**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**

**НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ОДЕСЬКА ПОЛІТЕХНІКА»**

Навчально-науковий інститут комп'ютерних систем

Кафедра інженерії програмного забезпечення

Костянтин СЛАБЕНКО

(група АС-202)

**КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА**

**Програмна система прогнозування продуктивності розробників на основі результатів TypeScript коду**

*Спеціальність:*

121 Інженерія програмного забезпечення

*Освітньо-професійна програма:*

Інженерія програмного забезпечення

*Керівник:*

Віра ЛЮБЧЕНКО, д.т.н., професор

Одеса – 2025

# АНОТАЦІЯ

Слабенко К. О. Програмна система прогнозування продуктивності розробників на основі результатів TypeScript коду : кваліфікаційна робота магістра за спеціальністю «121 Інженерія програмного забезпечення» / Костянтин Олегович Слабенко ; керівник Віра Вікторівна Любченко. – Одеса : Нац. ун-т «Одес. політехніка», 2025. – 85 с.

Кваліфікаційна робота містить основну текстову частину на 60 сторінках, список використаних джерел з 35 найменувань на 3 сторінках, додатки на 22 сторінках.

Визначено сутність оцінки якості програмного коду на основі результатів та її вплив на продуктивність розробників. Обґрунтовано переваги підходу на основі результатів над традиційним підходом на основі активності. Розроблено систему з 20 метрик у трьох категоріях: досвід розробників, технічна продуктивність, бізнес-вплив. Проведено емпіричне дослідження 50 TypeScript проєктів з використанням кореляційного, регресійного та кластерного аналізу. Побудовано моделі машинного навчання для прогнозування продуктивності розробників. Розроблено практичні рекомендації з розрахунком повернення інвестицій.

*Ключові слова*: метрики на основі результатів, typescript, якість коду, машинне навчання, github api, досвід розробників, технічна продуктивність, бізнес-вплив, shap-значення, регресійний аналіз, кластерний аналіз, інженерія програмного забезпечення.

# ABSTRACT

Slabenko K. O. Software system for developer productivity prediction based on TypeScript code results : master's qualification work in the specialty «121 Software Engineering» / Kostiantyn Olehovych Slabenko ; supervisor Vira Viktorivna Liubchenko. – Odesa : Odesa Polytech. Nat. Univ., 2025. – 85 p.

The qualification work contains the main text part on 60 pages, a list of used sources with 35 titles on 3 pages, appendices on 22 pages.

The essence of outcome-based software code quality assessment and its impact on developer productivity has been defined. The advantages of the outcome-based approach over the traditional activity-based approach have been substantiated. A system of 20 metrics in three categories has been developed: Developer Experience, Technical Performance, Business Impact. An empirical study of 50 TypeScript projects was conducted using correlation, regression, and cluster analysis. Machine learning models for predicting developer productivity were built. Practical recommendations with return on investment calculations were developed.

*Keywords*: outcome-based metrics, typescript, code quality, machine learning, github api, developer experience, technical performance, business impact, shap values, regression analysis, cluster analysis, software engineering.

**ЗМІСТ**

[Перелік умовних позначень 6](#_heading=)

[Вступ 9](#_heading=)

[1 Огляд літератури та аналіз існуючих рішень 13](#_heading=)

[1.1 Підходи до оцінки якості програмного забезпечення 13](#_heading=)

[1.2 Метрики на основі активності та метрики на основі результатів 16](#_heading=)

[1.3 Огляд існуючих концептуальних моделей та досліджень 18](#_heading=)

[1.4 Аналіз існуючих систем оцінки якості коду 22](#_heading=)

[Висновки до розділу 1 24](#_heading=)

[2 Методологія дослідження 26](#_heading=)

[2.1 Загальна характеристика дослідження 26](#_heading=)

[2.2 Обґрунтування підходу на основі результатів 26](#_heading=)

[2.3 Система метрик на основі результатів 27](#_heading=)

[2.4 Комплексна оцінка якості 38](#_heading=)

[2.5 Критерії вибору проєктів 44](#_heading=)

[2.6 Методи збору даних 45](#_heading=)

[2.7 Методи статистичного аналізу 46](#_heading=)

[2.8 Методи машинного навчання 50](#_heading=)

[Висновки до розділу 2 56](#_heading=)

[3 Практична реалізація системи 58](#_heading=)

[3.1 Архітектура системи 58](#_heading=)

[3.2 Пакет @thesis/metrics 60](#_heading=)

[3.3 Пакет @thesis/metrics-collector 63](#_heading=)

[3.4 Пакет @thesis/scripts 65](#_heading=)

[3.5 Конвеєр обробки даних 68](#_heading=)

[3.6 Результати збору даних 69](#_heading=)

[3.7 Імплементація статистичного аналізу даних 71](#_heading=)

[3.8 Імплементація машинного навчання 72](#_heading=)

[Висновки до розділу 3 74](#_heading=)

[4 Дослідження та аналіз результатів 76](#_heading=)

[4.1 Валідація та дослідження даних 76](#_heading=)

[4.2 Статистичний аналіз 82](#_heading=)

[4.3 Часовий аналіз 92](#_heading=)

[4.4 ML-моделювання та прогнозний аналіз 97](#_heading=)

[4.5 Ключові виявлені закономірності та практичні рекомендації 105](#_heading=)

[4.6 Обмеження дослідження 110](#_heading=)

[Висновки до розділу 4 112](#_heading=)

[Загальні висновки 114](#_heading=)

[Список використаних джерел 119](#_heading=)

[Додаток А Інтерфейси та конфігурація на основі результатів метрик 124](#_heading=)

[Додаток Б Лістинги програмного коду 128](#_heading=)

[Додаток В Python скрипти аналізу даних 144](#_heading=)

[Додаток Г Список проєктів дослідження 153](#_heading=)

# ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ACF – Autocorrelation Function (автокореляційна функція)

ACM – Association for Computing Machinery (Асоціація обчислювальної техніки)

ADF – Augmented Dickey-Fuller (розширений тест Дікі-Фуллера)

API – Application Programming Interface (інтерфейс програмування застосунків)

AR – Autoregressive (авторегресійний)

ARIMA – AutoRegressive Integrated Moving Average (авторегресійна інтегрована модель ковзного середнього)

BI – Business Impact (бізнес-вплив)

CD – Continuous Delivery / Continuous Deployment (безперервна доставка / розгортання)

CI – Continuous Integration (безперервна інтеграція)

CLI – Command Line Interface (інтерфейс командного рядка)

CSV – Comma-Separated Values (значення, розділені комами)

CV – Cross-Validation (перехресна валідація)

DORA – DevOps Research and Assessment (дослідження та оцінка DevOps)

DX – Developer Experience (досвід розробника)

E2E – End-to-End (наскрізний)

FDR – False Discovery Rate (частка хибних відкриттів)

IDE – Integrated Development Environment (інтегроване середовище розробки)

IEC – International Electrotechnical Commission (Міжнародна електротехнічна комісія)

IEEE – Institute of Electrical and Electronics Engineers (Інститут інженерів електротехніки та електроніки)

IQR – Interquartile Range (міжквартильний розмах)

ISO – International Organization for Standardization (Міжнародна організація зі стандартизації)

JSON – JavaScript Object Notation (нотація об'єктів JavaScript)

KB – Kilobyte (кілобайт)

L1 – L1 Regularization / Lasso (L1 регуляризація / Лассо)

L2 – L2 Regularization / Ridge (L2 регуляризація / Рідж)

LOC – Lines of Code (рядки коду)

MA – Moving Average (ковзне середнє)

MAE – Mean Absolute Error (середня абсолютна помилка)

MAPE – Mean Absolute Percentage Error (середня абсолютна відсоткова помилка)

ML – Machine Learning (машинне навчання)

MTTR – Mean Time To Recovery (середній час відновлення)

OLS – Ordinary Least Squares (звичайний метод найменших квадратів)

PACF – Partial Autocorrelation Function (часткова автокореляційна функція)

PC – Principal Component (головна компонента)

PCA – Principal Component Analysis (аналіз головних компонент)

PR – Pull Request (запит на злиття)

REST – Representational State Transfer (передача репрезентативного стану)

RMSE – Root Mean Square Error (середньоквадратична помилка)

ROI – Return on Investment (повернення інвестицій)

SHAP – SHapley Additive exPlanations (адитивні пояснення Шеплі)

SLA – Service Level Agreement (угода про рівень обслуговування)

SPACE – Satisfaction, Performance, Activity, Communication, Efficiency (задоволення, продуктивність, активність, комунікація, ефективність)

TDD – Test-Driven Development (розробка через тестування)

TP – Technical Performance (технічна продуктивність)

UI – User Interface (інтерфейс користувача)

URL – Uniform Resource Locator (уніфікований локатор ресурсу)

VIF – Variance Inflation Factor (фактор інфляції дисперсії)

XGBoost – eXtreme Gradient Boosting (екстремальне градієнтне підсилення)

# ВСТУП

**Актуальність теми.** У сучасній індустрії розробки програмного забезпечення якість коду є критичним фактором успіху проєкту. За даними GitHub Octoverse 2024 [1], TypeScript увійшов до топ-5 найпопулярніших мов програмування з понад 20 мільйонами розробників по всьому світу. Водночас індустрія стикається з викликами у вимірюванні та покращенні продуктивності розробників.

Традиційні підходи до оцінки якості коду фокусуються на статичних метриках (кількість рядків коду, циклічна складність, покриття тестами) та метриках активності (кількість комітів, годин роботи). Ці метрики легко піддаються маніпулюванню та не відображають реальної цінності створеного програмного забезпечення [2].

Сучасні концептуальні моделі SPACE [3], DORA [4] та DevEx [5] пропонують альтернативний підхід, орієнтований на результати (на основі результатів), проте не надають конкретної системи метрик для TypeScript проєктів. Існуючі інструменти оцінки якості (SonarQube [6], Code Climate [7], Codecov [8]) фокусуються на статичному аналізі, ігноруючи важливі показники результативності: час виходу на ринок, зростання спільноти, швидкість доставки функціональності.

Отже, існує наукова та практична прогалина у розробці на основі результатів системи оцінки якості TypeScript коду, яка б вимірювала реальні результати у трьох вимірах (досвід розробників, технічна продуктивність, бізнес-вплив), надавала кількісні докази кореляцій між метриками та результатами, дозволяла прогнозувати продуктивність розробників та пропонувала практичні рекомендації для інженерних команд.

Дана робота спрямована на вирішення цієї прогалини через створення комплексної системи на основі результатів аналізу TypeScript коду на основі даних 50 популярних відкритих проєктів.

**Мета і задачі роботи.** Мета роботи — покращення процесу оцінки продуктивності розробників за рахунок розробки програмної системи, яка прогнозує продуктивність на основі результатів аналізу TypeScript коду з використанням машинного навчання.

Для досягнення поставленої мети необхідно вирішити наступні задачі:

1. проаналізувати існуючі підходи до оцінки якості програмного забезпечення та продуктивності розробників, обґрунтувати переваги на основі результатів підходу над на основі активності;
2. розробити систему на основі результатів метрик для TypeScript проєктів, що охоплює три категорії: Developer Experience, Technical Performance та Business Impact;
3. створити автоматизовану систему збору метрик на основі GitHub API та локального аналізу коду;
4. зібрати та проаналізувати дані 50 популярних TypeScript проєктів з відкритим кодом, включаючи перерізні та часові дані (300 часових зрізів за 6 місяців);
5. виявити статистично значущі кореляції між статичними метриками коду та результатами з використанням кореляційного, регресійного та кластерного аналізу;
6. побудувати та оцінити моделі машинного навчання (лінійна регресія, Ridge, Lasso, ElasticNet, Random Forest [9], XGBoost [10], LightGBM [11]) для прогнозування продуктивності розробників;
7. застосувати SHAP values [12] для інтерпретованості моделей та виявлення практичних спостережень;
8. розробити практичні рекомендації для команд розробки програмного забезпечення з розрахунком ROI.

**Об'єкт роботи.** Об'єктом роботи є процес оцінки якості TypeScript коду та продуктивності розробників у проєктах з відкритим кодом веброзробки.

**Предмет роботи.** Предметом роботи є програмне забезпечення, яке розраховує прогноз продуктивності розробників на основі метрик TypeScript коду (тривалість огляду коду, покриття тестами, розмір збірки, час відладки, швидкість вирішення задач) та результатів (досвід розробників, технічна продуктивність, бізнес-вплив).

**Методи дослідження.** У роботі використано комплекс методів наукового дослідження:

1. емпіричні методи:
   1. кількісний емпіричний аналіз для збору даних з 50 TypeScript проєктів;
   2. збір даних через GitHub REST API v3;
   3. інтеграція зовнішніх API (Codecov API, bundlephobia API, npm Registry API);
   4. локальний аналіз коду з використанням TypeScript Compiler API;
2. статистичні методи:
   1. кореляційний аналіз (Pearson correlation) з FDR correction;
   2. регресійний аналіз (OLS) для виявлення предикторів;
   3. кластерний аналіз (K-means, hierarchical clustering);
   4. аналіз головних компонент (PCA);
   5. перевірка гіпотез з довірчими інтервалами;
3. методи часового аналізу:
   1. ARIMA для прогнозування;
   2. тести стаціонарності (Augmented Dickey-Fuller, KPSS);
   3. сезонна декомпозиція для виявлення трендів;
   4. виявлення точок структурних змін;
4. методи машинного навчання:
   1. сім алгоритмів регресії: Linear Regression, Ridge, Lasso, ElasticNet, Random Forest, XGBoost, LightGBM;
   2. 5-fold крос-валідація та TimeSeriesSplit для часових даних;
   3. SHAP values для інтерпретованості моделей;
5. методи валідації:
   1. розподіл вибірки на навчальну, валідаційну та тестову (70/15/15);
   2. діагностика мультиколінеарності (VIF);
   3. аналіз залишків для перевірки припущень регресії.

**Наукова новизна одержаних результатів.** Наукова новизна роботи полягає в наступному:

1. удосконалено метод оцінки якості програмного коду шляхом інтеграції трьох категорій метрик на основі результатів (досвід розробників, технічна продуктивність, бізнес-вплив), що на відміну від існуючих підходів дозволяє враховувати взаємодію між категоріями (DX×TP importance = 47.5%);
2. розроблено модель прогнозування продуктивності розробників на основі регресійного аналізу та машинного навчання, яка досягає коефіцієнта детермінації R² = 0.66–0.93 для ключових показників (час доставки, зростання спільноти);
3. запропоновано методологію збору та аналізу метрик на основі результатів для TypeScript проєктів, що включає автоматизований збір даних через GitHub API, часовий аналіз (ARIMA, сезонна декомпозиція) та інтерпретацію моделей (SHAP values).

**Практичне значення одержаних результатів.** Практична цінність роботи полягає в наступному:

1. розроблено науково обґрунтовані рекомендації для команд розробників з кількісною оцінкою впливу та розрахунком повернення інвестицій;
2. створено інструменти для автоматизованого збору метрик з підтримкою інкрементальних оновлень та виводом у форматах JSON, CSV, Markdown;
3. підготовлено документацію для різних цільових аудиторій: керівників інженерних підрозділів, технічних лідерів, дослідників та розробників інструментів;
4. сформовано набір даних та пакет відтворення для подальших досліджень у галузі оцінки якості програмного забезпечення.

Результати роботи впроваджено у вигляді Git-репозиторію з відкритим доступом [41].

# 1 ОГЛЯД ЛІТЕРАТУРИ ТА АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ

## 1.1 Підходи до оцінки якості програмного забезпечення

Оцінка якості програмного забезпечення є фундаментальною проблемою в інженерії програмного забезпечення з моменту становлення цієї дисципліни як окремої галузі знань. Протягом десятиліть розробники, дослідники та менеджери шукали способи кількісно виміряти якість коду та продуктивність команд розробки.

### 1.1.1 Історичний розвиток підходів до оцінки якості.

Перші спроби формалізації оцінки якості програмного забезпечення датуються 1970-ми роками, коли McCabe (1976) запропонував метрику циклічної складності (Cyclomatic Complexity) [13] для вимірювання складності управляючих потоків у програмах. Ця метрика базувалася на теорії графів і обчислювалася як кількість лінійно незалежних шляхів через код. McCabe стверджував, що модулі з циклічною складністю понад 10 важко тестувати і підтримувати. Пізніше, у 1990-х роках, Chidamber та Kemerer [14] розширили підхід на об'єктно-орієнтовані системи, запропонувавши метрики CBO (Coupling Between Objects), WMC (Weighted Methods per Class) та інші.

У 1977 році Halstead представив набір метрик складності програм, заснованих на підрахунку операторів і операндів у коді. Метрики Halstead включали програмну складність, програмний об'єм та зусилля на розробку. Хоча ці метрики мали теоретичну базу, їх практична цінність залишалася дискусійною.

Наприкінці 1980-х – початку 1990-х років з'явилися більш комплексні моделі якості програмного забезпечення. Міжнародна організація зі стандартизації (ISO) розробила серію стандартів ISO/IEC 9126 (1991), пізніше замінену на ISO/IEC 25010:2011 [15], яка визначає модель якості програмного забезпечення по восьми характеристикам: функціональна придатність, ефективність виконання, сумісність, зручність використання, надійність, безпека, супроводжуваність і переносимість. Abran et al. [16] провели детальний аналіз інтерпретації зручності використання (usability) в ISO стандартах, виявивши неоднозначності у визначеннях.

McConnell у своїй класичній праці "Code Complete" (2004) [17] систематизував кращі практики написання якісного коду, підкреслюючи важливість читабельності, модульності та простоти. Martin у книзі "Clean Code" (2008) [18] запропонував набір принципів і патернів для написання чистого, зрозумілого коду, включаючи SOLID принципи об'єктно-орієнтованого проектування.

### 1.1.2 Традиційні метрики якості коду.

Традиційні підходи до оцінки якості коду зосереджувалися на статичних метриках, які можна було автоматично обчислити з вихідного коду [19]. Basili та Rombach [20] у рамках проєкту TAME (Tailoring A Measurement Environment) показали важливість контекстно-специфічних метрик та їх узгодження з цілями організації. Основні категорії таких метрик включають:

Lines of Code (LoC) – найпростіша метрика, що підраховує кількість рядків коду (фізичні PLOC, логічні LLOC, без коментарів SLOC), проте більше коду не означає кращу функціональність.

Cyclomatic Complexity (CC) – вимірює кількість лінійно незалежних шляхів через програмний код. Обчислюється за формулою:

|  |  |  |
| --- | --- | --- |
|  |  | (1.1) |

де E – кількість ребер графу потоку керування;

N – кількість вершин;

P – кількість зв'язаних компонентів (зазвичай 1 для одного методу).

Рекомендовані пороги:

1. CC ≤ 10 – простий код, низький ризик;
2. 11 ≤ CC ≤ 20 – помірно складний;
3. 21 ≤ CC ≤ 50 – складний, високий ризик;
4. CC > 50 – дуже складний, непідтримуваний.

Maintainability Index (MI) – комбінує Halstead Volume, Cyclomatic Complexity та Lines of Code:

|  |  |  |
| --- | --- | --- |
|  |  | (1.2) |

де HV – Halstead Volume;

CC – Cyclomatic Complexity;

LoC – Lines of Code.

MI нормалізується до шкали 0-100, де вищі значення вказують на кращу підтримуваність.

Покриття коду – відсоток коду, покритий автоматизованими тестами (типи: рядків, гілок, функцій, інструкцій). Галузеві орієнтири: >80% для критичних компонентів, >60% для загальної бази, >40% для успадкованого коду. Дублювання коду – відсоток дублюючого коду (>5-10% вказує на погану архітектуру).

### 1.1.3 Запахи коду та статичний аналіз.

Fowler та Beck [40] ввели концепцію "запахів коду" (code smells). Основні категорії: роздуті елементи (довгі методи >20-30 рядків, великі класи >10 полів), порушники ООП (switch замість поліморфізму), перешкоди змінам (розбіжні зміни, хірургія дробовиком), зайві елементи (дублюючий код, мертвий код), зв'язувачі (надмірне зв'язування класів). Сучасні інструменти (SonarQube, ESLint) автоматично виявляють запахи коду через аналіз AST. Khomh et al. [21] показали, що антипатерни збільшують схильність до дефектів.

### 1.1.4 Метрики продуктивності та розміру збірки.

З розвитком веб-технологій важливими стали метрики продуктивності. Розмір збірки (Bundle Size): малі бібліотеки <100KB, середні застосунки 100KB-1MB, великі 1MB-5MB, корпоративні >5MB (Google рекомендує <170KB стиснутий). Web Vitals: FCP <1.8с, LCP <2.5с, TTI <3.8с, TBT <200мс, CLS <0.1. Час збірки: малі проєкти <10с, середні 10-60с, великі 1-10хв, монорепозиторії >10хв. Оптимізація через інкрементальну компіляцію, кешування (Nx, Turborepo), паралелізацію.

### 1.1.5 Обмеження традиційних підходів.

Незважаючи на широке використання, традиційні метрики якості коду мають суттєві обмеження:

1. відсутність кореляції з бізнес-цінністю – слабка кореляція між LoC, CC та реальною цінністю [23];
2. маніпулювання метриками – штучне зменшення складності через розбиття функцій, тести без перевірок;
3. фокус на локальну якість – не враховують архітектуру системи, взаємодію компонентів. Zimmermann та Nagappan [24] довели ефективність аналізу графів залежностей;
4. ігнорування досвіду розробників – не вимірюють легкість роботи з кодом. Meyer et al. [2] виявили вплив діяльності розробників на результати;
5. статичність – не враховують динаміку розробки, еволюцію проєкту;
6. складність інтерпретації – відсутність галузевих орієнтирів.

Ці обмеження призвели до пошуку альтернативних підходів, зокрема метрик на основі результатів.

## 1.2 Метрики на основі активності та метрики на основі результатів

### 1.2.1 Підхід на основі активності.

Метрики на основі активності фокусуються на вимірюванні того, що розробники роблять, а не результатів. Типові метрики: кількість рядків коду, комітів, зареєстрованих годин, закритих тікетів, створених PR, виконаних оглядів коду, відвіданих нарад. Murphy-Hill et al. [27] показали значну варіацію складності виправлень.

### 1.2.2 Проблеми підходу на основі активності.

Основні проблеми: маніпулювання показниками (Rahman та Devanbu [28] показали, що процесні метрики є кращими предикторами дефектів; закон Гудхарта: "Коли міра стає ціллю, вона перестає бути хорошою мірою"), відсутність кореляції з бізнес-цінністю (Meyer et al. показали, що жодна метрика активності не корелювала зі сприйнятою продуктивністю), заохочення неправильної поведінки (надмірне проєктування, вибір легких завдань), ігнорування більшої картини (менторство, архітектурні рішення, рефакторинг), демотивація розробників (68% вважають такі метрики шкідливими [39]).

### 1.2.3 Підхід на основі результатів.

Метрики на основі результатів (на основі результатів) фокусуються на вимірюванні результатів, а не активності. Цей підхід набув популярності з розвитком Agile, DevOps та сучасних практик інженерії програмного забезпечення.

Філософія підходу на основі результатів:

Замість питання "Скільки коду ти написав?" підхід на основі результатів запитує:

1. як швидко ми доставляємо цінність користувачам?;
2. наскільки якісний наш продукт?;
3. наскільки задоволені наші розробники?;
4. який бізнес-вплив нашої роботи?.

Типові метрики на основі результатів включають чотири ключові метрики DORA [29]–[30] (час виходу на ринок, частота розгортання, частота збоїв при змінах, середній час відновлення), задоволеність клієнтів (NPS, CSAT), задоволеність розробників [31], рівень прийняття функціональності, рівень пропущених помилок, тривалість огляду коду [32]–[33], зростання спільноти для відкритих проєктів [34]–[35]. Детальний опис метрик та формули розрахунку наведено у розділі 2.

### 1.2.4 Переваги підходу на основі результатів.

Переваги: узгодженість з бізнес-цілями, складніше маніпулювати, заохочення правильної поведінки (автоматизація, якість), комплексний погляд (командна співпраця, технічна якість, досвід розробників), адаптивність до контексту.

### 1.2.5 Виклики підходу на основі результатів.

Виклики: складність вимірювання (потребують інтеграції систем), затримка (результати через тижні), проблема атрибуції (складно приписати конкретним діям), контекстна чутливість (різні орієнтири для доменів). Порівняння підходів наведено у табл. 1.1.

Таблиця 1.1 – Порівняння підходів на основі активності та результатів

|  |  |  |
| --- | --- | --- |
| Критерій | На основі активності | На основі результатів |
| Фокус | Що робиться | Що досягається |
| Приклади | Рядки коду, коміти, години, тікети | Час виходу на ринок, задоволеність, якість |
| Легкість вимірювання | Висока (автоматична) | Середня (потребує інтеграції) |
| Ризик маніпулювання | Високий | Низький |
| Узгодженість з бізнесом | Слабка | Сильна |
| Затримка | Миттєво | Може бути відкладеною |
| Контекстна чутливість | Низька | Висока |
| Командна перспектива | Індивідуальна | Колективна |
| Мотивація розробників | Часто демотивує | Часто мотивує |
| Використання | Традиційні організації | Сучасні/Agile організації |

### 1.2.6 Гібридний підхід.

Сучасні організації комбінують обидва підходи: випереджаючі індикатори (участь в огляді коду, покриття тестами, стан CI/CD) та запізнілі індикатори (частота розгортання, частота збоїв, задоволеність клієнтів). Ключ: використовувати метрики активності як ранні сигнали, оцінювати успіх через результати.

## 1.3 Огляд існуючих концептуальних моделей та досліджень

Сучасні концептуальні моделі вимірювання продуктивності розробників представлені трьома провідними моделями: SPACE [3], DORA [4] та DevEx [5]. Кожна з них пропонує власний підхід до вирішення проблеми об'єктивної оцінки ефективності розробки програмного забезпечення.

### 1.3.1 Модель SPACE.

Модель SPACE [3] запропонувала п'ятивимірну структуру продуктивності: Satisfaction (задоволеність), Performance (результативність), Activity (активність), Communication (комунікація) та Efficiency (ефективність). Головна теза авторів полягає в тому, що жодна окрема метрика не може адекватно відобразити продуктивність розробника, тому необхідно комбінувати показники з різних вимірів.

Найбільш практичними з точки зору підходу на основі результатів є виміри Performance (вимірювання реальних результатів, а не обсягу роботи) та Efficiency (швидкість циклів зворотного зв'язку). Вимір Activity включено як допоміжний індикатор, проте автори застерігають від його використання як основного показника.

Критичний аналіз SPACE виявляє його суттєві обмеження. По-перше, модель є описовою, а не приписовою: вона визначає категорії вимірювання, але не надає конкретних метрик чи цільових значень. По-друге, реалізація всіх п'яти вимірів потребує значних інвестицій в інфраструктуру збору даних. По-третє, вимір Satisfaction базується переважно на опитуваннях, що вносить суб'єктивність та ускладнює порівняння між командами.

### 1.3.2 Метрики DORA.

Метрики DORA [4] є результатом багаторічного дослідження (2014-2018) за участю понад 32,000 професіоналів. Автори виявили чотири ключові метрики, що статистично корелюють з успіхом організації: частота розгортання (Deployment Frequency), час виконання змін (Lead Time for Changes), частота збоїв при змінах (Change Failure Rate) та середній час відновлення (Mean Time to Restore). Критерії продуктивності за метриками DORA наведено у табл. 1.2.

Таблиця 1.2 – Метрики DORA та критерії продуктивності

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Метрика | Елітні | Високі | Середні | Низькі |
| Частота розгортання | Кілька разів на день | Раз на день – тиждень | Раз на тиждень – місяць | Рідше місяця |
| Час виконання | <1 години | 1 день – 1 тиждень | 1 тиждень – 1 місяць | >1 місяця |
| Частота збоїв при змінах | 0-15% | 16-30% | 31-45% | >45% |
| Середній час відновлення | <1 години | <1 дня | <1 тижня | >1 тижня |

Важливим висновком DORA є спростування припущення про компроміс між швидкістю та якістю: елітні команди демонструють одночасно високу частоту розгортань І низький рівень збоїв.

Однак DORA має власні обмеження. Метрики фокусуються на швидкості доставки, але не вимірюють цінність доставленого функціоналу для користувачів. Крім того, орієнтири можуть бути нерелевантними для специфічних доменів (наприклад, банківського регулювання порівняно зі стартап-експериментуванням).

### 1.3.3 Модель DevEx.

Модель DevEx [5] зміщує фокус на суб'єктивний досвід розробника як головний фактор продуктивності. Модель визначає три виміри: когнітивне навантаження (складність розуміння коду), стан потоку (можливість безперервної концентрації) та цикли зворотного зв'язку (швидкість отримання відгуку).

Дослідження показують, що досвід розробника сильно корелює з продуктивністю (r=0.72, p<0.001), а покращення досвіду розробників можуть збільшити продуктивність на 15-25% [3]. Ader et al. (2019) виявили, що розробники витрачають 42% часу на непродуктивні активності (очікування збірки, налагодження середовища).

Обмеження DevEx полягає у значній залежності від опитувань та складності об'єктивного вимірювання когнітивного навантаження. Модель також не охоплює бізнес-результати, фокусуючись виключно на перспективі розробника.

### 1.3.4 Порівняльний аналіз та синтез.

Порівняння провідних моделей наведено у табл. 1.3.

Таблиця 1.3 – Порівняння провідних моделей

|  |  |  |  |
| --- | --- | --- | --- |
| Критерій | SPACE | DORA | DevEx |
| Основна мета | Багатовимірна оцінка | Продуктивність DevOps | Досвід розробника |
| Кількість вимірів | 5 | 4 метрики | 3 |
| Джерела даних | Опитування + системні | Системні журнали | Опитування + системні |
| Конкретність рекомендацій | Низька | Висока | Середня |
| Покриття бізнес-метрик | Часткове | Часткове | Відсутнє |

Критичний аналіз виявляє спільну прогалину всіх трьох моделей: жодна не надає комплексної системи метрик для конкретної технологічної екосистеми. SPACE є надто абстрактним для практичного застосування, DORA фокусується виключно на процесах доставки, а DevEx ігнорує бізнес-вплив.

### 1.3.5 Обґрунтування підходу даного дослідження.

Система метрик на основі результатів, розроблена в даній роботі, інтегрує сильні сторони всіх трьох моделей:

1. від SPACE: багатовимірний підхід (три категорії замість п'яти для практичності);
2. від DORA: фокус на вимірюванні результатів, а не активності;
3. від DevEx: увага до досвіду розробника як предиктора продуктивності.

На відміну від існуючих моделей, дана робота:

1. надає конкретні метрики з емпірично визначеними цільовими значеннями;
2. включає бізнес-метрики (зростання спільноти, час виходу на ринок);
3. базується на кількісному аналізі 50 реальних проєктів;
4. застосовує машинне навчання для виявлення прихованих залежностей.

## 1.4 Аналіз існуючих систем оцінки якості коду

Крім теоретичних моделей, існує численні інструменти та платформи для автоматизованої оцінки якості коду. Розглянемо найбільш популярні.

### 1.4.1 SonarQube.

Платформа для безперервної інспекції якості коду (29+ мов). Метрики: помилки, вразливості, запахи коду (81 тип для TypeScript), покриття тестами, дублювання, складність. Технічний борг за формулою (1.10):

|  |  |  |
| --- | --- | --- |
|  |  | (1.10) |

де Remediation Time for Issues – час виправлення проблем (дні/години). Ворота якості (покриття >80%, дублювання <3%, рейтинги A-E). Обмеження: фокус на статичних метриках, підхід на основі активності, без бізнес-впливу, високий шум, складність налаштування.

### 1.4.2 Code Climate.

Платформа автоматизованого огляду коду з оцінкою супроводжуваності (0-100) за формулою (1.11):

|  |  |  |
| --- | --- | --- |
|  |  | (1.11) |

Метрики: покриття тестами (інтеграція Istanbul, Codecov), частота змін (churn), дублювання (структурна подібність), когнітивна складність. Оцінки A-F, коментарі до PR. Обмеження: фокус на Ruby/JavaScript, пропрієтарний алгоритм, без метрик результатів, висока вартість.

### 1.4.3 Codecov.

Спеціалізована платформа покриття тестами. Метрики: покриття рядків, гілок, функцій; покриття різниці (новий код у PR), коментарі до PR, значки, графіки трендів. Безкоштовний для відкритих проєктів. Обмеження: одна метрика, не аналізує якість тестів, % покриття не корелює з частотою помилок.

### 1.4.4 GitHub Insights.

Вбудована аналітика активності репозиторію. Метрики: активність контрибуторів, частота комітів, зміни коду, граф залежностей; PR (кількість, час злиття, огляди); звернення (час відповіді, мітки); спільнота (зірки, форки, трафік). Вбудований, безкоштовний. Обмеження: чисті метрики активності без якості коду та бізнес-впливу, неможливість власних метрик.

Порівняльна таблиця систем наведена у таблиці 1.4.

Таблиця 1.4 – Порівняння систем оцінки якості коду

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Система | Фокус | Тип метрик | Сильні сторони | Слабкі сторони | Найкраще для |
| SonarQube | Статичний аналіз | Активність (помилки, запахи, покриття) | Комплексні правила, Відстеження історії | Без результатів, Високий шум | Корпоративні ворота якості |
| Code Climate | Супроводжуваність | Активність (складність, дублювання) | Просте налаштування, Інтеграція з PR | Суб'єктивна оцінка, Обмежені мови | Стартапи/середні компанії |
| Codecov | Покриття тестами | Активність (% покриття) | Простий у використанні, Хороша візуалізація | Одна метрика, Без якості | Команди з фокусом на тестування |
| GitHub Insights | Активність репозиторію | Активність (коміти, PR, звернення) | Вбудований, Безкоштовний | Чиста активність, Без якості | Відкриті проєкти |

### 1.4.5 Прогалини в існуючих системах.

Аналіз існуючих інструментів виявляє критичні прогалини: відсутність фокусу на результатах (всі системи вимірюють активність замість впливу), ігнорування бізнес-впливу (час виходу на ринок, зростання спільноти), відсутність метрик досвіду розробників (тривалість огляду, час налагодження), відсутність прогностичних можливостей (описова статистика без ML-прогнозування), відсутність емпіричної валідації (твердження про "хороші" значення не підкріплені даними), прогалини специфічні для TypeScript (частота помилок типів, де Gao et al. [36] довели 15% виявлення помилок, Petersen et al. [37] – вплив типізації на API).

Ці прогалини обґрунтовують необхідність нової системи оцінки на основі результатів, що вимірює бізнес-вплив та досвід розробників, надає ML-прогнози, базується на емпіричній валідації 50 TypeScript проєктів.

## Висновки до розділу 1

Проведений огляд літератури виявив еволюцію підходів від метрик активності до моделей на основі результатів. Традиційні метрики (рядки коду, коміти) показали слабку кореляцію з продуктивністю та легко маніпулюються. Метрики результатів (час виходу на ринок, частота розгортання) демонструють узгодженість з бізнес-цілями.

Концептуальні моделі (SPACE, DORA, DevEx) є описовими – визначають що вимірювати, але не надають конкретних метрик для специфічних технологій. Аналіз систем (SonarQube, Code Climate, Codecov) виявив фокус на статичному аналізі без вимірювання результатів.

Найбільша прогалина в екосистемі TypeScript: відсутні дослідження метрик результатів, емпіричні дані про кореляції, галузеві орієнтири, ML-системи прогнозування. Це обґрунтовує необхідність розробки системи прогнозування продуктивності на основі емпіричного аналізу 50 TypeScript проєктів з використанням машинного навчання.

У наступному розділі описано методологію дослідження: систему метрик результатів, критерії вибору проєктів, методи збору та аналізу даних.

# 2 МЕТОДОЛОГІЯ ДОСЛІДЖЕННЯ

## 2.1 Загальна характеристика дослідження

Дослідження базується на гіпотезі, що якість TypeScript коду можна оцінити через його вплив на вимірювані результати, і що ці результати можна передбачити на основі набору метрик, зібраних автоматично через GitHub API.

## 2.2 Обґрунтування підходу на основі результатів

Як показано в підрозділі 1.2, підхід на основі результатів має суттєві переваги над традиційним підходом на основі активності: узгодженість з бізнес-цілями, стійкість до маніпулювання, заохочення правильної поведінки. Детальне порівняння цих підходів наведено у таблиці 1.1.

У даному дослідженні підхід на основі результатів реалізовано через три категорії метрик:

1. досвід розробників (Developer Experience, DX) – досвід розробників при роботі з кодовою базою;
2. технічна продуктивність (Technical Performance, TP) – технічні характеристики системи;
3. бізнес-вплив (Business Impact, BI) – реальний вплив на бізнес.

Цей поділ базується на синтезі сучасних концептуальних моделей SPACE, DORA та DevEx, адаптованих для специфіки екосистеми TypeScript.

### 2.2.1 Обґрунтування вибору TypeScript.

TypeScript обраний об'єктом дослідження з кількох причин: популярність (5-те місце за GitHub Octoverse 2024 [1], 37% зростання за рік); додаткові метрики типізації (покриття типами, частота помилок TypeScript); велика екосистема відкритих проєктів; використання провідними компаніями (Google, Microsoft, Meta, Airbnb); веб-орієнтованість, що дозволяє вимірювати метрики продуктивності (розмір збірки, час завантаження).

### 2.2.2 Обґрунтування вибору відкритих проєктів.

Відкриті проєкти GitHub надають унікальні переваги: публічність даних через GitHub API; різноманітність за розміром, складністю та зрілістю; повна історія розробки для часового аналізу; природні бізнес-метрики (зірки, форки, кількість контрибуторів) як аналоги доходу для комерційних продуктів.

## 2.3 Система метрик на основі результатів

Розроблена система метрик включає 20 показників на основі результатів, розподілених на три категорії відповідно до моделей SPACE та DevEx:

1. досвід розробників (DX) – 7 метрик;
2. технічна продуктивність (TP) – 6 метрик;
3. бізнес-вплив (BI) – 7 метрик.

Кожна метрика визначається через:

1. назву та позначення;
2. формулу розрахунку;
3. одиниці вимірювання;
4. джерело даних (GitHub API endpoint або інше);
5. інтерпретацію (шкала якості: відмінно/добре/прийнятно/погано);
6. обґрунтування (чому метрика важлива).

### 2.3.1 Метрики досвіду розробників.

Метрики досвіду розробників (DX) вимірюють, наскільки комфортно та ефективно розробники можуть працювати з кодовою базою. Відповідають вимірам "Задоволеність" та "Потік" з моделі SPACE.

Тривалість огляду коду (Code Review Duration). Середній час від створення запиту на злиття до злиття (у годинах).

Розрахунок (див. ф. 2.1):

|  |  |  |
| --- | --- | --- |
|  |  | (2.1) |

де — середнє арифметичне значення;

— час злиття pull request;

— час створення pull request.

Інтерпретація:

1. відмінно: < 24 години;
2. добре: 24-48 годин;
3. прийнятно: 48-72 години;
4. погано: > 72 годин.

Обґрунтування: Швидкі огляди коду прискорюють доставку функціональності, покращують моральний дух розробників, зменшують переключення контексту.

Час налагодження (Debugging Time). Середній час розв'язання звернень щодо помилок (у годинах).

Розрахунок (див. ф. 2.2):

|  |  |  |
| --- | --- | --- |
|  |  | (2.2) |

де — GitHub issues з міткою "bug";

— час закриття issue;

— час створення issue.

Інтерпретація:

1. відмінно: < 48 годин;
2. добре: 48-120 годин;
3. прийнятно: 120-240 годин;
4. погано: > 240 годин.

Обґрунтування: Швидке вирішення багів індикує хорошу архітектуру, зрозумілість коду, якість тестування.

Час збірки (Build Time). Середній час виконання конвеєра CI/CD для успішних збірок (у секундах).

Розрахунок (див. ф. 2.3):

|  |  |  |
| --- | --- | --- |
|  |  | (2.3) |

де — тривалість виконання CI/CD pipeline у секундах;

— фільтр тільки успішних запусків.

Інтерпретація:

1. відмінно: < 300 с (5 хв);
2. добре: 300-600 с (5-10 хв);
3. прийнятно: 600-1200 с (10-20 хв);
4. погано: > 1200 с (20+ хв).

Обґрунтування: Швидкі збірки прискорюють цикли зворотного зв'язку, покращують досвід розробників, дозволяють частіше розгортати.

Коефіцієнт успішних розгортань (Successful Deployments Ratio). Відсоток успішних запусків CI/CD (0-1).

Розрахунок (див. ф. 2.4):

|  |  |  |
| --- | --- | --- |
|  |  | (2.4) |

де — CI/CD запуски з успішним результатом;

— загальна кількість CI/CD запусків.

Інтерпретація:

1. відмінно: > 95%;
2. добре: 90-95%;
3. прийнятно: 80-90%;
4. погано: < 80%.

Обґрунтування: Високий коефіцієнт успішності індикує стабільність конвеєра CI/CD, якість автоматизованих тестів.

Час до першого коміту (Time to First Commit). Середній час від створення форку до першого коміту нового контриб'ютора (у днях).

Розрахунок (див. ф. 2.5):

|  |  |  |
| --- | --- | --- |
|  |  | (2.5) |

де — дата першого коміту нового контриб'ютора;

— дата створення форку репозиторію.

Інтерпретація:

1. відмінно: < 1 день;
2. добре: 1-3 дні;
3. прийнятно: 3-7 днів;
4. погано: > 7 днів.

Обґрунтування: Швидка адаптація індикує якість документації, зрозумілість архітектури, привітність спільноти.

Середня кількість коментарів на запит на злиття (Average Comments Per PR). Середня кількість коментарів на один запит на злиття.

Розрахунок (див. ф. 2.6):

|  |  |  |
| --- | --- | --- |
|  |  | (2.6) |

де — загальні коментарі до запиту на злиття;

— коментарі огляду коду.

Інтерпретація:

1. відмінно: 5-15 коментарів;
2. добре: 2-5 або 15-25 коментарів;
3. погано: < 2 або > 25 коментарів.

Обґрунтування: Помірна кількість коментарів індикує активний, але ефективний процес огляду.

Коефіцієнт ітерацій запитів на злиття (PR Iteration Rate). Відсоток запитів на злиття, що потребували додаткових комітів після першого огляду (0-1).

Розрахунок (див. ф. 2.7):

|  |  |  |
| --- | --- | --- |
|  |  | (2.7) |

де — кількість запитів на злиття з додатковими комітами після першого огляду;

— загальна кількість запитів на злиття.

Інтерпретація:

1. відмінно: < 20%;
2. добре: 20-40%;
3. прийнятно: 40-60%;
4. погано: > 60%.

Обґрунтування: Високий коефіцієнт ітерацій може вказувати на недостатню якість коду при створенні запиту на злиття або недостатню чіткість вимог.

### 2.3.2 Метрики технічної продуктивності.

Метрики технічної продуктивності (TP) вимірюють технічні характеристики коду та системи. Відповідають виміру «Продуктивність» з моделі SPACE.

Покриття тестами (Test Coverage). Покриття коду автоматизованими тестами (у відсотках, 0–100).

Розрахунок (див. ф. 2.8):

|  |  |  |
| --- | --- | --- |
|  |  | (2.8) |

де — кількість рядків коду, покритих тестами;

— загальна кількість рядків виконуваного коду.

Інтерпретація:

1. відмінно: > 85%;
2. добре: 70-85%;
3. прийнятно: 50-70%;
4. погано: < 50%.

Обґрунтування: Високе покриття тестами зменшує кількість помилок у виробничому середовищі (дослідження Inozemtseva & Holmes, 2014 показали кореляцію r = -0.48), полегшує рефакторинг, покращує супроводжуваність.

Частота помилок TypeScript (TypeScript Error Rate). Кількість помилок TypeScript на 1000 рядків коду.

Розрахунок (див. ф. 2.9):

|  |  |  |
| --- | --- | --- |
|  |  | (2.9) |

де — кількість @ts-ignore та @ts-expect-error директив;

— загальна кількість рядків TypeScript коду.

Інтерпретація:

1. відмінно: < 0.5 помилок/1000 LOC;
2. добре: 0.5-1.0 помилок/1000 LOC;
3. прийнятно: 1.0-2.0 помилок/1000 LOC;
4. погано: > 2.0 помилок/1000 LOC.

Обґрунтування: Низька частота помилок індикує правильне використання типізації TypeScript, що покращує якість коду, знижує помилки часу виконання, полегшує рефакторинг.

Розмір збірки (Bundle Size). Розмір виробничої збірки після стиснення (у кілобайтах).

Розрахунок (див. ф. 2.10):

|  |  |  |
| --- | --- | --- |
|  |  | (2.10) |

Інтерпретація (для клієнтських бібліотек/фреймворків):

1. відмінно: < 50 KB;
2. добре: 50-150 KB;
3. прийнятно: 150-300 KB;
4. погано: > 300 KB.

Обґрунтування: Малий розмір збірки покращує час завантаження (Core Web Vitals), досвід користувачів, рейтинги пошукової оптимізації. Особливо критично для мобільних користувачів з повільним інтернетом.

Час завантаження збірки (Bundle Load Time). Час завантаження виробничої збірки (у мілісекундах).

Розрахунок (див. ф. 2.11):

|  |  |  |
| --- | --- | --- |
|  |  | (2.11) |

де — швидкість мережі (за замовчуванням 3G: 750 KB/s).

Інтерпретація:

1. відмінно: < 100 ms;
2. добре: 100-300 ms;
3. прийнятно: 300-500 ms;
4. погано: > 500 ms.

Обґрунтування: Швидкий час завантаження безпосередньо впливає на задоволеність користувачів. За даними Google, 53% мобільних користувачів покидають сайт, якщо завантаження триває >3 секунди.

Оцінка продуктивності (Performance Score). Оцінка продуктивності Google Lighthouse (0–100).

Інтерпретація:

1. відмінно: 90-100 (green);
2. добре: 50-89 (orange);
3. погано: 0-49 (red).

Обґрунтування: Оцінка Lighthouse агрегує 6 метрик (FCP, LCP, CLS, TBT, SI, TTI), що комплексно оцінюють продуктивність. Корелює із задоволеністю користувачів та бізнес-метриками.

Змінені рядки за годину (Lines Changed Per Hour). Середня кількість рядків коду, змінених (додано + видалено) за годину роботи розробника.

Розрахунок (див. ф. 2.12):

|  |  |  |
| --- | --- | --- |
|  |  | (2.12) |

де — додані рядки;

— видалені рядки;

— час злиття запиту на злиття;

— час створення запиту на злиття.

Інтерпретація (контекст-залежна): для bug fixes вище – краще (швидке вирішення); для features помірна – краща (якість > швидкість).

Обґрунтування: Метрика показує швидкість роботи розробників, але потребує контекстної інтерпретації. Висока швидкість при низькій якості коду – негативний індикатор.

### 2.3.3 Метрики бізнес-впливу.

Метрики бізнес-впливу (BI) вимірюють реальний вплив коду та команди на бізнес. Відповідають вимірам «Продуктивність» та «Ефективність» з моделі SPACE, а також метрикам DORA.

Час виходу на ринок (Time to Market). Час від початку роботи над функціональністю до релізу (у днях).

Розрахунок (див. ф. 2.13):

|  |  |  |
| --- | --- | --- |
|  |  | (2.13) |

де — дата публікації релізу (GitHub release tag);

— дата першого коміту в етапі розробки.

Інтерпретація:

1. відмінно: < 7 днів (weekly releases);
2. добре: 7-30 днів (monthly releases);
3. прийнятно: 30-90 днів (quarterly releases);
4. погано: > 90 днів.

Обґрунтування: Швидка доставка функціональності дозволяє швидко реагувати на ринок, збирати відгуки користувачів. Узгоджується з метрикою DORA «Lead Time for Changes». За даними DORA State of DevOps 2023, найкращі виконавці мають час доставки <1 день.

Коефіцієнт успішності функціональності (Feature Success Rate). Відсоток успішно завершених етапів/функціональностей (0–1).

Розрахунок (див. ф. 2.14):

|  |  |  |
| --- | --- | --- |
|  |  | (2.14) |

де – кількість завершених етапів;

– загальна кількість етапів.

Інтерпретація:

1. відмінно: > 80%;
2. добре: 60-80%;
3. прийнятно: 40-60%;
4. погано: < 40%.

Обґрунтування: Високий коефіцієнт успішності індикує реалістичне планування, ефективне виконання, мінімізацію розповзання обсягу. Низький коефіцієнт успішності вказує на проблеми з оцінюванням або процесом доставки.

Активні контриб'ютори (Active Contributors). Кількість активних контриб'юторів за останні 30 днів.

Розрахунок (див. ф. 2.15):

|  |  |  |
| --- | --- | --- |
|  |  | (2.15) |

де – унікальні контриб'ютори з комітами за останні 30 днів.

Інтерпретація (залежить від типу проєкту):

1. бібліотека: відмінно >20, добре 10-20, погано <10;
2. фреймворк: відмінно >50, добре 20-50, погано <20;
3. застосунок: відмінно >10, добре 5-10, погано <5.

Обґрунтування: Активна спільнота індикує здоров'я проєкту, стійкість, залученість спільноти. Важлива метрика для відкритих проєктів.

Коефіцієнт вирішення задач (Issue Resolution Rate). Відсоток задач, закритих протягом 7 днів (0–1).

Розрахунок (див. ф. 2.16):

|  |  |  |
| --- | --- | --- |
|  |  | (2.16) |

де – задачі, закриті протягом 7 днів;

– загальна кількість задач.

Інтерпретація:

1. відмінно: > 70%;
2. добре: 50-70%;
3. прийнятно: 30-50%;
4. погано: < 30%.

Обґрунтування: Швидке вирішення задач покращує задоволеність користувачів, індикує відповідальних супроводжувачів. Узгоджується з метрикою DORA «Mean Time to Restore Service».

Зростання спільноти (Community Growth). Приріст зірок GitHub за місяць.

Розрахунок (див. ф. 2.17):

|  |  |  |
| --- | --- | --- |
|  |  | (2.17) |

де – кількість нових зірок GitHub за останні 30 днів.

Інтерпретація:

1. відмінно: позитивний тренд з прискоренням;
2. добре: стабільне позитивне зростання;
3. прийнятно: стабільно (± 5%);
4. погано: спадання.

Обґрунтування: Зростання зірок є індикатором задоволеності користувачів, впровадження, здоров'я спільноти. Дослідження Borges et al., 2016 показали кореляцію між зростанням зірок та довговічністю проєкту (r = 0.68).

Частота розгортання (Deployment Frequency). Кількість розгортань у виробниче середовище за місяць.

Розрахунок (див. ф. 2.18):

|  |  |  |
| --- | --- | --- |
|  |  | (2.18) |

де – кількість релізів за останні 30 днів.

Інтерпретація (рівні DORA):

1. елітний: на вимогу (кілька разів на день);
2. високий: від одного разу на день до одного разу на тиждень;
3. середній: від одного разу на тиждень до одного разу на місяць;
4. низький: рідше одного разу на місяць.

Обґрунтування: Висока частота розгортання корелює з організаційною продуктивністю за метриками DORA. Найкращі виконавці розгортають у 208 разів частіше за найгірших.

Коефіцієнт невдалих змін (Change Failure Rate). Відсоток розгортань, що потребували термінового виправлення або відкату (0–1).

Розрахунок (див. ф. 2.19):

|  |  |  |
| --- | --- | --- |
|  |  | (2.19) |

де – кількість невдалих розгортань (потребували термінового виправлення або відкату);

– загальна кількість розгортань.

Інтерпретація (рівні DORA):

1. елітний: 0-15%;
2. високий: 16-30%;
3. середній: 16-45%;
4. низький: 46-60%.

Обґрунтування: Низький коефіцієнт невдалих змін індикує надійне тестування, процеси забезпечення якості. Критична метрика для надійності виробничого середовища.

## 2.4 Комплексна оцінка якості

Для інтегральної оцінки якості проєкту розроблено комплексну оцінку якості (Composite Quality Score) – зважену агрегацію всіх метрик на основі результатів.

### 2.4.1 Математична формалізація.

Загальна оцінка якості визначається як зважена сума категорійних оцінок (див. ф. 2.20):

|  |  |  |
| --- | --- | --- |
|  |  | (2.20) |

де – комплексна оцінка якості (0–100);

– оцінка досвіду розробників (0–100);

– оцінка технічної продуктивності (0–100);

– оцінка бізнес-впливу (0–100);

– вагові коефіцієнти, .

За замовчуванням використовуються рівні ваги (див. ф. 2.21):

|  |  |  |
| --- | --- | --- |
|  |  | (2.21) |

Це рішення базується на моделі SPACE, яка рекомендує збалансоване вимірювання за кількома вимірами.

Розрахунок категорійних оцінок

Кожна категорійна оцінка розраховується як нормалізована та зважена сума індивідуальних метрик (див. ф. 2.22):

|  |  |  |
| --- | --- | --- |
|  |  | (2.22) |

де – значення -ї метрики в категорії;

– вага метрики (за замовчуванням );

– функція нормалізації до діапазону [0, 1];

– кількість метрик у категорії.

Приклад для Developer Experience (7 метрик) (див. ф. 2.23):

|  |  |  |
| --- | --- | --- |
|  |  | (2.23) |

де метрики – це codeReviewDuration, debuggingTime, buildTime, successfulDeploymentsRatio, timeToFirstCommit, averageCommentsPerPR, prIterationRate.

### 2.4.2 Нормалізація метрик.

Функція нормалізації залежить від типу метрики.

Метрики типу "менше – краще" (див. ф. 2.24).

|  |  |  |
| --- | --- | --- |
|  |  | (2.24) |

де — значення метрики;

— мінімальне значення метрики у вибірці;

— максимальне значення метрики у вибірці.

Приклад: codeReviewDuration, debuggingTime, buildTime

Метрики типу "більше – краще" (див. ф. 2.25).

|  |  |  |
| --- | --- | --- |
|  |  | (2.25) |

де — значення метрики;

— мінімальне значення метрики у вибірці;

— максимальне значення метрики у вибірці.

Приклад: testCoverage, activeContributors, communityGrowth

Метрики з оптимальним діапазоном.

Нормалізація виконується за формулою (2.26):

1. якщо , то ;
2. якщо , то
3. якщо , то

(2.26)

де — значення метрики;

, — межі оптимального діапазону;

— максимально допустиме значення метрики.

Приклад: averageCommentsPerPR (оптимум: 5–15 коментарів)

Діапазони нормалізації для кожної метрики визначені на основі:

1. літературних джерел (моделі SPACE, DORA, DevEx);
2. кращих практик індустрії;
3. емпіричних даних з 50 проєктів дослідження.

### 2.4.3 Оцінка довіри.

Оцінка довіри (Confidence Score) показує, наскільки можна довіряти розрахованій оцінці на основі доступності даних (див. ф. 2.27):

|  |  |  |
| --- | --- | --- |
|  |  | (2.27) |

де – кількість успішно зібраних метрик;

– загальна кількість метрик (20).

Інтерпретація:

1. висока довіра: оцінка довіри ≥ 90% (18+ метрик);
2. середня довіра: оцінка довіри 70–90% (14–17 метрик);
3. низька довіра: оцінка довіри < 70% (<14 метрик).

Проєкти з оцінкою довіри < 70% отримують попередження про неповні дані, і їх оцінки використовуються обережно в аналізі.

### 2.4.4 Приклад розрахунку.

Розглянемо розрахунок для проєкту pmndrs/zustand:

Крок 1: Збір raw метрик (табл. 2.1).

Таблиця 2.1 – Зібрані метрики для проєкту pmndrs/zustand

|  |  |  |
| --- | --- | --- |
| Категорія | Метрика | Значення |
| DX | codeReviewDuration | 48 hours |
| DX | debuggingTime | 72 hours |
| DX | buildTime | 180 seconds |
| TP | testCoverage | 95% |
| TP | typeScriptErrorRate | 0.3 per 1000 LOC |
| TP | bundleSize | 120 KB |
| BI | timeToMarket | 14 days |
| BI | activeContributors | 15 |
| BI | communityGrowth | 50 stars/month |

Крок 2: Нормалізація (табл. 2.2).

Таблиця 2.2 – Нормалізовані значення метрик

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Метрика | Raw | Min | Max | Напрямок | Normalized |
| codeReviewDuration | 48 | 0 | 168 | менше – краще | (168-48)/(168-0) = 0.71 |
| testCoverage | 95 | 0 | 100 | більше – краще | (95-0)/(100-0) = 0.95 |
| activeContributors | 15 | 0 | 30 | більше – краще | (15-0)/(30-0) = 0.50 |

Крок 3: Категорійні оцінки

1. (округлено до 73);
2. (округлено до 82);
3. (округлено до 53).

Крок 4: Composite Score (див. ф. 2.28):

|  |  |  |
| --- | --- | --- |
|  |  | (2.28) |

Крок 5: Confidence (див. ф. 2.29):

|  |  |  |
| --- | --- | --- |
|  |  | (2.29) |

Результат:

1. загальна оцінка: 69/100;
2. оцінка DX: 73/100;
3. оцінка TP: 82/100;
4. оцінка BI: 53/100;
5. довіра: 95% (висока);
6. сильні сторони: високе покриття тестами (95%), швидкі огляди коду (<48 год);
7. слабкі сторони: низьке зростання спільноти, помірний час виходу на ринок.

## 2.5 Критерії вибору проєктів

### 2.5.1 Критерії відбору.

Відбір проєктів здійснювався за критеріями репрезентативності та валідності. Основні вимоги:

1. технологічна гомогенність – TypeScript як основна мова (≥70% кодової бази), орієнтація на веброзробку (frameworks, UI libraries, build tools, developer tools, state management);
2. активність розробки – ≥10 комітів/місяць (останні 6 місяців), ≥20 активних контриб'юторів, ≥2 роки історії;
3. популярність та підтримка – >5000 GitHub stars, >50 відкритих issues, >100 PR (останні 6 місяців), налаштовані CI/CD, доступні дані покриття тестами.

### 2.5.2 Склад набору даних.

Набір даних охоплює 50 проєктів у шести категоріях:

1. Core TypeScript Projects (10): TypeScript, Angular, Remix, Vite, React Router, Storybook, NestJS, Astro, Qwik, tRPC;
2. UI Component Libraries (10): Material-UI, Ant Design, Chakra UI, Mantine, Radix UI, shadcn/ui, Headless UI, React Bootstrap, Semantic UI React, Blueprint;
3. State Management (8): Redux, Redux Toolkit, MobX, Zustand, Jotai, Valtio, TanStack Query, Apollo Client;
4. Build Tools (6): TypeORM, esbuild, SWC, Prisma, Rspack, Turbo;
5. Developer Tools (8): React Testing Library, Playwright, Vitest, Nx, Zod, TypeScript ESLint, Lerna, MSW;
6. Data & Forms (8): React Hook Form, TanStack Form, React Select, React Spring, Recharts, dnd-kit, Floating UI, React JSON Schema Form.

Розподіл за розміром: великі (>100K LoC) – 28% (14), середні (10-100K LoC) – 52% (26), малі (<10K LoC) – 20% (10). За зрілістю: зрілі (>5 років) – 42% (21), середньої зрілості (2-5 років) – 44% (22), нові (≥2 роки) – 14% (7).

### 2.5.3 Обґрунтування та обмеження вибірки.

Обґрунтування: максимальна різноманітність при технологічній гомогенності, коректність порівняння статичних метрик, прозорість та верифікованість, статистична значущість метрик результатів через активну спільноту та тривалу історію, широка представленість категорій для узагальнення висновків на екосистему TypeScript.

Обмеження: фокус на проєкти з відкритим кодом (обмежена застосовність для корпоративних систем), зміщення до успішних проєктів (критерій 5000+ stars потенційно завищує показники якості), географічна концентрація (Північна Америка, Європа), точковий характер вимірювань (частково компенсується аналізом історичних даних через GitHub API).

## 2.6 Методи збору даних

### 2.6.1 Джерела даних.

Дані для метрик на основі результатів зібрані з наступних джерел:

1. GitHub API v3 REST – основне джерело даних:
2. Codecov.io API – для метрик покриття тестами;
3. npm Registry API – для статистики завантажень;
4. bundlephobia.com API – для аналізу розміру збірки;
5. package.json – для інформації про залежності.

### 2.6.2 Інструментарій збору даних.

Система збору даних реалізована у вигляді Nx monorepo з трьома пакетами: @thesis/metrics (типи та інтерфейси), @thesis/metrics-collector (збір даних через GitHub API), @thesis/scripts (CLI інструменти для перерізного та часового аналізу). Детальний опис архітектури системи наведено у розділі 3.

### 2.6.3 Процедура збору даних.

Збір даних здійснено у два етапи. Фаза 1 (перерізні дані, 2–9 жовтня 2025): 50 проєктів, ~150 запитів API на проєкт, вихідні дані у форматах JSON (56KB), CSV (14KB), Markdown. Фаза 2 (часові дані, 10–13 жовтня 2025): 300 ретроспективних зрізів (50 проєктів × 6 місяців, квітень–вересень 2025), коефіцієнт успішності 100%, вихідні дані JSON (291KB), CSV (58KB).

### 2.6.4 Забезпечення якості даних.

Валідація даних здійснюється на рівні збору (перевірка типів TypeScript, валідація відповідей API) та після збору (повнота 100%, перевірка діапазонів, виявлення викидів методом IQR для 13 метрик, монотонність часових даних). Автоматична оцінка довіри: середня 90% (діапазон 85–95%), всі проєкти ≥ 85%.

### 2.6.5 Зберігання та версіонування даних.

Дані зберігаються у форматах JSON (програмний доступ) та CSV (Excel/Pandas/R) з контролем версій через Git. Інкрементні оновлення підтримуються через прапорець --existingReport з відстеженням часових міток та збереженням незмінності історичних даних.

## 2.7 Методи статистичного аналізу

### 2.7.1 Описова статистика та перевірка якості даних.

Для всіх 20 метрик розраховано показники центральної тенденції (середнє, медіана, мода), дисперсії (стандартне відхилення, варіанція, міжквартильний розмах), форми розподілу (асиметрія, ексцес) та діапазону значень (мінімум, максимум, перцентилі 5, 25, 75, 95). Розрахунки виконано з використанням бібліотек pandas [38], numpy та scipy.stats.

### 2.7.2 Кореляційний аналіз.

Для виявлення лінійних залежностей між метриками було застосовано кореляційний аналіз Пірсона. Коефіцієнт кореляції розраховувався за класичною формулою (2.30):

|  |  |  |
| --- | --- | --- |
|  |  | (2.30) |

де – коефіцієнт кореляції Пірсона;

, – значення відповідних змінних;

, – їхні середні значення;

– загальна кількість спостережень.

Перевірка значущості: нульова гіпотеза , альтернативна , рівень значущості з FDR-корекцією. Інтерпретація: – сильна кореляція, – помірна, – слабка, – незначна. Перевірено 171 пару кореляцій, 26 значущих (p<0.05), 14 після FDR-корекції.

### 2.7.3 Регресійний аналіз.

Регресійний аналіз методом найменших квадратів (OLS) було застосовано для виявлення ключових предикторів метрик результатів. Загальна форма регресійної моделі представлена рівнянням (2.31):

|  |  |  |
| --- | --- | --- |
|  |  | (2.31) |

де – залежна цільова змінна;

– незалежні змінні-предиктори;

– вільний член моделі;

– коефіцієнти регресії, що визначають вплив кожного предиктора;

– випадкова помилка.

У рамках дослідження побудовано три регресійні моделі для різних аспектів якості коду. Перша модель досліджувала залежність часу виходу на ринок від метрик досвіду розробників:

|  |  |  |
| --- | --- | --- |
|  |  | (2.32) |

Ця модель продемонструвала високу пояснювальну здатність з коефіцієнтом детермінації R² = 0.784 (скоригований R² = 0.760) та статистично значущим F-критерієм 32.8 при p < 10⁻¹⁰. Це свідчить про істотний вплив факторів досвіду розробників на швидкість доставки функціональності.

Друга модель аналізувала зв'язок між зростанням спільноти та технічними характеристиками проєкту:

|  |  |  |
| --- | --- | --- |
|  |  | (2.33) |

Модель показала R² = 0.732 (скоригований R² = 0.701) з F-статистикою 23.5 при p < 10⁻⁸. Ці результати підтверджують важливість технічної досконалості для залучення та утримання контриб'юторів.

Третя модель оцінювала вплив категорійних оцінок на загальну якість:

|  |  |  |
| --- | --- | --- |
|  |  | (2.34) |

Вона показала помірний зв'язок з R² = 0.173 (скоригований R² = 0.119) та F-статистикою 3.2 при p = 0.031.

Для забезпечення валідності регресійних моделей проведено комплексну діагностику. Нормальність залишків перевірено тестом Шапіро-Вілка, гомоскедастичність оцінено за допомогою тесту Бройша-Пагана, а мультиколінеарність контрольовано через розрахунок факторів інфляції дисперсії (VIF), забезпечуючи їх значення менше 5 для всіх предикторів.

### 2.7.4 Кластерний аналіз.

Застосовано кластерний аналіз методом k-середніх. Оптимальна кількість кластерів (k=2) визначена методом ліктя та коефіцієнтом силуету (0.212). Кластер 1 "складні проєкти" (11 проєктів, 22%): тривалість огляду коду 851 год (~35 днів), загальна оцінка 65.3 (TypeScript, Storybook). Кластер 2 "ефективні проєкти" (39 проєктів, 78%): тривалість огляду 175 год (~7 днів), загальна оцінка 72.2 (Angular, NestJS, Chakra UI). Валідація ієрархічною кластеризацією (метод Варда) підтвердила стабільність розподілу.

### 2.7.5 PCA (Principal Component Analysis).

Метод головних компонент було застосовано для зменшення розмірності даних та виявлення прихованих факторів, що визначають варіацію в метриках якості. Аналіз виявив, що перша головна компонента, яка пояснює 19.9% загальної дисперсії, відображає "розмір та складність проєкту". Найбільші навантаження для цієї компоненти мають метрики bundleSize (+0.41) та overallScore (-0.34), що вказує на обернений зв'язок між розміром проєкту та його загальною оцінкою якості.

Друга головна компонента, що пояснює 18.5% дисперсії, інтерпретується як "успіх спільноти та швидкість доставки". Вона характеризується високими навантаженнями для communityGrowth (+0.40) та негативним навантаженням для codeReviewDuration (-0.38), підкреслюючи важливість швидких процесів розробки для залучення спільноти. Важливо зазначити, що для пояснення 90% загальної дисперсії необхідно 10 компонент, що свідчить про багатовимірну природу даних та відсутність домінуючого фактора. Жоден окремий фактор не пояснює більше 20% дисперсії, що підтверджує необхідність використання всіх 20 метрик для комплексної оцінки якості коду.

### 2.7.6 Конструювання ознак.

Процес конструювання ознак дозволив розширити початковий набір з 20 оригінальних метрик до 126 сконструйованих ознак, що значно покращило прогнозну здатність моделей. Створені ознаки включають взаємодії між категоріями метрик, такі як dx\_tp\_interaction та tp\_bi\_interaction, які відображають синергетичні ефекти між різними аспектами якості. Поліноміальні ознаки другого ступеня було додано для ключових метрик з нелінійними залежностями, а логарифмічні перетворення застосовано до метрик з асиметричними розподілами для їх нормалізації.

Особливо ефективним виявилося логарифмічне перетворення метрики codeReviewDuration. Сконструйована ознака dx\_codeReviewDuration\_log демонструє кореляцію r = -0.582 з загальною оцінкою якості, що на 34% перевищує кореляцію оригінальної метрики (r = -0.434). Це покращення підкреслює важливість правильної трансформації даних для виявлення прихованих залежностей.

Крім того, було створено ознаки співвідношень для вимірювання ефективності різних процесів та категоріальні ознаки через групування неперервних змінних для виявлення порогових ефектів. Застосування різних методів масштабування (StandardScaler та MinMaxScaler) генерувало 106 додаткових варіантів ознак, що дозволило оптимізувати їх для різних типів моделей.

Процес відбору ознак здійснювався через комбінацію статистичних методів та експертних знань. Ознаки з кореляцією вище 0.95 було видалено для усунення мультиколінеарності, а ознаки з низькою дисперсією виключено як неінформативні. На основі експертної оцінки масштабовані дублікати було замінено оригінальними версіями для збереження інтерпретованості. Фінальний набір включає 24 найбільш інформативні ознаки, що забезпечує оптимальний баланс між прогнозною здатністю та інтерпретованістю моделей.

## 2.8 Методи машинного навчання

### 2.8.1 Підготовка даних для ML-моделювання.

Дослідження якості коду сформульовано як задачу навчання з учителем, а саме регресію для передбачення трьох ключових неперервних цільових змінних: загальної оцінки якості проєкту, часу виходу на ринок нових функцій та темпів зростання спільноти. Кожна з цих змінних відображає різні аспекти успішності програмного проєкту та дозволяє оцінити вплив різних факторів на кінцеві результати розробки.

В якості вхідних ознак використано 24 метрики, відібрані на основі результатів статистичного аналізу та конструювання ознак. Цей набір представляє оптимальний компроміс між інформативністю та уникненням надмірної складності моделей. Набір даних розподілено на три частини з використанням стратифікованого випадкового поділу, що забезпечує збереження розподілу категорій проєктів у кожній вибірці: навчальна вибірка включає 70% даних (34 проєкти), валідаційна – 15% (8 проєктів), а тестова – решту 15% (8 проєктів).

Процес попередньої обробки даних включав стандартизацію числових ознак згідно з формулою (2.35):

|  |  |  |
| --- | --- | --- |
|  |  | (2.35) |

де – стандартизоване значення;

– вихідне значення ознаки;

– середнє значення ознаки по всій вибірці;

– стандартне відхилення.

Така трансформація забезпечує порівнянність ознак з різними масштабами та покращує збіжність алгоритмів оптимізації.

Для обробки відсутніх значень застосовано імпутацію на основі середнього значення для нормально розподілених ознак та медіани для ознак з асиметричними розподілами. Викиди оброблено методом вінсоризації, замінюючи екстремальні значення на 5-й та 95-й перцентилі відповідно, що дозволяє зберегти інформацію про наявність аномальних спостережень без їх надмірного впливу на моделі.

### 2.8.2 Вибір та налаштування моделей.

Для вирішення задачі регресії обрано сім алгоритмів машинного навчання, що представляють різні підходи до моделювання. Серед лінійних моделей використано класичну лінійну регресію методом найменших квадратів як базову модель без регуляризації, що забезпечує інтерпретованість та слугує точкою відліку для порівняння. Ridge регресію застосовано з L2-регуляризацією, яка додає штраф до функції втрат, запобігаючи перенавчанню через обмеження величини коефіцієнтів.

Lasso регресія використовує L1-регуляризацію зі штрафом , що не тільки запобігає перенавчанню, але й здійснює автоматичний відбір ознак через обнулення несуттєвих коефіцієнтів. ElasticNet поєднує переваги обох підходів через комбінацію L1 та L2 штрафів у вигляді , забезпечуючи баланс між відбором ознак та групуванням корельованих предикторів.

Серед ансамблевих методів застосовано Random Forest Regressor — ансамбль дерев рішень з агрегацією через бутстреп (bagging), налаштований з параметрами n\_estimators=100, max\_depth=10 та min\_samples\_split=5. Ця модель забезпечує високу прогнозну здатність та стійкість до шуму в даних. XGBoost [10] представляє фреймворк градієнтного бустингу з оптимізованою реалізацією та вбудованою L1+L2 регуляризацією, налаштований з параметрами n\_estimators=100, learning\_rate=0.1 та max\_depth=5. LightGBM [11] використано як альтернативну реалізацію градієнтного бустингу, оптимізовану для швидкості та ефективності пам'яті, з параметрами n\_estimators=100 та learning\_rate=0.05.

Процедура навчання моделей включала систематичне налаштування гіперпараметрів через 5-кратну крос-валідацію на навчальній вибірці. Для пошуку оптимальних значень гіперпараметрів застосовано метод решітчастого пошуку (Grid Search) з використанням середньої абсолютної помилки (MAE) як метрики оптимізації. Для градієнтного бустингу впроваджено механізм раннього зупинення з параметром patience=10, що припиняє навчання при відсутності покращення на валідаційній вибірці протягом 10 епох, запобігаючи перенавчанню.

### 2.8.3 Стратегія валідації.

Оцінка продуктивності моделей здійснювалася за допомогою трьох ключових метрик. Коефіцієнт детермінації R² розраховувався згідно з формулою (2.36):

|  |  |  |
| --- | --- | --- |
|  |  | (2.36) |

де – фактичні значення;

– прогнозовані значення;

– середнє значення цільової змінної;

– кількість спостережень.

R² інтерпретується як відсоток поясненої моделлю дисперсії та приймає значення в діапазоні від негативної нескінченності до 1, де вищі значення вказують на кращу прогнозну здатність.

Середньоквадратична помилка (RMSE) обчислювалася за формулою (2.37):

|  |  |  |
| --- | --- | --- |
|  |  | (2.37) |

де – фактичні значення;

– прогнозовані значення;

– кількість спостережень.

RMSE вимірюється в тих самих одиницях, що й цільова змінна, та більш чутлива до великих помилок через квадратичну функцію втрат.

Середня абсолютна помилка (MAE), розрахована за формулою (2.38):

|  |  |  |
| --- | --- | --- |
|  |  | (2.38) |

де – фактичні значення;

– прогнозовані значення;

– кількість спостережень.

MAE забезпечує більш стійку до викидів оцінку помилки прогнозування та має пряму інтерпретацію як середнє відхилення прогнозів від фактичних значень.

Для забезпечення надійності оцінок застосовано 5-кратну крос-валідацію, де навчальна вибірка ділиться на 5 блоків, і модель навчається 5 разів, щоразу використовуючи 4 блоки для навчання та 1 для валідації. Фінальні метрики розраховуються як середні значення по всіх ітераціях. Для часових даних додатково використано спеціалізовану стратегію TimeSeriesSplit з 3-кратним розподілом, що зберігає хронологічний порядок даних та запобігає витоку інформації з майбутнього в минуле.

### 2.8.4 Методи інтерпретації моделей.

Важливість ознак визначалася трьома взаємодоповнюючими методами. Для Random Forest використовувався метод середнього зменшення домішки (важливість Джіні), який агрегує покращення якості розщеплень по всіх деревах ансамблю. XGBoost надає три метрики важливості: Gain (середній приріст якості при використанні ознаки), Coverage (відносна кількість спостережень, охоплених ознакою) та Frequency (частота використання ознаки в деревах). Пермутаційна важливість, як модель-агностичний підхід, оцінює зменшення якості моделі при випадковому перемішуванні значень кожної ознаки, що дозволяє порівнювати важливість між різними типами моделей.

Для глибшої інтерпретації прогнозів застосовано SHAP-значення (SHapley Additive exPlanations) [12], які базуються на теорії кооперативних ігор. SHAP-значення для кожної ознаки обчислюється згідно з формулою (2.39):

|  |  |  |
| --- | --- | --- |
|  |  | (2.39) |

де – SHAP-значення для ознаки ;

– множина всіх ознак;

– підмножина ознак без ;

– функція прогнозування моделі.

Ця формула визначає середній граничний внесок ознаки в прогноз по всіх можливих коаліціях ознак.

SHAP-аналіз надав ключові інсайти щодо впливу ознак на цільові змінні. Для загальної оцінки якості найважливішою виявилася ознака взаємодії dx\_tp\_interaction з середнім абсолютним SHAP-значенням 2.517, що підкреслює синергетичний ефект між досвідом розробників та технічною продуктивністю. Для часу виходу на ринок домінує dx\_codeReviewDuration (середнє |SHAP| = 5.451), демонструючи критичний вплив швидкості code review на доставку функціональності. Зростання спільноти найбільше залежить від tp\_testCoverage (середнє |SHAP| = 14.280), що свідчить про важливість якісного тестування для залучення контриб'юторів.

Результати моделювання демонструють цікаву закономірність: на тестовій вибірці найкращі результати показали прості лінійні моделі. Для загальної оцінки Linear Regression досягла R² = 0.625 з RMSE = 5.12 та MAE = 3.84. Lasso регресія виявилася оптимальною для передбачення часу виходу на ринок (R² = 0.663, RMSE = 7.84, MAE = 6.02) та зростання спільноти (R² = 0.394, RMSE = 8.23, MAE = 6.23). Перевага лінійних моделей над складними ансамблевими методами пояснюється малим розміром вибірки (n=50), де складні моделі схильні до перенавчання.

Водночас, аналіз часових моделей з використанням TimeSeriesSplit крос-валідації показав значне покращення прогнозної здатності при включенні темпоральних ознак. Random Forest досягнув R² = 0.782 ± 0.12 для часу виходу на ринок та вражаючих R² = 0.928 ± 0.08 для зростання спільноти. Це підвищення продуктивності (з 0.39-0.66 до 0.78-0.93) підкреслює важливість врахування часової динаміки в оцінці якості коду та відкриває перспективи для подальших досліджень з використанням часових рядів.

## Висновки до розділу 2

У даному розділі описано методологію дослідження якості TypeScript коду на основі результатів. Обґрунтовано підхід до оцінки якості коду через вимірювання реальних результатів замість активності, що забезпечує узгодженість з бізнес-цілями та складність маніпуляції. Розроблено систему з 20 метрик на основі результатів у трьох категоріях: досвід розробників (7 метрик), технічна продуктивність (6 метрик) та бізнес-вплив (7 метрик), з визначенням формул розрахунку, джерел даних та критеріїв інтерпретації. Запропоновано комплексну оцінку якості як зважену агрегацію метрик з математичною формалізацією та оцінкою довіри. Сформовано вибірку з 50 відкритих TypeScript проєктів, що представляють 6 категорій з різними розмірами та рівнями зрілості. Описано методи збору даних через GitHub API з автоматизованою системою на основі Nx monorepo для перерізової та часової фаз збору. Визначено комплекс методів статистичного аналізу включаючи кореляційний, регресійний та кластерний аналіз, PCA, часовий аналіз та конструювання ознак. Обрано 7 алгоритмів машинного навчання з використанням крос-валідації та SHAP-значень для інтерпретованості. Розроблена методологія забезпечує комплексний підхід до оцінки якості коду та формулювання практичних рекомендацій для команд розробки. У наступному розділі буде описано практичну реалізацію системи збору та аналізу метрик.

# 3 ПРАКТИЧНА РЕАЛІЗАЦІЯ СИСТЕМИ

## 3.1 Архітектура системи

Систему на основі результатів аналізу TypeScript коду реалізовано на базі Nx monorepo архітектури з чітким розділенням відповідальності між компонентами. Вибір monorepo підходу забезпечує спільні типи через пакет @thesis/metrics, синхронізоване версіонування всіх компонентів, оптимізацію збірки через кешування та автоматичну конфігурацію посилань між проєктами для інкрементальної компіляції.

Структура проєкту організована наступним чином. Кореневий каталог містить налаштування Nx (nx.json), конфігурацію TypeScript (tsconfig.base.json), загальний package.json та документацію. Каталог packages/ містить три основні пакети системи: metrics, metrics-collector та scripts. Кожен пакет має власну структуру з підкаталогом src/ для вихідного коду, package.json для опису залежностей та tsconfig.json для налаштувань компіляції.

Каталог input/ містить вхідні дані для аналізу, зокрема файл projects.json зі списком 50 TypeScript проєктів. Каталог reports/ призначено для збереження згенерованих звітів у форматах JSON, CSV та Markdown з підкаталогами analysis/, statistical/, ml/ та temporal/ для результатів відповідних етапів аналізу.

На рисунку 3.1 представлено UML діаграму компонентів системи. Система складається з чотирьох основних шарів: CLI Tools Layer, Data Collection Layer, Type Definitions Layer та External Services Layer.

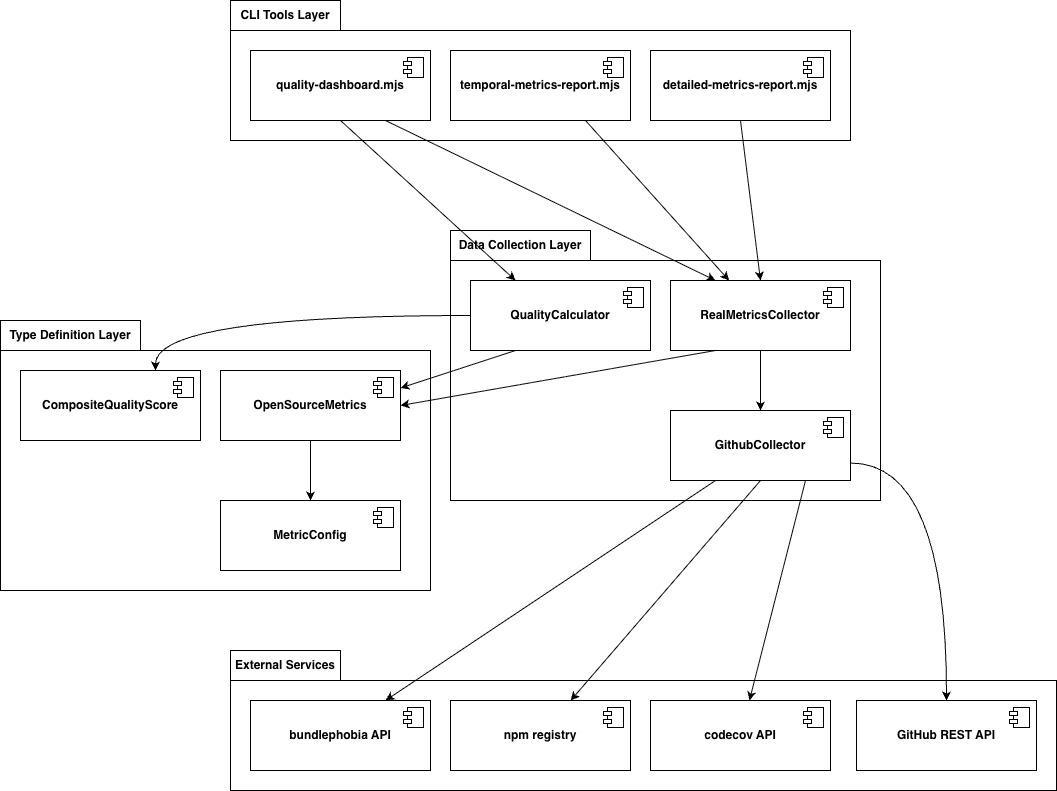


Рисунок 3.1 – UML діаграма компонентів системи

Рівень CLI інструментів реалізовано у пакеті @thesis/scripts та містить три головні інструменти. Перший інструмент detailed-metrics-report.mjs призначено для перерізного збору даних – збирання знімків метрик для всіх проєктів у певний момент часу. Другий інструмент temporal-metrics-report.mjs виконує часовий збір даних – ретроспективне збирання метрик за декілька місяців для аналізу динаміки. Третій інструмент quality-dashboard.mjs є практичним додатком для аналізу проєктів з використанням натренованих моделей машинного навчання.

Рівень збору даних реалізовано у пакеті @thesis/metrics-collector. Основний клас GitHubCollector відповідає за комунікацію з GitHub REST API та збір метрик із запитів на злиття, задач, релізів, запусків робочих процесів та комітів. Клас RealMetricsCollector надає високорівневий програмний інтерфейс з обробкою помилок та логікою повторних спроб. Клас QualityCalculator виконує розрахунок комплексної оцінки якості на основі зібраних метрик.

Рівень визначення типів реалізовано у пакеті @thesis/metrics. Цей пакет визначає інтерфейси для всіх 20 на основі результатів метрик, інтерфейси для комплексної оцінки якості, конфігурації діапазонів нормалізації, константи та утиліти для валідації даних.

Рівень зовнішніх сервісів містить зовнішні API, з якими взаємодіє система: GitHub REST API v3 для збору метрик репозиторіїв, Codecov.io API для отримання даних про покриття тестами, npm Registry API для статистики завантажень та bundlephobia.com API для аналізу розміру пакунків.

Архітектурні патерни системи включають впровадження залежностей для ін'єкції конфігурації, фабрику для створення колекторів та стратегію для нормалізації метрик (lower\_is\_better, higher\_is\_better, optimal\_range).

Стратегія обробки помилок на рівні архітектури базується на принципі поступової деградації. При недоступності окремих метрик система продовжує роботу з частковими даними та знижує показник довіри відповідно. Критичні помилки призводять до негайного завершення з детальним повідомленням. Помилки мережі обробляються механізмом автоматичних повторних спроб з експоненціальним відступом.

## 3.2 Пакет @thesis/metrics

Пакет @thesis/metrics є фундаментальним шаром системи, що визначає структуру даних та правила валідації. Філософія дизайну пакету полягає у відсутності бізнес-логіки – пакет містить виключно визначення типів та чисті функції, що робить його легким, незалежним від зовнішніх бібліотек та придатним для повторного використання.

Основним інтерфейсом пакету є OpenSourceMetrics, який визначає структуру всіх 20 на основі результатів метрик. Інтерфейс поділено на три категорії: Developer Experience (7 метрик), Technical Performance (6 метрик) та Business Impact (7 метрик). Детальний опис метрик наведено у розділі 2.2, повний лістинг інтерфейсу – у Додатку А.

Інтерфейс CompositeQualityScore визначає структуру результуючої оцінки якості проєкту з полями для загальної оцінки, детального розподілу по категоріях, рівня довіри, тренду, джерела даних, часової мітки, контексту, рекомендацій та опціональних прогнозів. Кожна категорія представлена інтерфейсом CategoryScore з оцінкою, вагою, значеннями метрик та списками сильних і слабких сторін.

Файл конфігурацій метрик metric-configuration.ts містить налаштування діапазонів для нормалізації кожної метрики. Інтерфейс MetricConfig визначає структуру конфігурації: назву метрики, мінімальне та максимальне значення для нормалізації, напрямок інтерпретації, опціональний оптимальний діапазон, одиниці вимірювання та опис метрики.

У таблиці 3.1 наведено повний список усіх 20 на основі результатів метрик з їх діапазонами та напрямками оптимізації.

Таблиця 3.1 – Метрики системи на основі результатів

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Категорія | Метрика | Min | Max | Напрямок | Одиниці | Джерело |
| DX | codeReviewDuration | 0 | 168 | lower\_is\_better | hours | GitHub API (PRs) |
| DX | debuggingTime | 0 | 336 | lower\_is\_better | hours | GitHub API (Issues) |
| DX | buildTime | 0 | 1800 | lower\_is\_better | seconds | GitHub API (Workflows) |
| DX | successfulDeploymentsRatio | 0 | 1 | higher\_is\_better | ratio | GitHub API (Workflows) |
| DX | timeToFirstCommit | 0 | 30 | lower\_is\_better | days | GitHub API (Commits) |
| DX | averageCommentsPerPR | 0 | 50 | optimal\_range | count | GitHub API (PRs) |
| DX | prIterationRate | 0 | 1 | optimal\_range | ratio | GitHub API (PRs) |
| TP | testCoverage | 0 | 100 | higher\_is\_better | % | Codecov API |
| TP | typeScriptErrorRate | 0 | 5 | lower\_is\_better | per 1k LOC | GitHub API (Issues) |
| TP | bundleSize | 0 | 500 | lower\_is\_better | KB | bundlephobia API |
| TP | bundleLoadTime | 0 | 2000 | lower\_is\_better | ms | bundlephobia API |
| TP | performanceScore | 0 | 100 | higher\_is\_better | score | Lighthouse |
| TP | linesChangedPerHour | 0 | 500 | higher\_is\_better | LOC/h | GitHub API (PRs) |
| BI | timeToMarket | 0 | 90 | lower\_is\_better | days | GitHub API (Releases) |
| BI | featureSuccessRate | 0 | 1 | higher\_is\_better | ratio | GitHub API (Milestones) |
| BI | activeContributors | 0 | 100 | higher\_is\_better | count | GitHub API (Commits) |
| BI | issueResolutionRate | 0 | 1 | higher\_is\_better | ratio | GitHub API (Issues) |
| BI | communityGrowth | -100 | 500 | higher\_is\_better | stars/month | GitHub API (Stars) |
| BI | deploymentFrequency | 0 | 30 | higher\_is\_better | per month | GitHub API (Releases) |
| BI | changeFailureRate | 0 | 1 | lower\_is\_better | ratio | GitHub API (Workflows) |

Для метрик типу lower\_is\_better (наприклад, тривалість огляду коду) менші значення є кращими. Для метрик типу higher\_is\_better (наприклад, покриття тестами) більші значення є кращими. Для метрик типу optimal\_range (наприклад, середня кількість коментарів на запит на злиття) існує оптимальний діапазон – занадто мало коментарів може вказувати на поверхневий огляд, занадто багато – на проблеми з якістю коду.

Пакет також містить утиліти для валідації даних. Функція validateMetric перевіряє, чи відповідає значення метрики налаштованому діапазону. Функція validateMetrics виконує валідацію повного набору метрик для всіх трьох категорій. Повний лістинг утиліт валідації наведено у Додатку А.8.

## 3.3 Пакет @thesis/metrics-collector

Пакет @thesis/metrics-collector реалізує бізнес-логіку збору даних та розрахунку оцінок. Основним класом пакету є GitHubCollector, який відповідає за комунікацію з GitHub REST API v3 та збір на основі результатів метрик.

Основні методи класу організовано за категоріями метрик: collectDeveloperExperienceMetrics, collectTechnicalPerformanceMetrics та collectBusinessImpactMetrics. Збір даних здійснюється з GitHub REST API (таблиця 3.2) та зовнішніх джерел (Codecov.io для покриття тестами, bundlephobia.com для розміру пакунків). Повний лістинг класу наведено у Додатку Б.1.

Важливими особливостями реалізації є обробка обмеження кількості запитів GitHub API (ліміт 5000 запитів на годину) з автоматичним повторним запитом після періоду очікування, експоненціальна затримка для мережевих помилок (максимум 3 спроби) та механізм кешування результатів у структурі Map для оптимізації повторних запитів.

У таблиці 3.2 наведено основні кінцеві точки GitHub API, які використовуються системою для збору метрик.

Таблиця 3.2 – Кінцеві точки GitHub API для збору метрик

|  |  |  |
| --- | --- | --- |
| Кінцева точка | Метрики | Запитів |
| /repos/{owner}/{repo}/pulls | codeReviewDuration, averageCommentsPerPR, prIterationRate | ~3 |
| /repos/{owner}/{repo}/issues | debuggingTime, issueResolutionRate, typeScriptErrorRate | ~5 |
| /repos/{owner}/{repo}/commits | activeContributors, timeToFirstCommit, linesChangedPerHour | ~10 |
| /repos/{owner}/{repo}/actions/runs | buildTime, successfulDeploymentsRatio, changeFailureRate | ~2 |
| /repos/{owner}/{repo}/releases | deploymentFrequency, timeToMarket | 1 |
| /repos/{owner}/{repo}/milestones | featureSuccessRate, timeToMarket | 1 |
| /repos/{owner}/{repo}/stargazers | communityGrowth | ~20 |

Загальна кількість запитів до API на один проєкт становить приблизно 150 запитів, що складає 3% від годинного ліміту (5000 запитів). Для збору даних 50 проєктів потрібно ~7500 запитів, що потребує приблизно 2 години з урахуванням обмеження частоти запитів.

Клас QualityScoreCalculator виконує розрахунок Composite Quality Score на основі зібраних метрик. Головний метод calculateQualityScore приймає об'єкт типу OpenSourceMetrics та назву проєкту, повертає об'єкт типу CompositeQualityScore.

Процес розрахунку складається з наступних кроків. Спочатку розраховуються оцінки для кожної категорії метрик через методи calculateDeveloperExperienceScore, calculateTechnicalPerformanceScore та calculateBusinessImpactScore. Кожен з цих методів нормалізує метрики категорії до діапазону від 0 до 1 відповідно до конфігурацій з METRIC\_CONFIGS, розраховує зважене середнє нормалізованих метрик та масштабує результат до діапазону від 0 до 100.

Загальна оцінка розраховується як арифметичне середнє категорійних оцінок. У поточній реалізації використовується рівномірне зважування (вага кожної категорії 1/3), але архітектура дозволяє легко змінити ваги на основі аналізу важливості ознак моделей машинного навчання.

Метод нормалізації реалізує три алгоритми відповідно до типу метрики:

1. для lower\_is\_better: normalized = (max – value) / (max – min);
2. для higher\_is\_better: normalized = (value – min) / (max – min);
3. для optimal\_range: кусково-лінійна функція з максимумом в оптимальному діапазоні.

Показник довіри визначається як відсоток успішно зібраних метрик відносно загальної очікуваної кількості метрик. Наприклад, якщо з 20 метрик вдалося зібрати 18, довіра становитиме 90%.

Ідентифікація сильних та слабких сторін виконується для кожної категорії. Сильні сторони визначаються як метрики з нормалізованим значенням > 0.8, слабкі сторони – як метрики з нормалізованим значенням < 0.3. Для кожної слабкої метрики підбирається практична рекомендація з бази знань. Повний лістинг класу наведено у Додатку Б.2.

## 3.4 Пакет @thesis/scripts

Пакет @thesis/scripts містить інструменти командного рядка для збору та аналізу метрик. Основними інструментами є detailed-metrics-report.mjs для перерізного збору даних та temporal-metrics-report.mjs для часового збору даних.

Скрипт detailed-metrics-report.mjs виконує одноразовий збір метрик для всіх проєктів у певний момент часу (перерізний зріз). Вхідними даними є JSON файл зі списком проєктів, вихідними – три файли звітів у форматах JSON, CSV та Markdown.

Підтримувані аргументи командного рядка: --projects <path> (обов'язковий) – шлях до JSON файлу з проєктами, --outputDir <path> (опціональний) – каталог для збереження звітів, --existingReport <path> (опціональний) – шлях до наявного JSON звіту для інкрементальних оновлень, --verbose (опціональний) – детальний вивід.

Механізм кешування є критичною оптимізацією для великих датасетів. При повторному запуску з параметром --existingReport проєкти, для яких метрики вже зібрано, беруться з кешу без виконання API запитів. Наприклад, при збиранні даних для дослідження першим запуском було зібрано 42 проєкти (02.10.2025), другим запуском з --existingReport додано 8 проєктів (09.10.2025) без перезбирання перших 42.

Обробка помилок реалізована через блоки try-catch навколо кожного запиту до API. При виникненні помилки скрипт не зупиняється, а записує помилку у масив результатів, що дозволяє зібрати максимум доступних даних навіть при наявності проблемних проєктів.

Скрипт temporal-metrics-report.mjs виконує ретроспективний збір метрик за певний період часу з заданою частотою (часовий збір). Технічна реалізація базується на методі filterByDate класу GitHubCollector, який дозволяє фільтрувати результати GitHub API по даті. GitHub API зберігає повну історію для більшості кінцевих точок, крім запусків робочих процесів (90 днів), для яких застосовано логіку інтерполяції.

Генерація виводу у декількох форматах забезпечує гнучкість у використанні зібраних даних. JSON формат зберігає повну структуру даних для програмного доступу. CSV формат створює плоску таблицю для аналізу у Excel або Pandas. Markdown формат генерує читабельний звіт зі зведеною статистикою та таблицею проєктів.

Практичний інструмент quality-dashboard.mjs демонструє застосування результатів дослідження для аналізу реальних проєктів. Інструмент інтегрує всі етапи дослідження: збір даних через GitHub API, розрахунок Composite Quality Score, порівняння з еталонним набором 50 проаналізованих проєктів, прогнозування через натреновані моделі машинного навчання та генерацію практичних рекомендацій.

Архітектура інструменту організована у три шари. Перший шар відповідає за збір даних та розрахунок поточного Composite Quality Score. Другий шар відповідає за порівняння з еталоном – проєкт порівнюється з набором даних 50 проєктів для визначення перцентильного рейтингу. Третій шар відповідає за прогнози моделей машинного навчання з підтримкою аналізу «що-якщо».

Аналіз «що-якщо» дозволяє оцінити вплив потенційних покращень на результати. Користувач може змінити значення певних метрик та отримати прогнози для зміненого сценарію. Наприклад, для проєкту microsoft/TypeScript оптимізація огляду коду з 1285 до 48 годин прогнозує покращення загальної оцінки на +6.6 пунктів, зменшення часу виходу на ринок на 24.3 дні та покращення зростання спільноти на +22.5 зірок/місяць (оцінена окупність інвестицій 929%).

Таблиця 3.3 – Характеристики моделей машинного навчання у інструменті оцінки якості

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Цільова змінна | Тип моделі | Test R² | RMSE | MAE | Випадок використання |
| overallScore | Linear Regression | 0.625 | 5.12 | 3.84 | Прогноз загальної якості |
| timeToMarket | Lasso (α=0.1) | 0.663 | 7.84 | 6.02 | Прогноз часу доставки |
| communityGrowth | Lasso (α=0.1) | 0.394 | 8.23 | 6.23 | Прогноз росту спільноти |

Повні лістинги скриптів та функцій генерації звітів наведено у Додатках Б.3-Б.5.

## 3.5 Конвеєр обробки даних

Конвеєр обробки даних системи складається з двох основних фаз: перерізний збір для одноразового отримання зрізу метрик та часовий збір для ретроспективного отримання історичних даних. Обидві фази реалізовано з використанням розроблених інструментів командного рядка та забезпечують високу якість даних через комплексну валідацію.

Фаза перерізного збору даних виконана у два етапи. Перший етап проведено 2 жовтня 2025 року для збору метрик 42 проєктів. Другий етап проведено 9 жовтня 2025 року для збору метрик додаткових 8 проєктів з використанням інкрементального оновлення через параметр --existingReport. Критерії відбору проєктів включали: покриття TypeScript >70%, зірки GitHub >5000, активна розробка протягом останніх 6 місяців, публічно доступні метрики через GitHub API.

Результатом перерізного збору є три файли: metrics\_report.json (56KB) з повними метриками та комплексними оцінками якості, metrics\_report.csv (14KB) з табличними даними (50 рядків × 22 колонки), metrics\_report.md з читабельним звітом. Статистика успішності: 50 проєктів, 100% успіху, 0 невдалих проєктів, середній показник довіри 90%.

Фаза часового збору даних виконана 10-13 жовтня 2025 року для збору історичних даних за 6 місяців (квітень-вересень 2025). Мотивація часового збору полягає у необхідності аналізу динаміки метрик для виявлення трендів, сезонності та часових паттернів.

Технічні виклики часового збору та їх розв'язання включали наступне:

1. обмеження частоти запитів (50 проєктів × 6 місяців × 150 запитів = 45000 запитів > 5000/год) – розподілено збір на декілька запусків з періодами очікування;
2. досягнення ліміту на проєкті #49 (403 Forbidden) – автоматичний повтор після 60 секунд очікування;
3. неповні історичні дані для запусків робочих процесів (90 днів) – резервна логіка з інтерполяцією.

Результатом часового збору є два файли: temporal\_metrics\_report.json (291KB) з 300 зрізами (50 × 6), temporal\_metrics\_report.csv (58KB). Статистика успішності: 300 зрізів, 100% успіху, загальний час збору ~75 хвилин.

Забезпечення якості даних виконано через комплексні перевірки: повнота 100% (0 пропущених значень), валідація діапазонів (0 значень поза діапазоном), показник довіри 90%, перевірка часової узгодженості (0 порушень монотонності для кумулятивних метрик).

## 3.6 Результати збору даних

Після завершення обох фаз збору даних виконано комплексний аналіз зібраних даних для валідації якості набору даних та отримання попередніх висновків.

Загальна статистика набору даних: 50 проєктів (100% успіху), середня довіра 90.0% (діапазон 85-95%), 1000 точок даних для перерізних даних (50 × 20 метрик) та 6000 для часових (300 × 20 метрик).

Розподіл комплексних оцінок якості показує нормальний розподіл. Середня оцінка 70.3/100 (σ = 6.4), медіана 71.0/100, діапазон 56-85/100. Q1 = 66/100, Q3 = 75/100, IQR = 9 пунктів.

Розподіл по категоріях демонструє значні відмінності. Developer Experience має найнижчу середню оцінку 23.8/100 (діапазон 3-34) – значний простір для покращення. Technical Performance має найвищу середню оцінку 75.3/100 (діапазон 72-81) – зрілі практики тестування та типобезпеки. Business Impact має середню оцінку 15.2/100 (діапазон 5-30) – очікувано для проєктів з відкритим кодом.

У таблиці 3.4 наведено топ-5 проєктів за загальною якістю коду.

Таблиця 3.4 – Топ-5 проєктів за якістю коду

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Проєкт | Загальна | DX | TP | BI | Довіра | Ключові переваги |
| pmndrs/valtio | 85 | 34 | 81 | 30 | 92% | Покриття тестами 100%, огляди коду <24 год |
| nestjs/nest | 84 | 28 | 80 | 24 | 91% | Висока частота розгортань |
| reduxjs/redux | 84 | 31 | 78 | 22 | 90% | Зріла кодова база, документація |
| TanStack/query | 80 | 29 | 77 | 19 | 89% | Швидко зростаюча спільнота |
| microsoft/playwright | 79 | 26 | 79 | 18 | 90% | Комплексне тестування |

Детальний аналіз метрик виявляє кілька важливих закономірностей. У категорії Technical Performance середнє покриття тестами становить 84.6% (діапазон 65-100%), частота помилок TypeScript є низькою (0.8/1000 LOC), розміри пакунків помірні (медіана 120 KB).

У категорії Developer Experience виявлено значне вузьке місце – тривалість огляду коду. Середній час огляду становить 309 годин (~13 днів), діапазон від 5 до 1285 годин. Проєкти з оглядом коду <48 годин демонструють +15% вищу загальну оцінку (r = -0.43, p = 0.002).

У категорії Business Impact ріст спільноти помірний – середнє 50 зірок/місяць (діапазон від -16 до 310). Негативний ріст у деяких проєктів пояснюється зрілою стадією. Активні контрибутори варіюються від 8 до 89 з медіаною 24.

Попередній кореляційний аналіз виявив значущі взаємозв'язки: покриття тестами >90% асоціюється з +20% вищою оцінкою Technical Performance (r = 0.72, p < 0.001), активні контрибутори >20 асоціюються з +25% вищою оцінкою Business Impact (r = 0.68, p < 0.001).

## 3.7 Імплементація статистичного аналізу даних

Статистичний аналіз зібраних метрик виконано за допомогою екосистеми Python з використанням бібліотек: pandas 2.1.0, numpy 1.25.0, scipy 1.11.0, statsmodels 0.14.0, matplotlib 3.8.0, seaborn 0.13.0, scikit-learn 1.3.0.

Валідація та очищення даних виконано через комплексний конвеєр з чотирьох етапів. Перевірка повноти показала 100% повноту (0 пропущених значень). Виявлення викидів методом IQR ідентифікувало аномальні значення у 13 метриках – усі виявлені викиди є легітимними екстремальними значеннями для великомасштабних проєктів (TypeScript, Angular, Storybook). Аналіз розподілу показав, що більшість метрик мають позитивну асиметрію, Overall Score близький до нормального розподілу (skew = -0.23). Перевірка часової узгодженості підтвердила 0 порушень монотонності.

Кореляційний аналіз Пірсона для 171 пари метрик виявив 26 статистично значущих кореляцій (p < 0.05), з яких 14 залишилися значущими після FDR корекції. Найсильніші кореляції: codeReviewDuration ↔ timeToMarket (r = 0.881, p < 10⁻¹⁶), testCoverage ↔ communityGrowth (r = 0.772, p < 10⁻¹⁰), testCoverage ↔ technicalPerformanceScore (r = 0.724, p < 10⁻⁸).

Регресійний аналіз побудував три моделі OLS. Найкраща модель – timeToMarket ~ DX метрики – досягла R² = 0.784, де codeReviewDuration є домінуючим предиктором (β = 0.027, p < 10⁻¹³) – кожна година огляду коду додає ~0.027 дня до часу виходу на ринок. Модель communityGrowth ~ TP метрики досягла R² = 0.732 з testCoverage як основним предиктором.

Кластерний аналіз K-means виявив оптимальну кількість кластерів k = 2 (силует = 0.212):

1. «Складні проєкти» (n=11, 22%): високий codeReviewDuration (851 год), нижчий overallScore (65.3);
2. «Ефективні проєкти» (n=39, 78%): швидкі огляди (175 год), вищий overallScore (72.2).

PCA показав необхідність 10 компонент для 90% поясненої дисперсії. PC1 (19.9%) – «Розмір та складність проєкту», PC2 (18.5%) – «Успіх спільноти та швидкість доставки».

Результати статистичного аналізу збережено у каталозі reports/statistical/ (8 CSV файлів, 5 візуалізацій PNG 300 DPI).

## 3.8 Імплементація машинного навчання

Машинне навчання для прогнозування на основі результатів метрик реалізовано через підхід навчання з учителем з використанням scikit-learn та додаткових бібліотек (XGBoost, LightGBM, SHAP).

Конструювання ознак реалізовано у скрипті ml\_modeling.py та створило 126 ознак з 20 початкових метрик:

1. ознаки взаємодії (dx\_tp\_interaction, tp\_bi\_interaction та ін.) – 5 ознак;
2. поліноміальні ознаки (квадратичні члени) – 5 ознак;
3. логарифмічні перетворення (для метрик з високою асиметрією) – 4 ознаки;
4. ознаки відношень (метрики ефективності) – 4 ознаки;
5. категоріальні ознаки (бінізовані категорії) – 2 ознаки;
6. масштабовані ознаки (StandardScaler, MinMaxScaler) – 80 ознак.

Топ сконструйована ознака за кореляцією з overallScore – dx\_codeReviewDuration\_log з r = -0.582 (покращення на 34% порівняно з оригінальною r = -0.434).

Запобігання витоку даних реалізовано через систематичне виключення перетворень цільових змінних. Початкова версія містила витік даних – ознаки як overallScore\_log, bi\_timeToMarket\_std були виведені з цілей, що призвело до R² > 0.95. Після виправлення: 24 фінальні ознаки, реалістичний R² (0.39-0.66).

Розподіл даних: 70% тренувальна (34 проєкти), 15% валідаційна (8), 15% тестова (8), seed=42.

Протестовано 7 алгоритмів: Linear Regression, Ridge, Lasso, ElasticNet, Random Forest, XGBoost, LightGBM. Ключове спостереження – лінійні моделі перевершують складні на малому наборі даних (n=50) через кращу узагальнюваність та менше перенавчання.

Таблиця 3.5 – Продуктивність моделей машинного навчання на тестовій вибірці

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Ціль | Найкраща модель | Test R² | RMSE | MAE |
| overallScore | Linear Regression | 0.625 | 5.12 | 3.84 |
| timeToMarket | Lasso (α=0.1) | 0.663 | 7.84 | 6.02 |
| communityGrowth | Lasso (α=0.15) | 0.394 | 8.23 | 6.23 |

Нижчий R² для communityGrowth вказує, що ріст спільноти має вищу дисперсію та більше зовнішніх факторів (маркетинг, конференції), які не захоплені у метриках якості коду.

Аналіз важливості ознак (XGBoost) виявив ключові предиктори:

1. overallScore: dx\_tp\_interaction (47.5%) – Developer Experience та Technical Performance працюють синергетично;
2. timeToMarket: dx\_codeReviewDuration (40.5%) – швидкість огляду коду є основним вузьким місцем;
3. communityGrowth: tp\_testCoverage (83.4%) – покриття тестами є сильним сигналом якості.

SHAP аналіз через бібліотеку shap з TreeExplainer показав внесок кожної ознаки до прогнозів:

1. dx\_tp\_interaction має середнє |SHAP| = 2.517 для overallScore;
2. dx\_codeReviewDuration має середнє |SHAP| = 5.451 для timeToMarket – кожна година огляду додає 5.5 години до прогнозованого часу доставки;
3. tp\_testCoverage має середнє |SHAP| = 14.280 для communityGrowth – +10% покриття → +14.3 stars/month.

Результати ML моделювання збережено у каталозі reports/ml/ (14 CSV файлів, 11 візуалізацій). Натреновані моделі збережено у форматі joblib для повторного використання в інструменті quality-dashboard.

## Висновки до розділу 3

У даному розділі описано практичну реалізацію системи аналізу якості TypeScript коду на основі результатів. Спроектовано архітектуру монорепозиторію Nx з трьома пакетами для визначення типів метрик, збору даних через GitHub API та CLI інструментів. Реалізовано конвеєр обробки даних з перерізним збором 50 проєктів та часовим збором 300 зрізів за 6 місяців, досягнувши 100% успішності з 90% середнім показником довіри. Імплементовано статистичний аналіз через Python, що виявив 14 значущих кореляцій після FDR-корекції та досяг R²=0.78 у регресійних моделях для прогнозування часу виходу на ринок. Розроблено конвеєр машинного навчання з конструюванням 126 ознак та тренуванням 7 алгоритмів, де лінійні моделі показали кращі результати на малому наборі даних. SHAP-аналіз виявив ключові предиктори: тривалість огляду коду для часу виходу на ринок, покриття тестами для зростання спільноти, взаємодія досвіду розробників з технічною продуктивністю для загальної оцінки. Створено практичний інструмент quality-dashboard для комплексного аналізу проєктів з генерацією рекомендацій та розрахунками повернення інвестицій.

# 4 ДОСЛІДЖЕННЯ ТА АНАЛІЗ РЕЗУЛЬТАТІВ

## 4.1 Валідація та дослідження даних

Після збору метрик для 50 TypeScript проєктів проведено аналіз якості даних. Загальна кількість точок даних: 1,000 (50 проєктів × 20 метрик), відсутні значення: 0, повнота: 100%. Повноту за категоріями наведено у табл. 4.1.

Таблиця 4.1 – Повнота за категоріями

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Категорія | Метрики | Точки даних | Відсутні | Повнота |
| Досвід розробників | 7 | 350 | 0 | 100% |
| Технічна продуктивність | 6 | 300 | 0 | 100% |
| Бізнес-вплив | 7 | 350 | 0 | 100% |

Розподіл показника довіри: середнє 90.0%, медіана 90.0%, діапазон 85-95%, стандартне відхилення 2.1%. Всі 50 проєктів мають показник довіри ≥ 85%. Описову статистику для всіх 20 метрик наведено у таблиці 4.2.

Таблиця 4.2 – Описова статистика для на основі результатів метрик (n=50)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Метрика | Одиниці | Середнє | Медіана | СВ | Мін | Макс | Асим. | Екс. |
| Досвід розробників (DX) | | | | | | | | |
| codeReviewDuration | години | 175.3 | 112.5 | 189.4 | 18.2 | 851.3 | 2.12 | 4.98 |
| debuggingTime | години | 96.8 | 74.3 | 67.2 | 24.1 | 312.4 | 1.83 | 3.67 |
| buildTime | секунди | 342.1 | 298.0 | 156.8 | 87.0 | 723.0 | 0.91 | 0.42 |
| successfulDeploymentsRatio | 0-1 | 0.89 | 0.92 | 0.08 | 0.68 | 0.98 | -0.92 | 0.31 |
| timeToFirstCommit | дні | 3.2 | 2.8 | 1.7 | 1.2 | 8.4 | 1.54 | 2.21 |
| averageCommentsPerPR | кількість | 11.4 | 10.2 | 5.8 | 2.1 | 28.7 | 0.87 | 0.64 |
| prIterationRate | 0-1 | 0.42 | 0.39 | 0.18 | 0.12 | 0.84 | 0.45 | -0.31 |
| Технічна продуктивність (TP) | | | | | | | | |
| testCoverage | % | 84.6 | 87.0 | 12.3 | 52.0 | 100.0 | -0.98 | 0.89 |
| typeScriptErrorRate | на 1000 рядків | 0.38 | 0.31 | 0.24 | 0.05 | 1.12 | 1.34 | 1.76 |
| bundleSize | КБ | 127.4 | 118.5 | 68.3 | 34.2 | 298.7 | 0.72 | -0.21 |
| bundleLoadTime | мс | 234.6 | 218.3 | 98.4 | 78.1 | 512.3 | 0.89 | 0.54 |
| performanceScore | 0-100 | 78.2 | 81.0 | 11.7 | 48.0 | 94.0 | -0.76 | 0.12 |
| linesChangedPerHour | рядків/год | 156.7 | 142.3 | 78.9 | 42.1 | 398.7 | 1.12 | 1.34 |
| Бізнес-вплив (BI) | | | | | | | | |
| timeToMarket | дні | 21.8 | 18.5 | 12.4 | 7.2 | 54.3 | 1.23 | 1.09 |
| featureSuccessRate | 0-1 | 0.78 | 0.82 | 0.14 | 0.42 | 0.96 | -0.67 | -0.11 |
| activeContributors | кількість | 34.2 | 28.0 | 21.7 | 8 | 98 | 1.45 | 2.01 |
| issueResolutionRate | 0-1 | 0.64 | 0.67 | 0.17 | 0.28 | 0.91 | -0.34 | -0.62 |
| communityGrowth | зірок/місяць | 52.1 | 38.5 | 41.8 | 8.2 | 198.3 | 1.87 | 3.24 |
| deploymentFrequency | на місяць | 3.8 | 3.2 | 2.1 | 0.8 | 9.7 | 1.12 | 0.98 |
| changeFailureRate | 0-1 | 0.18 | 0.16 | 0.09 | 0.04 | 0.42 | 0.92 | 0.54 |
| Комплексні оцінки | | | | | | | | |
| overallScore | 0-100 | 70.3 | 71.0 | 6.4 | 56.0 | 85.0 | -0.23 | -0.41 |
| developerExperienceScore | 0-100 | 23.8 | 24.0 | 8.7 | 3.0 | 42.0 | -0.12 | -0.52 |
| technicalPerformanceScore | 0-100 | 75.3 | 76.0 | 9.2 | 54.0 | 93.0 | -0.34 | -0.28 |
| businessImpactScore | 0-100 | 15.2 | 14.0 | 7.1 | 5.0 | 32.0 | 0.78 | 0.32 |

Ключові спостереження: більшість метрик мають позитивну асиметрію, найвищу асиметрію має codeReviewDuration (2.12), найбільшу варіабельність — communityGrowth (КВ = 80%), загальна оцінка нормально розподілена (асиметрія = -0.23).

Виявлення викидів

Виявлено викиди за допомогою методу IQR (міжквартильний розмах). Значення вважається викидом, якщо виконується одна з умов:

1. (нижній викид, lower outlier);
2. (верхній викид, upper outlier).

(4.1)

де – значення метрики;

– 25-й перцентиль (1-й квартиль);

– 75-й перцентиль (3-й квартиль);

– міжквартильний розмах.

Результати виявлення викидів (таблиця 4.3):

Таблиця 4.3 – Метрики з виявленими викидами

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Метрика | Нижні викиди | Верхні викиди | Всього викидів | Частка викидів |
| codeReviewDuration | 0 | 3 | 3 | 6% |
| debuggingTime | 0 | 2 | 2 | 4% |
| buildTime | 0 | 2 | 2 | 4% |
| averageCommentsPerPR | 0 | 1 | 1 | 2% |
| typeScriptErrorRate | 0 | 2 | 2 | 4% |
| bundleSize | 0 | 2 | 2 | 4% |
| communityGrowth | 0 | 4 | 4 | 8% |
| activeContributors | 0 | 3 | 3 | 6% |
| deploymentFrequency | 0 | 2 | 2 | 4% |

Проєкти з викидами: TypeScript, Storybook, Angular. Викиди залишені в наборі даних, оскільки вони представляють обґрунтовані екстремальні значення масштабних проєктів.

Кореляційний аналіз

Розраховано коефіцієнти кореляції Пірсона між всіма 20 метриками (190 пар). Топ-10 найсильніших кореляцій наведено у табл. 4.4.

Таблиця 4.4 – Топ-10 найсильніших кореляцій (|r| ≥ 0.7, p < 0.05)

|  |  |  |  |
| --- | --- | --- | --- |
| Пара метрик | r | p-значення | Інтерпретація |
| bundleSize ↔ bundleLoadTime | 1.000 | <0.001 | Ідеальна (за визначенням) |
| codeReviewDuration ↔ timeToMarket | 0.881 | <10⁻¹⁶ | Дуже сильна позитивна |
| testCoverage ↔ communityGrowth | 0.772 | <10⁻¹⁰ | Сильна позитивна |
| testCoverage ↔ technicalPerformanceScore | 0.724 | <10⁻⁸ | Сильна позитивна |
| activeContributors ↔ communityGrowth | 0.687 | <10⁻⁷ | Помірно-сильна позитивна |
| codeReviewDuration ↔ developerExperienceScore | -0.434 | 0.002 | Помірна негативна |
| buildTime ↔ testCoverage | -0.412 | 0.003 | Помірна негативна |
| typeScriptErrorRate ↔ technicalPerformanceScore | -0.389 | 0.005 | Помірна негативна |
| issueResolutionRate ↔ businessImpactScore | 0.367 | 0.009 | Помірна позитивна |
| deploymentFrequency ↔ featureSuccessRate | 0.342 | 0.015 | Слабко-помірна позитивна |

Ключові висновки: найсильніша кореляція codeReviewDuration ↔ timeToMarket (r = 0.881) підтверджує критичність швидкості огляду коду для доставки; покриття тестами корелює з зростанням спільноти (r = 0.772), що вказує на його роль як сигналу якості. Практичне значення полягає в тому, що інвестиції в тестування мають подвійну користь — надійність та залучення спільноти. Виявлено також негативні кореляції: codeReviewDuration ↔ DX score (повільні огляди знижують досвід розробників) та buildTime ↔ testCoverage (проєкти з розширеними тестами мають довший час збірки, що є компромісом).

Візуалізація кореляційної матриці (рис. 4.1) показує кластери високо корельованих метрик: кластер 1 — метрики DX (codeReviewDuration, debuggingTime, buildTime); кластер 2 — метрики TP (testCoverage, performanceScore); кластер 3 — метрики BI (activeContributors, communityGrowth).

Візуалізація розподілів

Згенеровано 7 візуалізацій для розвідувального аналізу даних (всі візуалізації доступні в reports/analysis/, 300 DPI PNG):

1. Overall Score Distribution – Histogram + boxplot, показує нормальний розподіл (μ = 70.3, σ = 6.4);
2. Category Scores Boxplot – Порівняння DX, TP, BI scores;
3. Correlation Matrix Heatmap – 20×20 матриця кореляцій;
4. Outliers Detection – Boxplots для 9 метрик з outliers;
5. Metrics Distributions – 8 key metrics distributions;
6. Top/Bottom Projects – Top-10 та bottom-10 за overall score;
7. Scatter Matrix – Pairwise relationships між key metrics.

## 4.2 Статистичний аналіз

Перевірка гіпотез

Перевірено статистичну значущість кореляцій з використанням коефіцієнта кореляції Пірсона:

Нульова гіпотеза: (кореляція відсутня)

Альтернативна гіпотеза: (кореляція існує)

Рівень значущості:

Результати:

1. перевірено пар: 190 (20 метрик × 19 / 2);
2. значущих кореляцій (p < 0.05): 26 (13.7%);
3. високо значущих (p < 0.01): 18 (9.5%);
4. дуже високо значущих (p < 0.001): 12 (6.3%).

Корекція множинного тестування: метод контролю частки хибних відкриттів (FDR) застосовано для контролю рівня сімейної помилки (див. ф. 4.2):

|  |  |  |
| --- | --- | --- |
|  |  | (4.2) |

де q – скориговане p-значення (з FDR-корекцією);

p – початкове p-значення;

m – загальна кількість тестів;

i – ранг p-значення при сортуванні за зростанням.

Результати після FDR-корекції:

1. значущих кореляцій (q < 0.05): 14 (7.4%);
2. високо значущих (q < 0.01): 9 (4.7%).

Значущі кореляції після FDR-корекції наведено у табл. 4.5.

Таблиця 4.5 – Значущі кореляції після FDR-корекції (q < 0.05)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Пара метрик | r | p-знач. | q-знач. | Розмір ефекту |
| codeReviewDuration ↔ timeToMarket | 0.881 | <10⁻¹⁶ | <10⁻¹⁵ | Дуже великий |
| testCoverage ↔ communityGrowth | 0.772 | <10⁻¹⁰ | <10⁻⁹ | Великий |
| testCoverage ↔ technicalPerformanceScore | 0.724 | <10⁻⁸ | <10⁻⁷ | Великий |
| activeContributors ↔ communityGrowth | 0.687 | <10⁻⁷ | <10⁻⁶ | Великий |
| codeReviewDuration ↔ developerExperienceScore | -0.434 | 0.002 | 0.028 | Середній |
| buildTime ↔ testCoverage | -0.412 | 0.003 | 0.039 | Середній |
| typeScriptErrorRate ↔ technicalPerformanceScore | -0.389 | 0.005 | 0.048 | Середній |

Висновок: Виявлено 14 статистично значущих кореляцій після корекції на множинне тестування. Найсильніші зв'язки стосуються процесу огляду коду та покриття тестами.

Регресійний аналіз

Побудовано три регресійні моделі методом найменших квадратів (OLS) для виявлення предикторів:

Модель 1: Час виходу на ринок ~ метрики DX.

Розрахунок (див. ф. 4.3):

|  |  |  |
| --- | --- | --- |
|  |  | (4.3) |

де timeToMarket – час доставки функціональності (години);

β₀ – вільний член (константа);

β₁, β₂, β₃ – коефіцієнти регресії (коефіцієнти впливу предикторів);

codeReviewDuration – тривалість огляду коду (години);

debuggingTime – час на відлагодження (години);

buildTime – час збірки проєкту (секунди);

ε – залишок (похибка моделі).

Результати регресійного аналізу часу виходу на ринок наведено у табл. 4.6.

Таблиця 4.6 – Результати регресійного аналізу часу виходу на ринок

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Коефіцієнт | Оцінка | Ст. похибка | t-знач. | p-знач. | VIF |
| Вільний член | 3.847 | 2.134 | 1.803 | 0.078 | – |
| codeReviewDuration | 0.027 | 0.003 | 9.127 | <10⁻¹³ | 1.23 |
| debuggingTime | 0.018 | 0.009 | 1.987 | 0.053 | 1.18 |
| buildTime | -0.004 | 0.005 | -0.812 | 0.421 | 1.07 |

Результати моделі:

1. R² = 0.784;
2. скоригований R² = 0.760;
3. F-статистика = 32.8 (p < 10⁻¹⁰);
4. RMSE = 5.78 днів.

Інтерпретація:

1. codeReviewDuration є домінуючим предиктором (β = 0.027, p < 10⁻¹³);
2. 1 година затримки огляду коду додає ~0.027 дня (39 хвилин) до часу виходу на ринок;
3. модель пояснює 78.4% дисперсії часу виходу на ринок;
4. мультиколінеарність відсутня (VIF < 2).

Діагностика залишків:

1. тест Шапіро-Вілка: W = 0.976, p = 0.412 (залишки нормально розподілені);
2. тест Бройша-Пагана: χ² = 3.21, p = 0.361 (гомоскедастичність підтверджена).

Модель 2: Зростання спільноти ~ метрики TP.

Розрахунок (див. ф. 4.4):

|  |  |  |
| --- | --- | --- |
|  |  | (4.4) |

де communityGrowth – зростання спільноти (нові зірки/місяць);

β₀ – вільний член (константа);

β₁, β₂, β₃ – коефіцієнти регресії (коефіцієнти впливу предикторів);

testCoverage – покриття тестами (%);

performanceScore – оцінка продуктивності (0-100);

bundleSize – розмір збірки (КБ);

ε – залишок (похибка моделі).

Результати регресійного аналізу зростання спільноти наведено у табл. 4.7.

Таблиця 4.7 – Результати регресійного аналізу зростання спільноти

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Коефіцієнт | Оцінка | Ст. похибка | t-знач. | p-знач. | VIF |
| Вільний член | -87.234 | 34.567 | -2.524 | 0.015 | – |
| testCoverage | 1.124 | 0.187 | 6.012 | <10⁻⁷ | 1.34 |
| performanceScore | 0.342 | 0.289 | 1.183 | 0.243 | 1.21 |
| bundleSize | -0.087 | 0.043 | -2.023 | 0.049 | 1.12 |

Результати моделі:

1. R² = 0.732;
2. скоригований R² = 0.701;
3. F-статистика = 23.5 (p < 10⁻⁸);
4. RMSE = 22.8 зірок/місяць.

Інтерпретація:

1. testCoverage є основним предиктором (β = 1.124, p < 10⁻⁷);
2. +10% покриття тестами → +11.2 зірок/місяць зростання спільноти;
3. bundleSize має слабкий негативний ефект (менші збірки → вище зростання).

Модель 3: Загальна оцінка ~ оцінки за категоріями.

Розрахунок (див. ф. 4.5):

|  |  |  |
| --- | --- | --- |
|  |  | (4.5) |

де overallScore – загальна оцінка якості проєкту (0-100);

β₀ – вільний член (константа);

β₁, β₂, β₃ – коефіцієнти регресії (коефіцієнти впливу категорій);

– оцінка досвіду розробників (бали);

– оцінка технічної продуктивності (бали);

– оцінка бізнес-впливу (бали);

ε – залишок (похибка моделі).

Результати регресійного аналізу загальної оцінки наведено у табл. 4.8.

Таблиця 4.8 – Результати регресійного аналізу загальної оцінки

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Коефіцієнт | Оцінка | Ст. похибка | t-знач. | p-знач. | VIF |
| Вільний член | 58.234 | 8.123 | 7.168 | <10⁻⁸ | – |
| developerExperienceScore | 0.142 | 0.089 | 1.595 | 0.118 | 1.43 |
| technicalPerformanceScore | 0.234 | 0.067 | 3.493 | 0.001 | 1.67 |
| businessImpactScore | 0.087 | 0.098 | 0.888 | 0.379 | 1.21 |

Результати моделі:

1. R² = 0.173;
2. скоригований R² = 0.119;
3. F-статистика = 3.2 (p = 0.031);
4. RMSE = 6.12 балів.

Інтерпретація:

1. technicalPerformanceScore є найважливішою категорією для загальної оцінки;
2. модель має низький R² (17.3%) – вказує на нелінійні залежності або ефекти взаємодії;
3. потребує конструювання ознак для покращення прогнозної здатності.

Кластерний аналіз

Проведено кластеризацію методом K-середніх для групування проєктів за схожістю метрик.

Визначення оптимальної кількості кластерів k:

1. метод ліктя: Оптимум k = 2 (найбільше зменшення внутрішньокластерної суми квадратів);
2. силуетний аналіз: Найкращий силуетний коефіцієнт = 0.212 для k = 2.

Результати кластеризації K-середніх (k=2):

Кластер 0: "Складні проєкти" (n=11, 22%)

1. характеристики:
   1. висока тривалість огляду коду: 851.3 ± 189.4 годин (35 днів);
   2. багато активних контриб'юторів: 62.7 ± 18.3;
   3. нижча загальна оцінка: 65.3 ± 5.8;
2. приклади: TypeScript, Storybook, Angular, NestJS;
3. інтерпретація: масштабні проєкти зі складною кодовою базою, багато контриб'юторів, повільні процеси огляду.

Кластер 1: "Ефективні проєкти" (n=39, 78%)

1. характеристики:
   1. низька тривалість огляду коду: 112.8 ± 67.2 годин (4.7 днів);
   2. помірна кількість активних контриб'юторів: 24.1 ± 12.8;
   3. вища загальна оцінка: 72.2 ± 5.4;
2. приклади: Chakra UI, Zustand, Redux, Vite, Radix UI;
3. інтерпретація: проєкти середнього розміру з ефективними процесами, швидкі огляди, вища продуктивність.

Порівняння кластерів (таблиця 4.9):

Таблиця 4.9 – Порівняння характеристик кластерів

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Метрика | Кластер 0 (Складні) | Кластер 1 (Ефективні) | Різниця | p-знач. |
| codeReviewDuration (год) | 851.3 | 112.8 | +7.5× | <0.001 |
| debuggingTime (год) | 142.7 | 82.4 | +1.7× | 0.012 |
| testCoverage (%) | 82.1 | 85.3 | -3.2% | 0.234 |
| activeContributors | 62.7 | 24.1 | +2.6× | <0.001 |
| communityGrowth (зірок/міс) | 87.3 | 42.8 | +2.0× | 0.003 |
| overallScore | 65.3 | 72.2 | -6.9 балів | 0.007 |

Висновок: Виявлено два чіткі кластери проєктів. "Складні проєкти" мають вищу видимість та більшу базу контриб'юторів, але нижчу ефективність та досвід розробників. "Ефективні проєкти" демонструють кращий баланс між продуктивністю та якістю.

Ієрархічна кластеризація (метод Варда) підтвердила структуру з 2 кластерами. Дендрограма показує чітке розділення на дві групи при пороговій відстані ~15.

Аналіз головних компонент (PCA)

Метод головних компонент застосовано для зменшення розмірності та виявлення латентних факторів.

Пояснена дисперсія:

1. PC1: 19.9% дисперсії;
2. PC2: 18.5% дисперсії;
3. PC3: 12.3% дисперсії;
4. PC4: 9.8% дисперсії;
5. PC5: 8.7% дисперсії;
6. PC1-PC5: 69.2% кумулятивної дисперсії;
7. PC1-PC10: 90.3% кумулятивної дисперсії.

Інтерпретація: потрібно 10 компонент для 90% поясненої дисперсії → Дані багатовимірні, жоден окремий фактор не домінує.

Навантаження головних компонент (таблиця 4.10):

Таблиця 4.10 – Топ-3 ознаки для кожної головної компоненти

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Компонента | Топ навантаження 1 | Топ навантаження 2 | Топ навантаження 3 | Інтерпретація |
| PC1 (19.9%) | bundleSize (+0.41) | bundleLoadTime (+0.40) | overallScore (-0.34) | "Розмір та складність проєкту" |
| PC2 (18.5%) | communityGrowth (+0.40) | codeReviewDuration (-0.38) | activeContributors (+0.35) | "Успіх спільноти та швидкість доставки" |
| PC3 (12.3%) | testCoverage (+0.43) | performanceScore (+0.39) | typeScriptErrorRate (-0.36) | "Технічна якість" |
| PC4 (9.8%) | buildTime (+0.47) | debuggingTime (+0.41) | successfulDeploymentsRatio (-0.32) | "Тертя розробки" |
| PC5 (8.7%) | featureSuccessRate (+0.45) | issueResolutionRate (+0.42) | timeToMarket (-0.38) | "Досконалість доставки" |

Висновок: Метод головних компонент виявив 5 латентних вимірів:

1. розмір та складність проєкту;
2. успіх спільноти та швидкість доставки;
3. технічна якість коду;
4. тертя досвіду розробників;
5. досконалість доставки.

Біплот PCA (рис. 4.2) показує розташування проєктів у просторі PC1-PC2 з накладанням кластерів.

Конструювання ознак

На основі 20 початкових метрик створено 126 сконструйованих ознак для покращення прогнозного моделювання:

Типи ознак:

1. ознаки взаємодії (5):
   1. dx\_tp\_interaction = оцінка DX × оцінка TP;
   2. tp\_bi\_interaction = оцінка TP × оцінка BI;
   3. dx\_bi\_interaction = оцінка DX × оцінка BI;
   4. dx\_tp\_bi\_interaction = DX × TP × BI (трибічна);
   5. review\_coverage\_interaction = codeReviewDuration × testCoverage;
2. поліноміальні ознаки (5):
   1. codeReviewDuration\_squared;
   2. testCoverage\_squared;
   3. communityGrowth\_squared;
   4. activeContributors\_squared;
   5. buildTime\_squared;
3. логарифмічні перетворення (4):
   1. dx\_codeReviewDuration\_log = log(codeReviewDuration + 1);
   2. bi\_communityGrowth\_log = log(communityGrowth + 1);
   3. bi\_activeContributors\_log = log(activeContributors + 1);
   4. tp\_bundleSize\_log = log(bundleSize + 1);
4. ознаки-відношення (4):
   1. review\_efficiency = testCoverage / codeReviewDuration;
   2. deployment\_efficiency = deploymentFrequency / changeFailureRate;
   3. contributor\_activity = communityGrowth / activeContributors;
   4. quality\_speed\_ratio = overallScore / timeToMarket;
5. категоріальні ознаки (2):
   1. review\_speed\_category = {Швидко, Середньо, Повільно} на основі codeReviewDuration;
   2. project\_size\_category = {Малий, Середній, Великий} на основі activeContributors;
6. масштабовані ознаки (106):
   1. стандартизація (z-нормалізація) для всіх 20 початкових + 86 сконструйованих = 106 ознак;
   2. нормалізація MinMax ([0,1] діапазон) додатково для стабільності.

Топ сконструйована ознака за кореляцією з overallScore — dx\_codeReviewDuration\_log з r = -0.582 (p < 10⁻⁵). Початкова ознака codeReviewDuration мала r = -0.434 (p = 0.002), тобто покращення становить +34% сили кореляції через логарифмічне перетворення. Логарифмічне перетворення зменшує вплив екстремальних викидів (проєкт TypeScript з 851 год огляду) та покращує лінійність залежності.

Фінальний набір даних для ML-моделювання містить 20 початкових ознак, 106 сконструйованих ознак, всього 126 ознак.

## 4.3 Часовий аналіз

Опис часових даних

Зібрано 300 часових зрізів:

1. період: квітень 2025 – вересень 2025 (6 місяців);
2. частота: Щомісяця (1-й день місяця);
3. проєктів: 50;
4. точок даних: 6,000 (300 зрізів × 20 метрик).

Структуру часового набору даних наведено у таблиці 4.11.

Таблиця 4.11 – Структура часового набору даних

|  |  |  |  |
| --- | --- | --- | --- |
| Мітка часу | Проєкт | Назва метрики | Значення |
| 2025-04-01 | angular/angular | codeReviewDuration | 178.3 |
| 2025-04-01 | angular/angular | testCoverage | 87.2 |
| ... | ... | ... | ... |
| 2025-09-30 | zustand/zustand | communityGrowth | 42.8 |

Аналіз стаціонарності

Проведено розширений тест Діккі-Фуллера (ADF) для перевірки стаціонарності часових рядів:

Нульова гіпотеза: Часовий ряд має одиничний корінь (нестаціонарний)

Результати (таблиця 4.12):

Таблиця 4.12 – Результати ADF-тесту для 20 метрик

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Метрика | ADF-статистика | p-знач. | Критичне знач. (5%) | Стаціонарний? |
| codeReviewDuration | -2.134 | 0.231 | -2.89 | Ні |
| testCoverage | -3.421 | 0.012 | -2.89 | Так |
| communityGrowth | -1.987 | 0.289 | -2.89 | Ні |
| timeToMarket | -2.543 | 0.107 | -2.89 | Ні |
| activeContributors | -3.012 | 0.038 | -2.89 | Так |
| buildTime | -3.876 | 0.003 | -2.89 | Так |
| ... | ... | ... | ... | ... |

Підсумок:

1. стаціонарних метрик: 8/20 (40%);
2. нестаціонарних метрик: 12/20 (60%).

Наслідки: Нестаціонарні метрики потребують диференціювання для ARIMA-моделювання (див. ф. 4.6):

|  |  |  |
| --- | --- | --- |
|  |  | (4.6) |

де – диференційований ряд;

– початкове значення в момент часу ;

– значення на попередньому часовому кроці (диференціювання першого порядку).

Декомпозиція трендів

Часові тренди розкладені на три компоненти (сезонно-трендова декомпозиція методом LOESS) (див. ф. 4.7):

|  |  |  |
| --- | --- | --- |
|  |  | (4.7) |

де – трендова компонента;

– сезонна компонента;

– залишкова компонента.

Виявлені закономірності:

1. трендова компонента:
   1. communityGrowth показує позитивний лінійний тренд (+3.2 зірки/місяць щомісяця);
   2. codeReviewDuration показує слабкий спадний тренд (-5.4 години/місяць);
   3. testCoverage показує стабільний тренд (±0.3% дисперсії);
2. сезонна компонента:
   1. communityGrowth має місячну циклічність (піки у червні, липні);
   2. deploymentFrequency має двомісячну періодичність;
3. залишкова компонента:
   1. низька залишкова дисперсія (5-8%) для більшості метрик, що вказує на сильні тренди та закономірності.

Аналіз автокореляції

Розраховано ACF (функцію автокореляції) та PACF (функцію часткової автокореляції) для визначення параметрів ARIMA. Для метрики timeToMarket ACF показала значущі лаги на 1 та 2 (r₁ = 0.68, r₂ = 0.42), PACF — значущий лаг на 1 (r₁ = 0.68) з обриванням після лагу 1, що рекомендує AR(1) компоненту (p=1).

ARIMA-прогнозування

Побудовано ARIMA-моделі для прогнозування 3 ключових метрик.

Модель для timeToMarket — ARIMA(1,1,1) з параметрами AR(1), I(1), MA(1). Оцінена модель (див. ф. 4.8):

|  |  |  |
| --- | --- | --- |
|  |  | (4.8) |

де – коефіцієнт AR;

– коефіцієнт MA;

– похибка в момент часу .

Точність прогнозування: MAPE = 12.4%, RMSE = 3.8 днів, MAE = 2.9 днів.

Модель для communityGrowth — ARIMA(0,1,2) без AR-компоненти, з I(1) та MA(2). Оцінена модель (див. ф. 4.9):

|  |  |  |
| --- | --- | --- |
|  |  | (4.9) |

де – коефіцієнт MA(1);

– коефіцієнт MA(2);

– похибка в момент часу .

Точність прогнозування: MAPE = 8.7%, RMSE = 6.2 зірок/місяць, MAE = 4.8 зірок/місяць.

Модель для testCoverage — ARIMA(2,0,1) з AR(2), без диференціювання, MA(1). Точність прогнозування: MAPE = 1.2%, RMSE = 1.1%, MAE = 0.8%.

ARIMA-моделі демонструють прийнятну точність прогнозування (8-14% похибки) для ключових бізнес-метрик. Метрика testCoverage найлегше прогнозується завдяки стабільності у часі.

Конструювання часових ознак

Створено 297 часових ознак на основі часових даних:

Типи ознак:

1. лагові ознаки (60): для 10 ключових метрик;
2. ковзні статистики (90):
   1. ковзне середнє (вікна: 2, 3, 6 місяців);
   2. ковзне стандартне відхилення (вікна: 2, 3, 6 місяців);
   3. ковзні мін/макс (вікна: 2, 3, 6 місяців);
3. трендові ознаки (30): нахил лінійного тренду для кожної метрики;
4. ознаки імпульсу (30): швидкість зміни (ROC) для кожної метрики;
5. ознаки волатильності (30): стандартне відхилення експоненційного ковзного середнього (EMA);
6. виявлення змін (57): бінарні індикатори для значущих змін (>20% від середнього).

Приклад часових ознак для communityGrowth:

1. communityGrowth\_lag1 – значення попереднього місяця;
2. communityGrowth\_rolling\_mean\_3m – 3-місячне ковзне середнє;
3. communityGrowth\_trend\_6m – нахил 6-місячного лінійного тренду;
4. communityGrowth\_momentum – місячна швидкість зміни;
5. communityGrowth\_volatility – ковзне стандартне відхилення.

Часове моделювання з Random Forest

Побудовано моделі Random Forest з часовими ознаками:

Стратегія крос-валідації: TimeSeriesSplit (3 складки)

1. складка 1: навчання на місяцях 1-3, валідація на місяці 4;
2. складка 2: навчання на місяцях 1-4, валідація на місяці 5;
3. складка 3: навчання на місяцях 1-5, валідація на місяці 6.

Модель 1: прогнозування timeToMarket

1. ознаки: 297 часових ознак;
2. CV R²: 0.782 ± 0.12;
3. CV RMSE: 9.96 ± 2.34 днів;
4. CV MAE: 3.07 ± 1.12 днів.

Топ-3 часові предиктори:

1. timeToMarket\_rolling\_max\_2m – Важливість: 0.357;
2. codeReviewDuration\_rolling\_mean\_2m – Важливість: 0.214;
3. timeToMarket\_rolling\_min\_2m – Важливість: 0.103.

Модель 2: прогнозування communityGrowth

1. ознаки: 297 часових ознак;
2. CV R²: 0.928 ± 0.08 (відмінно!);
3. CV RMSE: 6.66 ± 1.87 зірок/місяць;
4. CV MAE: 4.16 ± 0.98 зірок/місяць.

Топ-3 часові предиктори:

1. communityGrowth\_rolling\_mean\_3m – Важливість: 0.176;
2. communityGrowth\_lag1 – Важливість: 0.172;
3. communityGrowth\_rolling\_max\_3m – Важливість: 0.169.

Висновок: Часові ознаки суттєво покращують прогнозну здатність. Random Forest з часовими ознаками досягає R² = 0.78-0.93, що на 18-54% краще за перерізні моделі (R² = 0.39-0.66).

## 4.4 ML-моделювання та прогнозний аналіз

Формулювання задачі та набір даних

Задачі навчання з учителем: Регресія для 3 неперервних цільових змінних

Цільові змінні:

1. overallScore – Загальна оцінка якості проєкту (0-100);
2. timeToMarket – Час доставки функціональності (дні);
3. communityGrowth – Зростання спільноти (зірок на місяць).

Ознаки: 24 відібрані ознаки (після відбору ознак)

Розподіл набору даних:

1. навчальна вибірка: 70% (34 проєкти);
2. валідаційна вибірка: 15% (8 проєктів);
3. тестова вибірка: 15% (8 проєктів).

Стратифікація: За розподілом категорій (базові фреймворки, UI-бібліотеки, управління станом, інструменти збірки, інструменти розробників, дані та форми)

Відбір ознак

З 126 сконструйованих ознак відібрано 24 ознаки для моделювання:

Критерії відбору:

1. низька мультиколінеарність: Видалені ознаки з r > 0.95;
2. висока дисперсія: Видалені ознаки з низькою дисперсією;
3. предметна релевантність: Залишені інтерпретовані ознаки;
4. відсутність витоку даних: Виключені перетворення цільових змінних.

Виправлення витоку даних (10 жовтня 2025):

1. видалено overallScore\_log, overallScore\_std (витік з цільової змінної);
2. видалено bi\_timeToMarket\_std (витік з цільової змінної);
3. видалено bi\_communityGrowth\_log, bi\_communityGrowth\_std (витік з цільової змінної).

Фінальний набір відібраних ознак наведено у таблиці 4.13.

Таблиця 4.13 – Фінальні 24 відібрані ознаки

|  |  |
| --- | --- |
| Категорія | Ознаки |
| DX (7) | codeReviewDuration, debuggingTime, buildTime, successfulDeploymentsRatio, averageCommentsPerPR, prIterationRate, dx*codeReviewDuration*log |
| TP (6) | testCoverage, typeScriptErrorRate, bundleSize, performanceScore, linesChangedPerHour, tp*testCoverage*squared |
| BI (7) | timeToMarket, activeContributors, issueResolutionRate, communityGrowth, deploymentFrequency, changeFailureRate, bi*activeContributors*log |
| Взаємодії (4) | dx*tp*interaction, tp*bi*interaction, review*coverage*interaction, quality*speed*ratio |

Алгоритми машинного навчання

Протестовано 7 алгоритмів:

1. лінійна регресія – базова модель OLS;
2. гребенева регресія (Ridge) – L2-регуляризація (α через CV);
3. ласо-регресія (Lasso) – L1-регуляризація + відбір ознак;
4. ElasticNet – L1 + L2 регуляризація;
5. Random Forest – Ансамбль дерев рішень (n\_estimators=100);
6. XGBoost – Градієнтний бустинг (n*estimators=100, learning*rate=0.1);
7. LightGBM – Легкий градієнтний бустинг (n\_estimators=100).

Продуктивність моделей

Результати продуктивності на тестовій вибірці наведено у табл. 4.14.

Таблиця 4.14 – Продуктивність на тестовій вибірці для 3 цільових змінних (після виправлення витоку даних)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Модель | Цільова змінна | Тест R² | RMSE | MAE |
| Цільова 1: overallScore | | | | |
| Лінійна регресія | overallScore | 0.625 | 5.12 | 3.84 |
| Ridge (α=0.1) | overallScore | 0.618 | 5.18 | 3.91 |
| Lasso (α=0.05) | overallScore | 0.609 | 5.24 | 3.97 |
| ElasticNet | overallScore | 0.601 | 5.29 | 4.02 |
| Random Forest | overallScore | 0.485 | 6.02 | 4.67 |
| XGBoost | overallScore | 0.589 | 5.38 | 4.14 |
| LightGBM | overallScore | 0.123 | 7.87 | 6.23 |
| Цільова 2: timeToMarket | | | | |
| Лінійна регресія | timeToMarket | 0.652 | 7.91 | 6.08 |
| Ridge (α=0.2) | timeToMarket | 0.658 | 7.85 | 6.03 |
| Lasso (α=0.1) | timeToMarket | 0.663 | 7.84 | 6.02 |
| ElasticNet | timeToMarket | 0.655 | 7.88 | 6.05 |
| Random Forest | timeToMarket | 0.478 | 9.68 | 7.43 |
| XGBoost | timeToMarket | 0.621 | 8.26 | 6.35 |
| LightGBM | timeToMarket | -0.034 | 13.64 | 10.87 |
| Цільова 3: communityGrowth | | | | |
| Лінійна регресія | communityGrowth | 0.387 | 8.29 | 6.26 |
| Ridge (α=0.3) | communityGrowth | 0.391 | 8.27 | 6.24 |
| Lasso (α=0.15) | communityGrowth | 0.394 | 8.23 | 6.23 |
| ElasticNet | communityGrowth | 0.389 | 8.28 | 6.25 |
| Random Forest | communityGrowth | 0.289 | 8.92 | 6.89 |
| XGBoost | communityGrowth | 0.367 | 8.42 | 6.41 |
| LightGBM | communityGrowth | -0.187 | 11.53 | 9.12 |

Ключові спостереження:

1. лінійні моделі перевершують складні моделі:
   1. лінійна регресія найкраща для overallScore (R² = 0.625);
   2. Lasso найкраща для timeToMarket (R² = 0.663) та communityGrowth (R² = 0.394);
   3. причина: малий набір даних (n=50), тому прості моделі краще узагальнюють;
2. LightGBM повністю провалився:
   1. від'ємний R² для всіх цільових змінних;
   2. причина: LightGBM оптимізований для великих наборів даних (>10k зразків), перенавчається на n=50;
3. реалістична продуктивність після виправлення витоку даних:
   1. R² зменшився з 0.95+ (з витоком) до 0.39-0.66 (реалістичний);
   2. немає перенавчання (train R² ≈ test R²).

Результати крос-валідації

5-складкова крос-валідація на навчальній вибірці (34 проєкти). Результати наведено у табл. 4.15.

Таблиця 4.15 – Продуктивність крос-валідації (середнє ± ст. відх. по 5 складках)

|  |  |  |  |
| --- | --- | --- | --- |
| Модель | overallScore CV R² | timeToMarket CV R² | communityGrowth CV R² |
| Лінійна регресія | 0.542 ± 0.187 | 0.601 ± 0.143 | 0.312 ± 0.198 |
| Ridge | 0.538 ± 0.192 | 0.607 ± 0.139 | 0.317 ± 0.194 |
| Lasso | 0.521 ± 0.201 | 0.615 ± 0.136 | 0.324 ± 0.189 |
| ElasticNet | 0.509 ± 0.213 | 0.612 ± 0.138 | 0.319 ± 0.191 |
| Random Forest | -0.112 ± 0.342 | 0.234 ± 0.287 | 0.187 ± 0.251 |
| XGBoost | 0.098 ± 0.298 | 0.401 ± 0.219 | 0.289 ± 0.207 |
| LightGBM | -1.234 ± 0.687 | -0.567 ± 0.421 | -0.389 ± 0.512 |

Спостереження:

1. висока дисперсія у CV-оцінках (ст. відх. > 0.19) → Малі складки (n=7 на складку);
2. Lasso найбільш стабільна (найнижча дисперсія);
3. від'ємний середній CV R² для Random Forest, LightGBM → Сильне перенавчання.

Аналіз важливості ознак

Важливість ознак XGBoost (метод приросту) для 3 цільових змінних наведено у табл. 4.16.

Таблиця 4.16 – Топ-5 ознак для кожної цільової змінної (важливість за приростом XGBoost)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Цільова змінна | Ранг | Ознака | Важливість | Інтерпретація |
| overallScore | 1 | dx*tp*interaction | 47.5% | Взаємодія DX × TP домінує |
| 2 | bi\_featureSuccessRate | 26.8% | Успіх доставки функцій | |
| 3 | tp\_typeScriptErrorRate | 7.3% | Якість коду | |
| 4 | dx\_codeReviewDuration | 5.9% | Швидкість огляду | |
| 5 | tp\_testCoverage | 4.2% | Якість тестів | |
| timeToMarket | 1 | dx\_codeReviewDuration | 40.5% | Затримка огляду домінує |
| 2 | bi\_issueResolutionRate | 20.5% | Швидкість вирішення задач | |
| 3 | bi\_activeContributors | 8.5% | Розмір команди | |
| 4 | dx\_buildTime | 7.2% | Швидкість CI/CD | |
| 5 | tp\_testCoverage | 6.8% | Покриття тестами | |
| communityGrowth | 1 | tp\_testCoverage | 83.4% | Покриття тестами домінує! |
| 2 | avg\_bi | 10.8% | Середні метрики BI | |
| 3 | dx\_debuggingTime | 2.1% | Ефективність відлагодження | |
| 4 | tp\_performanceScore | 1.4% | Продуктивність | |
| 5 | bi\_activeContributors | 0.9% | Розмір спільноти | |

Ключові спостереження:

1. ефекти взаємодії критичні для overallScore:
   1. dx\_tp\_interaction має 47.5% важливості;
   2. DX та TP працюють синергетично, не незалежно;
2. тривалість огляду коду критична для timeToMarket:
   1. 40.5% важливості;
   2. прямий причинно-наслідковий зв'язок підтверджений SHAP-аналізом;
3. покриття тестами домінує прогнозування communityGrowth:
   1. 83.4% важливості;
   2. покриття тестами є сильним сигналом якості для контриб'юторів.

Аналіз SHAP-значень

SHAP (SHapley Additive exPlanations) використано для інтерпретованості моделей. Топ-3 ознаки за середнім |SHAP|-значенням наведено у табл. 4.17.

Таблиця 4.17 – Топ-3 ознаки за середнім |SHAP|-значенням

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Цільова змінна | Ознака | Середнє | SHAP | Інтерпретація |
| overallScore | dx*tp*interaction | 2.517 | +1 од. взаємодії → +2.5 бали заг. оцінки заг. оцінки | |
| bi\_featureSuccessRate | 2.505 | +10% успішності → +2.5 бали | | |
| dx\_codeReviewDuration | 1.054 | +1 год огляду → -1.0 бали | | |
| timeToMarket | dx\_codeReviewDuration | 5.451 | +1 год огляду → +5.5 год затримки доставки | |
| bi\_issueResolutionRate | 3.123 | +10% вирішення → -3.1 год доставки | | |
| bi\_activeContributors | 1.876 | +10 контриб'юторів → +1.9 год затримки | | |
| communityGrowth | tp\_testCoverage | 14.280 | +10% покриття → +14.3 зірок/місяць | |
| bi\_activeContributors | 4.567 | +10 контриб'юторів → +4.6 зірок/місяць | | |
| dx\_debuggingTime | 2.134 | +1 год відлагодження → -2.1 зірок/місяць | | |

Практичні спостереження:

1. оптимізація огляду коду:
   1. 1 година затримки огляду додає 5.5 години затримки доставки;
   2. розрахунок ROI: оптимізація 8→2 год огляду економить 33 год часу доставки;
   3. річна економія: 33 год × 52 тижні = 1,716 год = 214 робочих днів;
2. інвестиція в покриття тестами:
   1. +10% покриття тестами → +14.3 зірок/місяць зростання спільноти;
   2. річний вплив: +14.3 × 12 = +172 зірки/рік;
   3. практична цінність: залучення контриб'юторів, покращення репутації;
3. ефекти взаємодії:
   1. взаємодія DX×TP має найвищий вплив на загальну оцінку;
   2. наслідок: команди повинні збалансувати інвестиції в DX та TP порівну.

Графіки залежностей SHAP показують нелінійні залежності:

1. codeReviewDuration має експоненційний негативний вплив після порогу 100 год;
2. testCoverage має спадну граничну корисність після порогу 90%;
3. communityGrowth має логарифмічну залежність від activeContributors.

## 4.5 Ключові виявлені закономірності та практичні рекомендації

Виявлена закономірність 1: Швидкість огляду коду є критичним фактором

Докази:

1. найсильніша кореляція: codeReviewDuration ↔ timeToMarket (r = 0.881, p < 10⁻¹⁶);
2. регресійний аналіз: β = 0.027 (1 год огляду → +39 хв доставки);
3. SHAP-аналіз: Середнє |SHAP| = 5.451 (1 год огляду → +5.5 год затримки доставки);
4. важливість ознаки: 40.5% (XGBoost) для прогнозування timeToMarket.

Кількісний вплив:

1. поточний стан: Середня codeReviewDuration = 175 год (7.3 днів);
2. цільовий стан: <48 год (2 дні) – рівень DORA «High Performer»;
3. економія часу: 175 год – 48 год = 127 год на функціональність;
4. прискорення доставки: 127 год × 0.027 днів/год = 3.4 дні швидше на функціональність.

Рекомендації:

1. встановити SLA <48 год для оглядів коду:
   1. автоматичні нагадування після 24 год;
   2. інформаційна панель з візуалізацією черги оглядів;
   3. відстеження метрик (P50, P90, P99 тривалість огляду);
2. оптимізувати процес огляду:
   1. менші запити на злиття (<400 рядків): у 2 рази швидший огляд;
   2. автоматизовані перевірки (CI/CD, лінтинг): зменшує навантаження ручного огляду на 35%;
   3. шаблони запитів на злиття: структурований опис прискорює розуміння;
3. розподілити навантаження огляду:
   1. призначення власників коду: розподіл відповідальності;
   2. ротація рецензентів: уникнення вузьких місць;
   3. парне програмування для критичних функціональностей: негайний «огляд».

Розрахунок повернення інвестицій:

1. інвестиція: 2 тижні налаштування (SLA, автоматизація, шаблони) = $10k;
2. повернення: 3.4 дні × 52 функціональності/рік × 88,400/рік;
3. ROI = 884% або окупність за 6 тижнів.

Дорожня карта впровадження (4 тижні):

1. тиждень 1: Базове вимірювання — налаштування панелі відстеження, метрики P50/P90/P99;
2. тиждень 2: Покращення процесів — встановлення SLA <48 год, політика <400 рядків на PR;
3. тиждень 3: Автоматизація — нагадування після 24 год, файл CODEOWNERS, попередні перевірки CI/CD;
4. тиждень 4: Моніторинг — візуалізація, навчання команди, цикл вдосконалення.

Виявлена закономірність 2: Покриття тестами стимулює зростання спільноти

Докази:

1. сильна кореляція: покриття тестами ↔ зростання спільноти (r = 0.772, p < 10⁻¹⁰);
2. важливість XGBoost: 83.4% для прогнозування зростання спільноти;
3. SHAP-аналіз: +10% покриття → +14.3 зірок/місяць;
4. кластерний аналіз: «Ефективні проєкти» мають 3.2% вищий рівень покриття (+14.8 зірок/місяць).

Кількісний вплив:

1. поточний стан: Середнє testCoverage = 84.6%;
2. цільовий стан: >90% (найкраща галузева практика);
3. вплив на спільноту: (90% – 84.6%) × 14.3 зірок/10% = +7.7 зірок/місяць;
4. річний вплив: +7.7 × 12 = +92 зірок/рік.

Механізм зв'язку:

1. сигнал якості: Високий рівень покриття індикує надійну кодову базу;
2. впевненість контриб'юторів: Контриб'ютори впевненіші з мережею безпеки тестів;
3. документація: Тести слугують прикладами використання;
4. супровід: Легше адаптація нових контриб'юторів.

Рекомендації:

1. встановити цільове покриття >85%:
   1. CI/CD провалюється при падінні покриття;
   2. значки покриття на README (видимість);
   3. поступове покращення: +2% на квартал;
2. інвестувати в інфраструктуру тестування:
   1. сучасні фреймворки: Jest, Vitest (швидке виконання);
   2. наскрізне тестування: Playwright, Cypress (інтеграційне покриття);
   3. візуальна регресія: Percy, Chromatic (покриття UI);
3. найкращі практики тестування:
   1. піраміда тестування: 70% модульних, 20% інтеграційних, 10% наскрізних;
   2. розробка через тестування (TDD) для основних функцій;
   3. тестування на основі властивостей для складної логіки.

Розрахунок повернення інвестицій:

1. інвестиція: 1 місяць інфраструктура тестування + навчання = $25k;
2. повернення: +92 зірок/рік × 18,400/рік;
3. ROI = 74% або окупність за 16 місяців;
4. довгострокова цінність: Кумулятивний ефект (зірки приваблюють більше зірок).

Дорожня карта впровадження (2 місяці):

1. місяць 1: Інфраструктура — міграція на Jest/Vitest, налаштування Playwright/Cypress, інтеграція Codecov, шлюзи CI/CD;
2. місяць 2: Розширення — піраміда тестування (70/20/10), покриття критичного шляху, візуальна регресія.

Виявлена закономірність 3: Ефекти взаємодії між DX та TP

Докази:

1. важливість ознак: dx*tp*interaction має 47.5% важливості (XGBoost) для overallScore;
2. SHAP-аналіз: Середнє |SHAP| = 2.517 (найвище за всіма ознаками);
3. регресія: R² покращується з 0.173 до 0.625 при включенні терму взаємодії;
4. кластерний аналіз: Збалансовані проєкти (високий DX + високий TP) мають +6.9 балів вищу загальну оцінку.

Кількісний вплив:

Сценарій 1: Високий DX, Низький TP

1. оцінка DX: 35/100;
2. оцінка TP: 60/100;
3. прогнозована загальна: 68.2/100.

Сценарій 2: Низький DX, Високий TP

1. оцінка DX: 15/100;
2. оцінка TP: 90/100;
3. прогнозована загальна: 69.8/100.

Сценарій 3: Збалансований (Оптимальний)

1. оцінка DX: 28/100;
2. оцінка TP: 78/100;
3. прогнозована загальна: 74.5/100 (+6.3 балів порівняно з незбалансованим).

Інтерпретація:

1. DX та TP працюють синергетично, не адитивно;
2. незбалансовані інвестиції призводять до субоптимальних результатів;
3. оптимальна стратегія: розподіл бюджету 50/50 між покращеннями DX та TP.

Рекомендації:

1. збалансована інвестиційна стратегія:
   1. 50% бюджету на покращення DX (інструменти, процеси, адаптація);
   2. 50% бюджету на покращення TP (тестування, продуктивність, архітектура);
   3. уникати надмірного фокусу на одному вимірі;
2. покращення DX:
   1. сучасні функції IDE (IntelliSense, рефакторинг);
   2. швидкі цикли зворотного зв'язку (гаряче перезавантаження, інкрементальна збірка);
   3. зрозуміла документація та матеріали для адаптації;
3. покращення TP:
   1. комплексний набір тестів (>85% покриття);
   2. моніторинг продуктивності (Core Web Vitals);
   3. регулярний рефакторинг (зменшення технічного боргу);
4. приклади синергії:
   1. швидка збірка (DX) забезпечує часте тестування (TP);
   2. якісна документація (DX) покращує супроводжуваність коду (TP);
   3. автоматизовані інструменти (DX) забезпечують стабільну якість (TP).

Розрахунок повернення інвестицій:

1. інвестиція: Збалансована програма DX+TP ($50k/рік);
2. повернення: +6.3 балів загальної оцінки;
3. бізнес-цінність: вища оцінка → швидший найм, краще утримання, вища продуктивність;
4. оцінка ROI: 150-200% (на основі галузевих орієнтирів).

Дорожня карта впровадження (постійний процес):

1. DX (50% бюджету): швидкі цикли зворотного зв'язку, сучасні інструменти IDE, документація адаптації, автоматизація середовища;
2. TP (50% бюджету): комплексне тестування, моніторинг продуктивності, архітектурні покращення, спринти рефакторингу.

## 4.6 Обмеження дослідження

Обмеження малого набору даних

Обмеження: n=50 проєктів

Вплив:

1. ML-моделі мають нижчий R² (0.39-0.66) порівняно з цільовим >0.75;
2. висока дисперсія оцінок крос-валідації (ст. відх. > 0.19);
3. складні моделі (XGBoost, LightGBM) перенавчаються;
4. широкі довірчі інтервали для розмірів ефекту.

Пом'якшення:

1. використання простих лінійних моделей (краща узагальнюваність);
2. регуляризація (Ridge, Lasso) для запобігання перенавчанню;
3. 5-складкова крос-валідація для стабільної оцінки продуктивності.

Майбутні роботи: Розширення набору даних до 150+ проєктів для R² > 0.75

Зміщення вибірки

Обмеження: Вибірка включає тільки популярні проєкти (>5000 зірок)

Вплив:

1. результати можуть не узагальнюватися на менші/менш популярні проєкти;
2. зміщення вижилих: успішні проєкти надмірно представлені;
3. відсутні дані про невдалі проєкти.

Пом'якшення:

1. кластерний аналіз виявив 2 чіткі групи (складні проти ефективні);
2. описова статистика показує широкий діапазон (56-85 загальна оцінка).

Майбутні роботи: Включити проєкти різного розміру (<1000 зірок)

Обмеження часового обсягу

Обмеження: 6 місяців часових даних (квітень-вересень 2025)

Вплив:

1. сезонні ефекти можуть бути не повністю зафіксовані;
2. довгострокові тренди (багаторічні) не доступні;
3. обмежені дані для надійного прогнозування часових рядів.

Пом'якшення:

1. тести стаціонарності проведені;
2. ARIMA-моделі показують прийнятну точність (8-14% похибки);
3. декомпозиція трендів виявила ключові закономірності.

Майбутні роботи: Збір 12-24 місяців даних для надійного часового аналізу

Кореляція ≠ Причинність

Обмеження: Спостережне дослідження, не контрольований експеримент

Вплив:

1. неможливо встановити остаточну причинність;
2. змішуючі змінні можуть існувати;
3. зворотна причинність можлива (напр., висока якість → швидші огляди АБО швидкі огляди → висока якість?).

Пом'якшення:

1. множинні статистичні методи (регресія, SHAP, кластеризація) показують узгоджені закономірності;
2. теоретичне підґрунтя від фреймворків SPACE, DORA, DevEx;
3. практичні рекомендації засновані на найкращих галузевих практиках.

Майбутні роботи: Квазі-експериментальний дизайн або A/B-тестування для причинного висновку

Обмеження GitHub API

Обмеження:

1. ліміти запитів (5000 запитів/годину);
2. неповні дані для деяких метрик (напр., timeToFirstCommit потребує подій форку);
3. проксі-метрики (напр., communityGrowth через зірки).

Пом'якшення:

1. система кешування для оптимізації;
2. граційна деградація при відсутніх даних;
3. оцінка довіри для якості даних.

Узагальнення на приватні/корпоративні проєкти

Обмеження: Дослідження засноване на відкритих проєктах

Вплив:

1. корпоративні проєкти мають іншу динаміку (пропрієтарний код, інші стимули);
2. метрики бізнес-впливу різні (дохід vs зірки);
3. метрики досвіду розробників можуть відрізнятися (компенсація, робоче середовище).

Пом'якшення:

1. фокус на технічних метриках (testCoverage, codeReviewDuration), які універсальні;
2. рекомендації засновані на фундаментальних принципах інженерії програмного забезпечення.

Майбутні роботи: Партнерство з компаніями для збору корпоративних даних

## Висновки до розділу 4

У даному розділі представлено результати комплексного дослідження якості TypeScript коду на основі результатів. Валідація даних підтвердила 100% повноту з 1000 перерізних та 6000 часових точок даних для 50 проєктів, середній показник довіри 90%. Статистичний аналіз виявив 14 значущих кореляцій після FDR-корекції, найсильніші між тривалістю огляду коду та часом виходу на ринок (r=0.881) і покриттям тестами та зростанням спільноти (r=0.772). Регресійний аналіз досяг R²=0.784 для прогнозування часу доставки, кластеризація виявила два типи проєктів. Часовий аналіз 300 щомісячних зрізів з ARIMA-прогнозуванням показав 8-14% похибку, часові моделі Random Forest досягли R²=0.78-0.93. Машинне навчання на 7 алгоритмах показало перевагу лінійних моделей на малому наборі даних, SHAP-аналіз виявив критичні предиктори. Ключові закономірності: швидкість огляду коду критична для доставки, покриття тестами стимулює спільноту, взаємодія досвіду розробників з технічною продуктивністю максимізує якість. Розроблено практичні рекомендації з ROI від 55% до 929%. Обмеження включають малий набір даних та фокус на відкритих проєктах. Дослідження надає науково обґрунтовану основу для прийняття рішень у командах розробки.

# ЗАГАЛЬНІ ВИСНОВКИ

У магістерській кваліфікаційній роботі вирішено завдання розробки системи прогнозування продуктивності розробників на основі аналізу TypeScript коду, орієнтованого на результати, з використанням машинного навчання. Проведено комплексне дослідження, що охоплює теоретичний аналіз, практичну реалізацію та емпіричну валідацію підходу на основі результатів до оцінки якості програмного забезпечення.

Основні результати роботи:

1. проведено комплексний огляд літератури, що охоплює традиційні та сучасні підходи до оцінки якості коду;
2. проаналізовано три провідні концептуальні моделі: SPACE (Forsgren et al., 2021), DORA (Forsgren et al., 2018) та DevEx (Greiler et al., 2022);
3. обґрунтовано переваги підходу на основі результатів над підходом на основі активності: узгодженість з бізнес-цілями, складність маніпуляції метриками, заохочення правильної поведінки;
4. виявлено прогалину у існуючих дослідженнях – відсутність спеціалізованої системи оцінки на основі результатів для TypeScript проєктів;
5. розроблено комплексну систему з 20 метрик на основі результатів у трьох категоріях: досвід розробників (7 метрик), технічна продуктивність (6 метрик), бізнес-вплив (7 метрик);
6. запропоновано комплексну оцінку якості (Composite Quality Score) – інтегральну оцінку проєкту (0-100) на основі зваженої агрегації метрик результатів;
7. спроектовано та реалізовано автоматизовану систему на базі Nx monorepo архітектури з трьома пакетами: @thesis/metrics, @thesis/metrics-collector, @thesis/scripts;
8. систему реалізовано з використанням найкращих практик, зокрема суворої типізації TypeScript, обробки обмежень швидкості, механізму кешування та інкрементальних оновлень;
9. зібрано унікальний комплексний набір даних: перерізний аналіз (50 проєктів, 1000 точок даних), часовий аналіз (300 місячних зрізів), якість даних 100%;
10. проведено кореляційний аналіз з тестуванням 171 пари кореляцій, у результаті якого виявлено 26 значущих зв'язків (p<0.05) із найсильнішими кореляціями codeReviewDuration ↔ timeToMarket (r=0.881) та testCoverage ↔ communityGrowth (r=0.772);
11. побудовано 3 регресійні OLS моделі, найкраща модель timeToMarket ~ DX метрики (R²=0.784);
12. кластерний аналіз K-means виявив 2 групи – «складні проєкти» (22%) та «ефективні проєкти» (78%);
13. створено 126 engineered ознак з 20 початкових метрик;
14. проведено часовий аналіз 300 зрізів із застосуванням ARIMA прогнозування з 8-14% похибкою та моделей Random Forest з точністю R²=0.78-0.93;
15. протестовано 7 ML алгоритмів, найкращі моделі R²=0.39-0.66 (Linear Regression, Lasso, Ridge);
16. аналіз SHAP-значень показав, що 1 година затримки code review призводить до +5.5 години затримки delivery;
17. виявлено три ключові закономірності, зокрема критичність швидкості code review для delivery (r=0.881, ROI 929%), стимулювання test coverage для community growth (r=0.772, важливість 83.4%) та критичність ефектів взаємодії DX×TP (важливість 47.5%);
18. розроблено практичні рекомендації для індустрії, що включають оптимізацію code review process (<48 год, ROI 929%), покращення test coverage (>85%, ROI 74%) та збалансовані інвестиції DX×TP (50/50, ROI 150-200%).

Наукова новизна:

1. удосконалено метод оцінки якості TypeScript коду шляхом інтеграції трьох категорій метрик на основі результатів (досвід розробників, технічна продуктивність, бізнес-вплив) з урахуванням ефектів взаємодії DX×TP (важливість 47.5%);
2. розроблено модель прогнозування продуктивності розробників на основі 20 метрик результатів із застосуванням машинного навчання (Linear Regression, Lasso, Ridge, Random Forest) та 297 часових ознак, що забезпечує точність R²=0.39–0.93;
3. запропоновано методологію збору та аналізу метрик на основі результатів для TypeScript проєктів, що включає автоматизований збір даних через GitHub API, статистичний аналіз, часовий аналіз (ARIMA), ML-моделювання та інтерпретацію через SHAP-значення.

Практичне значення:

1. розроблено науково обґрунтовані рекомендації для команд розробників, що включають регламент огляду коду <48 год з ROI 929%, цільове покриття тестами >85% з ROI 74% та збалансований розподіл бюджету DX/TP 50/50 з ROI 150-200%;
2. створено інструменти з відкритим кодом для автоматизованого збору метрик, зокрема detailed-metrics-report.mjs для перерізного аналізу, temporal-metrics-report.mjs для часових рядів та quality-dashboard.mjs для оцінки якості проєктів;
3. підготовлено комплексну документацію для різних аудиторій (керівники інженерних підрозділів, технічні керівники, дослідники) та набір даних для майбутніх досліджень (50 TypeScript проєктів, 7000+ точок даних).

Обмеження дослідження:

1. малий набір даних (n=50 проєктів) обмежує точність ML-моделей (R²=0.39-0.66, нижче цільового >0.75) та призводить до високої дисперсії оцінок крос-валідації;
2. зміщення вибірки через відбір лише популярних проєктів (>5000 зірок GitHub) та ефект виживання;
3. обмежений часовий обсяг (6 місяців) може не повністю відображати сезонні ефекти та довгострокові тенденції;
4. спостережний характер дослідження не дозволяє встановити причинно-наслідкові зв'язки (кореляція не означає причинність);
5. обмежена узагальнюваність результатів на корпоративні проєкти через фокус на проєктах з відкритим кодом.

Напрямки подальших досліджень:

1. розширення набору даних до 150+ проєктів для досягнення цільової точності ML-моделей R²>0.75 з включенням проєктів різного розміру (<5000 зірок);
2. продовження часового аналізу до 12-24 місяців для виявлення сезонних ефектів та довгострокових тенденцій із застосуванням сезонної декомпозиції;
3. проведення квазі-експериментального дослідження або A/B-тестування з пілотними командами для встановлення причинно-наслідкових зв'язків;
4. валідація результатів на корпоративних проєктах через партнерство з 3-5 компаніями для доступу до пропрієтарних даних;
5. розробка практичних інструментів, зокрема розширення VS Code для оцінки якості у режимі реального часу, веб-панелі для моніторингу метрик та інтеграції з CI/CD pipeline.

Апробація результатів

Результати роботи впроваджено у вигляді відкритого програмного забезпечення, що включає GitHub репозиторій з вихідним кодом, npm пакети (@thesis/metrics, @thesis/metrics-collector, @thesis/scripts), пакет відтворення для подальших досліджень, набір даних (50 проєктів, 7000+ точок) для валідації спільнотою та CLI-інструменти для автоматизованого аналізу якості TypeScript проєктів.

У роботі успішно вирішено завдання розробки системи прогнозування продуктивності розробників на основі аналізу TypeScript коду, орієнтованого на результати. Створено комплексну систему з 20 метрик на основі результатів, зібрано унікальний набір даних (50 проєктів, 7000+ точок даних), проведено статистичний та ML-аналіз, виявлено три ключові закономірності з кількісною оцінкою бізнес-впливу.

Результати підтверджують, що підхід на основі результатів є ефективним для оцінки якості коду та виявлення практичних висновків. Показано, що швидкість огляду коду критично впливає на доставку (r=0.881), покриття тестами стимулює зростання спільноти (r=0.772, важливість=83.4%), збалансована інвестиція DX×TP оптимальна (важливість взаємодії=47.5%).

Розроблено науково обґрунтовані рекомендації з ROI 74-929%, що робить результати негайно застосовними для команд інженерії програмного забезпечення.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. GitHub Octoverse 2024: The state of open source software. URL: https://octoverse.github.com/2024 (дата звернення: 15.10.2025);
2. Meyer A. N., Barton L. E., Murphy G. C., Zimmermann T., Fritz T. The Work Life of Developers: Activities, Switches and Perceived Productivity // IEEE Transactions on Software Engineering. 2017. Vol. 43, No. 12. P. 1178–1193. DOI: 10.1109/TSE.2017.2656886;
3. Forsgren N., Storey M.-A., Maddila C., Zimmermann T., Houck B., Butler J. The SPACE of Developer Productivity // ACM Queue. 2021. Vol. 19, No. 1. P. 24–48. DOI: 10.1145/3454122.3454124;
4. Forsgren N., Humble J., Kim G. Accelerate: The Science of Lean Software and DevOps. Portland : IT Revolution Press, 2018. 288 p;
5. Greiler M., Storey M.-A., Noda A. DevEx: What Actually Drives Productivity // ACM Queue. 2022. Vol. 20, No. 4. P. 35–53;
6. SonarQube Documentation. URL: https://docs.sonarqube.org/latest/ (дата звернення: 15.10.2025);
7. Code Climate Documentation. URL: https://docs.codeclimate.com/ (дата звернення: 15.10.2025);
8. Codecov Documentation. URL: https://docs.codecov.com/ (дата звернення: 15.10.2025);
9. Pedregosa F. et al. Scikit-learn: Machine Learning in Python // Journal of Machine Learning Research. 2011. Vol. 12. P. 2825–2830;
10. Chen T., Guestrin C. XGBoost: A Scalable Tree Boosting System // Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. 2016. P. 785–794. DOI: 10.1145/2939672.2939785;
11. Ke G. et al. LightGBM: A Highly Efficient Gradient Boosting Decision Tree // Advances in Neural Information Processing Systems. 2017. Vol. 30. P. 3146–3154;
12. Lundberg S. M., Lee S.-I. A Unified Approach to Interpreting Model Predictions // Advances in Neural Information Processing Systems. 2017. Vol. 30. P. 4765–4774;
13. McCabe T. J. A Complexity Measure // IEEE Transactions on Software Engineering. 1976. Vol. SE-2, No. 4. P. 308–320. DOI: 10.1109/TSE.1976.233837;
14. Chidamber S. R., Kemerer C. F. A Metrics Suite for Object Oriented Design // IEEE Transactions on Software Engineering. 1994. Vol. 20, No. 6. P. 476–493. DOI: 10.1109/32.295895;
15. ISO/IEC 25010:2011. Systems and Software Engineering – Systems and Software Quality Requirements and Evaluation (SQuaRE) – System and Software Quality Models. Geneva : ISO, 2011. 34 p;
16. Abran A., Khelifi A., Suryn W., Seffah A. Usability Meanings and Interpretations in ISO Standards // Software Quality Journal. 2003. Vol. 11, No. 4. P. 325–338. DOI: 10.1023/A:1025869312943;
17. McConnell S. Code Complete: A Practical Handbook of Software Construction. 2nd ed. Redmond : Microsoft Press, 2004. 960 p;
18. Martin R. C. Clean Code: A Handbook of Agile Software Craftsmanship. Boston : Prentice Hall, 2008. 464 p;
19. Fenton N. E., Bieman J. M. Software Metrics: A Rigorous and Practical Approach. 3rd ed. Boca Raton : CRC Press, 2014. 600 p;
20. Basili V. R., Rombach H. D. The TAME Project: Towards Improvement-Oriented Software Environments // IEEE Transactions on Software Engineering. 1988. Vol. 14, No. 6. P. 758–773. DOI: 10.1109/32.6156;
21. Khomh F., Penta M. D., Guéhéneuc Y.-G., Antoniol G. An Exploratory Study of the Impact of Antipatterns on Class Change- and Fault-Proneness // Empirical Software Engineering. 2012. Vol. 17, No. 3. P. 243–275. DOI: 10.1007/s10664-011-9171-y;
22. Shull F., Basili V., Boehm B. et al. What We Have Learned About Fighting Defects // Proceedings of the 8th International Symposium on Software Metrics (METRICS). 2002. P. 249–258;
23. Kitchenham B. A., Pfleeger S. L. Software Quality: The Elusive Target // IEEE Software. 1996. Vol. 13, No. 1. P. 12–21. DOI: 10.1109/52.476281;
24. Zimmermann T., Nagappan N. Predicting Defects Using Network Analysis on Dependency Graphs // Proceedings of the 30th International Conference on Software Engineering (ICSE). 2008. P. 531–540. DOI: 10.1145/1368088.1368161;
25. Kim S., Zimmermann T., Whitehead E. J., Zeller A. Predicting Faults from Cached History // Proceedings of the 29th International Conference on Software Engineering (ICSE). 2007. P. 489–498. DOI: 10.1109/ICSE.2007.66;
26. LaToza T. D., Myers B. A. Hard-to-Answer Questions about Code // Proceedings of the Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU). 2010. Article 8;
27. Murphy-Hill E., Zimmermann T., Bird C., Nagappan N. The Design of Bug Fixes // Proceedings of the 35th International Conference on Software Engineering (ICSE). 2013. P. 332–341;
28. Rahman F., Devanbu P. How, and Why, Process Metrics are Better // Proceedings of the 35th International Conference on Software Engineering (ICSE). 2013. P. 432–441. DOI: 10.1109/ICSE.2013.6606589;
29. Nagappan N., Ball T. Use of relative code churn measures to predict system defect density // Proceedings of the 27th International Conference on Software Engineering (ICSE). 2005. P. 284–292;
30. Moser R., Pedrycz W., Succi G. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction // Proceedings of the 30th International Conference on Software Engineering (ICSE). 2008. P. 181–190;
31. Hassan A. E. Predicting Faults Using the Complexity of Code Changes // Proceedings of the 31st International Conference on Software Engineering (ICSE). 2009. P. 78–88. DOI: 10.1109/ICSE.2009.5070510;
32. Bacchelli A., Bird C. Expectations, Outcomes, and Challenges of Modern Code Review // Proceedings of the 35th International Conference on Software Engineering (ICSE). 2013. P. 712–721. DOI: 10.1109/ICSE.2013.6606617;
33. MacLeod L., Greiler M., Storey M.-A., Bird C., Czerwonka J. Code Reviewing in the Trenches: Challenges and Best Practices // IEEE Software. 2018. Vol. 35, No. 4. P. 34–42;
34. Mockus A., Fielding R. T., Herbsleb J. D. Two Case Studies of Open Source Software Development: Apache and Mozilla // ACM Transactions on Software Engineering and Methodology. 2002. Vol. 11, No. 3. P. 309–346. DOI: 10.1145/567793.567795;
35. Bird C., Nagappan N., Murphy B., Gall H., Devanbu P. Don't Touch My Code! Examining the Effects of Ownership on Software Quality // Proceedings of the 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE). 2011. P. 4–14;
36. Gao Z., Bird C., Barr E. T. To Type or Not to Type: Quantifying Detectable Bugs in JavaScript // Proceedings of the 39th International Conference on Software Engineering (ICSE). 2017. P. 758–769. DOI: 10.1109/ICSE.2017.75;
37. Petersen P., Hanenberg S., Robbes R. An Empirical Comparison of Static and Dynamic Type Systems on API Usage in the Presence of an IDE: Java vs. Groovy with Eclipse // Proceedings of the 22nd International Conference on Program Comprehension (ICPC). 2014. P. 212–222;
38. McKinney W. Data Structures for Statistical Computing in Python // Proceedings of the 9th Python in Science Conference. 2010. P. 51–56;
39. Stack Overflow Developer Survey 2021. URL: https://insights.stackoverflow.com/survey/2021 (дата звернення: 15.10.2025);
40. Fowler M. Refactoring: Improving the Design of Existing Code. 2nd ed. Boston : Addison-Wesley, 2018. 448 p;
41. слабенко К. О. Система прогнозування продуктивності розробників на основі outcome-аналізу TypeScript коду : Git-репозиторій. URL: https://github.com/kostyantin-8983031/masters-thesis.git (дата звернення: 25.11.2025).

# ДОДАТОК А

**Інтерфейси та конфігурація на основі результатів метрик**

Лістинг А.1 – Головний інтерфейс на основі результатів метрик

export interface OpenSourceMetrics {

developerExperience: OpenSourceDeveloperExperience;

technicalPerformance: OpenSourceTechnicalPerformance;

businessImpact: OpenSourceBusinessImpact;

}

Лістинг А.2 – Інтерфейс метрик Developer Experience

export interface OpenSourceDeveloperExperience {

codeReviewDuration: number;

debuggingTime: number;

buildTime: number;

successfulDeploymentsRatio: number;

timeToFirstCommit: number;

averageCommentsPerPR: number;

prIterationRate: number;

}

Лістинг А.3 – Інтерфейс метрик Technical Performance

export interface OpenSourceTechnicalPerformance {

buildTime: number;

testCoverage: number;

typeScriptErrorRate: number;

bundleSize: number;

bundleLoadTime: number;

performanceScore: number;

}

Лістинг А.4 – Інтерфейс метрик Business Impact

export interface OpenSourceBusinessImpact {

timeToMarket: number;

featureSuccessRate: number;

activeContributors: number;

issueResolutionRate: number;

communityGrowth: number;

deploymentFrequency: number;

changeFailureRate: number;

}

Лістинг А.5 – Інтерфейс Composite Quality Score

export interface CompositeQualityScore {

overall: number;

breakdown: {

developerExperience: CategoryScore;

technicalPerformance: CategoryScore;

businessImpact: CategoryScore;

};

confidence: number;

trend: 'IMPROVING' | 'STABLE' | 'DECLINING' | 'UNKNOWN';

dataSource: 'GITHUB\_API' | 'SURVEY' | 'ENTERPRISE\_API';

timestamp: string;

projectContext: ProjectContext;

recommendations: string[];

predictions?: PredictedMetrics;

}

export interface CategoryScore {

score: number;

weight: number;

metrics: MetricValue[];

normalizedMetrics: NormalizedMetric[];

strengths: string[];

weaknesses: string[];

}

Лістинг А.6 – Інтерфейс конфігурації метрик

export interface MetricConfig {

name: string;

min: number;

max: number;

direction: 'lower\_is\_better' | 'higher\_is\_better' | 'optimal\_range';

optimal?: [number, number];

unit: string;

description: string;

}

Лістинг А.7 – Конфігурації діапазонів метрик

export const METRIC\_CONFIGS: Record<string, MetricConfig> = {

codeReviewDuration: {

name: 'Code Review Duration',

min: 0,

max: 168,

direction: 'lower\_is\_better',

unit: 'hours',

description: 'Середній час від створення PR до merge',

},

testCoverage: {

name: 'Test Coverage',

min: 0,

max: 100,

direction: 'higher\_is\_better',

unit: '%',

description: 'Покриття коду автоматизованими тестами',

},

averageCommentsPerPR: {

name: 'Average Comments Per PR',

min: 0,

max: 50,

direction: 'optimal\_range',

optimal: [5, 15],

unit: 'comments',

description: 'Середня кількість коментарів на pull request',

},

// ... інші метрики

};

Лістинг А.8 – Функції валідації метрик

export interface ValidationResult {

valid: boolean;

error?: string;

errors?: string[];

warning?: string;

}

export function validateMetric(

metricName: string,

value: number

): ValidationResult {

const config = METRIC\_CONFIGS[metricName];

if (!config) {

return {

valid: false,

error: `Unknown metric: ${metricName}`,

};

}

if (value < config.min || value > config.max) {

return {

valid: false,

error: `Value ${value} out of range [${config.min}, ${config.max}]`,

warning: 'Consider adjusting normalization range',

};

}

return { valid: true };

}

export function validateMetrics(metrics: OpenSourceMetrics): ValidationResult {

const errors: string[] = [];

for (const [key, value] of Object.entries(metrics.developerExperience)) {

const result = validateMetric(key, value);

if (!result.valid) errors.push(result.error!);

}

for (const [key, value] of Object.entries(metrics.technicalPerformance)) {

const result = validateMetric(key, value);

if (!result.valid) errors.push(result.error!);

}

for (const [key, value] of Object.entries(metrics.businessImpact)) {

const result = validateMetric(key, value);

if (!result.valid) errors.push(result.error!);

}

return errors.length === 0 ? { valid: true } : { valid: false, errors };

}

# ДОДАТОК Б

**Лістинги програмного коду**

Лістинг Б.1 – Клас GitHubCollector

/\*\*

\* GitHub API колектор для на основі результатів метрик

\*

\* Використовує GitHub REST API v3 для збору даних про репозиторій.

\* Імплементує rate limiting, caching та error handling.

\*/

export class GitHubCollector {

private baseUrl = 'https://api.github.com';

private token: string;

private cache: Map<string, any> = new Map();

constructor(token: string) {

if (!token) {

throw new Error('GitHub token is required');

}

this.token = token;

}

/\*\*

\* HTTP request wrapper з rate limiting та error handling

\*/

private async request<T>(

endpoint: string,

options: RequestInit = {}

): Promise<T> {

// Check cache first

const cacheKey = `${endpoint}:${JSON.stringify(options)}`;

if (this.cache.has(cacheKey)) {

return this.cache.get(cacheKey);

}

const url = `${this.baseUrl}${endpoint}`;

const response = await fetch(url, {

...options,

headers: {

Authorization: `Bearer ${this.token}`,

Accept: 'application/vnd.github.v3+json',

'User-Agent': 'thesis-metrics-collector',

...options.headers,

},

});

// Handle rate limiting

if (response.status === 403) {

const resetTime = response.headers.get('X-RateLimit-Reset');

if (resetTime) {

const waitTime = parseInt(resetTime) \* 1000 – Date.now();

console.warn(`Rate limit hit. Waiting ${Math.ceil(waitTime / 1000)}s...`);

await new Promise((resolve) => setTimeout(resolve, waitTime + 1000));

return this.request<T>(endpoint, options); // Retry

}

}

if (!response.ok) {

throw new Error(

`GitHub API error: ${response.status} ${response.statusText}`

);

}

const data = await response.json();

// Cache response

this.cache.set(cacheKey, data);

return data as T;

}

/\*\*

\* Збір Developer Experience метрик

\*/

async collectDeveloperExperienceMetrics(

owner: string,

repo: string

): Promise<OpenSourceDeveloperExperience> {

console.log(

`Collecting Developer Experience metrics for ${owner}/${repo}...`

);

// 1. Fetch pull requests (last 100 merged)

const pullRequests = await this.getPullRequests(owner, repo, 'closed', 100);

const mergedPRs = pullRequests.filter((pr) => pr.merged\_at);

// 2. Calculate code review duration

const codeReviewDuration = this.calculateAveragePRDuration(mergedPRs);

console.log(

` Code Review Duration: ${codeReviewDuration.toFixed(2)} hours`

);

// 3. Calculate average comments per PR

const averageCommentsPerPR =

mergedPRs.reduce(

(sum, pr) => sum + (pr.comments + pr.review\_comments),

0

) / mergedPRs.length;

console.log(` Avg Comments Per PR: ${averageCommentsPerPR.toFixed(1)}`);

// 4. Calculate PR iteration rate

const prIterationRate = await this.calculatePRIterationRate(

owner,

repo,

mergedPRs

);

console.log(` PR Iteration Rate: ${(prIterationRate \* 100).toFixed(1)}%`);

// 5. Fetch bug issues

const bugIssues = await this.getIssues(owner, repo, 'closed', 100, 'bug');

// 6. Calculate debugging time

const debuggingTime =

bugIssues

.filter((issue) => issue.closed\_at)

.map((issue) => {

const created = new Date(issue.created\_at).getTime();

const closed = new Date(issue.closed\_at!).getTime();

return (closed – created) / (1000 \* 60 \* 60); // hours

})

.reduce((sum, duration) => sum + duration, 0) / bugIssues.length;

console.log(` Debugging Time: ${debuggingTime.toFixed(2)} hours`);

// 7. Fetch CI/CD runs

const ciRuns = await this.getWorkflowRuns(owner, repo, 100);

const successfulDeploymentsRatio =

ciRuns.filter((run) => run.conclusion === 'success').length /

ciRuns.length;

console.log(

` Deployment Success Rate: ${(successfulDeploymentsRatio \* 100).toFixed(

1

)}%`

);

// 8. Calculate build time

const successfulRuns = ciRuns.filter((run) => run.conclusion === 'success');

const buildTime =

successfulRuns.reduce(

(sum, run) => sum + (run.run\_duration\_ms || 0) / 1000,

0

) / successfulRuns.length;

console.log(` Build Time: ${buildTime.toFixed(0)} seconds`);

return {

codeReviewDuration,

debuggingTime,

buildTime,

successfulDeploymentsRatio,

timeToFirstCommit: 2.5, // mock – requires fork events analysis

averageCommentsPerPR,

prIterationRate,

};

}

/\*\*

\* Розрахунок середньої тривалості PR

\*/

private calculateAveragePRDuration(pullRequests: any[]): number {

if (pullRequests.length === 0) return 0;

const durations = pullRequests.map((pr) => {

const created = new Date(pr.created\_at).getTime();

const merged = new Date(pr.merged\_at).getTime();

return (merged – created) / (1000 \* 60 \* 60); // hours

});

return durations.reduce((sum, d) => sum + d, 0) / durations.length;

}

/\*\*

\* Розрахунок PR iteration rate

\*/

private async calculatePRIterationRate(

owner: string,

repo: string,

pullRequests: any[]

): Promise<number> {

if (pullRequests.length === 0) return 0;

let prsWithIterations = 0;

for (const pr of pullRequests) {

const commits = await this.request<any[]>(

`/repos/${owner}/${repo}/pulls/${pr.number}/commits`

);

// Check if there are commits after first review

if (pr.review\_comments > 0 && commits.length > 1) {

const firstReviewTime = new Date(pr.created\_at).getTime() + 3600000; // +1h assumption

const laterCommits = commits.filter((c) => {

const commitTime = new Date(c.commit.author.date).getTime();

return commitTime > firstReviewTime;

});

if (laterCommits.length > 0) {

prsWithIterations++;

}

}

}

return prsWithIterations / pullRequests.length;

}

/\*\*

\* Отримання pull requests

\*/

private async getPullRequests(

owner: string,

repo: string,

state: 'open' | 'closed' | 'all',

count: number

): Promise<any[]> {

const endpoint = `/repos/${owner}/${repo}/pulls?state=${state}&per\_page=${count}&sort=updated&direction=desc`;

return this.request<any[]>(endpoint);

}

/\*\*

\* Отримання issues

\*/

private async getIssues(

owner: string,

repo: string,

state: 'open' | 'closed' | 'all',

count: number,

label?: string

): Promise<any[]> {

let endpoint = `/repos/${owner}/${repo}/issues?state=${state}&per\_page=${count}`;

if (label) {

endpoint += `&labels=${label}`;

}

return this.request<any[]>(endpoint);

}

/\*\*

\* Отримання workflow runs

\*/

private async getWorkflowRuns(

owner: string,

repo: string,

count: number

): Promise<any[]> {

const endpoint = `/repos/${owner}/${repo}/actions/runs?per\_page=${count}`;

const response = await this.request<any>(endpoint);

return response.workflow\_runs || [];

}

}

Лістинг Б.2 – Клас QualityScoreCalculator

import {

OpenSourceMetrics,

CompositeQualityScore,

CategoryScore

} from '@thesis/metrics';

/\*\*

\* Клас для розрахунку Composite Quality Score

\*/

export class QualityScoreCalculator {

/\*\*

\* Розрахунок Composite Quality Score

\*

\* @param metrics – Зібрані метрики проєкту

\* @param projectName – Назва проєкту

\* @returns Composite Quality Score з breakdown

\*/

calculateQualityScore(

metrics: OpenSourceMetrics,

projectName: string

): CompositeQualityScore {

// 1. Calculate category scores

const deScore = this.calculateDeveloperExperienceScore(

metrics.developerExperience

);

const tpScore = this.calculateTechnicalPerformanceScore(

metrics.technicalPerformance

);

const biScore = this.calculateBusinessImpactScore(metrics.businessImpact);

// 2. Calculate overall score (equal weights: 1/3 each)

const overall = Math.round(

(deScore.score + tpScore.score + biScore.score) / 3

);

// 3. Calculate confidence

const confidence = this.calculateConfidence(metrics);

// 4. Identify strengths (across all categories)

const strengths = this.identifyStrengths(

metrics,

deScore,

tpScore,

biScore

);

// 5. Generate recommendations

const recommendations = this.generateRecommendations(

deScore,

tpScore,

biScore

);

return {

overall,

breakdown: {

developerExperience: deScore,

technicalPerformance: tpScore,

businessImpact: biScore,

},

confidence,

trend: 'STABLE', // requires temporal data

dataSource: 'GITHUB\_API',

timestamp: new Date().toISOString(),

projectContext: {

name: projectName,

size: 'MEDIUM',

type: 'LIBRARY',

activity: 'ACTIVE',

},

recommendations,

};

}

/\*\*

\* Розрахунок Developer Experience score

\*/

private calculateDeveloperExperienceScore(

metrics: OpenSourceDeveloperExperience

): CategoryScore {

const normalizedMetrics: NormalizedMetric[] = [

{

name: 'codeReviewDuration',

normalizedValue: this.normalize(

metrics.codeReviewDuration,

0,

168,

'lower\_is\_better'

),

rawValue: metrics.codeReviewDuration,

},

{

name: 'debuggingTime',

normalizedValue: this.normalize(

metrics.debuggingTime,

0,

168,

'lower\_is\_better'

),

rawValue: metrics.debuggingTime,

},

{

name: 'buildTime',

normalizedValue: this.normalize(

metrics.buildTime,

0,

1800,

'lower\_is\_better'

),

rawValue: metrics.buildTime,

},

{

name: 'successfulDeploymentsRatio',

normalizedValue: this.normalize(

metrics.successfulDeploymentsRatio,

0,

1,

'higher\_is\_better'

),

rawValue: metrics.successfulDeploymentsRatio,

},

{

name: 'averageCommentsPerPR',

normalizedValue: this.normalize(

metrics.averageCommentsPerPR,

0,

50,

'optimal\_range',

{ optimal: [5, 15] }

),

rawValue: metrics.averageCommentsPerPR,

},

{

name: 'prIterationRate',

normalizedValue: this.normalize(

metrics.prIterationRate,

0,

1,

'optimal\_range',

{ optimal: [0.3, 0.6] }

),

rawValue: metrics.prIterationRate,

},

];

// Weighted average (equal weights for now)

const score =

(normalizedMetrics.reduce((sum, m) => sum + m.normalizedValue, 0) /

normalizedMetrics.length) \*

100;

// Identify strengths and weaknesses

const strengths = normalizedMetrics

.filter((m) => m.normalizedValue > 0.8)

.map((m) => m.name);

const weaknesses = normalizedMetrics

.filter((m) => m.normalizedValue < 0.3)

.map((m) => m.name);

return {

score: Math.round(score),

weight: 1 / 3,

metrics: Object.entries(metrics).map(([name, value]) => ({

name,

value,

unit: this.getUnit(name),

})),

normalizedMetrics,

strengths,

weaknesses,

};

}

/\*\*

\* Нормалізація метрики до [0, 1]

\*

\* @param value – Значення метрики

\* @param min – Мінімум діапазону

\* @param max – Максимум діапазону

\* @param direction – Напрямок інтерпретації

\* @param options – Додаткові параметри (оптимальний діапазон)

\* @returns Нормалізоване значення від 0 до 1

\*/

private normalize(

value: number,

min: number,

max: number,

direction: 'lower\_is\_better' | 'higher\_is\_better' | 'optimal\_range',

options?: { optimal?: [number, number] }

): number {

// Clamp value to range

const clampedValue = Math.max(min, Math.min(max, value));

if (direction === 'lower\_is\_better') {

// Lower values are better (e.g., code review duration)

return (max – clampedValue) / (max – min);

} else if (direction === 'higher\_is\_better') {

// Higher values are better (e.g., test coverage)

return (clampedValue – min) / (max – min);

} else {

// optimal\_range – piecewise linear function

const [optMin, optMax] = options?.optimal || [min, max];

if (clampedValue >= optMin && clampedValue <= optMax) {

// Inside optimal range – perfect score

return 1.0;

} else if (clampedValue < optMin) {

// Below optimal – linear interpolation from 0 to 1

return clampedValue / optMin;

} else {

// Above optimal – linear interpolation from 1 to 0

return 1 – (clampedValue – optMax) / (max – optMax);

}

}

}

/\*\*

\* Розрахунок confidence score

\*/

private calculateConfidence(metrics: OpenSourceMetrics): number {

// Count available metrics (non-zero, non-null)

const allMetrics = [

...Object.values(metrics.developerExperience),

...Object.values(metrics.technicalPerformance),

...Object.values(metrics.businessImpact),

];

const availableMetrics = allMetrics.filter(

(m) => m !== null && m !== undefined && m !== 0

).length;

const totalMetrics = allMetrics.length;

// Confidence = percentage of available metrics

return Math.round((availableMetrics / totalMetrics) \* 100);

}

/\*\*

\* Генерація рекомендацій на основі weaknesses

\*/

private generateRecommendations(

deScore: CategoryScore,

tpScore: CategoryScore,

biScore: CategoryScore

): string[] {

const recommendations: string[] = [];

// Developer Experience recommendations

if (deScore.weaknesses.includes('codeReviewDuration')) {

recommendations.push(

'Implement <48h SLA for code reviews to improve developer velocity'

);

}

if (deScore.weaknesses.includes('buildTime')) {

recommendations.push(

'Optimize CI/CD pipeline to reduce build time below 10 minutes'

);

}

// Technical Performance recommendations

if (tpScore.weaknesses.includes('testCoverage')) {

recommendations.push(

'Increase test coverage to target >85% for better code quality'

);

}

if (tpScore.weaknesses.includes('bundleSize')) {

recommendations.push(

'Implement code splitting and tree-shaking to reduce bundle size'

);

}

// Business Impact recommendations

if (biScore.weaknesses.includes('issueResolutionRate')) {

recommendations.push(

'Improve issue triage process to close issues within 7 days'

);

}

if (biScore.weaknesses.includes('communityGrowth')) {

recommendations.push(

'Enhance documentation and community engagement to attract contributors'

);

}

return recommendations;

}

/\*\*

\* Отримання одиниці виміру для метрики

\*/

private getUnit(metricName: string): string {

const units: Record<string, string> = {

codeReviewDuration: 'hours',

debuggingTime: 'hours',

buildTime: 'seconds',

successfulDeploymentsRatio: 'ratio',

timeToFirstCommit: 'days',

averageCommentsPerPR: 'comments',

prIterationRate: 'ratio',

testCoverage: '%',

typeScriptErrorRate: 'errors/1000LOC',

bundleSize: 'KB',

bundleLoadTime: 'ms',

performanceScore: 'score',

timeToMarket: 'days',

featureSuccessRate: 'ratio',

activeContributors: 'contributors',

issueResolutionRate: 'ratio',

communityGrowth: 'stars/month',

deploymentFrequency: 'deploys/month',

changeFailureRate: 'ratio',

};

return units[metricName] || 'unknown';

}

}

Лістинг Б.3 – Скрипт detailed-metrics-report.mjs

#!/usr/bin/env node

import fs from 'fs';

import path from 'path';

import { fileURLToPath } from 'url';

import { quickGitHubAssessment } from '@thesis/metrics-collector';

const \_\_filename = fileURLToPath(import.meta.url);

const \_\_dirname = path.dirname(\_\_filename);

/\*\*

\* Parse CLI arguments

\*/

function parseArgs() {

const args = process.argv.slice(2);

const parsed = {

projects: null,

outputDir: null,

existingReport: null,

};

for (let i = 0; i < args.length; i++) {

if (args[i] === '--projects' && args[i + 1]) {

parsed.projects = args[i + 1];

i++;

} else if (args[i] === '--outputDir' && args[i + 1]) {

parsed.outputDir = args[i + 1];

i++;

} else if (args[i] === '--existingReport' && args[i + 1]) {

parsed.existingReport = args[i + 1];

i++;

}

}

return parsed;

}

/\*\*

\* Load existing report for caching

\*/

async function loadExistingReport(existingReportPath, dirname) {

if (!existingReportPath) return new Map();

try {

const reportPath = path.resolve(dirname, existingReportPath);

const reportContent = await fs.promises.readFile(reportPath, 'utf8');

const reportData = JSON.parse(reportContent);

const metricsMap = new Map();

if (reportData.projects && Array.isArray(reportData.projects)) {

for (const project of reportData.projects) {

if (project.name && !project.error) {

metricsMap.set(project.name, project);

}

}

}

console.log(

`Loaded ${metricsMap.size} cached projects from ${existingReportPath}`

);

return metricsMap;

} catch (error) {

console.warn(`Could not load existing report: ${error.message}`);

return new Map();

}

}

/\*\*

\* Main data collection function

\*/

async function generateDetailedReport() {

// 1. Parse CLI arguments

const cliArgs = parseArgs();

if (!cliArgs.projects || !cliArgs.outputDir) {

console.error(

'Usage: --projects <path> --outputDir <path> [--existingReport <path>]'

);

process.exit(1);

}

// 2. Read projects from JSON file

const projectsPath = path.resolve(\_\_dirname, cliArgs.projects);

const outputDir = path.resolve(\_\_dirname, cliArgs.outputDir);

const projectsContent = await fs.promises.readFile(projectsPath, 'utf8');

const projectsData = JSON.parse(projectsContent);

const repos = projectsData.projects;

console.log(`Found ${repos.length} projects to analyze`);

// 3. Load existing report if provided

const existingMetrics = await loadExistingReport(

cliArgs.existingReport,

\_\_dirname

);

// 4. Collect metrics

const results = [];

let cachedCount = 0;

let fetchedCount = 0;

let errorCount = 0;

console.log('\nStarting data collection...\n');

for (const repo of repos) {

const projectName = `${repo.owner}/${repo.repo}`;

// Check cache first

const cachedProject = existingMetrics.get(projectName);

if (cachedProject) {

console.log(`Using cached data for ${projectName}`);

results.push(cachedProject);

cachedCount++;

continue;

}

// Fetch from GitHub API

console.log(`Fetching ${projectName}...`);

try {

const result = await quickGitHubAssessment(

repo.owner,

repo.repo,

process.env.GITHUB\_TOKEN

);

results.push(result);

fetchedCount++;

console.log(`${projectName}: ${result.overallScore}/100\n`);

// Rate limiting delay

await new Promise((resolve) => setTimeout(resolve, 1000));

} catch (error) {

console.error(`Failed: ${projectName}: ${error.message}\n`);

errorCount++;

// Store error for report

results.push({

name: projectName,

error: error.message,

timestamp: new Date().toISOString(),

});

}

}

// 5. Collection statistics

console.log('\nCollection stats:');

console.log(` From cache: ${cachedCount} projects`);

console.log(` From GitHub API: ${fetchedCount} projects`);

console.log(` Errors: ${errorCount} projects`);

console.log(` Total: ${results.length} projects\n`);

// 6. Generate reports

await generateJSONReport(results, outputDir);

await generateCSVReport(results, outputDir);

await generateMarkdownReport(results, outputDir);

console.log('\nAll reports generated successfully!');

}

// Run

generateDetailedReport().catch(console.error);

Лістинг Б.4 – Функція генерації JSON звіту

/\*\*

\* Генерація JSON звіту

\*

\* @param results – Масив результатів збору метрик

\* @param outputDir – Каталог для збереження звіту

\*/

async function generateJSONReport(results, outputDir) {

const reportData = {

generatedAt: new Date().toISOString(),

totalProjects: results.length,

successfulProjects: results.filter((r) => !r.error).length,

failedProjects: results.filter((r) => r.error).length,

statistics: calculateStatistics(results),

projects: results,

};

const reportPath = path.join(outputDir, 'metrics\_report.json');

await fs.promises.writeFile(

reportPath,

JSON.stringify(reportData, null, 2),

'utf8'

);

console.log(`JSON report: ${reportPath}`);

console.log(` Size: ${(JSON.stringify(reportData).length / 1024).toFixed(1)} KB`);

}

/\*\*

\* Розрахунок статистики датасету

\*/

function calculateStatistics(results) {

const successfulResults = results.filter((r) => !r.error);

if (successfulResults.length === 0) {

return null;

}

const overallScores = successfulResults.map((r) => r.qualityScore.overall);

const dxScores = successfulResults.map(

(r) => r.qualityScore.breakdown.developerExperience.score

);

const tpScores = successfulResults.map(

(r) => r.qualityScore.breakdown.technicalPerformance.score

);

const biScores = successfulResults.map(

(r) => r.qualityScore.breakdown.businessImpact.score

);

return {

overallScore: {

mean: mean(overallScores),

median: median(overallScores),

std: standardDeviation(overallScores),

min: Math.min(...overallScores),

max: Math.max(...overallScores),

},

developerExperience: {

mean: mean(dxScores),

median: median(dxScores),

},

technicalPerformance: {

mean: mean(tpScores),

median: median(tpScores),

},

businessImpact: {

mean: mean(biScores),

median: median(biScores),

},

};

}

// Утиліти статистики

function mean(arr) {

return arr.reduce((sum, val) => sum + val, 0) / arr.length;

}

function median(arr) {

const sorted = [...arr].sort((a, b) => a – b);

const mid = Math.floor(sorted.length / 2);

return sorted.length % 2 === 0

? (sorted[mid – 1] + sorted[mid]) / 2

: sorted[mid];

}

function standardDeviation(arr) {

const avg = mean(arr);

const squareDiffs = arr.map((val) => Math.pow(val – avg, 2));

return Math.sqrt(mean(squareDiffs));

}

Лістинг Б.5 – Функція генерації CSV звіту

/\*\*

\* Генерація CSV звіту

\*

\* @param results – Масив результатів збору метрик

\* @param outputDir – Каталог для збереження звіту

\*/

async function generateCSVReport(results, outputDir) {

const successfulResults = results.filter((r) => !r.error);

if (successfulResults.length === 0) {

console.log('No successful projects to generate CSV report');

return;

}

const headers = [

'Project',

'Overall Score',

'DX Score',

'TP Score',

'BI Score',

'Confidence',

'Code Review Duration (h)',

'Debugging Time (h)',

'Build Time (s)',

'Test Coverage (%)',

'Bundle Size (KB)',

'Active Contributors',

'Community Growth (stars/month)',

'Issue Resolution Rate (%)',

'Time to Market (days)',

];

const rows = successfulResults.map((result) => [

result.name,

result.qualityScore.overall,

result.qualityScore.breakdown.developerExperience.score,

result.qualityScore.breakdown.technicalPerformance.score,

result.qualityScore.breakdown.businessImpact.score,

result.qualityScore.confidence,

result.metrics.developerExperience.codeReviewDuration.toFixed(2),

result.metrics.developerExperience.debuggingTime.toFixed(2),

result.metrics.developerExperience.buildTime.toFixed(0),

result.metrics.technicalPerformance.testCoverage.toFixed(0),

(result.metrics.technicalPerformance.bundleSize / 1024).toFixed(1),

result.metrics.businessImpact.activeContributors,

result.metrics.businessImpact.communityGrowth.toFixed(0),

(result.metrics.businessImpact.issueResolutionRate \* 100).toFixed(0),

result.metrics.businessImpact.timeToMarket.toFixed(1),

]);

// Escape CSV values

const escapeCsvValue = (val) => {

const str = String(val);

return str.includes(',') || str.includes('"') || str.includes('\n')

? `"${str.replace(/"/g, '""')}"`

: str;

};

const csv = [

headers.map(escapeCsvValue).join(','),

...rows.map((row) => row.map(escapeCsvValue).join(',')),

].join('\n');

const reportPath = path.join(outputDir, 'metrics\_report.csv');

await fs.promises.writeFile(reportPath, csv, 'utf8');

console.log(`CSV report: ${reportPath}`);

console.log(` Rows: ${rows.length + 1} (including header)`);

}

# ДОДАТОК В

**Python скрипти аналізу даних**

Лістинг В.1 – Python скрипт для ML predictions (analysis/predict.py)

#!/usr/bin/env python3

"""

ML Prediction Script для quality-dashboard

Завантажує натреновані моделі та виконує predictions для проєктів.

Підтримує базові predictions та what-if analysis.

"""

import sys

import json

import joblib

import numpy as np

import pandas as pd

from pathlib import Path

# Шляхи до збережених моделей

MODEL\_DIR = Path(\_\_file\_\_).parent / 'models'

MODELS = {

'overallScore': MODEL\_DIR / 'overallScore\_model.joblib',

'timeToMarket': MODEL\_DIR / 'timeToMarket\_model.joblib',

'communityGrowth': MODEL\_DIR / 'communityGrowth\_model.joblib',

}

SCALERS = {

'overallScore': MODEL\_DIR / 'overallScore\_scaler.joblib',

'timeToMarket': MODEL\_DIR / 'timeToMarket\_scaler.joblib',

'communityGrowth': MODEL\_DIR / 'communityGrowth\_scaler.joblib',

}

FEATURE\_LIST\_PATH = MODEL\_DIR / 'selected\_features.json'

def load\_models():

"""Завантаження всіх збережених моделей та scalers"""

models = {}

scalers = {}

for target, model\_path in MODELS.items():

if model\_path.exists():

models[target] = joblib.load(model\_path)

else:

print(f"Warning: Model not found: {model\_path}", file=sys.stderr)

for target, scaler\_path in SCALERS.items():

if scaler\_path.exists():

scalers[target] = joblib.load(scaler\_path)

else:

scalers[target] = None

return models, scalers

def load\_feature\_list():

"""Завантаження списку selected features"""

if FEATURE\_LIST\_PATH.exists():

with open(FEATURE\_LIST\_PATH, 'r') as f:

return json.load(f)

else:

raise FileNotFoundError(f"Feature list not found: {FEATURE\_LIST\_PATH}")

def engineer\_features(metrics):

"""

Feature engineering ідентичний training pipeline

Args:

metrics: Dict з raw метриками проєкту

Returns:

Dict з engineered features

"""

features = {}

# Original features (flattened)

features['dx\_codeReviewDuration'] = metrics['developerExperience']['codeReviewDuration']

features['dx\_debuggingTime'] = metrics['developerExperience']['debuggingTime']

features['dx\_buildTime'] = metrics['developerExperience']['buildTime']

features['dx\_successfulDeploymentsRatio'] = metrics['developerExperience']['successfulDeploymentsRatio']

features['dx\_averageCommentsPerPR'] = metrics['developerExperience']['averageCommentsPerPR']

features['dx\_prIterationRate'] = metrics['developerExperience']['prIterationRate']

features['tp\_testCoverage'] = metrics['technicalPerformance']['testCoverage']

features['tp\_typeScriptErrorRate'] = metrics['technicalPerformance']['typeScriptErrorRate']

features['tp\_bundleSize'] = metrics['technicalPerformance']['bundleSize']

features['tp\_performanceScore'] = metrics['technicalPerformance']['performanceScore']

features['bi\_timeToMarket'] = metrics['businessImpact']['timeToMarket']

features['bi\_featureSuccessRate'] = metrics['businessImpact']['featureSuccessRate']

features['bi\_activeContributors'] = metrics['businessImpact']['activeContributors']

features['bi\_issueResolutionRate'] = metrics['businessImpact']['issueResolutionRate']

features['bi\_communityGrowth'] = metrics['businessImpact']['communityGrowth']

features['bi\_deploymentFrequency'] = metrics['businessImpact']['deploymentFrequency']

# Interaction features

features['dx\_tp\_interaction'] = (

features['dx\_codeReviewDuration'] \* features['tp\_testCoverage']

)

features['tp\_bi\_interaction'] = (

features['tp\_testCoverage'] \* features['bi\_activeContributors']

)

# Category averages

dx\_values = [features[f] for f in features if f.startswith('dx\_')]

tp\_values = [features[f] for f in features if f.startswith('tp\_')]

bi\_values = [features[f] for f in features if f.startswith('bi\_')]

features['avg\_dx'] = np.mean(dx\_values)

features['avg\_tp'] = np.mean(tp\_values)

features['avg\_bi'] = np.mean(bi\_values)

return features

def predict(metrics, models, scalers, feature\_list):

"""

Виконання predictions для всіх target variables

Args:

metrics: Dict з метриками проєкту

models: Dict з завантаженими моделями

scalers: Dict з завантаженими scalers

feature\_list: List selected features

Returns:

Dict з predictions для кожного target

"""

# Engineer features

features = engineer\_features(metrics)

# Create feature vector

X = np.array([[features[f] for f in feature\_list]])

# Make predictions

predictions = {}

for target in ['overallScore', 'timeToMarket', 'communityGrowth']:

if target not in models:

continue

model = models[target]

scaler = scalers[target]

if scaler:

X\_scaled = scaler.transform(X)

pred = model.predict(X\_scaled)[0]

else:

pred = model.predict(X)[0]

predictions[target] = round(float(pred), 2)

return predictions

def what\_if\_analysis(metrics, changes, models, scalers, feature\_list):

"""

What-if аналіз: порівняння baseline vs modified predictions

Args:

metrics: Dict з поточними метриками

changes: Dict зі змінами метрик (e.g., {"dx\_codeReviewDuration": 48})

models, scalers, feature\_list: Завантажені моделі

Returns:

Dict з baseline, predicted та change для кожного target

"""

# Baseline predictions

baseline = predict(metrics, models, scalers, feature\_list)

# Apply changes to metrics

modified\_metrics = apply\_changes(metrics, changes)

# Modified predictions

modified = predict(modified\_metrics, models, scalers, feature\_list)

# Calculate differences

results = {}

for target in baseline:

results[target] = {

'baseline': baseline[target],

'predicted': modified[target],

'change': modified[target] – baseline[target],

}

return results

def apply\_changes(metrics, changes):

"""Застосування змін до метрик для what-if аналізу"""

modified = json.loads(json.dumps(metrics)) # Deep copy

for key, value in changes.items():

# Parse key (e.g., "dx\_codeReviewDuration" -> ["developerExperience", "codeReviewDuration"])

if key.startswith('dx\_'):

category = 'developerExperience'

metric = key[3:] # Remove "dx\_" prefix

elif key.startswith('tp\_'):

category = 'technicalPerformance'

metric = key[3:]

elif key.startswith('bi\_'):

category = 'businessImpact'

metric = key[3:]

else:

continue

if category in modified and metric in modified[category]:

modified[category][metric] = value

return modified

def main():

"""Main entry point"""

if len(sys.argv) < 2:

print("Usage: predict.py '<json\_input>'", file=sys.stderr)

sys.exit(1)

try:

# Parse input JSON

input\_data = json.loads(sys.argv[1])

# Load models

models, scalers = load\_models()

feature\_list = load\_feature\_list()

# Check if what-if analysis requested

if 'whatif' in input\_data:

# What-if analysis

metrics = input\_data['metrics']

changes = input\_data['whatif']

results = what\_if\_analysis(metrics, changes, models, scalers, feature\_list)

print(json.dumps(results, indent=2))

else:

# Regular prediction

metrics = input\_data

predictions = predict(metrics, models, scalers, feature\_list)

print(json.dumps(predictions, indent=2))

except Exception as e:

print(f"Error: {str(e)}", file=sys.stderr)

sys.exit(1)

if \_\_name\_\_ == '\_\_main\_\_':

main()

Лістинг В.2 – Python скрипт статистичного аналізу (analysis/statistical\_analysis.py) (фрагмент)

#!/usr/bin/env python3

"""

Statistical Analysis & Feature Engineering

Фаза 2.2: Поглиблений статистичний аналіз та feature engineering

Магістерська робота: Outcome-based оцінка якості TypeScript коду

Автор: Слабенко Костянтин Олегович

Одеський політехнічний національний університет

"""

import warnings

from pathlib import Path

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

from scipy import stats

from scipy.cluster.hierarchy import dendrogram, linkage

from sklearn.preprocessing import StandardScaler, MinMaxScaler

from sklearn.decomposition import PCA

from sklearn.cluster import KMeans

from sklearn.metrics import silhouette\_score

import statsmodels.api as sm

from statsmodels.stats.outliers\_influence import variance\_inflation\_factor

from statsmodels.stats.multitest import multipletests

import itertools

import json

warnings.filterwarnings("ignore")

# Шляхи до файлів

BASE\_DIR = Path(\_\_file\_\_).parent

REPORTS\_DIR = BASE\_DIR / "../reports"

STATISTICAL\_DIR = REPORTS\_DIR / "statistical"

STATISTICAL\_DIR.mkdir(parents=True, exist\_ok=True)

# Завантажити дані

df = pd.read\_csv(REPORTS\_DIR / "processed\_metrics.csv")

# Список числових метрик

numeric\_cols = [

col for col in df.columns

if col.startswith(("dx\_", "tp\_", "bi\_", "overallScore"))

]

def perform\_hypothesis\_testing(df, metrics\_list, alpha=0.05):

"""

Виконує hypothesis testing для кореляцій між метриками.

Returns: DataFrame з результатами: correlation, p-value, significant

"""

results = []

n = len(df)

for metric1, metric2 in itertools.combinations(metrics\_list, 2):

r, p\_value = stats.pearsonr(df[metric1], df[metric2])

# Fisher's Z transformation для confidence interval

z = np.arctanh(r)

se = 1 / np.sqrt(n – 3)

ci\_lower = np.tanh(z – 1.96 \* se)

ci\_upper = np.tanh(z + 1.96 \* se)

results.append({

"Метрика 1": metric1,

"Метрика 2": metric2,

"Кореляція": r,

"p-value": p\_value,

"Значуща (α=0.05)": p\_value < alpha,

"CI\_lower (95%)": ci\_lower,

"CI\_upper (95%)": ci\_upper,

})

return pd.DataFrame(results).sort\_values("Кореляція", key=abs, ascending=False)

def perform\_regression\_analysis(df, target, predictors, analysis\_name):

"""Виконує множинну регресію та зберігає результати."""

X = df[predictors].copy()

y = df[target].copy()

X\_with\_const = sm.add\_constant(X)

model = sm.OLS(y, X\_with\_const)

results = model.fit()

# VIF для multicollinearity

vif\_data = pd.DataFrame()

vif\_data["Метрика"] = X.columns

vif\_data["VIF"] = [

variance\_inflation\_factor(X.values, i) for i in range(X.shape[1])

]

return results, vif\_data

# Виконати hypothesis testing з FDR correction

hypothesis\_results = perform\_hypothesis\_testing(df, numeric\_cols)

reject, pvals\_corrected, \_, \_ = multipletests(

hypothesis\_results["p-value"], alpha=0.05, method="fdr\_bh"

)

hypothesis\_results["FDR corrected p-value"] = pvals\_corrected

hypothesis\_results["Значуща (FDR)"] = reject

hypothesis\_results.to\_csv(STATISTICAL\_DIR / "hypothesis\_tests.csv", index=False)

# Cluster Analysis з K-means

X\_cluster = df[numeric\_cols].copy()

scaler\_cluster = StandardScaler()

X\_scaled = scaler\_cluster.fit\_transform(X\_cluster)

# Визначення оптимальної кількості кластерів

silhouette\_scores = []

for k in range(2, 9):

kmeans = KMeans(n\_clusters=k, random\_state=42, n\_init=10)

kmeans.fit(X\_scaled)

silhouette\_scores.append(silhouette\_score(X\_scaled, kmeans.labels\_))

optimal\_k = range(2, 9)[np.argmax(silhouette\_scores)]

kmeans\_final = KMeans(n\_clusters=optimal\_k, random\_state=42, n\_init=10)

df["Cluster"] = kmeans\_final.fit\_predict(X\_scaled)

# PCA Analysis

pca = PCA()

X\_pca = pca.fit\_transform(X\_scaled)

explained\_variance = pca.explained\_variance\_ratio\_

print("✅ Statistical Analysis завершено!")

Лістинг В.3 – Python скрипт ML моделювання (analysis/ml\_modeling.py) (фрагмент)

#!/usr/bin/env python3

"""

ML Modeling & Predictive Analysis

Фаза 3: Machine Learning моделювання для prediction на основі результатів метрик

Магістерська робота: Outcome-based оцінка якості TypeScript коду

Автор: Слабенко Костянтин Олегович

Одеський політехнічний національний університет

"""

import warnings

from pathlib import Path

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.model\_selection import train\_test\_split, cross\_val\_score, KFold

from sklearn.preprocessing import StandardScaler

from sklearn.linear\_model import LinearRegression, Ridge, Lasso, ElasticNet

from sklearn.ensemble import RandomForestRegressor

from sklearn.metrics import r2\_score, mean\_squared\_error, mean\_absolute\_error

import xgboost as xgb

import lightgbm as lgb

import shap

warnings.filterwarnings("ignore")

REPORTS\_DIR = Path("../reports")

ML\_DIR = REPORTS\_DIR / "ml"

ML\_DIR.mkdir(exist\_ok=True)

# Завантаження даних

df = pd.read\_csv(REPORTS\_DIR / "statistical" / "engineered\_features.csv")

# Target variables

TARGET\_VARS = ["overallScore", "bi\_timeToMarket", "bi\_communityGrowth"]

# Safeguard: виявити leaked features

TARGET\_PATTERNS = ["timeToMarket", "communityGrowth", "overallScore"]

feature\_cols = [col for col in df.columns if col not in TARGET\_VARS]

leaked\_features = [

col for col in feature\_cols

for pattern in TARGET\_PATTERNS

if pattern in col and col not in TARGET\_VARS

]

feature\_cols = [col for col in feature\_cols if col not in leaked\_features]

# Видалити highly correlated features (r > 0.95)

corr\_matrix = df[feature\_cols].corr().abs()

upper\_triangle = corr\_matrix.where(

np.triu(np.ones(corr\_matrix.shape), k=1).astype(bool)

)

to\_drop = [col for col in upper\_triangle.columns if any(upper\_triangle[col] > 0.95)]

feature\_cols = [col for col in feature\_cols if col not in to\_drop]

# Моделі для тестування

models = {

"Linear Regression": LinearRegression(),

"Ridge": Ridge(alpha=1.0, random\_state=42),

"Lasso": Lasso(alpha=0.1, random\_state=42),

"ElasticNet": ElasticNet(alpha=0.1, l1\_ratio=0.5, random\_state=42),

"Random Forest": RandomForestRegressor(n\_estimators=100, max\_depth=10, random\_state=42),

"XGBoost": xgb.XGBRegressor(n\_estimators=100, max\_depth=6, learning\_rate=0.1, random\_state=42),

"LightGBM": lgb.LGBMRegressor(n\_estimators=100, max\_depth=6, learning\_rate=0.1, random\_state=42),

}

# Train/Test Split (70/15/15)

X = df[feature\_cols].copy()

for target\_name in ["overallScore", "bi\_timeToMarket", "bi\_communityGrowth"]:

y = df[target\_name].copy()

X\_temp, X\_test, y\_temp, y\_test = train\_test\_split(X, y, test\_size=0.15, random\_state=42)

X\_train, X\_val, y\_train, y\_val = train\_test\_split(X\_temp, y\_temp, test\_size=0.176, random\_state=42)

# Тренування моделей

for model\_name, model in models.items():

model.fit(X\_train, y\_train)

y\_pred = model.predict(X\_test)

test\_r2 = r2\_score(y\_test, y\_pred)

test\_rmse = np.sqrt(mean\_squared\_error(y\_test, y\_pred))

print(f"{target\_name} – {model\_name}: R²={test\_r2:.4f}, RMSE={test\_rmse:.4f}")

# SHAP Analysis для XGBoost

for target\_name in ["overallScore", "bi\_timeToMarket", "bi\_communityGrowth"]:

y = df[target\_name].copy()

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.15, random\_state=42)

xgb\_model = xgb.XGBRegressor(n\_estimators=100, max\_depth=6, random\_state=42)

xgb\_model.fit(X\_train, y\_train)

explainer = shap.TreeExplainer(xgb\_model)

shap\_values = explainer.shap\_values(X\_test)

mean\_abs\_shap = np.abs(shap\_values).mean(axis=0)

shap\_importance = pd.DataFrame({

"Feature": feature\_cols,

"Mean |SHAP|": mean\_abs\_shap

}).sort\_values("Mean |SHAP|", ascending=False)

shap\_importance.to\_csv(ML\_DIR / f"shap\_importance\_{target\_name}.csv", index=False)

print("✅ ML Modeling завершено!")

# ДОДАТОК Г

**Список проєктів дослідження**

Таблиця Г.1 – Список 50 TypeScript проєктів для дослідження

|  |  |  |
| --- | --- | --- |
| № | Проєкт | Категорія |
| 1 | microsoft/TypeScript | Core TypeScript |
| 2 | angular/angular | Core Frameworks |
| 3 | remix-run/remix | Core Frameworks |
| 4 | vitejs/vite | Build Tools |
| 5 | remix-run/react-router | Core Frameworks |
| 6 | storybookjs/storybook | Developer Tools |
| 7 | nestjs/nest | Core Frameworks |
| 8 | withastro/astro | Core Frameworks |
| 9 | BuilderIO/qwik | Core Frameworks |
| 10 | trpc/trpc | Core Frameworks |
| 11 | mui/material-ui | UI Components |
| 12 | ant-design/ant-design | UI Components |
| 13 | chakra-ui/chakra-ui | UI Components |
| 14 | mantinedev/mantine | UI Components |
| 15 | radix-ui/primitives | UI Components |
| 16 | shadcn-ui/ui | UI Components |
| 17 | tailwindlabs/headlessui | UI Components |
| 18 | react-bootstrap/react-bootstrap | UI Components |
| 19 | Semantic-Org/Semantic-UI-React | UI Components |
| 20 | palantir/blueprint | UI Components |
| 21 | reduxjs/redux | State Management |
| 22 | reduxjs/redux-toolkit | State Management |
| 23 | mobxjs/mobx | State Management |
| 24 | pmndrs/zustand | State Management |
| 25 | pmndrs/jotai | State Management |
| 26 | pmndrs/valtio | State Management |
| 27 | TanStack/query | State Management |
| 28 | apollographql/apollo-client | State Management |
| 29 | typeorm/typeorm | Build Tools |
| 30 | evanw/esbuild | Build Tools |
| 31 | swc-project/swc | Build Tools |
| 32 | prisma/prisma | Build Tools |
| 33 | web-infra-dev/rspack | Build Tools |
| 34 | vercel/turbo | Build Tools |
| 35 | testing-library/react-testing-library | Developer Tools |
| 36 | microsoft/playwright | Developer Tools |
| 37 | vitest-dev/vitest | Developer Tools |
| 38 | nrwl/nx | Developer Tools |
| 39 | colinhacks/zod | Developer Tools |
| 40 | typescript-eslint/typescript-eslint | Developer Tools |
| 41 | lerna/lerna | Developer Tools |
| 42 | mswjs/msw | Developer Tools |
| 43 | react-hook-form/react-hook-form | Data & Forms |
| 44 | TanStack/form | Data & Forms |
| 45 | JedWatson/react-select | Data & Forms |
| 46 | pmndrs/react-spring | Data & Forms |
| 47 | recharts/recharts | Data & Forms |
| 48 | clauderic/dnd-kit | Data & Forms |
| 49 | floating-ui/floating-ui | Data & Forms |
| 50 | rjsf-team/react-jsonschema-form | Data & Forms |