DOI: 10.15514/ISPRAS-2025-37(4)-22

Автоматическая генерация рецензий к коду: эволюция инструкций и интеллектуальная фильтрация

В.В.Качанов, ORCID: 0000-0002-9371-6483 <*vkachanov@ispras.ru*>

Институт системного программирования им. В.П. Иванникова РАН,
Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.

Аннотация. В работе рассматривается задача автоматической генерации рецензий к исходному коду. Предложен и апробирован подход на основе больших языковых моделей, значительно повышающий практическую ценность рецензий кода за счет фокуса на применимости, настройки инструкций и интеллектуальной фильтрации. Рассмотрено систематическое инкрементальное применение следующих стратегий: пошаговое рассуждение, структурированный вывод, расширение контекста, обучение с примерами. Использован подход со специальной инструкцией к языковой модели для интеллектуального ранжирования и фильтрации нерелевантных комментариев. Использование предложенного подхода к конструированию инструкций к языковой модели позволило улучшить точность применимых рецензий в 2.5 раза по сравнению с базовой до 37%.

Ключевые слова: автоматизация рецензирования исходного кода; большие языковые модели; качество набора данных.

Для цитирования: Качанов В.В. Автоматическая генерация рецензий к коду: эволюция инструкций и интеллектуальная фильтрация. Труды ИСП РАН, том 37, вып. 4, часть 2, 2025 г., стр. 117–132. DOI: 10.15514/ISPRAS–2025–37(4)–22.

Automatic Code Review Generation: Instruction Evolution and Intelligent Filtering

V.V. Kachanov, ORCID: 0000-0002-9371-6483 *<vkachanov@ispras.ru>*

Ivannikov Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Abstract. Code review is essential for software quality but labor-intensive in distributed teams. Current automated comment generation systems often rely on evaluation metrics focused on textual similarity. These metrics fail to capture the core goals of code review, such as identifying bugs, security flaws, and improving code reliability. Semantically equivalent comments can receive low scores if worded differently, and inaccurate suggestions can create confusion for developers. This work aims to develop an automated code review generator focused on producing highly relevant and applicable feedback for code changes. The approach leverages Large Language Models, moving beyond basic generation. The core methodology involves the systematic design and incremental application of sophisticated prompt engineering strategies. Key strategies include step-by-step reasoning instructions, providing the model with relevant examples (few-shot learning), enforcing structured output formats, and expanding contextual understanding. Crucially, a dedicated intelligent filtering stage is introduced: a LLM-as-a-Judge technique acts as an evaluator to rigorously rank generated comments and filter out irrelevant, redundant, or misleading suggestions before presenting results. The approach was implemented and tested using the Qwen/Qwen2.5-Coder-32B-Instruct model. Evaluation by original code authors demonstrated significant improvements. The optimal prompt strategy yielded a 2.5 times increase in the proportion of applicable reviews (reaching 37%) and a 1.6 times increase in good comments (reaching 61%) compared to a baseline. Providing examples enhanced comment quality, and the evaluator filter proved highly effective in boosting output precision. These results represent a substantial advance towards generating genuinely useful, actionable feedback. The approach significantly enhances the practical utility and user experience of automated code review tools for software developers by prioritizing relevance and applicability.

Keywords: code review automation; large language models; dataset quality.

For citation: Kachanov V.V. Automatic Code Review Generation: Instruction Evolution and Intelligent Filtering. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 4, part 2, 2025, pp. 117-132 (in Russian). DOI: 10.15514/ISPRAS-2025-37(4)-22.

1. Введение

В условиях современных распределённых команд разработки программного обеспечения, проведение рецензирования исходного кода (code review) продолжает оставаться крайне важным аспектом обеспечения качества, несмотря на свою внушительную трудоемкость. Традиционный подход, требующий тщательного ручного анализа каждого набора изменений или патча, неизбежно сопряжён со значительными временными затратами и когнитивной нагрузкой на разработчиков. В свете этих вызовов, научное и инженерное сообщество проявляет растущий интерес к системам автоматической генерации комментариев к коду. Эти системы рассматриваются как перспективный инструмент для оптимизации процесса рецензирования, повышения его эффективности и масштабируемости.

В начальной фазе исследований данной проблематики, доминирующую роль занимали подходы, основанные на методах информационного поиска, которые фокусировались преимущественно на извлечении подходящих комментариев из обширных предварительно собранных репозиториев существующих рецензий [1]. Однако, достижения в области глубокого обучения (deep learning) инициировали значительную трансформацию парадигмы, сместив акцент исследований в сторону генеративных моделей искусственного интеллекта. Работа коллектива исследователей под руководством Tufano [2], с архитектурой трансформера T5 [3] в основе, получила дальнейшее развитие и была усовершенствована благодаря внедрению стратегий предварительного обучения моделей на масштабных гибридных корпусах, сочетающих фрагменты программного кода с пояснениями на

естественном языке [4]. Следующее поколение моделей, таких как CodeReviewer [5] и AUGER [6] интегрировали в свою структуру специализированные механизмы предварительного обучения (pretrain), целенаправленно настроенные именно на задачу рецензирования кода для повышения точности и релевантности. Альтернативным подходом оставался метод информационного поиска, продолживший свое развитие в работе CommentFinder [7], предлагающей оптимизированные решения на базе поисковых алгоритмов. В совокупности, всё многообразие этих исследовательских усилий наглядно иллюстрирует сложный характер современных подходов к решению актуальной проблемы автоматизации в инженерной практике создания программных продуктов.

Тем не менее, несмотря на очевидный прогресс в области генерации рецензий к исходному коду, существующие методологии оценки качества автоматически создаваемых комментариев зачастую вызывают серьёзные вопросы. Практика показывает, что они в значительной степени опираются на поверхностные текстовые метрики (такие, как BLEU и ROUGE), которые, по сути, игнорируют истинные, фундаментальные цели процесса рецензирования кода – а именно, выявление потенциальных дефектов, уязвимостей безопасности, архитектурных просчётов и, в конечном итоге, реальное улучшение качества и надёжности программного кода [5, 8]. Более того, установлено, что недостаточно точные комментарии не просто бесполезны, но и потенциально вредны, так как способны вносить путаницу в процесс разработки и дезориентировать программистов [8-9]. Ключевым недостатком стандартных метрик является и то, что они систематически "штрафуют" и занижают оценку технически корректных и семантически эквивалентных комментариев, если их лексическая форма отличается от предоставленного эталонного варианта. Именно поэтому, центральной целью настоящего исследования является разработка генератора комментариев для анализа патчей и коммитов, основанного на использовании больших языковых моделей (LLM) и сфокусированного в первую очередь на обеспечении высокой степени релевантности и практической применимости даваемых рекомендаций. Этот акцент обусловлен убеждением, что метрики применимости (включая точность обнаружения реальных ошибок, оценку практической полезности предлагаемых исправлений, уровень ложных срабатываний) гораздо адекватнее отражают подлинную цель рецензирования – улучшение качества исходного кода. В рамках данной работы также освещаются: архитектура предложенной системы, поэтапная эволюция и настройка используемых инструкций/промптов (prompt engineering), механизмы фильтрации выходных данных для повышения их качества и результаты комплексной оценки эффективности генератора на репрезентативных наборах реальных производственных данных.

2. Существующие решения

Был проведен анализ существующих подходов к генерации рецензий к исходному коду. Авторы работы [10] описывают процесс генерации комментариев при помощи LLM, а также применение подхода LLM-as-a-Judge для автоматической оценки качества рецензий и сравнения нескольких. Авторы использовали подходы, называемые ‘few-shot’ и ‘reasoning’ [11]. 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 [12] описывается промышленная система для автоматизированного code review в компании ByteDance – BitsAI-CR. Выделяются 4 базовые категории, 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 [13].

В работе [8] 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 [9] 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. Предлагаемые методы

3.1 Базовая инструкция

В работе [10] 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 генерируемая рецензия – достаточно длинная. Также при отсутствии 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.

3.2 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) [14-15].

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 [16] представлены методы интеграции структурированного вывода, реализованные на базе библиотеки Pydantic [17]. 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.

3.3 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:

```
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
```

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:

1. Предложения по улучшению, уже реализованные в анализируемом коде, например, добавить работу с контекстным менеджером with open(...), при явном его наличии;
2. Предложения добавить избыточные проверки, например, проверки наличия ключей в словарях там, где это гарантировано контекстом;
3. Предложения вставить каждый вызов метода в операторы перехвата исключений (try-except), игнорируя принцип разумной достаточности;
4. Предложения, противоречащие проектным практикам (например, замена метода ‘print’ на метод ‘logging’ в проектах, где такая библиотека логирования не используется);
5. Ложное определение дублирования с последующими рекомендациями по «оптимизации повторяющихся вызовов»;
6. Генерация комментариев с неявной или неочевидной мотивацией, затрудняющей понимание разработчиком цели предложенных изменений;
7. Рекомендации по изменению устоявшихся практик проекта, не подлежащих рефакторингу по соображениям согласованности или усилий.

3.4 Уточнение требований

В работе BitsAI [12] 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.

3.5 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) [18] выполняется подбор нескольких примеров, наиболее похожих на заданный, по заранее определенной мере, например, косинусному расстоянию между векторными представлениями. В работе [7] для этой задачи применялся подход с подсчетом косинусного расстояния между векторами, полученными из тел функций с помощью метода упрощенного представления текста CountVectorizer [19]. Для решения этой задачи был использован подход, описан в работе [7], который был модернизирован

следующим образом: а) замена метода CountVectorizer на схожий подход Bag-of-Words без учета количества слов (токенов) в участке кода, б) замена косинусного расстояния на среднее гармоническое между векторными представлениями, в) использование взвешенной функции SequenceMatcher, которая учитывает больший вклад токенов, попавших в область, выделенную автором для рецензии. Предложенные изменения показали значительный прирост качества определения схожих методов с учетом выделенных участков на закрытом наборе тестов.

Кроме того, важно иметь относительно чистый набор рецензий, которые будут подаваться в качестве примеров [8], такие как "Done" или "что это такое?" могут только ухудшить качество итоговой генерации. Для решения этой задачи использовался классификатор комментариев к исходному коду [20]. 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.

Полученная система автоматического рецензирования с автоподбором примеров значительно превосходит версию с двумя фиксированными примерами, описанными выше. При этом всё ещё остаются ситуации избыточности предложений, сомнительных улучшений, которые в конкретном контексте не нужны или неприменимы, лишние проверки, попытки исправить несуществующий ошибку.

3.6 Дополнительная фильтрация

В работе [10] 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 [21]. В исследовании [22] 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 [10] 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 [22], 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 [23].

4. Результаты

На рис. 1 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.

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. 1. Примеры рецензий и их ручной разметки.
Fig. 1. Examples of review and its manual labels.

Рис. 2 демонстрирует репрезентативный пример кода и соответствующих рецензий, сгенерированных моделью при использовании различных промптов, описанных в Разделе 3. 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 глобальных переменных. Рецензия, сформированная на основе базового запроса, содержит ошибочное утверждение о недостаточной выразительности регулярных выражений и их неспособности корректно обрабатывать целевые значения.

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. 2. Примеры рецензий при различных инструкциях.
Fig. 2. Examples of review from different prompts.

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 согласующуюся 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.

Для численной оценки применимости предложенных рецензий был использовано два тестовых стенда. Первый был составлен из закрытого набора 123 добавленных и измененных методов, извлеченных из 10 коммитов локального сервера Gerrit. Корпус включает 11 новых и 25 модифицированных файлов, реализованных на языке Python. 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. Среднее время обработки одного набора изменений составило примерно 75 секунд, а среднее время дополнительной оценки одной рецензии около 7 секунд.

В табл. 1 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 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. 1 для базовой инструкции.

В табл. 2 и 3 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 настройка 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.

Второй тестовый набор состоял из 1000 экземпляров открытого набора данных с примерами изменения кода и соответствующих им рецензий CRer, представленного в работе [10]. 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.

Табл. 1. Результаты оценки качества рецензий (без фильтров).
Table 1. Results of the reviews evaluation (without filters).

Метод	Общее количество	Ошибочные	Средние (шт, %)	Применимые (шт, %)
Базовая инструкция	150	95	33, 22	22, 14.7
Расширенный контекст	123	74	27, 21.9	22, 17.9
Улучшение требований	90	**48**	16, 17.8	**26, 28.89**
Фиксированные примеры	150	92	37, 24.6	21, 14
Автоподбор примеров	122	60	**38, 31.1**	24, 19.7

Табл. 2. Результаты оценки качества рецензий после мягкой фильтрации.
Table 2. Results of the reviews evaluation after soft filtering.

Метод	Общее количество	Ошибочные	Средние (шт, %)	Применимые (шт, %)
Базовая инструкция	106	61	26, 24.52	19, 17.9
Расширенный контекст	80	42	21, 26.25	17, 21.25
Улучшение требований	64	**29**	14, 21.87	**21, 32.81**
Фиксированные примеры	78	34	27, **34.6**	17, 21.79
Автоподбор примеров	92	40	**31**, 33.9	**21**, 22.83

Табл. 3. Результаты оценки качества рецензий после жесткой фильтрации.
Table 3. Results of the reviews evaluation after hard filtering.

Метод	Общее количество	Ошибочные	Средние (шт, %)	Применимые (шт, %)
Базовая инструкция	76	42	19, 25	15, 19.74
Расширенный контекст	51	22	14, 27.4	15, 29.41
Улучшение требований	54	**21**	13, 24.07	**20, 37.04**
Фиксированные примеры	47	24	12, 25.53	11, 23.4
Автоподбор примеров	66	26	**22, 33.3**	18, 27.27

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.

5. Заключение

В рамках данного исследования была осуществлена разработка и детальная проработка специализированных инструкций, направленных на эффективное решение комплексной задачи автоматизированной генерации содержательных и применимых рецензий для изменений исходного кода. Для достижения этой цели был проведен анализ и последующее практическое внедрение ряда современных подходов к конструированию промптов, доказавших свою эффективность в смежных областях обработки естественного языка и кода. К числу этих использованных методик относятся: пошаговые рассуждения, предоставление модели ограниченного количества релевантных примеров, структурированный вывод и интеллектуальная фильтрация отдельной инструкцией оценщика качества. Для проведения экспериментов была выбрана открытая модель Qwen/Qwen2.5-Coder-32B-Instruct. Проведена оценка применимости предложенных рецензий непосредственными авторами изменений. Наилучший из реализованных и протестированных подходов к составлению инструкций продемонстрировал значительный качественный скачок: доля применимых рецензий увеличилась в 2.5 раза, одновременно повысив процент неплохих комментариев в 1.6 раза, что позволило достичь точности в 37% и 61% соответственно. Анализ экспериментальных данных позволил выявить, что предоставление модели небольшого количества примеров повышает общее количество неплохих комментариев. Кроме того, было продемонстрировано, что использование отдельного ИИ эксперта-оценщика, может быть эффективно в качестве интеллектуального ранжирующего фильтра для рецензий. Основная цель такого фильтра – повышение точности и количества полезных комментариев в финальном выводе системы при одновременном отсеивании нерелевантных, избыточных или потенциально вводящих в заблуждение предложений, тем самым существенно повышая итоговую ценность и пользовательский опыт автоматизированного рецензирования для конечного пользователя.

Список литературы / References

[1]. Siow, J. K., Gao, C., Fan, L., Chen, S., & Liu, Y. (2020). Core: Automating review recommendation for code changes. 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 284-295.

[2]. Tufano, R., Pascarella, L., Tufano, M., Poshyvanyk, D., & Bavota, G. (2021). Towards automating code review activities. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 163-174.

[3]. Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Liu, P. J. (2020). Exploring the limits of transfer learning with a unified text-to-text transformer. Journal of machine learning research, 21(140), 2020, pp. 1-67.
[4]. Tufano, R., Masiero, S., Mastropaolo, A., Pascarella, L., Poshyvanyk, D., Bavota, G. (2022). Using pre-trained models to boost code review automation. Proceedings of the 44th international conference on software engineering pp. 2291-2302.
[5]. Li, Z., Lu, S., Guo, D., Duan, N., Jannu, S., Jenks, G., Sundaresan, N. (2022). Automating code review activities by large-scale pre-training. Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1035-1047.
[6]. Li, L., Yang, L., Jiang, H., Yan, J., Luo, T., Hua, Z., Zuo, C. (2022). AUGER: automatically generating review comments with pre-training models. Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1009-1021.
[7]. Hong, Y., Tantithamthavorn, C., Thongtanunam, P., Aleti, A. (2022). Commentfinder: a simpler, faster, more accurate code review comments recommendation. Proceedings of the 30th ACM joint European software engineering conference and symposium on the foundations of software engineering, pp. 507-519.
[8]. Liu, C., Lin, H. Y., Thongtanunam, P. (2025). Too noisy to learn: Enhancing data quality for code review comment generation. In 2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR), pp. 236-248.
[9]. Yu, Y., Zhang, L., Rong, G., Shen, H., Zhang, J., Yan, H., Tian, Z. (2024). Distilling Desired Comments for Enhanced Code Review with Large Language Models. Доступно по ссылке: https://arxiv.org/abs/2412.20340 .
[10]. Lu, J., Li, X., Hua, Z., Yu, L., Cheng, S., Yang, L., Zuo, C. (2025,). Deepcrceval: Revisiting the evaluation of code review comment generation. International Conference on Fundamental Approaches to Software Engineering, pp. 43-64.
[11]. Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Zhou, D. (2022). Chain-of-thought prompting elicits reasoning in large language models. Advances in neural information processing systems, vol, 35, pp. 24824-24837.
[12]. Sun, T., Xu, J., Li, Y., Yan, Z., Zhang, G., Xie, L., Sui, K. (2025). BitsAI-CR: Automated Code Review via LLM in Practice. Доступно по ссылке: https://arxiv.org/abs/2501.15134 .
[13]. Hui B, Yang J, Cui Z, Yang J, Liu D, Zhang L, Liu T, Zhang J, Yu B, Lu K, Dang K. (2024). Qwen2. 5-coder technical report. arXiv preprint arXiv:2409.12186.
[14]. Structured outputs: Everything you should know, Доступно по ссылке: https://humanloop.com/blog/structured-outputs, дата обращения 10.06.2025.
[15]. Structured outputs in llms: Definition, techniques, applications, benefits, Доступно по ссылке: https://www.leewayhertz.com/structured-outputs-in-llms/, дата обращения 10.06.2025.
[16]. How to return structured data from a model, Доступно по ссылке: https://python.langchain.com/docs/how_to/structured_output/, дата обращения 10.06.2025.
[17]. Pydantic, Доступно по ссылке: https://docs.pydantic.dev/latest/, дата обращения 10.06.2025.
[18]. Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Kiela, D. (2020). Retrieval-augmented generation for knowledge-intensive nlp tasks. Advances in neural information processing systems, vol. 33, pp. 9459-9474.
[19]. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O. Duchesnay, É. (2011). Scikit-learn: Machine learning in Python. The Journal of machine Learning research, vol. 12, pp. 2825- 2830.
[20]. 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, 2024г., стр. 83-85.
[21]. Llm-as-a-judge: a complete guide to using llms for evaluations, Доступно по ссылке: https://www.evidentlyai.com/llm-guide/llm-as-a-judge, дата обращения 10.06.2025.
[22]. Zheng, L., Chiang, W. L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Stoica, I. (2023). Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, vol. *36*, pp. 46595-46623.
[23]. LLM review prompts, Доступно по ссылке: https://github.com/vova98/llm_review_prompts, дата обращения 10.06.2025.

Информация об авторах / Information about authors

Владимир Владимирович КАЧАНОВ – аспирант кафедры интеллектуальных систем МФТИ (НИУ), программист отдела системного программирования Института системного программирования с 2021 года. Сфера научных интересов: программная инженерия, машинное обучение, обработка естественного языка.

Vladimir Vladimirovich KACHANOV – postgraduate student of the Department of Intelligent Systems at MIPT (National Research University), programmer of the Department of Systems Programming at the Institute of Systems Programming since 2021. Research interests: software engineering, machine learning, natural language processing.