DOI: https://doi.org/10.26565/2304-6201-2025-68-03

УДК (UDC) 004.8

Костюченко Альбіна Валентинівна *магістрант кафедри системного програмування та спеціалізованих комп'ютерних систем, "Київський політехнічний інститут імені Ігоря Сікорського", 03056, Україна, Київ, вул. Політехнічна, 14-а*
e-mail: albina.kostyuchenko03@gmail.com
https://orcid.org/0009-0004-7382-7209

Петрашенко Андрій Васильович *к.т.н., доцент кафедри системного програмування та спеціалізованих комп'ютерних систем, Національний технічний університет України "Київський політехнічний інститут імені Ігоря Сікорського", 03056, Україна, Київ, вул. Політехнічна, 14-а*
e-mail: petrashenko@gmail.com*;*
https://orcid.org/0000-0003-0239-1706

Метод генерації опису програмного коду з використанням моделі штучного інтелекту

Актуальність. Тема є актуальною, оскільки у даний час існує багато великих проектів, які розробляються протягом тривалого періоду часу та потребують підтримки та розуміння коду без пояснень. Швидкий розвиток технологій та необхідність постійної розробки нових функцій і підтримки вже існуючих потребує постійного оновлення документації. Написання хорошої документації є цінною навичкою, що потребує досвіду, концентрації та розуміння структури проекту. Як наслідок, велика кількість розробників вважають процес написання документації важким і думають, що час, витрачений на це, можна було б використати більш продуктивно. Саме тому є попит на сервіси, які допомогають автоматизувати цей процес.
Мета. Метою даної роботи є підвищення ефективності автоматизованої генерації програмної документації. У рамках виконання даного завдання було опрацьовано необхідний теоретичний матеріал, вивчено уже існуючі рішення даної проблеми та розроблено і реалізовано власний новий метод генерування опису програмного коду, який більш точно визначав призначення фрагментів коду, чітко розумів структуру та залежності між його складовими.
Методи дослідження. Дослідження базується на аналізі літератури, статистичних методах, а також методах машинного навчання та інтелектуального аналізу даних. Зокрема було використано методи синтаксичного аналізу коду та побудови абстрактного синтаксичного дерева (AST), метод формування навчального корпусу, методи навчання та донавчання трансформерних та графових моделей. Для оцінки переваг донавченої моделі було застосовано метод порівняльного моделювання та автоматизованої оцінки якості тексту (у даному випадку BERTScore).
Результати. Донавчання моделі T5 на спеціалізованому наборі даних із прокоментованим кодом у поєднанні з лексичним аналізом дозволило підвищити якість генерації приблизно на 4% за метрикою F1 порівняно з базовою моделлю. Це свідчить про те, що адаптація моделі до конкретної доменної задачі є ефективною та здатною суттєво покращити результат.
Висновки. На основі зібраних даних було запропоновано власний підхід покращення якості генерації опису коду з використанням донавченої моделі T5 та створеної моделі GNN з подальшою реалізацією, що і є результатом досліження. Запропонована система поєднує кращі практики синтаксичного аналізу, графового моделювання і трансформерної генерації, забезпечуючи практично застосовне рішення для автоматичного створення документації. Можна стверджувати, що поєднання «seq2seq» моделей, методів токенізації та адаптації великих трансформерів, а також аналізу коду через GNN і структурні AST-представлення забезпечує комплексний підхід до автоматизації роботи з кодом, дозволяючи поєднувати локальні й глобальні контексти, швидко адаптувати модель під специфічні задачі та ефективно генерувати змістовні коментарі та документацію. Такий інтегрований підхід має потенціал для подальшого розвитку систем штучного інтелекту у сфері автоматичного аналізу коду, підвищення продуктивності розробників та забезпечення якості програмного забезпечення. Результати дослідження можуть бути застосовані на практиці для швидкого та ефективного створення документації до розроблюваного програмного забезпечення та великих проектів мовою Python.
Ключові слова: *машинне навчання, T5, GNN, генератор опису коду, навчання моделі, обробка природної мови, документація, AST.*

Як цитувати: Костюченко А. В., Петрашенко А. В. Метод генерації опису програмного коду з використанням моделі штучного інтелекту. *Вісник Харківського національного університету імені В. Н. Каразіна, серія Математичне моделювання. Інформаційні технології. Автоматизовані системи управління*. 2025. вип. 68. С. 30-42. https://doi.org/10.26565/2304-6201-2025-68-03

How to quote: A. Kostiuchenko, and A. Petrashenko, “Method for generating source code description using an artificial intelligence model”, *Bulletin of V. N. Karazin Kharkiv National University, series Mathematical modelling. Information technology. Automated control systems*, vol. 68, pp. 30-42, 2025. https://doi.org/10.26565/2304-6201-2025-68-03

Вступ

У наш час сфера інформаційних технологій розвивається надшвидко. На даний момент Github, найбільший у світі веб-сервіс для зберігання та публікації коду, налічує більше 47 мільйонів публічних репозиторіїв. Проте, на жаль, не весь код, що існує, є зрозумілим та має чіткі та розгорнуті пояснення у вигляді коментарів, ще рідше зустрічається повноцінна, чітко структурована документація. І це є досить серйозною проблемою, оскільки над великими проектами часто працює не одна сотня розробників, котрі час від часу змінюються. Відсутність якісної документації підвищує кількість часу, необхідного для розуміння та редагування коду.

Саме тому існує програмне забезпечення, яке генерує документацію на основі коментарів у коді, назв та функціональності класів та методів або загального контексту. Найстаріша система для документування коду була створена ще у 1997 році, проте стрімко розвиватись ця галузь програмного забезпечення почала менше двадцяти років потому – з появою моделей-трансформерів. Певні програми створюють опис коду за шаблонами, інші використовують обробку природної мови для аналізу залежностей та функцій, які виконують ті чи інші частини коду.

Генератори за шаблонами, такі як Swagger для API або Doxygen для різних мов програмування, працюють за принципом аналізу структури коду та специфікацій і подальшого створення документації у стандартизованому форматі. Вони дозволяють швидко отримати базову документацію: наприклад, список методів, параметрів, типів даних і прості пояснення до функцій. Основні переваги таких систем – це висока швидкість, передбачуваність результату та мінімальна потреба у навчанні чи складних ресурсах. Однак ці підходи мають й істотні обмеження: вони не враховують семантику бізнес-логіки, не генерують інтелектуальні пояснення коду та обмежені тими шаблонами, які передбачені розробником генератора. Тобто документація часто є формальною, поверхневою і не допомагає зрозуміти, що насправді робить конкретний фрагмент коду.

Інший клас систем – це генератори на основі штучного інтелекту, наприклад, GitHub Copilot або Tabnine. Вони використовують великі трансформерні моделі, навчальні на масивних репозиторіях коду, і здатні пропонувати коментарі, завершення коду та навіть цілі функції на основі контексту. Переваги таких рішень очевидні: вони адаптуються до стилю коду, можуть пропонувати нестандартні рішення та інтелектуальні коментарі. Водночас вони мають ряд недоліків: висока обчислювальна складність, залежність від хмарних сервісів та великих навчальних наборів, ризик генерації помилкового або небезпечного коду, а також відсутність контролю над точністю та стилем коментарів. Крім того, такі системи є закритими і не дозволяють повністю кастомізувати процес генерації під конкретну доменну задачу.

Метою даної роботи є підвищення ефективності автоматизованої генерації програмної документації. Метод, застосований у даному проекті, поєднує переваги штучного інтелекту і параметроефективного дообучення, одночасно мінімізуючи недоліки обох попередніх методів. Використання моделі T5, донавченої методом LoRA на наборі фрагментів коду з коментарями, дозволяє генерувати семантично точні та структуровані коментарі, що враховують як синтаксис, так і логіку коду. На відміну від генераторів за шаблонами, створена ссистема не обмежена суворими правилами форматування і здатна створювати інтелектуальні пояснення. На відміну від загальних AI-сервісів, таких як Copilot, модель навчена на специфічному наборі даних, що дозволяє контролювати якість коментарів, адаптувати їх під конкретний стиль або проєкт, а також повністю працювати локально без залежності від сторонніх хмарних сервісів.

1. Існуючі рішення за темою

Найвідомішим генератором документації на даний момент є Doxygen – це система документування вихідних текстів, яка була вперше випущена 26 жовтня 1997 року і на даний момент підтримує десять мов програмування, серед яких PHP, Java, C, Fortran і VHDL. Система підтримує генерацію документації у форматах HTML, XML, LaTeX, RTF та man. Doxygen досить широко застосовується і використовується у таких проектах як Torque Game Engine, AbiWord, Mozilla, Crystal Space та інших.

У першу чергу програма аналізує файл конфігурації, котрий керує налаштуваннями проекту. Отримані налаштування зберігаються у файлі src/config.h у класі Config. Сам парсер, написаний за допомогою flex (Fast Lexical Analyzer Generator) – генератора лексичних аналізаторів, розбиває текст на окремі токени за допомогою вказаних регулярних виразів. Парсер находиться у окремій бібліотеці src/config.l.

Кожен параметр конфігурації має тип String, List, Enum, Int або Bool. Їхні значення глобально доступні через функції типу Config_getXXX(), де XXX – тип параметра. Наприклад, функція Config_getBool (GENERATE_TESTLIST) повертає вказівник на булеве значення True, якщо testlist доступний у файлі конфігурації.

Далі всі файли з вхідним кодом, які були вказані у конфігураційному файлі, за замовчуванням проходять обробку спеціалізованим препроцесором C, який створений за допомогою того ж flex і має кілька відмінностей від стандартного та знаходиться у файлі src/pre.l. По-перше, препроцесор за замовчуванням не розширює виклики макросів, хоча це можна налаштувати, по-друге – команди #include аналізуються, але вказані там файли не підключаються. Кожен файл викликається функцією preprocessFile(), а результат попередньої обробки додається до буферу символів у форматі «0x06 file name N; 0x06 preprocessed contents of file N».

Отримані файли обробляє аналізатор мови (файл src/scanner.l), який працює як кінцевий автомат з використанням flex. Задача аналізатора – це перетворення отриманого на попередньому кроці буфера символів у абстрактне синтаксичне дерево. Для всіх мов працює один великий сканер, що сповільнює роботу програми.

Після цього створюються словники вилучених класів, файлів, просторів імен, змінних, функцій, пакетів, сторінок і груп. Окрім створення словників, під час цього кроку обчислюються відносини (наприклад, відносини спадкування) між отриманими сутностями. Наявні у коді блоки коментарів зберігаються як рядки в сутностях, які вони документують.

Якщо увімкнено перегляд вихідного коду або якщо в документації зустрічаються фрагменти коду, запускається аналізатор вихідного коду. Синтаксичний аналізатор коду намагається зробити перехресне посилання на вихідний код, який він аналізує, із задокументованими сутностями. Він також виконує підсвічування синтаксису джерел. Вихідні дані записуються безпосередньо до вихідних генераторів. Основна точка входу для аналізатора коду parseCode()оголошена в src/code.h.

Після збору даних і перехресних посилань програма оформлює вихідні дані у різних форматах. Для цього використовуються методи, надані абстрактним класом OutputGenerator. XML-дані генеруються безпосередньо із зібраних структур даних. У майбутньому розробники планують використовувати XML як проміжну мову. Перевага наявності проміжної мови у тому, що незалежно розроблені інструменти, написані різними мовами, зможуть витягувати інформацію з вихідних даних XML.

Ще одним відомим інструментом, який у певному сенсі став стандартом для веб-розробки, є Swagger – інструмент для створення документації до API-сервісів. Він створює окрему сторінку з інтерактивною документацією, де відповідно до шаблону описано кожну із задокументованих функцій з прикладами використання та зразками вхідих даних, які функція отримує або створює. Такий підхід дозволяє зручно протестувати розроблений API одразу у браузері.

Інструкції для Swagger можна записувати у форматі JSON або YAML. Структура опису API відповідає стандартам OAS (OpenAPI Specification). Зокрема можна вказати специфікацію Swagger, назву, короткий опис та версію створеного API, окремі кінцеві точки (шляхи) та методи (операції) HTTP, які підтримуються цими кінцевими точками, методи аутентифікації, тощо.

Серверні елементи Swagger, такі як Swagger Codegen, Swagger Parser, написані мовою Java, інтерактивна документація Swagger UI створюється за допомогою мов Javascript, HTML та CSS. Swagger Editor, який дозволяє перевіряти специфікації на відповідність стандартам OpenAPI Specification та тестувати API у браузері, створений за допомогою ReactJS. Також для різних мов

та фреймворків існують бібліотеки Swagger Annotations, Flask-Swagger, FastAPI та інші, які дозволяють додавати анотації у програмний код.

Далі анотації після валідації автоматично перетворюються у OAS-файл. Swagger-сервери аналізують вхідний код, використовуючи статичний аналіз та рефлексію, щоб отримати інформацію про API. Потім всі зібрані дані перетворюються у інтерактивний односторінковий додаток, що складається з шаблонів. При цьому використовуються бібліотеки для рендерингу, такі як Handlebars.

2. Огляд досліджень та публікацій за темою

На одному з перших етапів дослідження було опрацьовано ряд публікацій, які так чи інакше розглядають використання моделей штучного інтелекту.

Зокрема у статті “Automatic comment generation for source code using external information by neural networks for computational thinking” автори пропонують підхід до автоматичної генерації коментарів до вихідного коду, орієнтований на освітнє середовище та розвиток «комп’ютерного мислення». У статті вирішуються декілька фундаментальних проблем, що пов’язані із розумінням коду та побудовою зрозумілих коментарів, і для цього пропонуються конкретні архітектурні рішення. Дана робота є важливим дослідженням у галузі обробки коду нейронними мережами, оскільки вона розглядає не лише сам код, але й додатковий контекст, який супроводжує програму, що є суттєво новим.

Першою ключовою проблемою, на яку звертають увагу автори, є недостатність інформації, що міститься безпосередньо у вихідному коді. У багатьох випадках синтаксис або структура програми не дозволяють однозначно визначити її призначення, і без додаткового контексту модель не здатна коректно інтерпретувати логіку роботи. Для подолання цього обмеження дослідники вводять концепцію зовнішньої інформації, що включає формулювання задачі, опис вхідних і вихідних даних, текст навчального завдання та інші пояснювальні матеріали. Цей текст конкатенується з програмним кодом і подається на вхід моделі типу encoder–decoder на базі LSTM. Таким чином, модель працює не лише з абстрактними синтаксичними структурами, а й з семантичним описом, що значно підвищує релевантність і змістовність згенерованих коментарів.

Автори багатьох попередніх робіт створювали програми, що генерують описи на рівні всієї функції чи файлу, однак іноді необхідно пояснювати роботу програми поетапно, на рівні окремих рядків або логічних блоків. Тому автори роблять акцент на деталізації коментарів і розробляють метод розбиття коду на структурні фрагменти – цикли, умовні оператори, декларації змінних та інші елементи. Для кожного такого блоку формується пара «код–коментар», що дозволяє навчати модель точковому поясненню дій програми, а не створенню узагальнених описів. Такий підхід робить результати моделі особливо корисними для початківців, оскільки пояснення стають локальними й прив’язаними до конкретних частин коду.

Ще однією задачею, яку автори вирішують у своїй роботі, є оцінювання якості згенерованих коментарів. Оскільки автоматичні метрики мають обмеження, дослідники використовують комбінований підхід. Для статистичного порівняння було застосовано метрики BLEU, ROUGE та METEOR, а для суб’єктивної оцінки зрозумілості, правильності та корисності отриманих пояснень залучили студентів і викладачів. Така двошарова система оцінювання забезпечує більш повне розуміння ефективності моделі в реальних освітніх сценаріях.

У технічному плані модель побудована на класичній архітектурі seq2seq з використанням LSTM в ролі енкодера та декодера, доповненої механізмом уваги. Це дозволяє декодеру фокусуватися на різних частинах коду під час генерації коментаря. Навчання моделі здійснюється за допомогою функції Cross-Entropy Loss, що є стандартним підходом у задачах генерування тексту. Об’єднання коду і зовнішнього тексту в один послідовний вхід дозволяє моделі будувати репрезентації, які враховують як синтаксичну, так і семантичну інформацію.

Таким чином, дана робота вирішує одразу кілька ключових проблем: нестачу семантичного контексту, відсутність деталізованих описів окремих блоків коду та складність об’єктивного оцінювання якості згенерованих коментарів, обмежуючись лише автоматичним аналізом і метриками.

Хоча запропонований підхід не враховує структурні залежності між модулями та не використовує абстрактні синтаксичні дерева чи графові моделі, він задає концептуальну основу, на якій можна будувати більш складні системи. У контексті даної магістерської роботи ця публікація служить важливим джерелом для порівняння, оскільки запропонована в дослідженні

система може бути розширена за рахунок використання AST, графових нейронних мереж та сучасних моделей трансформерного типу, що дозволяє досягти вищої точності та структурної узгодженості згенерованої документації.

Наступним опрацьованим дослідженням є “Retrieve and Refine: Exemplar-based Neural Comment Generation”, у якому автори пропонують інноваційний підхід до автоматичної генерації коментарів для коду, який поєднує нейронні трансформерні моделі із механізмами пошуку найбільш релевантних прикладі у великій базі коментованого коду. На відміну від традиційних seq2seq-моделей, які генерують коментар безпосередньо з програмного коду, автори використовують двоступеневу архітектуру типу retrieve-and-refine, що дозволяє моделі використовувати вже існуючі якісні коментарі як відправну точку для побудови нового тексту.

Автори стверджують, що моделі мають обмежені можливості щодо генерації осмислених коментарів лише на основі вихідного коду. Особливо це помітно, якщо програма містить складні логічні конструкції, або використовує доменну термінологію, яку модель без прикладів іноді інтерпретує неправильно. Для подолання цього обмеження дослідники впроваджують компонент retrieve, який за допомогою подібності коду знаходить приклад – існуючий фрагмент коду з якісним коментарем. Таким чином, система отримує не лише сам код, а й коментар, створений спеціалістом для схожої структури або логічного шаблону.

Другою важливою складовою системи є компонент refine, який дозволяє трансформеру змінювати, адаптувати та редагувати знайдений коментар під новий фрагмент коду. Автори застосовують модифіковану архітектуру типу енкодер–декодер, де енкодер отримує як сам код, так і коментар-прецедент. Декодер, у свою чергу, генерує новий коментар, поєднуючи інформацію з обох джерел. Тобто модель не створює текст з нуля, а відштовхується від уже наявного людського прикладу, що суттєво підвищує якість, змістовність та стильову природність згенерованих коментарів.

Третьою проблемою, на яку звертають увагу автори, є недостатня структурна адекватність коментарів, згенерованих чистими нейронними моделями. Моделі без прикладів часто дають надто загальні, неконкретні або неправильні пояснення. Підхід retrieve-and-refine використовує механізм стилістичного та семантичного наслідування: знайдений приклад надає релевантну лексику, структуру пояснення та термінологію. Завдяки цьому вихідний коментар має більше шансів бути зрозумілим, технічно точним і близьким до того, що пишуть реальні розробники.

Ще однією важливою частиною роботи є методика оцінювання моделі. Автори застосовують стандартні текстові метрики (BLEU, ROUGE), але також тестують модель на семантичних метриках, які краще відображають відповідність змісту, а не лише поверхневу схожість токенів. Крім того, вони проводять людську оцінку коментарів із залученням програмістів, які оцінюють такі параметри, як коректність, інформативність та відповідність коду. Результати демонструють, що підхід retrieve-and-refine суттєво перевершує моделі, які генерують текст без доступу до прецедентів.

З технічної точки зору архітектура моделі передбачає окремий етап обчислення схожості коду, для якого можуть використовуватися багатовимірні векторні представлення, ембеддинги або моделі з попереднім навчанням. Після того як найбільш схожі приклади відібрано, нейронна модель, заснована на трансформері, здійснює подальше уточнення, створюючи фінальний коментар. Навчання моделі виконується за допомогою крос-ентропійної функції втрат (Cross-Entropy Loss), що дозволяє покращити якість тексту на виході.

Що стосується графових нейронних мереж, ідея формування дерева проекту і його збереження у вигляді графа не є новою. Зокрема у доповіді «JGNN: Graph Neural Networks on Native Java» автор описує, як перетворити програмний проект у граф залежностей. Використовуються зв’язки імпорту, виклики функцій, структуру пакета та результати аналізу AST. Отриманий у результаті дослідження граф орієнтований і неоднорідний, оскільки містить кілька типів вузлів і зв'язків.

Розглянуто різні архітектури графованих нейронних мереж (GCN, GraphSAGE, GAT), націлювання ребер, детально пояснено механізм передачі повідомлень - властивість GNN, що дозволяє вузлу збирати інформацію від сусідів і розглядати контекст всього проекту.

Проте метою проекту було не документування коду, а пошук вразливостей. Створена модель навчена поділяти вузли графа на вразливі та безпечні. Було важливо, щоб GNN враховувала не тільки локальні характеристики файлу, але і його оточення: наявність небезпечних залежностей,

з'єднань, маршрутів поширення даних. Це особливо актуально для сучасних проектів з великою кількістю зовнішніх бібліотек.

У статті показано, що класичні інструменти обмежуються лінійним або модульним аналізом коду, у той час як графові методи фіксують залежності від файлу до файлу глобально. Автор наводить приклади вразливостей, які точно визначені через інтермодульний контекст. З цією метою формується група програмних проектів з помітними вразливостями. Вилучення залежностей і побудова графу залежностей відіграє важливу роль, що вимагає попереднього аналізу AST і метаданих проекту, таких як package.json, requirement.txt тощо.

3. Теоретичні відомості щодо основних складових системи

3.1. Особливості структури та навчання моделі T5

Модель Text-to-Text Transfer Transformer, також відома як T5, була створена Google Research у рамках проекту "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer". T5 використовується для обробки природної мови і використовує формат text-to-text, тобто вхідні і вихідні дані представлені у вигляді тексту, що спрощує роботу з моделлю. На даний момент існує кілька конфігурацій моделі, зокрема Small, що містить близько 60M параметрів, Base (близько 220 мільйонів), Large (близько 770 мільйонів), 3B (3 мільярдів) і 11B (більше 11 мільярдів параметрів).

Робочий процес T5 складається з двох етапів: попереднього навчання (pre-training) та тонкого налаштування (fine-tuning). Під час попереднього навчання модель навчається на величезному корпусі загальнодоступних текстових даних. Стандартна T5 навчена на корпусі C4, який складається з 745 ГБ фільтрованих текстів, очищених від нерелевантних даних, частин коду та дублікатів. Метою навчання є передбачення замаскованих токенів у вхідному тексті, подібно до популярної моделі BERT. Однак, на відміну від BERT, де дані кодуються в числа, T5 використовує підхід тексту в текст, де і вхід, і вихід є текстовими представленнями.

Тонке налаштування тренує модель під конкретну задачу, надаючи завдання в текстовому вигляді, наприклад, «summarize: [text]». Наприклад, існує модифікація CodeT5, навчена на масивах коду та натренована для вирішення задач розробки програмного забезпечення.

Для виконання задач машинного навчання, особливо глибоких нейронних мереж, компанія Google розробила TPUs (Tensor Processing Units) – спеціальні прискорювачі, кожен з яких складається з великої кількості ядер, кожен з яких виконує обчислення в форматі тензорів і використовує принципи паралельності даних та паралельності моделі. TPUs використовуються у тому числі і у моделі T5. Окрім цього, використовується алгоритм Adafactor – оптимізатор, який доповнює TPU і дозволяє більш ефективно використовувати пам’ять, що особливо актуально при роботі з великими моделями та за умов браку ресурсів.

Для донавчання моделі було використано метод LoRA, який дозволяє адаптувати великі трансформери, вводячи додаткові матриці низького рангу до ваг ключових шарів моделі. Зазвичай це роблять для ваг query та value у механізмі attention. Особливостями методу є те, що основні параметри моделі заморожуються, натомість додаються менші адаптерні матриці A та B (1). Вихід кожного шару змінюється таким чином:

$$W' = W + \Delta W = W + A \cdot B \quad (1)$$

де W – незмінні ваги оригінальної моделі, ΔW – навчувані матриці низького рангу.

Градієнти обчислюються лише для цих матриць, а решта моделі лишається незмінною. Це дозволяє моделі швидко підлаштовуватися під нову доменну інформацію при дуже малому додатковому обсязі пам’яті.

Такий метод може трохи поступатися у точності повному fine-tuning і не завжди підходить для завдань, де потрібна глибока модифікація всіх шарів моделі. Проте алгоритм LoRA компенсує свої недоліки тим, що у рази зменшує потребу в пам’яті у процесі навчання та забезпечує швидке тренування, оскільки оновлюються лише додаткові параметри. Такий метод підходить навіть для великих моделей.

Метод LoRA часто застосовують для адаптації моделі на невеликому наборі даних та загалом для донавчання для вузьконаправленої задачі. Наприклад, для створення спеціалізованих чат-ботів або донавчання під специфічний стиль тексту.

У даному випадку було доцільним використати саме цей підхід, адже наявні ресурси були обмеженими і створення моделі з нуля або навіть повний fine-tuning були б складними і не дали

бажаного результату. Модель T5-base вже навчена на великому корпусі очищених даних і еефективно працює з текстом, тому часткова її адаптація для задач коментування коду можна вважати раціональним з точки зору використання ресурсів. Завдяки замороженню основних параметрів T5 зберігається знання про загальні мовні закономірності та семантику коду, що дозволяє моделі генерувати коментарі більш природно та інформативно.

3.2. Особливості структури та навчання моделей типу GNN

У рамках цієї роботи була розроблена та навчена графова нейронна мережа (GNN) для аналізу залежностей між файлами та кодовими модулями з метою побудови діаграм імпорту та експорту.

Представлення коду формується таким чином: кожен файл або модуль розглядається як вузол графа, а ребра між вузлами відповідають залежностям, таким як імпорт, виклики функцій або посилання на класи. Кожне з'єднання може бути додатково закодовано з характеристиками, що відображають його тип і важливість. Кожен вузол має вкладення, які відображають його характеристики: наявність docstring, кількість функцій і класів, довжину коду і частоту виклику функцій. Ребра кодуються за допомогою двійкових індикаторів типів залежностей, що дає змогу розрізняти ключові імпорти та вторинні посилання.

Для архітектури моделі було обрано багатошарову Graph Attention Network (GAT), що складається з чотирьох шарів, кожен з яких агрегує інформацію від сусідніх вузлів за допомогою механізму уваги. Такий підхід дозволяє диференціювати важливість різних залежностей, забезпечуючи більш точне представлення структури проекту. оскільки він дозволяє призначати різні ваги сусіднім вузлам при агрегуванні інформації. Це особливо важливо в задачах аналізу коду, де деякі імпорти або ключові функції мають набагато більший вплив на структуру та поведінку програми.

Послідовність шарів GAT виконують агрегацію ознак сусідніх вузлів з подальшим застосуванням нелінійної функції активації ReLU. Між шарами використовується нормалізація BatchNorm і регуляризація, що забезпечує стабільність навчання і запобігає перенавчанню. На вихідному рівні модель генерує вкладення вузлів, які потім використовуються для побудови візуальної діаграми залежностей.

Навчання моделі проводилося за допомогою оптимізатора AdamW з швидкістю навчання 1e-3 і зниженням ваги 1e-5. Cross-Entropy Loss використовувався як функція втрат для класифікації вузлів за важливістю та ідентифікації ключових залежностей. Щоб запобігти перетренованості, на вузли та ребра наносили відсівання 0, 1. Для великих графів застосовувалося пакетування з torch_geometri at ataLoader, що дозволило ефективно тренувати модель на декількох файлах і сховищах одночасно.

Під час навчання були обрані оптимальні параметри: чотири шари ГАТ, 8 голівок уваги в кожному шарі, розмір вбудовувань вузлів 128 і відсівання 0, 1. Ця конфігурація дозволила моделям точно визначити ключові залежності, включаючи важливі імпорт і функції, а також сформувати схему проекту в графік, придатний для візуалізації та подальшої генерації документації. В результаті навчений GNN забезпечує високу точність передбачень залежностей і є ефективним інструментом для побудови структурованих схем коду, інтегрованих в систему автоматичної генерації документів, що використовується в цій роботі.

3.3. Роль ембедингів у запропонованому методі

Абстрактне синтаксичне дерево (AST) є ключовим представленням програмного коду, яке відображає структуру програми у вигляді ієрархії вузлів – операторів, функцій, класів, імпортів, викликів методів тощо. Однак саме по собі AST є символічним, а тому не може бути безпосередньо використане нейронними мережами. Для інтеграції цієї структурної інформації в GNN необхідно перетворити кожен вузол AST на числовий вектор. Такі вектори називаються AST Node Embeddings.

AST Node Embedding – це компактне векторне представлення вузла AST, яке кодує як його тип (наприклад, FunctionDef, Assign, ClassDef), так і його внутрішній зміст (імена змінних, ключові слова, літерали) та структурні характеристики - глибина у дереві, кількість дочірніх елементів, розмір блоку коду, тощо. Формування таких ембеддингів включає декілька етапів.

Спочатку, за допомогою ANTLR або стандартних засобів Python, вихідний код перетворюється на AST. Кожен вузол дерева класифікується за типом, а тип вузла проходить

через вбудований embedding-шар, який навчається подібно до словникових embedding у NLP. Це дозволяє моделі зрозуміти, що певні конструкції мови програмування мають схожі ролі.

Далі з вузла виділяється семантичний зміст: імена змінних, назви функцій, значення літералів або ключові слова. Ці токени обробляються за допомогою токенізатора, після чого їхні векторні представлення агрегуються – наприклад, через середнє або згортку. Таким чином модель отримує інформацію про текстовий зміст вузла.

Окрім цього, формуються числові ознаки, що описують структуру: кількість дочірніх вузлів, довжина тіла функції, кількість параметрів, рівень вкладеності. Це дозволяє врахувати ієрархічні властивості AST. Усі ці компоненти об’єднуються конкатенацією, після чого проходять через проекційний шар, який переводить інформацію в компактний вектор фіксованої розмірност. Такий вектор і є AST Node Embedding.

Отримані ембеддинги виконують подвійну роль у системі. По-перше, вони передаються до GNN, яка аналізує залежності між функціями, файлами або модулями, поширюючи інформацію через структуру графа. Це дозволяє моделі визначити важливі компоненти коду, знайти критичні зв’язки та сформувати узагальнене представлення структури програми.

По-друге, узагальнені графові ембеддинги інтегруються в модель T5 під час генерації коментарів або документації. Завдяки цьому T5 отримує не лише локальний контекст окремого фрагменту коду, але й розуміння його ролі всередині усього проекту. Такий підхід забезпечує значно більшу точність та інформативність коментарів порівняно з моделями, що працюють виключно з текстом коду.

Таким чином, використання AST Node Embeddings є центральним елементом і основною особливістю всієї розробленої архітектури, бо вони поєднують структурні та семантичні властивості програмного коду у формі, придатній для глибинного навчання, та забезпечують спільну роботу модулів GNN та T5 у рамках запропонованої системи.

4. Опис структури створеної системи

Система, що була розроблена, є комплексним рішенням для аналізу вихідного коду, побудови схем залежностей і автоматичної генерації коментарів до файлів проекту. Архітектура проекту включає кілька ключових компонентів, кожен із яких вирішує окремі завдання та інтегрується з іншими частинами системи.

Для реалізації завантаження коду було обрано модуль GitPython, оскільки він надає високорівневий інтерфейс для клонування репозиторіїв, роботи з гілками та вилучення окремих файлів. GitPython обертає функціональність Git у Python-об'єкти, що дозволяє інтегрувати завантаження репозиторіїв у єдину систему без необхідності прямої взаємодії з командним рядком. Такий підхід підвищує переносимість коду та спрощує його інтеграцію з іншими компонентами проекту, такими як модуль парсингу та графового аналізу.

Для отримання структурної інформації з вихідного коду застосовується ANTLR. За допомогою рекурсивних функцій чи спеціальних Visitor-класів ANTLR дозволяє обійти всі вузли дерева. Це дозволяє отримати структурну інформацію про програму: визначення функцій та класів, аргументи функцій, що використовуються змінні, блоки умовних операторів, цикли. З вузлів AST створюється послідовність токенів, придатна навчання моделі T5. Кожен токен є окремим елементом коду, який може бути зіставлений з коментарями, формуючи пари «код — коментар» для донавчання.

На основі AST можна виявити імпорти модулів та виклики функцій, що використовується для побудови графа залежностей файлів та модулів. Токенізований код, отриманий після розбору AST, використовується як вхідні дані для моделі T5. Перетворення в токени із збереженням структури програми дозволяє моделі «розуміти» контекст функцій та класів, що суттєво підвищує якість автоматичної генерації коментарів.

На основі AST формується граф залежностей, де вузли – це файли чи модулі, а ребра – їх взаємозв'язки через імпорти чи виклики функцій. Для обробки графа використовується Graph Neural Network (GNN) з архітектурою Graph Attention Network (GAT), яка агрегує інформацію від сусідніх вузлів та створює ембеддінги, що відображають контекст кожного вузла. Ці ембеддинги потім передаються T5 і використовуються для візуалізації.

Для візуалізації графа застосовується бібліотека Graph-tool, що дозволяє будувати наочні схеми залежностей. Алгоритми розміщення вузлів (наприклад, sfdp_layout) забезпечують зручне представлення графа для аналізу та документування проекту.

Модель T5-base, донавчена за допомогою методу LoRA, використовується для створення коментарів до вихідного коду. Вхідні дані моделі включають токенізований код, підготовлений на основі AST, та ембеддинги вузлів графа, що формуються GNN. Це дозволяє моделі враховувати як локальні синтаксичні особливості коду, так і глобальні залежності між файлами, що підвищує точність та інформативність створюваних коментарів. Для токенізації використовується SentencePiece, що оптимально працює з субсловними одиницями і дозволяє обробляти рідкісні ідентифікатори.

Навчання моделі проводиться з використанням PyTorch, оптимізація через AdamW, а функція втрат - Cross-Entropy Loss. Для генерації послідовностей використовується beam search, що дозволяє вибирати найімовірніші токени.

Для підвищення продуктивності та паралельної обробки великих репозиторіїв використовується Celery, що забезпечує асинхронне виконання завдань. Це дозволяє одночасно обробляти кілька репозиторіїв, будувати графи та генерувати коментарі без блокування основних потоків виконання.

Для представлення результатів роботи системи як документації застосовується бібліотека Mako. Вона дозволяє автоматично формувати HTML-документи, що містять візуалізовані графи залежностей із GNN, вихідний код та згенеровані коментарі, а також структуровані звіти щодо кожного модуля. Для взаємодії з користувачем використовується стек Flask та ReactJS.

5. Тестування системи

5.1. Про метрику BERTScore

BERTScore - це сучасна метрика для оцінки якості згенерованого тексту, що використовує попередньо навчені трансформерні моделей, таких як BERT, RoBERTa або їх багатомовні аналоги. На відміну від традиційних метрик на кшталт BLEU або ROUGE, котрі орієнтовані на точний збіг токенів між передбаченим та еталонним текстом, BERTScore враховує семантичну схожість слів та фраз. Це робить метрику більш придатною для завдань генерації природної мови, включаючи коментування коду та рефакторинг документа.

Спочатку і еталонний текст, і згенерований розбиваються на токени, що відповідають обраній моделі трансформера (у даному випадку моделі T5. Для кожного токена обчислюється ембедінг за допомогою моделі BERT або її аналога, який можна обрати. Ембедінги відображають сенс слова з урахуванням його оточення. Обчислення ембедінгу для кожного токена за допомогою трансформера вимагає GPU та великого обсягу пам'яті при великих наборах даних, проте це виправдовується вищою точністю аналізу.

Далі для кожного токена згенерованого тексту знаходиться токен з еталонного тексту, з яким його ембеддинг має найбільшу косинусну схожість (2).

$$sim\left(h_i^g, h_j^r\right) = \frac{h_i^g \cdot h_j^r}{\|h_i^g\| \cdot \|h_j^r\|} \tag{2}$$

На основі цих пар обчислюються три показники:

1) P (Precision) (3) – наскільки згенерований токен схожий на еталон:

$$P = \frac{1}{|x|}\sum_i \max_j sim\ (x_i, y_j) \tag{3}$$

2) R (Recall) (4) – наскільки еталонний токен «покритий» створеним токеном:

$$R = \frac{1}{|y|}\sum_j \max_i sim\ (y_j, x_i) \tag{4}$$

3) F1 (5) – гармонічне середнє значення між R та P:

$$F1 = 2 \cdot \frac{P \cdot R}{P + R} \tag{5}$$

де x_i – токени передбачення, y_j – токени еталону, $sim\ (x_i, y_j)$ – косинусна схожість між векторами.

Під час аналізу враховуються смислові збіги, а не лише ідентичні слова. Наприклад, "create user" та "add a new user" будуть оцінені як близькі за змістом, хоча структура коментаря та окремі слова відрізняються. Більше того, контекстні ембеддінги дозволяють відрізняти омоніми та правильно враховувати значення слова у конкретному місці. Проте тестування коротких фрагментів можуть давати нестабільні результати і це слід враховувати. Значення F1 показує

семантичний збіг, але не завжди зрозуміло, які конкретні помилки припустилася моделі генерації тексту.

Також передбачена можливість за потреби використовувати різні трансформери для обчислення ембеддингів, у тому числі багатомовні, що корисно для проектів із кодом та коментарями мовою, відмінною від англійської.

Загалом якість оцінювання залежить від обраної моделі BERT – слабка чи невідповідна модель може знизити точність метрики. Проте дослідження показують, що BERTScore краще корелює з оцінкою якості тексту, виставленої людьми, порівняно з BLEU і ROUGE, що може бути корисним, якщо тестування із залученням людей не передбачене або не є доцільним. У цілому можна стверджувати, що метрика ідеально підходить для автоматизованого тестування якості коментарів та подальшого аналізу результатів експериментів.

5.2. Отримані результати

Для оцінки результату донавчання власної моделі результати було порівняно з T5-base та mT5-base – базовим та багатомовним варіантами моделі T5, результати наведені у таблицях 1 та 2.

Таблиця 1. Порівняння результатів різних моделей
Table 1. Comparison of results of different models

Модель	Precision	Recall	F1
T5-base (базова)	0.851	0.857	0.854
mT5-base (багатомовна)	0.863	0.868	0.865
T5 (донавчена на коді з коментарями)	0.894	0.901	0.898

Таблиця 2. Порівняння еталонних та згенерованих коментарів
Table 2. Comparison of reference and generated comments

№	Фрагмент коду	Еталонний коментар	T5-base	mT5-base	T5 (донавчена)	BERTScore F1
1	def add(a, b): return a + b	Returns the sum of two numbers.	Adds numbers together.	Returns sum of values.	Computes and returns the sum of two input numbers.	0.93
2	for i in range(len(items)): process(items[i])	Iterates through the list and processes each element.	Loop over list.	Iterates items in list.	Loops through all items and processes each element.	0.89
3	if not os.path.exists(path): os.mkdir(path)	Creates directory if it does not exist.	Check and create folder.	Make directory if missing.	Checks if the directory exists, and creates it if necessary.	0.91
4	user_input = input("Enter your name: ")	Reads user input from console.	Gets input.	Takes user text.	Reads a name entered by the user from the console.	0.88
5	try: open(file) except FileNotFoundError: print("Error")	Handles missing file errors gracefully.	File error handling.	Catch missing file.	Tries to open a file and prints an error if it is not found.	0.90

Результати демонструють, що донавчена модель T5, інтегрована з лексичним аналізом та структурними представленнями коду через AST, стабільно перевищує базову T5 та mT5-small у точності генерації коментарів. Зокрема, при середній довжині функції до 30 рядків спостерігалося підвищення F1 приблизно на 4–5% порівняно з базовою моделлю. Це свідчить про ефективність адаптації моделі до доменної задачі та важливість включення структурної інформації.

Загалом базова англомовна модель демонструє задовільні результати на простих текстах, але погано узгоджується з коментарями, що містять терміни програмування. Модель mT5-base завдяки багатомовному переднавчанню модель краще працює із текстами, що містять технічну лексику або синтаксис коду. Проте зберігається певна втрата точності на доменних фразах. Донавчання власної моделі на корпусі з коментарями до коду значно підвищує семантичну узгодженість вихідного тексту з оригінальним коментарем. Модель краще розуміє структурні зв'язки між кодом і природною мовою.

Аналіз залежності точності від складності програмного коду показав, що при збільшенні кількості умовних операторів, вкладених циклів та взаємозв'язків між модулями точність генерації дещо знижується. Наприклад, для функцій із складністю понад 10 умовних або циклічних конструкцій F1 зменшувалася порівняно з більш простими функціями. Однак поєднання AST-представлень та графової моделі GNN дозволяє частково компенсують цю тенденцію, оскільки система зберігає інформацію про глобальні залежності між компонентами коду.

Досліджено також вплив довжини AST на якість генерації. Було встановлено, що для дуже коротких AST (до 15 вузлів) модель часто пропускає деталі коду, генеруючи лише загальні коментарі. Для середньої довжини AST (15–50 вузлів) точність коментарів є найвищою, оскільки модель здатна повністю захопити локальні та глобальні структури коду. При надзвичайно довгих AST (понад 100 вузлів) спостерігається певне зниження точності, що пов'язано із складністю відстеження всіх зв'язків та обмеженням контексту у трансформерній моделі. Для зменшення цього ефекту у системі застосовуються додаткові механізми обрізання та пріоритетного аналізу ключових вузлів AST.

Висновки

Дане дослідження є актуальним і має суттєву наукову та практичну цінність. Основна особливість роботи полягає у створенні інтегрованої системи генерування документації для програмного коду, яка поєднує донавчену трансформерну модель T5 та графову модель GNN, з використанням абстрактних синтаксичних дерев (AST) у лінійному представленні для аналізу структури коду. Запропонований підхід забезпечує комплексне урахування локальних та глобальних контекстів програмних фрагментів, дозволяючи системі ефективно визначати призначення коду, взаємозв'язки між його компонентами та формувати змістовні коментарі й документацію.

Результати дослідження підтвердили ефективність адаптації трансформерної моделі до конкретної доменної задачі. Донавчання T5 на спеціалізованому корпусі з прокоментованим кодом у поєднанні з лексичним аналізом підвищило якість генерації приблизно на 4% за метрикою F1 порівняно з базовою моделлю.

Це демонструє, що цільове донавчання великих трансформерних моделей може суттєво покращити результати автоматичного документування програмного коду, особливо для великих Python-проектів. Крім того, використання GNN для моделювання залежностей між файлами та модулями дозволяє візуалізувати структуру проекту, забезпечуючи додаткову інформацію для розуміння коду.

Практичне значення результатів полягає у створенні реального інструмента для автоматичного генерування документації, що дозволяє розробникам значно скоротити час на створення пояснень до коду та підтримку великих програмних систем. Використання комбінації «seq2seq» трансформерів, методу донавчання LoRA, токенізації через SentencePiece, аналізу AST і графового моделювання через GNN створює інтегрований підхід, який забезпечує високу точність та гнучкість системи. Така комплексна архітектура дозволяє моделі швидко адаптуватися до нових проектів і специфічних задач.

Наукове значення роботи полягає у тому, що дослідження демонструє можливість об'єднання різних методів машинного навчання для вирішення проблем автоматизації документування коду,

відкриваючи перспективи для подальшого розвитку інтелектуальних систем аналізу програмних проектів. Досягнуті результати можуть стати основою для розробки нових методів генерації документації, включно з багатомовними системами, інтеграцією з IDE, а також автоматизованим контролем якості програмного забезпечення.

Перспективи подальших досліджень включають удосконалення графових моделей для більш точного відображення залежностей між модулями, оптимізацію архітектури трансформера під специфічні мови програмування, інтеграцію семантичного аналізу коду та підвищення масштабованості системи для великих корпоративних проектів. Крім того, можлива реалізація адаптивних систем, які самостійно оновлюють документацію під час внесення змін у код, що значно підвищить продуктивність розробників і зменшить ризики помилок у великих програмних системах.

Таким чином, результати роботи демонструють ефективність поєднання трансформерних та графових моделей, а також забезпечують комплексний підхід до автоматизації створення документації. Запропонована система має потенціал для широкого практичного застосування та подальшого розвитку у сфері автоматичного аналізу коду й інтелектуальної підтримки програмної розробки.

REFERENCES

1. Van Heesch D. *Doxygen: Source Code Documentation Generator* [Electronic resource]. – Available at: https://www.doxygen.nl/
2. GitHub. *GitHub Copilot Documentation* [Electronic resource]. – Available at: https://docs.github.com/copilot
3. Swimm. *Auto-Docs: AI-Powered Code Documentation Generator* [Electronic resource]. – Available at: https://swimm.io/
4. Shiina H., Onishi S., Takahashi A., Kobayashi N. Automatic comment generation for source code using external information by neural networks for computational thinking // *International Journal of Smart Computing and Artificial Intelligence*. — 2021. — Vol. 5, No. 2. — P. 15–28.
5. Wei B., Li G., Xia X., Fu Q., Jin Z. Retrieve and Refine: Exemplar-based Neural Comment Generation // *Proceedings of the 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. — 2020. — P. 1–12.
6. Krasanakis A., Papadopoulos S., Kompatsiaris I. JGNN: Graph Neural Networks on Native Java // *Proceedings of the 20th International Conference on Mining Software Repositories (MSR)*. — 2023. — P. 1–12.
7. Vaswani A., Shazeer N., Parmar N. et al. Attention Is All You Need // *Advances in Neural Information Processing Systems*. – 2017. – № 30. – Available at: https://arxiv.org/abs/1706.03762
8. Raffel C., Shazeer N., Roberts A. et al. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer (T5) [Electronic resource]. – 2020. – Available at: https://arxiv.org/abs/1910.10683
9. Hu E. J., Shen Y., Wallis P. et al. LoRA: Low-Rank Adaptation of Large Language Models [Electronic resource]. – 2021. – Available at: https://arxiv.org/abs/2106.09685
10. Zhang T., Kishore V., Wu F. et al. BERTScore: Evaluating Text Generation with BERT [Electronic resource]. – 2020. – Available at: https://arxiv.org/abs/1904.09675

Albina Kostiuchenko *Master's student of the Department of System Programming and Specialized Computer Systems, National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", 03056, Ukraine, Kyiv, Polytechnichna St., 14-a*
e-mail: albina.kostyuchenko03@gmail.com
https://orcid.org/0009-0004-7382-7209

Andrii Petrashenko *Ph.D., Associate Professor of the Department of System Programming and Specialized Computer Systems, National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", 03056, Ukraine, Kyiv, Polytechnichna St., 14-a*
e-mail: petrashenko@gmail.com;
https://orcid.org/0000-0003-0239-1706

Method for generating source code description using an artificial intelligence model

Relevance. The topic is relevant, since currently there are many large projects that are being developed over a long period of time and require support and understanding of the code without explanations. The rapid development of technologies and the need to constantly develop new features and support existing ones require constant updating of documentation. Writing good documentation is a valuable skill that requires experience, concentration and understanding of the project structure. As a result, a large number of developers consider the process of writing documentation difficult and think that the time spent on it could be used more productively. That is why there is a demand for services that help automate this process.
Goal. The purpose of this work is to increase the efficiency of automated generation of software documentation. As part of this task, the necessary theoretical material was worked out, existing solutions to this problem were studied, and our own new method of generating a description of the program code was developed and implemented, which more accurately determined the purpose of code fragments, clearly understood the structure and dependencies between its components.
Research methods. The study is based on literature analysis, statistical methods, as well as machine learning and data mining methods. In particular, the methods of syntactic code analysis and construction of an abstract syntax tree (AST), the method of forming a training corpus, methods of training and retraining of transformer and graph models were used. To assess the advantages of the retrained model, the method of comparative modeling and automated text quality assessment (in this case, BERTScore) was used.
The results. Retraining the T5 model on a specialized dataset with commented code in combination with lexical analysis allowed to increase the quality of generation by approximately 4% in terms of the F1 metric compared to the base model. This indicates that adapting the model to a specific domain task is effective and can significantly improve the result.
Conclusions. Based on the collected data, an own approach was proposed to improve the quality of code description generation using the retrained T5 model and the created GNN model with further implementation, which is the result of the research. The proposed system combines the best practices of syntactic analysis, graph modeling, and transformer generation, providing a practically applicable solution for automatic documentation creation. It can be argued that the combination of "seq2seq" models, tokenization and adaptation methods of large transformers, as well as code analysis via GNN and structural AST representations provides a comprehensive approach to automating work with code, allowing you to combine local and global contexts, quickly adapt the model to specific tasks, and effectively generate meaningful comments and documentation. Such an integrated approach has the potential for further development of artificial intelligence systems in the field of automatic code analysis, increasing developer productivity, and ensuring software quality. The research results can be applied in practice for fast and effective creation of documentation for developed software and large projects in the Python language.
Keywords: *machine learning, T5, GNN, code description generator, model training, natural language processing, documentation, AST.*

Надійшла у першій редакції 29.09.2025, в останній - 30.10.2025.
The first version has been received on 29.09.2025, the final version – on 30.10.2025.