Инженерный вестник Дона, №2 (2026)
ivdon.ru/ru/magazine/archive/n2y2026/10726

Генерация документации на основе графового представления кода и больших языковых моделей

Ю.Е. Карякин, А.А. Каспшицкий, Д.И. Паращук, Т.И. Карповская

Тюменский государственный университет, Тюмень

Аннотация: В статье рассматриваются проблемы генерации и актуализации документации программного обеспечения с использованием больших языковых моделей. Представлен обзор существующих подходов, включая суммаризацию кода, систем использующих подходы дополненной генерации, ассистентов встроенных в среду разработки, выявлены их ограничения в части потери архитектурного контекста и возникновения структурных галлюцинаций. Предложена концепция графо-дополненной системы документирования, где «источником истины» выступает направленный граф знаний о коде, построенный путем статического анализа кода и анализа библиотечных зависимостей. Описан алгоритм построения графа, включающий извлечение узлов, анализ байт-кода библиотек и классификацию семантических связей. Эффективность подхода подтверждена экспериментальным внедрением на промышленном микросервисе, где система продемонстрировала способность корректно восстанавливать контекст и генерировать содержательную документацию без искажения фактов.
Ключевые слова: автоматическое документирование, большие языковые модели, граф знаний, дополненная генерация текста, статический анализ, семантический поиск, векторное представление, микросервисная архитектура, интерфейс структуры программы, байт-код, техническая документация, структурный анализ.

Введение

Современная разработка программного обеспечения сталкивается с фундаментальной и хорошо известной проблемой: растущим разрывом между исходным кодом и сопровождающей его документацией. Темпы эволюции продукта часто опережают возможности поддерживать документацию в актуальном состоянии, в результате чего технические спецификации и описания требований устаревают. Это усугубляется не только быстрыми итерациями, но и накоплением неформальных договорённостей, которые отклоняются от первоначального архитектурного дизайна и фактической реализации продукта. Со временем эти «теневые изменения» закрепляются в поведении системы, увеличивая разрыв между кодом и его описанием.

Когда этот разрыв становится критическим, документация перестаёт отражать реальное состояние программного обеспечения. Она превращается во фрагментированный, устаревший артефакт, который разработчики часто воспринимают как бюрократическую обязанность, а не как живой компонент проекта. Это ведёт к накоплению технического долга, замедляет погружение новых инженеров, снижает архитектурную прозрачность и увеличивает вероятность регрессий при рефакторинге.

В последние годы растёт интерес к применению больших языковых моделей для автоматизации процессов документирования. Существующие работы охватывают несколько направлений:

– **суммаризация кода** (CodeBERT, CodeT5) [1, 2, 3];

– **диалоговые ассистенты** в средах разработки (GitHub Copilot, Cody);

– **системы дополненной генерации**[4, 5], позволяющие взаимодействовать с репозиториями на естественном языке;

– **графовые подходы** [6, 7], представляющие код как структуру для высокоуровневых рассуждений.

Несмотря на существенный прогресс, ключевой вопрос остаётся открытым: возможно ли создать систему документации, которая эволюционирует вместе с кодовой базой, оставаясь точной, синхронизированной и самоподдерживающейся без постоянного ручного вмешательства?

1. Обзор существующих подходов

Существующие работы в области автоматизации документирования кода можно классифицировать на пять основных направлений.

1. Суммаризация кода. Методы, генерирующие краткие описания отдельных фрагментов кода (например, функций или классов), в том числе с использованием трансформеров (CodeBERT, GraphCodeBERT) [8, 9, 10]. К

положительным сторонам данных решений можно отнести высокую точность локальных описаний, низкие требования к вычислительным ресурсам и возможность офлайн-обработки. Однако их существенным ограничением является использование исключительно синтаксической информации, из-за чего они не отражают архитектурный контекст и не описывают взаимодействие модулей.

2. ассистенты, интегрированные в среду разработки. К этой категории относятся GitHub Copilot, JetBrains AI Assistant и аналогичные инструменты. Преимуществом является помощь в реальном времени при написании и чтении кода с учетом локального контекста открытых файлов. Однако данные решения имеют существенные недостатки: модель ограничена размером контекстного окна и не видит кодовую базу целиком, кроме того, код передаётся на облачный сервер, что может противоречить политике безопасности в компании.

3. Автоматическая документация внешних интерфейсов. Подход обеспечивает высокую точность описания внешних контрактов (эндпоинтов, дата трансфер объектов, схем), тем не менее, ограничивается границами системы. Он эффективно отвечает на вопрос «что может сделать программа через интерфейсы», но не объясняет внутреннюю бизнес-логику и детали реализации («как это устроено»), оставляя «слепые зоны» в документации.

4. Дополненная генерация по кодовой базе. Системы, использующие векторный поиск для нахождения релевантных фрагментов текста и кода под запрос пользователя [4]. Ключевое достоинство таких систем — возможность задавать вопросы к большим репозиториям на естественном языке. Обратной стороной является высокий уровень шума при поиске, так как близость векторов не гарантирует архитектурную релевантность, а отсутствие структурного мышления и ограничения контекстного окна затрудняют анализ сложных цепочек вызовов.

5. Графовые представления. Использование направленных графов, где узлы — это сущности (классы, методы, сервисы), а ребра — связи (вызовы, наследование, потоки данных) [6, 7]. Этот метод обеспечивает глобальную видимость архитектуры, компактное представление контекста и поддержку многоходовых рассуждений, однако сопряжен с высокой технической сложностью построения графа и поддержания его синхронизации с кодом.

Сравнительный анализ существующих решений показывает прямую зависимость между полнотой контекста и сложностью реализации (таблица 1).

Таблица 1

Сравнительная таблица существующих подходов

№ п/п	Подход	Качество документации	Локальный контекст	Межсервисный контекст	Сложность реализации
1	Суммаризация	Локально высокое	Да	Нет	Низкая
2	Ассистенты среды разработки	Хорошее (в моменте)	Да	Частично	Средняя
3	Документация программных интерфейсов	Высокое (интерфейсы)	Частично	Нет	Средняя
4	Дополненная генерация	Вариативное	Да	Ограничено окном	Средняя
5	Графовый подход	Высокое (архитектурное)	Да	Да (полный)	Высокая

2. Концепция графо-дополненного фреймворка

На основе проведённого анализа предлагается смена парадигмы автоматического документирования. Вместо подходов, в которых генерация документации осуществляется напрямую из неструктурированного контекста исходного кода, вводится промежуточный уровень - направленный граф знаний о программной системе.

В предлагаемом фреймворке исходный код рассматривается как единственный источник истины. Задача генерации документации преобразуется из «текст-в-текст» в процесс интерпретации формализованного представления кода: языковая модель взаимодействует не с «сырыми» исходниками, а с их структурным и семантическим представлением.

Фреймворк опирается на строгое разделение ответственности между двумя слоями:

1. **Слой фактов**. Анализатор кода строит граф, содержащий сущности (узлы) и отношения между ними (рёбра). Этот слой не включает вероятностных допущений: если граф фиксирует зависимость или вызов, то он соответствует реально существующему фрагменту программы.

2. **Интерпретирующий слой**. Языковая модель выступает не как генератор, работающий с неструктурированным кодом, а как интерпретатор локального подграфа. Её задача — определить, какие зависимости релевантны текущему контексту, и преобразовать выделенный фрагмент графа в человекочитаемое описание.

Такой подход существенно снижает вероятность структурных галлюцинаций. Модель ограничена топологией предоставленного подграфа и, в отличие от классических подходов, не может порождать зависимости и вызовы, отсутствующие в графе анализа.

Рассмотрим первый слой. Фундаментом фреймворка является конвейер статического анализа кода, который преобразует исходный код и внешние зависимости проекта в единую структурированную модель. Построение графа выполняется в три этапа: анализ исходного кода проекта, анализ библиотек и связывание между ними.

На первом этапе система работает с синтаксическим деревом программы. Для каждого файла исходного кода выполняется обход, направленный на извлечение деклараций и локальных зависимостей.

Алгоритм построения графа исходного кода:

Шаг 1. Экстракция узлов. Парсер идентифицирует сущности языка (классы, интерфейсы, методы, поля, аннотации) и формирует узлы графа. Каждому узлу присваиваются уникальный идентификатор - полное квалифицированное имя и тип узла, например, CLASS, METHOD, ENDPOINT.

Шаг 2. Извлечение метаданных. Из синтаксической структуры извлекаются модификаторы доступа, аргументы, возвращаемые типы, документационные комментарии, позиция в файле исходного кода и дополнительные атрибуты, которые сохраняются как свойства узла.

Шаг 3. Регистрация локальных вызовов. Внутри тела методов фиксируются все обращения к другим символам (вызовы функций, использование полей, создание объектов). Эти зависимости сохраняются как «сырые» ссылки, так как целевой узел может находиться в другом файле или модуле.

Результатом работы алгоритма является множество узлов с описанными «сырыми» связями.

Современные программные системы опираются на значительное число сторонних зависимостей. Игнорирование этого слоя делает модель поведения системы неполной. Поэтому применяется отдельный анализ библиотечного окружения.

Алгоритм построения библиотечного графа:

Шаг 1. Анализ зависимостей. Система разбирает конфигурации сборки, строит дерево транзитивных зависимостей и определяет пути к артефактам.

Шаг 2. Сканирование скомпилированного кода. Поскольку исходный код библиотек часто отсутствует, анализируются скомпилированные классы. Извлекаются только публичные элементы (интерфейсы): имена классов, сигнатуры методов, типы параметров.

Шаг 3. Генерация «теневых» узлов. Для каждого публичного элемента формируется узел в графе. Это позволяет учитывать взаимодействия с элементами библиотек, даже если их внутренняя реализация не анализируется.

Этап построения библиотечного графа является ключевым для обнаружения интеграционных точек (HTTP-клиенты, Kafka-топики, маршруты Camel и др.).

На заключительном этапе происходит объединение полученных фрагментов в единый ориентированный граф.

Алгоритм связывания:

Шаг 1. Построение индекса символов. Все узлы — и собственного кода, и библиотек — индексируются по полному имени.

Шаг 2. Разрешение импортов. Анализируются секция «import», что позволяет сопоставлять короткие имена (User) с полными (com.example.domain. User).

Шаг 3. Материализация ребер. Все «сырые» ссылки сопоставляются с узлами в индексе:

Шаг 3.1. Если целевой символ найден в коде, формируется ребро «CALLS» между узлами;

Шаг 3.2. Если он обнаружен в библиотеке, создаётся связь между внутренним и внешним узлом.

Результатом работы алгоритмов является связный, направленный граф, в котором исходный код и используемые фреймворки образуют единую модель взаимодействия системы.

Рассмотрим типологию графа. Рёбра графа представляют собой ключевой механизм структурирования знаний о программе, отображая связи между компонентами. Система различает несколько классов связей, каждая из которых фиксирует определённый аспект поведения системы. Связи формируются на основе структурных и семантических признаков кода и библиотек. Выделяются следующие виды связей:

– **Структурные связи.** Отражают иерархию и внутреннюю композицию программных сущностей.

– **Связи вызовов.** Фиксируют точные точки взаимодействия между кодовыми элементами.

– **Интеграционные связи.** Отражают взаимодействие с внешними технологиями и компонентами.

– **Абстрактные семантические связи.** Формируются на основе распознавания паттернов. Примеры: преобразование вызова kafkaTemplate.send() в ребро «PRODUCES»; создание виртуальных узлов для топиков или внешних ресурсов.

Эта типология обеспечивает представление системы на разных уровнях. Структурные связи фиксируют архитектуру, связи вызовов - поведение на уровне исполнения, а интеграционные связи позволяют моделировать взаимодействие с инфраструктурой. В совокупности перечисленные типы связей образуют многоуровневую модель программной системы, пригодную для навигации, анализа и генерации документации.

Рассмотрим второй слой фреймворка. Запрос пользователя проходит через цепочку предобработки, где он нормализуется, с помощью языковых моделей из него извлекаются ключевые слова (имена классов и методов), а также генерируются переформулированные варианты запроса. На этом этапе система выполняет двухуровневый поиск: «точный» поиск по графу кода (таблицы node и edge) для нахождения конкретных сущностей и их

зависимостей, и «векторный» поиск по базе знаний (таблица chunk с pgvector) для нахождения смысловых совпадений. Найденные результаты объединяются, убираются дубликаты и фильтруются: приоритет отдается чанкам, содержащим точные совпадения ключевых слов, отсеивая менее релевантный «шум». Чанк — это базовый информационный блок, содержащий совокупность текстовых (исходный код, описание) и структурных признаков сущности.

На финальном этапе формируется обогащенный контекст: в него сначала помещаются точно найденные узлы графа и их соседи, а затем — топ-результаты векторного поиска. Этот контекст вместе с историей диалога отправляется в языковую модель с инструкцией отвечать строго по предоставленным данным и указывать идентификаторы источников. Результат возвращается клиенту в текстовом формате обмена данными (JavaScript Object Notation — JSON), который содержит текстовый ответ, список использованных источников (с кодом и метаданными) и отладочную информацию.

3. Результаты

Реализация предложенного фреймворка охватывает все основные компоненты и демонстрирует его применимость на практике. Система построена как модульный конвейер, включающий три вида анализа: исходного кода, библиотечного окружения и интеграционных зависимостей.

Реализованы оба этапа формирования графа: анализ исходных файлов и анализ зависимостей. Для исходного кода применяется статический анализ на основе парсера структуры интерфейсов программы, обеспечивающий извлечение деклараций, структурных элементов и локальных вызовов. Для библиотек используется байткод-анализ через инструментарий манипулирования байткодом, позволяющий извлекать публичные контракты и интеграционные точки. На данных этапах формируются все типы узлов

графа: классы, методы, поля, интерфейсы, аннотированные сущности и узлы библиотек. Каждая связь отражает реально существующую зависимость, обнаруженную в коде или библиотеках.

Интеграция графов внешних зависимостей и графа исходного кода обеспечивает единое представление системы. Реализован алгоритм разрешения ссылок, включающий индексацию символов по полному имени, разрешение импортов и материализацию рёбер. Особое внимание уделено интеграционным вызовам: HTTP-endpoint, Kafka-топики, Camel-маршруты моделируются как полнофункциональные узлы графа.

Для каждого узла кода формируется одно представление в виде чанка (chunk). В текущей реализации чанки включают:

– исходный фрагмент кода или документацию;

– структурированные метаданные об узле;

– связи и зависимости;

– векторное представление для последующего поиска.

Для построения векторного представления реализован модуль embedding-вычислений и хранения в pgvector-индексах.

Генерация документации узла реализована как двухэтапный процесс. Сначала строится техническое описание компонента, за это отвечает модель «qwen2.5-coder:14b», затем оно адаптируется в человекочитаемую форму, за это отвечает модель «qwen2.5:14b-instruct». Оба этапа используют локальный подграф и структурированные подсказки, полученные из графа знаний.

Система предоставляет интерфейс для интерактивного взаимодействия (ответов на вопросы) по кодовой базе. Поиск выполняется не по текстовым фрагментам, а по семантическим представлениям графа, что обеспечивает более точные результаты.

Для оценки работоспособности и практической значимости фреймворк был опробован на реальном, промышленном сервисе распределения

исполнителей, реализованном на Spring Boot и Kotlin. Проект представляет собой микросервис, в котором 121 исходный файл в 86 директориях, общий размер репозитория — 812 КБ.

В результате работы конвейера статического анализа для данного сервиса был построен связный граф кода следующего масштаба:

– количество узлов: 417;

– количество рёбер: 746, из которых 414 структурных и 332 связи вызовов.

Сформированная модель знаний не только фиксирует статический каркас приложения, но и успешно агрегирует поведенческий контекст, создавая необходимый фундамент для последующей генерации документации средствами языковых моделей.

При обращении к фреймворку его ответ отражал реальную роль класса в работе сервиса и был сформирован на основе найденных узлов. Данный эксперимент подтверждает, что графо-ориентированный подход успешно восстанавливает контекст, определяет функциональную роль элемента (с помощью LLM) и формирует корректное объяснение, соответствующее актуальному состоянию кода.

Заключение

В работе представлен и реализован новый подход к автоматической генерации и актуализации технической документации, основанный на концепции графо-дополненного фреймворка. Ключевой особенностью предложенного метода является отказ от прямой обработки неструктурированного текста («текст-в-текст») в пользу использования графа знаний о коде как промежуточного структурного представления и единственного «источника истины».

Разработанный алгоритм построения графа, включающий статический анализ исходного кода, анализ байт-кода библиотечных зависимостей и

классификацию семантических связей, позволил создать многоуровневую модель программной системы. Это обеспечило преодоление главных ограничений существующих систем дополненной генерации и ассистентов основанных на больших языковых моделях - потери архитектурного контекста и возникновения структурных галлюцинаций. Разделение ответственности между детерминированным слоем фактов и интерпретирующим слоем гарантировало, что генеративная модель оперирует только реально существующими зависимостями и вызовами.

Экспериментальная апробация фреймворка на промышленном микросервисе (Spring Boot/Kotlin, 121 файл, 417 узлов) подтвердила эффективность гибридного поиска, объединяющего точный обход графа и векторный поиск по семантическим чанкам. Полученные результаты демонстрируют, что система способна корректно восстанавливать функциональную роль компонентов, выявлять неочевидные интеграционные связи и формировать содержательную документацию, синхронизированную с актуальным состоянием кодовой базы.

Таким образом, предложенный подход предоставляет возможность создания самоподдерживающихся систем документирования, которые эволюционируют вместе с программным продуктом, существенно снижая технический долг и повышая прозрачность архитектуры для разработчиков.

Исследование выполнено при поддержке Министерства науки и высшего образования Российской Федерации в рамках проекта «Фундаментальные проблемы методики разработки и связанного с ней правового и этического регулирования в сфере применения систем и моделей искусственного интеллекта» (FEWZ-2024-0052).

Литература (References)

1. Wang Y., Wang W., Joty S., et al. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation //

Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP). 2021, pp. 8696–8708. DOI: 10.18653/v1/2021.emnlp-main.685.

2. Zhang C., Wang Q., Zhou Q., et al. A Survey of Automatic Source Code Summarization // Symmetry. 2022. Vol. 14(3). P. 471. DOI: 10.3390/sym14030471.
3. Guo D., Lu S., Duan N., et al. UniXcoder: Unified Cross-Modal Pre-training for Code Representation // Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). 2022, pp. 7212–7225. DOI: 10.18653/v1/2022.acl-long.499.
4. Lewis P., Perez E., Piktus A., et al. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks // Advances in Neural Information Processing Systems (NeurIPS), 2020, vol. 33, pp. 9459-9474. URL: proceedings.neurips.cc/paper/2020/hash/6b493230-Paper.pdf.
5. Zhang F., Chen B., Zhang Y., et al. RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation // Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing. 2023, pp. 2471-2484. DOI: 10.18653/v1/2023.emnlp-main.151.
6. Pan S., Luo L., Wang Y., et al. Unifying Large Language Models and Knowledge Graphs: A Roadmap // IEEE Transactions on Knowledge and Data Engineering, 2024, vol. 36, no. 2. URL: ieeexplore.ieee.org/document/10321159.
7. Du Y., Yu Z. Pre-training Code Representation with Semantic Flow Graph for Effective Bug Localization // ESEC/FSE ’23: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2023, pp. 560-572. DOI: 10.1145/3611643.3616266.

8. Guo D., Ren S., Lu S., et al. GraphCodeBERT: Pre-training Code Representations with Data Flow // ICLR 2021: The Ninth International Conference on Learning Representations. 2021. URL: openreview.net/forum?id=jLoC4ez43PZ.
9. Zeng Z., Tan H., Zhang H., et al. An Extensive Study on Pre-trained Models for Program Understanding and Generation // ISSTA '22: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. 2022, pp. 39-51. DOI: 10.1145/3533767.3534390.
10. Li J., et al. Code Summarization Based on Graph Embedding and Pre-training // SEKE 2023: Proceedings of the 35th International Conference on Software Engineering and Knowledge Engineering. 2023, pp. 120-126. DOI: 10.18293/SEKE2023-018.

Дата поступления: 10.12.2025

Дата публикации: 24.01.2026