УДК 004.08 https://doi.org/10.15407/pp2026.03.053

Д.О. Вітковський, В.М. Шимкович

МАТЕМАТИЧНІ МОДЕЛІ ПРОГРАМНИХ КОМПОНЕНТІВ СЕРВІСІВ ПРОВАЙДЕРІВ ІНФОКОМУНІКАЦІЙНИХ ПОСЛУГ

Автоматизація розв'язування інтелектуальних задач, зокрема проєктування програмних компонентів сервісів провайдерів інфокомунікаційних послуг, за допомогою штучного інтелекту є важливою і актуальною в платформах підтримки життєвого циклу сервісів ІТ-компаній розробників за моделлю End-to-End. У даній роботі розглянута проблема синтезу програмного коду компонентів сервісів за допомогою моделей глибокого навчання. Проаналізовано вимоги до компонентів сервісів, проаналізовано можливі програмні коди компонентів сервісів. Розроблено математичну модель для опису вимог до програмних компонентів сервісів. Проаналізовано математичні моделі представлення програмного коду мовою Java, що на цей час існують, та можуть бути використані для математичного опису програмного коду компонентів сервісу. Розроблено математичну модель програмного коду компонентів сервісу, що відрізняється від існуючих контролем та адаптацією до особливостей граматики конкретної мови. Це дозволяє продовжувати використання унімодальних мовних моделей. Запропонована модель є складовою платформи підтримки життєвого циклу сервісів у інформаційних системах провайдерів. У інтегральній системі засобів автоматизації процесів життєвого циклу сервісів запропоновані математичні моделі дозволять навчати та впроваджувати глибокі нейронні мережі для проєктування програмного коду компонентів сервісів.

Ключові слова: життєвий цикл сервісів, проєктування сервісів, штучний інтелект, нейронні мережі, глибоке навчання, генерація програмного коду.

D.O. Vitkovskyi, V.M. Shymkovych

MATHEMATICAL MODELS OF SOFTWARE COMPONENTS FOR INFOCOMMUNICATION SERVICE PROVIDERS

The automation of solving intellectual tasks—specifically, the design of software components for information and communication service providers—using artificial intelligence is an important and timely issue in service lifecycle management platforms for IT development companies based on the End-to-End model. This paper addresses the problem of synthesizing service component code using deep learning models. The requirements for service components are analyzed, and possible service component codes are examined. A mathematical model has been developed to describe the requirements for service software components. The paper analyzes existing mathematical models for representing program code in the Java language that can be used for the mathematical description of service component code. A mathematical model of the program code of service components has been developed, which differs from existing ones in control and adaptation to the features of the grammar of a specific language. This allows for the continued use of unimodal language models. The proposed model is a component of the platform for supporting the life cycle of services in information systems of providers. Within an integrated system of tools for automating service lifecycle processes, the proposed mathematical models will enable the training and implementation of deep neural networks for designing the program code of service components.

Keywords: service lifecycle, service design, artificial intelligence, neural networks, deep learning, code generation.

Вступ

Сервісно-орієнтований підхід дедалі більше зміцнює свої позиції в ІТ-сфері. Набір інструментів для його впровадження в різних галузях є досить широким і постійно розширюється. Особливо динамічно розвиваються засоби підтримки життєвого циклу сервісів в інформаційних системах провайдерів інфокомунікацій. Це зумовлено як сучасною практикою побудови таких систем на основі моделі E2E, так і активним розвитком та впровадженням технологій штучного інтелекту для розв’язання складних інтелектуальних завдань.

Проблематика проєктування програмних компонентів сервісів в інформаційних системах телекомунікаційних провайдерів набула актуальності разом із впровадженням архітектурного підходу в сучасних методологіях розробки програмного забезпечення. Відтоді проєктування сервісів почали розглядати як формування їхньої архітектури, що базується на типовому наборі компонентів. Водночас це не обмежилося лише зміною термінології — процес побудови архітектури сервісів ускладнився новими аспектами та перетворився на комплексну задачу, ефективне вирішення якої є важливим для провайдерів.

Для подолання цієї складності широко застосовуються технології, що дозволяють описувати архітектуру сервісів за допомогою графічних мов із використанням напрацьованих шаблонів. Утім, подібні рішення зазвичай мають індивідуальний характер і адаптуються під конкретні ІТ-компанії, які забезпечують діяльність провайдерів інфокомунікаційних послуг, надаючи відповідні компоненти та технології, зокрема в межах моделі E2E.

Розроблення програмного коду сервісів і їхніх компонентів належить до класу складних інтелектуальних задач, які неможливо звести до чітко формалізованих правил. Саме тому для їхньої часткової або повної автоматизації доцільно залучати методи та моделі штучного інтелекту. На сьогодні ШІ, передусім методи машинного навчання та нейронні мережі, уже довели свою ефективність у широкому спектрі прикладних задач — від аналізу зображень і роботи з природною мовою до синтезу медіаконтенту, обробки мовлення та підтримки процесів ухвалення рішень. Нейронні мережі особливо цінні завдяки здатності апроксимувати складні залежності та навчатися на даних, що дозволяє застосовувати їх як універсальний інструмент для моделювання інтелектуальної поведінки. Якщо задати формалізоване представлення вхідних і вихідних параметрів задачі, можна навчити модель відтворювати взаємозв’язки між ними у режимі навчання з учителем. У такому підході задача проєктування програмного коду сервісів трансформується у задачу навчання моделі, яка за заданими вимогами здатна генерувати відповідні рішення. Це відкриває можливість автоматизованого створення компонентів сервісів, що розробляються ІТ-компаніями для провайдерів інфокомунікаційних послуг у межах E2E моделі.

Ідея автоматизувати написання коду виникла ще за часів перших ЕОМ, коли програми писали на найнижчому рівні — рівні машинного коду. З початком 2010-х років парадигма автоматичної генерації коду зазнала змін завдяки накопиченню масштабних масивів відкритого вихідного коду та розвитку ініціатив з інтелектуального аналізу програмних репозиторіїв (Mining Software Repositories — MSR) [1]. Дослідники почали розглядати код як неструктуровані дані для методів машинного навчання, що отримало назву «Великий Код». Наступним кроком стала адаптація принципів обробки природної мови (Natural Language Processing — NLP), де завдання генерації коду формулювалося як задача статистичного машинного перекладу [2]. Для її вирішення були впроваджені рекурентні нейронні мережі (Recurrent Neural Networks — RNN) [3], які дозволили ефективно моделювати часові та структурні залежності в коді, долаючи проблему градієнта, що зникає [2].

Дослідження переконливо продемонстрували здатність RNN генерувати ймовірнісні послідовності викликів методів та виконувати контекстне автодоповнення коду.

Незважаючи на ефективність моделей Sequence-to-Sequence (Seq2Seq), їхнє застосування виявило суттєве алгоритмічне обмеження — проблему відкритого словника (Out-of-Vocabulary — OoV). Оскільки програмісти постійно створюють нові унікальні ідентифікатори, класичні нейромережі з фіксованим словником були змушені замінювати їх на беззмістовний токен «<unk>». Рішенням стала інтеграція механізмів копіювання на базі архітектури Pointer Networks, запропонованої у 2015 році [4]. Замість генерації слова виключно з фіксованого словника, гібридні мережі з вказівниками навчилися обчислювати ймовірність копіювання токенів безпосередньо з локального контексту, реплікуючи унікальні ідентифікатори розробника в процесі генерації [5].

У період 2016–2017 років виникли гібридні рішення, які інтегрували глибоке навчання з методами формального синтезу. Наприклад, архітектура DeepCoder (2016) [6] використовувала нейромережу для прогнозування ймовірності використання конкретних високорівневих функцій на основі прикладів вводу-виводу [6]. Ці ймовірності передавалися класичному синтезатору, що радикально звужувало простір пошуку алгоритмів і значно прискорювало генерацію складного коду [6]. Фундаментальна зміна парадигми відбулася 2017 року з розробкою архітектури Transformer. Механізм внутрішньої уваги усунув обмеження послідовної обробки RNN [2], забезпечивши ефективну паралелізацію обчислень. Це сприяло появі перших комерційних інструментів, що забезпечували предиктивну генерацію коду безпосередньо в інтегрованому середовищі розробки (IDE).

Подальше масштабування зумовило появу Великих мовних моделей (BMM, Large Language Models — LLM) з мільярдами параметрів. Сучасні системи, такі як StarCoder, GPT, Codex, здатні генерувати комплексні функціональні блоки за природномовними специфікаціями, формуючи парадигму асистованої розробки, де генерація коду є процесом інтелектуальної колаборації між інженером та автономним програмним агентом. [7]

Метою дослідження є розробка методу та математичних моделей структурної токенізації і детокенізації програмного коду, адаптованих для автоматизації реалізації E2E сервісів.

Основним внеском даної роботи є наступне:

1) розроблено модель вихідного коду у вигляді послідовності спеціалізованих структурних токенів, що зберігає синтаксичну ієрархію без необхідності модифікації базової архітектури нейронної мережі;

2) забезпечено мінімізацію загальної довжини вихідної та кінцевої моделі шляхом усунення синтаксичного шуму (знаків пунктуації, неінформативних вузлів тощо);

3) розглянуто обернене перетворення від моделі до коду, здатного забезпечити гарантоване відновлення згенерованої мовною моделлю послідовності у синтаксично правильний програмний код.

1. Огляд літератури

На сьогодні існує низка підходів, які вирішують проблему зниження якості семантичної інформації під час лінійної токенізації програмного коду для мовних моделей. Традиційні підходи, такі як BPE чи SentencePiece, ігнорують формальну ієрархічну природу коду. Наукова спільнота запропонувала структурні методи до вирішення цієї проблеми, опис яких наведено у підрозділі 1.1. Порівняння методів наведено у таблиці 1.

Таблиця 1.

Порівняння рішень та методів подання, адаптованих до коду

Автори	Модель	Метод	Рік	Датасет	Бенчмарк	Метрики
Guo et al. [8]	GraphCode-BERT	Лінійна токенізація (BPE, WordPiece) із додаванням змінних з DFG; кодування залежності між вузлами матрицею маскованої уваги	2021	CodeSearch-Net	CodeSearch-Net, Big-CloneBench	Code search: MRR; Clone detection: Prec., Rec., F1; Translation/ Refinement: BLEU, Accuracy
Jiang et al. [9]	TreeBERT	Розкладання AST на шляхи від кореневого до листкових вузлів; моделювання мови, з маскуванням деревом, та передбачення порядку вузлів	2021	CuBERT, ETH Py150, Java-{small, med, large}, DeepCom,	ETH Py150, Java-large, DeepCom, CodeNN	Code summarization: Prec., Rec., F1; Documentation: BLEU;
Wang et al. [10]	SynCoBERT	BPE токенізатор; Мультимодальне контрастне навчання на ідентифікаторах та ребрах AST	2021	CodeSearch-Net, Big-CloneBench, POJ-104	CodeXGLUE, CodeSearch	Clone detection: Prec., Rec., F1, MAP@R; Defect detection: Acc.; Translation: BLEU, EM, CodeBLEU
Ahmad et al. [11]	PLBART	Попереднє тренування задачею знешумлення набору, токенізованого SentencePiece, із подальшим калібруванням для задач розуміння та генерації	2021	Java/Python (GitHub) та Stack-Overflow	Code-XGLUE	Code summarization: smoothed BLEU-4; Code generation/ translation: BLEU, EM, CodeBLEU; Repair: EM, BLEU; Vuln. detection: Acc.; Clone detection: F1
Wang et al. [12]	CodeT5	T5 архітектура (кодувальник-декодувальник), тренована розуміти ідентифікатори (назви змінних, функцій, типів тощо)	2021	8.35M функцій з GitHub/Code-SearchNet	CodeXGLUE	Code summarization: smoothed BLEU-4; Code generation: EM, BLEU, CodeBLEU; Code translation: EM, BLEU; Defect detection: Acc.; Clone detection: F1
Niu et al. [13]	SPT-Code	Попереднє тренування через XML-подібний обхід AST (X-SBT) для лінеаризації дерева	2022	CodeSearch-Net (CSNw/doc)	CodeXGLUE	Code summarization: BLEU, METEOR, ROUGE-L; Code completion: Acc@1 (aka Acc.), Acc@5; Code translation/bug fixing: Acc., BLEU; Code search: MRR

Автори	Модель	Метод	Рік	Датасет	Бенчмарк	Метрики
Guo et al. [14]	UniXcoder	Попереднє тренування з використовуючи спеціальні токени для вузлів AST; RoBERTa токенізатор	2022	CodeSearch-Net, C4	CodeXGLUE	Clone detection: MAP@R, Rec., Prec., F1; Code search: MRR; Code summarization: BLEU-4; Code generation: EM, BLEU-4; Code completion: EM, Edit Sim; Code-to-code search: MAP
Chakraborty et al. [15]	NatGen	Збільшення «природності» коду; Тренування на приведенні синтаксично правильного, але «зашумленого» марними конструкціями («неприродного») коду до більш природного виду	2022	CodeXGLUE, CONCODE, BugFix (Tufano et al.)	CodeXGLUE	Code generation: EM, SM, DM, CodeBLEU, CS, Median ED; Code translation: EM, SM, DM, CodeBLEU; Bug fixing: Acc.; Code summarization: BLEU
Tipirneni et al. [16]	StructCoder	CodeT5 токенізатор; конкатенація з вкладеннями AST та DFG	2024	CodeSearch-Net	APPS, Code-XGLUE	Code translation/ generation: BLEU, xMatch, CodeBLEU
Gong et al. [17]	AST-T5	Сегментація тренувального набору даних, токенізованого GPT-4 з урахуванням меж AST	2024	The Stack Dedup	HumanEval, MBPP, Code-XGLUE, Defect Detect	Code generation: Pass@1, EM; Code translation: EM; Clone detection: F1; Defect detection: Acc.
Bartkowiak P., Graliński F. [18]	Tree-Enhansed CodeBERTa	Розширення Code-BERTa вкладеннями глибини вузла та індексу вузла серед сусідніх вузлів у AST	2025	CodeSearch-Net	Semantic-CloneBench, BigClone-Bench, Code-SearchNet	Masked language modeling/ Clone detection: Acc., Prec., Rec., F1;

Якість роботи моделей оцінюється за допомогою специфічних метрик. Оскільки стандартні NLP-метрики часто бувають оманливими при застосуванні до задач роботи з кодом: програма може мати високий лексичний збіг, але містити синтаксичну помилку, на кшталт відсутності обов'язкової пунктуації, що робитиме її нефункціональною. Основні метрики включають:

- MRR (Mean Reciprocal Rank), MAP (Mean Average Precision) та MAP@R: здебільшого застосовуються для задач пошуку коду та виявлення клонів. MRR оцінює позицію першої правильної або релевантної відповіді. MAP вимірює усереднену точність релевантних результатів. MAP@R встановлює обмеження глибини у R.
- Accuracy, Precision, Recall, F1-score: стандартні метрики класифікації, які широко застосовуються для задач на кшталт детекції дефектів, вразливостей або

виявлення клонів (бінарна класифікація «клон»/«не клон»).

- Pass@k: оцінює функціональну коректність. Модель генерує k варіантів коду, і метрика вважається успішною, якщо хоча б один варіант успішно компілюється та проходить усі одиничні тести (англ. unit tests).
- EM (Exact Match) / xMatch: повний збіг канонічних форм. Вимірює відсоток згенерованих фрагментів коду, які абсолютно ідентичні еталонному коду після нормалізації (видалення незначущих елементів, форматування).
- ED (Edit Distance) та Edit Sim (Edit Similarity): метрики оцінки мінімальної кількості операцій (вставка, видалення, заміна, пермутація) на рівні символів/лексем, які необхідно виконати для перетворення згенерованого коду на еталонний (напр., відстань Левенштейна).
- CodeBLEU: гібридна метрика, що є зваженою комбінацією лексичного збігу (n грам), синтаксичного збігу (SM, Syntax Match) та семантичного збігу (DM, Dataflow Match).
- CS (Copies from Source): відсоток прикладів, де згенерований вихідний код є безпосередньою копією вхідних даних. Ця метрика використовується, зокрема, у NatGen [15].
- BLEU, (smoothed) BLEU-4, ROUGE(-L), METEOR: метрики для оцінки генерації природномовних текстів (документування, коментарі тощо). BLEU-4 оцінює точність збігу 4-грам (smoothed версія використовує математичне згладжування для уникнення нульових значень). ROUGE-L вимірює найдовшу спільну підпослідовність між згенерованим текстом і еталоном. METEOR додатково враховує синоніми та морфологічну спорідненість слів для більш гнучкої оцінки.

1.1. Використання формальності мов програмування. Підвалиною більшості сучасних рішень для генерації коду на основі мовних моделей є використання формальності мов програмування (МП) на користь процесу навчання. Ідейно, такий підхід дозволяє мовним моделям уникнути непевного вивчення синтаксису МП як комбінації токенів звичайного тексту, а натомість вивчати закономірності та зв'язки між структурами коду. Аналізуючи передові підходи, можна прослідкувати їхній розвиток та ідентифікувати специфічні недоліки кожного з них у застосуванні до задач автоматизації розробки програмного забезпечення (ПЗ).

Однією з перших мовних моделей на основі архітектури Transformer, яка почала використовувати не лише текстове представлення коду, стала GraphCodeBERT [8]. Її підхід продовжує використання лінійної токенізації, але додає змінні графу потоку даних (Data Flow Graph — DFG) у кінець послідовності, фокусуючи механізм уваги на їхніх семантичних зв'язках. Порівняно із суто текстовими моделями, це суттєво покращує розуміння логіки роботи коду. Однак його недоліком є те, що обчислення DFG та побудова масок уваги є ресурсоємним процесом, який ускладнює роботу моделі в реальному часі.

У моделі TreeBERT [9] було запропоновано структурне розкладання абстрактного синтаксичного дерева (Abstract Syntax Tree — AST) на множину шляхів від кореневого вузла до термінальних. Хоча це краще фіксує ієрархію порівняно з лінійною послідовністю, метод вимагає побудови повного AST. Це робить його малопридатним для ітеративних задач генерації коду, оскільки проміжним станом розробки майже завжди є синтаксично неповна послідовність, з якої неможливо побудувати завершене дерево.

Метод SynCoBERT [10] застосовує мультимодальне контрастне навчання, змушуючи базовий кодувальник передбачати структурні ребра між вузлами AST та класифікувати ідентифікатори. Попри покращення, модель розглядає код та AST як окремі паралельні модальності. Це добре працює для задач пошуку або класифікації, але не формує єдиної структурно цілісної послідовності, необхідної для послідовної генерації коду, бо відсутнє так зване єдине джерело правди, а контекст моделі вичерпується швидше.

Модель PLBART [11] використовує принцип автокодувальника, що усуває шум. Під час тренування до коду токенізованого SentencePiece, навмисно додається шум через маскування чи видалення токенів, для стимулювання моделі до самостійного вивчення граматики. Оскільки модель засвоює синтаксис виключно неявно, вона позбавлена вбудованих механізмів обмеження генерації за формальними правилами мови. У задачах автоматизації розробки це призводить до того, що під час генерації довгих методів або складних архітектурних структур підвищується ризик продукування синтаксично некоректного коду.

Розробники моделі CodeT5 [12] зберегли лінійну BPE токенізацію, компенсувавши відсутність явної структурної інформації спеціалізованими завданнями, такими як прогнозування замаскованих ідентифікаторів. Хоча модель добре навчена розпізнавати семантичні ролі змінних та функцій, вона оперує одновимірним текстом. Ієрархічні структурні зв'язки, такі як приналежність ідентифікатора до певного класу чи його локальна область видимості, не задаються формально, а неявно вивчаються механізмом уваги, чого часто недостатньо для точного синтезу архітектурних компонентів.

У моделі SPT-Code [13] для інтеграції структури використано XML-подібний обхід AST (X-SBT). Для уникнення швидкого вичерпання ліміту контекстного вікна мовної моделі, автори не виконують обхід вузлів на рівні виразів (expression-level), залишаючи внутрішню структуру складних вузлів нерозкритою. Крім того, тут присутня та сама інформаційна надлишковість що й у випадку з SynCoBERT.

Автори UniXcoder [14] теж використовують спеціалізовані токени для представлення нетермінальних вузлів AST, але лише під час попереднього навчання. Однак на етапі виведення використання цієї модальності оминається й модель покладається лише на плоский текст. Це дозволяє зекономити контекстне вікно, але призводить до того, що явні структурні обмеження відсутні під час генерації коду.

Підхід NatGen [15] фокусується на приведенні синтаксично правильного, але штучно ускладненого коду до більш «природного» вигляду, з урахуванням манер розробника. Це підвищує якість попереднього навчання, але не змінює базову форму представлення даних — модель продовжує оперувати лінійними текстовими токенами.

У StructCoder [16] і кодувальник, і декодувальник працюють із графовими структурами (AST та DFG). Передбачення шляхів AST та ребер DFG є корисними допоміжними завданнями та не збільшують кількість токенів, що передбачаються (важливо для авторегресивного декодування). Однак ці додаткові завдання теж вимагають впровадження архітектурних змін до мережі.

Автори AST-T5 [17] пропонують враховувати структурні межі AST під час сегментації тренувального набору даних за допомогою алгоритму динамічного програмування. Це вирішує проблему «розірваних» тверджень під час навчання, однак текст все ще обробляється стандартним BPE.

У [18] пропонують модифікувати архітектуру Transformer, інтегрувавши позиційні вкладення глибини вузла та індексу серед сусідніх вузлів у AST, що помітно покращує якість роботи Tree-Enhanced CodeBERTa порівняно із базовою CodeBERTa.

1.2. Постановка задачі. Автоматизація проєктування програмних компонентів сервісів за допомогою ВММ потребує ефективного подання синтаксичних та семантичних структур коду. Як показав аналіз існуючих рішень, застосування традиційних алгоритмів текстової токенізації призводить до втрати явної структурної та ієрархічної інформації і підвищує імовірність генерації синтаксично некоректного коду під час синтезу складних архітектур. Водночас наявні структурні підходи мають недоліки, з точки зору практичної зручності інтеграції їх у робочий процес інженерії

ПЗ — через свою схильність до швидкого вичерпання ліміту контекстного вікна, або необхідність внесення архітектурних змін до механізмів уваги чи позиційного кодування. Останнє унеможливлює використання донавчання (fine-tunning) фундаментальних ВММ, що вже існують, та блокує використання стандартних високопродуктивних рушіїв виведення.

З огляду на це, виникає об’єктивна необхідність у розробці підходу до представлення програмного коду, який би забезпечував баланс між збереженням структурної інформації, оптимальністю довжини контексту та сумісністю зі стандартними ВММ.

2. Математичні моделі програмних компонентів

У даній роботі пропонуються математичні моделі, що передбачають використання спеціалізованих токенів для представлення вузлів конкретного синтаксичного дерева (Concrete Syntax Tree — CST), очищеного від синтаксичного шуму та надлишковості й відформатоване з використанням системи реєстрів вузлів. Система реєстрів дозволяє наблизити CST до рівня абстрактності відповідного AST. Використання системи реєстрів також дозволяє аналогічним конструкціям різних мов програмування бути представленими тими самими токенами з точки зору словника моделі. Так тернарний оператор («?:») мови Java є логічно еквівалентним використанню розгалуження потоку керування («if») у позиції виразу в мові Kotlin.

2.1. Математична модель програмного коду. Для роботи ВММ, програмний код необхідно перетворити у послідовність векторів вкладень, адже саме з ними модель далі виконує обчислення. Першим кроком перетворення є операція токенізації, за якої текст розбивається на семантичні одиниці — токени.

Найпростішим алгоритмом токенізації можна вважати поділ тексту по пробілах, але такий алгоритм є неоптимальним та породжує велику кількість унікальних токенів, як множину усіх можливих поєднань префіксів, суфіксів, коренів, закінчень тощо. Сучасні моделі для природного тексту зазвичай використовують статистичні підходи, такі як кодування пар байтів (Byte Pair Encoding — BPE), WordPiece чи SentencePiece.

Пропонується використання комбінації токенізатора тексту природною мовою та підходу на основі обходу CST [10, 13, 14, 16]. Вузли дерева, ближчі до природної мови, такі як ідентифікатори, літерали, коментарі, документація тощо, підлягають токенізації з використанням статистичних методів, згаданих вище. Для виконання розбору коду на CST, використовується популярна бібліотека «tree-sitter» [21], разом із допоміжними бібліотеками граматик відповідних мов.

Без попередньої обробки розібрані «tree-sitter» синтаксичні дерева містять багато термінальних вузлів, що не несуть фактичного смислового навантаження, такі як знаки пунктуації та дужки. Однак є винятки, а також деякі з цих термінальних вузлів пунктуації можуть нести важливий сенс, або мати абсолютно різний сенс залежно від типу батьківського вузла. Так у «tree-sitter» граматиці для мови Java вузли “<” та “>” одночасно слугують і для відокремлення параметрів типу, і для позначення двійкових логічних операторів «менше» та «більше» відповідно.

За таких умов було вирішено розробити систему регістрів для типів вузлів. Під час реєстрації вузла даної граматики до реєстру записується конфігурація, що спрацьовує залежно від виду даного вузла та виду його батьківського вузла. Конфігурація вказує ім’я токена, та чи генерувати парні токени, одиничний токен, залишити сам вміст вузла, чи пропустити вузол, перейшовши до його дочірнього вузла за наявності.

Такий підхід дозволяє мінімізувати кількість токенів відповідної граматики, позбутися токенів пунктуації, зменшити шум та запобігти тому, що один токен буде мати ортогональні значення залежно від батьківського вузла.

Оскільки «tree-sitter» граматики орієнтовані на підсвітку синтаксису та швидкодію, а не повноту, деякі з них можуть видавати CST, у якому не вказані усі вузли. Наприклад, «tree-sitter-properties» [22] не видає дочірні вузли змісту для вузлу «value», якщо серед дочірніх вузлів «value» є вузли «substitution». Для протидії цьому необхідно виконувати спеціальну обробку вузлів, подібних до «value», аби почергово видавати токени змісту та токени дочірніх вузлів.

Запропонуємо наступну математичну модель роботи алгоритму токенізації.

Нехай парсер $\mathrm{p}: S \to T$ [21] є відношення між множиною $S \subset A^*$ усіх програмних кодів (де A^* — множина усіх текстів) та множиною T синтаксичних дерев; C — множина станів курсорів обходу дерева; курсор $c_{\text{root}} \in C$, що починає у кореневому вузлі дерева $t \in T$, можна отримати функцією $\mathrm{walk}: T \to C$; $\mathrm{node}: C \to N$ — функція, що повертає поточний вузол курсору; $\mathrm{child}_\mathrm{l}: C \to C_\perp$ — функція, що повертає стан курсору, встановленого на першому дочірньому вузлі вихідного курсору, за наявності; $\mathrm{sibling}_\mathrm{l}: C \to C_\perp$ — функція, що повертає стан курсору, встановленого на наступному сусідньому вузлі вихідного, за наявності; $\mathrm{parent}: C \to C_\perp$ — функція, що повертає стан курсору, встановленого на батьківському вузлі, за наявності; $\mathrm{depth}: C \to \mathbb{N}_{>0}$ — функція, що повертає глибину курсору, де

$$\mathbb{N}_{>0} = \{m \mid \forall m \in \mathbb{N} : m > 0\};$$

$\left(\mathrm{r}: N \times N_\perp \to V\right) \in R = V^{N \times N_\perp}$ — реєстр правил — функція, яка приймає вузол синтаксичного дерева $n = \mathrm{node}(c)$ і його батьківський вузол $n_p = \mathrm{node}\left(\mathrm{parent}(c)\right)$ як контекст, та повертає правило обробки вузла —

$$V = \begin{Bmatrix} \text{Парний}, \text{Парний з переплетенням}, \\ \text{Одинак}, \text{Зміст}, \text{Пропустити} \end{Bmatrix};$$

$\mathrm{l}: R \to C \to L^*$ — лексер — каррована функція вищого порядку, яка приймає реєстр з R та повертає функцію, що є відношенням між множиною C та множиною L^* послідовностей токенів (слів) з алфавіту токенів L; $\mathrm{l}_\mathrm{nat}: A^* \to L^*$ — токенізатор природної мови; $\perp$ — значення, що позначає відсутність значення, також відоме як NIL або NULL, $\forall B: B_\perp = B \cup \{\perp\}$. Тоді токенізатор $\mathrm{f}_\mathrm{l}: S \to L^*$ певної мови визначається як

$$\mathrm{f}_\mathrm{l}(s) = \left(\mathrm{l}(\mathrm{r}_\mathrm{l}) \circ \mathrm{walk} \circ \mathrm{p}_\mathrm{l}\right)(s), \qquad (1)$$

де $\mathrm{p}_\mathrm{l} \in T^S$ — це парсер цієї мови, $\mathrm{r}_\mathrm{l} \in R$ — реєстр правил обробки вузлів синтаксичного дерева цієї мови.

Далі наведено опис лексера l, як «генератора», що породжує елементи послідовності $(\mathbf{l} = l_1 \ldots l_m) \in L^*$. На рис. 1–5 зображено псевдокод лексера l, де $a.b$ — посилання на поле b об'єкту a (за передумови $a \neq \perp$),

$$a?.b = \begin{cases} a.b, & \text{якщо } a \neq \perp \\ \perp & \text{інакше} \end{cases},$$

$$a??b = \begin{cases} a, & \text{якщо } a \neq \perp \\ b & \text{інакше} \end{cases}.$$

Генератор 1. AST-To-Tokens (Рис. 1). Вхід**:** реєстр $\mathrm{r} \in R = V^{N \times N_\perp}$ правил обробки вузлів; курсор дерева $c_{\text{root}} \in C$; текст $s \in S$ вихідного коду, що токенізується.

Етап 1. Ініціалізувати порожніми стек закривних токенів $(\mathbf{c}_{\text{to-close}} = \varepsilon) \in C^*$ та стек «переплетених» вузлів $(\mathbf{i} = \varepsilon) \in I^*$, де $I = \mathbb{N}_{>0} \times \mathbb{N}_0^2$. Ініціалізуємо $c \leftarrow c_{\text{root}}$. $\forall i \in I : \mathrm{pr}_1(i) \in \mathbb{N}_{>0}, \mathrm{pr}_2(i) \in \mathbb{N}_0, \mathrm{pr}_3(i) \in \mathbb{N}_0$, де pr_j — проєкція j-го елементу кортежу: $\mathrm{pr}_1(i)$ — глибина, на якій знаходиться переплетений вузол, $\mathrm{pr}_2(i)$ — поточне значення індексу лівої межі підрядку s, що ініціалізується як $\mathrm{start}(n_i)$, та оновлюється після обробки чергового вузла, дочірнього до n_i, $\mathrm{pr}_3(i)$ — значення індексу правої межі підрядку в s, що

залишається рівним $\mathrm{end}(n_i)$; n_i — переплетений вузол, котрого стосується i. $\forall s \in S, n \in N : \mathrm{start}(n) \in \mathbb{Z}_0, \mathrm{end}(n) \in \mathbb{Z}_0$ — індекси початку та кінця підрядка у s, котрому відповідає вузол n.

Етап 2. Обхід дерева:

Етап 2.1. Читаємо поточний вузол дерева $n = \mathrm{node}(c)$, значення його глибини у дереві $d = \mathrm{depth}(c)$ та батьківський вузол

$$c_p = \mathrm{parent}(c),$$
$$n_p = \begin{cases} \mathrm{node}(c_p), & \text{якщо } c_p \neq \perp \\ \perp & \text{інакше} \end{cases}.$$

Етап 2.2. Доки верхнім елементом стеку $\mathbf{c}_{\text{to-close}}$ є курсор вузла на глибині більшій або рівній за поточну, виконуємо кроки 2.2.1–2.2.3, інакше — переходимо до етапу 2.3.

Крок 2.2.1. Видати елементи генератору Emit-Trailing (Генератор 2) для глибини d.

Крок 2.2.2) Вилучити верхній елемент стеку: $(c_{\text{to-close}}, \mathbf{c}'_{\text{to-close}}) \leftarrow \mathrm{pop}(\mathbf{c}_{\text{to-close}})$, $\mathbf{c}_{\text{to-close}} \leftarrow \mathbf{c}'_{\text{to-close}}$, де $\mathrm{pop}[T] : T^* \setminus \{\varepsilon\} \to T \times T^*$.

Крок 2.2.3. Видати закривний токен $\mathrm{l}_{\text{close}}(\mathrm{node}(c_{\text{to-close}})) \in L$. Перейти до кроку 2.2.

Етап 2.3. Видати токени проміжного змісту відповідно до $\mathbf{i}$ відносно вузла n (Рис. 5):

Крок 2.3.1. Починаючи з верхніх елементів $\mathbf{i}$, знайти без вилучення запис $i_p \in I_\perp$ про вузол на глибині $d_p = \mathrm{depth}(\mathrm{parent}(c)) \in d - 1$.

Крок 2.3.2. Якщо $i_p \neq \perp$ та $\mathrm{pr}_2(i_p) < \mathrm{start}(n)$, видати токени проміжного змісту як природної мови: $\mathrm{l}_{\text{nat}}(\mathrm{slice}(s, \mathrm{pr}_2(i_p), \mathrm{start}(n)))$, де $\mathrm{slice} : A^* \times \mathbb{Z}_0^2 \to A^*$ — повертає зріз вихідного рядка у заданих межах.

Етап 2.4. Видати токени, відповідно до виду вузла n (Рис. 2):

Крок 2.4.1. Визначити варіант обробки вузла $v = \mathrm{r}(n, n_p)$.

Крок 2.4.2. Якщо $v \in \{\text{Парний}, \text{Парний з переплетенням}\}$:

Крок 2.4.2.1. Видати відкривний токен $\mathrm{l}_{\text{open}}(n) \in L$.

Крок 2.4.2.2. Додати поточний вузол у чергу закривних: $\mathbf{c}_{\text{to-close}} \leftarrow \mathrm{push}(\mathbf{c}_{\text{to-close}}, c)$, де $\mathrm{push}[T] : T^* \times T \to T^*$ — додає елемент на вершину стеку.

Крок 2.4.2.3. Якщо $v = \text{Парний з переплетенням}$, додати поточний вузол до стеку вузлів із переплетеними дочірніми: $\mathbf{i} \leftarrow \mathrm{push}(\mathbf{i}, (d, \mathrm{start}(n), \mathrm{end}(n)))$

Крок 2.4.3. Якщо $v = \text{Одинак}$ — видати токен-одинак $\mathrm{l}_{\text{singleton}}(n) \in L$.

Крок 2.4.4. Якщо $v = \text{Зміст}$ — видати елементи $\mathrm{l}_{\text{nat}}(\mathrm{slice}(s, \mathrm{start}(n), \mathrm{end}(n)))$.

Етап 2.5. Оскільки поточний вузол n був оброблений — за наявності, пересунути вказівник $\mathrm{pr}_2(i_p)$ на кінець цього вузла:

Крок 2.5.1. Якщо $i_p \neq \perp$, то встановити $\mathrm{pr}_2(i_p) \leftarrow \mathrm{end}(n)$.

Етап 2.6. $c_c = \mathrm{child}_1(c) \in C_\perp$. Якщо $c_c \neq \perp$, то переходимо до етапу 2.1.

Етап 2.7. $c_s = \mathrm{sibling}_1(c) \in C_\perp$. Якщо $c_s \neq \perp$, то переходимо до етапу 2.1.

Етап 2.8. $c_p = \mathrm{parent}(c) \in C_\perp$. Якщо $c_p \neq \perp$, то переходимо до етапу 2.7.

Етап 2.9. Обхід дерева завершено — видати токени, що залишилися:

Крок 2.9.1. $(c_{\text{to-close}}, \mathbf{c}'_{\text{to-close}}) \leftarrow \mathrm{pop}(\mathbf{c}_{\text{to-close}})$, $\mathbf{c}_{\text{to-close}} \leftarrow \mathbf{c}'_{\text{to-close}}$. Якщо $c_{\text{to-close}} = \perp$ — **кінець**.

Крок 2.9.2. Видати елементи генератору Emit-Trailing (Генератор 2) для глибини d.

Крок 2.9.3. Видати закривний токен $\mathrm{l}_{\text{close}}(\mathrm{node}(c_{\text{to-close}})) \in L$. Перейти до кроку 2.9.1.

Генератор 2. Emit-Trailing (Рис. 3). Вхід: стек станів обробки переплетених вузлів $\mathbf{i} \in I^*$, де $I = \mathbb{Z}_{>0} \times \mathbb{Z}_0^2$; глибина

вузла $d \in \mathbb{N}_{>0}$; текст $s \in S$ вихідного коду, що токенізується.

Крок 1. Нехай $i = \mathrm{peek}(\mathbf{i})$, де $\mathrm{peek}[T] : T^* \to T_\perp$.

Крок 2. Якщо $i = \perp \vee \mathrm{pr}_1(i) < d \vee \vee \mathrm{pr}_2(i) \geq \mathrm{pr}_3(i)$ — **кінець**.

Крок 3. $(i, \mathbf{i}) \leftarrow \mathrm{pop}(\mathbf{i})$.

Крок 4. Видати елементи $\mathrm{l}_{\mathrm{nat}}(\mathrm{slice}(s, \mathrm{pr}_2(i), \mathrm{pr}_2(i)))$.

На рис. 6 можна побачити, яким чином виглядає результат переплетеної токенізації вмісту вузлів «val» та «subst».

```
1: ▷ Вхід: курсор AST, вихідний код, реєстр токенів
2: generator AST-To-Tokens(cursor, source, registry)
3:     struct Pending-Close
4:         name : String,
5:         depth : unsigned int,
6:     end
7:     struct Interleaved-State
8:         depth : unsigned int,
9:         last : unsigned int,                    ▷ Індекс після останнього дочірнього вузла
10:        end : unsigned int,                     ▷ Індекс кінця вузла
11:    end
12:
13:    pending-closes : Stack[Pending-Close] ← ∅
14:    interleaved : Stack[Interleaved-State] ← ∅
15:
16:    loop
17:        node ← cursor.node
18:        depth ← cursor.depth
19:
20:        while pending-closes.top.depth ≥ depth do
21:            yield from Emit-Trailing(depth)
22:            (t, d) ← pop-from(pending-closes)
23:            yield Close(t)
24:        end
25:
26:        yield from Emit-Gap(node.start, depth − 1)
27:        yield from Emit-Node-Tokens(node, depth)
28:        Update-Last-Pos(node.end, depth − 1)
29:
30:        if goto-first-child(cursor) then
31:            continue
32:        end
33:
34:        loop
35:            if goto-next-sibling(cursor) then
36:                break
37:            end
38:            if not goto-parent(cursor) then
39:                ▷ Видати решту закривних токенів
40:                while pending-closes ≠ ∅ do
41:                    (t, d) ← pop-from(pending-closes)
42:                    yield from Emit-Trailing(d)
43:                    yield Close(t)
44:                end
45:
46:                return
47:            end
48:        end
49:    end
50: end
```

Рис. 1. Функція AST-To-Tokens токенізації AST

```
1: generator Emit-Node-Tokens(node, depth)
2:   parent-kind ← node.parent?.kind
3:   config ← lookup(registry, node.kind, parent-kind)
4:
5:   match config with
6:     case Paired(t, mode) then
7:       yield Open(t)
8:       push-to(pending-closes, (t, depth))
9:       if mode = Interleaved then
10:        push-to(interleaved, (depth, node.start, node.end))
11:      end
12:    case Singleton(t) then
13:      yield Singleton(t)
14:    case Content then
15:      yield Content(slice(source, node.start, node.end))
16:    case Skip then
17:      return
18:    end
19:  end
20: end
```

Рис. 2. Допоміжна функція Emit-Node-Tokens

```
1: generator Emit-Trailing(depth)
2:   top ← pop-from(interleaved, if : it ↦ it.depth ≥ depth)
3:   top ← top ?? return
4:   if top.last < top.end then
5:     yield Content(slice(source, top.last, top.end))
6:   end
7: end
```

Рис. 3. Допоміжна функція Emit-Trailing

```
1: function Update-Last-Pos(child-end, parent-depth)
2:   state ← find-top-in(interleaved, it ↦ it.depth = parent-depth)
3:   state ← state ?? return
4:   state.last ← child-end
5: end
```

Рис. 4. Допоміжна функція Update-Last-Pos

```
1: generator Emit-Gap(child-start, parent-depth)
2:   state ← find-top-in(interleaved, it ↦ it.depth = parent-depth)
3:   state ← state ?? return
4:   if state.last < child-start then
5:     yield Content(slice(source, state.last, child-start))
6:   end
7: end
```

Рис. 5. Допоміжна функція Emit-Gap

(а)
```
jdbc.url=jdbc:mysql://${application.server}:3306
```

(б)
```
<properties:file>
  <properties:prop>
    <properties:key>
      j
      db
      c
      <properties:key_sep/>
      url
    </properties:key>
    <properties:val>
      j
      db
      c
      :
      mys
      ql
      ://
      <properties:subst>
        <properties:key>
          application
          <properties:key_sep/>
          server
        </properties:key>
      </properties:subst>
      :
      3
      306
    </properties:val>
  </properties:prop>
</properties:file>
```

Рис. 6. Приклад токенізації (б) коду Java properties (а)

2.2. Перехід від математичної моделі до програмного коду. Після того, як буде завершено генерацію токенів, необхідно виконати детокенізацію, аби перетворити отриману послідовність на програмний код. Оскільки не будь-яка послідовність токенів відповідає синтаксично правильному коду, детокенізатор можна визначити як часткову функцію g з множини L^* послідовностей токенів у множину S кодів.

$$\mathrm{g}\,|_{L_{\text{valid}}} : L^* \to S \text{, визначену на } L_{\text{valid}} = \left\{ \mathbf{l} \in L^* : \exists s \in S : \mathrm{f}\left(s\right) = \mathbf{l} \right\} \quad (2)$$

Спеціальні токени вимагають спеціального алгоритму детокенізації. У випадку із XML-подібним представленням синтаксичного дерева, алгоритм має виконувати згортку токенів у синтаксичні структури відповідної мови. Для виконання цього детокенізатор має тримати внутрішній стан окремо для

кожної синтаксичної структури, що цього вимагає.

2.3. Подання архітектурних вимог. Важливим етапом розробки складної системи є її попереднє проєктування. Для представлення архітектури, що проєктується, прийнято, зокрема, використовувати уніфіковану мову моделювання (Unified Modeling Language — UML). Оскільки UML це мова візуального представлення на основі діаграм, вона є зручною для розуміння людиною, однак не для ВММ, оскільки ті адаптовані для роботи із послідовностями, а не із зображеннями. Існують мультимодальні моделі, натреновані розуміти зображеннями, але вони вимогливіші до обчислювальних ресурсів та вимагають додаткового навчання. До того ж, це робити не обов’язково. Діаграми UML, несуть одне й те саме формальне значення, незалежно від обраної для візуалізації кольорової палітри, форми вузлів, чи їхнього розташування на зображенні. Отже, UML діаграми можливо описати, використовуючи формальне текстове представлення, таке як PlantUML [23] чи Mermaid [24]. Це дозволяє зекономити обчислювальні ресурси.

Окрім питомо синтаксичних вузлів, прив’язаних до конкретної мови програмування (хоча деякі з них є типовими та присутні у різних МП), словник токенів можливо доповнити семантичними вузлами, що описують логічне значення певних структур у коді застосунку.

Семантичні вузли залежать не від МП, а від природи та архітектури застосунку. Можна виділити такі характерні сервісам семантичні вузли як об’єкт передачі даних, об’єкт доступу до даних, сервіс, репозиторій, сутність тощо, як і відповідні їм підвузли. Такі семантичні вузли можливо використовувати у поєднанні з фреймворками для розробки сервісів, наприклад, Spring Boot [25] для Java. Їхнє виокремлення вимагає додаткового аналізу програмного коду, але є гіпотетично можливим.

Одним із можливих варіантів цілісного подання як текстового опису, так і коду, разом з текстовим представленням діаграм архітектури, може бути формат текстової розмітки Markdown [26]. Код програми та діаграм можливо наводити як вкладені блоки коду, з позначенням відповідної мови у «info string». Код Markdown далі підлягає розбору в CST. Водночас розбір вкладених блоків коду виконується спираючись на ім’я мови, вказане у відповідному «info string».

На рис. 7 зображено результат (б) токенізації коду мовою Markdown, що містить блок коду мовою Java (а).

(а)

```
~~~java
public class Example {
  String foo() {
    return "Hello, World!";
  }
}
~~~
```

(б)

```
<markdown:document>
  <markdown:section>
    <markdown:code_block>
      <markdown:info_string>
        <markdown:language>
          java
        </markdown:language>
      </markdown:info_string>
      <markdown:code_block_content>
        <java:program>
          <java:class>
            <java:public/>
            <java:id>
              Example
            </java:id>
            <java:class_body>
              <java:method>
                <java:type_id>
                  String
                </java:type_id>
                <java:id>
                  foo
                </java:id>
                <java:params>
                </java:params>
                <java:block>
                  <java:return>
                    <java:str>
                      <java:str_content>
                        Hello
                        ,
                        GWorld
                        !
                      </java:str_content>
                    </java:str>
                  </java:return>
                </java:block>
              </java:method>
            </java:class_body>
          </java:class>
        </java:program>
      </markdown:code_block_content>
    </markdown:code_block>
  </markdown:section>
</markdown:document>
```

Рис. 7. Приклад токенізації (б) коду Markdown із блоком коду Java (а)

Основною відмінністю розробленого алгоритму токенізації порівняно із розглянутими раніше [8–18], зокрема тими, що теж явно токенізують вузли синтаксичного дерева [10, 13, 14, 16] є те, що токенізація дерева є контрольованою та адаптованою до особливостей граматики конкретної мови. А також текст природною мовою токенізується засобами токенізації природної мови, залишаючись у послідовності токенів частиною дерева, що дозволяє продовжувати використання унімодальних мовних моделей.

Таблиця 2.

Порівняння довжин послідовностей

	l_s	l_s/l_p	l_s/l_u	l_s/l_n
$E(X)$	503,05	2,14	1,72	2,6
$\sigma(X)$	362316,21	0,11	0,086	0,342
		l_u/l_p	l_u/l_n	l_p/l_n
$E(X)$		1,25	1,55	1,23
$\sigma(X)$		0,0041	0,148	0,087

У таблиці 2 наведено порівняння співвідношень між довжинами послідовностей токенів для кожного методу токенізації, а також між питомими кількостями байтів, що припадають на кожний токен, отриманий відповідним методом; l_s — довжина окремого прикладу коду з вибірки (у байтах), l_p — довжина послідовності токенів, побудованої обходом CST з використанням реєстру, l_u — довжина послідовності токенів, побудованої звичайним обходом CST, l_n — довжина послідовності токенів, отриманої токенізатором природної мови. Для токенізації коду як тексту природною мовою використано RoBERTa. Вимірювання проводилося на вибірці з 1000 прикладів набору даних Code-Search-Net (Java).

Як бачимо, токенізатори, що використовують обхід синтаксичного дерева, породжують більшу кількість токенів (мають меншу питому кількість байтів на токен), ніж токенізатори природної мови. Розроблений метод токенізації дозволяє використовувати меншу кількість токенів, ніж звичайний обхід дерева, і разом із цим дозволяє краще обробляти спеціальні випадки граматик.

Висновки

У даній статті розглянута проблема подання програмного коду компонентів сервісів в інформаційних системах провайдерів інфокомунікаційних послуг як математичної моделі. Запропоновано метод перетворення програмного коду компонентів сервісів на математичну модель і навпаки. Це дозволяє розробити інформаційну систему на базі моделі глибокого навчання для автоматизованої розробки програмного коду сервісів та підтримки їхнього життєвого циклу.

References

1. Zimmermann, T., Zeller, A., Weissgerber, P. and Diehl, S., (2005). Mining version histories to guide software changes. *IEEE Transactions on Software Engineering* [online]. **31**(6), 429–445. Available from: doi:10.1109/TSE.2005.72
2. Le, T. H. M., Chen, H. and Babar, M. A., (2020). Deep Learning for Source Code Modeling and Generation: Models, Applications, and Challenges. *ACM Comput. Surv.* [online]. **53**(3), article no: 62. Available from: doi:10.1145/3383458
3. Raychev, V., Vechev, M. and Yahav, E., (2014). Code completion with statistical language models. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* [online]. New York, NY, USA: Association for Computing Machinery. pp. 419–428. Available from: doi:10.1145/2594291.2594321
4. Vinyals, O., Fortunato, M. and Jaitly, N., (2017). *Pointer Networks* [online]. [Preprint]. Available from: doi:10.48550/arXiv.1506.03134
5. Li, J., Wang, Y., Lyu, M. R. and King, I., (2018). Code Completion with Neural Attention and Pointer Networks. In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18, 13–19 July 2018, Stockholm, Sweden* [online]. International Joint Conferences on Artificial Intelligence Organization. pp. 4159–4165. Available from: doi:10.24963/ijcai.2018/578
6. Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S. and Tarlow, D., (2017). *DeepCoder: Learning to Write Programs* [online]. [Preprint]. Available from: doi:10.48550/arXiv.1611.01989
7. Wang, Z., Chu, Z., Doan, T. V., Ni, S., Yang, M. and Zhang, W., (2024). *History, Development, and Principles of Large Language Models-An Introductory Survey* [online]. [Preprint]. Available from: doi:10.48550/arXiv.2402.06853

8. Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., Tufano, M., Deng, S. K., Clement, C., Drain, D., Sundaresan, N., Yin, J., Jiang, D. and Zhou, M., (2021). *GraphCodeBERT: Pre-training Code Representations with Data Flow* [online]. [Preprint]. Available from: doi:10.48550/arXiv.2009.08366

9. Jiang, X., Zheng, Z., Lyu, C., Li, L. and Lyu, L., (2021). TreeBERT: A tree-based pre-trained model for programming language. In: C. de Campos and M. H. Maathuis, eds. *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence* [online]. PMLR. pp. 54–63. [Viewed 9 May 2026]. Available from: https://proceedings.mlr.press/v161/jiang21a.html

10. Wang, X., Wang, Y., Mi, F., Zhou, P., Wan, Y., Liu, X., Li, L., Wu, H., Liu, J. and Jiang, X., (2021). *SynCoBERT: Syntax-Guided Multi-Modal Contrastive Pre-Training for Code Representation* [online]. [Preprint]. Available from: doi:10.48550/arXiv.2108.04556

11. Ahmad, W., Chakraborty, S., Ray, B. and Chang, K.-W., (2021). Unified Pre-training for Program Understanding and Generation. In: K. Toutanova, A. Rumshisky, L. Zettlemoyer, D. Hakkani-Tur, I. Beltagy, S. Bethard, R. Cotterell, T. Chakraborty and Y. Zhou, eds. *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* [online]. Association for Computational Linguistics. pp. 2655–2668. Available from: doi:10.18653/v1/2021.naacl-main.211

12. Wang, Y., Wang, W., Joty, S. and Hoi, S. C. H., (2021). *CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation* [online]. [Preprint]. Available from: doi:10.48550/arXiv.2109.00859

13. Niu, C., Li, C., Ng, V., Ge, J., Huang, L. and Luo, B., (2022). SPT-code: sequence-to-sequence pre-training for learning source code representations. In: *Proceedings of the 44th International Conference on Software Engineering, 21–29 May 2022, Pittsburgh, Pennsylvania* [online]. New York, NY, United States: Association for Computing Machinery. pp. 2006–2018. Available from: doi:10.1145/3510003.3510096

14. Guo, D., Lu, S., Duan, N., Wang, Y., Zhou, M. and Yin, J., (2022). UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In: S. Muresan, P. Nakov and A. Villavicencio, eds. *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 22–27 May 2022, Dublin, Ireland* [online]. Kerrville, TX, USA: Association for Computational Linguistics. pp. 7212–7225. Available from: doi:10.18653/v1/2022.acl-long.499

15. Chakraborty, S., Ahmed, T., Ding, Y., Devanbu, P. T. and Ray, B., (2022). NatGen: generative pre-training by "naturalizing" source code. In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* [online]. New York, NY, USA: Association for Computing Machinery. pp. 18–30. Available from: doi:10.1145/3540250.3549162

16. Tipirneni, S., Zhu, M. and Reddy, C. K., (2024). StructCoder: Structure-Aware Transformer for Code Generation. *ACM Trans. Knowl. Discov. Data* [online]. **18**(3), 1–20. Available from: doi:10.1145/3636430

17. Gong, L., Elhoushi, M. and Cheung, A., (2024). AST-T5: Structure-Aware Pretraining for Code Generation and Understanding. In: R. Salakhutdinov, Z. Kolter, K. Heller, A. Weller, N. Oliver, J. Scarlett and F. Berkenkamp, eds. *Proceedings of the 41st International Conference on Machine Learning, PMLR 235, 21–27 July 2024, Vienna, Austria* [online]. MLResearchPress. pp. 15839–15853. [Viewed 8 April 2026]. Available from: https://proceedings.mlr.press/v235/gong24c.html

18. Bartkowiak, P. and Graliński, F., (2025). Seamlessly Integrating Tree-Based Positional Embeddings into Transformer Models for Source Code Representation. In: H. Fei, K. Tu, Y. Zhang, X. Hu, W. Han, Z. Jia, Z. Zheng, Y. Cao, M. Zhang, W. Lu, N. Siddharth, L. {\O}vrelid, N. Xue and Y. Zhang, eds. *Proceedings of the 1st Joint Workshop on Large Language Models and Structure Modeling (XLLM 2025), Vienna, Austria* [online]. Kerrville, TX, USA: Association for Computational Linguistics. pp. 91–98. Available from: doi:10.18653/v1/2025.xllm-1.10

19. Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., Sundaresan, N., Zhou, M., Blanco, A. and Ma, S., (2020). *CodeBLEU: a Method for Automatic Evaluation of Code Synthesis* [online]. [Preprint]. Available from: doi:10.48550/arXiv.2009.10297
20. Hao, S., Sukhbaatar, S., Su, D., Li, X., Hu, Z., Weston, J. and Tian, Y., (2025). *Training Large Language Models to Reason in a Continuous Latent Space* [online]. [Preprint]. Available from: doi:10.48550/arXiv.2412.06769
21. Brunsfeld, M. et al., (2026). tree-sitter/tree-sitter [online]. Version 0.26.9. [computer software]. Available from: doi:10.5281/zenodo.20293153
22. Tree-sitter Grammars, (2025). tree-sitter-properties [online]. Version 0.3.0. [computer software]. [Viewed 2 April 2026]. Available from: https://github.com/tree-sitter-grammars/tree-sitter-properties
23. PlantUML, (2026). PlantUML [online]. Version 1.2026.2. [computer software]. [Viewed 2 April 2026]. Available from: https://github.com/plantuml/plantuml
24. mermaid-js, (2026). mermaid [online]. Version 11.14.0. [computer software]. [Viewed 2 April 2026]. Available from: https://github.com/mermaid-js/mermaid
25. Spring Boot [online], (2026). Version 4.0.5. [computer software]. [Viewed 2 April 2026]. Available from: https://spring.io/projects/spring-boot
26. Tree-sitter Grammars, (2026). tree-sitter-markdown [online]. Version 0.5.3. [computer software]. [Viewed 2 April 2026]. Available from: https://github.com/tree-sitter-grammars/tree-sitter-markdown/

Дата першого надходження до видання: 12.06.2026
Внутрішня рецензія отримана: 01.07.2026
Зовнішня рецензія отримана: 06.07.2026
Дата прийняття до друку: 10.08.2026
Дата публікації: 29.08.2026

Про авторів:

Вітковський Данило Олександрович,
аспірант
Vitkovskyi Danylo,
post-graduate student
https://orcid.org/0009-0004-9809-403X

Шимкович Володимир Миколайович,
кандидат технічних наук, доцент
Shymkovych Volodymyr,
Ph.D. (technical sciences), associate professor
https://orcid.org/0000-0003-4014-2786

Місце роботи авторів:

Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського»,
National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute"
проспект Берестейський 37.
E-mail:
vitkovskyi.d.o.-it51f@edu.kpi.ua
v.shymkovych@kpi.ua