УДК 004.416.6:004.855
DOI https://doi.org/10.24144/2616-7700.2025.47(2).148-167

О. Г. Когуч[1], Я. М. Матвійчук[2]

[1] Національний університет «Львівська політехніка»,
аспірант кафедри Систем штучного інтелекту ІКНІ
oksana.h.kohuch@lpnu.ua
ORCID: https://orcid.org/0009-0004-6861-8082

[2] Національний університет «Львівська політехніка»,
професор кафедри Систем штучного інтелекту ІКНІ,
доктор технічних наук
yaroslav.m.matviychuk@lpnu.ua
ORCID: https://orcid.org/0000-0002-5570-182X

МЕТОДИ МАШИННОГО НАВЧАННЯ В МОДЕРНІЗАЦІЇ ПРОГРАМНИХ КОМПОНЕНТІВ КОМПЛЕКСНИХ ІНФОРМАЦІЙНИХ СИСТЕМ

Значна кількість сучасних державних і приватних організацій використовують інформаційні системи, побудовані на основі технологій попередніх поколінь, із застосуванням програмних компонентів, створених ще десятки років тому. Такі системи є критично важливими для функціонування бізнесу, однак їхнє оновлення й підтримка вимагають значних людських, часових та фінансових ресурсів, а традиційні підходи до модернізації в більшості випадків виявляються надмірно тривалими, складними для управління та пов'язаними з високим рівнем ризиків. Це зумовлює потребу у нових технологічних підходах, здатних зменшити вартість та пришвидшити процес трансформації без втрати надійності.

Методи машинного навчання (ML) постають як перспективний інструмент, здатний частково автоматизувати ключові етапи модернізації програмних систем. У статті систематизовано сучасні напрями їх застосування: від діагностики технічного стану та виявлення прихованих залежностей у кодовій базі до автоматизованого переписування окремих модулів і тестування оновлених фрагментів. Особливу увагу приділено аналізу сильних сторін і наявних обмежень цих підходів, а також технологічних і практичних викликів, які супроводжують їх упровадження. Доведено, що результативність ML-рішень безпосередньо визначається масштабом системи та якістю доступних вихідних даних.

Робота ґрунтується на теоретичному аналізі та результатах практичного оцінювання ефективності використання ML. Окремий акцент зроблено на питаннях інтеграції подібних технологій у робочі процеси, адаптації команд програмістів до нової парадигми та перспективам подальших досліджень у цій сфері. Також розглядаються формалізовані метрики, що дають змогу об'єктивно оцінити успіх модернізації.

Дослідження демонструє, що комплексне поєднання сучасних ML-методів із перевіреними традиційними підходами зменшує витрати на підтримку застарілих систем і забезпечує поступовий перехід до сучасних архітектур за умови ретельного контролю ризиків та валідації результатів. Такий підхід може слугувати основою для масштабованої й сталої стратегії цифрової трансформації організацій.

Ключові слова: монолітні застарілі системи, модернізація монолітних систем, машинне навчання у модернізації систем, автоматизація програмної модернізації.

1. Вступ та постановка проблеми. Попри динамічний розвиток сучасних інформаційних технологій (ІТ), у багатьох стратегічно важливих галузях (фінансах, охороні здоров'я, державному управлінні та промисловості) все ще функціонують монолітні, застарілі інформаційні системи. Створені на основі технологій минулих десятиліть, ці рішення залишаються критично важливими для

діяльності організацій, хоча з технічного й архітектурного погляду давно втратили актуальність. Їм притаманні складна бізнес-логіка, тісна взаємозалежність компонентів, недостатня документація та визначальна роль у ключових бізнес-процесах. Потреба в їх модернізації є очевидною, проте реалізація цього процесу супроводжується численними технічними, організаційними та економічними викликами.

Традиційні підходи, зокрема ручний рефакторинг і повне переписування коду, мають обмежену масштабованість, потребують залучення висококваліфікованих фахівців, добре обізнаних зі старими технологіями, і пов'язані з високими ризиками порушення стабільності. Модернізацію додатково ускладнюють приховані залежності, застарілі архітектурні патерни, наявність анти-патернів, тривалі цикли тестування та нестача автоматизованих засобів перевірки якості.

У цьому контексті методи машинного навчання розглядаються як потенційно ефективний інструмент для часткової автоматизації окремих етапів оновлення систем. Зокрема, великі мовні моделі (LLM) і трансформери, доповнені доменними адаптерами та векторними поданнями коду, здатні точніше відображати специфіку застарілого програмного забезпечення. Графові нейронні мережі (GNN) показують перспективні, але переважно експериментальні результати у виявленні структурних залежностей та визначенні природних «ліній розрізу» між модулями. Seq2Seq-архітектури можуть забезпечити автоматичний переклад і переформатування фрагментів коду зі збереженням їхньої семантики, що є необхідним для подальшої міграції, проте їхня точність на великих (> 1 MLOC) модулях ще не підтверджена. Методи навчання з підкріпленням (RL) оптимізують генерацію тестів і верифікацію змін, однак поки застосовуються здебільшого на сервісах до 50-500 kLOC. У сукупності ці ML-рішення демонструють обнадійливі результати в задачах розпізнавання шаблонів, реконструкції архітектурної логіки, виявлення внутрішніх залежностей та автоматизованої генерації коду, але їхня ефективність на гігантських легасі-кодових базах лишається предметом подальших досліджень.

Водночас інтеграція ML-інструментів у практику модернізації створює низку процесно-організаційних викликів. Необхідно адаптувати чинні інженерні практики та робочі процеси до нової парадигми, в якій результати мають ймовірнісний характер, переглянути цикли прийняття рішень, критерії якості та механізми контролю змін, забезпечити безперервний моніторинг та пояснюваність ML-моделей у задачах модернізації, а також підтримувати прозорість і відтворюваність результатів, що є критично важливими для функціонування інформаційних систем.

Таким чином постає комплексна науково-практична задача: систематизувати й критично проаналізувати сучасні ML-підходи для модернізації інформаційних систем; оцінити їхній потенціал та обмеження на різних масштабах коду; сформулювати перевірні й відтворювані метрики успіху впровадження; і окреслити межі ефективного застосування в умовах реального функціонування різноманітних інформаційних середовищ. Розв'язання цієї задачі потребує глибокого вивчення наявних методів, публічних бенчмарків, та їхньої адаптації до специфіки застарілої програмно-організаційної інфраструктури.

Мета та завдання. Метою дослідження є комплексна систематизація та критичний аналіз сучасних методів машинного навчання в контексті їх застосу-

вання для модернізації компонентів великих монолітних інформаційних систем. Дослідження передбачає узагальнення підтверджених у літературі та промислових пілотах результатів щодо ефективності ML-підходів, оцінювання їхніх переваг і обмежень з урахуванням масштабованості ($\leq/\geq$ 1 MLOC), інфраструктурних витрат і пояснюваності, виявлення наукових прогалин, зокрема нестачі відкритих бенчмарків для систем $>$ 1 MLOC та неузгодженості метрик, формулювання перспективних напрямів подальших досліджень.

Наукова новизна. Проведене дослідження має теоретичну і прикладну значущість у контексті використання методів машинного навчання для модернізації програмних компонентів складних інформаційних систем. Наукова новизна дослідження визначається такими результатами:

- розроблено методологічну схему узгодження класів ML-моделей з етапами процесу модернізації (аналіз – трансформація – верифікація), що забезпечує системність результатів;
- проведено апробацію (пілотну оцінку) GraphCodeBERT, MonoEmbed-GNN, CodeT5+, CodeT5-DA та DeepREST-RL на великому кодовому репозиторії ($\approx$ 850 kLOC), що дозволило оцінити їхню ефективність у завданнях виявлення технічного боргу, структурної декомпозиції, автоматизованого рефакторингу та тестування в контексті запропонованої методологічної схеми;
- запропоновано формалізований набір метрик (SES, ΔTD, ΔMTTR) для кількісного оцінювання результатів застосування ML-підходів у процесі модернізації програмного забезпечення;
- уточнено орієнтовні (не нормативні) діапазони метрик ефективності (SES = 0,15–0,45; ΔTD $\geq$ 10 %; ΔMTTR $\geq$ 10 %), які можуть бути використані як початкові орієнтири для оцінювання результативності ML-модернізацій у промислових умовах.

Отримані результати розширюють наукові уявлення про застосування методів машинного навчання у завданнях технічного оновлення складних інформаційних систем і формують підґрунтя для подальшої стандартизації підходів до оцінювання ефективності таких процесів.

Аналіз останніх досліджень і публікацій. У сучасній науковій літературі модернізація програмного забезпечення все частіше розглядається крізь призму використання машинного навчання. Такий підхід демонструє потенціал у зниженні технічного боргу, підвищенні підтримуваності коду та спрощенні переходу до сучасних архітектур. Дослідження у цій галузі охоплюють чотири основні напрями застосування ML: виявлення архітектурних дефектів, декомпозиція монолітів, трансформація коду з урахуванням семантики та автоматизоване тестування [3, 5, 6]. У більшості праць ML-моделі розглядаються як засоби часткової автоматизації процесів модернізації.

У прикладних дослідженнях описано ефективність конкретних моделей у вирішенні окремих задач. Великі мовні моделі або трансформери застосовуються переважно для аналізу кодової бази, виявлення «запахів коду», дублювань і шаблонів проектування [4, 8, 10]. У свою чергу, графові нейронні мережі довели свою придатність у декомпозиції монолітних систем, зокрема шляхом кластеризації залежностей і пошуку меж модулів [3, 7]. Seq2Seq-моделі, зокрема архітектури трансформерного типу, використовуються для автоматизованого переписування коду зі збереженням його семантики (через тести/специфікації),

що сприяє переходу до більш сучасних стилів реалізації [5]. Методи з підкріпленням застосовуються в задачах генерації тестів, де їхньою перевагою є здатність адаптуватися до особливостей потоку керування (control flow) й оптимізувати покриття. Водночас література фіксує і певні обмеження таких рішень. Серед них нестабільність результатів, помилки генерації коду, порушення семантики при трансформації та відсутність пояснень, що знижує довіру до результатів моделей. [1, 5].

Попри схожі результати, різні автори по-різному оцінюють масштабованість таких моделей: у [3] та [7] декомпозиція монолітів розглядається як цілком розв'язна задача, тоді як у [13] наголошено на низькій відтворюваності результатів у промислових умовах. Це свідчить про відсутність єдиної позиції щодо готовності ML-підходів до реального використання. Крім того, порівняння робіт [5] і [10] показує, що навіть для однакових типів архітектур (трансформерів) результати суттєво відрізняються залежно від домену коду, розміру корпусу та ступеня ручної пост-обробки. Наявні дослідження демонструють перспективність ML-рішень, але ще не підтверджують їх універсальності.

Окремо увага дослідників зосереджується на питаннях оцінки ефективності застосованих підходів. У ряді праць запропоновано формалізовані, але різноспрямовані метрики для оцінювання результатів модернізації, включно з оцінкою технічного боргу, модульності архітектури та стабільності змін [6, 12]. Проте наразі не існує єдиного підходу до визначення і валідації порогових значень таких метрик, що ускладнює порівняння результатів у різних дослідженнях. Це відображає ширшу проблему відсутності уніфікованої теоретичної основи для модернізації систем, що неодноразово підкреслювалася в концептуальних дослідженнях [9].

Додаткові виклики пов'язані з обмеженою масштабованістю сучасних моделей на великих кодових базах, відсутністю репрезентативних публічних датасетів і значними інфраструктурними витратами на обробку [8, 11, 13]. Серед інших практичних бар'єрів є виявлення структурних проблем, таких як повторювані фрагменти з однаковими наборами змінних у різних функціях, що важко ідентифікувати без аналізу ширшого контексту коду. У таких випадках ефективними є ML-рішення, які враховують проєктний рівень закономірностей, зокрема, каскадні підходи до виявлення й автоматичного рефакторингу [10]. У відповідь на це в окремих роботах пропонуються рішення, спрямовані на підвищення пояснюваності моделей, адаптацію до специфіки застарілого коду та створення відкритих еталонних бенчмарків. Такі ініціативи можуть стати основою для стабільної інтеграції ML-інструментів у практику модернізації складних програмних систем.

Таким чином, сучасна наукова література демонструє суттєвий прогрес у використанні ML-методів для модернізації ПЗ, але також виявляє низку невирішених проблем: від нестачі єдиних метрик і масштабованих бенчмарків до відсутності узгодженої концептуальної рамки для порівняння результатів різних досліджень. Це створює об'єктивне підґрунтя для подальших робіт, спрямованих на формування узгодженої теоретичної та методичної бази модернізації програмних систем.

Виклад основного матеріалу. Більшість великих компаній і державних установ у фінансовому, телекомунікаційному та промисловому секторах зали-

шається залежною від монолітних інформаційних систем, створених кілька десятиліть тому. Їхній програмний код спирається на застарілі бібліотеки, фрагментарні оновлення та численні ручні модифікації, що істотно ускладнює забезпечення стабільності, масштабованості та подальшого розвитку таких систем. Технічна документація переважно неповна або втратила актуальність, що створює додаткові труднощі під час модернізації, тестування та інтеграції з сучасними технологічними платформами. Унаслідок цього знижується ефективність упровадження інноваційних методів розроблення програмного забезпечення та переходу до сучасних архітектурних підходів.

2. Характеристика проблемного поля. З часом в інформаційній системі накопичився значний технічний борг: численні «запахи коду» (наприклад, надмірно великі класи чи дублювання функціоналу) ускладнюють внесення змін, роблячи їх трудомісткими та ризикованими. Монолітна архітектура створює щільні міжмодульні залежності, тому модифікація окремого компонента може призвести до збоїв у роботі цілих підсистем, а кожен новий реліз вимагає високовартісного регресійного тестування. Це узгоджується з висновками [1], де підкреслено, що глибокі історичні шари коду значно ускладнюють його підтримку і унеможливлюють швидке оновлення за допомогою ручних методів рефакторингу. Додаткову складність створює кадровий чинник: первинні розробники вже покинули компанію, а новим спеціалістам потрібні тривалі періоди адаптації для розуміння кодової бази. Окрім цього, застаріла архітектура перешкоджає масштабуванню та ускладнює забезпечення безпеки.

Існує низка відомих стратегій модернізації, проте кожна має відчутні обмеження. Повне переписування системи («Big-Bang Rewrite») теоретично дозволяє реалізувати сучасну архітектуру, однак вимагає багаторічних інвестицій і створює ризик тимчасового паралічу бізнес-процесів у період до запуску нової версії. Поступовий підхід за шаблоном «Strangler Fig» знижує ризики завдяки етапному винесенню функціональності у мікросервіси, однак вимагає трудомісткого аналізу меж сервісів, написання тестів і забезпечення сумісності. Як зазначено в [2], ефективна декомпозиція в мікросервіси можлива лише за наявності чіткої інтерпретації меж сервісів, що стає складним без автоматизованих інструментів аналізу коду та залежностей. Традиційний ручний рефакторинг залежить від обмеженої кількості експертів, яким складно гарантувати сталість стандартів якості та відслідковувати вплив змін на архітектуру. Ці обмеження також висвітлено в [13], де наголошується, що масштабні рефакторингові проєкти без підтримки автоматизованих засобів практично некеровані в умовах складної архітектури. У підсумку, такі підходи виявляються недостатньо масштабованими для систем, що охоплюють сотні тисяч або мільйони рядків коду, і тому зростає потреба в автоматизованих засобах, здатних зменшити навантаження на людські ресурси.

Методи машинного навчання пропонують ефективні засоби підтримки модернізації в кількох критичних напрямах. Кодові трансформер-моделі (наприклад, CodeBERT чи GraphCodeBERT), доповнені доменними адаптерами, дозволяють ідентифікувати архітектурні патерни та зони технічного боргу, формуючи пріоритетні списки для рефакторингу. За результатами останніх досліджень точність виявлення «запахів коду» зростає на 10–15% для репозиторіїв < 300 kLOC, але ефективність на базах > 1 MLOC поки не доведена. Графові

нейронні мережі (GNN), як-от MonoEmbed-GNN, аналізують структуру коду та пропонують природні межі декомпозиції у мікросервіси, проте поки їх випробувано лише на проєктах до 200 kLOC. Seq2Seq-моделі, наприклад CodeT5+ та CodeT5-DA, трансформують код у сучасні стилі або інші мови програмування, зберігаючи семантичну точність на функціях до 200 рядків. Методи навчання з підкріпленням (RL), такі як DeepREST, автоматично генерують тести, підвищуючи покриття на 15–18 % у невеликих сервісах. У сукупності ці інструменти охоплюють повний цикл модернізації (від діагностики до трансформації та верифікації) й можуть суттєво зменшити тягар на команду розробки, але потребують масштабних реплікацій, щоб підтвердити стійкість результатів у промислових умовах. Систематизований огляд таких рішень у [6] підтверджує їх доцільність для архітектурної трансформації у ряді промислових сценаріїв, але з істотними застереженнями щодо масштабу та домену.

Втім, інтеграція ML-рішень не позбавлена складнощів. LLM-моделі можуть генерувати некоректний або нефункціональний код особливо за наявності доменного зсуву чи дефіциту якісних даних. GNN-моделі іноді створюють нестабільні варіанти декомпозиції в разі неінформативних або нерівномірно зважених залежностей. Seq2Seq-підходи можуть породжувати синтаксично правильні, але семантично неточні трансформації, що вимагає залучення розробника. RL-моделі демонструють нестійкість за неправильно визначеної функції винагороди, що веде до слабкого тестового покриття. Як наголошено в [1], обмеження у якості даних та нестабільність генерації коду залишаються ключовими бар'єрами для промислового застосування ML-моделей. Попри технічні виклики, зберігається проблема обмеженої пояснюваності моделей, що знижує рівень довіри з боку бізнесу. Водночас застосування ML зумовлює зростання сукупної вартості володіння системою (total cost of ownership, TCO), оскільки потребує додаткових ресурсів на підтримку, адаптацію та інтеграцію, що ставить під питання економічну доцільність таких рішень.

Незважаючи на згадані обмеження, машинне навчання залишається перспективним напрямом модернізації програмного забезпечення. У ситуаціях, коли монолітні застарілі системи стримують розвиток IT-інфраструктури, а традиційні методи модернізації є занадто повільними й затратними, ML-підходи відкривають шлях до швидшої, гнучкішої та безпечнішої трансформації. Вони дають змогу автоматизувати аналіз, спростити декомпозицію, забезпечити верифікацію змін і суттєво зменшити навантаження на інженерні команди за умови чітко визначених меж зовнішньої валідності та прозорих процедур контролю якості.

3. Методологічні основи впровадження ML у модернізацію. Методологічні засади інтеграції методів машинного навчання в життєвий цикл модернізації інформаційних систем передбачають поетапне зіставлення типових задач з відповідними класами ML-моделей. Добір моделей машинного навчання здійснювався за трьома критеріями:

- Спеціалізація на програмному коді та його структурі, що передбачає навчання або адаптацію моделей до домену програмного забезпечення;
- Наявність емпірично підтверджених, відкрито опублікованих результатів у контексті задач модернізації, зокрема на основі відкритих реалізацій або бенчмарків;

- Функціональна відповідність етапам модернізації, тобто здатність моделі розв'язувати завдання, пов'язані з аналізом, трансформацією або верифікацією коду.

Основу аналізу становить визначення відповідного класу моделей для кожного етапу (діагностики технічного стану, трансформації архітектури та верифікації змін) та очікуваний ефект від використання моделей.

Таблиця 1.

Узгодження завдань модернізації з класами моделей ML та очікуваними результатами їх застосування

Завдання	**Клас моделей**	**Технологічне рішення**	**Чому обрано саме цю модель**	**Очікуваний ефект**	**Перевірені альтернативи**
Виявлення технічного боргу	Модифіковані трансформенні моделі доповнені доменним адаптером	*GraphCode BERT*	Добре узагальнює стилістичні шаблони та графовий контекст, а доменний адаптер підвищує точність на досліджуваному репозиторії	10–15 % зростання точності виявлення «запахів коду»	CodeT5+, StarCoder-Base, PolyCoder, DeepSmells
Декомпозиція моноліту	Графові нейронні мережі	*MonoEmbed-GNN*	Враховує глобальну структуру графа викликів, що дає кращу модулярність (Modularity Score)	+15% зростання показника модульності	GIN-VN, Graph2 Micro, CodeGraph BERT
Автоматична міграція коду	Seq2Seq архітектури	*CodeT5+/* CodeT5-DA	Encoder-decoder трансформує послідовність токенів у цільовий стиль мови та підтримує перевірку семантичної еквівалентності	10–15 % економії часу на ручне перенесення функцій	AlphaCode, PLBART, InCoder, SantaCoder
Автоматична генерація тестів	Навчання з підкріпленням (RL)	*DeepREST*	Агент RL дозволяє формулювати покриття тестами як функцію винагороди й поступово вдосконалювати стратегії тест-генерації	15–20 % зростання покриття тестами при незмінному бюджеті виконання	RLSQM, CURE, TestART

Очікувані ефекти наведено за опублікованими експериментами 50-300 kLOC, проте узагальнення на репозиторії > 1 MLOC потребує додаткових досліджень.

Для обґрунтованого впровадження ML-методів у процес модернізації недостатньо лише зіставлення моделей із типами завдань. Необхідно також визначити, за якими критеріями оцінювати їх ефективність у практичному застосуванні. Це дає змогу контролювати якість виконання й реалістично формалізувати очікувані результати. Нижче подано перелік ключових метрик успіху, які можуть слугувати орієнтирами під час оцінювання впливу ML-підходів на різні етапи модернізації.

Таблиця 2.

Метрики успіху та їх порогові значення в модернізації ПЗ з використанням ML

Метрика	**Формула**	**Порогове значення**
SES (Software Evolution Score)	SES = 0,4*ΔCoverage + 0,6*ΔModularity	0,15–0,45
ΔTD (Change in Technical Debt)	ΔTD = (TD_before − TD_after) / TD_before	≥ 10 %
ΔMTTR (Change in Mean Time to Repair)	ΔMTTR = (MTTR_before − MTTR_after) / MTTR_before	≥ 10 %

Пояснення метрик:

- ΔCoverage — приріст рядкового або гілкового покриття тестами після змін, у відсотках.
- ΔModularity — зміна показника модульності графа викликів, у відсотках.
- SES - інтегральна метрика корпоративного рівня, що комбінує покриття та модульність. Коефіцієнти 0,4 та 0,6 відображають середню вагу цих складників у трьох внутрішніх кейс-стаді; значення 0,15-0,45 відповідає реально досяжним приростам (≈ 15-20 % кожної складової).
- ΔTD — відносне зменшення технічного боргу (вимірюється в людиногодинах). Порогове ≥ 10 % узгоджується з медіаною промислових пілотів, а > 20 % лишається ціллю.
- ΔMTTR — відносне скорочення середнього часу на відновлення працездатності системи після змін. 10 % вважається мінімально відчутним ефектом, а вищі значення залежать від автоматизації виробничого циклу.

Порогові значення метрик SES, ΔTD і ΔMTTR залишаються робочими гіпотезами, що ґрунтуються на огляді відкритих кейсів і внутрішньому досвіді кількох компаній. Вони не є нормативом і мають коригуватися під конкретний домен, масштаб кодової бази й доступний бюджет інфраструктури.

Попри наявність формалізованих метрик успіху, інтеграція ML-рішень у процес модернізації систем залишається складним і багатовимірним завданням. На практиці впровадження супроводжується низкою ризиків (як технічних, так і організаційних), що можуть істотно вплинути на якість результатів або навіть зірвати ініціативу. Нижче узагальнено типові ризики, які виникають під час впровадження ML у задачах модернізації, а також окреслено можливі шляхи їхньої мінімізації.

Таким чином, методологічна інтеграція машинного навчання в процес модернізації інформаційних систем передбачає не лише вибір моделей і визначення метрик ефективності, а й проактивне врахування супутніх ризиків з готовністю до їхнього проактивного усунення. Зіставлення завдань із відповідними ML-архітектурами, застосування формалізованих метрик успіху, а також ідентифікація типових викликів і способів їх мінімізації створюють основу для побудови цілісної стратегії впровадження рішень штучного інтелекту в практику модернізації. Це дає змогу забезпечити обґрунтованість рішень, керованість процесу

Таблиця 3.

Потенційні ризики і стратегії мінімізації при застосуванні ML у задачах модернізації

Ризик	**Технологічне рішення**	**Причина невдачі**	**Стратегія мінімізації**
Хибна класифікація	GraphCodeBERT + LoRA	Неповні тренувальні дані, доменний зсув	Донавчання на доменних даних, вибірках, фільтрація за правилами, ручний аудит згенерованого коду
Некоректна декомпозиція	MonoEmbed-GNN	Нерівномірна вага ребер, мала вибірка класів	Нормалізація ваг зв'язків, перехресна перевірка декомпозиції на різних підсистемах, фільтрація за порогом впевненості
Порушення семантичної еквівалентності	CodeT5+ та CodeT5-DA	Обмеження контексту вхідного коду, недостатня перевірка семантики, перенавчання на синтаксисі	Розширення контекстного вікна, автоматична перевірка та тестування на властивостях
Низьке покриття тестами	DeepREST RL	Невдала функція винагороди, відсутність еталонного середовища	Переформулювання мети навчання, поетапне навчання, моделювання середовища виконання
Пояснюваність результатів	Усі моделі	Відсутність доступних атрибутів для пояснення	Використати методи локальних і глобальних пояснень (LIME, SHAP) ведення журналу рішень, визначення зони відповідальності моделей
Зростання експлуатаційних витрат	Усі моделі	Часті запити до моделей, відсутність кешування	Використання полегшених мовних моделей, пакетних обробок запитів, офлайн-режим роботи моделей, використання економних GPU-серверів.

трансформації та підвищення надійності отриманих результатів у прикладному середовищі.

4. Експериментальна оцінка ефективності ML-методів у модернізації. На основі проведеного аналізу узагальнено досвід застосування методів машинного навчання на ключових етапах модернізації інформаційних систем. Процес аналізу охоплює три послідовні фази: аналіз, трансформацію та верифікацію. Для кожного етапу підібрано відповідні ML-моделі з урахуванням їх функціональних властивостей, а також оцінено їхню практичну ефективність на основі релевантних метрик у робочих умовах. Моделі були протестовані на реальному програмному репозиторії обсягом понад 850 тисяч рядків Java/Python коду ($\approx$ 850 kLoC), що дозволило оцінити їхню ефективність у масштабному і практично значущому середовищі. Отримані результати демонструють, що використання машинного навчання може суттєво підвищити рівень автоматизації та пришвидшити виконання завдань модернізації.

4.1. Аналіз вхідної системи. На цьому етапі головним завданням є виявлення технічного боргу, архітектурних аномалій та критичних залежностей між модулями. Отримані результати формують базу для пріоритезації змін і мінімізації ризиків.

Таблиця 4.

Результати застосування ML-моделей на етапі аналізу застарілої системи

Завдання	**Технологічне рішення**	**Аргументація**	**Результат ефективності**
Виявлення технічного боргу	GraphCodeBERT + LoRA-адаптер	Навчена на масштабному корпусі коду; виявляє дублювання, надмірну складність і антипатерни, що знижують підтримуваність	ΔTD $\approx$ 11–17%, Maintainability Index $\approx$ +13 %
Визначення структурних залежностей	MonoEmbed-GNN	GNN Аналізує граф викликів залежностей і класифікує вузли, що дозволяє локалізувати щільні кластери та природні межі між модулями	ΔModularity $\approx$ +15-40 % (0.38$\rightarrow$0.53)

Пояснення метрик. ΔTD — відносне зменшення технічного боргу; Maintainability Index — інтегральний показник підтримуваності; ΔModularity — приріст модульності графа викликів.

4.2. Трансформація архітектури. Фаза трансформації охоплює автоматизований рефакторинг і переклад коду відповідно до цільової архітектури. Ключовою вимогою є збереження семантичної еквівалентності між вихідною та оновленою версіями.

Таблиця 5.

Результати застосування ML-моделей на етапі трансформації застарілої системи

Завдання	**Технологічне рішення**	**Аргументація**	**Результат ефективності**
Переписування фрагментів коду	CodeT5+ та CodeT5-DA (Seq2Seq)	Автоматично транслює код на іншу мову або сучасний стиль, гарантуючи збереження семантики (через тести та специфікації)	Semantic Preservation Rate (SPR) $\approx$ 0.88 $\pm$0.02, ΔMTTR $\approx$ 5-8 % при SPR $\approx$ 0.88 та ΔMTTR $\approx$ 8-11 % при SPR $\approx$ 0.90

Пояснення метрики. Semantic Preservation Rate (SPR) — це показник, що відображає частку коду, у якому після трансформації повністю збережено функціональну еквівалентність де вихідні і цільові версії мають однакову поведінку при ідентичному вхідному наборі включно зі збереженням сигнатур та структури викликів. ΔMTTR — це відносне скорочення середнього часу відновлення системи внаслідок зменшення кількості та складності помилок після рефакторингу.

4.3. Верифікація змін. Заключна фаза перевіряє, що модернізована система зберігає функціональну стабільність. Автоматизоване тестування зменшує ризик регресій і скорочує час випуску.

Пояснення метрик. ΔCoverage — це відносна зміна частки коду, охопленого рядковим/гілковим покриттям, після автогенерації сценаріїв; F1-measure —

Таблиця 6.

Результати застосування ML-моделей на етапі верифікації змін при модернізації застарілої системи

Завдання	**Технологічне рішення**	**Аргументація**	**Результат ефективності**
Доповнення тестів	CodeT5-DA	Автоматично генерує додаткові модульні тести на основі сигнатур методів і коментарів, розширюючи покриття коду без порушення компіляції.	ΔCoverage $\approx$ +10-12 %, Mutation Score $\approx$ +8-10 %, Compile Rate $\approx$ 97 %
Генерація тестів	DeepREST-RL	Генерує тести з урахуванням найбільш ризикованих шляхів виконання, оптимізуючи покриття критичних гілок коду в умовах обмежених ресурсів часу та обчислень	ΔCoverage $\approx$ +15-18%, F1-measure $\approx$ 0.84 $\pm$0.02

узагальнений показник ефективності автоматичної генерації тестів, що поєднує повноту (recall) і точність (precision), і слугує інтегральною метрикою якості згенерованих тестів. Mutation Score — показує частку штучно внесених помилок (mutants), які були виявлені тестами, і відображає здатність тестів виявляти логічні помилки в коді. Compile Rate — частка згенерованих тестів, що успішно компілюються без ручного виправлення, і характеризує стабільність роботи моделі під час генерації коду тестів.

Запропонований ланцюжок утворює логічно зв'язаний цикл модернізації від первинного аналізу технічного боргу до фінального етапу перевірки внесених змін. У сукупності на кейсі 850 kLOC це дало: ΔTD $\approx$ 15 %, ΔModularity +0,15 п.п, ΔCoverage $\approx$ 15 %, ΔMTTR $\approx$ 6-9 %. Така комбінація помітно знижує навантаження на команду, підвищує якість коду та скорочує загальний час модернізації хоч її масштабованість на системах $>$ 1 MLOC ще потребує підтвердження.

4.4. Інтегроване технічне рішення та експериментальна конфігурація. Для перевірки ефективності розробленого підходу створено інтегрований конвеєр машинного навчання, який об'єднує етапи аналізу, кластеризації, рефакторингу, міграції та тестування програмного коду. Конвеєр складається з таких основних компонентів: GraphCodeBERT, MonoEmbed-GNN, CodeT5+, CodeT5 DA та DeepREST-RL. Кожна модель відповідає за окрему фазу процесу модернізації, забезпечуючи поступовий перехід від виявлення проблем коду

до його автоматичного вдосконалення.

Експериментальну частину проведено на двох промислових відкритих репозиторіях: TensorFlow (≈880 тис. рядків коду на Python) і Apache Hadoop (≈872 тис. рядків коду на Java). Для аналізу було використано репрезентативну частину вихідного коду, що охоплює ключові компоненти систем. Обидві системи характеризуються великою кількістю тестів, складною архітектурою та добре підтримуваною інфраструктурою, що робить їх придатними для оцінки масштабованості й відтворюваності результатів.

Етап 1. Аналіз технічного боргу: На початковому етапі дослідження виконувалося виявлення технічного боргу, дублікатів коду та циклічних залежностей у великих кодових базах Python і Java.

Для цього застосовано модель GraphCodeBERT, донавчену із використанням Low-Rank Adaptation (LoRA), що дозволило скоротити кількість оновлюваних параметрів приблизно на 95 % порівняно з повним fine-tuning. Такий підхід зменшив обчислювальні витрати без втрати точності.

Архітектура моделі GraphCodeBERT побудована на основі RoBERTa-base з додатковим компонентом Graph Guided Attention. Конфігурація моделі включала: 12 шарів трансформера, 12 голів уваги, hidden size = 768. Для LoRA використано параметри: r = 8, б = 16, dropout = 0.05, що забезпечило баланс між узагальнювальною здатністю моделі та стабільністю градієнтів. Оптимізація здійснювалася методом AdamW (LR = 2×10^{-5}, batch size = 8), навчання тривало 8 епох до повної збіжності без ознак перенавчання.

Python-проєкт TensorFlow:

Для Python-репозиторіїв TensorFlow модель GraphCodeBERT із LoRA-адаптацією аналізувала графи викликів (CG), гафи потоку даних (DFG) та абстрактні синтаксичні дерева (AST) для ідентифікації дублювання коду, циклічних залежностей і ділянок із підвищеною когнітивною складністю. Середній час навчання становив ≈10 годин GPU, а F1-міра досягла 0,93, що свідчить про високу точність виявлення технічного боргу у Python-кодових базах.

Java-проєкт Apache Hadoop: Для Java-проєкту Apache Hadoop застосовано аналогічну конфігурацію GraphCodeBERT + LoRA. Модель обробляла представлення коду у вигляді CFG, DFG і AST-графів для виявлення дублікатів і циклічних залежностей між класами та пакетами. Середній час навчання склав ≈9,5 годин GPU, а F1-міра - 0,92 (на валідаційній вибірці), що підтверджує стабільність і переносимість підходу для різних мовних екосистем.

Узагальнення результатів: Використання GraphCodeBERT із LoRA-адаптацією забезпечило:

- високу якість виявлення технічного боргу (F1 ≈ 0,92–0,93);
- ефективне виявлення дублювання коду, циклічних залежностей і складних фрагментів у великих кодових базах.

Отримані результати підтверджують придатність моделі GraphCodeBERT+LoRA для використання у системах автоматизованого аудиту якості програмного забезпечення та як базового етапу подальшої архітектурної сегментації й рефакторингу коду.

Етап 2. Архітектурна кластеризація: На другому етапі дослідження здійснювалася структурна кластеризація архітектури програмних систем із метою виявлення внутрішніх залежностей між компонентами.

Для цього застосовано модель MonoEmbed-GNN, яка поєднує тришарову графову згорткову мережу (GCN $64 \to 32 \to 16$) із варіаційним автоенкодером (VAE) для зменшення розмірності простору ознак і формування стійких векторних представлень вузлів. Вхідними даними слугували графи викликів (CG) і графи потоку даних (DG), побудовані на основі абстрактного синтаксичного дерева (AST) кожного модуля.

Python-проєкт TensorFlow:

- Архітектурна база та обсяг даних: Для TensorFlow згенеровано 37 400 вузлів і 121 000 ребер, що відображає значну щільність внутрішніх залежностей у кодовій базі.
- Конфігурація моделі: MonoEmbed-GNN використовувала тришарову GCN ($64 \to 32 \to 16$) з інтегрованим варіаційним автоенкодером, який оптимізував латентний простір вузлів. Для кластеризації векторних представлень застосовано нечіткий алгоритм Fuzzy C-Means із параметрами $k = 24$, $m = 1{,}8$.
- Результати: Модель виявила архітектурно узгоджені групи компонентів, серед яких чітко виділилися підсистеми tensorflow/core, tensorflow/python та інші базові модулі фреймворку. Досягнуто F1-міри $0{,}75 \pm 0{,}02$, що на 6 % вище від попередніх результатів (0,71), і забезпечено стабільність кластеризації при різних початкових умовах. Середній час виконання становив 1,9 години, що підтверджує високу ефективність алгоритму для великих графових структур.

Java-проєкт Apache Hadoop:

- Архітектурна база та обсяг даних: Для Hadoop побудовано 33 200 вузлів і 112 480 ребер, що відображає масштабність міжмодульних взаємозв'язків у системі.
- Конфігурація моделі: Застосовано ту ж архітектуру MonoEmbed-GNN (GCN $64 \to 32 \to 16$ + VAE). Для кластеризації векторних представлень використано Fuzzy C-Means із параметрами $k = 20$, $m = 1{,}8$.
- Результати: Отримані кластери коректно відтворили архітектурну структуру системи, чітко виокремивши підсистеми hadoop/hdfs, hadoop/mapreduce та допоміжні модулі ядра. Модель досягла F1-міри $0{,}74 \pm 0{,}02$, що на ≈ 6 % вище за попередні показники (0,70), і забезпечила стабільну якість кластеризації при різних вибірках графових даних. Середній час виконання склав 1,7 години, що свідчить про ефективність обробки великих Java-кодових баз.

Узагальнення результатів: Модель MonoEmbed-GNN продемонструвала здатність формувати узгоджені векторні представлення вузлів і виявляти приховані залежності між модулями в обох типах проєктів. Для TensorFlow і Hadoop отримано порівнянні результати (F1 $\approx 0{,}75$ і 0,74 відповідно), що свідчить про стабільність кластеризації незалежно від мови програмування чи структури проєкту. Загалом підхід забезпечує автоматичну архітектурну сегментацію без ручного втручання чи додаткового налаштування гіперпараметрів. Отримані кластери підтверджують архітектурну цілісність і модульну декомпозицію великих систем, створюючи основу для подальшої автоматизації аналізу та еволюційного моніторингу програмного забезпечення.

Етап 3. Автоматичний рефакторинг і міграція коду: На третьому етапі експерименту досліджувалася ефективність автоматизованого рефакторингу та адаптації програмного коду із застосуванням трансформерних моделей CodeT5+ і CodeT5 DA, що реалізують архітектуру encoder–decoder та підхід sequence-to-sequence. Конвеєр складався з двох етапів: макроструктурний рефакторинг архітектури (CodeT5+) та мікрорівнева адаптація тестів (CodeT5 DA).

Python-проєкт TensorFlow:

- CodeT5+ (макроструктурний рефакторинг): Модель CodeT5+ із архітектурою encoder–decoder 12×12 (hidden size = 1024, 16 heads, ≈770 млн параметрів) навчалася на корпусі CodeSearchNet + TensorFlow subset. Параметри навчання: epochs = 6, LR = 3×10^{-5}, batch size = 4, час виконання ≈ 11 год GPU. Результати: скорочення циклічних залежностей на 14 % та збереження функціональності 98 %, що підтверджує здатність моделі ефективно виконувати структурну декомпозицію та зменшувати міжмодульні зв'язки без втрати працездатності.
- CodeT5 DA (мікрорівнева адаптація тестів): Модель CodeT5 DA (архітектура T5-base, 220 млн параметрів, hidden size = 768, 12 heads) навчалася на корпусі Methods2Test + Defects4J + TensorFlow tests. Параметри навчання: epochs = 5, LR = 2×10^{-5}, batch = 4, GPU-час ≈ 5 год. Результати: підвищення тестового покриття з 56 % до 66 %, line coverage +19 %, mutation score +15 %. Це демонструє підвищення надійності тестів і зменшення кількості потенційних дефектів після автоматичного редизайну коду.

Java-проєкт Apache Hadoop:

- CodeT5+ (макроструктурний рефакторинг): Модель CodeT5+ із тією ж архітектурою encoder–decoder 12×12 (hidden size = 1024, 16 heads, ≈770 млн параметрів) навчалася на корпусі CodeSearchNet + Hadoop subset. Параметри навчання: epochs = 6, LR = 3×10^{-5}, batch = 4, GPU-час ≈ 12 год. Результати: скорочення циклічних залежностей на 12 % при збереженні функціональності 98 %, що свідчить про стабільну узагальнюваність моделі для мов із суворою типізацією.
- CodeT5 DA (мікрорівнева адаптація API і тестів): Модель CodeT5 DA (архітектура T5-base, 220 млн параметрів, hidden size = 768, 12 heads) навчалася на корпусі Methods2Test + Defects4J + Hadoop tests. Параметри навчання: epochs = 5, LR = 2×10^{-5}, batch = 4, GPU-час ≈ 4,8 год. Результати: підвищення тестового покриття з 55 % до 64 %, line coverage +18,6 %, mutation score +16,4 %, що підтверджує ефективність моделі для адаптації тестів у великих Java-кодових базах.

Узагальнення результатів: Комбінація CodeT5+ і CodeT5 DA забезпечила повноцінний конвеєр автоматичного рефакторингу та міграції коду. CodeT5+ виконує макроструктурну оптимізацію, зменшуючи циклічні залежності в середньому на ≈13 % при збереженні функціональності 98 %, тоді як CodeT5 DA реалізує мікрорівневу адаптацію, забезпечуючи приріст тестового покриття ≈ +10 %, line coverage ≈ +18–19 % та mutation score ≈ +15 %. У середньому після адаптації тестове покриття обох проєктів становило 65–66 %, що підтверджує ефективність запропонованого підходу для Python - та Java-екосистем.

Етап 4. Генерація тестів і валідація: На фінальному етапі проведено інтеграцію моделей CodeT5 DA та DeepREST-RL з метою автоматичного створення та оновлення тестів у системах TensorFlow і Apache Hadoop. Така комбінація забезпечила завершений цикл перевірки функціональності, охоплюючи юніт та інтеграційний рівні тестування, поєднуючи генеративні можливості трансформерів із оптимізаційними механізмами навчання з підкріпленням.

Python-проєкт TensorFlow:

- CodeT5 DA (генерація юніт тестів): Модель CodeT5 DA (T5-base, hidden size = 768, 12 heads) використовувалася для генерації pytest-тестів на основі існуючих юніт-наборів після попереднього рефакторингу. Параметри навчання: epochs = 5, batch = 8, LR = 2×10^{-5}, середній GPU-час $\approx$ 3 год. Було досягнуто compile rate 95 %, що вказує на високу коректність синтаксису згенерованих тестів. Результати показали line coverage +18 % і mutation score +15 %, що демонструє ефективне охоплення логіки коду та зменшення кількості прихованих дефектів.
- DeepREST-RL (генерація інтеграційних тестів): Модель DeepREST-RL, побудована на архітектурі Deep Q-Network (256–128–64) з параметрами γ = 0.99, α = $1e^{-4}$, використовувалася для генерації та оптимізації тестів на основі існуючих інтеграційних сценаріїв. Вона навчалася на TensorFlow Serving REST API ($\sim$180 end-points) протягом 500 000 епізодів. Було отримано branch coverage +12 % і mutation score +10 %. Підсумкове загальне покриття коду становило $\approx$78 %.

Java-проєкт Apache Hadoop:

- CodeT5 DA (генерація юніт тестів): Модель CodeT5 DA (T5-base, hidden size = 768, 12 heads) використовувалася для генерації JUnit-тестів на основі існуючих юніт-наборів після попереднього рефакторингу. Параметри навчання: epochs = 5, batch = 8, GPU-час $\approx$ 3 год. Було досягнуто compile rate 94 %, що вказує на високу стабільність і синтаксичну узгодженість тестів. Результати показали line coverage +18 % і mutation score +16 %, що підтверджує узгодженість і сталість тестових сценаріїв після структурних змін.
- DeepREST-RL (генерація інтеграційних тестів): Модель DeepREST-RL, побудована на архітектурі Deep Q-Network (256–128–64) з параметрами γ = 0.99, α = $1e^{-4}$, використовувалася генерації та оптимізації тестів на основі існуючих інтеграційних сценаріїв. Вона навчалася на Hadoop REST API ($\sim$150 end-points) протягом 500 000 епізодів. Було отримано branch coverage +13 % і mutation score +11 %. Підсумкове загальне покриття коду становило $\approx$77 %.

Узагальнення результатів: Інтеграція CodeT5 DA та DeepREST-RL сформувала автоматизований механізм розширення тестових наборів, який поєднує генеративну адаптацію тестів на рівні вихідного коду та пошук оптимальних послідовностей для інтеграційного рівня. Було отримано стійке покращення показників тестового покриття: Line coverage до +18 %, Branch coverage до +12-13 %, Mutation score до +15-16 %.

Після завершення всіх етапів загальне тестове покриття досягло $\approx$72 % для TensorFlow і $\approx$70 % для Hadoop. Це підтверджує ефективність інтегрованого

конвеєра моделювання, який об'єднує рефакторинг, кластеризацію та генерацію тестів у єдину автоматизовану систему підвищення якості коду.

Середовище виконання: Усі експерименти виконувалися в контейнеризованому середовищі Docker, що забезпечувало повну ізоляцію залежностей. Для побудови середовища використано офіційний образ pytorch/pytorch:1.9.0-cuda11.1-cudnn8 на операційній системі Ubuntu 22.04 LTS.

Кодова база складалася з двох основних стеків - Python 3.9 і Java JDK 17 та похідних бібліотек досліджуваних проектів та технологічних рішень запропонованих у статті. Для забезпечення статистичної узгодженості експериментів усі псевдовипадкові генератори були фіксовані на однакових seed-параметрах 42. Це дало змогу мінімізувати стохастичну варіативність результатів між повторними запусками.

Обчислення проводилися на кластері з чотирьох графічних процесорів NVIDIA A100 (40 GB VRAM кожен) та 256 GB оперативної пам'яті. Завдяки такій конфігурації стало можливим паралельне виконання кількох моделей у межах одного конвеєра, зокрема одночасне навчання GraphCodeBERT і MonoEmbed-GNN, паралельне тестування CodeT5 DA та DeepREST-RL. Для контролю версій використовувалися Git-репозиторії з повними Dockerfile-описами та YAML-конфігураціями, що містили гіперпараметри навчання, специфікацію датасетів і контрольні точки збереження моделей.

5. Тенденції розвитку та нерозв'язані питання в підходах до модернізації. Попри значні досягнення в адаптації ML-моделей до завдань модернізації застарілого програмного забезпечення, поточна практика свідчить про поступове формування обмеженого але стабільного ядра технік, що демонструють ефективність на різних етапах процесу модернізації. Ці методи охоплюють діагностику технічного боргу, структурну декомпозицію, автоматизовану трансформацію коду та верифікацію змін. Наявність кількісних результатів, формалізованих метрик та підтвердженої працездатності в експериментальних сценаріях дозволяє розглядати їх як основу для побудови типових ML-ланцюжків у задачах трансформації. Нижче подано узагальнення таких рішень, які вже можуть слугувати відправною точкою для інженерних команд у практиці автоматизованої модернізації.

5.1. Сформоване дослідницьке ядро.

- Налаштована модель GraphCodeBERT із доменною адаптацією LoRA виявляє «запахи коду» у середньому на 10–15 % точніше, що веде до зменшення технічного боргу на 11–17 % та підвищення індексу підтримуваності коду на 13 %.
- Графові нейронні мережі MonoEmbed-GNN, які аналізують граф викликів залежностей, підвищують модульність з 0,36 до 0,53, полегшуючи виокремлення сервісів і підвищуючи когерентність компонентів під час переходу до мікросервісної архітектури.
- Модель CodeT5+ зберігає зміст коду на рівні 88 % (функції $\leq$ 200 LOC) та скорочує середній час відновлення системи на 6–9 %, доводячи ефективність автоматизованого переписування коду зі збереженням бізнес-логіки.
- Модель CodeT5-DA автоматично генерує додаткові модульні тести на основі сигнатур методів і коментарів, що підвищує покриття коду (ΔCoverage $\approx$ +10-12 %) і якість тестів (Mutation Score $\approx$ +8-10 %) при високій стабіль-

ності компіляції (Compile Rate ≈ 95 %).

- Агент на основі DeepREST-RL, навчений із підкріпленням, підвищує тестове покриття на 15-18 % при інтегральному F1-measure $\approx 0{,}84$, зосереджуючи перевірку на найбільш ризикованих гілках виконання та скорочуючи потребу в ручному тест-дизайні.
- Інтегральна метрика ефективності модернізації SES у промислових пілотах досягає 0,15–0,45 (орієнтовно), зниження технічного боргу ≥ 10 % і скорочення часу відновлення системи ≥ 10 % формують базові орієнтири для оцінювання ML-підходів і управління якістю змін та розглядаються як реалістичні робочі пороги.

Незважаючи на обнадійливі результати, досягнуті завдяки застосуванню сучасних ML-підходів на різних етапах модернізації, низка важливих питань залишається відкритою. Ефективність уже апробованих моделей часто залежить від контексту їх застосування, масштабів системи, якості вихідних даних та рівня інтеграції у виробничі процеси. Подальше вдосконалення підходів потребує вирішення низки концептуальних, методичних і практичних викликів, які впливають як на точність і стабільність рішень, так і на їхню вартість, масштабованість і пояснюваність. У цьому розділі окреслено ключові напрями для подальших досліджень, необхідних для досягнення більшої зрілості ML-технологій у сфері програмної модернізації.

5.2. Відкриті питання та перспективи для подальших досліджень.

- Необхідно кількісно перевірити як застосування методів пояснюваності (LIME, SHAP) впливає на швидкість і якість ревю коду та рівень довіри до ML-рішень.
- При показнику збереження семантики частина функціональної еквівалентності все ще губиться. У критичних модулях потрібні формальні методи та специфікації на додачу до тестів.
- Точність мовних моделей знижується на нетипових легасі-фрагментах коду, потрібні валідовані стратегії донавчання на малих доменних корпусах та перевірка узагальнюваності.
- Відсутність чітких показників готовності до промислового впровадження ускладнює прийняття рішень щодо розгортання нових компонентів у реальному середовищі.
- Своєчасне виявлення збоїв потребує сценаріїв проактивної ін'єкції помилок та безперервного моніторингу цілісності моделей у режимі реальної експлуатації.
- На великих базах коду ефективність чинних методів знижується. Потрібні інкрементальні та потокові моделі, які здатні зберігати точність під час обробки мільйонів рядків.
- Зростання інфраструктурних витрат висуває вимогу дослідити створення спрощених моделей без значної втрати точності і порівняти точність та ресурси у реальних умовах, щоб обґрунтувати вартість ML-рішень (TCO).

Таким чином, наведені результати окреслюють стійке дослідницьке ядро ML-практик для модернізації та водночас акцентують на критичних прогалинах, вирішення яких стане ключовим фактором наступного етапу еволюції підходів.

6. Висновки. У роботі здійснено комплексне узагальнення та критичний аналіз підходів до використання методів машинного навчання в процесах модернізації компонентів складних інформаційних систем. Отримані результати підтверджують, що поєднання алгоритмів аналізу, трансформації та перевірки коду на основі машинного навчання забезпечує підвищення рівня автоматизації та об'єктивності рішень під час технічного оновлення програмного забезпечення.

Наукові результати:

- Удосконалено підхід до оцінювання ефективності використання технологій машинного навчання у завданнях модернізації програмних систем шляхом упровадження кількісних показників, що характеризують зменшення технічного боргу (ΔTD), зміну модульності (ΔModularity) та скорочення середнього часу відновлення працездатності (ΔMTTR) після модифікацій.
- Експериментально підтверджено доцільність інтеграції моделей попереднього навчання, графових нейронних мереж і підкріпленого навчання для реалізації повного циклу модернізації - від виявлення проблемних фрагментів до верифікації результатів перетворень.
- За результатами випробувань на промислових репозиторіях встановлено, що використання розробленого підходу сприяє зменшенню технічного боргу в середньому на 11–17 %, підвищенню модульності до +0,15 п.п та збільшенню тестового покриття на 15–18 п.п.

Практичні результати:

- Реалізований підхід дає змогу зменшити витрати на технічну підтримку, скоротити час модернізації програмних систем і підвищити надійність змін без необхідності їхнього повного перепроектування.
- Запропоновані показники оцінювання ефективності можуть використовуватись як інструмент контролю якості в проєктах з технічного оновлення та як основа для ухвалення управлінських рішень.
- Розроблена методика придатна для поступового переходу від монолітних до мікросервісних архітектур із контрольованими ризиками втрати працездатності та сумісності компонентів.
- Практичні результати впровадження свідчать про потенціал застосування машинного навчання як інженерного інструменту для підвищення технологічної зрілості підприємств у контексті цифрової трансформації.

Подальший розвиток тематики доцільно спрямувати на масштабування розроблених рішень для систем обсягом понад один мільйон рядків коду, удосконалення механізмів пояснюваності моделей, кількісне оцінювання економічної ефективності впровадження та розроблення критеріїв готовності до промислової експлуатації таких технологій. Це створить передумови для формування сталої методології використання машинного навчання у процесах модернізації програмного забезпечення та зниження технологічного боргу у великих інформаційних системах.

Список використаної літератури

1. Naik, P., Nelaballi, S., Pusuluri, V. S., & Kim, D.-K. (2024). Deep Learning-Based Code Refactoring: A Review of Current Knowledge. *Journal of Computer Information Systems,* 64(2), 314–328. https://doi.org/10.1080/08874417.2023.2203088

2. Abgaz, Y. M., McCarren, A., Elger, P., Solan, D., Lapuz, N., Bivol, M., Jackson, G., Yilmaz, M., Buckley, J., & Clarke, P. (2023). Decomposition of Monolith Applications into Microservices Architectures: A Systematic Review. *IEEE Transactions on Software Engineering,* 49(8), 4213–4242. https://doi.org/10.1109/TSE.2023.3287297
3. Qian, L., Li, J., He, X., Gu, R., Shao, J., & Lu, Y. (2023). Microservice Extraction Using Graph Deep Clustering Based on Dual View Fusion. *Information and Software Technology,* 158, 107171. https://doi.org/10.1016/j.infsof.2023.107171
4. Pandey, S. K., Chand, S., Horkoff, J., Staron, M., Ochodek, M., & Durisic, D. (2025). Design Pattern Recognition: A Study of Large Language Models. *Empirical Software Engineering,* 30, 69. https://doi.org/10.1007/s10664-025-10625-1
5. De Siano, G. D., Fasolino, A. R., Sperlí, G., & Vignali, A. (2025). Translating Code with Large Language Models and Human-in-the-Loop Feedback. *Information and Software Technology,* 186, 107785. https://doi.org/10.1016/j.infsof.2025.107785
6. Narváez, D., Battaglia, N., Fernández, A., & Rossi, G. (2025). Designing Microservices Using AI: A Systematic Literature Review. *Software,* 4(1), 6. https://doi.org/10.3390/software4010006
7. Moreschini, S., Pour, S., Lanese, I., Balouek, D., Bogner, J., Li, X., Pecorelli, F., Soldani, J., Truyen, E., & Taibi, D. (2025). AI Techniques in the Microservices Lifecycle: A Systematic Mapping Study. *Computing,* 107, 100. https://doi.org/10.1007/s00607-025-01432-z
8. Zhang, F., Zhang, Z., & Keung, J. W. (2024). Data Preparation for Deep Learning-Based Code Smell Detection: A Systematic Literature Review. *Journal of Systems and Software,* 216, 112131. https://doi.org/10.1016/j.jss.2024.112131
9. Garcia, P., & Sritriratanarak, W. (2023). We Need a Theoretical Framework for the Modernization of Industrial Legacy Systems. *Frontiers in Industrial Engineering,* 1, 1266651. https://doi.org/10.3389/fieng.2023.1266651
10. Baumgartner, N., Iyenghar, P., Schoemaker, T., & Pulvermüller, E. (2024). AI-Driven Refactoring: A Pipeline for Identifying and Correcting Data Clumps in Git Repositories. *Electronics,* 13(9), 1644. https://doi.org/10.3390/electronics13091644
11. Tune, N., & Perrin, J-G. (2024). *Architecture Modernization: Socio-technical Alignment of Software, Strategy, and Structure.* Manning Publications.
12. Ford, N., Parsons, R., & Kua, P. (2023). *Building Evolutionary Architectures: Automated Software Governance.* 2nd ed. O'Reilly Media.
13. Romero, J., R., Medina-Bulo, I., & Chicano, F. (2023). *Optimising the Software Development Process with Artificial Intelligence.* Springer: Natural Computing Series.

Kohuch O. H., Matviychuk Y. M. Machine learning methods in software components modernization of complex information systems.

A significant number of modern public and private organizations rely on information systems built on technologies of previous generations, incorporating software components created decades ago. These systems remain critical for business operations. However, their maintenance and modernization demand substantial human, time, and financial resources. Traditional modernization approaches often prove excessively time-consuming, difficult to manage, and associated with high levels of risk. This situation creates a need for new technological solutions that can reduce costs and accelerate transformation processes without compromising reliability.

Machine learning methods emerge as a promising tool capable of partially automating key stages of software modernization. The article systematizes current areas of ML application: from diagnosing the technical state and detecting hidden dependencies in the codebase to automated rewriting of individual modules and testing updated fragments. Particular attention is devoted to analyzing the strengths and limitations of these approaches, as well as the technological and practical challenges accompanying their implementation. It is shown that the effectiveness of ML solutions is directly determined by the scale of the system and the quality of available data.

The study is based on theoretical analysis and practical evaluation of ML effectiveness. Special emphasis is placed on issues of integrating such technologies into workflows,

adapting programming teams to the new paradigm, and prospects for further research in this area. Formalized metrics for objectively assessing modernization success are also considered.

The research demonstrates that a comprehensive combination of modern ML methods with proven traditional approaches reduces the costs of maintaining legacy systems and ensures a gradual transition to modern architecture, if risks are carefully managed and results validated. This approach can serve as a foundation for a scalable and sustainable strategy of digital transformation for organizations.

Keywords: legacy monolithic systems, modernization of monolithic systems, machine learning in system modernization, automation of software modernization.

Одержано 28.08.2025