпирическую оценку эффективности предложенной системы автоматического рецензирования. В качестве тестовых данных использовались 10 патчсетов с большим количестов рецензий из внутреннего репозитория Gerrit, содержащих совокупно 132 экспертных комментария от членов команды разработки. Аннотированные рецензии относились к 15 файлам из 4 проектов, репрезентативных для текущей деятельности команды. Указанные коммиты были направлены на вход большой языковой модели (архитектура которой описана в разделе 3.5) для генерации

	300k	300k dev	150k	150k dev	45k	45k dev
BLEU4-top1	0.0045	0.0531	0.004	0.0046	0.0039	0.0037
BLEU4-top3	0.0102	0.0111	0.0092	0.0099	0.0086	0.0082
BLEU4-top5	0.0142	0.0152	0.0125	0.0135	0.0115	0.0109
BLEU4-top10	0.0199	0.0204	0.0179	0.0184	0.0159	0.0156
cosine-sim-top1	0.2702	0.2786	0.2563	0.2623	0.2444	0.2422
cosine-sim-top3	0.3551	0.3587	0.3391	0.3444	0.3234	0.3212
cosine-sim-top5	0.3878	0.3898	0.3716	0.3757	0.3536	0.3521
cosine-sim-top10	0.4285	0.4286	0.4122	0.4142	0.3922	0.3898

Таблица 4.2: MLCF regression Python

	300k	300k dev	150k	150k dev	45k	45k dev
BLEU4-top1	0.0169	0.0173	0.0162	0.0166	0.0154	0.0154
BLEU4-top3	0.0206	0.0211	0.0200	0.0202	0.0191	0.0193
BLEU4-top5	0.0226	0.0235	0.0215	0.0219	0.0216	0.0215
BLEU4-top10	0.0247	0.0253	0.0239	0.0242	0.0228	0.0228
cosine-sim-top1	0.3919	0.3927	0.3694	0.3704	0.3412	0.3466
cosine-sim-top3	0.4395	0.4433	0.4197	0.4208	0.4029	0.4042
cosine-sim-top5	0.4612	0.4629	0.4401	0.4422	0.4217	0.4220
cosine-sim-top10	0.4793	0.4802	0.4566	0.4579	0.4312	0.4314

Таблица 4.3: AUGER regression Python

автоматических рецензий, которые впоследствии прошли ручную оценку релевантности и применимости.

Эксперимент проводился в двух конфигурациях: с включением в промпт требования предоставить исправленную версию кода (переменная scheme) и без него. Модели инкрементально ставились задачи анализа представленного кода по нескольким аспектам: соблюдение best practices, выявление багов и граничных случаев, оптимизация производительности, повышение читаемости и сопровождаемости.

Анализ результатов первой серии (с требованием исправления) выявил статистический перекос в распределении генерируемых комментариев. Из 199 сгенерированных рецензий 95 относились к категории best practices. Наиболее

	300k	300k dev	150k	150k dev	45k	45k dev
BLEU4-top1	0.0129	0.0135	0.0119	0.0124	0.0107	0.0109
BLEU4-top3	0.0176	0.0181	0.0167	0.0170	0.0155	0.0158
BLEU4-top5	0.0199	0.0203	0.0188	0.0191	0.0174	0.0178
BLEU4-top10	0.0212	0.0219	0.0204	0.0208	0.0196	0.0199
cosine-sim-top1	0.3642	0.3657	0.3431	0.3425	0.3212	0.3267
cosine-sim-top3	0.4194	0.4206	0.3952	0.3995	0.3764	0.3780
cosine-sim-top5	0.4444	0.4456	0.4207	0.4266	0.3993	0.3998
cosine-sim-top10	0.4633	0.4669	0.4354	0.4375	0.4096	0.4104

Таблица 4.4: CodeReviewer regression Python

частыми рекомендациями были предложения по добавлению комментариев и docstring к методам, что, хотя и является в целом позитивной практикой, в данном контексте часто оказывалось избыточным или нерелевантным. Также значительная часть best practices примеров говорила о необходимости контекстных менеджеров для работы с файлами «Use context manager for file operations to ensure proper handling of resources.», или о необходимости писать типы данных переменных и возвращаемого значения в методах «Use type hints for function parameters and return types.» Оба эти замечания в общем случае полезны, но в большинстве ситуаций на которые указала модель — уже и так применялся контекстный менеджер или были написаны type hints, таким образом рецензия оказывалась не actionable. По результатам верификации лишь 6 из 95 комментариев данной категории были признаны полностью или частично релевантными.

Во второй серии экспериментов (без требования генерации исправленного кода) модель предложила 158 рецензий, из которых 71 также относилась к best practices с аналогичными проблемами релевантности. Полезными были признаны только 7 из них.

Требование предоставить исправленный код не привело к ожидаемому повышению качества рецензий и не нивелировало склонность модели к генерации избыточных замечаний. С одной стороны такие комментарии, в которых предлагается исправить то, что уже и так есть можно отлавливать самостоятельно анализируя предлагаемый fix suggestion и сравнивая его с текущей версией участка кода на который указывает модель. Такое действие потребует подсчета

метрики схожести участков кода (расстояние Левенштейна, BLEU и прочие) либо метрик основанных на семантических моделях для попытки уловить различия и установки некоторых граничных значений метрик. С другой стороны — можно просто отсеять все замечаний, которые модель отнесла к теме best practices и предлагать их только по особому желанию пользователя (строки в таблице результатов с суффиксом «no bp»). И третим вариантом является исправление инструкций, в которых описать свое желание, чтобы модель не указывала слишком общие и неприменимые замечания.

Таблица 4.5 содержит сводную информацию о результатах ручной проверки результатов. Анализ представленных данных позволяет выявить существенное влияние формата инструкции на качество генерируемых моделью рецензий. Наибольшее количество абсолютно применимых комментариев (18) наблюдается при использовании Инструкции 1 с требованием предоставить исправление, однако общий процент релевантных (применимых и средних) комментариев в этом случае остается низким (19.6%). Исключение комментариев категории best practices (bp) закономерно снижает общее количество генераций, но при этом повышает долю релевантных предложений до 31.7-33.3%.

Наилучший результат по удельному весу релевантных комментариев (45.7%) демонстрирует Инструкция 2 с фиксом, что свидетельствует о её эффективности в контексте поставленной задачи. Таким образом, модификация инструкций и фильтрация заведомо низкокачественных категорий комментариев являются ключевыми факторами для повышения практической ценности автоматического рецензирования.

	Применимые	Средние	Плохие	Неплохие, %
Инструкция 1 с фиксом	**18**	**22**	160	19.6
Инструкция 1 без фикса	17	20	122	23.4
Инструкция 1 с фиксом без bp	16	18	70	31.7
Инструкция 1 без фикса без bp	14	16	58	33.3
Инструкция 2 с фиксом	16	16	38	**45.7**
Инструкция 2 без фикса	15	18	57	36.6

Таблица 4.5: Ручная проверка простого кода

4.1.1. Сравнение качества комментариев, полученных с исторических данных и с использованием LLM

В следующей секции будут представлены примеры работы различных подходов.

```
def test_update_snapshot_status(self):
    volume_id = self.create_volume(client=self.vol_other_client)
    snapshot_id = self.create_snapshot(
        client=self.snap_other_client, volume_id=volume_id
    )

    <review_tag>self.os_project_admin.snapshots_client_latest.
   reset_snapshot_status(
        snapshot_id=snapshot_id, status='creating'
    )
    waiters.wait_for_volume_resource_status(
        client=self.os_project_admin.snapshots_client_latest,
        resource_id=snapshot_id, status='creating'
    )<review_tag>
    self.do_request(
        'update_snapshot_status', expected_status=202,
        snapshot_id=snapshot_id, status='error', progress='80%'
    )
```

Листинг 4.1: Участок кода для рецензирования

Рассмотрим метод на языке Python, представленный в листинге 4.1. Метками <review_tag> отмечены границы кода к которому был оставлен оригнальный комментарий: «Why do you need to reset the snapshot status by admin?». Модель на основе подхода AUGER предложила следующие 5 предложений отсортированных по степени уверенности.

- «-1: We don't need to reset the snapshot status in the waiters.»
- «-1: We don't need to reset the snapshot status.»
- «-1: We don't need to reset the snapshot status here.»
- «I don't think we need to reset the snapshot status here.»
- «-1: We don't need to reset the snapshot status in the waiters. We should also check that the status is "creating".»

Можно заметить, что все предложенные комментарии схожи между собой и достаточно похожи по смыслу с оригинальным. Хотя пересечения обучающей и тестовой выборок нет, но модель видимо корректно уловила суть замечаний

в проекте. Также рассмотрим рецензии предложенные инструментом MLCF. Кроме самих рецензий инструмент позволяет посмотреть на исторический пример, который он посчитал похожим на целевой.

- «-1: this should be done at the resource setup phase», `https://review.opendev.org/#/c/openstack/cinder/+/463906/12/cinder/tests/tempest/api/volume/test_volume_revert.py@60`,
- «To be removed until Blazar supports the removal of host extra capabilities / PATCH update { "op": "remove "path": "extracapability_key"} is implemented?» `https://review.opendev.org/#/c/openstack/heat/+/534576/14/heat/engine/resources/openstack/blazar/host.py@139`,
- «Probably it would be good to move this to the client where you get executions based on job_id.» `https://review.opendev.org/#/c/openstack/heat/+/231080/9/heat/engine/resources/openstack/sahara/job.py@229`,
- «I do not prefer to have assert for status code in cleanup code because its better to have concrete exception to find the failure easily.» `https://review.opendev.org/#/c/openstack/tempest/+/91747/14/tempest/api/compute/keypairs/test_keypairs.py@34`,
- «Shouldn't we wait here for resource deletion as response status is 202. we can use the in-build method wait_for_resource_deletion() by overriding is_resource_deleted() method.» `https://review.opendev.org/#/c/openstack/tempest/+/91747/4/tempest/api/compute/keypairs/test_keypairs.py@32`,

При детальном анализе результатов было установлено, что предложенные автоматически сгенерированные комментарии демонстрируют низкий уровень контекстной релевантности и практической применимости. Несмотря на наличие определенного семантического соответствия между исходными участками кода, рецензии, ассоциированные с ними в исторических данных, не могут быть эффективно адаптированы к целевым программным сущностям. Существенным ограничением применяемого алгоритма является отсутствие механизма семантической адаптации найденных рецензий к специфике целевого кода, что приводит к их полной неприменимости даже при попытке содержательной интерпретации.

Далее рассмотрим предлложения от LLM с zero-shot промптом:

- Consider adding a timeout to the ‘wait_for_volume_resource_status‘ function to prevent indefinite waiting in case of unexpected issues.
- Ensure that ‘self.create_volume‘ and ‘self.create_snapshot‘ handle exceptions and return valid IDs. Currently, if these methods fail, the subsequent operations will likely raise errors.
- Consider caching the ‘snapshots_client_latest‘ if it is used frequently to avoid repeated attribute access.
- The test does not handle potential exceptions that could be raised by ‘create_volume‘, ‘create_snapshot‘, ‘reset_snapshot_status‘, or ‘wait_for_volume_resource_status‘. This could lead to test failures that are not due to the functionality being tested.

Сгенерированные комментарии носят общий характер: рекомендации по добавлению дополнительных проверок, внедрению кэширования результатов или реализации обработки ошибок. Данные предложения, хотя и являются корректными с точки зрения общепринятых практик разработки, не учитывают конкретный контекст программного проекта. Важно отметить, что модель формирует подобные рекомендации исключительно на основе анализа локального участка кода, без учета архитектурных особенностей системы. Определенным преимуществом подхода можно считать способность модели адаптировать формулировки комментариев с использованием реальных имен методов и переменных из анализируемого кода, что повышает уровень персональной направленности рекомендаций.

Рассмотрим пример плохой работы модели на основе AUGER, представленный в листинге 4.2.

с оригинальным комментарием «Why not use the waiter function for this? https://github.com/openstack/designate-tempest-plugin/blob/master/designate_tempest_plugin/common/waiters.py#L24».

Для которого модель сгенерировала следующие предложения:

- «I don’t think this is a good idea, but I think it’s better to move it to the top of the file.»
- «I don’t think it’s a good idea to make this a constant.»
- «I don’t think it’s a good idea to have a test for this.»
- «I don’t think this is a good idea, but I think it’s better to move it to the top of the file as well.»

```
    recordset_data = data_utils.rand_recordset_data(
        record_type=\'A\', zone_name=self.zone[\'name\'])
    record = self.client.create_recordset(
        self.zone[\'id\'], recordset_data)[1]
    self.addCleanup(self.client.delete_recordset,
                    zone[\'id\'], record[\'id\'])

    LOG.info("Delete a Zone and wait till it\'s done")
    body = self.zone_client.delete_zone(zone[\'id\'])[1]
    LOG.info(\'Ensure we respond with DELETE+PENDING\')
    self.assertEqual(\'DELETE\', body[\'action\'])
    self.assertEqual(\'PENDING\', body[\'status\'])

    LOG.info(\'Ensure successful deletion of Recordset\')
    <review_tag>self.assertRaises(lib_exc.NotFound,<review_tag>
        lambda: self.client.show_recordset(self.zone[\'id\'], record[\'id\']))
```

Листинг 4.2: Пример рецензирования системой AUGER

- «I don’t think this is a good idea, but I think it’s better to have a test for this.»

При детальном анализе результатов отмечается высокая степень однородности генерируемых комментариев, которые преимущественно выражают сомнение в необходимости тех или иных фрагментов кода («it seems unnecessary»). Подобные замечания могут быть релевантны в определенных контекстах, таких как обнаружение недостижимого кода или неиспользуемых переменных, однако в рассматриваемом случае их применимость оказывается ограниченной. Статистический анализ показывает систематический характер данной проблемы: 2683 из 5000 сгенерированных примеров содержат фразу «I don’t think», а 911 примеров включают конструкцию «I don’t think it’s a good idea». Хотя подобные формулировки формально могут предварять конструктивные предложения, их стереотипный характер представляет методологическую проблему.

4.2. Интеграционное тестирование и оценка эффективности комплексной системы автоматического рецензирования

В рамках проведения интеграционного тестирования разработанная система была внедрена в рабочую среду локального репозитория Gerrit с целью оценки её эффективности в условиях реального процесса код-ревью. Эксперимент предусматривал длительный сбор откликов, позволяющих проанализировать

устойчивость работы системы. Различные реализованные конфигурации инструментов автоматизированного рецензирования применялись для анализа изменений исходного кода в производственной среде. В таблице 4.6 представлены агрегированные результаты эксперимента, включающие количественные показатели сгенерированных рецензий и их экспертной оценки. В качестве базовой языковой модели использовалась NousResearch/Nous-Hermes-2-Mistral-8x7B-DPO [88].

	Всего	Плохие	Средние	Применимые	Неплохие, %
MLCF	120	110	9	1	8.33%
AUGER	296	267	21	8	9.79%
CodeReviewer	100	91	8	1	9%
Nous-Hermes-2	160	122	28	9	23.13%

Таблица 4.6: Результаты оценки рецензий группой разработчиков

Анализ данных демонстрирует существенное превосходство подхода на основе большой языковой модели, который показал наивысший процент условно успешных рецензий (23.13% против менее 10% у других методов). Категория «Неплохих» включает комментарии, оцененные как «Средние» и «Применимые», что свидетельствует об их практической ценности. Наибольшее количество применимых комментариев (9) также сгенерировано LLM-подходом на базе модели Hermes [89]. Традиционные методы (MLCF, AUGER, CodeReviewer) демонстрируют статистически близкие результаты с преобладанием низкокачественных рецензий (более 90% в категории «Плохие»). Полученные результаты подтверждают перспективность использования LLM для задач автоматизированного рецензирования и указывают на необходимость дальнейшей оптимизации промптов и механизмов пост-обработки для увеличения доли релевантных комментариев.

На основании анализа современных бенчмарков и исследовательских работ, можно выделить несколько перспективных открытых моделей, подходящих для задач генерации и рецензирования кода.

- **DeepSeek-Coder**[90] — специализированная модель для генерации кода, показавшая высокие результаты на бенчмарках HumanEval и MultiPL-E. Отличается хорошим пониманием контекста и способностью работать с различными языками программирования.

- **CodeQwen**[91] (7B) — модель от Alibaba, оптимизированная для задач программирования. Проявляет высокую эффективность в анализе кода и предложении исправлений. В тестах демонстрирует адекватные советы и относительно быстрое время обработки — около 5 минут на аналогичном оборудовании
- **Llama 3.1** [92] (70B Hermes 3) — крупная мультиязычная модель с расширенными возможностями анализа кода. Занимает высокие позиции в бенчмарках по интеграции с внешними API и работе с документацией.
- **Code Llama**[93] (70B) — специализированная версия Llama, созданная для задач программирования. Способна генерировать код по описанию на естественном языке, исправлять ошибки и проводить ревью.
- **Qwen2.5-Coder**[94] (32B) — модель от Alibaba, демонстрирующая сбалансированное соотношение интеллекта, производительности и стоимости.

Базовые характеристики, такие как размер модели, наличие специализации при подготовке данных обучения, декларируемое качество работы в задачах понимания кода и генерации кода представлены в таблице 4.7.

Модель	Размер	Специализация	Понимание кода	Генерация кода
DeepSeek-Coder	6.7B, 33B	Генерация кода	Высокое	Очень высокое
CodeQwen	7B	Универсальная	Высокое	Высокое
Llama 3.1	70B	Универсальная	Очень высокое	Высокое
Code Llama	70B	Генерация кода	Высокое	Очень высокое
Qwen2.5-Coder	7B, 32B	Генерация кода	Высокое	Высокое

Таблица 4.7: Сравнение моделей для код-генерации и рецензирования

Эксперимент по проверке качества генерируемых рецензий, аналогичный показанному в таблице 4.6 был проведен для различных больших языковых моделей.

Результаты эксперимента, представленные в 4.8 демонстрируют существенное преимущество всех больших языковых моделей перед традиционными методами автоматизированного рецензирования, однако между собой получили сравнимые результаты:

Модель	Всего	Плохие	Средние	Применимые	Неплохие, %
Nous-Hermes-2-8x7B	160	122	28	9	23,13%
Qwen2.5-72B	117	83	24	9	28,21%
Qwen2.5-Coder-32B	128	90	26	12	29,69%
Qwen2.5-Coder-7B	137	105	23	9	23,36%
DeepSeek-Coder-33B	127	97	21	9	23,62%
DeepSeek-Coder-6.7B	124	95	20	9	23,39%
Llama 3.1-70B	117	81	25	11	30,77%
Code Llama-70B	112	83	21	7	25,00%

Таблица 4.8: Результаты оценки рецензий, сгенерированных различными LLM

- Qwen2.5-Coder-32B показывает наилучшие результаты в категории «Применимые» комментарии (12), что свидетельствует о её практической полезности в рабочих условиях.
- Llama 3.1-70B демонстрирует наибольший процент «Неплохих» комментариев (30,77%), однако при этом значительно больше по размерам и медленнее по скорости работы, чем остальные.
- DeepSeek-Coder-6.7B и Qwen2.5-Coder-7B показывают результаты, сопоставимые с оригинальной моделью Hermes (23,39% и 23,36% против 23,13%).
- Заметны сопоставимые результаты моделей DeepSeek-Coder и Qwen2.5-Coder для практически одинаковых размеров.

По итогам решено взять в качестве целевой модель **Qwen/Qwen2.5-Coder-32B**, так как она показала наилучие результаты для моделей до 32 миллиардов параметров.

4.3. Результаты валидации итоговой системы

4.3.1. Экспериментальная оценка

Для численной оценки применимости предложенного метода генерации рецензий было проведено два эксперимента. Первый эксперимент был выполнен на закрытом наборе данных, содержащем 123 добавленных и измененных метода, извлеченных из 10 коммитов локального сервера Gerrit. Корпус включал

11 новых и 25 модифицированных файлов, реализованных на языке Python. Выбор Python обусловлен его широкой распространенностью в задачах, связанных с ML и анализом кода. Код в выбранных коммитах характеризуется наличием типичных недостатков (например, нарушение стилевых соглашений, избыточность, потенциальные ошибки), что представляет собой репрезентативный набор задач для системы автоматического рецензирования. Аннотирование данных выполнялось двумя авторами исходного кода с целью точной классификации предложений, сгенерированных LLM, по следующим категориям: применимые, приемлемые (средние), но несущественные в данном контексте, и ошибочные, с низкой вероятностью внедрения разработчиком. Согласованность при ручной разметке, измеренная каппой Коэна, составила 0.81. В качестве базовой модели использовалась большая языковая модель Qwen/Qwen2.5-Coder-32B-Instruct. Данная модель была выбрана как репрезентативный пример современной LLM, ориентированной на задачи программирования; исследование применимости подхода к другим моделям представляет значительный интерес для будущих работ. Среднее время обработки одного набора изменений составило примерно 75 секунд, а среднее время дополнительной оценки одной рецензии около 7 секунд.

Таблица 4.9: Результаты оценки качества рецензий (без фильтров)

Метод	Плохие	Средние (шт, %)	Применимые (шт, %)	Общее
Базовая инструкция	95	33, 22	22, 14.7	150
Расширенный контекст	74	27, 21.9	22, 17.89	123
Улучшение требований	**48**	16, 17.77	**26**, **28.89**	90
Фиксированные примеры	92	37, 24.6	21, 14.00	150
Автоподбор примеров	60	**38**, **31.14**	24, 19.67	122

В табл. 4.9 приведены результаты разметки сгенерированных рецензий до применения фильтров. Наилучшие результаты показывает подход с улучшенными требованиями в инструкции как по точности (28.89%), так и по количеству применимых (26 против 22 у базовой инструкции). Стоит отметить, что автоматический подбор примеров повышает количество и точность неплохих (средних и применимых) рецензий (50.82%). Также интересным наблюдением

Таблица 4.10: Результаты оценки качества рецензий после мягкой фильтрации

Метод	Плохие	Средние (шт, %)	Применимые (шт, %)	Общее
Базовая инструкция	61	26, 24.52	19, 17.92	106
Расширенный контекст	42	21, 26.25	17, 21.25	80
Улучшение требований	**29**	14, 21.87	**21**, **32.81**	64
Фиксированные примеры	34	27, **34.6**	17, 21.79	78
Автоподбор примеров	40	**31**, 33.96	**21**, 22.83	92

Таблица 4.11: Результаты оценки качества рецензий после жесткой фильтрации

Метод	Плохие	Средние (шт, %)	Применимые (шт, %)	Общее
Базовая инструкция	42	19, 25	15, 19.74	76
Расширенный контекст	22	14, 27.4	15, 29.41	51
Улучшение требований	**21**	13, 24.07	**20**, **37.04**	54
Фиксированные примеры	24	12, 25.53	11, 23.40	47
Автоподбор примеров	26	**22**, **33.3**	18, 27.27	66

является достаточно низкие показатели у подхода с фиксированными примерами, точность применимых замечаний даже ниже, чем у базовой инструкции. Таким образом применение вышеописанных методов улучшения инструкции повысило точность полностью применимых рецензий в 2 раза, а неплохих примерно в 1.4 раза. В результатах тестирования работы BitsAI на модели Qwen2.5-Coder-32B-instruct указана средняя точность полезных комментариев на уровне 10%, что немного ниже, но сравнимо с полученным в табл. 4.9 для базовой инструкции.

В табл. 4.10 и 4.11 приведены результаты применения инструкции из секции 3.6 в качестве мягкой и жесткой пост-фильтрации сгенерированных рецензий, соответственно. По метрике точности жесткая фильтрация показывает более высокие (до 37% применимых) результаты, однако общее количество, в том числе и неплохих комментариев сильно снижается (40 против 52 для подхода с автоподбором примеров). Из таблицы видно, что применение одной только фильтрации к базовой инструкции не дает такого прироста точности, как на-

стройка инструкции, однако в сумме с остальными подходами значительно повышает итоговую точность (в 2.5 раза применимые до 37% и в 1.6 раза неплохие рецензии до 61%). Также подход с мягкой и жесткой фильтрациями добавляет гибкость в настройку итогового инструмента.

Было проведено тестирование эффективности дообучения на наборах данных различного объема с обратным откликом пользователей по генерируемым рецензиям: по 50, 100 и 150 положительных и отрицательных примеров соответственно. На рисунке 4.1 представлена динамика доли рецензий удовлетворительного качества на тестовой выборке. Как показывают результаты, при объеме выборки в 50 примеров наблюдается незначительное снижение показателей по сравнению с исходной моделью. Однако при увеличении объема данных до 100 и 150 примеров качество генерации существенно улучшается, за счет фильтраций, так как модель-судья начинает лучше ставить оценки и убирать плохие рецензии.

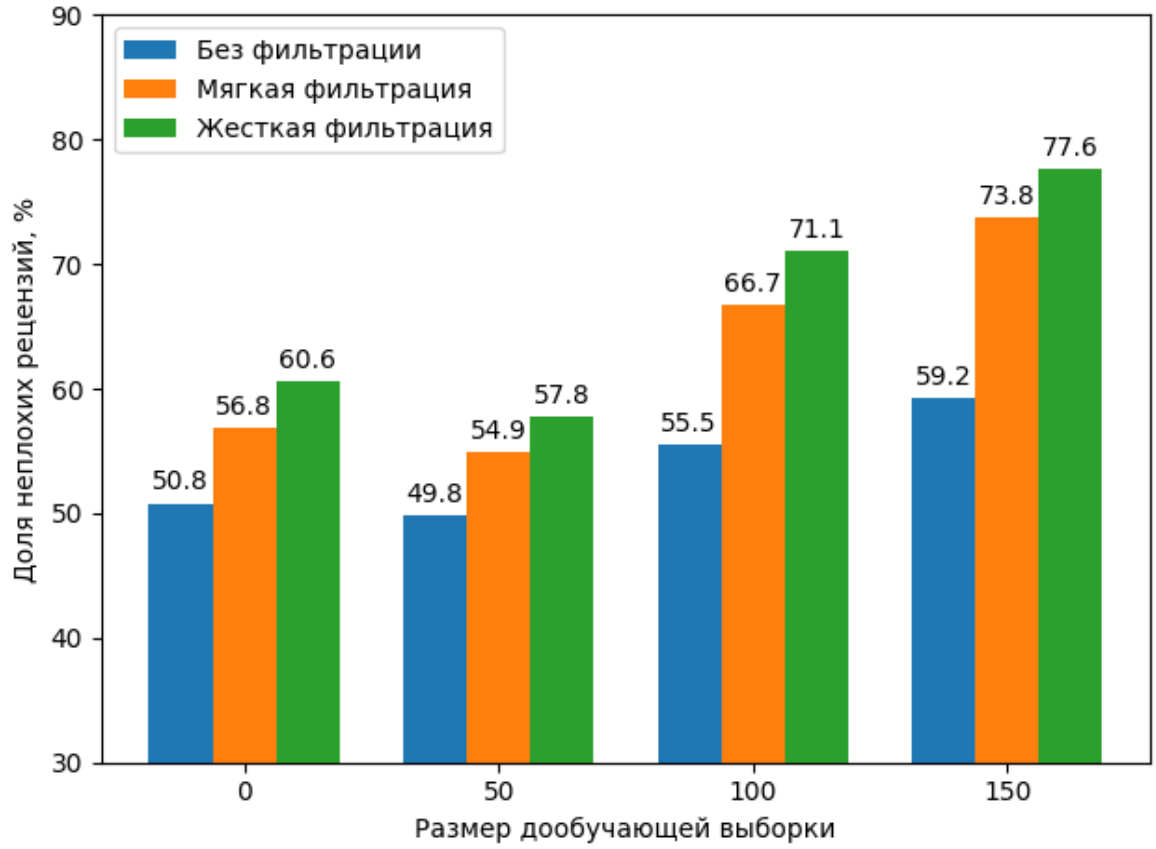

Рис. 4.1: Доля неплохих рецензий при дообучении модели

В таблице 4.12 представлены результаты оценки качества рецензий, сгенерированных дообученной моделью на наборе из 150 положительных и 150 отри-

Таблица 4.12: Результаты оценки качества рецензий дообученной модели

Метод	Плохие	Средние (шт, %)	Применимые (шт, %)	Общее
Без фильтрации	33, 40.7	28, 34.5	20, 24.7	81
Мягкая фильтрация	17, 26.1	28, 43.1	20, 30.7	65
Жесткая фильтрация	11, 22.4	19, 38.8	19, 38.8	49

цательных примеров. Наибольшую точность демонстрирует подход с жесткой фильтрацией, где доля применимых рецензий достигает 38.8% при таком же проценте средних комментариев. Применение мягкой фильтрации также показывает значительное улучшение по сравнению с базовым подходом без фильтрации — точность применимых рецензий увеличивается с 24.7% до 30.7%.

Интересно отметить, что жесткая фильтрация позволяет достичь баланса между применимыми и средними рецензиями (по 38.8%), при этом значительно снижая долю плохих комментариев до 22.4% по сравнению с 40.7% в варианте без фильтрации. Однако стоит учитывать, что общее количество рецензий при жесткой фильтрации сокращается почти в 1.7 раза (с 81 до 49). Мягкая фильтрация представляет собой компромиссный вариант, сохраняя большее количество рецензий (65) при существенном улучшении их качества.

Второй эксперимент был проведен для проверки масштабируемости метода на более крупном наборе данных. Использовался открытый набор данных CRer [10], содержащий 1000 примеров изменений кода на Python и Java и соответствующих рецензий. Исследуемая модель (инструкция с улучшенными требованиями) сгенерировала рецензии для представленных фрагментов кода. Относительная релевантность выводов оценивалась с помощью ранжирующей системы DeepCRCEval, сортирующей комментарии по степени соответствия целевому коду.

Данный оценщик осуществляет сортировку списка рецензий по степени их соответствия (релевантности) целевому поданному участку кода. В каждый ранжируемый пакет для одного фрагмента кода включались N комментариев, сгенерированных усовершенствованной моделью, и один комментарий, полученный с использованием промпта из системы DeepCRCEval. Ключевое различие заключается в формате вывода: базовый подход генерирует единое сообщение

на фрагмент кода, тогда как усовершенствованная инструкция пишет список отдельных замечаний. Для обеспечения сопоставимости результатов при вариативном значении N, рецензии, сгенерированные базовым запросом к системе DeepCRCEval, которым оценщик присваивал позицию ниже первой, принудительно устанавливались на ранг 2. Эта мера предотвратила искажение метрики средне обратного ранга (Mean Reciprocal Rank, MRR) из-за не фиксированного количества конкурирующих рецензий от усовершенствованной модели в разных пакетах. Усовершенствованный подход продемонстрировал значимое превосходство, достигнув среднего ранга (MRR) 0.915, в то время как результат базовой инструкции составил 0.585. Полученные значения средне обратного ранга свидетельствуют о более высокой эффективности разработанного подхода в генерации применимых рецензий.

4.3.2. Качественный анализ

На рис. 4.2 приведены примеры рецензий и ручной разметки, которую проводили эксперты. Применимые комментарии говорят о явных проблемах, таких как отсутствие контекстного менеджера для корректной обработки закрытия файла или о неэффективной обработке элементов файла, которые можно сократить использованием генератора списка. К рецензиям среднего качества можно отнести сообщение о некорректном именовании переменных или о несогласованном формате чтения файла и указанном типе данных, что так же может быть опечаткой и не приведет к значительным нарушениям в работе программы. Такие замечания могут оказаться полезными, либо натолкнуть на мысль о необходимости внести изменения в данный участок кода. К ошибочным комментариям относятся просьбы исправить код на тождественный существующему или желание внедрить в программу библиотеку записи логов, хотя в данном контексте это абсолютно не нужно.

Рис. 4.3 демонстрирует репрезентативный пример кода и соответствующих рецензий, сгенерированных моделью при использовании различных промптов, описанных в 3.5.1.. Ключевым замечанием к анализируемой функции от настоящих рецензентов является повторная компиляция регулярных выражений при каждом вызове метода, что создает избыточные вычислительные затраты. С точки зрения оптимизации производительности, предпочтительным решением было бы вынесение предварительно скомпилированных выражений в область

Пример кода

```
import os

def process_user_data(user_id: int, file_name: str, other_line: str) -> None:
    data_efficasy = 0

    full_path = os.path.join('data/', file_name)
    file: IO[str] = open(full_path, 'rb')
    for line in file:
        if line == other_line:
            data_efficasy += 1

    print(f"Data processed with efficasy: {data_efficasy}")
```

Применимые комментарии

```
Line: 7
The file should be opened using a context manager to ensure it gets closed properly.
Suggested fix: with open(full_path, 'rb') as file:
```

```
Line: 8
The loop might be transformed to sum function with generator comprehension expression.
Suggested fix: data_efficasy = sum(1 for line in file if if line == other_line)
```

Средние комментарии

```
Line: 2
Variable name 'data_efficasy' is misspelled. It should be 'data_efficiency'.
Suggested fix: data_efficiency = 0
```

```
Line: 7
The file is opened in binary mode ('rb') but read as text. Change the mode to 'r' for text
files.
Suggested fix: file: IO[str] = open(full_path, 'r')
```

Ошибочные комментарии

```
Line: 1
The function should import the required module 'os' at the beginning of the file.
Suggested fix: import os
```

```
Line: 12
Avoid using print for logging. Use a logging library like logging for better control and
flexibility.
Suggested fix: import logging; logging.basicConfig(level=logging.INFO); logging.info(f"Data
processed with efficasy: {data_efficasy}")
```

Рис. 4.2: Примеры рецензий и их ручной разметки

глобальных переменных.

Рецензия, сформированная на основе базового запроса, содержит ошибочное утверждение о недостаточной выразительности регулярных выражений и их неспособности корректно обрабатывать целевые значения. Данное заключение не соответствует действительности, так как текущая реализация метода обеспечивает полное покрытие требуемых случаев. Запрос с расширенным контекстом, напротив, корректно идентифицирует проблему повторной компиляции регулярных выражений. В предложенном исправлении модель выносит регулярные выражения за его пределы, однако вносит необоснованные изменения в сами выражения.

Усовершенствованный вариант запроса генерирует рецензию, не только со-

Пример кода

```
def get_compiler(config_file_path: Optional[Path]) -> str:
    default_compiler = 'gcc-10'

    if config_file_path is not None and config_file_path.exists():
        with open(config_file_path, 'r', encoding='utf-8') as f:
            config = f.read()
        gcc_pattern = re.compile(r'CONFIG_CC_VERSION_TEXT=.+gcc.*?(\d+)')
        clang_pattern = re.compile(
            r'CONFIG_CC_VERSION_TEXT=.+clang version (\d+)'
        )

        gcc_match = gcc_pattern.search(config)
        if gcc_match:
            version = gcc_match.group(1)
            return f'gcc-{version}'

        clang_match = clang_pattern.search(config)
        if clang_match:
            version = clang_match.group(1)
            return f'clang-{version}'

    return default_compiler
```

Базовая инструкция

The regular expressions used to extract the compiler version are too simplistic and may not handle all valid version formats. For example, GCC and Clang versions can include multiple numbers (e.g., gcc-10.2.0 or clang version 12.0.0). Update the regular expressions to capture the full version string. Additionally, consider adding error handling for cases where the version string is not found or is malformed.

Расширенный контекст

The regex patterns are compiled inside the function, which means they are recompiled every time the function is called. Move the pattern compilation outside the function to improve performance.
Suggested fix:

```
gcc_pattern = re.compile(r'CONFIG_CC_VERSION_TEXT=.+gcc.*?\s(\d+)\b')
clang_pattern = re.compile(r'CONFIG_CC_VERSION_TEXT=.+clang version\s(\d+)\b')
def get_compiler(config_file_path: Optional[Path]) -> str:",
```

Улучшение требований

The gcc_pattern and clang_pattern are compiled every time the function is called. These patterns are static and can be compiled once outside the function to improve performance.
Suggested fix:

```
gcc_pattern = re.compile(r'CONFIG_CC_VERSION_TEXT=.+gcc.*?(\d+)')
clang_pattern = re.compile(r'CONFIG_CC_VERSION_TEXT=.+clang version (\d+)')
def get_compiler(config_file_path: Optional[Path]) -> str:
    default_compiler = 'gcc-10'
    if config_file_path is not None and config_file_path.exists():
        with open(config_file_path, 'r', encoding='utf-8') as f:
            config = f.read()
        gcc_match = gcc_pattern.search(config)
        if gcc_match:
            version = gcc_match.group(1)
            return f'gcc-{version}'
        clang_match = clang_pattern.search(config)
        if clang_match:
            version = clang_match.group(1)
            return f'clang-{version}'
    return default_compiler
```

Автоподбор примеров

Avoid reading the entire file into memory by using Path.read_text() for better readability and performance.
Suggested fix:

```
config = config_file_path.read_text()
```

Рис. 4.3: Примеры рецензий при различных инструкциях

гласующуюся с предыдущей инструкцией, но и предлагает полный исправленный метод с использованием вынесенных глобальных переменных, сохраняя при этом исходные регулярные выражения без изменений. Наконец, рецензия, полученная при использовании запроса с автоматически подобранными примерами содержит новое замечание о том, что вместо самостоятельного чтения файла через функцию open можно использовать встроенный метод read_text(), так как переменная config_file_path является объектом типа Path из библиотеки pathlib.

4.4. Выводы

Экспериментальное исследование, проведенное в четвертой главе, демонстрирует комплексную оценку эффективности систем автоматического рецензирования исходного кода на основе больших языковых моделей. Результаты показывают, что использование больших языковых моделей позволяет достичь значительного улучшения качества генерируемых рецензий по сравнению с традиционными подходами (MLCF, AUGER, CodeReviewer). Доля приемлемых комментариев (категории «Средние» и «Применимые») при использовании LLM более чем в два раза превышает показатели альтернативных методов.

Важным выводом является подтверждение зависимости качества генерации от объема и характера обучающих данных. Эксперименты выявили рост всех метрик при увеличении размера выборки: схожесть по метрике косинусного расстояния увеличилось на 15%, BLEU-4 — на 43% для топ-1 результатов. При этом использование тематически отфильтрованных данных обеспечивает лучшие результаты даже при меньшем объеме.

Значительный вклад вносит оптимизация инструкций и применение механизмов пост-обработки. Модификация инструкций с автоматическим подбором примеров позволила увеличить количество применимых рецензий в 2 раза, а их доля при жесткой фильтрации достигла 37%. Однако выявлен компромисс между точностью и полнотой — жесткая фильтрация значительно сокращает общее количество комментариев.

Практическая валидация в реальной среде Gerrit подтвердила эффективность подхода, хотя и выявила некоторое снижение качества при работе с разнородными патчсетами. Сравнительный анализ на наборе CRer продемонстрировал превосходство улучшенной системы (MRR = 0,915 против 0,585 у базовой

версии).

Важным практическим выводом является подтверждение экономической эффективности внедрения таких систем. Для компании со сложившейся практикой код-ревью автоматизация процесса рецензирования позволяет снизить издержки на эту операцию на 15-20%. Это достигается за счет сокращения времени, затрачиваемого разработчиками на рутинный анализ кода, ускорения процесса интеграции новых участников в проект и уменьшения количества ошибок, обнаруживаемых на поздних стадиях разработки.

Экономический эффект реализуется через несколько механизмов: сокращение временных затрат опытных разработчиков на ревью, ускорение процесса адаптации новых сотрудников за счет обучающего эффекта автоматических рецензий, снижение стоимости исправления дефектов благодаря их более раннему выявлению.

Заключение

Основные результаты диссертационной работы заключаются в следующем.

- Разработан и реализован метод сбора данных об исправлениях антипаттернов кода из истории разработки проектов с открытым исходным кодом. Показано, что собранные таким образом данные позволяют обучать новые более точные классификаторы.
- Разработаны и реализованы методы детектирования антипаттернов исходного кода, сочетающие метрики кода с методами машинного обучения. Полученный подход позволяет повысить точность и полноту результатов классификации.
- Разработаны методы автоматического рецензирования исходного кода, основанные на применении языковых моделей, которые предлагают разработчику применимые советы по улучшению исходного кода. Показана эффективность внедренных новых модификаций в существующие методы.
- Разработан и реализован программный комплекс, сочетающий работу детекторов антипаттернов исходного кода в систему автоматического рецензирования.

Список иллюстраций

Список таблиц

Список литературы

1. *Качанов В. В.* Автоматическая генерация рецензий к коду: эволюция инструкций и интеллектуальная фильтрация // *Труды ИСП РАН.* — 2025. — Vol. 37, no. 4. — Pp. 117–132.
2. *Качанов В. В., Хитрова А. С., Марков С. И.* Извлечение именованных сущностей из рецензий к исходному коду // *Труды ИСП РАН.* — 2023. — Vol. 35, no. 5. — Pp. 193–214.
3. Технический долг в жизненном цикле разработки ПО: запахи кода / В. В. Качанов, М. К. Ермаков, Г. А. Панкратенко et al. // *Труды ИСП РАН.* — 2021. — Vol. 33, no. 6. — Pp. 95–110.
4. *Качанов В. В., Марков С. И., Цурков В. И.* МАШИННОЕ ОБУЧЕНИЕ В ПРОБЛЕМЕ ТЕХНИЧЕСКОГО ДОЛГА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ // *Известия Российской академии наук. Теория и системы управления.* — 2023. — no. 4. — Pp. 98–104.
5. *Петрова П. А., Марков С. И., Качанов В. В.* Создание набора данных для комбинированной классификации рецензий исходного кода // Интеллектуализация обработки информации. Тезисы докладов 15-й международной конференции. — 2024. — Pp. 83–851.
6. NOBRAINER: ИНСТРУМЕНТ ПРЕОБРАЗОВАНИЯ C/C++ КОДА НА ОСНОВЕ ПРИМЕРОВ / В. В. Савченко, К. С. Сорокин, И. Е. Бронштейн et al. // *Программирование.* — 2020. — no. 5. — Pp. 33–46.
7. *Bacchelli Alberto, Bird Christian.* Expectations, outcomes, and challenges of modern code review // 2013 35th International Conference on Software Engineering (ICSE). — 2013. — Pp. 712–721.
8. *Ebert Christof, Jones Capers.* Embedded Software: Facts, Figures, and Future // *Computer.* — 2009. — Vol. 42, no. 4. — Pp. 42–52.
9. Best Kept Secrets of Peer Code Review / J. Cohen, E. Brown, S. Teleki, B. DuRette. — Printing Systems, 2006. `https://books.google.ru/books?id=b9ywHanWN5kC`.
10. *Jabrayilzade Elgun, Evtikhiev Mikhail, Tüzün Eray, Kovalenko Vladimir.* Bus Factor In Practice. — 2022. `https://arxiv.org/abs/2202.01523`.

11. *Paul Rajshakhar, Turzo Asif Kamal, Bosu Amiangshu.* Why Security Defects Go Unnoticed during Code Reviews? A Case-Control Study of the Chromium OS Project. — 2021. https://arxiv.org/abs/2102.06909.
12. Commentfinder: a simpler, faster, more accurate code review comments recommendation / Yang Hong, Chakkrit Tantithamthavorn, Patanamon Thongtanunam, Aldeida Aleti // Proceedings of the 30th ACM joint European software engineering conference and symposium on the foundations of software engineering. — 2022. — Pp. 507–519.
13. *Ratcliff John W, Metzener David E et al.* Pattern matching: The gestalt approach // *Dr. Dobb's Journal.* — 1988. — Vol. 13, no. 7. — P. 46.
14. *Li Lingwei, Yang Li, Jiang Huaxi et al.* AUGER: Automatically Generating Review Comments with Pre-training Models. — 2022. https://arxiv.org/abs/2208.08014.
15. Automating code review activities by large-scale pre-training / Zhiyu Li, Shuai Lu, Daya Guo et al. // Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. — 2022. — Pp. 1035–1047.
16. *Raffel Colin, Shazeer Noam, Roberts Adam et al.* Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. — 2023. https://arxiv.org/abs/1910.10683.
17. *Lu Junyi, Li Xiaojia, Hua Zihan et al.* DeepCRCEval: Revisiting the Evaluation of Code Review Comment Generation. — 2025. https://arxiv.org/abs/2412.18291.
18. *Wei Jason, Wang Xuezhi, Schuurmans Dale et al.* Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. — 2023. https://arxiv.org/abs/2201.11903.
19. *Sun Tao, Xu Jian, Li Yuanpeng et al.* BitsAI-CR: Automated Code Review via LLM in Practice. — 2025. https://arxiv.org/abs/2501.15134.
20. Qwen2. 5-Coder Technical Report / Binyuan Hui, Jian Yang, Zeyu Cui et al. // *arXiv preprint arXiv:2409.12186.* — 2024.
21. *Fowler Martin.* Refactoring: Improving the Design of Existing Code. — Boston, MA, USA: Addison-Wesley, 1999.

22. *Fenton Norman E, Neil Martin.* Software metrics: successes, failures and new directions // *Journal of Systems and Software.* — 1999. — Vol. 47, no. 2-3. — Pp. 149–157.
23. PMD - source code analyzer. — `https://github.com/pmd/pmd`. — Accessed: 2025-06-10.
24. *Sharma Tushar, Mishra Pratibha, Tiwari Rohit.* Designite: A software design quality assessment tool // Proceedings of the 1st international workshop on bringing architectural design thinking into developers' daily activities. — 2016. — Pp. 1–4.
25. SonarQube Server - AUTOMATED CODE QUALITY AND SECURITY REVIEWS. — `https://www.sonarsource.com/products/sonarqube/`. — Accessed: 2025-06-10.
26. *Ferme Vincenzo.* JCodeOdor: A software quality advisor through design flaws detection // *Master's thesis, Universita degli Studi di Milano-Bicocca.* — 2013.
27. *Kokol Peter, Kokol Marko, Zagoranski Sašo.* Code smells: A synthetic narrative review // *arXiv preprint arXiv:2103.01088.* — 2021.
28. *Barbez Antoine, Khomh Foutse, Guéhéneuc Yann-Gaël.* A machine-learning based ensemble method for anti-patterns detection // *Journal of Systems and Software.* — 2020. — Vol. 161. — P. 110486.
29. *Fontana Francesca Arcelli, Zanoni Marco.* Code smell severity classification using machine learning techniques // *Knowledge-Based Systems.* — 2017. — Vol. 128. — Pp. 43–58.
30. On the feasibility of transfer-learning code smells using deep learning / Tushar Sharma, Vasiliki Efstathiou, Panos Louridas, Diomidis Spinellis // *arXiv preprint arXiv:1904.03031.* — 2019.
31. Comparing and experimenting machine learning techniques for code smell detection / Francesca Arcelli Fontana, Mika V Mäntylä, Marco Zanoni, Alessandro Marino // *Empirical Software Engineering.* — 2016. — Vol. 21. — Pp. 1143–1191.
32. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation / Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta et al. // Proceedings of the 40th International Conference on Software Engineering. — 2018. — Pp. 482–482.

33. *Madeyski Lech, Lewowski Tomasz.* MLCQ: Industry-relevant code smell data set // Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering. — 2020. — Pp. 342–347.
34. *Lenarduzzi Valentina, Saarimäki Nyyti, Taibi Davide.* The technical debt dataset // Proceedings of the fifteenth international conference on predictive models and data analytics in software engineering. — 2019. — Pp. 2–11.
35. Automatic software refactoring via weighted clustering in method-level networks / Ying Wang, Hai Yu, Zhiliang Zhu et al. // *IEEE Transactions on Software Engineering.* — 2017. — Vol. 44, no. 3. — Pp. 202–236.
36. *Karampatsis Rafael-Michael, Sutton Charles.* How often do single-statement bugs occur? the manysstubs4j dataset // Proceedings of the 17th international conference on mining software repositories. — 2020. — Pp. 573–577.
37. Landfill: An open dataset of code smells with public evaluation / Fabio Palomba, Dario Di Nucci, Michele Tufano et al. // 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories / IEEE. — 2015. — Pp. 482–485.
38. Detecting code smells using machine learning techniques: Are we there yet? / Dario Di Nucci, Fabio Palomba, Damian A Tamburri et al. // 2018 ieee 25th international conference on software analysis, evolution and reengineering (saner) / IEEE. — 2018. — Pp. 612–621.
39. A large-scale empirical study on the lifecycle of code smell co-occurrences / Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta et al. // *Information and Software Technology.* — 2018. — Vol. 99. — Pp. 1–10.
40. Trusted Code Smells Dataset. — `https://doi.org/10.5281/zenodo.7612725`. — Accessed: 2025-06-10.
41. *Alon Uri, Zilberstein Meital, Levy Omer, Yahav Eran.* code2vec: Learning Distributed Representations of Code. — 2018. `https://arxiv.org/abs/1803.09473`.
42. Towards automating code review activities / Rosalia Tufano, Luca Pascarella, Michele Tufano et al. // 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE) / IEEE. — 2021. — Pp. 163–174.
43. Bleu: a method for automatic evaluation of machine translation / Kishore Papineni, Salim Roukos, Todd Ward, Wei-Jing Zhu // Proceedings of the 40th annual meeting of the Association for Computational Linguistics. — 2002. — Pp. 311–318.

44. A review of recurrent neural networks: LSTM cells and network architectures / Yong Yu, Xiaosheng Si, Changhua Hu, Jianxun Zhang // *Neural computation.* — 2019. — Vol. 31, no. 7. — Pp. 1235–1270.
45. Codebert: A pre-trained model for programming and natural languages / Zhangyin Feng, Daya Guo, Duyu Tang et al. // *arXiv preprint arXiv:2002.08155.* — 2020.
46. *Wang Yue, Wang Weishi, Joty Shafiq, Hoi Steven C. H.* CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. — 2021. `https://arxiv.org/abs/2109.00859`.
47. *Grootendorst Maarten.* BERTopic: Neural topic modeling with a class-based TF-IDF procedure // *arXiv preprint arXiv:2203.05794.* — 2022.
48. *McInnes Leland, Healy John, Melville James.* UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction. — 2020. `https://arxiv.org/abs/1802.03426`.
49. hdbscan: Hierarchical density based clustering. / Leland McInnes, John Healy, Steve Astels et al. // *J. Open Source Softw.* — 2017. — Vol. 2, no. 11. — P. 205.
50. Bag of tricks for efficient text classification / Armand Joulin, Edouard Grave, Piotr Bojanowski, Tomas Mikolov // *arXiv preprint arXiv:1607.01759.* — 2016.
51. *Reimers Nils, Gurevych Iryna.* Sentence-bert: Sentence embeddings using siamese bert-networks // *arXiv preprint arXiv:1908.10084.* — 2019.
52. Automatic Classification of Review Comments in Pull-based Development Model. / Zhixing Li, Yue Yu, Gang Yin et al. // SEKE. — 2017. — Pp. 572–577.
53. Automated code review comment classification to improve modern code reviews / Miroslaw Ochodek, Miroslaw Staron, Wilhelm Meding, Ola Söder // International Conference on Software Quality / Springer. — 2022. — Pp. 23–40.
54. Towards Automated Classification of Code Review Feedback to Support Analytics / Asif Kamal Turzo, Fahim Faysal, Ovi Poddar et al. // 2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) / IEEE. — 2023. — Pp. 1–12.

55. *Luhn Hans Peter.* A statistical approach to mechanized encoding and searching of literary information // *IBM Journal of research and development.* — 1957. — Vol. 1, no. 4. — Pp. 309–317.
56. Scikit-learn: Machine learning in Python / Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort et al. // *the Journal of machine Learning research.* — 2011. — Vol. 12. — Pp. 2825–2830.
57. *Rehurek Radim, Sojka Petr.* Software framework for topic modelling with large corpora // *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks.* — 2010.
58. *Quinlan J. Ross.* Induction of decision trees // *Machine learning.* — 1986. — Vol. 1. — Pp. 81–106.
59. *Cramer Jan Salomon.* The origins of logistic regression // *Tinbergen Institute discussion paper.* — 2002.
60. *Tikhonov Andrei Nikolaevich.* Regularization of incorrectly posed problems // Moscow "Science-/ Soviet Mathematics Doklady. — 1963.
61. Support vector machines / Marti A. Hearst, Susan T Dumais, Edgar Osuna et al. // *IEEE Intelligent Systems and their applications.* — 1998. — Vol. 13, no. 4. — Pp. 18–28.
62. *Ho Tin Kam.* Random decision forests // Proceedings of 3rd international conference on document analysis and recognition / IEEE. — Vol. 1. — 1995. — Pp. 278–282.
63. *Friedman Jerome H.* Greedy function approximation: a gradient boosting machine // *Annals of statistics.* — 2001. — Pp. 1189–1232.
64. Bert: Pre-training of deep bidirectional transformers for language understanding / Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova // Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics human language technologies. — 2019. — Pp. 4171–4186.
65. *Bosu Amiangshu, Greiler Michaela, Bird Christian.* Characteristics of Useful Code Reviews: An Empirical Study at Microsoft. — 2015. — 05. — Pp. 146–156.
66. Dissecting GitHub Code Reviews: A Text Classification Experiment. — `https://mfadhel.com/github-code-reviews/`. — Accessed: 2025-06-10.

67. *Sarker Jaydeb, Turzo Asif Kamal, Dong Ming, Bosu Amiangshu.* Automated Identification of Toxic Code Reviews Using ToxiCR. — 2023. https://arxiv.org/abs/2202.13056.
68. CleanCodeReview. — https://doi.org/10.5281/zenodo.13828619. — Accessed: 2025-06-10.
69. Transformers: State-of-the-art natural language processing / Thomas Wolf, Lysandre Debut, Victor Sanh et al. // Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations. — 2020. — Pp. 38–45.
70. TensorFlow: Large-scale machine learning on heterogeneous systems / Abadi Martín, Agarwal Ashish, Barham Paul et al. // *Software available from tensorflow. org.* — 2015. — Vol. 7.
71. *Popović Maja.* chrF: character n-gram F-score for automatic MT evaluation // Proceedings of the tenth workshop on statistical machine translation. — 2015. — Pp. 392–395.
72. *Popović Maja.* chrF++: words helping character n-grams // Proceedings of the second conference on machine translation. — 2017. — Pp. 612–618.
73. *Zhang Tianyi, Kishore Varsha, Wu Felix et al.* BERTScore: Evaluating Text Generation with BERT. — 2020. https://arxiv.org/abs/1904.09675.
74. *He Pengcheng, Liu Xiaodong, Gao Jianfeng, Chen Weizhu.* DeBERTa: Decoding-enhanced BERT with Disentangled Attention. — 2021. https://arxiv.org/abs/2006.03654.
75. *Liu Chunhua, Lin Hong Yi, Thongtanunam Patanamon.* Too Noisy To Learn: Enhancing Data Quality for Code Review Comment Generation. — 2025. https://arxiv.org/abs/2502.02757.
76. *Yu Yongda, Zhang Lei, Rong Guoping et al.* Distilling Desired Comments for Enhanced Code Review with Large Language Models. — 2025. https://arxiv.org/abs/2412.20340.
77. Structured outputs: Everything you should know. — https://humanloop.com/blog/structured-outputs. — Accessed: 2025-06-10.
78. Structured outputs in llms: Definition, techniques, applications, benefits. — https://www.leewayhertz.com/structured-outputs-in-llms/. — Accessed: 2025-06-10.

79. How to return structured data from a model. — https://python.langchain.com/docs/how_to/structured_output/. — Accessed: 2025-06-10.
80. Pydantic. — https://docs.pydantic.dev/latest/. — Accessed: 2025-06-10.
81. *Lewis Patrick, Perez Ethan, Piktus Aleksandra et al.* Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. — 2021. https://arxiv.org/abs/2005.11401.
82. Llm-as-a-judge: a complete guide to using llms for evaluations. — https://www.evidentlyai.com/llm-guide/llm-as-a-judge. — Accessed: 2025-06-10.
83. *Zheng Lianmin, Chiang Wei-Lin, Sheng Ying et al.* Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. — 2023. https://arxiv.org/abs/2306.05685.
84. LLM review prompts. — https://github.com/vova98/llm_review_prompts. — Accessed: 2025-06-10.
85. *Ouyang Long, Wu Jeff, Jiang Xu et al.* Training language models to follow instructions with human feedback. — 2022. https://arxiv.org/abs/2203.02155.
86. *Rafailov Rafael, Sharma Archit, Mitchell Eric et al.* Direct Preference Optimization: Your Language Model is Secretly a Reward Model. — 2024. https://arxiv.org/abs/2305.18290.
87. *Ethayarajh Kawin, Xu Winnie, Muennighoff Niklas et al.* KTO: Model Alignment as Prospect Theoretic Optimization. — 2024. https://arxiv.org/abs/2402.01306.
88. *teknium, Quesnelle Jeffrey, Guang Chen.* Hermes 3 Technical Report. — 2024.
89. *Ayed Fadhel, Maatouk Ali, Piovesan Nicola et al.* Hermes: A Large Language Model Framework on the Journey to Autonomous Networks. — 2024. https://arxiv.org/abs/2411.06490.
90. *Guo Daya, Zhu Qihao, Yang Dejian et al.* DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence. — 2024. https://arxiv.org/abs/2401.14196.
91. *Bai Jinze, Bai Shuai, Chu Yunfei et al.* Qwen Technical Report. — 2023. https://arxiv.org/abs/2309.16609.

92. *Grattafiori Aaron, Dubey Abhimanyu, Jauhri Abhinav et al.* The Llama 3 Herd of Models. — 2024. https://arxiv.org/abs/2407.21783.
93. *Rozière Baptiste, Gehring Jonas, Gloeckle Fabian et al.* Code Llama: Open Foundation Models for Code. — 2024. https://arxiv.org/abs/2308.12950.
94. *Hui Binyuan, Yang Jian, Cui Zeyu et al.* Qwen2.5-Coder Technical Report. — 2024. https://arxiv.org/abs/2409.12186.
95. *Kachanov V. V., Khitrova A. S., Markov S. I.* Named Entity Recognition for Code Review Comments // *Program Comput Soft.* — 2024. — Vol. 50. — Pp. 511–523.
96. *Kachanov V. V., Markov S. I., Tsurkov V. I.* Machine Learning for Software Technical Debt Detection // *J. Comput. Syst. Sci. Int.* — 2023. — Vol. 62. — Pp. 689–694.
97. NOBRAINER: A Tool for Example-Based Transformation of C/C++ Code / V. V. Savchenko, K. S. Sorokin, I. E. Bronshtein et al. // *Program. Comput. Softw.* — 2020. — Vol. 46, no. 5. — Pp. 362–372.
98. Automatic code review by learning the revision of source code / Shu-Ting Shi, Ming Li, David Lo et al. // Proceedings of the AAAI Conference on Artificial Intelligence. — AAAI'19/IAAI'19/EAAI'19. — AAAI Press, 2019. — 8 pp. https://doi.org/10.1609/aaai.v33i01.33014910.
99. *Hamdy ABEER, Tazy Mostafa.* Deep hybrid features for code smells detection // *Journal of Theoretical and Applied Information Technology.* — 2020. — Vol. 98, no. 14. — Pp. 2684–2696.
100. *Bank Dor, Koenigstein Noam, Giryes Raja.* Autoencoders // *Machine learning for data science handbook: data mining and knowledge discovery handbook.* — 2023. — Pp. 353–374.
101. *Zhang Guoqiang Peter.* Neural networks for classification: a survey // *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews).* — 2000. — Vol. 30, no. 4. — Pp. 451–462.
102. *Fenton Norman E, Neil Martin.* Software metrics: successes, failures and new directions // *Journal of Systems and Software.* — 1999. — Vol. 47, no. 2. — Pp. 149–157. https://www.sciencedirect.com/science/article/pii/S0164121299000357.

103. *Casey Beatrice, Santos Joanna C. S., Perry George.* A Survey of Source Code Representations for Machine Learning-Based Cybersecurity Tasks. — 2024. https://arxiv.org/abs/2403.10646.

104. Representation vs. Model: what matters most for source code vulnerability detection / Wei Zheng, Abubakar Omari Abdallah Semasaba, Xiaoxue Wu et al. // 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) / IEEE. — 2021. — Pp. 647–653.

105. *Xia Xiaoling, Wang Yu, Yang Ye.* Source Code Vulnerability Detection Based On SAR-GIN // 2021 2nd International Conference on Electronics, Communications and Information Technology (CECIT) / IEEE. — 2021. — Pp. 1144–1149.

106. Modeling and discovering vulnerabilities with code property graphs / Fabian Yamaguchi, Nico Golde, Daniel Arp, Konrad Rieck // 2014 IEEE symposium on security and privacy / IEEE. — 2014. — Pp. 590–604.

107. *Alon Uri, Zilberstein Meital, Levy Omer, Yahav Eran.* A General Path-Based Representation for Predicting Program Properties. — 2018. https://arxiv.org/abs/1803.09544.

108. *Alon Uri, Brody Shaked, Levy Omer, Yahav Eran.* code2seq: Generating Sequences from Structured Representations of Code. — 2019. https://arxiv.org/abs/1808.01400.

109. *Sharma Tushar, Spinellis Diomidis.* A survey on software smells // *Journal of Systems and Software.* — 2018. — Vol. 138. — Pp. 158–173.

110. A survey of machine learning for big code and naturalness / Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, Charles Sutton // *ACM Computing Surveys (CSUR).* — 2018. — Vol. 51, no. 4. — Pp. 1–37.

111. Empirical standards for software engineering research / Paul Ralph, Nauman bin Ali, Sebastian Baltes et al. // *arXiv preprint arXiv:2010.03525.* — 2020.

112. *Khomh Foutse, Di Penta Massimiliano, Gueheneuc Yann-Gael.* An exploratory study of the impact of code smells on software change-proneness // 2009 16th Working Conference on Reverse Engineering / IEEE. — 2009. — Pp. 75–84.

113. Sourcerercc: Scaling code clone detection to big-code / Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko et al. // Proceedings of the 38th international conference on software engineering. — 2016. — Pp. 1157–1168.

114. Code and named entity recognition in stackoverflow / Jeniya Tabassum, Mounica Maddela, Wei Xu, Alan Ritter // *arXiv preprint arXiv:2005.01634*. — 2020.

115. *Sharnagat Rahul*. Named entity recognition: A literature survey // *Center For Indian Language Technology*. — 2014. — Vol. 1. — P. 1.

116. *Bikel Daniel M, Schwartz Richard, Weischedel Ralph M*. An algorithm that learns what's in a name // *Machine learning*. — 1999. — Vol. 34. — Pp. 211–231.

117. Conditional random fields: Probabilistic models for segmenting and labeling sequence data / John Lafferty, Andrew McCallum, Fernando Pereira et al. // Icml / Williamstown, MA. — Vol. 1. — 2001. — P. 3.

118. *McCallum Andrew, Li Wei*. Early results for named entity recognition with conditional random fields, feature induction and web-enhanced lexicons // Proceedings of the seventh conference on Natural language learning at HLT-NAACL 2003. — 2003. — Pp. 188–191.

119. *Cortes Corinna, Vapnik Vladimir*. Support-vector networks // *Machine learning*. — 1995. — Vol. 20. — Pp. 273–297.

120. *McNamee Paul, Mayfield James*. Entity extraction without language-specific resources // COLING-02: The 6th Conference on Natural Language Learning 2002 (CoNLL-2002). — 2002.

121. Neural architectures for named entity recognition / Guillaume Lample, Miguel Ballesteros, Sandeep Subramanian et al. // *arXiv preprint arXiv:1603.01360*. — 2016.

122. *Batbaatar Erdenebileg, Ryu Keun Ho*. Ontology-based healthcare named entity recognition from twitter messages using a recurrent neural network approach // *International journal of environmental research and public health*. — 2019. — Vol. 16, no. 19. — P. 3628.

123. Attention is all you need / Ashish Vaswani, Noam Shazeer, Niki Parmar et al. // *Advances in neural information processing systems*. — 2017. — Vol. 30.

124. Evaluating pretrained transformer-based models on the task of fine-grained named entity recognition / Cedric Lothritz, Kevin Allix, Lisa Veiber et al. // 28th International Conference on Computational Linguistics. — 2020.

125. *Xi Zhiheng, Chen Wenxiang, Guo Xin et al.* The Rise and Potential of Large Language Model Based Agents: A Survey. — 2023. https://arxiv.org/abs/2309.07864.

126. *Xia Chunqiu Steven, Deng Yinlin, Dunn Soren, Zhang Lingming.* Agentless: Demystifying LLM-based Software Engineering Agents. — 2024. https://arxiv.org/abs/2407.01489.

127. *Liu Yizhou, Gao Pengfei, Wang Xinchen et al.* MarsCode Agent: AI-native Automated Bug Fixing. — 2024. https://arxiv.org/abs/2409.00899.

128. *Tang Xunzhu, Kim Kisub, Song Yewei et al.* CodeAgent: Autonomous Communicative Agents for Code Review. — 2024. https://arxiv.org/abs/2402.02172.

129. *Pornprasit Chanathip, Tantithamthavorn Chakkrit.* Fine-tuning and prompt engineering for large language models-based code review automation // *Information and Software Technology.* — 2024. — Vol. 175. — P. 107523.