

Міністерство освіти і науки України
Державний університет "Житомирська політехніка"

Кваліфікаційна наукова
праця на правах рукопису

МОСКАЛИК ДМИТРО ОЛЕКСАНДРОВИЧ

УДК [004.94+004.052]:004.414.2

ДИСЕРТАЦІЯ

**МОДЕЛЮВАННЯ ЕФЕКТИВНОСТІ РОЗРОБКИ ПРОГРАМНОГО
ЗАБЕЗПЕЧЕННЯ В ЗАЛЕЖНОСТІ ВІД ТИПУ АРХІТЕКТУРИ
СЕРВЕРНИХ ПРОГРАМНИХ СИСТЕМ**

121 – Інженерія програмного забезпечення

Подається на здобуття наукового ступеня доктора філософії

Дисертація містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело



Москалик Д.О.

Науковий керівник: **Антонюк Дмитро Сергійович**, кандидат педагогічних наук,
доцент, доцент кафедри інженерії програмного забезпечення

Житомир 2026

АНОТАЦІЯ

Москалик Д.О. Моделювання ефективності розробки програмного забезпечення в залежності від типу архітектури серверних програмних систем. – Кваліфікаційна наукова праця на правах рукопису.

Дисертація на здобуття наукового ступеня доктора філософії за спеціальністю 121 "Інженерія програмного забезпечення". – Державний університет "Житомирська політехніка". – Житомир, 2026.

В сучасному світі велика кількість серверних програмних систем розроблені на базі монолітної багатошарової архітектури, тому що даний тип архітектури є найдоцільнішим на початку життєвого циклу програмного продукту через його простоту та швидкість розробки при невеликій кількості розробників. Проте з появою необхідності подальшого масштабування як компанії, так і процесів розробки програмного забезпечення, виникає потреба зміни архітектурного підходу програмної системи, оскільки монолітна багатошарова архітектура є недостатньо гнучкою для ефективного горизонтального масштабування. Найбільшою популярністю для подальшого розвитку програмної системи наразі користується сервісно-орієнтований архітектурний підхід, проте з часом все більше компаній піддають сумніву доцільність даного вибору, що може свідчити про недосконалість проведеного аналізу при прийнятті минулих архітектурних рішень. Альтернативним варіантом масштабованої архітектури є також модульний монолітний архітектурний підхід, який так само дозволяє тримати під контролем комплексний рівень складності системи шляхом розділення її на окремі частини (модулі) з можливістю обмеження складності кожної з них. Основним аргументом на користь вибору сервісно-орієнтованої архітектури зазвичай є додаткові можливості для автономності компонентів системи та їх ізоляції один від одного. Але існує також і ряд суб'єктивних факторів, що спонукають розробників схилитись саме до даного архітектурного вибору, на кшталт наявності більшої свободи в прийнятті технічних рішень, популярності застосування сервісно-орієнтованого підходу у великих та авторитетних ІТ-

компаніях та зростаючий попит на спеціалістів з відповідним досвідом на ринку праці. Тим часом рідко в аргументації вибору архітектури можна побачити оцінку вартості використання даного підходу, а якщо така і є, то вона стосується перш за все вартості розміщення сервісів, тоді як вплив на вартість процесу подальшої розробки системи зазвичай ігнорується через складність його оцінки. Проблема полягає в тому, що на даний момент не існує об'єктивного методу оцінки ефективності розробки програмного забезпечення в порівнянні між модульною монолітною та сервісно-орієнтованою архітектурами.

Метою дисертаційної роботи є моделювання ефективності розробки програмного забезпечення в залежності від типу архітектури серверних програмних систем шляхом симуляції процесів розробки програмного забезпечення в контексті різних типів програмної архітектури.

У **вступі** обґрунтовано актуальність теми дослідження, сформульовано мету та визначено завдання дослідження, описано методи дослідження, представлено інформацію про наукову новизну та практичне значення отриманих результатів, їх апробацію та публікації по дослідженню.

В **першому розділі** було проведено аналіз останніх досліджень у сфері моделювання процесів розробки програмного забезпечення, детально розглянуто структуру таких процесів з точки зору їх основних етапів, архітектурних підходів та організаційних факторів, а також досліджені відомі фактори впливу на продуктивність процесу розробки програмного забезпечення та методи оцінки ефективності розробки. Розглянуто основні парадигми симуляційного моделювання, такі як системна динаміка, дискретно-подійне та агентно-орієнтоване моделювання, причому останнє визнано найперспективнішим завдяки здатності відтворювати індивідуальну поведінку учасників процесу розробки програмного забезпечення та їх взаємодію між собою. Проаналізовано класичні моделі оцінювання трудовитрат (COCOMO, закон Брукса), які, попри фундаментальну цінність, потребують адаптації до сучасних реалій з врахуванням гнучких методологій управління проектами, географічно розподілених команд розробки та різних програмних архітектурних

стилів. Детально досліджено структуру сучасного життєвого циклу розробки та факторів впливу на продуктивність, включаючи різні види власності програмного коду та процес рецензування коду.

У **другому розділі** основну увагу було приділено підготовці математичного та алгоритмічного підґрунтя для проведення програмних симуляцій процесів розробки програмного забезпечення. На основі історичних даних проєктів «Apache Software Foundation» досліджено розподіл складності задач розробки, в результаті якого було запропоновано алгоритм генерації потоку робочих задач з імовірнісним розподілом складності, що базується на зворотному розподілі Гауса, та включає адаптацію для компенсації аномальних емпіричних значень, що не відповідають даному розподілу. Проаналізовано вісім модульних проєктів з відкритим кодом для моделювання зв'язності між програмними модулями, в результаті якого виявлено логнормальний розподіл розміру модулів і частот їх зміни, а також степеневу залежність зв'язності модулів від частоти внесення змін в їх програмний код. Для оптимізації обчислень запропоновано використання незалежних частот зміни модулів замість детального моделювання зв'язності кожної окремої їх пари. За допомогою лінійної регресії виведено коефіцієнти залежності параметрів логнормального розподілу незалежних імовірностей зміни модулів від загальної кількості модулів в програмній системі, що моделюється. Обґрунтовано вибір агентно-орієнтованого підходу як найбільш гнучкого для моделювання взаємодій учасників процесу розробки програмного забезпечення та розроблено архітектуру програмного комплексу для проведення симуляцій на основі фреймворку MARS LIFE, що враховує модульну структуру кодової бази, різні види власності програмного коду, склад, кількість та географічний розподіл команд розробки, втому розробників та їх вплив на моделювання процесу рецензування програмного коду.

В **третьому розділі** розроблено симуляційну модель процесу розробки програмного забезпечення на основі агентно-орієнтованого моделювання з використанням фреймворку MARS LIFE, проведено комплексний аналіз

результатів 8080 симуляцій та побудовано регресійні моделі оцінки ефективності процесу розробки для різних типів програмної архітектури. Встановлено, що архітектурна модель є визначальним фактором продуктивності. Мікросервісна архітектура демонструє найвищу середню продуктивність, а модульна монолітна із сильною власністю програмного коду – найнижчу. Географічний розподіл команд між часовими поясами знижує продуктивність для всіх типів архітектури, але з різною інтенсивністю. Збільшення кількості модулів в модульній монолітній архітектурі з сильною власністю коду спричиняє колапс продуктивності на 78%, натомість для мікросервісної дає приріст у 15%. Оптимальний розмір команд також варіюється між різними типами архітектури, де для мікросервісної це малі команди по 2 розробники, а для монолітних систем важливий паритет розробників і тестувальників. Побудовано п'ять типів регресійних моделей оцінки ефективності процесу розробки для різних типів програмної архітектури, серед яких ансамблеві (Gradient Boosting та Random Forest) забезпечують найвищу точність із коефіцієнтом детермінації від 0,92 до 0,97. Сформульовано практичні рекомендації щодо оптимальних параметрів для кожної архітектурної моделі. Симуляційна модель процесу розробки програмного забезпечення була провалідована методом експертних оцінок, а регресійні моделі – методом 10-кратного перехресного затвердження.

У дисертаційній роботі отримано низку нових **наукових результатів**, зокрема:

вперше: розроблено агентно-орієнтовану симуляційну модель процесу розробки програмного забезпечення для різних типів архітектури серверних програмних систем, що охоплює повний життєвий цикл задач від програмування до розгортання та дозволяє кількісно оцінити вплив архітектурного стилю на ефективність процесу розробки програмного забезпечення; побудовано параметричні та ансамблеві регресійні моделі оцінки ефективності розробки програмного забезпечення для кожного з досліджених архітектурних підходів; виявлено та кількісно описано вплив географічного розподілу, розміру та складу команд розробки, виду практики власності програмного коду, обсягу коду та

кількості модулів на ефективність процесу розробки програмного забезпечення для монолітної, модульної монолітної та мікросервісної програмних архітектур;

уточнено: закономірності впливу географічного розподілу команд між часовими поясами на продуктивність розробки для різних архітектурних моделей; характеристики масштабування команд розробки для різних архітектурних підходів;

подальшого розвитку набули алгоритми генерації потоку вхідних задач та моделювання зв'язності між модулями програмної системи для симуляції процесів розробки програмного забезпечення.

Практична значимість отриманих результатів визначається наступним:

побудовані регресійні математичні моделі оцінки ефективності розробки програмного забезпечення можуть бути використані як інструмент додаткової аргументації при первинному виборі між сервісно-орієнтованим та модульним монолітним архітектурними підходами для майбутніх серверних програмних систем, а також при оцінці доцільності переходу з поточного архітектурного рішення на альтернативне;

розроблений програмний комплекс на основі фреймворку MARS LIFE дозволяє проводити детальне симуляційне моделювання процесів розробки програмного забезпечення для різних типів програмної архітектури з метою ідентифікації вузьких місць та факторів, що сповільнюють процес розробки, для подальшої оптимізації організації процесу;

надані рекомендації по максимізації ефективності процесу розробки для кожного архітектурного підходу можуть бути безпосередньо використані керівниками програмних проєктів при плануванні складу команд, розподілі ресурсів та прийнятті рішень щодо географічного розташування розподілених команд розробки.

Результати дослідження викладено у 9 наукових працях, з них 3 статті у фахових виданнях, включених до переліку наукових фахових видань України з присвоєнням категорії «Б», та 6 у матеріалах науково-технічних конференцій, з яких 1 публікація у матеріалах міжнародної наукової конференції,

проіндексованої у базі даних Scopus.

Ключові слова: агентно-орієнтовані симуляції, мікросервісна архітектура, моделювання, модульна монолітна архітектура, модульність, оцінка ефективності, процес розробки програмного забезпечення, регресійні моделі.

ABSTRACT

Moskalyk D.O. Modeling the efficiency of software development process depending on the architecture type of server software systems. – Qualifying scientific work as a manuscript.

Dissertation for the degree of Doctor of Philosophy in specialty 121 “Software Engineering”. – Zhytomyr Polytechnic State University. – Zhytomyr, 2026.

Many modern server-side software systems are built on monolithic layered architectures, as this architectural style is the most practical choice at the early stages of the software product lifecycle due to its simplicity and rapid development with small teams. However, as the need arises to scale both the organization and its software development processes, so does the need to revisit the system’s architectural approach, since monolithic layered architectures lack the flexibility required for effective horizontal scaling. The most widely adopted path for further system evolution is the service-oriented architectural approach, yet a growing number of companies have begun to question whether this choice is truly warranted, suggesting that the analysis of underlying past architectural decisions may have been insufficient. An alternative scalable architecture is the modular monolith, which likewise enables control over system complexity by decomposing the system into discrete parts (modules) while constraining the complexity of each. The primary argument in favor of service-oriented architecture is typically the additional autonomy and isolation it affords to system components. At the same time, several subjective factors also drive developers toward this architectural choice, such as greater freedom in making technical decisions, the prevalence of service-oriented approaches at large and reputable IT companies, and the growing market demand for specialists with relevant experience. Meanwhile, cost considerations are rarely considered in the architectural decision-making process, and when they do, they tend to focus on service hosting costs, while the impact on ongoing development costs is typically overlooked due to its estimation difficulty. The core problem is that no objective method currently exists for comparing software development efficiency between modular monolithic and service-oriented architectures.

The **purpose** of the dissertation is to model the efficiency of software development depending on the type of architecture of server software systems by simulating software development processes in the context of different types of software architecture.

The **introduction** establishes the relevance of the research topic, formulates the objective, defines the research tasks, describes the methods employed, and presents information on the scientific novelty and practical significance of the results, along with their validation and associated publications.

The first chapter provides a comprehensive review of recent research on modeling software development processes, examines the structure of such processes in terms of their key stages, architectural approaches, and organizational factors, and investigates known factors affecting development productivity as well as methods for evaluating development efficiency. The chapter reviews the principal simulation modeling paradigms, such as system dynamics, discrete-event simulation, and agent-based modeling, with the latter identified as the most promising due to its ability to reproduce the individual behavior of development process participants and their interactions. Classical effort estimation models (COCOMO, Brooks's Law) are analyzed and, despite their foundational value, are shown to require adaptation to modern realities, including agile project management methodologies, geographically distributed development teams, and diverse software architectural styles. The chapter also provides an in-depth examination of the modern software development lifecycle and productivity factors, including different code ownership practices and the code review process.

The second chapter focuses on establishing the mathematical and algorithmic foundations for conducting software simulations of the development process. Drawing on historical data from Apache Software Foundation projects, the distribution of task complexity was investigated, resulting in a proposed algorithm for generating a stream of work items with a probability-based complexity distribution grounded in the inverse Gaussian distribution, incorporating an adaptation mechanism to compensate for anomalous empirical values that deviate from this distribution. Eight open-source

modular projects were analyzed to model inter-module coupling, revealing a log-normal distribution of module sizes and modification frequencies, as well as a power-law relationship between module coupling and code modification frequency. To optimize computation, the use of independent module modification frequencies was proposed as an alternative to detailed pairwise coupling modeling. Linear regression was used to derive the coefficients relating the log-normal distribution parameters of independent module modification probabilities to the total number of modules in the simulated system. The chapter justifies the selection of the agent-based approach as the most flexible paradigm for modeling interactions among software development participants, and presents the architecture of a simulation framework built on the MARS LIFE platform, which accounts for the modular structure of the codebase, different code ownership practices, team composition, size and geographic distribution, developer fatigue, and its impact on the code review process.

The third chapter presents the simulation model of the software development process based on agent-based modeling using the MARS LIFE framework, provides a comprehensive analysis of the results from 8080 simulations, and constructs regression models for evaluating development efficiency across different architectural styles. The architectural model was found to be the determining factor in productivity. Microservice architecture exhibited the highest average productivity, while modular monolith with strong code ownership showed the lowest. Geographic distribution of teams across time zones reduced productivity for all architectural types, albeit with varying intensity. Increasing the number of modules in a modular monolith with strong code ownership caused a 78% productivity collapse, whereas for microservices it yielded a 15% improvement. Optimal team sizes also varied by architecture: microservices favored small teams of 2 developers, while monolithic systems benefited from parity between developers and testers. Five types of regression models were constructed for evaluating development efficiency across different architectural styles, among which ensemble methods (Gradient Boosting and Random Forest) achieved the highest accuracy, with coefficients of determination ranging from 0.92 to 0.97. Practical recommendations were formulated regarding optimal parameters for each

architectural model. The simulation model was validated through expert evaluation, and the regression models were validated using 10-fold cross-validation.

The dissertation produced several novel **scientific contributions**, specifically:

for the first time: an agent-based simulation model of the software development process was developed for different server-side system architectures, covering the full task lifecycle from coding to deployment and enabling quantitative assessment of the impact of architectural style on development efficiency; parametric and ensemble regression models were constructed for evaluating software development efficiency under each of the studied architectural approaches; the effects of geographic distribution, team size and composition, code ownership practices, codebase volume, and module count on development efficiency were identified and quantified for monolithic, modular monolithic, and microservice architectures;

refined: the patterns governing how geographic distribution of teams across time zones affects development productivity for different architectural models; the scaling characteristics of development teams under different architectural approaches;

further evolved: the algorithms for generating incoming task streams and modeling inter-module coupling in software systems for the purpose of simulating software development processes.

The **practical significance** of the results is as follows:

the regression models for evaluating software development efficiency can serve as a supplementary decision-support tool when choosing between service-oriented and modular monolithic architectural approaches for future server-side systems, as well as when assessing the viability of migrating from a current architecture to an alternative one;

the simulation framework built on the MARS LIFE platform enables detailed simulation modeling of software development processes under different architectural paradigms, with the aim of identifying bottlenecks and factors that slow down development, thereby informing process optimization;

the recommendations for maximizing development efficiency under each architectural approach can be directly applied by software project managers when

planning team composition, allocating resources, and making decisions regarding the geographic placement of distributed development teams.

The research findings have been published in 9 scientific works, including 3 articles in specialized journals listed in the registry of scientific specialized publications of Ukraine (Category "B") and 6 publications in scientific and technical conference proceedings, one of which appeared in the proceedings of an international scientific conference indexed in Scopus database.

Keywords: agent-based simulations, efficiency assessment, microservice architecture, modeling, modular monolithic architecture, modularity, software development process, regression models.

СПИСОК ПУБЛІКАЦІЙ ЗДОБУВАЧА ЗА ТЕМОЮ ДИСЕРТАЦІЇ
Наукові праці, в яких опубліковано
основні наукові результати дисертації.

Статті у фахових виданнях, включених до переліку наукових фахових видань України з присвоєнням категорії «Б»:

1. Москалик Д.О., Антонюк Д.С. Імовірнісний розподіл складності вхідних задач для дискретно-подійного та агентно-орієнтованого моделювання процесу розробки програмного забезпечення. *Вчені записки Таврійського національного університету імені В.І. Вернадського. Серія: Технічні науки*. 2024. Том 35 (74) № 3, Частина 1. С. 152-159. DOI: 10.32782/2663-5941/2024.3.1/23

2. Москалик Д.О., Антонюк Д.С. Моделювання зв'язності шляхом імовірнісного розподілу частот внесення змін до програмних модулів при симуляції процесу розробки програмного забезпечення. *Вісник Херсонського національного технічного університету*. 2025. Том 2 № 2 (93). С. 238-245. DOI: 10.35546/kntu2078-4481.2025.2.2.28

3. Москалик Д.О., Антонюк Д.С. Моделювання процесу рецензування коду в агентно-орієнтованих симуляціях розробки модульного програмного забезпечення. *Актуальні проблеми автоматизації та інформаційних технологій*. 2025. Том 29. С. 306-317. DOI: 10.15421/432528

Публікація у матеріалах міжнародної наукової конференції, що проіндексована у базі даних Scopus:

4. Mykhailo Kotenko, Dmytro Moskalyk, Valeriia Kovach, Viacheslav Osadchyi. Navigating the challenges and best practices in securing microservices architecture. *Proceedings of the Cybersecurity Providing in Information and Telecommunication Systems II*, Kyiv, Ukraine, October 26, 2024 (online). Edited by Volodymyr Sokolov, Vasyl Ustimenko, Tamara Radivilova, Mariya Nazarkevych. CEUR Workshop Proceedings, Vol. 3826, 2024. PP.1–16. URL: <https://ceur-ws.org/Vol-3826/paper1.pdf>

Публікації в матеріалах наукових конференцій:

5. Москалик Д.О., Антонюк Д.С. Вплив архітектури програмної системи на продуктивність розробки програмного забезпечення. *Інформаційні технології в освіті та науці* : Збірник наукових праць III Міжнародної науково-практичної конференції, м. Мелітополь, 25-26 травня 2023 р. / Мелітопольський державний педагогічний університет ім. Богдана Хмельницького. Мелітополь, 2023. С. 491-495. URL: <https://drive.google.com/file/d/1wcYwBHxWECzv9TT3-SIDBYUR4Z8eVURI/view>

6. Москалик Д.О., Антонюк Д.С. Вплив різновидів практики власності коду на процес розробки програмного забезпечення. *Комп'ютерні технології: інновації, проблеми, рішення* : тези доп. V Всеукраїнської науково-технічної конференції, м. Житомир, 01-02 грудня 2022 р. / Державний університет «Житомирська політехніка». Житомир, 2022. С. 32-33. URL: <https://conf.ztu.edu.ua/wp-content/uploads/2023/02/32.pdf>

7. Москалик Д.О., Антонюк Д.С. Підхід до моделювання процесу розробки програмного забезпечення для дослідження впливу архітектури на продуктивність розробки. *Тенденції розвитку технологій в автоматизації, електроенергетиці, приладобудуванні та робототехніці* : матеріали Всеукраїнської науково-практичної on-line конференції здобувачів вищої освіти і молодих учених, присвяченої Дню науки, м. Житомир, 13 травня 2024 р. / Державний університет «Житомирська політехніка». Житомир, 2024. С. 81. URL: <https://conf.ztu.edu.ua/wp-content/uploads/2024/06/sekcija-4.pdf>

8. Москалик Д.О., Антонюк Д.С. Моделювання зв'язності вихідного коду в симуляціях процесу розробки програмного забезпечення. *Сучасні інформаційні технології та системи в управлінні [Електронний ресурс]* : збірник матеріалів VI Міжнародної науково-практичної конференції молодих вчених, аспірантів і студентів, м. Київ, 10-11 квітня 2025 р. – Київ: КНЕУ, 2025. С. 314-315. URL:

https://kneu.edu.ua/userfiles//Faculty_of_Information_Systems_and_Technology/kaf%20ise/kmv/ZbD196rnik_tez_2025.pdf

9. Москалик Д.О., Антонюк Д.С. Аналіз розподілу складності задач при розробці програмного забезпечення з відкритим вихідним кодом. *Сучасні комп'ютерні системи та мережі в управлінні* : матеріали VI Всеукраїнської наук.-практ. Інтернет-конф. здобувачів вищої освіти та молодих вчених, м. Хмельницький, м. Херсон, 30 листопада 2023 р. / Херсонський національний технічний університет. Херсон, 2023. С. 31-32. URL: <https://kntu.net.ua/ukr/content/download/109941/620832/file/CICT 2023.pdf>

Зміст

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ.....	18
Вступ.....	19
РОЗДІЛ 1. Теоретичні аспекти моделювання процесів розробки програмного забезпечення та оцінки їх ефективності.....	27
1.1. Аналіз останніх досліджень у галузі ефективності розробки програмного забезпечення.....	27
1.2. Структура процесу розробки програмного забезпечення	38
1.3. Дослідження архітектурних факторів впливу на продуктивність розробки ПЗ та метрик оцінки ефективності процесу розробки	51
Висновки до першого розділу	62
РОЗДІЛ 2. Алгоритмічні основи моделювання процесу розробки програмного забезпечення	66
2.1. Алгоритм імовірнісної генерації потоку вхідних задач різної складності на основі історичних даних проєктів з відкритим вихідним кодом.....	66
2.2. Алгоритм моделювання зв'язності модулів програмної системи шляхом імовірнісного розподілу внесення змін на основі історичних даних проєктів з відкритим вихідним кодом	76
2.3. Архітектура програмного комплексу для проведення симуляцій процесів розробки програмного забезпечення	87
Висновки до другого розділу	98
РОЗДІЛ 3. Симуляційне моделювання процесів розробки програмного забезпечення для різних типів архітектури серверних програмних систем	102
3.1. Агентно-орієнтована симуляційна модель процесу розробки ПЗ для різних типів програмної архітектури.....	102
3.2. Аналіз результатів симуляцій процесів розробки ПЗ в контексті різних архітектурних моделей.....	112

3.3. Регресійні моделі оцінки ефективності процесу розробки для різних типів архітектури програмних систем.....	132
3.4. Валідація симуляційної моделі процесу розробки ПЗ методом експертних оцінок	155
Висновки до розділу 3	163
Висновки.....	166
Список використаних джерел	170
Додатки	183
Додаток А.....	183
Повний перелік подій, що генеруються програмним комплексом симуляції процесу розробки програмного забезпечення	183
Додаток Б	185
Вихідний код симуляційного програмного комплексу.....	185
Додаток В	225
Варіації базових сценаріїв симуляції процесу розробки програмного забезпечення	225
Додаток Г	226
Матеріали для опитування щодо валідності симуляційної моделі процесу розробки програмного забезпечення.....	226
Додаток Д.....	228
Таблиці проміжних і допоміжних даних для аналізу результатів експертного оцінювання симуляційної моделі процесу розробки ПЗ.....	228
Додаток Е	233
Список публікацій здобувача за темою дисертації	233
Додаток Ж	236
Відомості про апробацію результатів дисертації	236
Додаток З.....	238
Довідки про впровадження результатів дисертації	238
Додаток И.....	240
Звіт подібності дисертації	240

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

Скорочення, термін, позначення	Пояснення
ПЗ	Програмне забезпечення
SPSM	Software Process Simulation Modeling
SD	System Dynamics
DES	Discrete-Event Simulation
ABM	Agent-Based Modelling
API	Application Programming Interface
CI/CD	Continuous Integration / Continuous Deployment
MARS	Multi-Agent Research & Simulation
MAE	Mean Absolute Error
RSME	Root Mean Square Error

Вступ

Актуальність дослідження. В сучасному світі велика кількість серверних програмних систем розроблені на базі монолітної багатоваршавової архітектури, тому що даний тип архітектури є найдоцільнішим на початку життєвого циклу програмного продукту через його простоту та швидкість розробки при невеликій кількості розробників. Проте з появою необхідності подальшого масштабування як компанії, так і процесів розробки програмного забезпечення (ПЗ), виникає потреба зміни архітектурного підходу програмної системи, оскільки монолітна багатоваршавова архітектура є недостатньо гнучкою для ефективного горизонтального масштабування [1].

Найбільшою популярністю для подальшого розвитку програмної системи наразі користується сервісно-орієнтований архітектурний підхід, проте з часом все більше компаній піддають сумніву доцільність даного вибору [2], що може свідчити про недосконалість проведеного аналізу при прийнятті минулих архітектурних рішень.

Альтернативним варіантом масштабованої архітектури є також модульний монолітний архітектурний підхід, який так само дозволяє тримати під контролем комплексний рівень складності системи шляхом розділення її на окремі частини (модулі) з можливістю обмеження складності кожної з них.

Основним аргументом на користь вибору сервісно-орієнтованої архітектури зазвичай є додаткові можливості для автономності компонентів системи та їх ізоляції один від одного. Але існує також і ряд суб'єктивних факторів, що спонукають розробників схилитись саме до даного архітектурного вибору, на кшталт наявності більшої свободи в прийнятті технічних рішень, популярності застосування сервісно-орієнтованого підходу у великих та авторитетних ІТ-компаніях та зростаючий попит на спеціалістів з відповідним досвідом на ринку праці. Тим часом рідко в аргументації вибору архітектури можна побачити оцінку вартості використання даного підходу, а якщо така і є, то вона стосується перш за все вартості розміщення сервісів, тоді як вплив на вартість процесу подальшої розробки системи зазвичай ігнорується через

складність його оцінки [3, 4].

Першим відомим формальним методом аналізу архітектурних рішень є «Software Architecture Analysis Method» (SAAM), який був запропонований у 1994 році, а пізніше в інституті розробки програмного забезпечення університету Карнегі-Меллон розробили покращену версію даного методу під назвою «Architecture Tradeoff Analysis Method» (ATAM). Обидва методи створені для аналізу ефективності застосування окремих технічних рішень з точки зору задоволення вимог окремих архітектурних атрибутів якості на кшталт швидкодії, портативності, придатності до модифікації тощо.

Щодо аналізу вартості впровадження архітектурних рішень, найбільш відомим є метод «Cost-Benefit Analysis Method» (CBAM). Даний метод дозволяє оцінити вартість, переваги та ризики окремого потенційного архітектурного рішення, проте він передбачає, що в організації, де він застосовується, вже є набутий досвід планування та оцінки супутньої вартості впровадження технічних рішень.

Для оцінки ж вартості майбутніх змін до програмної системи в контексті використання різних архітектурних підходів був запропонований метод «Architecture-Level Modifiability Analysis» (ALMA). Даний метод дозволяє провести аналіз вартості змін до системи, базуючись на оцінці гіпотетичних сценаріїв, які можуть виникнути в майбутньому, тому якість даної оцінки буде сильно залежати від гіпотетичних сценаріїв та їх фактичної актуальності в майбутньому.

Вплив застосування сервісно-орієнтованої архітектури на вартість розробки програмного забезпечення досліджували в своїх роботах Міхаїл Переплетчиков (Mikhail Pereplechikov), Каспар Райан (Caspar Ryan), Захір Тарі (Zahir Tari), Том Юн (Tom Yoon), Памела Картер (Pamela Carter), проте проводилось тільки якісне порівняння відносно старих неструктурованих монолітних систем. По-перше, це також може бути ознакою складності кількісної оцінки вартості розробки. А по-друге, вказує на недостатність інформації щодо ефективності розробки сервісно-орієнтованих систем в

порівнянні саме з модульним монолітним підходом.

Більшість відомих практик, які дозволяють мінімізувати ризики та вартість майбутніх змін в програмній системі можуть бути застосовані однаковим чином як до сервісно-орієнтованої архітектури, так і до модульної монолітної. Саме тому, окрім задоволення функціональних вимог до кінцевої системи, вирішальним фактором у виборі конкретного архітектурного підходу може стати саме вартість чи ефективність подальшої розробки системи в контексті використання тієї чи іншої архітектури.

Проблема полягає в тому, що на даний момент не існує об'єктивного методу оцінки ефективності розробки програмного забезпечення в порівнянні між модульною монолітною та сервісно-орієнтованою архітектурами.

Процес розробки програмного забезпечення є соціотехнічним [5], у якому результат визначається не лише технічними рішеннями (архітектурою, інструментами, мовами програмування), а й складною взаємодією між людьми: комунікацією в команді, розподілом ролей, мотивацією, прийняттям рішень в умовах невизначеності, організаційною культурою. Соціотехнічний процес – це процес, в якому соціальна (людська, організаційна) та технічна підсистеми нерозривно пов'язані, і поведінка цілого не зводиться до суми їхніх окремих властивостей. Саме ця подвійна природа робить симуляційне моделювання особливо доцільним. Аналітичні моделі добре працюють там, де є можливість формалізувати залежності й отримати точне рішення, проте людський фактор вносить нелінійність, стохастичність і зв'язки, які складно описати аналітично.

Зв'язок роботи з науковими програмами, планами та темами. Тема дисертації затверджена Вченою радою Державного університету «Житомирська політехніка» (протокол № 9 від 31.08.2022 р.).

Дисертаційне дослідження виконувалось відповідно до НДР «Аналіз і моделювання процесу розробки програмних рішень в галузі електронної комерції» (12.01НДР-34-2023) кафедри інженерії програмного забезпечення Державного університету «Житомирська політехніка» (2023-2024 р.).

Мета дослідження – здійснити моделювання ефективності розробки програмного забезпечення в залежності від типу архітектури серверних програмних систем шляхом симуляції процесів розробки програмного забезпечення в контексті різних типів програмної архітектури.

Визначена мета досягається шляхом вирішення таких **завдань**:

1. Провести аналіз останніх досліджень у галузі ефективності розробки програмного забезпечення;
2. Розробити алгоритми імовірнісної генерації потоку вхідних задач різної складності та моделювання зв'язності модулів програмної системи на основі історичних даних проєктів з відкритим вихідним кодом;
3. Провести симуляції процесу розробки програмного забезпечення в контексті різних архітектурних моделей;
4. Визначити значення параметрів максимальної ефективності розробки окремо для кожного типу архітектури;
5. Побудувати регресійні моделі оцінки ефективності процесу розробки для різних типів архітектури програмних систем на основі результатів симуляцій.

Об'єктом дослідження є процеси розробки програмного забезпечення в залежності від архітектури серверних програмних систем.

Предмет дослідження – моделювання ефективності розробки програмного забезпечення в залежності від архітектури серверних програмних систем.

Для вирішення поставлених завдань використовувалась низка **методів**, зокрема *теоретичні*: аналіз останніх досліджень у галузі ефективності розробки програмного забезпечення, систематизація факторів впливу на продуктивність розробки програмного забезпечення для їх врахування в симуляційній моделі процесу розробки, порівняння процесів розробки програмного забезпечення в контексті різних типів архітектури для коректного моделювання впливу архітектури на організаційні аспекти процесу розробки; *емпіричні*: аналіз історичних даних складності задач розробки для імовірнісної генерації задач різної складності в симуляціях процесу розробки ПЗ, аналіз історії внесення змін

в окремі програмні модулі для коректного моделювання зв'язності між програмними модулями в симуляціях процесу розробки, агентно-орієнтоване моделювання процесу розробки програмного забезпечення для врахування в симуляційній моделі соціотехнічних аспектів процесу розробки; *статистичні*: регресійний аналіз результатів симуляцій процесу розробки для побудови математичних моделей оцінки ефективності процесу розробки ПЗ в залежності від типу архітектури та інших вхідних параметрів, апроксимація розподілів імовірностей складності задач для побудови алгоритму імовірнісної генерації задач різної складності в агентно-орієнтованих симуляційних моделях, кореляційний аналіз факторів впливу на продуктивність розробки для визначення найбільш вагомих параметрів для кожного архітектурного підходу.

Використання технологій штучного інтелекту. У процесі виконання дисертаційного дослідження технології штучного інтелекту використовувались як допоміжний інструмент візуалізації для кращого представлення окремих розроблених схем за власним описом.

Формування наукових положень, постановка мети і завдань дослідження, проведення експериментів, обробка результатів та формулювання висновків виконані автором самостійно. Технології штучного інтелекту не використовувались для генерування тексту дисертації чи формулювання наукових результатів.

Наукова новизна та теоретичне значення отриманих результатів полягають у тому, що:

вперше: розроблено агентно-орієнтовану симуляційну модель процесу розробки програмного забезпечення для різних типів архітектури серверних програмних систем, що охоплює повний життєвий цикл задач від програмування до розгортання та дозволяє кількісно оцінити вплив архітектурного стилю на ефективність процесу розробки програмного забезпечення; побудовано параметричні та ансамблеві регресійні моделі оцінки ефективності розробки програмного забезпечення для кожного з досліджених архітектурних підходів; виявлено та кількісно описано вплив географічного розподілу, розміру та складу

команд розробки, виду практики власності програмного коду, обсягу коду та кількості модулів на ефективність процесу розробки програмного забезпечення для монолітної, модульної монолітної та мікросервісної програмних архітектур;

уточнено: закономірності впливу географічного розподілу команд між часовими поясами на продуктивність розробки для різних архітектурних моделей; характеристики масштабування команд розробки для різних архітектурних підходів;

подальшого розвитку набули алгоритми генерації потоку вхідних задач та моделювання зв'язності між модулями програмної системи для симуляції процесів розробки програмного забезпечення.

Практичне значення отриманих результатів полягає в тому, що:

побудовані регресійні математичні моделі оцінки ефективності розробки програмного забезпечення можуть бути використані як інструмент додаткової аргументації при первинному виборі між сервісно-орієнтованим та модульним монолітним архітектурними підходами для майбутніх серверних програмних систем, а також при оцінці доцільності переходу з поточного архітектурного рішення на альтернативне;

розроблений програмний комплекс на основі фреймворку MARS LIFE дозволяє проводити детальне симуляційне моделювання процесів розробки програмного забезпечення для різних типів програмної архітектури з метою ідентифікації факторів, що сповільнюють процес розробки, для подальшої оптимізації організаційного процесу такої розробки;

надані рекомендації по максимізації ефективності процесу розробки для кожного архітектурного підходу можуть бути безпосередньо використані керівниками програмних проєктів при плануванні складу команд, розподілі ресурсів та прийнятті рішень щодо географічного розташування розподілених команд розробки.

Впровадження результатів дослідження в практику підтверджується довідками ТОВ “Візеვენ Європа” та “Sana Commerce” (довідки додаються).

Особистий внесок здобувача. Усі основні результати дисертаційного дослідження, які представлені до захисту, одержані автором особисто (звіт подібності сформований за допомогою програми StrikePlagiarism додається).

У працях, опублікованих у співавторстві, автору належать наступні результати. В роботах [1] та [9] здобувачем було проаналізовано історію внесення змін до вихідного коду проєктів, досліджено закономірності розподілу частот модифікації модулів та запропоновано алгоритм моделювання зв'язності модулів при симуляції внесення змін в кодову базу. У роботах [2] і [8] здобувачем було досліджено історичні дані проєктів Apache Software Foundation, апроксимовано імовірнісний закон розподілу розміру задач та побудовано адаптований алгоритм генерації випадкових задач з визначеним розподілом їх розміру. В роботі [3] здобувачем запропоновано підхід до моделювання процесу рецензування коду в агентно-орієнтованих симуляціях розробки ПЗ з врахуванням модульної структури кодової бази, виду власності програмного коду, часових поясів та людських факторів. В роботі [4] здобувачем досліджено та описано вплив мікросервісної архітектури на різні аспекти процесу розробки програмного забезпечення. В роботах [5], [6] і [7] здобувачем проведено аналіз минулих наукових досліджень та сформульовано висновки.

Апробація результатів дисертації. Основні результати наукових досліджень неодноразово доповідалися на ряді міжнародних та Всеукраїнських наукових конференціях та семінарах, зокрема: III Міжнародній науково-практичній конференції "Інформаційні технології в освіті та науці" (м. Мелітополь, 25-26 травня 2023 р.), Cybersecurity Providing in Information and Telecommunication Systems II (м. Київ, 26 жовтня 2024 р.), VI Міжнародній науково-практичній конференції молодих вчених, аспірантів і студентів "Сучасні інформаційні технології та системи в управлінні" (м. Київ, 10-11 квітня 2025 р.), V Всеукраїнській науково-технічній конференції "Комп'ютерні технології: інновації, проблеми, рішення" (м. Житомир, 01-02 грудня 2022 р.), Всеукраїнській науково-практичній інтернет-конференції здобувачів вищої освіти та молодих вчених "Сучасні комп'ютерні системи та мережі в управлінні"

(м. Херсон, 30 листопада 2023 р.), Всеукраїнській науково-практичній on-line конференції здобувачів вищої освіти і молодих учених, присвяченої Дню науки, "Тенденції розвитку технологій в автоматизації, електроенергетиці, приладобудуванні та робототехніці" (м. Житомир, 13 травня 2024 р.).

Публікації. Результати дослідження викладено у 9 наукових працях, з них 3 статті у фахових виданнях, включених до переліку наукових фахових видань України категорії «Б», та 6 у матеріалах науково-технічних конференцій, з яких 1 публікація у матеріалах міжнародної наукової конференції, проіндексованої у базі даних Scopus.

Структура та обсяг дисертації. Дисертаційна робота складається з анотації, вступу, трьох розділів, висновків до кожного розділу, загальних висновків, списку використаних джерел (102 найменування, з них 88 іноземною мовою), 9 додатків. Загальний обсяг дисертації – 243 сторінки, з них 151 сторінка основного тексту, у якому міститься 38 таблиць та 31 рисунок.

РОЗДІЛ 1. Теоретичні аспекти моделювання процесів розробки програмного забезпечення та оцінки їх ефективності

1.1. Аналіз останніх досліджень у галузі ефективності розробки програмного забезпечення

Розробка програмного забезпечення (ПЗ) залишається одним із найскладніших інженерних процесів, де точне прогнозування термінів, витрат і продуктивності є критично важливим для успіху проєктів. Згідно із сучасними даними, глобальний ринок розробки ПЗ у 2024 році досяг обсягу 737 мільярдів доларів США, а до 2034 року прогнозується зростання до 2,25 трильйона доларів [6]. Водночас відомо, що великі ІТ-проєкти систематично виходять за межі запланованих бюджетів і термінів, що призводить до значних фінансових втрат [7]. В умовах зростаючої складності програмних систем, розподілених команд та різноманітності архітектурних підходів, виникає гостра потреба у науково обґрунтованих методах прогнозування та моделювання процесів розробки [8, 9]. Дослідження останніх років зосереджуються на кількох ключових напрямках: імітаційному моделюванні процесів розробки ПЗ [10, 11], математичних моделях оцінювання трудовитрат [3], а також на емпіричному аналізі впливу архітектурних рішень на продуктивність команд розробки [12, 13, 14].

Проведемо систематизацію сучасного стану академічних досліджень у сфері моделювання процесів розробки ПЗ та оцінки продуктивності процесу розробки з особливим акцентом на дослідження впливу архітектурних стилів на ефективність розробки, ролі організаційних параметрів, таких як розмір команд розробки, кількість розробників, географічний розподіл команд, об'єму та модульної організації кодової бази, а також останніх досягнень щодо симуляційного моделювання процесів розробки програмного забезпечення.

Моделювання процесів розробки програмного забезпечення (Software Process Simulation Modeling, SPSM) – це напрямок досліджень, що використовує імітаційне моделювання для відтворення, аналізу та прогнозування поведінки

процесів розробки ПЗ. Концепцію було запроваджено Абдель-Хамідом Т. (Abdel-Hamid T.) та Маднаком С. (Madnick S.) у їхній піонерській роботі 1991 року [15], а ключові засади класифікації моделей було сформульовано Келлнером М. (Kellner M.), Мадачі Р. (Madachy R.) та Раффо Д. (Raffo D.) у 1999 році [16]. Систематичні огляди літератури, проведені за останні два десятиліття, засвідчують стабільне зростання кількості публікацій у галузі SPSM [17]. Зокрема, Гарсія-Гарсія Х. (García-García J.) та ін. у систематичному огляді, що охопив 8070 публікацій за період 2013–2019 років, відібрали 36 первинних досліджень та встановили, що симуляційне моделювання процесів розробки програмного забезпечення залишається активно досліджуваною темою з різноманітними цілями застосування [10]. Аналіз виявив десять основних цілей SPSM, згрупованих у когнітивні (розуміння процесу), тактичні (прогнозування і планування) та стратегічні (покращення процесу) рівні. Ключовим висновком сучасних оглядів є зміна домінуючих парадигм моделювання. За даними систематичного огляду Джан Х. (Zhang H.) та ін., системна динаміка (System Dynamics, SD) та дискретно-подійне моделювання (Discrete-Event Simulation, DES) становили відповідно 49% і 31% від усіх застосувань SPSM [18]. Проте Гарсія-Гарсія Х. (García-García J.) та ін. відзначають стійку тенденцію до зростання агентно-орієнтованого моделювання (Agent-Based Modelling, ABM) на тлі зниження популярності системної динаміки [10]. Дискретно-подійне моделювання, в свою чергу, зміцнює свої позиції в останні роки. Лю Б. (Liu B.) та ін. у роботі, присвяченій метрикам для моделювання програмних процесів, ідентифікували 131 дослідження, що сукупно включають 1975 необроблених метрик, класифікованих у 21 категорію [19]. Дослідники зазначають, що зовнішні метрики продукту та процесу використовуються у моделюванні рідко, тоді як аналогічні метрики ресурсів широко застосовуються. Перевірка та підтвердження залишається критичною проблемою. За даними Лі Ю. (Li Y.) та ін., 65,8% із 313 досліджених робіт з симуляційного моделювання процесів розробки програмного забезпечення не описують належним чином процес перевірки та підтвердження моделей [11].

Агентно-орієнтоване моделювання є симуляційною парадигмою, що дозволяє моделювати поведінку індивідуальних агентів (розробників, тестувальників, менеджерів) та їх взаємодію у контексті програмного проєкту [20]. Кожен агент має власний набір станів, правил поведінки та характеристик, що робить АВМ особливо придатним для моделювання неоднорідних команд розробки. Класичним прикладом застосування АВМ у симуляційному моделюванні процесів розробки програмного забезпечення є модель розробки ПЗ, описана у роботі Шериф Р. (Cherif R.) та Девідссон П. (Davidsson P.), де агенти-розробники переходять між станами «Розробка», «Відповіді на запитання стажерів» та «Комунікація» відповідно до заданих правил переходу [21]. Такий підхід дозволяє враховувати різну продуктивність досвідчених розробників та стажерів, моделювати вплив навчання на загальну продуктивність команди та аналізувати ефект стохастичних комунікаційних взаємодій. Дослідження компанії AVL продемонструвало практичне застосування АВМ для моделювання фази конструювання у рамках адаптованого Rational Unified Process для географічно розподіленого відділу розробки [22]. Автори запропонували практичну функцію зусиль для оцінки поведінки розробників та показали, що модель може бути калібрована та перевірена з використанням даних, що зазвичай доступні компаніям-розробникам ПЗ. Модель еволюції відкритого програмного забезпечення, описана Сміт Н. (Smith N.), Капілуппі А. (Capiluppi A.) та Фернандес-Раміль Х. (Fernández-Ramil J.), є ще одним важливим прикладом АВМ у SPSM [23]. У цій моделі численні агенти-розробники незалежно та паралельно генерують, розширюють та рефакторять модулі коду. Ключовою особливістю моделі є включення складності програмних модулів як обмежувального фактору продуктивності, придатності ПЗ до вимог та мотивації розробників. Перевірка моделі на чотирьох великих OSS-системах показала, що імітаційні результати відповідають реальним даним за більшістю показників. Серед сучасних інструментів для агентно-орієнтованого моделювання варто відзначити фреймворк «krABMaga», побудований на мові програмування Rust, що

забезпечує високу продуктивність та надійність дискретно-подійного багатоагентного моделювання [24, 25]. Фреймворк переосмислює архітектурні концепції бібліотеки MASON та пропонує оптимізовані структури даних для масштабних симуляцій.

Системна динаміка (SD), запроваджена у контексті симуляційного моделювання процесів розробки програмного забезпечення Абдель-Хамідом Т. (Abdel-Hamid T.) та Маднаком С. (Madnick S.), залишається найпоширенішою парадигмою моделювання [15]. SD-моделі описують процес розробки на макрорівні через потоки та запаси, що дозволяє аналізувати зворотні зв'язки між змінними проєкту: обсягом роботи, кількістю персоналу, рівнем дефектів тощо. Дискретно-подійне моделювання, яке посідає друге місце за поширеністю, розглядає процес розробки як послідовність дискретних подій, таких як створення коду, виявлення дефекту, рецензування коду. Дискретно-подійне моделювання є особливо ефективним для моделювання конкретних фаз розробки та аналізу черг і вузьких місць у процесі. Гібридні моделі, що поєднують SD, DES та ABM, привертають зростаючу увагу дослідників як спосіб більш реалістичного відтворення складних реальних процесів розробки ПЗ. Такі моделі дозволяють, наприклад, використовувати SD для стратегічного планування на рівні проєкту, DES для моделювання потоку завдань, а ABM для відтворення індивідуальної поведінки розробників.

Конструктивна модель вартості (Constructive Cost Model, COCOMO), розроблена Боемом Б. (Boehm B.), залишається однією з найбільш поширених параметричних моделей оцінювання трудовитрат на розробку ПЗ [3]. Модель базується на емпіричних даних реальних проєктів та використовує кількість рядків вихідного коду як основний вхідний параметр. Базова модель COCOMO визначає трудовитрати та тривалість розробки за формулами (1.1) та (1.2):

$$E = a \times L^b, \quad (1.1)$$

$$D = c \times E^d, \quad (1.2)$$

де E – трудовитрати, D – тривалість розробки, а параметри a , b , c , d залежать від типу проєкту. COCOMO класифікує проєкти на три типи: органічні (невеликі

команди, добре зрозумілі задачі), напіввідокремлені (середня складність) та вбудовані (висока складність, жорсткі обмеження). СОСОМО II, опублікована у 2000 році, суттєво розширила базову модель, додавши масштабні фактори (Scale Drivers), фактори трудовитрат (Cost Drivers) та підтримку сучасних практик розробки, включаючи повторне використання коду та компонентний підхід [4]. Модель було калібровано на 161 проєкті та вона враховує 17 факторів вартості, згрупованих у чотири категорії: атрибути продукту, апаратне забезпечення, персонал та проєкт. Серед обмежень моделі СОСОМО дослідники відзначають залежність від точності оцінки розміру майбутнього проєкту у тисячах рядків вихідного коду, що є проблематичним на ранніх етапах, недостатнє врахування архітектурного стилю системи, обмежену придатність для гнучких методологій розробки. Сучасні адаптації даної моделі включають інтеграцію з аналізом функціональних точок та застосування методів машинного навчання для підвищення точності оцінок.

Закон Брукса, сформульований Бруксом Ф. (Brooks F.) в книзі «Міфічний людино-місяць», стверджує, що додавання персоналу до запізненого програмного проєкту лише збільшує затримку [26]. Цей закон базується на двох ключових факторах: час навчання нових учасників та зростання комунікаційного навантаження. Математично комунікаційне навантаження описується кількістю каналів зв'язку, що розраховується за формулою (1.3):

$$C = \frac{n(n-1)}{2}, \quad (1.3)$$

де n – кількість учасників команди. Так, команда з 6 осіб має 15 каналів зв'язку, команда з 12 осіб – 66, а команда з 50 осіб – 1225 каналів [26]. Гантер Н. (Gunther N.) у своїх дослідженнях показав, що закон Брукса є окремим випадком Універсального закону масштабованості обчислень (Universal Scalability Law), який описує залежність продуктивності від кількості виконавців з урахуванням суперечності, узгодженості та паралельності [27]. Брукс оцінює, що частка продуктивної роботи різко знижується зі зростанням команди. Так, команда з 2 осіб може витрачати 70% часу на продуктивну роботу, тоді як команда з 20 осіб – менше 30%. Це математична закономірність, що має прямий вплив на вибір

архітектурного стилю, так як певні архітектурні моделі дозволяють зменшити потребу в комунікації між командами.

Програма дослідження DORA (DevOps Research and Assessment), що є частиною Google Cloud, протягом понад десяти років досліджує можливості, практики та метрики високопродуктивних команд. Звіт DORA 2024 охопив більше 39000 професіоналів із організацій різного розміру по всьому світу [28]. Початково DORA запропонувала чотири ключові метрики продуктивності розробки програмного забезпечення: частота розгортань, час виконання змін, частота збоїв при змінах та час відновлення після збоїв. У 2024 році фреймворк було суттєво оновлено шляхом додавання нової метрики «частка незапланованих доопрацювань», а метрику «час відновлення після збоїв» перейменовано на «час відновлення після невдалого розгортання» та перенесено з категорії стабільності до категорії пропускну здатності [28]. Звіт DORA 2025 запровадив сім архетипів команд замість попередньої чотирьохрівневої класифікації. Ключовим висновком є те, що ШІ-інструменти прискорюють індивідуальну продуктивність, але це прискорення може оголити слабкі місця у процесах, так як без надійного автоматизованого тестування, зрілих практик контролю версій та швидких циклів зворотного зв'язку, зростання обсягу змін до програмної системи призводить до нестабільності їх роботи та розгортання [29].

Монолітна архітектура передбачає побудову всіх компонентів програми, таких як інтерфейс користувача, бізнес-логіка та рівня доступу до даних, як єдиної цілісної кодової бази з єдиною базою даних. Цей підхід має низку переваг для продуктивності розробки на початкових етапах, такі як простота розробки та тестування, відсутність мережових затримок між компонентами, спрощене розгортання, налагодження та підтримка [30]. За емпіричними даними 2024-2025 років, експерти галузі досягли консенсусу, що для команд розміром менше 10 розробників монолітна архітектура забезпечує вищу продуктивність розробки програмного забезпечення. Проте зі зростанням системи монолітна архітектура стикається з серйозними обмеженнями, оскільки будь-яка зміна програмного коду потребує повторного розгортання всієї програмної системи, що уповільнює

цикли її випуску, кодова база стає складною для розуміння та супроводу, тоді як масштабування потребує збільшення ресурсів для всієї програми, а не лише для навантажених компонентів [31].

Мікросервісна архітектура розділяє програму на невеликі незалежні сервіси, кожен з яких реалізує окрему бізнес-функцію, взаємодіє з іншими через програмні інтерфейси (API, Application Programming Interface) та розгортається окремо від інших. За даними Software AG, вже на момент 2021 року приблизно 81% організацій вже впровадили мікросервіси, а 95% вважають, що мікросервіси є або будуть надзвичайно чи дуже важливими для діяльності їхніх організацій [32]. Емпіричні дослідження демонструють вимірювані покращення продуктивності: організації, що перебудували структуру навколо сервісних меж, повідомили про 64% покращення частоти розгортань та 68% зменшення часу виконання змін. Крос-функціональні команди, що мають повну власність над сервісом, демонструють 42% вищу продуктивність та 27% нижчу частоту дефектів порівняно з функціонально-орієнтованими організаційними структурами. Дослідження Маллампаті Б. (Mallampati B.) встановило, що організації з мікросервісною програмною архітектурою мають на 52% швидше входження нових розробників в процес розробки та на 47% покращену якість коду [1]. Водночас, мікросервісна архітектура має суттєві недоліки, що впливають на продуктивність, такі як зростання складності управління розподіленою системою, необхідність потужної інфраструктури розгортання та проблеми з узгодженістю даних між сервісами [33, 34]. За даними опитування O'Reilly 2024, 61% підприємств впровадили мікросервісну архітектуру, але 29% повернулися до монолітної архітектури через мікросервісну складність [12]. Gartner зазначає, що організації з невідповідною архітектурою мають на 40% вищі витрати на розробку та на 60% довший час виходу на ринок [13]. 90% команд, що перейшли на мікросервісну архітектуру, продовжують виконувати пакетне розгортання подібно до монолітних програмних систем, що нівелює основну архітектурну перевагу мікросервісів. Це свідчить про те, що перехід між

архітектурними моделями потребує не лише технічних, але й організаційних змін [35].

Модульна монолітна архітектура є гібридним підходом, що поєднує операційну простоту моноліту з модульністю та легшим супроводом, як у мікросервісів. Су Р. (Su R.) та Лі Х. (Li X.) у систематичному огляді літератури визначили модульний моноліт як архітектурний стиль, де програма складається з чітко визначених модулів, але розгортається як єдиний блок [2]. Систематичний огляд літератури Аль-Кора'н Л. (Al-Qora'n L.) та Аль-Саїд Ахмад А. (Al-Said Ahmad A.) є першим оглядом щодо модульної монолітної архітектури у хмарних середовищах. Із 369 знайдених записів було відібрано 15 первинних досліджень за період з 2020 по 2025 роки [36]. Огляд виявив непослідовне використання термінології у літературі, ключові фактори впровадження включають спрощене розгортання, простіший супровід та зменшення витрат на оркестрування, перша наукова стаття з модульної монолітної архітектури була опублікована у 2021 році, а у 2023 спостерігалось різке зростання публікацій. Практичні приклади впровадження модульної монолітної архітектури включають такі проекти, як Shopify, що деконструювала свій моноліт у модульну архітектуру для максимізації продуктивності розробників, Google Service Weaver, що дозволяє писати код як у модульній монолітній архітектурі та розгортати модулі як мікросервіси, фреймворки Spring Modulith та Light-hybrid-4j [2]. Барде К. (Barde K.) у дослідженні фінтех-сектору показав, що модульна монолітна архітектура забезпечує значні покращення порівняно з традиційною монолітною або мікросервісною в контексті високонавантажених додатків з інтенсивним використанням даних [24]. За даними звіту DORA 2024, високопродуктивні команди, що використовують модульні архітектури, здійснюють розгортання коду у 973 рази частіше з 5-кратно нижчою частотою збоїв при змінах і відновлюють сервіси у 6570 разів швидше при інцидентах [28].

Тапія Ф. (Tapia F.) та ін. у порівняльному дослідженні продуктивності застосували непараметричну регресійну математичну модель для пояснення залежностей між змінними продуктивності монолітних та мікросервісних систем

[37]. Результати підтвердили, що вибір архітектури безпосередньо впливає на ефективність управління ресурсами та операційну ефективність програмного забезпечення.

Дослідження продуктивності команд розробки стабільно вказують на оптимальний розмір команди із 5-9 розробників. За даними Worklytics, малі команди демонструють на 15% більше годин зосередженої роботи (4,8 проти 4,2 години на день), на 20% вищу швидкість обробки запитів на рецензування коду (14,9 проти 12,4 запитів/місяць), на 25% менший час на виконання змін (2,9 проти 3,8 днів) [38]. Великі організації стикаються з системними викликами продуктивності через складність координації, так як години зосередженої роботи знижуються на 10% відносно медіани (3,8 проти 4,2 години/день) [38]. Це узгоджується із законом Брукса, оскільки кількість комунікаційних каналів має квадратичне зростання і частка продуктивної роботи зменшується [26]. Усталені команди зазвичай працюють на 20-30% швидше, ніж новосформовані.

Географічний розподіл команд розробки є одним із найважливіших факторів, що впливають на продуктивність [39, 40, 41]. Дослідження AVL продемонструвало практичне застосування агентно-орієнтованого моделювання для географічно розподіленого відділу розробки, що підтверджує актуальність цієї проблематики [22]. Розподілені команди потребують структурованої комунікації та перекриття робочих годин для підтримки швидкості розробки [42]. Мікросервісна архітектура природно підтримує розподілену розробку, оскільки кожна команда може незалежно працювати над окремим сервісом. Закон Конвея безпосередньо пов'язує географічну структуру організації зі структурою програмного продукту, а дослідження Harvard Business School виявило, що продукти розподілених команд є значно більш модульними [43]. Використання сучасних технологій скорочує час розробки приблизно на 40%, що частково компенсує втрати продуктивності від географічного розподілу команд. Проте відстань між часовими поясами, культурні відмінності та мовні бар'єри залишаються суттєвими факторами, що потребують додаткового моделювання [44].

Застосування практики розподіленої власності над окремими частинами програмного коду є критичним фактором продуктивності, що тісно пов'язаний з архітектурним стилем. У мікросервісній архітектурі кожна команда повністю володіє своїм сервісом від розробки до експлуатації, що збільшує відповідальність та зменшує затримки комунікації. Дослідження Нагаппан Н. (Nagappan N.), Мерфі Б. (Murphy B.) та Базілі В. (Basili V.) (Університет Меріленду спільно з Microsoft) підтвердили зв'язок між організаційною структурою, власністю програмного коду та якістю програмного забезпечення [45]. Модульні архітектури полегшують визначення меж володіння: у модульному моноліті модулі мають чіткі інтерфейси, а у мікросервісах кожен сервіс є одиницею володіння. Кількість модулів та загальний обсяг кодової бази впливають на продуктивність нелінійно. Модель СОСОМО описує цю залежність через показник ступеня b у формулі (1.1) для складних систем, що означає експоненційне зростання трудовитрат зі збільшенням об'єму вихідного коду [3, 24]. Модульна декомпозиція дозволяє зменшити ефективне значення цього показника для окремих команд.

Проведений аналіз літератури засвідчує, що моделювання процесів розробки програмного забезпечення залишається активною та динамічною областю наукових досліджень. Протягом останніх двох десятиліть спостерігається стійка тенденція до зростання кількості публікацій, а серед парадигм моделювання дедалі більшої популярності набуває агентно-орієнтований підхід. На відміну від традиційної системної динаміки, яка описує процес розробки на макрорівні, агентно-орієнтоване моделювання дозволяє відтворювати індивідуальну поведінку розробників, враховувати неоднорідність команд та моделювати стохастичні комунікаційні взаємодії. Це робить агентно-орієнтоване моделювання особливо перспективним інструментом для дослідження впливу організаційних параметрів на продуктивність розробки. Класичні математичні моделі оцінювання трудовитрат, зокрема СОСОМО та закон Брукса, зберігають свою фундаментальну цінність як теоретична основа для розуміння залежностей між параметрами проєкту та продуктивністю [46].

Формула комунікаційного навантаження залишається ключовим інструментом для обґрунтування рішень щодо розміру команд, а параметрична модель СОСОМО продовжує використовуватися для початкових оцінок трудовитрат. Водночас ці моделі потребують суттєвої адаптації до сучасних реалій, оскільки вони недостатньо враховують вплив архітектурного стилю системи, погано пристосовані до гнучких методологій розробки та не відображають особливості розподілених команд [47]. Інтеграція класичних параметричних моделей із методами машинного навчання є одним із найбільш перспективних напрямків їхнього розвитку [48].

Аналіз літератури переконливо демонструє, що вибір архітектурного стилю має вимірюваний та статистично значущий вплив на продуктивність розробки [14, 30, 33]. Проте цей вплив не є однозначним і суттєво залежить від таких факторів, як розмір команди, складність системи, вимоги до масштабованості та рівня організаційної зрілості. Емпіричні дані свідчать, що для команд менше десяти розробників монолітна архітектура забезпечує вищу продуктивність, тоді як мікросервіси починають давати переваги лише при більших командах та складніших системах. Організації, що перебудовують структуру навколо мікросервісної архітектури, повідомляють про суттєве покращення частоти розгортань та зменшення часу виконання змін, однак майже третина підприємств, що впровадили мікросервіси, повернулася до монолітних архітектур через непередбачувану складність [35]. Модульно-монолітна архітектура є відносно новим, але швидко зростаючим напрямком як у наукових дослідженнях, так і в промисловій практиці. Модульний моноліт пропонує привабливий компроміс: він зберігає операційну простоту єдиного розгорнутого блоку, водночас забезпечуючи модульність та чіткі межі між компонентами. Практичний досвід таких компаній, як Shopify, а також поява фреймворків на кшталт Google Service Weaver та Spring Modulith підтверджують життєздатність цього підходу. Модульний моноліт може також слугувати проміжним етапом на шляху до повноцінної мікросервісної архітектури.

Результати програми дослідження DORA за 2024–2025 роки привносять додатковий вимір до розуміння продуктивності розробки. Ключовим висновком є те, що стабільність впровадження програмного забезпечення є критичнішою за швидкість розробки: зростання пропускної здатності без належних механізмів контролю якості призводить до нестабільності. Це спостереження безпосередньо стосується вибору архітектури, оскільки різні архітектурні стилі по-різному впливають на баланс між швидкістю та стабільністю. Крім того, впровадження ШІ-інструментів у процес розробки посилює як сильні, так і слабкі сторони існуючих процесів, що робить зрілість інженерних практик ще більш важливою.

На основі проведеного аналізу можна окреслити кілька пріоритетних напрямків подальших досліджень. Насамперед, існує потреба у розробці математичних моделей продуктивності, які б явно враховували архітектурний стиль як вхідний параметр, включаючи визначення порогових значень для переходу між архітектурами залежно від розміру команди та складності системи. Важливим напрямком є також застосування агентно-орієнтованого моделювання для порівняльного аналізу ефективності різних архітектурних стилів за різних організаційних конфігурацій з варіюванням кількості та розміру команд, географічного розподілу та виду власності коду.

1.2. Структура процесу розробки програмного забезпечення

Розробка програмного забезпечення є складним інженерним процесом, який об'єднує творчу, аналітичну та організаційну діяльність великої кількості фахівців різних спеціалізацій. Цей процес може видаватися непрозорим або навіть хаотичним, проте насправді він ґрунтується на чітко визначених принципах, етапах і практиках, які у сукупності називають життєвим циклом розробки програмного забезпечення (Software Development Life Cycle, SDLC) [49]. На рис. 1.1 наведено загальну схему, яка ілюструє послідовність та взаємозв'язок усіх етапів цього процесу.

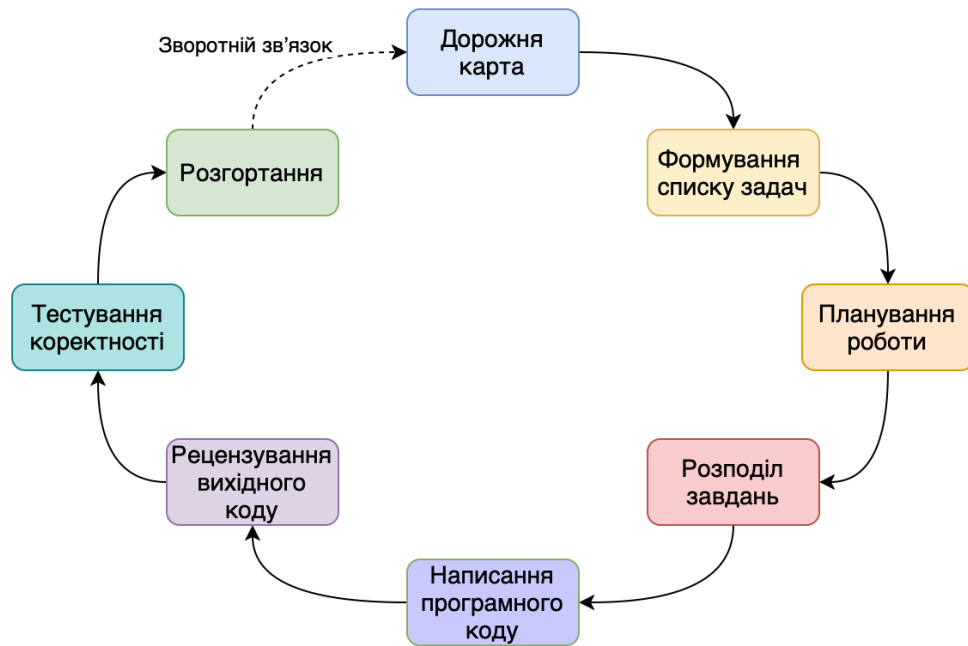


Рис. 1.1. Загальна схема життєвого циклу розробки програмного забезпечення

Будь-який професійний програмний продукт починається не з написання вихідного коду, а з формування стратегічного бачення, яке в індустрії називають продуктовою дорожньою картою (Product Roadmap) [7]. Це документ або візуальне подання, що описує загальний напрямок розвитку програмного продукту на найближчі місяці чи навіть роки. За своєю природою дорожня карта нагадує генеральний план розвитку міста: вона визначає, які райони будуть забудовані першими, де з'являться парки, а де промислові зони, але не містить креслень окремих будівель. Дорожню карту зазвичай формує продуктова команда, до складу якої входять менеджер продукту, бізнес-аналітики та представники вищого керівництва компанії. Вони аналізують потреби ринку, відгуки користувачів, конкурентне середовище та бізнес-цілі організації. На виході з'являється пріоритизований перелік великих функціональних блоків, які також називають ініціативами (від англійського «initiative»). Кожен такий блок описує значущу можливість продукту, наприклад, інтеграцію з платіжною системою, впровадження нового типу звітності або створення мобільного застосунку. Важливо розуміти, що дорожня карта є живим документом, який регулярно переглядається та коригується залежно від нових обставин, що робить

процес розробки адаптивним, а не жорстко детермінованим. Приклад візуальної дорожньої карти наведено на рис. 1.2.



Рис. 1.2. Продуктова дорожня карта з розподілом ініціатив за кварталами

Коли стратегічний напрямок визначено, наступним кроком стає деталізація великих ініціатив у конкретніші одиниці роботи. Цей процес називають формуванням детального впорядкованого переліку завдань для виконання (Product Backlog) [50], подібно до кошторису та специфікації в будівництві. Кожен елемент переліку завдань зазвичай оформлюється як так звана користувачка історія (User Story), тобто опис бажаної функціональності з точки зору кінцевого користувача. Наприклад, замість технічного формулювання на кшталт «реалізувати API для автентифікації» команда записує: «Як користувач, я хочу мати змогу увійти до системи за допомогою свого облікового запису Google, щоб не запам'ятовувати окремий пароль». Такий підхід забезпечує те, що технічна робота завжди прив'язана до реальної цінності для людини, яка користуватиметься даним програмним продуктом. Менеджер продукту спільно з технічними спеціалістами постійно працює над пріоритизацією черги завдань, визначаючи, які користувачькі історії є найважливішими з точки зору бізнесу, а які можуть зачекати. Цей процес є ітеративним і передбачає регулярні зустрічі, під час яких команда обговорює,

уточнює та оцінює складність кожного завдання. На рис. 1.3 зображено приклад пріоритизованої черги завдань.

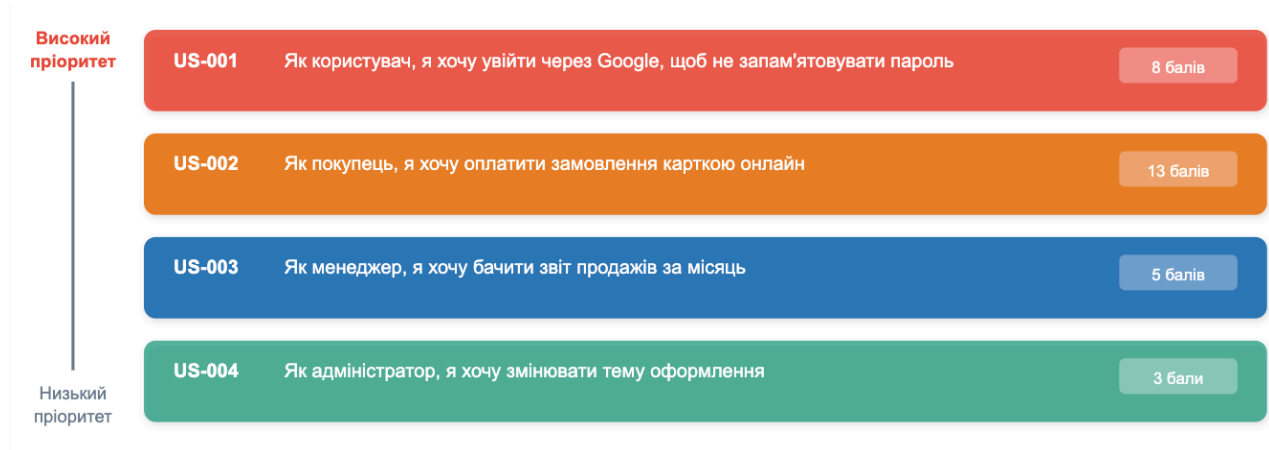


Рис. 1.3. Черга завдань з пріоритизованими користувацькими історіями

Сучасні програмні продукти рідко створюються однією невеликою групою розробників в одному офісі. Як правило, над складним продуктом працюють кілька розподілених команд, які можуть перебувати в різних містах, країнах або навіть часових поясах [26, 45]. Кожна така команда зазвичай складається з розробників – фахівців, що безпосередньо пишуть програмний код, та тестувальників – спеціалістів, відповідальних за перевірку якості створеного коду та відповідності всім поставленим вимогам. Окрім них, у команді може бути Scrum-майстер (Scrum Master), який організовує процес роботи, та власник продукту (Product Owner), який представляє інтереси бізнесу. Процес планування найчастіше організовано за методологією Scrum або Kanban [50]. У випадку Scrum робота поділяється на фіксовані проміжки часу, які називають ітераціями, тривалістю зазвичай від одного до чотирьох тижнів. На початку кожної ітерації відбувається зустріч з планування, під час якої команда обирає з черги завдань ті користувацькі історії, які вона зобов'язується виконати протягом наступної ітерації, відповідно до виставлених пріоритетів. Далі кожна користувацька історія розбивається на конкретні технічні завдання. Наприклад, одна користувацька історія про авторизацію через обліковий запис Google може перетворитися на кілька завдань: створення користувацького інтерфейсу кнопки входу, налаштування взаємодії із сервісом автентифікації Google, написання

серверної логіки обробки токенів авторизації, створення автоматизованих тестів та оновлення документації. Ці завдання розподіляються між членами команди з урахуванням їхньої спеціалізації та поточного навантаження [51, 52]. Коли над одним продуктом працює декілька команд одночасно, виникає потреба у координації. Для цього проводяться регулярні міжкомандні зустрічі-синхронізації, під час яких представники кожної команди повідомляють про свій прогрес, обговорюють залежності між завданнями різних команд та вирішують потенційні конфлікти [45]. Без такої координації різні частини продукту можуть виявитися несумісними між собою, подібно до того, як стіни двох будівельних бригад, що працюють з різних боків, можуть не зійтися в одній точці [26]. На рис. 1.4 зображено приклад планування та розподілу роботи між різними командами розробки.

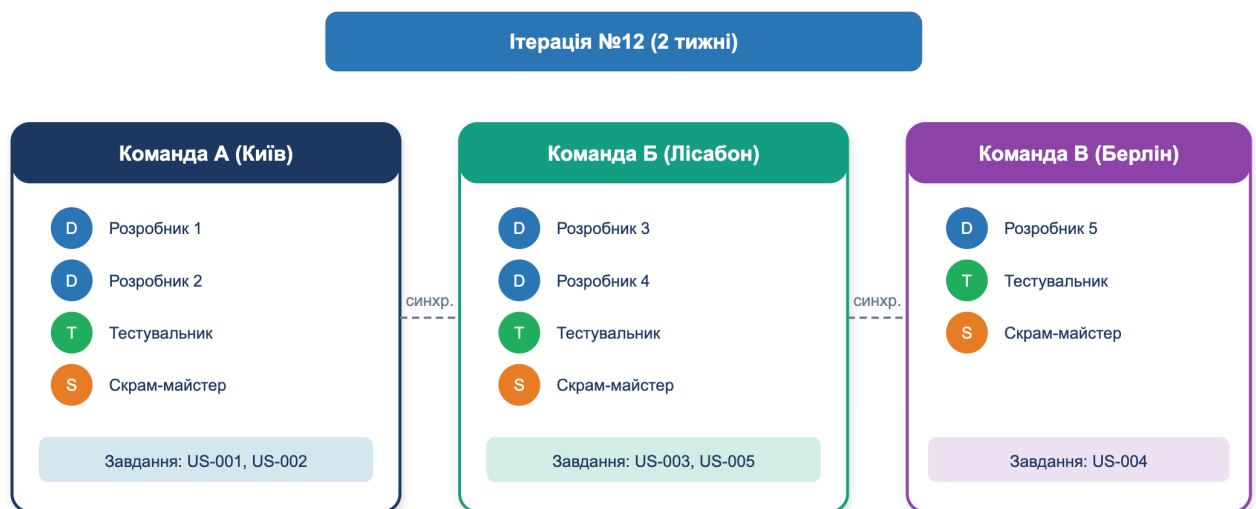


Рис. 1.4. Розподіл запланованої роботи між різними командами

Після завершення планування розробники переходять до безпосереднього створення програмного коду, що є центральною виробничою фазою всього процесу. Сучасна розробка програмного забезпечення спирається на систему контролю версій, найпоширенішою з яких є Git. Систему контролю версій можна порівняти з бібліотечним каталогом, який зберігає не лише поточний стан кожного документа, а й повну історію всіх змін, внесених будь-коли. Завдяки цьому різні розробники можуть одночасно працювати над різними частинами

коду, не заважаючи один одному. Робота над кожним окремим завданням починається зі створення так званої гілки (Branch) в системі контролю версій – ізольованої «копії» кодової бази, в якій розробник може вносити свої зміни, не впливаючи на основну версію продукту. Це подібно до того, як редактор працює над окремою копією рукопису, не змінюючи оригінал, доки правки не будуть узгоджені. Розробник поступово втілює технічне рішення, яке було обговорене на етапі планування, пишучи програмний код відповідною мовою програмування та використовуючи інструменти інтегрованого середовища розробки (IDE, Integrated Development Environment). Цей процес є не лише механічним, а й творчим: програміст приймає численні архітектурні рішення щодо того, як саме реалізувати ту чи іншу функціональність, прагнучи забезпечити не лише правильну роботу коду, але і його читабельність, швидкодію та простоту подальшої підтримки. Паралельно з написанням основного коду розробник зазвичай створює модульні тести (Unit Tests) – невеликі програми-перевірки, які автоматично підтверджують, що кожна окрема частина коду працює відповідно до поставлених вимог. На рис. 1.5 зображена паралельна робота декількох розробників над власними завданнями з використанням системи гілок Git.

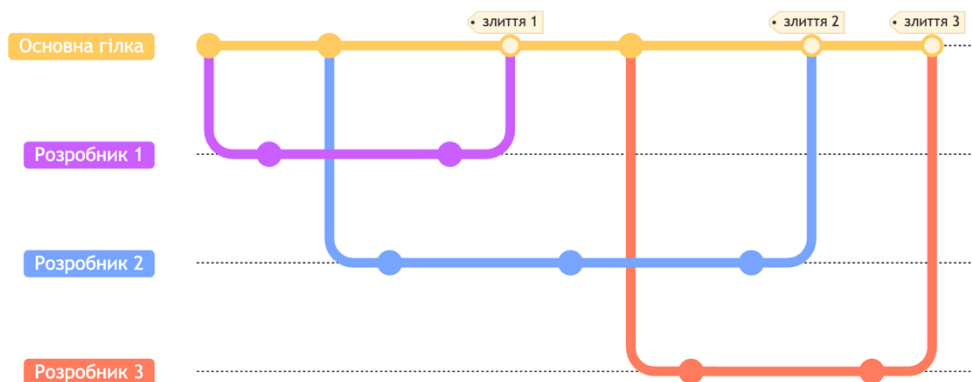


Рис. 1.5. Система гілок у Git: паралельна робота над завданнями

Процеси розробки програмного забезпечення значно відрізняються між монолітною, модульною монолітною та мікросервісною архітектурою, і ці

відмінності охоплюють різні аспекти розробки, включаючи організацію команд, підходи до розподілу задач, програмування, тестування та розгортання.

Програмування в монолітній архітектурі відбувається в єдиному кодовому середовищі, де всі компоненти програми розробляються разом [13, 37]. Це сприяє інтеграції між різними частинами коду, оскільки всі компоненти знаходяться в одному місці. Розробники мають доступ до всього коду, що спрощує управління залежностями та загальну координацію [43]. Однак, це також може призвести до проблем з масштабованістю програмного продукту, оскільки великі кодові бази стають складнішими для підтримки та розширення [53]. Так як в монолітній кодовій базі немає чітких розмежувань між різними функціями системи, а розробники постійно перемикаються в розробці з однієї частини коду на іншу, при зростанні розміру програмної системи когнітивне навантаження на розробників збільшується ще швидше [54]. На практиці, це часом може призводити до страху вносити зміни через ризик непередбачуваних побічних ефектів у віддалених частинах системи, що в результаті призводить до більш швидкого накопичення технічного боргу [43, 55]. Модульна монолітна архітектура передбачає розбиття програми на окремі модулі з чітко визначеними межами. Це дозволяє розробникам працювати над окремими частинами програми більш автономно, зберігаючи при цьому переваги єдиної кодової бази. Модулі мають чітко визначені інтерфейси, що знижує ризик виникнення проблем із залежностями. Кодова база все ще єдина, але модульність дозволяє краще працювати з окремим модулем ізольовано, сприймаючи інші модулі як «чорні скриньки» або сторонні бібліотеки з чітко визначеними програмними інтерфейсами, що приховують їх внутрішню складність. У мікросервісній архітектурі кожен сервіс є окремим компонентом із власною кодовою базою. Це дозволяє командам працювати над різними сервісами незалежно, використовуючи різні технології, підходи та мови програмування. Кожен мікросервіс має свій життєвий цикл, що дозволяє впроваджувати зміни та оновлення без впливу на інші частини системи. Це підвищує гнучкість, але також ускладнює управління залежностями між сервісами [56, 57]. В порівнянні з

модульними типами архітектури, в даному контексті складність зміщується з бізнес-логіки на інфраструктуру, мережеві взаємодії, узгодженість даних та розподілені транзакції, що вимагає додаткових витрат часу та зусиль на їх налаштування, підтримку та обслуговування [1]. Оскільки для мікросервісної архітектури притаманна сильна власність коду [58, 59, 60, 61], а для імплементації окремої функції системи може знадобитись внесення змін в декілька різних сервісів, то це може призводити до затримок завершення роботи однієї команди розробки через очікування іншої команди для внесення необхідних змін в програмний код мікросервіса, від якого залежить перший. Така сама проблема може виникати і при використанні модульної монолітної архітектури у зв'язці з сильною власністю коду.

Одним із найважливіших механізмів забезпечення якості у сучасній розробці програмного забезпечення є процес рецензування вихідного коду (Code Review) [62]. Після того як розробник завершив роботу над своїм завданням і переконався, що написаний код проходить всі модульні тести, він створює так званий запит на злиття (Pull Request або Merge Request), що є першим етапом процесу рецензування. Це формальна пропозиція включити його зміни до основної версії продукту, яка автоматично стає доступною для перегляду іншими членами команди. Процес рецензування програмного коду можна порівняти з рецензуванням наукової статті перед публікацією. Один або декілька колег ретельно аналізують запропоновані зміни, звертаючи увагу на правильність реалізації, дотримання архітектурних стандартів, потенційні помилки, безпеку, швидкодію та зрозумілість коду [63, 64, 65]. Рецензент залишає коментарі, ставить запитання та може запропонувати альтернативні рішення. Автор коду відповідає на коментарі, вносить виправлення та повторно подає роботу на перевірку. Цей діалог може тривати протягом декількох ітерацій, поки рецензент не дасть своє схвалення. Рецензування програмного коду виконує одразу кілька функцій: воно виявляє помилки на ранньому етапі, забезпечує поширення знань між членами команди (адже кожен рецензент дізнається про новий код, написаний колегою), підтримує єдність стилю та

стандартів у кодовій базі та запобігає ситуаціям, коли лише одна людина розуміє певну частину програмної системи [62]. У географічно розподілених командах рецензування програмного коду часто відбувається асинхронно: розробник у Києві може створити запит на злиття з основною гілкою ввечері, а його колега у Лісабоні прорецензує його наступного ранку, що дозволяє ефективно використовувати різницю у часових поясах [66]. Схематично процес рецензування коду зображений на рис. 1.6.

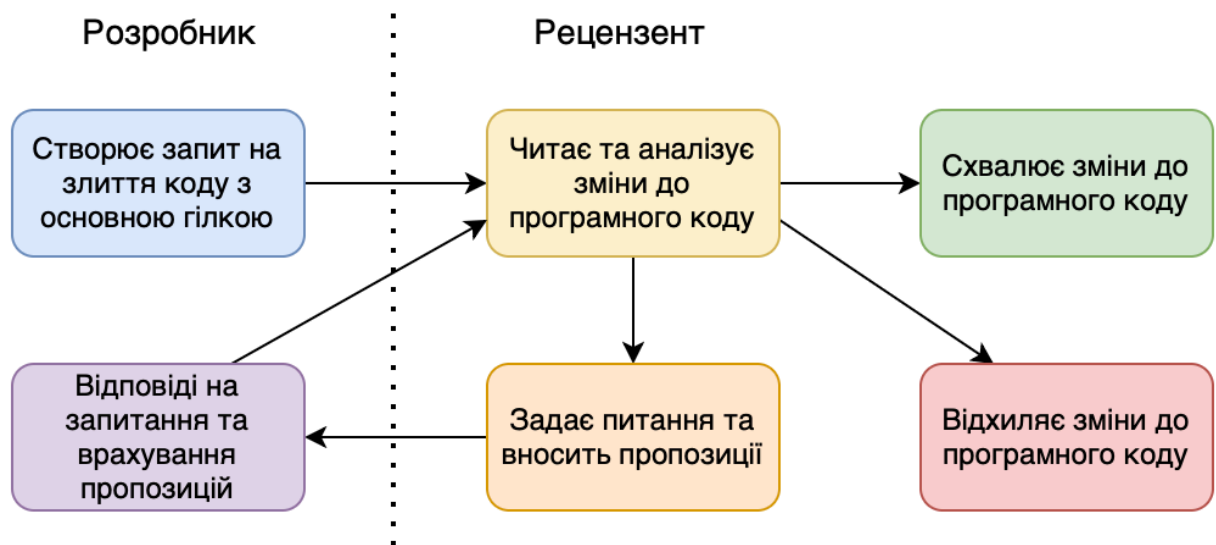


Рис. 1.6. Процес рецензування коду

Після схвалення запиту на злиття від всіх необхідних рецензентів, але до самого злиття змін до коду з основною гілкою, зазвичай відбувається етап тестування внесених змін. Пріоритетність рецензування перед тестуванням аргументується тим, що у випадку виявлення суттєвих недоліків в запропонованих змінах до програмного коду, автор змін може докорінно переробити своє технічне рішення, що призведе до марнування часу тестувальника на повторне тестування тієї ж задачі, а також знижує якість процесу тестування на кожній додатковій ітерації.

Загалом, процес тестування у розробці програмного забезпечення є не одноразовою подією, а безперервним багаторівневим процесом, який супроводжує весь життєвий цикл продукту [49]. Найбільш базовим рівнем є вже

згадані модульні тести, які перевіряють окремі компоненти коду ізольовано. Наступним рівнем є інтеграційне тестування, яке перевіряє, чи правильно взаємодіють між собою різні компоненти системи, що подібно до того, як у будівництві перевіряють, чи коректно електрична система працює разом із системою водопостачання. На вищому рівні знаходиться системне тестування, під час якого спеціалісти з тестування (тестувальники) перевіряють програмний продукт як цілісну систему, імітуючи реальні сценарії використання. Тестувальники виконують як заздалегідь підготовлені тестові сценарії, так і проводять дослідницьке тестування, під час якого вони намагаються знайти неочевидні помилки, діючи нестандартним чином, наприклад, натискаючи кнопки у несподіваній послідовності, вводячи незвичні дані або використовуючи програмний продукт в умовах, які не були передбачені розробниками. Окрім функціонального тестування, існують також спеціалізовані види перевірок, зокрема навантажувальне тестування (яке визначає, скільки одночасних користувачів може витримати система), тестування безпеки (яке виявляє вразливості, через які зловмисники можуть отримати несанкціонований доступ), тестування сумісності (яке перевіряє роботу продукту на різних пристроях та у різних браузерах) та регресійне тестування (яке гарантує, що нові зміни не зламали раніше працюючу функціональність). Значна частина тестування у сучасних командах автоматизована: спеціальні програми-скрипти запускаються при кожній зміні вихідного коду і протягом лічених хвилин виконують сотні чи навіть тисячі перевірок, що було б неможливо зробити вручну за прийнятний час. В індустрії розробки програмного забезпечення прийнято зображати структуру процесу забезпечення якості у вигляді піраміди тестування, що зображена на рис. 1.7, де в основі лежать найшвидші модульні тести, над ними розміщені повільніші та в меншій кількості інтеграційні тести, а на вершині знаходяться найповільніші та найбільш реалістичні системні тести, кількість яких зазвичай невелика через їх повільне виконання та дороговартісну підтримку.

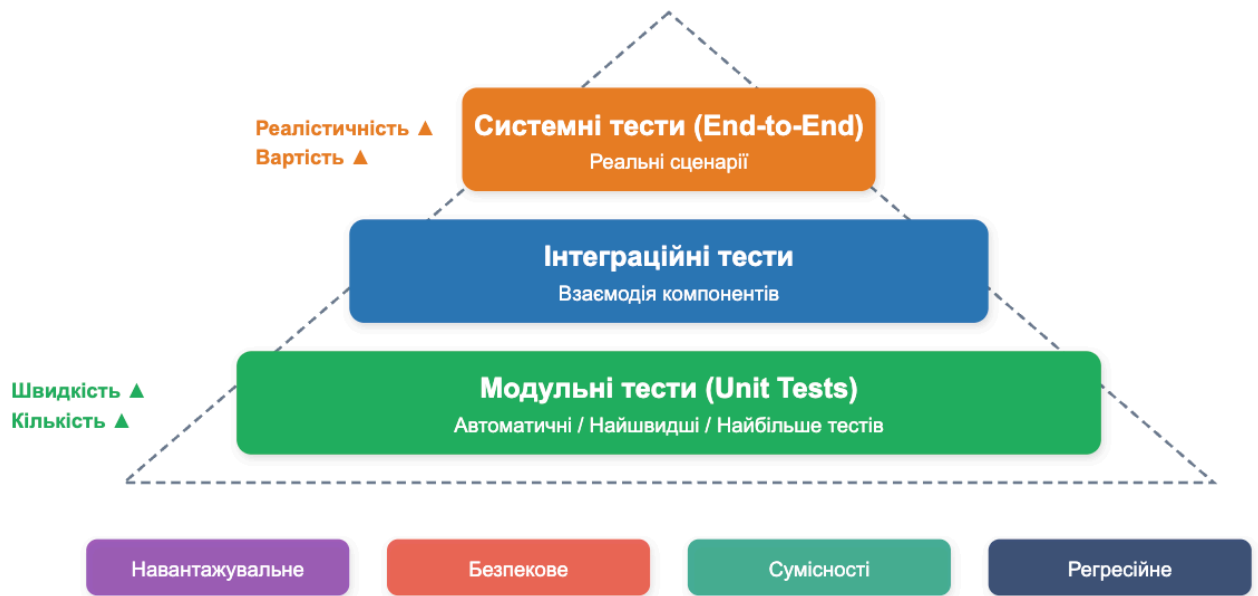


Рис. 1.7. Піраміда тестування та спеціалізовані види тестів

В контексті різних типів архітектури програмного забезпечення процес тестування також має свої особливості. Тестування в монолітній архітектурі здійснюється для всієї програми одночасно, що спрощує інтеграційне тестування, але через непередбачуваність внутрішніх взаємозв'язків між різними блоками коду вимагає повторної перевірки більшої кількості функцій системи [13, 37]. З одного боку, усі компоненти системи можна тестувати в єдиному середовищі, що є зручнішим та знижує ризик виникнення проблем із сумісністю, а з іншого, це може призвести до тривалого часу тестування, особливо для великих кодових баз. У модульній монолітній архітектурі тестування можна проводити як для всієї програми, так і для окремих модулів, але це вимагає додаткових інвестицій в розробку такої можливості. Автоматизація тестування окремих модулів дозволяє виявляти проблеми на ранніх етапах розробки, знижує ризик виникнення помилок при інтеграції змін, спрощує процес тестування та покращує якість коду. Мікросервісна архітектура потребує спеціальних підходів до тестування, оскільки кожен мікросервіс розробляється і тестується незалежно [1, 56]. Це включає тестування окремих сервісів, інтеграційне тестування взаємодій між сервісами та контрактне тестування, яке забезпечує сумісність

інтерфейсів. Даний підхід підвищує гнучкість тестування, але також ускладнює його через необхідність врахування великої кількості взаємодій між сервісами та необхідність проведення тестування швидкодії, безпеки, масштабованості, надійності окремо для кожного мікросервіса через вже згадане зміщення складності з бізнес-логіки на інфраструктуру, мережеві взаємодії, узгодженість даних та розподілені транзакції.

Фінальним етапом кожного циклу розробки є розгортання програмної системи (Deployment). Це процес перенесення готового та протестованого програмного забезпечення у виконавче середовище, тобто на сервери та інфраструктуру, де з ним взаємодіятимуть реальні користувачі. Цей етап можна порівняти з днем відкриття нового крила музею після завершення будівельних робіт і перевірок. У сучасній практиці розгортання здійснюється через конвеєри безперервної інтеграції та безперервного розгортання (CI/CD, Continuous Integration / Continuous Deployment). Конвеєр – це автоматизована послідовність кроків, яка починається з моменту, коли програмний код потрапляє до основної гілки системи контролю версій, і включає компіляцію коду, запуск усіх автоматизованих тестів, перевірку відповідності стандартам якості та, за умови успішного проходження всіх етапів, автоматичне або напівавтоматичне розгортання на серверах. Багато організацій використовують поступове розгортання (Gradual Rollout), коли нова версія продукту спочатку стає доступною лише невеликому відсотку користувачів, що дозволяє виявити потенційні проблеми в реальних умовах до того, як вони торкнуться всієї аудиторії користувачів. Якщо протягом визначеного періоду спостереження не виявляє критичних проблем, охоплення поступово збільшується до ста відсотків, а якщо ж виникає серйозна помилка, то розгортання нової версії програмної системи зупиняється, а сервери автоматично відкочується до попередньої стабільної версії. Після успішного розгортання команда продовжує слідкувати за станом системи, відстежуючи показники продуктивності, кількість помилок та реакцію користувачів. Ця інформація замикає цикл розробки, оскільки виявлені проблеми та побажання користувачів повертаються до черги завдань, звідки

починається наступна ітерація. Всі етапи процесу розгортання програмної системи схематично зображені на рис. 1.8.

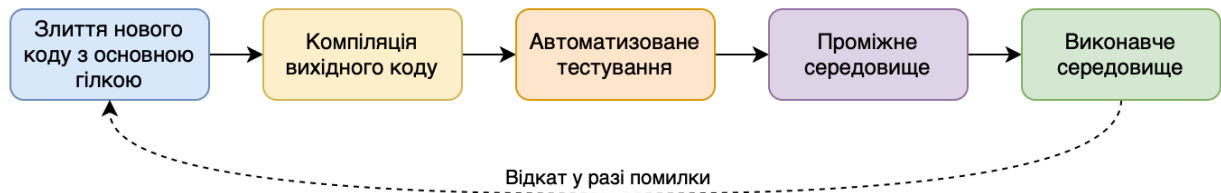


Рис. 1.8. Конвеєр розгортання (CI/CD)

Розгортання в монолітній та модульній монолітній архітектурах здійснюється як єдине ціле, що спрощує процес, але водночас означає, що будь-яка зміна в одному модулі вимагає повторного розгортання всього застосунку [2, 24, 37]. Перевагою процесу розгортання монолітної бази коду є простота його створення і підтримки та легкість повернення до попередньої версії системи у разі виявлення будь-яких проблем [13]. Недоліками є довга тривалість розгортання великої системи та ризик затримки всієї нової версії через виявлену проблему хоча б в одному її модулі. У мікросервісній архітектурі кожен мікросервіс розгортається незалежно, що спрощує процес оновлення та знижує ризики, пов'язані з розгортанням [1]. Це дозволяє впроваджувати зміни та нові функції без впливу на інші частини системи, хоча на практиці через тісну зв'язність на рівні даних або API, оновлення одного сервісу може вимагати узгодженого оновлення інших, перетворюючи систему на "розподілений моноліт", який набагато складніше розгортати, ніж звичайний. Крім того, мікросервісна архітектура також потребує складнішої інфраструктури для управління розгортаннями, включаючи використання інструментів оркестрації, таких як Kubernetes, а також наявність в команді експертизи в галузі DevOps (Development & Operations), що відволікає від продуктової розробки [28, 29].

Процес розробки програмного забезпечення є циклічним і ітеративним [15, 16, 49]. Від формування стратегічного бачення у продуктивній дорожній карті через деталізацію у переліку завдань, планування ітерацій розробки, розподіл

завдань між розподіленими командами розробки, написання та рецензування програмного коду, багаторівневе тестування й аж до розгортання у виконавчому середовищі, де кожен етап логічно впливає з попереднього й створює передумови для наступного. Сучасні методології розробки, зокрема Scrum та Kanban, наголошують на гнучкості, прозорості та постійному вдосконаленні, що дозволяє командам адаптуватися до змінних вимог ринку та потреб користувачів [50]. Розуміння цих етапів є важливим не лише для технічних фахівців, а й для менеджерів, замовників, інвесторів та всіх, хто взаємодіє з процесом створення цифрових продуктів, оскільки воно забезпечує спільну мову для ефективної комунікації та прийняття обґрунтованих рішень [7].

1.3. Дослідження архітектурних факторів впливу на продуктивність розробки ПЗ та метрик оцінки ефективності процесу розробки

Різноманітні технологічні тренди мають великий вплив на сучасні архітектури програмного забезпечення. При виборі архітектурної моделі для нового проєкту в першу чергу враховуються функціональні вимоги до майбутньої програмної системи, після чого розраховується вартість її розробки в контексті вибраного архітектурного підходу. Проблема полягає в тому, що різні архітектурні моделі мають різний вплив на продуктивність розробки програмного забезпечення, що в свою чергу впливає на кінцеву оцінку вартості розробки такої системи. Складність врахування впливу архітектури на продуктивність розробки на етапі вибору архітектурного підходу полягає в нелінійності та багатогранності зв'язків різних параметрів процесу розробки з різними архітектурними рішеннями. Для більш детального дослідження даних зв'язків та можливості порівняльного аналізу різних архітектурних підходів планується застосувати програмне моделювання ефективності розробки програмного забезпечення в залежності від типу архітектури програмних систем. Враховуючи комплексну природу такого впливу, а також для врахування фактору непередбачуваності в процесі розробки, найбільш доцільним буде використання методу агентного моделювання. Для створення відповідних

моделей необхідно провести аналіз факторів впливу на продуктивність розробки програмного забезпечення та їх відношення до різних аспектів певних архітектурних рішень.

Загальний перелік всіх факторів, що впливають на продуктивність розробки програмного забезпечення, можна поділити на чотири умовні категорії: персонал, програмний продукт, організаційні робочі процеси та управління проектом [54]. До факторів, пов'язаних з персоналом, відносяться навички та компетенції спеціалістів, попередній досвід, розмір команд розробки, їх кількість, згуртованість всередині команд, механізми взаємодії між командами, мотивація працівників, плинність кадрів та інші. Серед факторів програмного продукту можна виділити функціональну складність програмної системи, складність коду, архітектурну складність, технологічну платформу, мову програмування, розмір програмної системи, тип програмного продукту, зв'язність між окремими компонентами системи, якість коду та його ремонтпридатність, а також швидкодію програмного продукту. Організаційні процеси роботи містять такі фактори, як робоче середовище, географічний розподіл розробників, політика повторного використання розробок, інструменти та методології розробки, тестування та розгортання програмного забезпечення. До факторів управління проектом відносяться: якість формулювання вимог, якість планування робіт, розподіл роботи між командами, організація процесу управління знаннями тощо [51, 52, 54].

Виділення з повного переліку факторів лише тих, які тим чи іншим чином залежать від вибору архітектури програмної системи, дозволяє виявити наявність зв'язків між певними архітектурними якістьями системи та продуктивністю розробки програмного забезпечення.

В залежності від типу архітектури програмної системи змінюються також і вимоги до навичок та компетенцій розробників, так само як і до їх попереднього досвіду. Наприклад, при виборі мікросервісної архітектури, до базових необхідних компетенцій більшості розробників додаються ще знання по розгортанню та розміщенню веб-сервісів, мережевій безпеці, механізмам

автентифікації між окремими веб-сервісами, а також накладає додаткову відповідальність по управлінню версіями програмних компонентів, їх масштабуванню під навантаженням тощо. Для порівняння, у монолітній архітектурі всі вищенаведені завдання вимагають наявності відповідних знань лише у маленькій частини команди розробки, оскільки всі ці вимоги покриваються лише раз на всю програмну систему в цілому, а не на окремі її компоненти.

Вибір архітектурного підходу також може сприяти або перешкоджати взаємодії між командами у вигляді обсягу та частоти поширення знань, складності розподілу роботи між командами, зміни кількості чи розміру команд, можливості географічного розподілу виробничих ресурсів. Так, при використанні практики сильної власності коду, коли різні компоненти системи розробляються та підтримуються виключно різними командами, власна продуктивність однієї команди зростає за рахунок обмеження необхідних знань та відповідальності, проте загальна продуктивність розробки програмної системи може зменшуватись через підвищену складність планування та розподілу роботи між окремими командами.

Функціональна складність та розмір програмної системи має переважно негативний вплив на продуктивність розробки, за виключенням певного проміжку, коли відносно невелика та керована складність навпаки покращує продуктивність розробки за рахунок підвищеної мотивації розробників [53]. Вплив комбінованого ефекту складності програмної системи на продуктивність розробки, виявлений у вищезгаданому дослідженні [53, с. 1115], наведено на рис. 1.9. Застосування модульного підходу в контексті деяких архітектурних моделей дозволяє розподілити суцільну складність програмної системи між її окремими компонентами. Шляхом мінімізації зв'язності між модулями, локальна складність кожного модуля може бути обмежена для сповільнення зниження продуктивності розробки при подальшому зростанні функціональної складності та розміру програмного забезпечення.

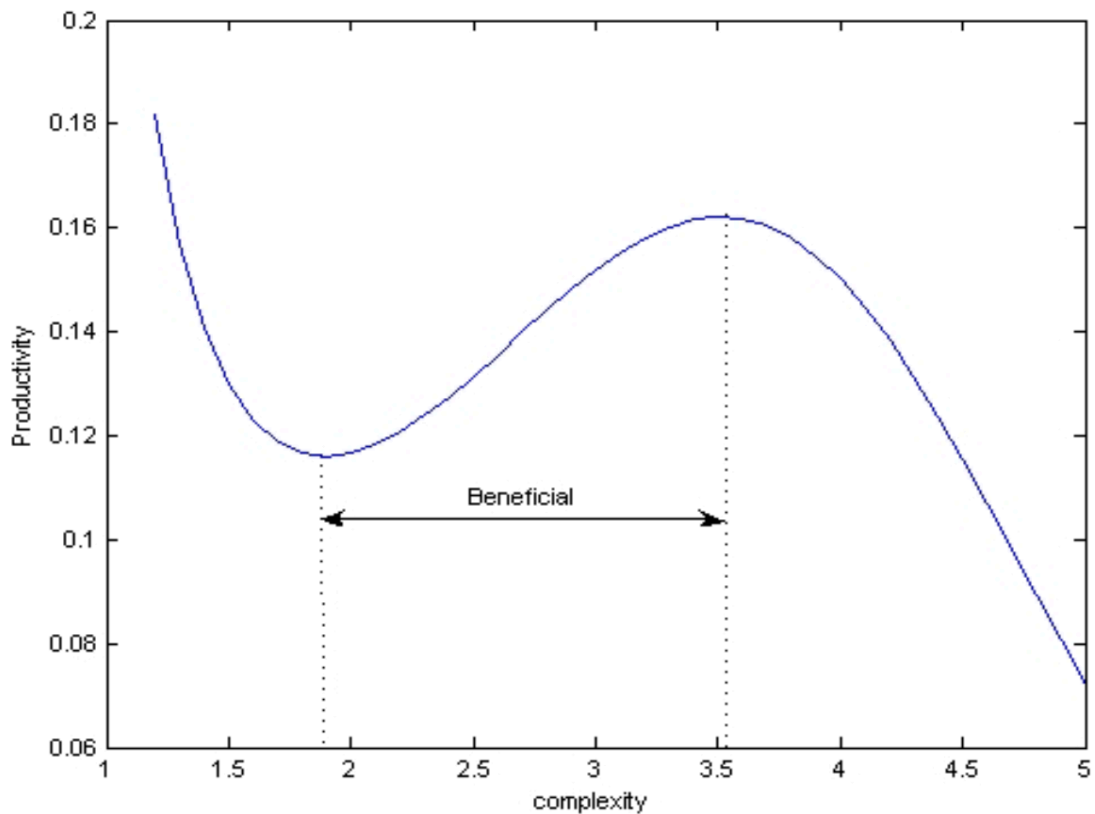


Рис. 1.9. Вплив комбінованого ефекту складності програмної системи на продуктивність розробки [53]

Врахування впливу різних архітектурних підходів на продуктивність розробки програмного забезпечення на етапі вибору архітектури майбутньої програмної системи може покращити точність оцінки вартості розробки такої системи. Проте лише фактів наявності таких зв'язків для цього недостатньо і є необхідність в подальшому дослідженні кількісних показників впливу різних вхідних параметрів процесу на продуктивність розробки, а також порівняння різних типових архітектурних підходів на предмет їх зв'язку з ефективністю процесу розробки програмного забезпечення.

Поняття власності вихідного коду існує вже декілька десятиріч, проте досі проводяться дослідження впливу даної практики на різні аспекти процесу розробки програмного забезпечення та не вщухають дискусії в професіональному середовищі щодо доцільності її застосування в тому чи іншому вигляді на практиці.

На даний момент розрізняють три основні типи власності коду: сильна власність (тільки обмежена група розробників може вносити зміни до окремої частини коду), слабка власність (будь-хто може вносити зміни, але лише під контролем «власників» коду) та колективна власність (всі володіють всім кодом). Схематично вони зображені на рис. 1.10. На практиці також можуть застосовуватись гібридні види власності коду.

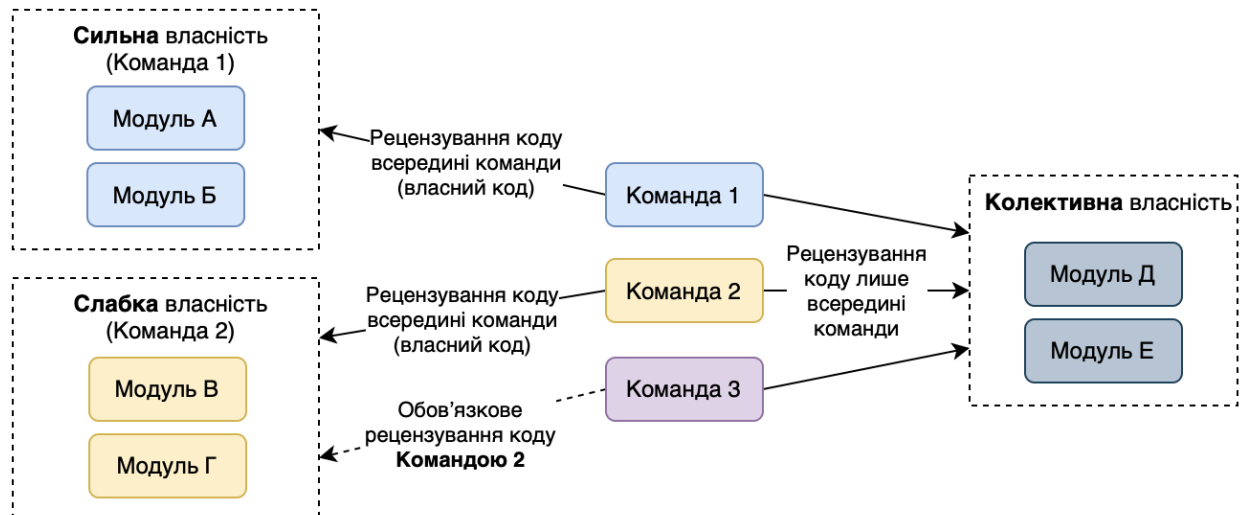


Рис. 1.10. Вплив власності коду на організаційні аспекти процесу розробки ПЗ

Дослідження впливу розподілу авторства коду на якість програмного забезпечення [58] показало, що найменша кількість дефектів спостерігається при використанні практики сильної власності коду. При дослідженні зв'язку власності коду з якістю програмного забезпечення в контексті сучасних практик рецензування коду [59] було виявлено, що рецензування будь-яких вхідних змін коду його «власниками» також позитивно впливає на результуючу якість програмної системи. В іншій роботі при аналізі впливу власності коду на складність його майбутньої підтримки [60] виявили кореляцію між кількістю авторів спільного коду та імовірністю погіршення його якості при внесенні чергових змін. В праці по дослідженню сприйняття власності коду командами розробки [61] було з'ясовано, що власність коду – це відчуття, яке посилюється при безпосередній участі в розробці певної частини коду, розумінні розробниками контексту всієї програмної системи, сприйнятті рівня якості коду

як високого, впевненості в задоволенні потреб користувачів програмного продукту та відчутті згуртованості команди розробки.

Хоча сильна власність коду в порівнянні з іншими її видами і дозволяє досягти найкращої якості коду, але також супроводжується певним переліком інших обмежень, які обов'язково мають бути взяті до уваги. Так, з одного боку, сильна власність сприяє досягненню найбільшої глибини знань в окремій частині коду, а з іншого – обмежує розуміння контексту всієї програмної системи та негативно впливає на процес розповсюдження знань між окремими командами розробки.

При необхідності швидкого масштабування процесів розробки, сильний тип власності коду є більш доцільним, оскільки об'єм знань, необхідних новим розробникам для початку ефективної роботи, фактично обмежений об'ємом власного коду і не потребує широкого спектру знань всієї системи в цілому. Проте, в такому випадку, варто також враховувати додаткову складність планування розробки програмної системи, оскільки кожна нова функціональна вимога може потребувати залучення та синхронізації роботи декількох різних команд розробки саме через фізичне обмеження можливості редагування вихідного коду, що знаходиться у власності інших команд.

Отже, на основі проведеного аналізу можна виділити наступні переваги та недоліки кожного виду власності коду в порівнянні з колективною власністю, що зображені в таблиці 1.1.

З таблиці 1.1 видно, що слабкий тип власності коду має більше переваг та менше недоліків в порівнянні з колективною власністю, а у сильної власності хоч і менше переваг та більше недоліків, але вони більш виражені. Як сильний, так і слабкий тип власності коду можуть принести свою користь в процес розробки програмного забезпечення, проте рішення про застосування того чи іншого підходу має спиратись на актуальні пріоритети конкретного проєкту, враховуючи всі недоліки та обмеження, які накладає такий вибір.

**Характеристика сильного та слабого видів власності коду в
порівнянні з колективною власністю**

Критерій	Слабка власність	Сильна власність
Якість коду	Краща за колективну	Найкраща
Глибина знань	Глибші знання	Найглибші знання
Обмін знаннями між командами	Сприяє обміну знаннями	Відсутність обміну знаннями
Масштабування розробки	Швидше масштабування	Найшвидше масштабування
Рецензування коду	Потребує додаткового рецензування	Не потребує додаткового рецензування
Планування	Без додаткових складнощів	Потребує координації між командами
Залежності між командами	Мінімальні залежності	Перехресні залежності між командами

Рецензування коду є одним із ключових процесів у сучасній розробці програмного забезпечення, який виконує функцію експертизи, контролю якості та менторства [62]. Вплив цього процесу на продуктивність розробника може бути аналізований через призму когнітивної психології, організаційної поведінки та теорій управління проєктами [54]. Наукова література вказує на те, що рецензування коду має багатогранний вплив, який залежить від контексту, технологій, культури команди та індивідуальних аспектів розробників [51, 52].

З позитивної сторони, рецензування коду сприяє зростанню продуктивності шляхом економії витрат часу на розв’язання проблем, які могли бути виявлені пізніше. Наприклад, дослідження, проведені в рамках роботи з високопродуктивними командами, показують, що рецензування зменшує кількість дефектів, які потрапляють до кінцевих користувачів, що веде до зменшення часу на виправлення і підтримку [28, 62]. Це відповідає теорії

"попереднього розгляду", яка стверджує, що виявлення помилок на етапі створення зменшує загальне когнітивне навантаження на команду [3]. Крім того, процес рецензування виступає як інструмент навчання – розробники отримують можливість аналізувати код інших, що поглиблює їхні знання про архітектуру, шаблони проєктування та найкращі практики. Це відповідає теорії соціального навчання, яка підкреслює цінність співпраці й обміну досвідом у процесі формування професійних навичок [51, 52].

Важливим аспектом рецензування є також сприяння формуванню "нормативної пам'яті" у команді – розробники краще засвоюють стандарти проєктування, стиль коду та принципи дизайну, що зменшує необхідність у повторному аналізі цих стандартів [58, 59]. Крім того, наукові дослідження показують, що конструктивний зворотній зв'язок в процесі рецензування може викликати позитивний емоційний стан, що збільшує витривалість, мотивацію та готовність до інновацій [51, 67].

З негативної сторони, рецензування коду може бути джерелом витрати ресурсів, особливо якщо процес не оптимізований або недостатньо відкалібрований [63, 68]. Науковці, які аналізують продуктивність команд, вказують на те, що неефективне рецензування, наприклад, з високим числом ітерацій чи нечітким зворотнім зв'язком, може викликати когнітивне перенавантаження, що призводить до зниження продуктивності [54, 67]. Також, відсутність чітких критеріїв для оцінки коду може призводити до суб'єктивної оцінки, що викликає конфлікти між розробниками, зниження довіри та веде до створення психологічних бар'єрів [45, 52].

Додатково, наукове дослідження вказує на те, що рецензування може пригнічувати ініціативність у розробниках, особливо якщо процес використовується як засіб контролю замість сприяння розвитку [51]. Так, у дослідженнях про теорію самодетермінації, встановлено, що забезпечення автономії, компетентності та пов'язаності є критичними факторами ефективності, і рецензування, яке не враховує ці фактори, може спричинити відчуття утримання та погіршити продуктивність [52, 54]. Також, розробники з

високим рівнем самоефективності можуть відчувати стрес через необхідність обґрунтовувати кожен свій вибір, що зменшує готовність до експериментальних підходів [53].

Ключовим фактором ефективності рецензування є його інтеграція в процеси розробки. Науковці, які аналізують Agile-методології, підкреслюють, що рецензування має бути частиною ітеративного циклу, а не окремим етапом, який відокремлений від інших завдань [28, 50]. Крім того, використання автоматизованих інструментів (наприклад, статичного аналізу коду) може зменшити кількість ручної роботи, що дозволить зосередитися на аналізі критичних аспектів, які автоматизований аналіз не зможе виявити.

Загалом, рецензування коду має багатогранний вплив на продуктивність розробки, що залежить від ефективності процесу, культурних норм команди та індивідуальних характеристик розробників. Якщо рецензування виконується з огляду на зручність, конструктивність та відкритість до ідей, воно може стати каталізатором зростання продуктивності. Але якщо процес не оптимізований, має формальний характер чи викликає високий рівень стресу, він може стати фактором зниження продуктивності.

Таким чином, є доцільним включення практики рецензування коду в симуляції процесу розробки програмного забезпечення для дослідження продуктивності розробки, особливо в контексті різних видів «власності коду», які часто свідомо чи несвідомо застосовуються при розробці модульних монолітних та мікросервісних програмних систем.

Оцінка продуктивності розробки програмного забезпечення є критичним аспектом управління проєктами, який допомагає визначити ефективність і якість процесів розробки. Існує кілька методів оцінки продуктивності, кожен з яких має свої переваги і застосовується в різних контекстах.

Нижче наведено основні метрики оцінки продуктивності розробки програмного забезпечення та аналіз доцільності їх використання для оцінки продуктивності розробки в симуляціях процесів розробки програмного забезпечення.

Кількість рядків коду (LOC, Lines of Code) є однією з найпростіших і найбільш використовуваних метрик оцінки продуктивності розробки. Вона підраховує загальну кількість рядків у вихідному коді, включаючи коментарі та порожні рядки. Ця метрика допомагає оцінити обсяг роботи, але має суттєві обмеження, оскільки не враховує складність коду, його якість або обчислювальну ефективність. Хоча кількість рядків коду і корелюється з продуктивністю розробки програмного забезпечення, проте дана метрика не враховує час на дизайн програмного рішення, а її абсолютні значення можуть суттєво відрізнятись в залежності від мови програмування, архітектури програмної системи, стилю написання коду тощо.

Варіанти використання (Use Case Points) ґрунтуються на аналізі функціональних вимог системи з точки зору кінцевого користувача. Ця метрика враховує кількість і складність варіантів використання, які описують різні сценарії взаємодії користувача з системою. Вона також враховує технічні та екологічні фактори, що впливають на складність реалізації кожного варіанту, такі як продуктивність, безпека та масштабованість системи. Це дозволяє враховувати всі аспекти розробки і забезпечує комплексну оцінку проєкту. Однак дана метрика має й свої недоліки. Один з них полягає в тому, що вона може бути складною у використанні та вимагати значних витрат часу на створення та аналіз варіантів використання. Це може збільшити тривалість початкової стадії планування проєкту. Крім того, точність оцінок залежить від якості та деталізації варіантів використання, а це вимагає від команди розробників високої кваліфікації та досвіду. Іншим недоліком є суб'єктивність оцінки складності варіантів використання, що може призвести до неточностей у прогнозах трудомісткості. Відсутність стандартизації в методах оцінки може також призвести до розбіжностей в оцінках між різними командами або проєктами. Додатково варто відмітити, що використання даної метрики вимагає застосування UML підходу до проєктування програмних систем, що вимагає додаткових знань та навичок. Крім того, кількість та складність варіантів використання може сильно відрізнятись в залежності від типу програмних

систем та їх специфіки, що ускладнює використання даної метрики для узагальненого порівняння продуктивності розробки програмного забезпечення в контексті різних архітектурних моделей.

Об'єктні бали (Object Points) використовуються переважно в об'єктно-орієнтованому програмуванні. Ця метрика оцінює кількість і складність об'єктів, таких як класи, інтерфейси та інші компоненти, з яких складається система. Вона враховує кількість методів і атрибутів у кожному об'єкті, а також взаємодію між об'єктами. Об'єктні бали допомагають оцінити зусилля, необхідні для створення та підтримки об'єктно-орієнтованої системи. Серед недоліків даного підходу варто відмітити його несумісність з не об'єктно-орієнтованими мовами програмування, а також суб'єктивність оцінки складності різних компонентів, що унеможлиблює порівняння продуктивності розробки між різними програмними системами. Так як в програмних симуляціях процесів розробки програмного забезпечення відсутній фактичний вихідний код, а відповідно відсутні і класи, інтерфейси тощо, застосування даної метрики є неможливим.

Функціональні точки (Function Points) є метрикою, яка оцінює розмір інформаційної системи на основі функціональних вимог до неї. Вона враховує кількість і складність зовнішніх входів, виходів, запитів, внутрішніх логічних файлів і зовнішніх інтерфейсів. Ця метрика дозволяє оцінити трудомісткість розробки, зосереджуючись на функціональності, яку система надає користувачам, незалежно від технології реалізації, що дозволяє порівнювати різні проєкти та команди, незалежно від використовуваних мов програмування або платформ. Функціональні точки є більш об'єктивною метрикою порівняно з LOC, оскільки враховують реальну корисність і функціональність системи. Основний недолік даного підходу полягає в складності та трудомісткості процесу вимірювання, який вимагає ретельного аналізу функціональних вимог і залучення досвідчених фахівців, що може збільшити час і витрати на початкових етапах проєкту. В контексті симуляції процесів розробки програмного забезпечення головний недолік даної метрики нівелюється через відсутність необхідності попередньої оцінки складності задач програмування, натомість

задачі різної складності генеруються у вигляді потоку вхідної роботи. Також варто підкреслити, що так як метрика оцінки продуктивності з використанням функціональних точок не залежить від технології реалізації певної функціональності, а та сама функціональність може мати різну складність в контексті різних типів архітектури, використання даної метрики в програмних симуляціях процесів розробки програмного забезпечення забезпечить об'єктивність порівняльного аналізу продуктивності розробки для різних архітектурних моделей.

Висновки до першого розділу

У першому розділі проведено комплексний аналіз сучасного стану досліджень у сфері моделювання процесів розробки програмного забезпечення, детально розглянуто структуру цього процесу з точки зору його основних етапів, архітектурних підходів та організаційних факторів, а також досліджені відомі фактори впливу на ефективність процесу розробки програмного забезпечення та методи оцінки продуктивності розробки.

Аналіз академічної літератури засвідчив, що симуляційне моделювання процесів розробки програмного забезпечення залишається активною та динамічною областю наукових досліджень, у якій спостерігається стійка тенденція до зростання кількості публікацій протягом останніх двох десятиліть. Серед парадигм моделювання системна динаміка та дискретно-подійне моделювання залишаються найпоширенішими, однак агентно-орієнтований підхід набуває дедалі більшої популярності завдяки можливості відтворювати індивідуальну поведінку розробників, враховувати неоднорідність команд та моделювати стохастичні комунікаційні взаємодії. Встановлено, що класичні математичні моделі оцінювання трудовитрат, зокрема COCOMO та закон Брукса, зберігають свою фундаментальну цінність для розуміння залежностей між параметрами проєкту та продуктивністю розробки. Формула комунікаційного навантаження, що описує квадратичне зростання каналів зв'язку зі збільшенням кількості учасників команди, є ключовим інструментом

обґрунтування рішень щодо оптимального розміру команд. Проте ці моделі потребують суттєвої адаптації до сучасних реалій, оскільки вони недостатньо враховують вплив архітектурного стилю системи, погано пристосовані до гнучких методологій розробки та не відображають особливості роботи географічно розподілених команд. Показано, що вибір архітектурного стилю має вимірюваний та статистично значущий вплив на продуктивність розробки, який суттєво залежить від розміру команди, складності системи та рівня організаційної зрілості. Монолітна архітектура забезпечує вищу продуктивність для невеликих команд завдяки простоті розробки, тестування та розгортання. Мікросервісна архітектура надає переваги у частоті розгортань та швидкості впровадження змін для великих команд і складних систем, проте супроводжується значним зростанням інфраструктурної складності, що змушує частину організацій повертатися до монолітного підходу. Модульна монолітна архітектура, як відносно новий, але швидко зростаючий напрямок, пропонує привабливий компроміс між операційною простотою моноліту та структурною модульністю мікросервісів.

Детальний огляд структури процесу розробки програмного забезпечення продемонстрував, що сучасний життєвий цикл розробки є циклічним та ітеративним процесом, що охоплює формування стратегічного бачення у продуктовій дорожній карті, деталізацію вимог у черзі завдань продукту, планування ітерацій розробки, розподіл завдань між розподіленими командами, безпосереднє програмування із використанням системи контролю версій, рецензування програмного коду, багаторівневе тестування та розгортання у виконавчому середовищі. Кожен із цих етапів має свої особливості залежно від обраного архітектурного стилю, що додатково підтверджує значущість архітектурних рішень для загальної ефективності процесу розробки.

Також в даному розділі був проведений аналіз наукової літератури для збору всіх факторів процесу розробки програмного забезпечення, що мають вплив на продуктивність розробки та відокремлення лише тих з них, що можуть мати різний вплив на продуктивність в залежності від обраної архітектури

програмної системи. Дослідження наявності або відсутності такого впливу, а також його міри можливе при проведенні відповідних симуляцій процесу розробки з різними вхідними значеннями для кожного з таких параметрів. При аналізі застосування практики власності коду було виділено наступні три окремі її види: сильна, слабка та колективна власність. Кожен вид власності коду має свої переваги та недоліки, тому доцільність їх застосування має оцінюватись в кожному окремому випадку, зважаючи на специфіку програмного продукту, його архітектуру, організаційну структуру команд розробки, знання та навички окремих розробників, а також бюджет проєкту. Так, наприклад, для мікросервісної архітектури найчастіше застосовується практика сильної власності коду, оскільки зазвичай вихідний код кожного окремого сервісу відокремлений від інших для швидкого масштабування процесів розробки такої програмної системи. В контексті програмних симуляцій процесу розробки програмного забезпечення доцільно провести однакові симуляції, але з різними типами власності коду для дослідження їх впливу на більш детальному рівні. Дослідження процесу рецензування коду виявило можливість як позитивного, так і негативного потенційного впливу на продуктивність розробки, що сильно залежить від якості впровадження такого процесу, внутрішньої культури в команді розробки, а також рівня автоматизації процесу. Враховуючи те, що організація процесу рецензування коду може суттєво відрізнятись в залежності від типу архітектури програмного забезпечення та типу практики «власності коду», є доцільним включити процес рецензування коду до моделі симуляції для дослідження як якісного, так і кількісного впливу на продуктивність розробки програмного забезпечення. Додатково були розглянуті сучасні метрики оцінки продуктивності розробки програмного забезпечення, такі як кількість рядків вихідного коду, варіанти використання, об'єктні бали та функціональні точки. При детальному аналізі переваг, недоліків та обмежень кожної метрики, а також доцільності їх застосування для вимірювання продуктивності розробки в програмних симуляціях процесів розробки програмного забезпечення в контексті різних архітектурних моделей, було виявлено, що метрика

функціональних точок найкраще підходить для поставленої задачі, а її основні недоліки нівелюються за рахунок рівня абстракції симуляційних моделей.

На основі проведеного аналізу визначено пріоритетні напрямки подальших досліджень, серед яких розробка математичних моделей оцінки продуктивності з явним урахуванням архітектурного стилю як вхідного параметра, застосування агентно-орієнтованого моделювання для порівняльного аналізу ефективності різних архітектурних підходів за різних організаційних конфігурацій.

Матеріали розділу 1 подано в таких публікаціях автора [69, 70, 71].

РОЗДІЛ 2. Алгоритмічні основи моделювання процесу розробки програмного забезпечення

2.1. Алгоритм імовірнісної генерації потоку вхідних задач різної складності на основі історичних даних проєктів з відкритим вихідним кодом

Теоретичні аспекти низькорівневого моделювання процесу розробки програмного забезпечення, включно з аналізом законів розподілу розміру вхідних задач, були досліджені різними вітчизняними та іноземними науковцями, серед яких Ярош І.В., Павловський Є.В., Назарова І.А. [72], Чіанг А.Д. (Chiang A.J.) і Джіанг А. (Jeang A.) [73], Бошар П.О. (Bochard P.O.) і Шварц Т. (Schwarz T.) [74], Лунесу М. (Lunesu M.), Тонеллі Р. (Tonelli R.), Маркезі Л. (Marchesi L.), Маркезі М. (Marchesi M.) [50]. В різних працях використовуються різні закони розподілу складності задач, такі як нормальний, випадковий, бета-розподіл, ступенева та дрібно-раціональна функції та інші, що вказує на необхідність детальнішого дослідження закону розподілу складності задач, включно з такими його параметрами, щоб забезпечити можливість його застосування в симуляціях процесу розробки програмного забезпечення, максимально наближеного до реальності. Крім того, багато законів розподілу не мають верхньої межі, що може потребувати його адаптації для можливості застосування в симуляціях процесу розробки.

Проведемо дослідження імовірнісного розподілу складності вхідних задач для дискретно-подійного та агентно-орієнтованого моделювання процесу розробки програмного забезпечення шляхом аналізу фактичного розподілу розміру задач на основі історичних даних проєктів з відкритим вихідним кодом.

Для дослідження імовірнісного розподілу складності задач при розробці програмного забезпечення доцільно проаналізувати набір даних «Apache Jira Issue Tracking Dataset» [75], що містить історичну інформацію про виконані задачі при розробці програмних проєктів з відкритим вихідним кодом, що належать до фонду «Apache Software Foundation».

Із загальної кількості у більш ніж мільйон різних типів записів, для аналізу розподілу складності практичних задач були обрані лише повністю завершені задачі з інформацією про фактично витрачений час, які мають тип дефекту («Bug»), покращення існуючої функції («Improvement») або створення нової («New Feature»). Типи записів більш високого (абстрактного) рівня, задачі по тестуванню та створенню документації та інші були виключені з вибірки, оскільки вони або включають в себе, або вже є включеними в задачі обраних типів.

Для більш детального аналізу закону розподілу складності задач розробки є доцільним дослідити розподіл як всіх задач разом, так і окремо кожного з обраних типів задач.

На рис. 2.1 зображені відповідні гістограми розподілу часу, витраченого на різні типи задач.

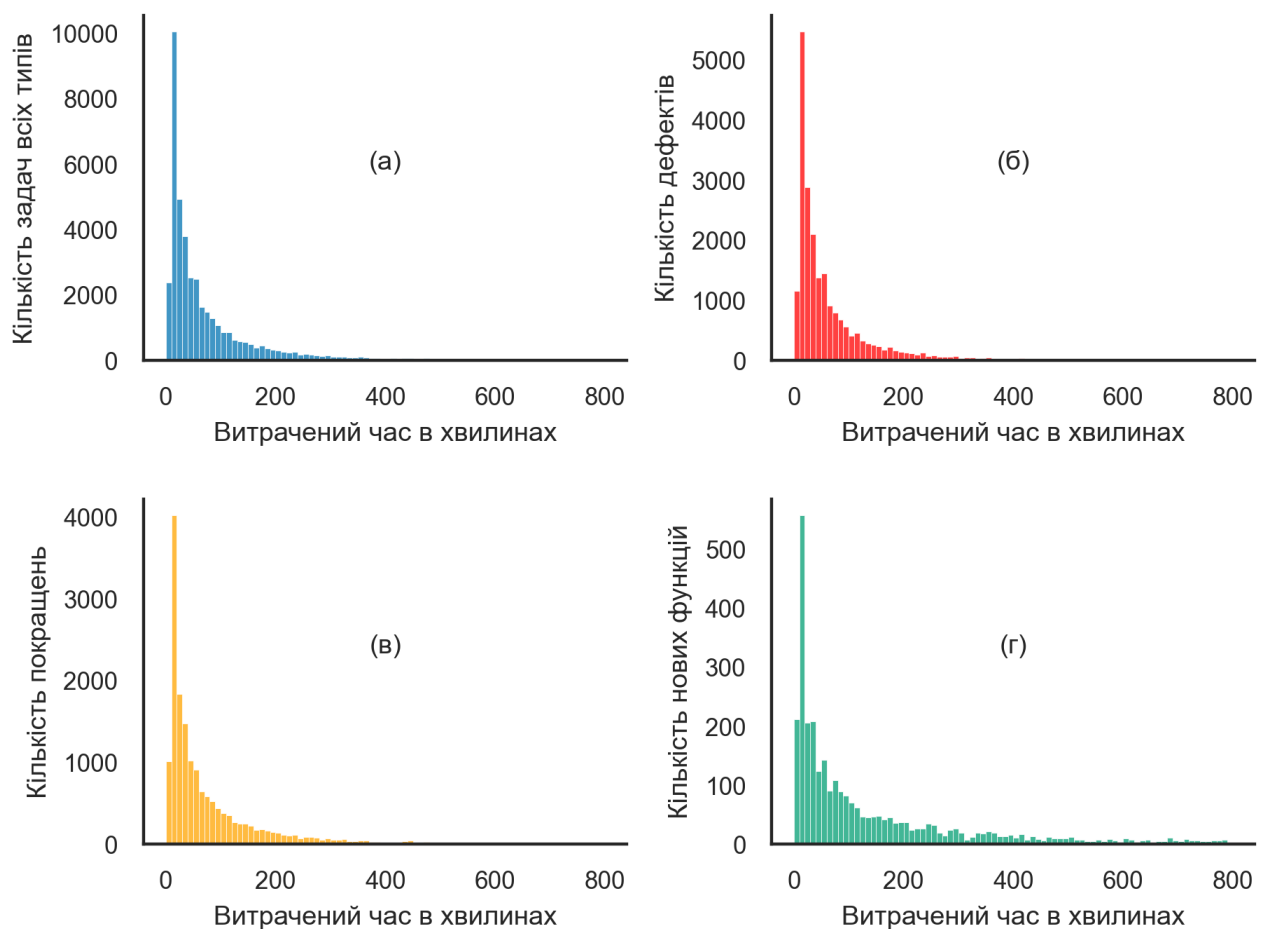


Рис. 2.1. Гістограми розподілу витраченого часу для (а) всіх типів задач, (б) лише дефектів, (в) покращень та (г) нових функцій

Вищенаведені гістограми мають однакову форму функції розподілу імовірностей незалежно від типу задач, що нагадує форму логнормального розподілу, проте для доведення даної гіпотези необхідно визначити числові параметри даного розподілу та порівняти графік отриманої функції густини імовірності з фактичними історичними даними.

Логнормальний розподіл випадкової величини X задається функцією:

$$X = e^{\mu + \sigma Z}, \quad (2.1)$$

де μ та σ – дійсні числа, при чому $\sigma > 0$, а Z – випадкова величина, задана стандартним нормальним розподілом. Параметри розподілу μ та σ можна знайти за допомогою методу максимальної правдоподібності [76, с. 332], використовуючи наступні формули:

$$\hat{\mu} = \frac{\sum_k \ln x_k}{n}, \quad (2.2)$$

$$\hat{\sigma} = \sqrt{\frac{\sum_k (\ln x_k - \hat{\mu})^2}{n}}, \quad (2.3)$$

де x_k – значення витраченого часу на k -ту задачу, n – загальна кількість задач.

Після визначення параметрів розподілу можливий розрахунок таких його числових характеристик, як середнє значення витраченого часу, медіана, мода та дисперсія. Результати розрахунків з точністю до 4 десяткових знаків наведені в таблиці 2.1.

Таблиця 2.1.

Параметри та характеристики логнормального розподілу складності задач

	Параметри та характеристики розподілу					
	μ	σ	Середнє	Медіана	Мода	Дисперсія
Всі типи задач	3.9687	1.0512	91.9440	52.9136	17.5249	17070.9490
Дефекти	3.8718	0.9600	76.1424	48.0269	19.1073	8774.9768
Покращення	4.0020	1.0685	96.8140	54.7048	17.4662	19983.3344
Нові функції	4.4581	1.3645	218.9741	86.3193	13.4134	260621.6962

Виходячи з гістограм на рис. 2.1 та обчислених характеристик розподілу з таблиці 2.1 можна зробити висновок, що розподіли складності дефектів та покращень наближені до загального розподілу всіх типів задач разом, в той час як для задач на розробку нових функцій програмної системи спостерігається більше середнє значення витраченого часу та підвищена імовірність його відхилення в сторону збільшення. З точки зору практичного процесу розробки програмного забезпечення, специфіка роботи над різними типами задач не має суттєвих відмінностей одна від одної, тому для генерації потоку вхідних задач при моделюванні процесу розробки програмного забезпечення достатньо використати параметри імовірнісного розподілу складності для всіх типів задач.

Відповідно до формули (2.1), окрім обрахованих параметрів логнормального розподілу, для генерації задач різної складності необхідно також мати можливість генерувати значення випадкової величини за стандартним нормальним розподілом. Для цього доцільно використати перетворення Бокса-Мюллера, так як даний підхід є більш обчислювально ефективним за метод зворотного перетворення та алгоритм Зіккурата. Полярна форма перетворення Бокса-Мюллера для моделювання стандартних нормально розподілених випадкових величин [77, с. 281] виглядає наступним чином:

$$R = x^2 + y^2, \quad (2.4)$$

$$z_0 = x \cdot \sqrt{\frac{-2 \ln R}{R}}, \quad (2.5)$$

$$z_1 = y \cdot \sqrt{\frac{-2 \ln R}{R}}, \quad (2.6)$$

де x та y – незалежні випадкові величини, рівномірно розподілені на інтервалі $[-1, 1]$, z_0 та z_1 – отримані незалежні випадкові величини, розподілені нормально з математичним сподіванням 0 і дисперсією 1, що задовольняє стандартному нормальному розподілу. Якщо при обрахунку значення R виявиться, що $R > 1$ чи $R = 0$, то значення випадкових величин x та y мають бути згенеровані повторно, поки дана умова щодо R не буде виконана.

На рис. 2.2 наведено оригінальну гістограму, побудовану на фактичних історичних даних з вибірки, та гістограму згенерованих випадкових чисел відповідно до логнормального розподілу з обчисленими параметрами.

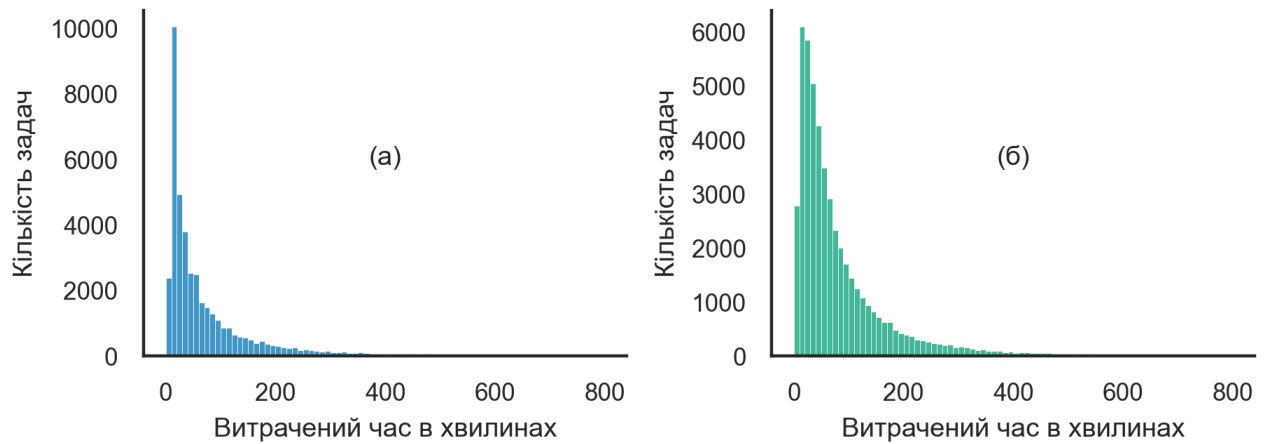


Рис. 2.2. Гістограми розподілу витраченого часу для (а) фактичних історичних даних, (б) згенерованих випадкових значень

З вищенаведених графіків видно, що отриманий розподіл імовірностей має більш пологий спуск, ніж фактичний закон розподілу, що спростовує гіпотезу наявності логнормального розподілу складності задач згідно історичних даних. Враховуючи переважаючу імовірність найчастішого значення витраченого часу на задачу і суттєво менші імовірності сусідніх значень, є доцільним застосувати інший закон розподілу імовірностей, що буде краще відображати розподіл фактичних даних. Серед загальновідомих законів розподілу найбільш наближену форму функції густини імовірностей має зворотний розподіл Гауса [78], тому варто визначити його параметри і так само порівняти результат з фактичним розподілом складності задач.

Параметри зворотного розподілу Гауса μ та λ можуть бути розраховані на основі вибірки даних методом максимальної правдоподібності [79, с. 265] за допомогою наступних формул:

$$\hat{\mu} = \sum_{i=1}^n x_i / n, \quad (2.7)$$

$$\hat{\lambda} = n / \sum_{i=1}^n \left(\frac{1}{x_i} - \frac{1}{\hat{\mu}} \right), \quad (2.8)$$

де x_i – значення витраченого часу на i -ту задачу, n – загальна кількість задач.

В таблиці 2.2 наведено розраховані параметри та властивості зворотного розподілу Гауса на основі фактичних даних вибірки з точністю до 4 десяткових знаків.

Таблиця 2.2.

Параметри та характеристики зворотного розподілу Гауса

μ	λ	Середнє	Медіана	Мода	Дисперсія
112.8197	49.1561	113.7829	55.1802	14.8830	29213.0746

Для генерації випадкових чисел, що відповідають зворотному розподілу Гауса, можна скористатись наступним алгоритмом [80, с. 89]:

$$x = \mu + \frac{\mu^2 Z^2}{2\lambda} - \frac{\mu}{2\lambda} \sqrt{4\mu\lambda Z^2 + \mu^2 Z^4}, \quad (2.9)$$

$$R = \begin{cases} x, & \text{якщо } U \leq \frac{\mu}{\mu+x} \\ \frac{\mu^2}{x}, & \text{якщо } U > \frac{\mu}{\mu+x} \end{cases}, \quad (2.10)$$

де μ та λ – параметри розподілу, Z – випадкова величина, задана стандартним нормальним розподілом, що може бути обчислена за формулами (2.4), (2.5) та (2.6), U – випадкова величина, рівномірно розподілена на інтервалі $[0, 1]$, R – випадкова величина за зворотним розподілом Гауса.

На рис. 2.3 зображені функції густини імовірності логнормального та зворотного розподілу Гауса, накладені на гістограму фактичних даних.

Як видно з рис. 2.3, зворотний розподіл Гауса краще повторює форму фактичного розподілу даних, ніж логнормальний, за виключенням аномально високої частоти найпоширенішого значення витраченого часу на одну задачу згідно фактичних даних.

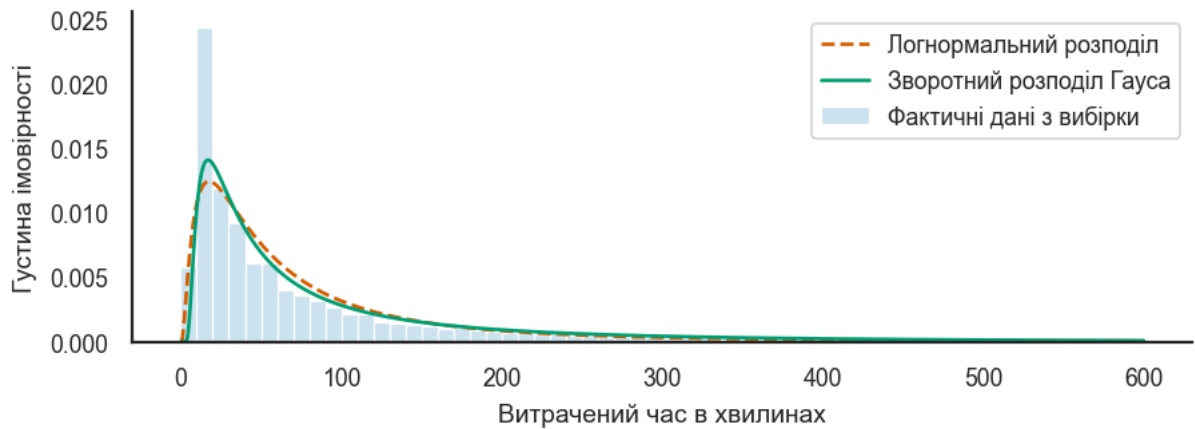


Рис. 2.3. Відповідність різних розподілів імовірностей фактичному розподілу даних

Під час аналізу значень витраченого часу, що найчастіше зустрічаються у поточній вибірці задач із набору історичних даних, було виявлено, що найчастіше значення витраченого часу дорівнює 20 хвилинам, а більше 99% значень є кратними 10. В таблиці 2.3 відображені 10 найчастіших значень витрат часу із відповідною кількістю задач із вибірки.

Таблиця 2.3.

Найчастіші значення витрат часу

Витрати часу (хв)	20	30	40	50	60	10	70	80	90	100
Кількість задач	9974	4913	3792	2494	2492	2304	1639	1486	1295	1085

Виходячи з виявленого факту кратності складності задач та властивості неперервності зворотного розподілу Гауса, алгоритм генерації складності можна адаптувати шляхом округлення всіх вихідних значень до найближчого цілого числа, кратного десяти, що є більшим за вхідне дійсне число. Округлення вгору до більшого числа є необхідним, щоб виключити можливість отримання на виході нульової складності задачі та не спотворити розподіл імовірностей.

Враховуючи аномально високу частоту значення витраченого часу у 20 хвилин, що не підпадає під зворотний закон розподілу Гауса, можна скоригувати

алгоритм генерації складності задач при моделюванні процесу розробки програмного забезпечення для компенсації даної аномалії. Відповідно до вибірки даних, імовірність найпоширенішого значення дорівнює 23,685%. Шляхом інтегрування функції розподілу імовірностей зворотного розподілу Гауса з використанням обчислених параметрів на інтервалі (10, 20] було отримано сумарну імовірність даного діапазону значень 13,2938%. Таким чином, реальна імовірність даного значення на 10,3912% більша за імовірність згідно з отриманим розподілом, проте компенсаційну імовірність необхідно обчислити із врахуванням загальної частоти найпоширенішого значення. Отже, для компенсації аномально високої імовірності найпоширенішого значення в 20 хвилин можна застосувати наступні формули:

$$P = \frac{P_D - P_G}{1 - P_G} = 11,9844\%, \quad (2.11)$$

$$E = \begin{cases} 20, & \text{якщо } U < P \\ R, & \text{якщо } U \geq P \end{cases}, \quad (2.12)$$

де P_D – імовірність найпоширенішого значення згідно фактичних даних, P_G – імовірність найпоширенішого значення згідно отриманого зворотного розподілу Гауса, U – випадкова величина, рівномірно розподілена на інтервалі $[0, 1)$, P – компенсаційна імовірність для коригування розподілу, R – випадкова величина, розподілена за отриманий зворотним розподілом Гауса, що обраховується за формулою (2.10), E – випадкова величина згідно адаптованого розподілу складності задач. Гістограма адаптованого розподілу зображена на рис. 2.4.

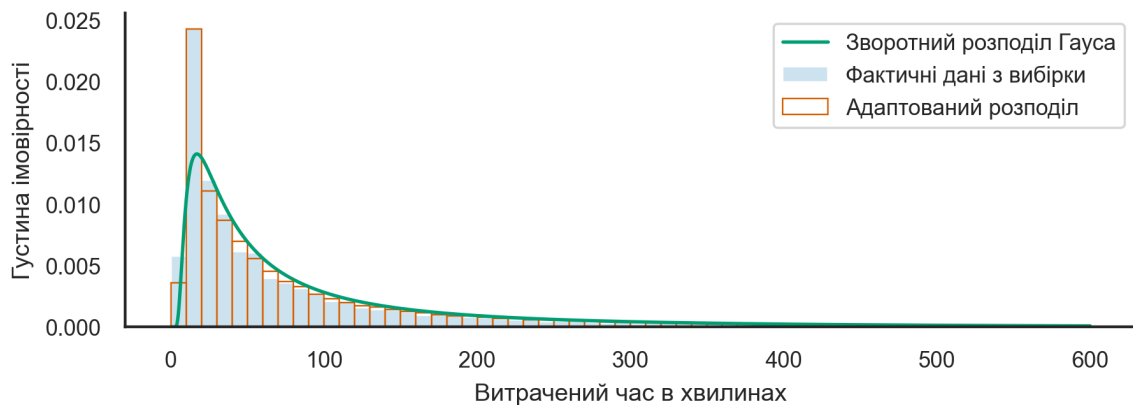


Рис. 2.4. Порівняння адаптованого розподілу імовірностей з фактичними даними

Ще однією особливістю отриманого розподілу, яка заважає моделюванню процесу розробки програмного забезпечення при генерації потоку вхідних задач є те, що графік його функції не обмежений в правій частині. Це може призвести до генерації надмірно великих значень складності задач розробки, що в свою чергу може призвести до спотворення результатів симуляцій, тому доцільно обмежувати максимальне значення складності задач шляхом повторної генерації при виході згенерованого значення за граничну верхню межу. Найбільше значення витраченого часу на одну задачу у поточній вибірці даних дорівнює 1200 годин, що еквівалентно 150 людино-дням при 8-годинному робочому дні. Задачі такого розміру дійсно зустрічаються на практиці, проте при моделюванні процесу розробки краще задавати граничне значення складності задач в якості параметру моделі, щоб час, необхідний для виконання задачі, не перевищував загальну тривалість періоду часу, що симулюється.

Таким чином, проведене у даному підрозділі дослідження дозволило сформулювати цілісний підхід до імовірнісного моделювання складності вхідних задач у процесі розробки програмного забезпечення. Початковий аналіз фактичних історичних даних із набору «Apache Jira Issue Tracking Dataset» підтвердив, що розподіл складності задач має характерну правосторонню асиметрію, притаманну багатьом реальним процесам, де більшість задач мають відносно невелику трудомісткість, тоді як значно менша частка задач потребує суттєвих витрат часу. Саме ця асиметрія обумовлює необхідність застосування спеціалізованих імовірнісних розподілів, здатних адекватно відтворити подібну форму. Порівняльний аналіз логнормального та зворотного розподілу Гауса продемонстрував, що жоден із них не відтворює фактичний розподіл даних бездоганно, проте зворотний розподіл Гауса забезпечує значно кращу відповідність реальній формі розподілу складності задач. Логнормальний розподіл, незважаючи на його широке застосування в суміжних дослідженнях, дає надмірно пологий хвіст функції густини імовірності, що призводить до завищення імовірності появи задач середньої та великої складності й відповідного заниження імовірності найпоширеніших значень. Зворотний

розподіл Гауса, натомість, більш точно передає концентрацію значень навколо моди розподілу і швидше спадає у правій частині, що краще узгоджується з емпіричними спостереженнями.

Водночас було встановлено, що навіть зворотний розподіл Гауса не здатний повністю відобразити аномально високу частоту значення витраченого часу у 20 хвилин, яка є наслідком дискретної природи фактичних даних. У реальних проєктах розробники часто фіксують витрачений час із округленням до певних інтервалів, що створює штучні піки на гістограмі. Запропонований механізм компенсації цієї аномалії через введення додаткової компенсаційної імовірності дозволяє зберегти безперервну природу зворотного розподілу Гауса і водночас врахувати особливості реального процесу обліку часу. Окрім вибору закону розподілу та його параметрів, важливим практичним аспектом є механізм обмеження верхньої межі складності згенерованих задач. Оскільки зворотний розподіл Гауса теоретично не має правої граничної точки, без відповідного обмеження генератор може створити задачу з настільки великою складністю, що її виконання перевищуватиме загальний горизонт моделювання. У контексті дискретно-подійної та агентно-орієнтованої симуляції це може спричинити ситуацію, коли один або кілька агентів-розробників будуть зайняті однією задачею протягом усього часу симуляції, що суперечить реальній практиці декомпозиції великих завдань на менші підзадачі. Тому введення параметризованої верхньої межі складності є не лише технічною необхідністю, а й відображенням реальних організаційних практик управління проєктами, де надмірно великі задачі зазвичай поділяються на менші перед початком роботи. Варто також зазначити, що адаптація неперервного розподілу до дискретних значень шляхом округлення до найближчого кратного десяти має подвійне обґрунтування. По-перше, вона відображає реальну практику обліку часу, де розробники рідко фіксують витрати з точністю до хвилини, надаючи перевагу округленим значенням. По-друге, така дискретизація спрощує подальше використання згенерованих значень у моделях, де час моделювання вимірюється в дискретних одиницях. Округлення вгору, а не до найближчого значення,

забезпечує додаткову гарантію того, що жодна задача не отримає нульову складність, яка була б фізично безглуздою у контексті реального процесу розробки.

Отримані результати мають безпосереднє прикладне значення для побудови імітаційних моделей процесу розробки програмного забезпечення. Параметри зворотного розподілу Гауса, розраховані на основі репрезентативної вибірки реальних проєктних даних, можуть бути безпосередньо використані для налаштування генератора вхідних задач у симуляційній моделі. Це дозволяє забезпечити реалістичність потоку задач, що надходять до моделі, і тим самим підвищити достовірність результатів моделювання в цілому. Крім того, запропонований підхід може бути адаптований для інших наборів історичних даних із різних організацій або проєктів, що робить його універсальним інструментом для калібрування імітаційних моделей в сфері програмної інженерії.

2.2. Алгоритм моделювання зв'язності модулів програмної системи шляхом імовірнісного розподілу внесення змін на основі історичних даних проєктів з відкритим вихідним кодом

Зв'язок модульності з різними метриками процесу розробки програмного забезпечення був досліджений ще декілька десятиліть тому, де було виявлено позитивний вплив модульної організації коду, слабкої зв'язності назовні та сильного зчеплення всередині модулів на продуктивність розробки, надійність програмної системи та зменшення вартості її подальшого обслуговування. Тому є доцільним моделювання модульної організації коду в низькорівневих симуляціях процесу розробки програмного забезпечення для врахування даного впливу на продуктивність розробки.

Незважаючи на те, що факт позитивного впливу модульності на продуктивність розробки вже давно відомий, проблема вимірювання та моніторингу зв'язності мікросервісів [56, 57] та питання оптимальної

модульності [81] все ще активно вивчається науковцями різних країн. Враховуючи комплексну природу взаємозалежностей різних компонентів програмної системи, що було детально висвітлено в праці «Demystifying dependence» Коппела Дж. (Koppel J.) та Джексона Д. (Jackson D.) [55], є доцільним визначення параметрів модульності шляхом дослідження еволюційної зв'язності модулів на основі історії внесення змін до вихідного коду існуючих модульних програмних систем. Даний підхід також широко використовується для передбачення майбутніх змін коду в пов'язаних модулях, як, наприклад, в працях Рольфснес Т. (Rolfesnes T.) та інші [82] і Мондал М. (Mondal M.) та інші [83].

Для дослідження параметрів модульності були обрані 8 мікросервісних та модульних монолітних програмних проєктів з відкритим вихідним кодом на платформі GitHub. Загальна інформація про обрані проєкти відображена в таблиці 2.4.

Таблиця 2.4.

Модульні проєкти для аналізу

Назва проєкту	Мова	Архітектура	К-ть модулів	К-ть змін	К-ть файлів	К-ть рядків коду
OrchardCore	C#	монолітна	116	4457	4702	183748
magento2	PHP	монолітна	220	79454	15543	1120353
ftgo-application	Java, Groovy	мікросервісна	23	130	315	9004
SimplCommerce	C#	монолітна	43	509	730	27560
Modular-monolith-by-example	C#	монолітна	11	185	754	14534
dotnet-starter-kit	C#	монолітна	25	23	288	6706
eShopOnContainers	C#	мікросервісна	8	879	434	23672
eShop	C#	мікросервісна	21	114	491	17603

Кожен проєкт містить вихідний код на різних мовах програмування, включаючи клієнтський код для імплементації інтерфейсу користувача, проте для дослідження параметрів модульності саме серверних типів архітектури були

проаналізовані лише ті частини вихідного коду, що організовані в модулі та представлені серверними мовами програмування. Довжина історії змін, кількість файлів та кількість рядків коду були відповідно відфільтровані та відображають лише характеристики модульних серверних частин коду, а також не включають в себе код автоматизованих тестів.

Так як компілятор мови C# не підтримує циклічних залежностей між окремими проєктами вихідного коду, загальновідомим шляхом вирішення таких технічних обмежень є розділення певного модуля на два окремих проєкти різного рівня абстракції. Так, більшість бібліотек фреймворку ASP.NET представлені двома пакетами: один містить лише абстракції, а інший код імплементації. Такий самий підхід використовується і в більшості кінцевих модульних проєктів, як, наприклад, в проєкті OrchardCore один окремий модуль може бути представлений двома проєктами вихідного коду, один з яких містить в своїй назві додатковий суфікс «.Core» після назви модуля. В інших проєктах також можуть використовуватись суфікси «.Abstractions» або «.Contracts». При аналізі вихідного коду в даному дослідженні такі проєкти сприймаються як складові частини відповідних модулів, а не як окремі модулі.

Як видно з таблиці 2.4, найбільша кількість модулів присутня в проєктах «OrchardCore» та «magento2», тому самі ці 2 проєкти доцільно проаналізувати на предмет функції розподілу розміру модулів. Гістограми розподілу розміру модулів для обох проєктів з лінійною та логарифмічною шкалою наведені на рис. 2.5. Для кращого сприйняття інформації, розмір модулів виражається у відсотках від загального об'єму модульного коду відповідного проєкту.

Так як гістограми з логарифмічною шкалою мають форму нормального розподілу, то реальний розподіл розмірів модулів даних програмних систем підпорядковується логнормальному розподілу, але з різними параметрами для різних проєктів.

Таким самим чином можна проаналізувати розподіл частот зміни різних модулів та дослідити зв'язок розміру модулів від частоти їх зміни.

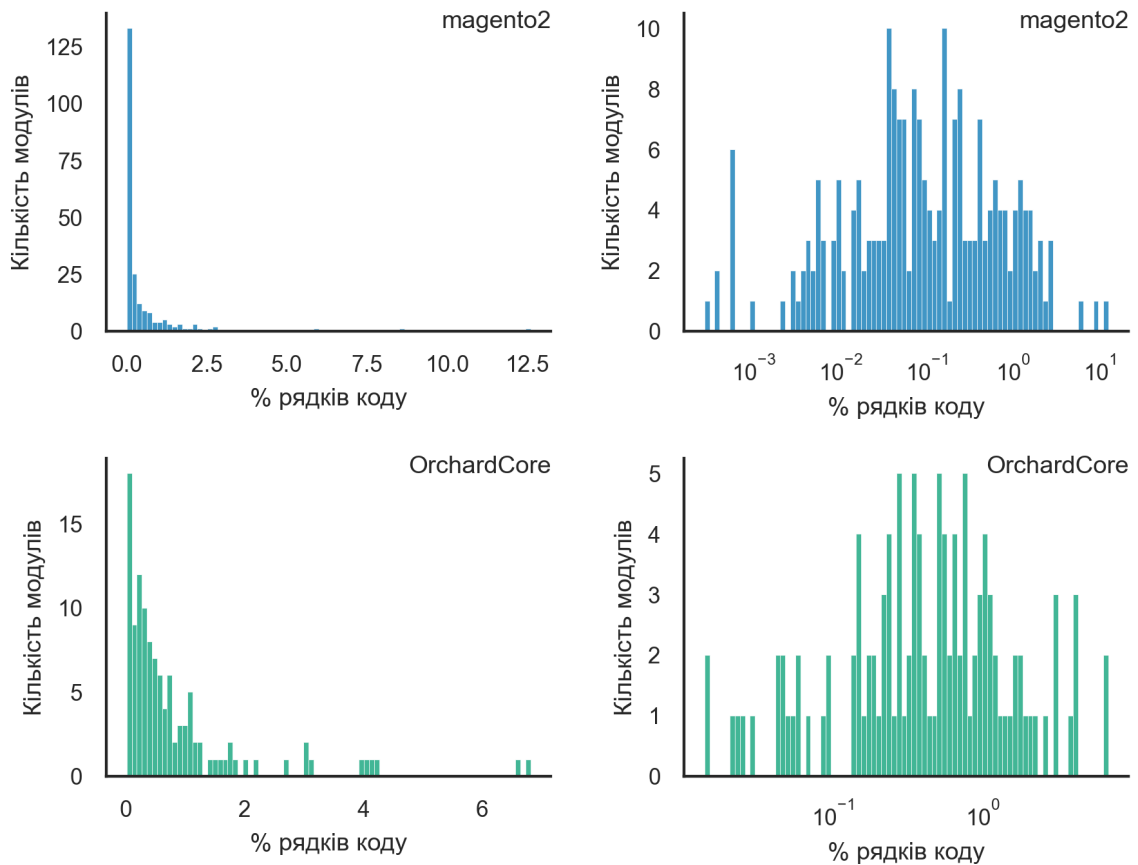


Рис. 2.5. Розподіл розміру модулів для проєктів «OrchardCore» та «magento2»

Частота зміни модуля може бути розрахована за допомогою формули (2.13), що також буде відповідати незалежній імовірності зміни модуля в кожній окремій транзакції модифікації вихідного коду системи.

$$change_frequency(A) = \frac{count(A)}{N}, \quad (2.13)$$

де $count(A)$ – кількість транзакцій модифікації коду, в яких вносяться зміни до модуля A , N – загальна кількість транзакцій.

Залежність частоти внесення змін в модуль від його розміру зображена на рис. 2.6 у вигляді діаграми розсіювання. З даної діаграми можна зробити висновок, що залежність між розміром модуля та частотою його зміни є прямою та має лінійний характер, тобто більші за розміром модулі змінюються частіше. Також варто підкреслити, що дана кореляція притаманна як монолітній архітектурі, так і мікросервісній.

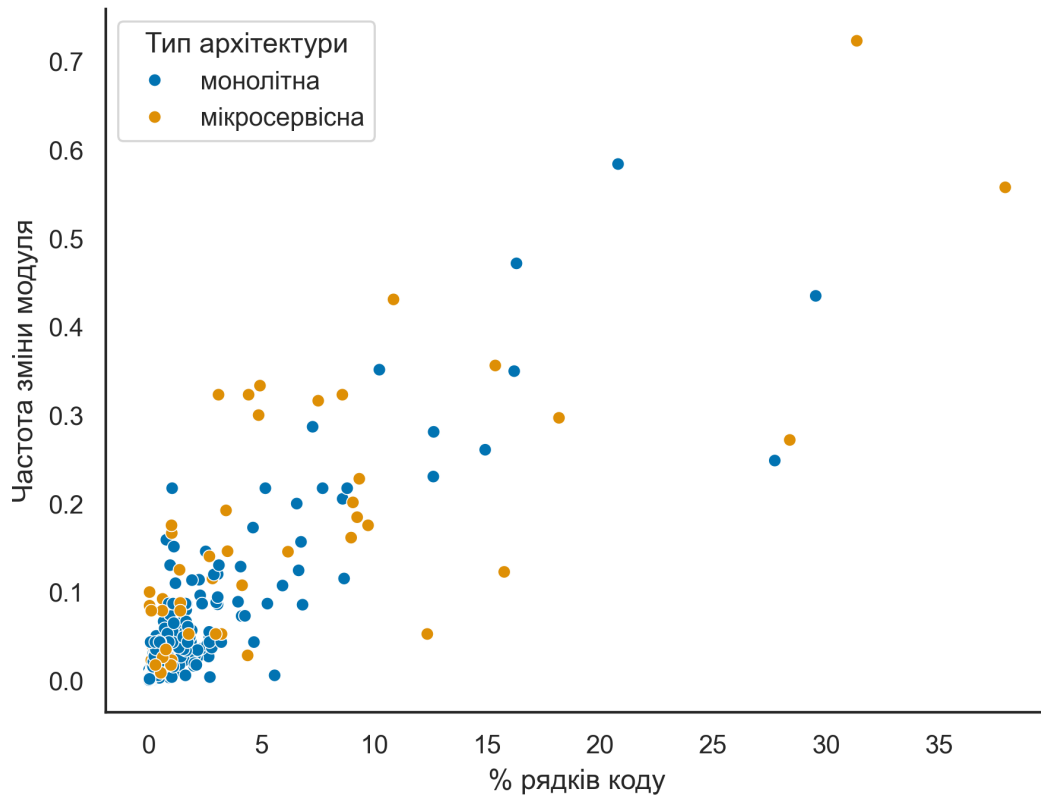


Рис. 2.6. Залежність частоти зміни модуля від його розміру

Наступним кроком є дослідження зв'язності модулів на основі історії транзакцій внесення змін до них. Це можливо зробити шляхом виявлення правил асоціації між модулями, що ґрунтується на аналізі частот зміни кожної пари модулів як окремо один від іншого, так і одночасно в рамках однієї транзакції. В методах передбачення майбутніх змін використовуються обидва параметри метрики: «support» (підтримка) та «confidence» (довіра), що розраховуються за формулами (2.14) та (2.15) [19, с. 202], проте для моделювання зв'язності достатньо буде розрахувати лише «confidence».

$$support(A \rightarrow B) = \frac{count(A \cup B)}{N}, \quad (2.14)$$

де $count(A \cup B)$ – кількість транзакцій зміни коду, в яких одночасно модифікуються модулі A і B, N – загальна кількість транзакцій.

$$confidence(A \rightarrow B) = \frac{count(A \cup B)}{count(A)}, \quad (2.15)$$

де $count(A \cup B)$ – кількість транзакцій зміни коду, в яких одночасно модифікуються модулі A і B, $count(A)$ – загальна кількість транзакцій, в яких змінюється модуль A.

Розраховані значення $confidence(A \rightarrow B)$ виражають вихідну зв'язність модуля A по відношенню до модуля B, або імовірність зміни модуля B при модифікації модуля A, та симетрично вхідну зв'язність модуля B по відношенню до модуля A. Для дослідження взаємозв'язку між зв'язністю модулів та їх незалежною імовірністю зміни доцільно розрахувати середні значення вихідної та вхідної зв'язності для кожного модуля та порівняти їх з частотами зміни модулів. На рис. 2.7 зображені діаграми залежності середніх значень вхідної та вихідної зв'язності всіх модулів з частотою їх зміни. Дані різних проєктів зображені різними кольорами.

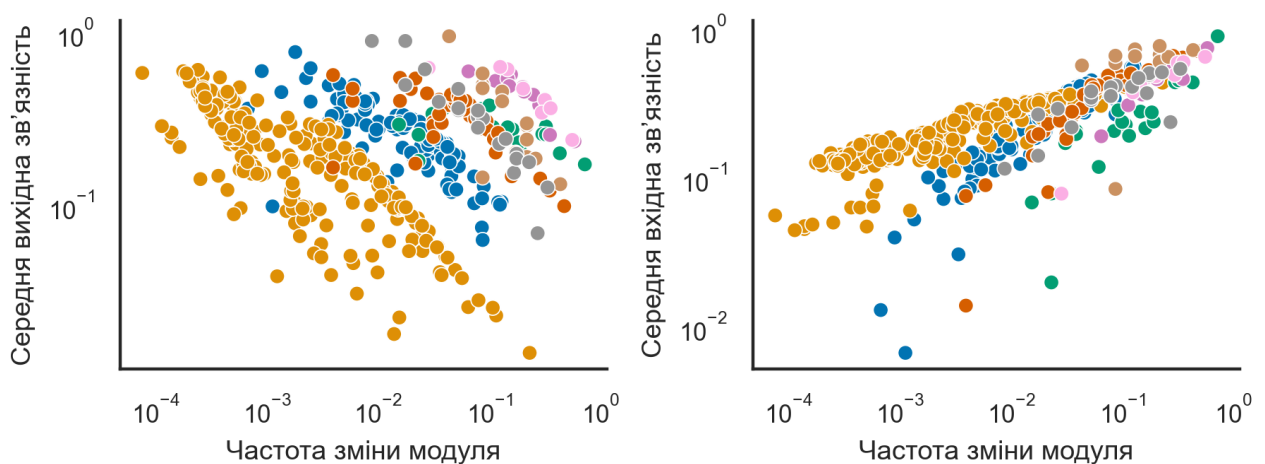


Рис. 2.7. Кореляція частоти зміни модулів та їх середніх значень вхідної та вихідної зв'язності

Логарифмічні шкали на діаграмах розкривають наявність степеневої функції залежності зв'язності модулів від частоти їх зміни, але параметри такої залежності різні для різних проєктів. Як видно з діаграм на рис. 2.7, чим більша частота зміни коду модуля, а відповідно і його розмір, тим менша імовірність внесення змін в інші модулі та тим більша імовірність зміни такого модуля при модифікації інших модулів.

Теоретично, для моделювання зв'язності модулів в дискретно-подійних та агентно-орієнтованих симуляціях можна на кожній ітерації внесення змін до коду обирати перший модуль та, комбінуючи параметри вхідної та вихідної зв'язності, симулювати зміни в пов'язаних модулях, але у даного підходу є декілька проблем. Розраховані раніше незалежні імовірності зміни окремих модулів вже включають в себе як вхідну, так і вихідну зв'язаність з іншими модулями, впливаючи на розраховані імовірності зміни один одного, тому дані імовірності не можна використовувати для вибору першого модуля для модифікації в певній транзакції внесення змін до вихідного коду системи. Крім того, моделювання як різної кількості модулів, так і зв'язності між кожною їх парою може бути обчислювально неефективним та сповільнювати виконання симуляцій. Також варто зауважити, що середня кількість зв'язаних модулів по всіх проєктах складає приблизно 96% від загальної кількості модулів у відповідній системі.

Враховуючи наведені обмеження та отримані дані, альтернативним підходом до моделювання зв'язності може бути використання незалежних імовірностей внесення змін для визначення всіх модулів програмної системи, що підлягають модифікації на кожній ітерації симуляції. Як видно з рис. 2.7, зв'язність модулів має степеневу залежність від частоти їх модифікації, але відрізняється від проєкта до проєкта. Так як імовірнісний розподіл розміру модулів має логнормальну форму, а частоти зміни модулів лінійно залежать від їх розміру, то закон розподілу частот модифікації модулів також має логнормальний характер. Враховуючи те, що чим менша кількість модулів в програмній системі, тим частіше кожен з них змінюється, є доцільним визначення параметрів логнормального розподілу частот модифікації модулів для кожного проєкту та дослідження їх взаємозв'язку з загальною кількістю модулів в кожному з них.

Для розрахунку коефіцієнтів логнормального розподілу частот зміни модулів окремо для кожного проєкту можна використати метод максимальної правдоподібності. На рис. 2.8 зображені співвідношення отриманих коефіцієнтів

розподілу відносно логарифмів кількості модулів у відповідному проєкті та їх екстраполяції, отримані шляхом лінійної регресії.

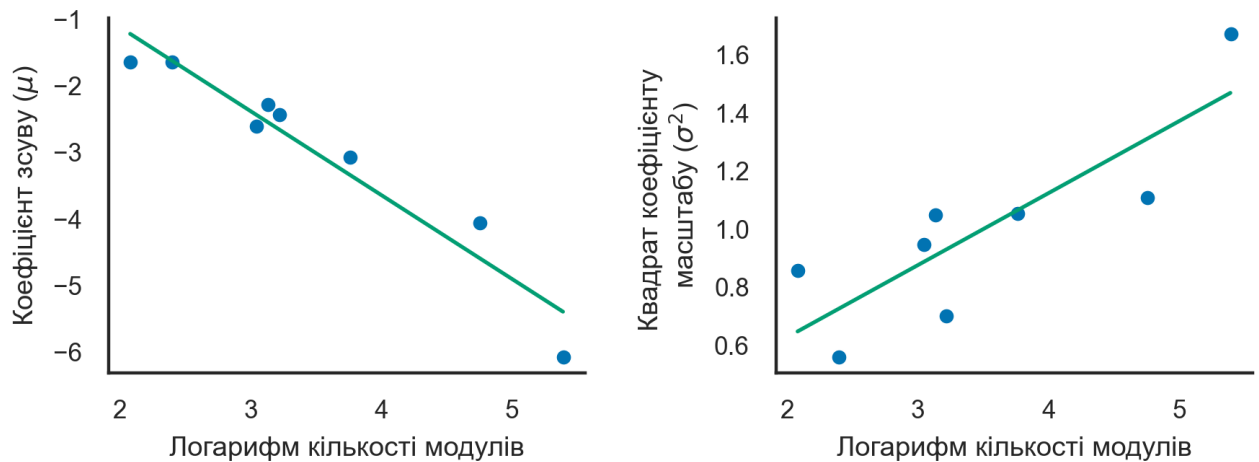


Рис. 2.8. Залежність параметрів логнормального розподілу частот зміни модулів від їх загальної кількості у відповідному проєкті

Згідно отриманих результатів, параметри логнормального розподілу частот зміни модулів від їх кількості при симуляції процесу розробки абстрактної програмної системи можуть бути розраховані за формулами (2.16) та (2.17).

$$\mu = -1.2624653134178174 \ln N + 1.407336741730882, \quad (2.16)$$

$$\sigma = \sqrt{0.24806030790958875 \ln N + 0.13184253926824063}, \quad (2.17)$$

де N – загальна кількість модулів в проєкті, а коефіцієнти нахилу та точок перетину з віссю y для обох параметрів логнормального розподілу були розраховані методом простої лінійної регресії (див. рис. 2.8).

На етапі підготовки до запуску симуляції процесу розробки, виходячи з необхідної кількості модулів, мають бути розраховані параметри розподілу частот їх зміни відповідно до формул (2.16) і (2.17) та відповідно до отриманого розподілу кожен модуль отримає своє значення незалежної імовірності модифікації на кожній ітерації симуляції. Так як логнормальний розподіл – неперервний, то існує ненульова імовірність генерації частоти зміни модуля, що буде більше 100%, тому під час генерації випадкових значень варто відсіювати всі згенеровані значення, більші за обрану верхню межу. Максимальне значення

імовірності зміни модуля серед всіх проаналізованих проєктів складає 72.3%, що також може бути використано в якості такої верхньої межі.

Так як згенеровані імовірності модифікації модулів незалежні, то існує ненульова імовірність, що на певній ітерації симуляції жоден модуль не буде змінений. В такому випадку процес відбору модулів, що підлягають модифікації на поточній ітерації симуляції має бути повторений знову, поки хоча б один модуль не увійде в поточну транзакцію внесення змін до коду системи. Спираючись на розраховані частоти зміни модулів проєктів, що аналізуються, середнє значення імовірності невнесення змін в жоден модуль серед всіх проєктів складає 5.6%, а максимальне значення досягає 13.4%. Для оптимізації обчислень в симуляціях процесу розробки програмного забезпечення пропонується підхід до коригування згенерованих імовірностей модифікації модулів таким чином, щоб максимально знизити імовірність невнесення змін в жоден модуль в процесі симуляції, але тим часом не спотворити розподіл частот змін між модулями. Для цього необхідно розрахувати поточну імовірність невнесення змін в жоден модуль за формулою (2.18), вирахувати коригуючий коефіцієнт за формулою (2.19) та перерахувати поточні імовірності модифікації всіх модулів за формулою (2.20).

$$Q = \prod_i^N (1 - p_i), \quad (2.18)$$

де N – загальна кількість згенерованих модулів, p_i – імовірність внесення змін в i -тий модуль, Q – поточна імовірність невнесення змін в жоден модуль.

$$K = \left(\frac{Q'}{Q} \right)^{\frac{1}{N}}, \quad (2.19)$$

де N – загальна кількість згенерованих модулів, Q – поточна імовірність невнесення змін в жоден модуль, $Q' > 0$ – бажана імовірність невнесення змін в жоден модуль, K – коригуючий коефіцієнт.

$$p'_i = 1 - K \cdot (1 - p_i), \quad (2.20)$$

де p_i – згенерована імовірність внесення змін в i -тий модуль, K – коригуючий коефіцієнт, p'_i – скоригована імовірність зміни i -го модуля.

Таким чином, запропонований у даному підрозділі підхід до моделювання зв'язності компонентів програмної системи ґрунтується на емпіричному аналізі реальних програмних проєктів з відкритим вихідним кодом і дозволяє відтворити статистично обґрунтовані закономірності розподілу змін між модулями під час симуляції процесу розробки. Суть підходу полягає в тому, що замість явного моделювання попарних залежностей між модулями, яке потребує зберігання та обробки квадратичної за розміром матриці зв'язності, використовується непрямий механізм моделювання через незалежні імовірності модифікації кожного модуля, параметри яких визначаються логнормальним розподілом в залежності від загальної кількості програмних модулів системи, що суттєво спрощує обчислювальну складність симуляції. Важливим практичним наслідком отриманих результатів є можливість параметризації симуляційної моделі на основі лише одного вхідного параметра – загальної кількості модулів у програмній системі, зі значення якого за допомогою формул (2.16) та (2.17) автоматично розраховуються параметри логнормального розподілу, які визначають імовірнісну структуру внесення змін до кожного окремого модуля. Така параметрична економність є значною перевагою при побудові узагальнених симуляційних моделей, оскільки дозволяє уникнути необхідності калібрування великої кількості окремих параметрів зв'язності для кожної конкретної конфігурації системи. Слід також зазначити, що виявлена лінійна залежність між розміром модуля та частотою його зміни має важливе теоретичне значення для розуміння природи еволюції програмних систем. Ця закономірність свідчить про те, що більші модулі, які, як правило, інкапсулюють ширшу функціональність, є більш схильними до модифікацій не лише через більшу кількість потенційних точок зміни, а й через те, що вони частіше виступають об'єктами залежності для інших компонентів системи. Відповідно, при проєктуванні модульної архітектури доцільно враховувати цю залежність і прагнути до збалансованого розподілу функціональності між модулями, щоб уникнути надмірної концентрації змін у невеликій кількості великих модулів. Крім того, результати аналізу підтверджують, що степенева залежність між вхідною та вихідною

зв'язністю модулів і частотою їх модифікації є універсальною характеристикою, притаманною як модульним монолітним архітектурам, так і мікросервісним системам. Це означає, що запропонована модель має достатній рівень узагальнення для застосування до різних типів архітектурних рішень без необхідності окремого калібрування для кожного типу архітектури. Водночас різниця у конкретних параметрах ступеневої залежності між проєктами вказує на вплив специфічних організаційних, технологічних та доменних факторів, які можуть бути предметом подальших досліджень.

Запропонований механізм коригування імовірностей модифікації модулів за допомогою формул (2.18), (2.19) та (2.20) забезпечує практичну придатність моделі для використання в симуляціях, де при виконанні кожної задачі мають бути внесені зміни до програмного коду хоча б одного модуля. При цьому запропонований метод коригування зберігає пропорційність розподілу імовірностей між модулями, що гарантує відсутність систематичного зміщення результатів симуляції.

Підсумовуючи, моделювання зв'язності компонентів шляхом імовірнісного розподілу внесення змін являє собою компромісний підхід між точністю відтворення реальних міжмодульних залежностей та обчислювальною ефективністю симуляції. Застосування логнормального розподілу з параметрами, що залежать від кількості модулів, дає змогу генерувати реалістичні сценарії модифікації коду без необхідності моделювання кожної окремої залежності. Даний підхід до моделювання зв'язності модулів може бути застосований в дискретно-подійних та агентно-орієнтових симуляціях процесу розробки програмного забезпечення для подальшого дослідження впливу модульності на продуктивність розробки. Подальші дослідження можуть бути спрямовані на уточнення коефіцієнтів регресії шляхом аналізу більшої вибірки проєктів, а також на врахування динаміки еволюції модульної структури в часі, включаючи додавання нових модулів, видалення застарілих та перерозподіл функціональності між існуючими компонентами системи.

2.3. Архітектура програмного комплексу для проведення симуляцій процесів розробки програмного забезпечення

Вибір архітектурної моделі при проектуванні майбутньої програмної системи є одним з ключових факторів, що визначають можливість успішного завершення даного проєкту та його подальшого ефективного розвитку і підтримки. В сучасному світі при виборі архітектурного підходу найбільша увага приділяється функціональним вимогам до системи та найбільш поширеним нефункціональним вимогам, таким як швидкодія, надійність, безпека, зручність використання тощо. В той же час, ефективність процесу розробки такого програмного продукту зазвичай не враховується через складність її оцінки та відсутність загальновідомих порівняльних характеристик впливу різних архітектурних підходів на ефективність розробки. Як показало дослідження 5400 ІТ-проєктів, в середньому великі проєкти перевищують початковий бюджет на 45%, 7% затримуються і не встигають завершитись до встановленого кінцевого терміну та містять на 56% менше функцій, ніж було заплановано від самого початку [7].

Проведення масштабного експерименту шляхом розробки однієї і тієї ж програмної системи із застосуванням різних архітектурних підходів для подальшого порівняльного аналізу є недоцільним через його високу вартість та необ'єктивність отриманих результатів вимірювань через різницю в знаннях та досвіді учасників експерименту, а також неможливості передбачення та виключення будь-яких інших випадкових факторів, що можуть спотворити результат експерименту. Ще одним із можливих шляхів дослідження такого впливу є проведення низькорівневих симуляцій процесів розробки для різних типів архітектури за допомогою дискретно-подійного та агентно-орієнтованого моделювання, що не потребує великих фінансових та часових витрат, а також абсолютно виключає вплив людського фактору на результати вимірювань.

Моделювання різних бізнес-процесів вже давно широко використовується як на практиці, так і в дослідницькій діяльності. Важливість ролі моделювання бізнес-процесів підприємств для їх подальшого вдосконалення та впровадження

інновацій підкреслюється в працях Шматковської Т., Коробчук Т. [84], а також Сараванос А. (Saravanos A.), Курінга М. (Curinga M.) [49]. Аналіз наукових досліджень щодо моделювання процесу розробки програмного забезпечення [10] виявив тенденції до спаду застосування парадигми системної динаміки та зростання популярності серед науковців дискретно-подійних та агентно-орієнтованих симуляцій, які також виявились найпоширенішими парадигмами.

Різні підходи до моделювання процесу розробки програмного забезпечення висвітлювались в працях вітчизняних науковців, серед яких Кордунова Ю.С., Фелтіновські М., Придатко О.В., Смотр О.О. [85], Приходько С.Б., Приходько Н.В., Книрик К.О. [86], а також іноземними дослідниками, такими як Сараванос А. (Saravanos A.), Курінга М. (Curinga M.) [49]. Однак в даних роботах процес розробки програмного забезпечення моделюється на рівні етапів життєвого циклу програмного проєкту, де рівень деталізації ходу виконання робіт недостатній для дослідження впливу програмної архітектури системи на ефективність його розробки.

До поширених підходів симуляційного моделювання процесів розробки програмного забезпечення відносяться агентно-орієнтоване, дискретно-подійне, стохастичне моделювання, системна динаміка, моделювання на основі процесів, а також різні гібридні методи, що поєднують елементи декількох підходів для отримання більш точної та детальної моделі. Аналіз останніх наукових досліджень щодо моделювання процесу розробки програмного забезпечення [10] показав, що найбільш поширеними серед науковців є агентно-орієнтоване та дискретно-подійне моделювання, при чому популярність агентно-орієнтованого підходу з часом зростає, а застосування системної динаміки, навпаки, йде на спад. Системна динаміка зазвичай застосовується для високорівневого моделювання різних етапів розробки програмних проєктів та надає лише середні значення визначених метрик [19], проте останнім часом спостерігається тенденція до застосування також симуляцій більш низького рівня, таких як агентно-орієнтоване, для моделювання тих самих високорівневих процесів [49]. Гібридні симуляції зустрічаються рідше та зазвичай поєднують системну динаміку або з агентно-орієнтованим, або з

дискретно-подійним підходами для одночасного моделювання на різних рівнях абстракції [87, 88]. Доцільність застосування агентно-орієнтованого підходу до моделювання також розглядається і в інших сферах діяльності, де необхідне відтворення складних шаблонів поведінки або вивчення процесів системи, що симулюється, зсередини [89].

Так як для дослідження впливу різних архітектурних підходів на ефективність розробки програмного забезпечення необхідний високий рівень деталізації внутрішніх процесів, що відбуваються під час розробки, тестування, розгортання та подальшої підтримки програмної системи, доцільно за основу симуляційної моделі взяти агентно-орієнтований або дискретно-подійний підхід.

Перевагами агентно-орієнтованих симуляцій є можливість моделювання складної поведінки системи, включно з динамічною взаємодією між агентами, а також здатністю агентів адаптуватись до змін середовища. Основними недоліками даного підходу є його обчислювальна складність та важкість валідації моделі. Дискретно-подійні симуляції є навпаки обчислювально ефективними, але їм бракує гнучкості для моделювання складних динамічних систем з комплексними взаємодіями між їх складовими елементами. Так, при застосуванні агентно-орієнтованої симуляційної моделі є можливість помітити та дослідити певні взаємозв'язки між елементами складної системи, які ще не були виявлені, але, так як для детального аналізу поведінки системи необхідно виконувати багато циклів симуляцій, обчислювальна ефективність дискретно-подійного підходу також має свої переваги. Враховуючи складну природу взаємодії між учасниками процесу розробки програмного забезпечення, найбільшу гнучкість для дослідження впливу такої взаємодії на ефективність розробки може надати лише парадигма агентно-орієнтованого моделювання. В агентно-орієнтованих симуляціях час представлений у вигляді послідовності ітерацій, де на кожній наступній ітерації кожен агент приймає рішення та вносить зміни в свій власний стан або в стан всієї моделі.

З метою моделювання процесу розробки програмного забезпечення для дослідження впливу різних архітектурних моделей на продуктивність розробки

програмних систем є доцільним створення програмного комплексу для проведення відповідних симуляцій на необхідному рівні деталізації процесу розробки, що дозволить знімати метрики продуктивності для різних архітектурних моделей.

Так як в симуляціях процесу розробки будуть використовуватись імовірнісні розподіли для генерації задач різної складності, переривань роботи над поточними задачами для виправлення раптово виявлених критичних дефектів, а також для симуляції спорадичної необхідності внесення змін в суміжні модулі програмної системи, необхідно забезпечити можливість виконання великої кількості симуляційних ітерацій з фіксованими вхідними параметрами, але з різним випадковим початковим значенням для отримання усереднених значень вимірюваних величин.

Крім того, важливо забезпечити можливість виконання симуляційних ітерацій з фіксованими початковими значеннями генерації випадкових величин для відтворення попередніх запусків. Відтворення результатів дозволяє дослідникам перевіряти, чи є отримані результати стабільними та надійними, що особливо важливо у наукових дослідженнях для підтвердження висновків. Якщо під час симуляції виникають помилки або несподівані результати, можливість повторити ті ж самі умови допомагає виявити і виправити ці помилки. Відтворення симуляцій сприяє кращій документованості досліджень та прозорості, що є критично важливим для наукових публікацій і технічних звітів, дозволяючи іншим дослідникам відтворити дослідження та перевірити результати. Можливість повторювати симуляції з тими ж умовами дозволяє проводити тестування та оптимізацію моделей і алгоритмів, оцінюючи їх продуктивність та ефективність. У випадках, коли необхідно використовувати або порівнювати результати з минулими симуляціями, можливість відтворення старих запусків надає можливість аналізувати історичні дані та робити висновки на основі змінних у часі факторів. Відтворення симуляцій в навчальних цілях допомагає студентам та новачкам зрозуміти процеси та взаємозв'язки в моделюванні, забезпечуючи стабільну платформу для вивчення та

експериментів. Загалом, відтворюваність є основою для наукового методу і технологічного розвитку, оскільки вона гарантує, що результати є достовірними, точними і можуть бути перевірені та використані іншими.

Ще одним важливим та критичним компонентом симуляційного програмного комплексу є генерація випадкових величин згідно різних імовірнісних розподілів. При чому, для деяких параметрів симуляції може бути корисною можливість вибору імовірнісного закону розподілу в якості вхідного параметру симуляційної моделі, щоб мати можливість порівняти результати симуляцій для різних законів розподілу. Наприклад, у випадку суттєвих розбіжностей виміряних значень при використанні різних імовірнісних розподілів може бути доцільним детальніше дослідження фактичного розподілу значень такої змінної на основі збору даних з реальних проєктів. Це допоможе удосконалити симуляційну модель та зробити її більш приближеною до реальності.

Багаторівнева архітектура програмного комплексу для запуску симуляцій та збору вихідних метрик зображена на рис. 2.9.

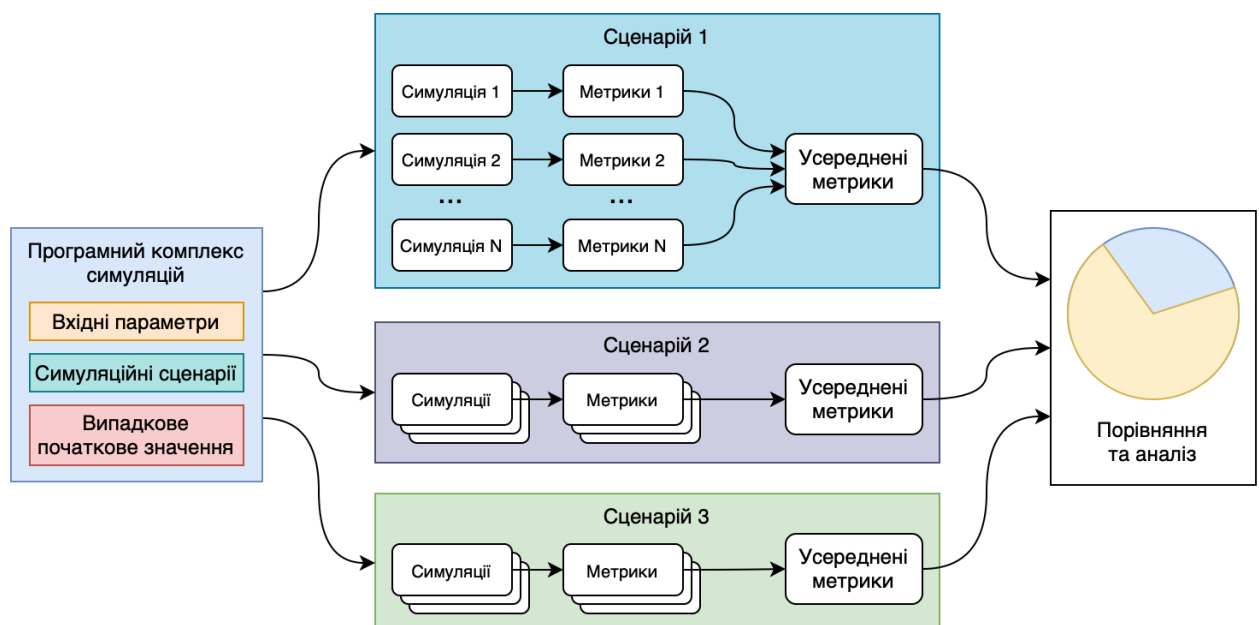


Рис. 2.9. Багаторівнева архітектура програмного комплексу запуску симуляцій та збору метрик

При моделюванні процесів розробки програмного забезпечення немає необхідності включати просторову складову в симуляційну модель, а от часова складова потребує додаткової деталізації для максимального наближення процесу розробки, що моделюється, до реального. Так, наприклад, під кінець робочого дня імовірність розпочати роботу над великою задачею з вищим пріоритетом нижча, ніж над меншою задачею, але з нижчим пріоритетом. Також необхідно враховувати час на перемикання між задачами при очікуванні результату рецензування коду, додаткові витрати часу для рецензування чужого коду, а також перерви в роботі через природні потреби.

За основу для побудови програмного комплексу проведення агентно-орієнтованих симуляцій процесів розробки програмного забезпечення було обрано розподілений та високомасштабуємий ABM фреймворк MARS LIFE (Multi-Agent Research & Simulation), розроблений в рамках дослідницького проєкту Гамбурзького університету прикладних наук для використання в дослідженнях та навчанні [90]. Він дозволяє створювати агентно-орієнтовані симуляції шляхом імплементації на мові C# класів агентів (Agent), шарів (Layer), сутностей (Entity) та середовищ (Environment) для покриття всіх необхідних аспектів симуляції. Агенти та активні шари є діючими елементами симуляції, які на кожному її кроці можуть аналізувати оточуючу обстановку та приймати рішення. Складові концепти даного фреймворку з їх особливостями на призначенням описані в таблиці 2.5.

Таблиця 2.5.

Складові концепти ABM фреймворку MARS LIFE

Концепт	Призначення	Особливості
Agent	Незалежні та активні учасники процесу симуляції, що мають свій стан та можуть приймати рішення.	Кількість агентів в системі може бути задана в конфігурації та змінюватись в процесу виконання симуляції.
Entity	Пасивні учасники симуляції, що мають свій власний стан, але не можуть приймати рішення.	Кількість сутностей в системі може бути задана в конфігурації та змінюватись в процесу виконання симуляції.

Layer	Шари схожі на ігрові майданчики, де агенти є гравцями. Вони містять інформацію про інших гравців та об'єднують в собі певні аспекти симуляційної моделі. Пасивні шари можуть лише реагувати на дії інших учасників, тоді як активні можуть також приймати власні рішення на кожному кроці симуляції.	Шари існують в єдиному екземплярі на кожен тип шару і також можуть бути налаштовані на початку симуляції.
Environment	Просторові середовища для симуляції переміщень агентів, ефективного відслідковування відстаней та зіткнень між ними.	Шари можуть містити одне або більше різних середовищ.

В симуляції процесу розробки програмного забезпечення немає необхідності моделювання жодних просторових середовищ, тому даний елемент симуляційного фреймворку в даній роботі не використовується.

З метою оптимізації обчислень в процесі виконання симуляцій було обрано розмір кроку АВМ симуляції, рівний 1 хвилині. Дане значення дозволяє змодельовати процес розробки програмного забезпечення на достатньому рівні деталізації та одночасно зменшити швидкість виконання симуляцій.

Для генерації випадкових значень відповідно до загальновідомих імовірнісних законів розподілу, таких як нормальний розподіл, логнормальний, зворотній розподіл Гауса та Weibull, доцільно використати готову бібліотеку «MathNet.Numerics» [91].

Процес рецензування програмного коду, що вже став невід'ємною частиною професійної розробки програмного забезпечення, наразі активно досліджується науковою спільнотою, як в контексті позитивного впливу на якість програмного забезпечення, так і в контексті негативного впливу на продуктивність розробки [62]. Оскільки різні архітектурні моделі програмного забезпечення диктують певні можливості і обмеження модульної організації коду із застосуванням різних практик власності коду, що прямо впливає на

організаційні аспекти його рецензування та соціальну взаємодію між розробниками, то, для детального аналізу впливу типу архітектури на продуктивність розробки, є необхідним також моделювання процесу рецензування програмного коду в агентно-орієнтованих симуляціях процесів розробки програмного забезпечення.

Аналіз останніх досліджень процесів рецензування програмного коду показав активний інтерес як від наукової спільноти до детальнішого вивчення цих процесів зсередини, так і зі сторони індустрії розробки програмного забезпечення для покращення даних процесів. Одні дослідження націлені на оптимізацію процесу рецензування загалом шляхом пошуку оптимальної стратегії розподілу запитів на рецензування коду поміж розробниками [63, 65, 68], інші досліджують способи передбачення часу завершення рецензування [49] або файлів вихідного коду, що потребують найбільшої уваги [92]. Решта дослідників вивчають емпіричні дані для більш глибокого розуміння механіки роботи процесів рецензування коду та взаємодії розробників між собою [64, 66, 67]. Результати даних досліджень можуть бути використані для параметричного моделювання процесу рецензування програмного коду на достатньому рівні деталізації в агентно-орієнтованих симуляціях розробки програмного забезпечення.

Розбиття кодової бази на окремі частини в модульній монолітній та мікросервісній архітектурах дозволяє застосовувати різні види практики власності коду щодо окремих модулів або сервісів. Однією з основних переваг мікросервісної архітектури є незалежність окремих сервісів один від одного, що забезпечує можливість їх автономної розробки та розгортання, саме тому, зазвичай, кожен сервіс закріплений за окремою командою розробників і лише вони можуть вносити в нього зміни. Даний вид власності коду називається сильною власністю. Модульна монолітна архітектура дозволяє застосування як сильної власності коду, так і слабкої, коли будь-який розробник може вносити зміни до програмного коду будь-якого модуля, проте команда розробки, під чиєю власністю знаходиться код певного модуля, зобов'язана здійснити рецензування

внесених змін до їх модуля для забезпечення необхідного рівня якості коду. Багатошаровій монолітній архітектурі зазвичай притаманна колективна власність коду без чітких розмежувань між розробниками.

Попередні дослідження процесів рецензування вказують на те, що попередня експертиза в певній області програмного коду зменшує кількість пропущених дефектів та витрати часу на рецензування [63], отже модульна структура кодової бази та різні види власності коду мають бути враховані в симуляційній моделі процесу рецензування коду. Для забезпечення даної вимоги, симуляційна модель процесу розробки програмного забезпечення повинна також моделювати модульну кодову базу, де для кожного модуля (сервіса) має бути призначена власність коду. На кожній ітерації симуляції при внесенні змін до програмного коду необхідно обирати модулі, в які вносяться зміни, та симулювати процес рецензування коду по кожному модулю окремо, в залежності від застосованого виду власності коду. Рецензування всієї транзакції внесення змін до вихідного коду має вважатись завершеним, лише коли завершене рецензування всіх окремих модулів, що входять в дану транзакцію.

Останні дослідження часу, необхідного на рецензування змін до програмного коду, показують, що час рецензування корелюється не так із зусиллями, витраченими на первинне внесення цих змін, як із кількістю змінених рядків програмного коду, де на рецензування кожних додаткових 100 рядків витрачається близько 25 хвилин [93]. Враховуючи дану кореляцію, модель кодової бази в симуляціях процесу розробки має також визначати розмір кожного модуля в кількості рядків вихідного коду. Так як попереднє дослідження показує, що частота зміни модуля в загальному випадку має лінійну залежність від його розміру, то є доцільним параметризувати симуляційну модель процесу розробки такими характеристиками кодової бази, як загальна кількість рядків коду та кількість окремих модулів або сервісів. Згідно вже досліджених параметрів розподілу, з даних параметрів можна вирахувати як частоти зміни окремих модулів, так і їх розмір. Варто враховувати, що наведені витрати часу на рецензування не включають в себе ані час очікування запиту на

рецензування в черзі, ні накладні витрати розробників при переключенні контексту з програмування на рецензування, що також мають бути окремо змодельовані.

В агентно-орієнтованих симуляціях доцільно, щоб самі агенти-розробники приймали рішення коли займатись власною розробкою, а коли перемикатись на рецензування програмного коду інших. Часте перемикання контексту для розробників є небажаним, тому зміна активності зазвичай відбувається після перерв, серед яких можна виділити початок робочого дня, обідня перерва, завершення наради, призупинення чи завершення рецензування коду чи окремої сесії роботи над поточною задачею.

При модулюванні процесу рецензування програмного коду для агентно-орієнтованих симуляцій також варто врахувати людський фактор, що також може впливати на продуктивність розробки в певному контексті. Так, наприклад, людині властиві втома та нудьга, що також тісно пов'язані з процесом рецензування. Втома повільно накопичується протягом робочого дня та може посилюватись при безперервному рецензуванні декількох запитів поспіль через підвищене розумове навантаження, що призводить до покрокового зниження ймовірності, що певний розробник візьме на рецензування ще один запит. Нудьга при довготривалому рецензуванні, як декількох різних пакетів змін, так і одного великого, лише підсилює цей ефект. В агентно-орієнтованій симуляції процесу розробки доцільно ввести додатковий атрибут агента, який відображає коефіцієнт втоми розробника. Значення даного коефіцієнта має поступово зменшуватись в залежності від виду активності агента, частково відновлюватись після перерви та повністю скидатись до початкового значення на початку дня.

Суттєвий вплив на затримку запитів на рецензування програмного коду в черзі мають також різні часові пояси при географічному розподілі команд розробки, коли початок та кінець робочого дня може бути різний як у різних команд, так і у різних розробників однієї команди. Так, запит на рецензування може бути створений посеред дня в одному часовому поясі, а саме рецензування почнеться вже наступного робочого дня в іншій часовій зоні. Виходячи з цього,

різниця в часових поясах може призводити до затримки рецензування в контексті будь-якого типу архітектури чи виду власності коду, хоча при географічному розподілі команд розробки частіше обирають мікросервісну архітектуру і сильну власність коду. Отже, різниця в часових поясах різних розробників також має бути врахована в симуляційній моделі процесу розробки програмного забезпечення в якості вхідного параметру.

Рецензування програмного коду – це ітеративний процес, в якому під час рецензування виявляються функціональні або технічні дефекти і рецензент відкладає дану задачу, поки автор змін не виправить існуючі дефекти. В залежності від кваліфікації автора, об'єму внесених змін та експертизи рецензента, ймовірність виявлення дефектів, а відповідно і кількість ітерацій рецензування може відрізнятись. Опитування розробників з різних компаній виявило, що 35-40% запитів на рецензування коду приймаються без внесення правок [94]. Це значення може бути використане в якості початкової ймовірності схвалення змін до програмного коду. Якщо ж в процесі рецензування були виявлені дефекти, то код повертається автору на доопрацювання, після чого процес рецензування продовжується, але ймовірність повторної відправки програмного коду на доопрацювання на кожній наступній ітерації рецензування має зменшуватись, поки всі зміни не будуть остаточно схвалені. Даний ітеративний процес має відбуватись паралельно для кожного модуля, що входить в поточну транзакцію внесення змін до програмної системи, проте ймовірність повернення програмного коду на доопрацювання має застосовуватись до всього пакету змін до всіх модулів разом. На кожній ітерації повернення коду вибір підмножини модулів, що потребують додаткових змін, має відбуватись лише серед тих модулів, зміни до яких ще не були схвалені. Причинами повернення програмного коду на доопрацювання можуть бути або необхідність маленьких виправлень (помилки в документації, відсутні тести тощо), або дефекти в самій запрограмованій логіці [67]. Для врахування всіх вищенаведених аспектів, коефіцієнт зменшення ймовірності повернення програмного коду на доопрацювання та ймовірності розподілу причин повернення можуть бути

представлені в якості параметрів симуляції. Схема ітеративного процесу рецензування модульного коду зображена на рис. 2.10.

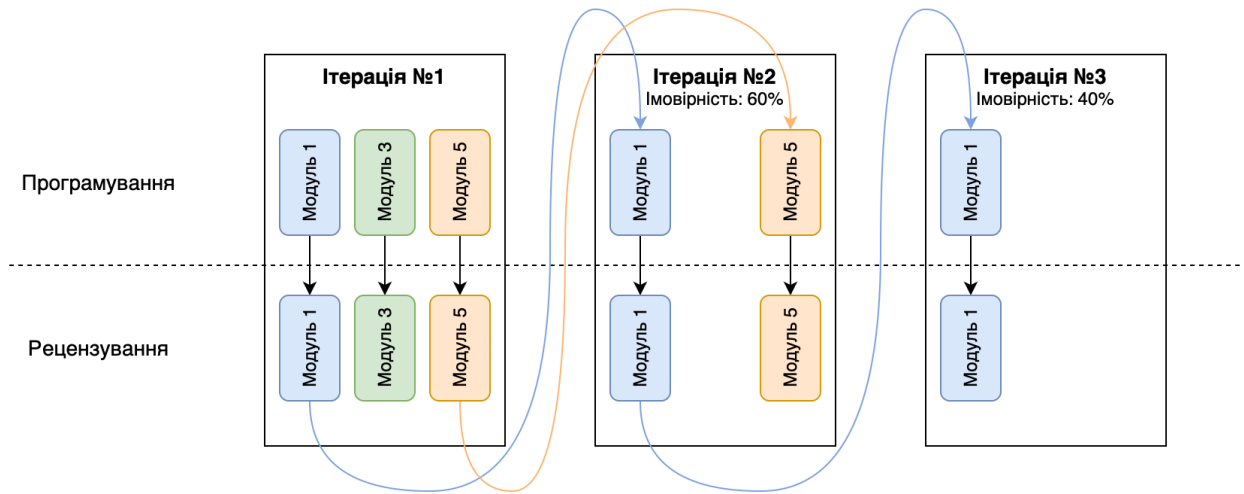


Рис. 2.10. Ітеративний процес рецензування модульного коду

Оскільки метою є моделювання процесу рецензування програмного коду саме для дослідження впливу архітектурної моделі на продуктивність розробки, то немає необхідності моделювати відхилення запитів на рецензування чи відхилення внесених змін загалом. З тієї ж причини, немає необхідності в моделюванні соціальних зв'язків між розробниками, що, як показують попередні дослідження, має суттєвий вплив на час очікування запиту на рецензування в черзі [64]. Затримка старту рецензування не потребує включення в симуляційну модель, але результати вимірювань з попередніх досліджень можуть бути використані для коригування параметрів симуляції для більшої відповідності емпіричним значенням.

Висновки до другого розділу

У даному розділі було проаналізовано розподіл складності задач розробки програмного забезпечення шляхом аналізу історичних даних проєктів з відкритим вихідним кодом, що входять до фонду «Apache Software Foundation». Було спростовано гіпотезу щодо відповідності історичних даних логнормальному розподілу імовірностей та запропоновано альтернативу у вигляді зворотного розподілу Гауса, що краще описує розподіл витраченого часу

на задачі з набору історичних даних, за виключенням аномально високої фактичної імовірності найпоширенішого значення в 20 хвилин. Для компенсації такої аномалії було запропоновано адаптацію алгоритму імовірнісного розподілу та обчислено відповідні числові параметри, а також запропоновано алгоритм обмеження максимального значення імовірнісного розподілу без спотворення його форми. Отриманий алгоритм розподілу імовірностей для генерації потоку вхідних задач різної складності є наближеним до фактичного розподілу витраченого часу з історичних даних та може бути використаний в низькорівневомих дискретно-подійному чи агентно-орієнтованому моделюванні процесу розробки програмного забезпечення для подальшого його дослідження.

Також були проаналізовані 8 модульних проєктів з відкритим вихідним кодом, як в монолітній архітектурі, так і у мікросервісному виконанні для дослідження алгоритму моделювання зв'язності між програмними модулями для низькорівневої симуляції процесу розробки програмного забезпечення на основі аналізу історії внесення змін. Всі проєкти різного розміру, містять різну кількість модулів та різну довжину історії змін. В результаті дослідження модульної структури та історії змін вихідного коду виявлено наявність логнормального імовірнісного розподілу як розміру програмних модулів, так і частоти їх модифікації, характер залежності між якими є лінійним, але параметри кореляції відрізняються між різними проєктами, що може бути зумовлено як різними мовами програмування, так і різною внутрішньою складністю коду модулів. Аналіз зв'язності модулів на основі видобування правил асоціації з історії їх незалежних та одночасних змін виявив степеневу функцію залежності середніх значень як вхідної, так і вихідної зв'язності від частоти зміни модуля, але параметри такої залежності також різні для різних проєктів та корелюються із кількістю модулів. З метою оптимізації обчислень при симуляції процесу розробки програмного забезпечення було запропоновано використання незалежних частот зміни модулів замість детального моделювання зв'язності кожної окремої їх пари. Для цього було досліджено параметри логнормального розподілу незалежних імовірностей зміни модулів для кожного проєкту та за

допомогою лінійної регресії виведено коефіцієнти залежності параметрів такого розподілу від кількості модулів в програмній системі, що моделюється. Так як всі імовірності зміни модулів незалежні, то існує ненульова імовірність невнесення змін в жоден модуль в процесі симуляції, що впливає на її обчислювальну ефективність. Тому було додатково запропоновано алгоритм коригування згенерованих частот зміни модулів для збільшення імовірності внесення змін хоча б в один модуль на кожній ітерації симуляції. Даний підхід до моделювання зв'язності модулів може бути застосований в дискретно-подійних та агентно-орієнтових симуляціях процесу розробки програмного забезпечення для подальшого дослідження впливу модульності на продуктивність розробки. Перспективним є додаткове дослідження еволюції модульної структури програмних систем, оскільки кількість, розмір та зв'язність модулів не є сталими в часі і також змінюються в процесі розробки таких систем.

На основі аналізу сучасних підходів до симуляційного моделювання процесів розробки програмного забезпечення, серед яких агентно-орієнтоване, дискретно-подійне, стохастичне моделювання, системна динаміка та гібридні методи, було обґрунтовано вибір агентно-орієнтованого підходу як найбільш придатного для досягнення поставленої мети. Даний вибір зумовлений тим, що агентно-орієнтоване моделювання забезпечує найвищу гнучкість при відтворенні складних взаємодій між учасниками процесу розробки, дозволяє моделювати динамічну поведінку системи та виявляти раніше невідомі взаємозв'язки між її елементами. Водночас було зазначено, що існуючі дослідження моделюють процес розробки переважно на рівні етапів життєвого циклу проєкту, що є недостатнім рівнем деталізації для вивчення впливу програмної архітектури на ефективність розробки. Визначено ключові вимоги до програмного комплексу для проведення симуляцій, зокрема забезпечення можливості виконання великої кількості симуляційних ітерацій з фіксованими вхідними параметрами, але з різними випадковими початковими значеннями для отримання статистично достовірних усереднених результатів. Обґрунтовано важливість відтворюваності симуляцій для забезпечення верифікації результатів,

виявлення помилок, документованості та прозорості досліджень, а також можливості їх незалежної перевірки іншими дослідниками.

Розроблено та описано багаторівневу архітектуру розробленого програмного комплексу та обґрунтовано вибір фреймворку MARS LIFE як основи для побудови агентно-орієнтованих симуляцій. Встановлено, що оптимальний розмір кроку симуляції становить одну хвилину, що забезпечує достатню деталізацію процесу розробки при прийнятній обчислювальній ефективності. Окрему увагу приділено моделюванню процесу рецензування програмного коду як невід’ємної складової сучасної розробки програмного забезпечення. Обґрунтовано необхідність врахування в симуляційній моделі модульної структури кодової бази, різних видів власності коду, ітеративної природи процесу рецензування програмного коду, а також таких факторів, як втома та нудьга розробників, географічний розподіл команд з різницею в часових поясах та тривалість сесій програмування. Було виявлено залежність часу рецензування від об’єму зміненого коду та запропоновано підхід до розрахунку необхідних витрат часу в програмних симуляціях процесу розробки модульного програмного забезпечення в залежності від кількості змінених рядків програмного коду. Таким чином, спроектований програмний комплекс дозволяє проводити детальне агентно-орієнтоване моделювання процесів розробки програмного забезпечення на рівні, достатньому для порівняльного аналізу впливу різних архітектурних підходів на продуктивність розробки, з урахуванням усіх ключових аспектів реального процесу розробки та рецензування програмного коду.

Матеріали розділу 2 подано в таких публікаціях автора [95, 96, 97, 98, 99, 100].

РОЗДІЛ 3. Симуляційне моделювання процесів розробки програмного забезпечення для різних типів архітектури серверних програмних систем

3.1. Агентно-орієнтована симуляційна модель процесу розробки ПЗ для різних типів програмної архітектури

Симуляційна модель процесу розробки програмного забезпечення створена з метою аналізу продуктивності, ефективності та часових характеристик процесу розробки у різних архітектурних середовищах. Ця модель включає кілька ключових компонентів, кожен з яких виконує певну функцію у симуляції, що дозволяє вивчати взаємодію між різними факторами, такими як тип архітектури кодової бази, розподіл відповідальності за модулі, ефективність рецензування коду та тестування. Симуляційний процес починається з формування потоку завдань, який включає неперервну подачу нових функціональних вимог, а також періодичні випадкові події у формі помилок та завдань технічного обслуговування. Цей потік подається у систему шляхом моделювання випадкових подій, що дозволяє аналізувати вплив різних факторів на продуктивність команд розробки. Помірність цього потоку впливає на загальну продуктивність команд, оскільки завдання, які з'являються в черзі, мають бути виконані у порядку введення або згідно з пріоритетами, що може викликати ефекти контекстного переключення. Крім того, кожне завдання має визначену тривалість розробки та тестування, а також може вимагати роботи з іншими модулями, що впливає на загальні показники роботи.

Центральним елементом моделі є групи спеціалістів, які об'єднані в окремі команди. Кожна команда має задану кількість програмістів та тестувальників, що визначає ресурсні можливості для виконання завдань. Структура команд може варіюватися, що дозволяє моделювати різні сценарії управління проєктами, включаючи розподілену або концентровану організацію праці. Кожна команда відповідає за модулі кодової бази, а розподіл відповідальності (слабка чи сильна власність коду) визначає, хто може змінювати модуль та як це впливає на процес

рецензування коду. Наприклад, у випадку сильного володіння модуль не може змінюватися сторонніми розробниками, а лише його власниками, тому і рецензування коду мають проводити розробники з тієї ж команди. Такий підхід дозволяє моделювати вплив управління доступом до кодової бази на продуктивність та час виконання завдань.

Кодова база, яка є об'єктом симуляції, може мати різну архітектурну структуру: моноліт, модульний моноліт або мікросервісна система. Кожна з цих архітектур має свої особливості, які впливають на спосіб модифікації, тестування та інтеграції змін. Наприклад, у монолітній архітектурі зміни в одному модулі можуть викликати необхідність перевірки всього коду, тоді як у мікросервісній – зміни можуть бути локалізованими. Крім того, кожен модуль кодової бази має визначену ймовірність модифікації, яка залежить від типу завдання (наприклад, функціональна вимога, виправлення дефекту або технічне обслуговування). Цей параметр визначається шляхом алгоритмічного розрахунку, що дозволяє моделювати нерівномірність розподілу завдань між модулями.

Кожна зміна коду, що вноситься в систему, може потребувати перевірки. Складність цього процесу визначається параметром, який описує зусилля, витрачені на перевірку коду, у вигляді відсотка від часу програмування. Крім того, кожне завдання може вимагати доопрацювання – цей процес також має свою вартість у формі відсотку від початкових зусиль на програмування, що зменшується на кожній наступній ітерації доопрацювання.

Для виконання аналізу, модель збирає кілька ключових метрик, які визначають продуктивність системи. Серед них – середня тривалість циклу функціональних завдань, яка визначає, наскільки швидко команда може виконувати запит на нову функціональність. Також аналізується пропускна здатність, що вимірює кількість завдань, які виконуються за одиницю часу. Контекстне переключення, яке виникає при зміні пріоритетів або завдань, вимірюється через час, який не використовується для продуктивної роботи, а витрачається на переключення між різними завданнями. Накладні витрати га рецензування коду аналізуються шляхом визначення відсотка часу, який

затрачається на ці процеси. Крім того, вимірюються відсотки часу, витраченого на тестування, простій тестувальників та загальний рівень "роботи в процесі", що дозволяє оцінювати ефективність використання ресурсів.

Всі параметри симуляції, такі як тип архітектури, розподіл відповідальності, ефективність рецензування коду та тестування, кількість модулів у кодовій базі та склад команд розробників, можуть бути налаштовані з метою вивчення їх впливу на оцінювані метрики. Наприклад, збільшення кількості модулів у модульній монолітній архітектурі може збільшити час, витрачений на інтеграцію змін, або призвести до необхідності збільшення витрат часу на тестування таких змін. Також збільшення частоти виникнення помилок або завдань технічного обслуговування може змінювати загальний баланс між функціональною розробкою та виправленням дефектів, що впливає на пропускну здатність системи та інші вихідні метрики.

Дана модель може бути використана для аналізу ефективності різних архітектурних підходів, способів управління кодовою базою, розподілу відповідальності між командами та різноманітних ефектів, що виникають при зміні параметрів процесу. Також вона дозволяє проводити порівняльні експерименти, наприклад, визначити, яка архітектура краще підходить до визначеного типу завдань, або як зміна кількості розробників і тестувальників впливає на загальну продуктивність команди. Основні концептуальні компоненти симуляційної моделі зображені на рис. 3.1.

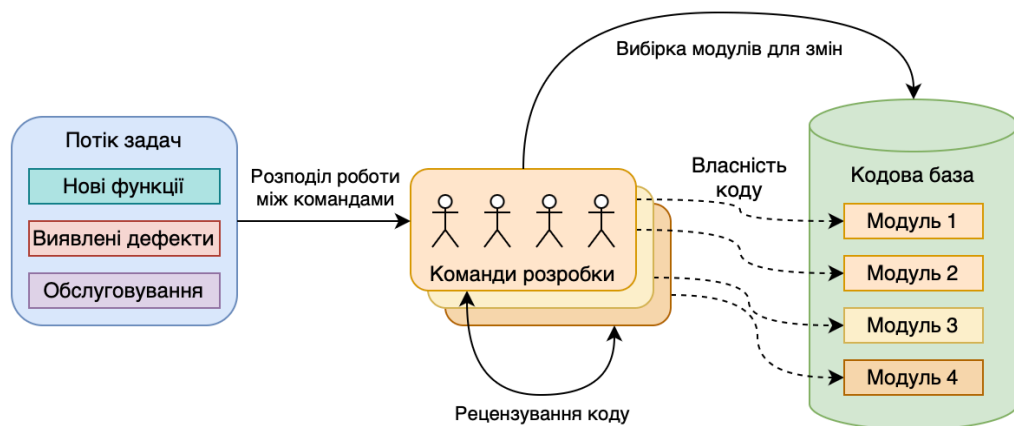


Рис. 3.1. Компоненти симуляційної моделі процесу розробки ПЗ

При ініціалізації симуляції створюються відповідні команди розробки, генерується модель кодової бази відповідного розміру з відповідною кількістю модулів та їх характеристик та у випадку сильної чи слабкої власності коду кожен модуль асоціюється з певною командою-власником.

Потік задач для програмування нових функцій генерується у «лінивому» режимі, коли виникає попит на нові задачі. Виявлені дефекти додаються в потік відповідно до імовірності їх виявлення, що виражається у відсотку завершених нових функцій. Задачі на технічне обслуговування розподілені в часі, а імовірність їх виникнення більша для мікросервісної архітектури, так як кожен окремий сервіс вимагає власного обслуговування, на відміну від монолітних архітектурних форм. Розподіл задач всіх видів між окремими командами розробки відбувається в залежності від виду власності коду, що використовується в даному програмному проєкті.

Стадії розробки нових функцій та їх взаємозв'язок з відповідними задачами розробки зображено на рис. 3.2.

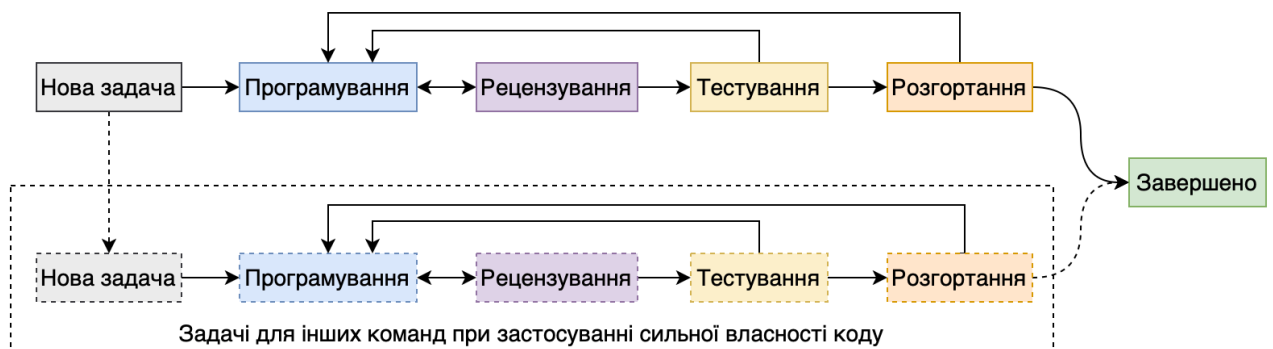


Рис. 3.2. Стадії розробки нових функцій

При генерації потоку вхідних задач, задачі генеруються групами, де кожна група об'єднана певною новою функціональністю програмної системи. При сильній власності коду, кожна задача автоматично призначається відповідальній команді розробки та передбачає зміни лише в тих програмних модулях, кодом яких володіє дана команда. Таким чином, щоб певна нова функціональність системи вважалась завершеною, необхідно, щоб всі дотичні команди розробки

виконали відповідні задачі в своїй області відповідальності. При колективному та слабкому виді власності коду всі задачі додаються в загальний потік та можуть бути виконані будь-якими командами, проте при слабкій власності коду інші команди, що відповідають за певні модулі, повинні провести рецензування внесених змін.

Задачі ж на виправлення виявлених дефектів та обслуговування або можуть бути виконані будь-якою командою розробки при колективній власності коду, або мають бути призначені конкретній команді, що володіє кодом відповідних модулів чи сервісів.

Після завершення початкового етапу програмування відповідна задача переходить в стадію рецензування, коли, відповідно до виду власності коду та змінених модулів, обираються команди розробки, що мають провести рецензування певної частини змін до коду. При колективній власності коду будь-який розробник з будь-якої команди може взяти задачу на рецензування всього набору змін. При слабкій власності коду, зміни у відповідні програмні модулі мають бути перевірені розробниками з відповідних команд. При сильній власності коду незалежно від типу архітектури, так як лише команда-власник може змінювати програмний код в певних модулях, то лише розробники з тієї ж команди приймають участь в рецензуванні змін до коду власних модулів.

Так як процес рецензування коду ітеративний за своєю природою, то на кожній ітерації кожен учасник-рецензент може або схвалити внесені зміни до коду певних модулів, або відправити їх на доопрацювання. Для досягнення емпіричного значення у 40% запитів на рецензування, що схвалюються без додаткових доопрацювань, розрахунок імовірності повернення змін до коду на доопрацювання кожним окремим рецензентом для кожного окремого програмного модуля відбувається відповідно до формули (3.1):

$$p_i = 1 - \sqrt[M \cdot N]{1 - P_k}, \quad (3.1)$$

де P_k – імовірність відправки змін до коду на доопрацювання на k -тій ітерації рецензування, M – загальна кількість модулів, що рецензуються на даній ітерації, N – мінімальна кількість рецензентів, що мають схвалити зміни, p_i – імовірність

відправки програмного коду окремого модуля на доопрацювання окремим рецензентом. На кожній наступній ітерації рецензування P_k поступово зменшується з кроком, що задається в налаштуваннях симуляції.

Після схвалення всіма учасниками останньої ітерації процесу рецензування, задача переходить в стадію тестування. Сам процес тестування внесених змін до коду починається лише тоді, коли будь-який тестувальник з поточної команди розробки завершить перевірку попередньої задачі або відправить її на доопрацювання та приступить до поточної. Процес тестування також є ітеративним, але не враховує вид власності коду та модульність кодової бази. З одного боку, монолітна архітектура потребує більших зусиль на тестування через вищу імовірність внесення дефектів в залежні модулі, проте з іншого боку, кожен сервіс в мікросервісній архітектурі вимагає додаткове тестування безпекових та інфраструктурних аспектів, тому в даній симуляційній моделі витрати часу на тестування не залежать від типу архітектури програмної системи. Імовірність схвалення змін до коду на кожній наступній ітерації тестування збільшується на відсоток, що задається в налаштуваннях симуляції.

Процеси розгортання програмних систем для монолітної та мікросервісної кодових баз суттєво відрізняються. Монолітні програмні системи мають розгортатись як одне ціле, навіть якщо зміни були внесені лише в один окремий модуль. При чому, якщо при розгортанні були виявлені проблеми, що потребують виправлення, то наступні пакети схвалених змін до програмного коду повинні чекати, поки всі поточні проблеми не будуть виправлені. В мікросервісній же архітектурі кожен окремий сервіс може бути розгорнутий незалежно від інших. Крім того, процес розгортання незалежного мікросервіса швидший за велику монолітну систему, оскільки є меншим за розміром та потребує менше часу на виконання всіх додаткових перед-релізних автоматизованих перевірок. Розгортання програмної системи також має імовірність виявлення певних проблем та бути повернуто розробнику на виправлення знайдених дефектів.

Відповідно до існуючих концептів в АВМ фреймворку MARS LIFE та необхідного рівня деталізації моделі процесу розробки програмного забезпечення були виокремлені наступні елементи симуляційної моделі.

З метою моделювання людських факторів та комплексного алгоритму прийняття рішень, симуляційна модель містить такі види агентів:

- Developer – агент-розробник, що займається програмуванням нових функцій, виправленням дефектів та рецензуванням коду.
- Tester – агент-тестувальник, який перевіряє виконані задачі на предмет наявності дефектів.

Модель також містить певні сутності, що самостійно не приймають ніяких рішень, а представляють лише ресурси, якими оперують інші учасники симуляції:

- Team – команда розробки, що об'єднує як розробників, так і тестувальників. Тестувальники перевіряють лише задачі, виконані розробниками з їх команди. В контексті власності програмного коду, модулі призначаються у власність команд, а не окремих розробників.
- Backlog Epic – нова функція програмної системи (узагальнена користувацька історія або ініціатива), що містить окремі задачі для програмування різними командами розробки.
- Backlog Task – задача програмування нової функції або виправлення дефекту. Містить час, необхідний на її виконання, перелік модулів програмного коду, куди вносяться зміни та зберігає поточний стан роботи на задачею.
- Codebase – кодова база програмного продукту. Містить інформацію про тип архітектури, вид власності програмного коду, загальну кількість рядків коду та список програмних модулів.
- Codebase Module – програмний модуль системи з інформацією про його відносний розмір у відсотках, команду-власника вихідного коду модуля та його взаємозалежності з іншими модулями.

Для моделювання організаційних аспектів процесу розробки програмного забезпечення та координації агентів було визначено наступні шари симуляції:

- **Software Product** – кореневий пасивний шар, що об’єднує всю інформацію про програмний продукт. Містить в собі кодову базу та структуру команд розробки.
- **Backlog** – пасивний шар, що відповідає за вхідний потік нових задач, об’єднаних у високорівневі нові функції програмної системи, відслідковує моменти їх початку та завершення та симулює виявлення пропущених дефектів. Також відповідає за розподіл задач між командами розробки відповідно до заданого виду власності коду.
- **Work Schedule** – активний шар, що відслідковує поточний час доби в контексті різних часових поясів та визначає час роботи, відпочинку, перерви на обід та перехідні моменти між ними. Використовується агентами для прийняття рішень щодо їх активності.
- **Code Review** – пасивний шар, що відслідковує задачі в процесі рецензування, відповідає за їх розподіл між агентами-рецензентами, керує процесами переходу задач до рецензування і повернення на доопрацювання та контролює необхідні умови для переведення задач на наступний етап.
- **Testing** – пасивний шар, що відслідковує задачі в процесі верифікації, розподіляє їх між агентами-тестувальниками та моделює імовірнісний процес виявлення дефектів.
- **Deployment** – активний шар, що симулює розгортання програмного продукту після внесення в нього змін в залежності від типу архітектури програмної системи.
- **Telemetry** – пасивний шар, який використовується іншими шарами та агентами для збору інформації про події, що відбуваються в процесі симуляції та відповідає за збереження потоку всіх подій у вихідний CSV файл табличного формату.

На рис. 3.3 зображена діаграма класів композиції шарів.

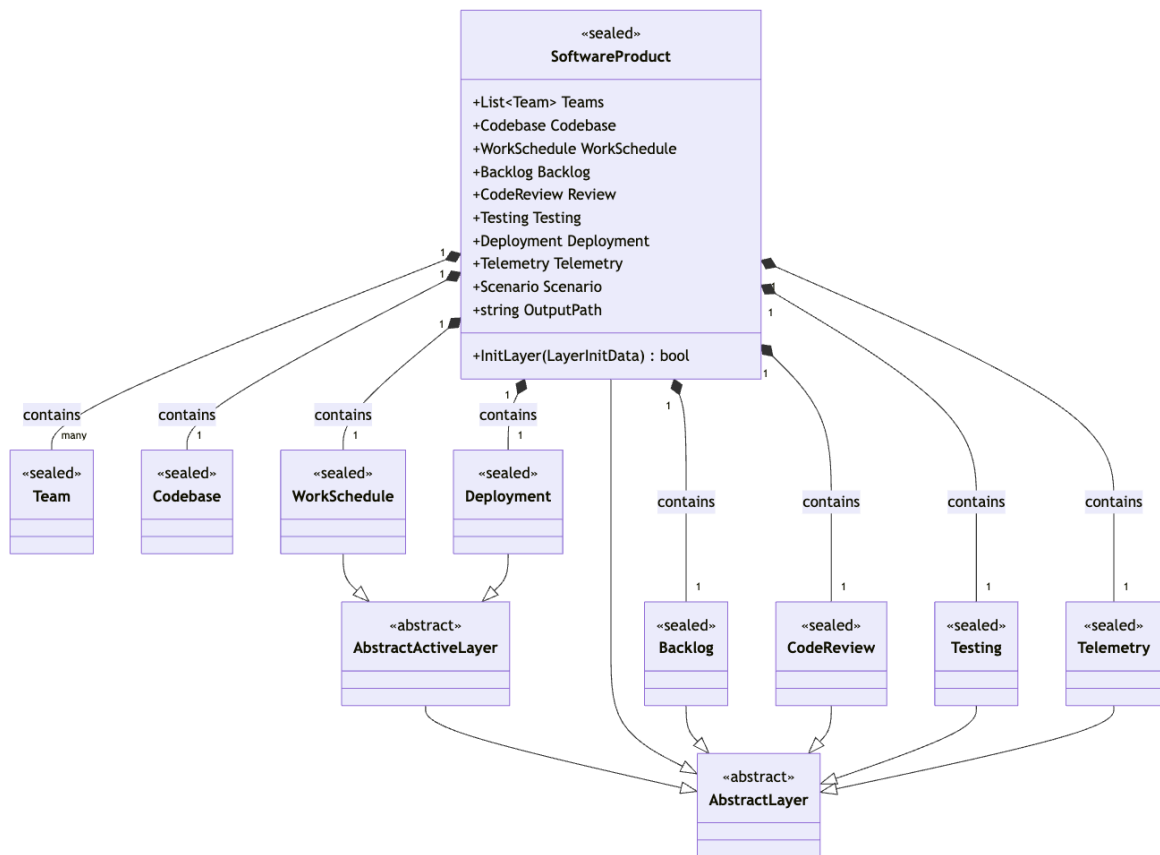


Рис. 3.3. Діаграма класів композиції шарів симуляції

Агенти напряду між собою не взаємодіють. Натомість, агенти повідомляють шарам, які відповідають за окремі аспекти процесу розробки програмного забезпечення, про результати своєї роботи або запитують нові задачі для програмування, рецензування та тестування. Кожен агент на початку кожного кроку симуляції запитує у шару робочого розкладу поточну фазу дня та відповідно на неї реагує. Так, жоден агент не буде виконувати жодну роботу в неробочий час. Діаграма класів агентів зображена на рис. 3.4.

В процесі виконання симуляції всі агенти та деякі шари генерують певні події, що відбуваються з ними або іншими сутностями, та звітують їх шару «Telemetry», який гарантує збереження послідовності подій, додає до них порядковий номер поточної ітерації агентно-орієнтованої симуляції та вивантажує записи про всі події у вихідний файл у фоновому режимі, не сповільнюючи сам процес симуляції.

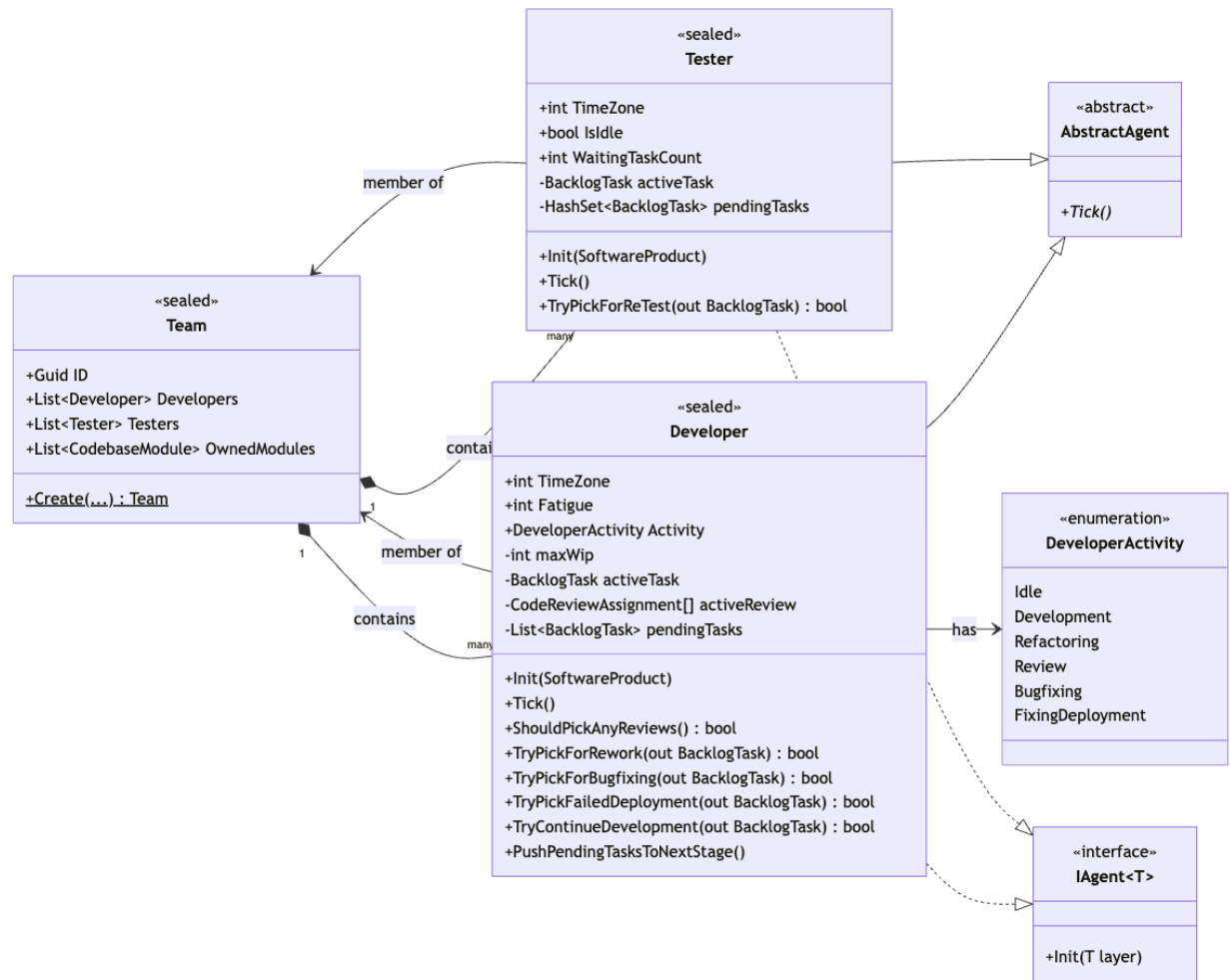


Рис. 3.4. Діаграма класів агентів

Прикладами таких подій є початок роботи над задачею, завершення процесу рецензування коду, успішність або неуспішність розгортання нової версії програмної системи та інші. Повний перелік всіх подій, що генеруються симуляційною моделлю, приведено в таблиці А.1 додатку А.

Повний вихідний код проєкту агентно-орієнтованої симуляції на основі фреймворку MARS LIFE наведений в додатку Б та доступний для завантаження на платформі Zenodo [101].

3.2. Аналіз результатів симуляцій процесів розробки ПЗ в контексті різних архітектурних моделей

Процес розробки програмного забезпечення є складною соціотехнічною системою, продуктивність якої залежить від численних взаємопов'язаних факторів: архітектурних рішень, організаційної структури команд, характеристик кодової бази, географічного розподілу розробників та практик управління робочим процесом. Зрозуміти вплив кожного з цих факторів окремо та в їхній взаємодії є критично важливим завданням для прийняття обґрунтованих управлінських рішень.

В даному розділі аналізується набір даних, отриманий шляхом агентного моделювання процесу розробки, який дозволяє систематично варіювати вхідні параметри, зберігаючи контрольовані умови. Метою є виявлення кореляційних залежностей між змінними, ідентифікація аномалій та нетипових закономірностей, визначення факторів, що найбільше впливають на продуктивність, та формулювання рекомендацій щодо оптимальних параметрів для кожної архітектурної моделі.

Отриманий набір даних містить результати виконання 8080 симуляцій, що складаються з 808 симуляційних сценаріїв по 10 стохастичних запусків на кожен з них. Базовий набір з 12 сценаріїв формується зі всіх можливих комбінацій різних видів архітектури програмного забезпечення (монолітна, модульна монолітна та мікросервісна), типів власності коду для модульної монолітної архітектури та 3 конфігураціях розподілу команд розробки між часовими поясами (всі команди в одному часовому поясі; рівномірно розподілені між 2 поясами; рівномірно розподілені між 5 різними часовими поясами). Базова конфігурація симуляцій приведена в таблиці 3.1 та включає 5 команд по 4 розробники та 2 тестувальники, 50 модулів (окрім звичайної монолітної архітектури), 500000 рядків коду. В кожному запуску симуляції тривалість процесу розробки складає 60 послідовних робочих днів без врахування вихідних.

Таблиця 3.1.

Базова конфігурація параметрів симуляції

Параметр	Значення
Кількість команд	5
Кількість розробників в одній команді	4
Кількість тестувальників в одній команді	2
Кількість модулів (окрім монолітної архітектури)	50
Обсяг кодової бази	500000 рядків коду
Тривалість симуляції	60 робочих днів
Стохастичних запусків на сценарій	10

Кожен базовий сценарій симулюється багаторазово в розрізі окремих варіацій вхідних параметрів, що описані в таблиці В.1 додатку В.

Середня продуктивність суттєво відрізняється між архітектурними моделями та видами власності коду. Середні значення завершених функцій на одного розробника зведені в таблиці 3.2.

Таблиця 3.2.

Середні значення продуктивності для різних архітектурних моделей

Архітектурна модель	Власність коду	Середнє значення	Стандартне відхилення	Медіана	Максимальне значення
Мікросервісна	Сильна	15,6	5,7	18,2	26,3
Модульна монолітна	Сильна	4,5	2,7	4,1	15,9
Модульна монолітна	Слабка	8,8	3,5	9,3	22,9
Звичайна монолітна	Колективна	13,7	3,3	13,3	27,4

Мікросервісна архітектура демонструє найвищий середній показник продуктивності на одного розробника. Звичайна монолітна архітектура показує трохи нижчу середню продуктивність, але її максимальне значення є найвищим серед усіх моделей, що свідчить про можливість досягнення виняткової

продуктивності за оптимальних умов. Модульна монолітна архітектура зі слабкою власністю коду демонструє помірну продуктивність, тоді як модель з сильною власністю коду є найменш продуктивною, що вимагає подальшого аналізу причин такої поведінки. Так як найкраща та найгірша моделі відрізняються за середніми показниками у 3,5 рази, це свідчить про визначальну роль архітектурних рішень у продуктивності розробки. Графіки залежності сумарної кількості завершених функцій програмної системи від загальної кількості розробників при однаковому співвідношенні кількості розробників до тестувальників зображені на рис. 3.4.

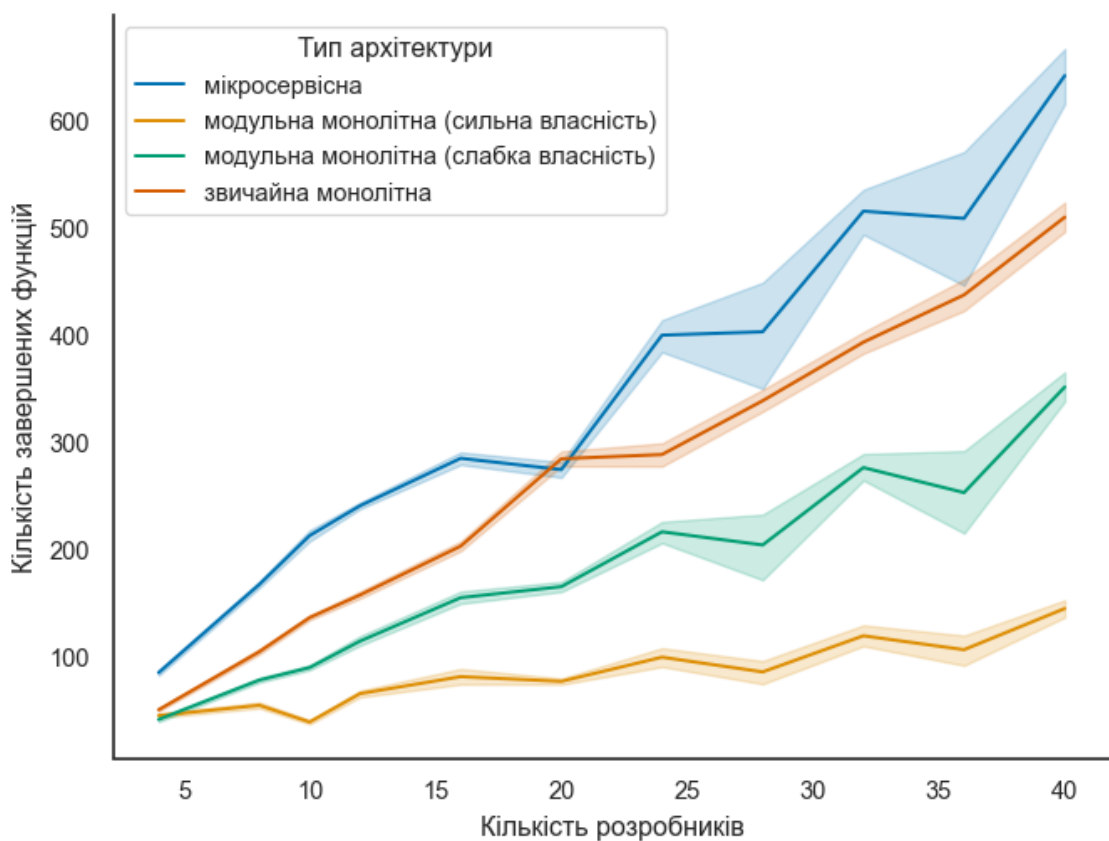


Рис. 3.4. Продуктивність розробки в залежності від кількості розробників

Як видно з графіків на рис. 3.4, різні архітектурні моделі відрізняються не тільки середнім значенням продуктивності, а і відповідною ефективністю масштабування розробки. Так як симуляції виконувались в розрізі різних варіативних параметрів, то аномалії, що спостерігаються при 20 розробниках

свідчать про додаткові кореляції продуктивності розробки з іншими вхідними параметрами, що потребує додаткового дослідження.

Таблиця 3.3 містить середні значення метрик, зібраних під час виконання симуляцій.

Таблиця 3.3.

Середні значення зібраних метрик

Метрика	Мікросервісна архітектура	Модульна монолітна (сильна власність коду)	Модульна монолітна (слабка власність коду)	Звичайна монолітна архітектура
Тривалість життєвого циклу підзадач (хвилин)	1649	9458	2734	5810
Тривалість відпочинку розробника (хвилин)	8172	21748	18682	11836
Тривалість відпочинку тестувальника (хвилин)	14138	1475	4490	12096
Відношення витрат на рецензування коду до витрат на розробку	0,56	0,56	0,66	0,63
Відношення витрат на тестування до витрат на розробку	0,33	5,88	2,21	0,33

Середня тривалість життєвого циклу підзадач (від початку розробки до успішного розгортання) демонструє суттєві відмінності між архітектурними моделями. Мікросервісна архітектура має найкоротшу середню тривалість, що відповідає її високій продуктивності. В звичайній монолітній архітектурі тривалість життєвого циклу задач в 1,66 разів більша, що пояснюється складнішим та повільнішим процесом розгортання монолітної кодової бази, так

як накладні витрати на тестування та рецензування коду несуттєво відрізняються від показників мікросервісної архітектури. Найвищий показник тривалості життєвого циклу задач спостерігається у модульній монолітній архітектурі з сильною власністю коду та перевищує відповідне значення мікросервісної архітектури майже у 6 разів. Основною причиною такої різниці є суттєве збільшення накладних витрат на тестування задач, що в контексті модульної монолітної архітектури з сильною власністю коду складає 588% та у 18 разів перевищує відповідне значення мікросервісної архітектури з таким же видом практики власності коду. Оскільки застосування практики сильної власності програмного коду вимагає розбиття функцій програмної системи на окремі підзадачі для кожного окремого модуля, які виконуються різними командами розробки в різні моменти часу, то у випадку модульної монолітної архітектури, кожна окрема команда після внесення змін у відповідні власні модулі кодової бази, повинна провести регресійне тестування не тільки власних модулів, а і всіх зв'язаних модулів. Витрати на рецензування коду значно менше варіюються між моделями, що свідчить про те, що процес рецензування є відносно інваріантним до архітектурної моделі.

Аналіз часу простою розробників та тестувальників розкриває фундаментальну асиметрію у завантаженості ролей. При мікросервісній архітектурі розробники мають найменший середній час простою, тоді як тестувальники відносно високий, що свідчить про збалансоване навантаження з деяким надлишком тестувальних ресурсів. Протилежна картина спостерігається у модульній монолітній архітектурі з сильною власністю коду, де розробники простоюють більше, а тестувальники суттєво менше, що зумовлено найбільшими накладними витратами процесу тестування. В контексті звичайної монолітної моделі обидві ролі мають приблизно рівний час простою, що свідчить про найбільш збалансований розподіл навантаження серед усіх моделей.

Для подальшого дослідження впливу різних входних параметрів симуляції на продуктивність розробки, доцільно провести кореляційний аналіз отриманого набору даних. В таблиці 3.4 зображено коефіцієнти кореляції Пірсона між

вхідними параметрами симуляцій та продуктивністю розробки на одного розробника.

Таблиця 3.4.

Коефіцієнти кореляції Пірсона між вхідними параметрами симуляцій та продуктивністю розробки на одного розробника

Параметр	Мікросервісна архітектура	Звичайна монолітна	Модульна монолітна з сильною власністю коду	Модульна монолітна зі слабкою власністю коду
Кількість часових поясів	-0,69	-0,33	-0,34	-0,59
Тривалість життєвого циклу підзадач	-0,56	-0,43	-0,47	-0,58
Відсоток часу тестування відносно часу розробки	0,00	0,01	-0,61	-0,40
Кількість модулів	0,10	-	-0,54	-0,37
Кількість рядків коду	0,04	-0,50	-0,25	-0,29
Кількість команд розробки	-0,09	-0,08	-0,26	-0,10
Максимальна кількість активних задач	0,06	0,20	-0,06	-0,01
Співвідношення кількості тестувальників до кількості розробників	0,17	0,01	0,47	0,32

Коефіцієнти кореляції Пірсона між вхідними змінними та вихідною продуктивністю розробки на одного розробника суттєво відрізняються залежно від архітектурної моделі. Це свідчить про різну природу обмежень у кожній моделі та підтверджує доцільність розгляду кожного типу архітектури окремо.

У випадку мікросервісної архітектури домінуючим негативним фактором є розподіл команд розробки між часовими поясами. Цікаво, що жоден з параметрів кодової бази (ні кількість модулів, ні обсяг коду) не має значущого впливу на продуктивність. Витрати на тестування мають нульову кореляцію, що підтверджує ефективність мікросервісного розмежування в ізоляції тестового

навантаження. Позитивна кореляція з кількістю тестувальників та розробників є слабкою, що свідчить про те, що продуктивність цієї моделі визначається переважно зовнішніми факторами (часові пояси) та внутрішньою ефективністю процесу розробки, а не масштабом ресурсів.

Для модульної монолітної архітектури з сильною власністю коду картина кардинально відрізняється. Найсильнішим негативним фактором є витрати на тестування, за яким слідує кількість модулів. Це пояснюється тим, що більша загальна кількість модулів програмного коду збільшує імовірність необхідності тестування більшої кількості зв'язаних суміжних модулів. Часові пояси мають помірний вплив, а от співвідношення тестувальників до розробників має найсильнішу позитивну кореляцію серед усіх моделей, що чітко вказує на тестування як вузьке місце всього процесу розробки. Так, збільшення кількості тестувальників безпосередньо підвищує продуктивність.

В контексті модульної монолітної архітектури зі слабкою власністю коду спостерігається проміжна картина: розподіл часових поясів та тривалість життєвого циклу задач є головними негативними факторами, накладні витрати на тестування та загальна кількість модулів також значущі, але менш виражені, ніж у випадку сильної власності програмного коду. Негативну кореляцію кількості різних часових поясів та тривалості життєвого циклу задач з продуктивністю розробки можна пояснити найсильнішою взаємозалежністю між різними командами розробки порівняно з іншими симуляційними сценаріями, оскільки кожна команда може вносити зміни до програмного коду будь-якого модуля, але вимушена чекати на інші команди-власники коду змінених модулів на етапі рецензування програмного коду.

Для звичайної немодульної монолітної кодової бази головним негативним фактором є загальна кількість рядків коду, так як за відсутності модульної організації програмного коду, вся кодова база безпосередньо визначає складність кожної зміни та повинна бути перевірена та розгорнута як єдине ціле. Часові пояси мають відносно помірний вплив, що є найнижчим показником серед усіх моделей, так як кожна команда діє абсолютно незалежно від інших команд.

Наступним кроком аналізу отриманих результатів симуляцій є дослідження впливу окремих факторів на продуктивність в контексті різних архітектурних моделей. Найбільш універсально значущим фактором, що демонструє статистично значущий ефект для всіх чотирьох моделей, хоча з суттєво різною інтенсивністю, є вплив географічного розподілу команд розробки поміж різних часових поясів. Графіки продуктивності процесу розробки в залежності від кількості різних часових поясів зображено на рис. 3.5.

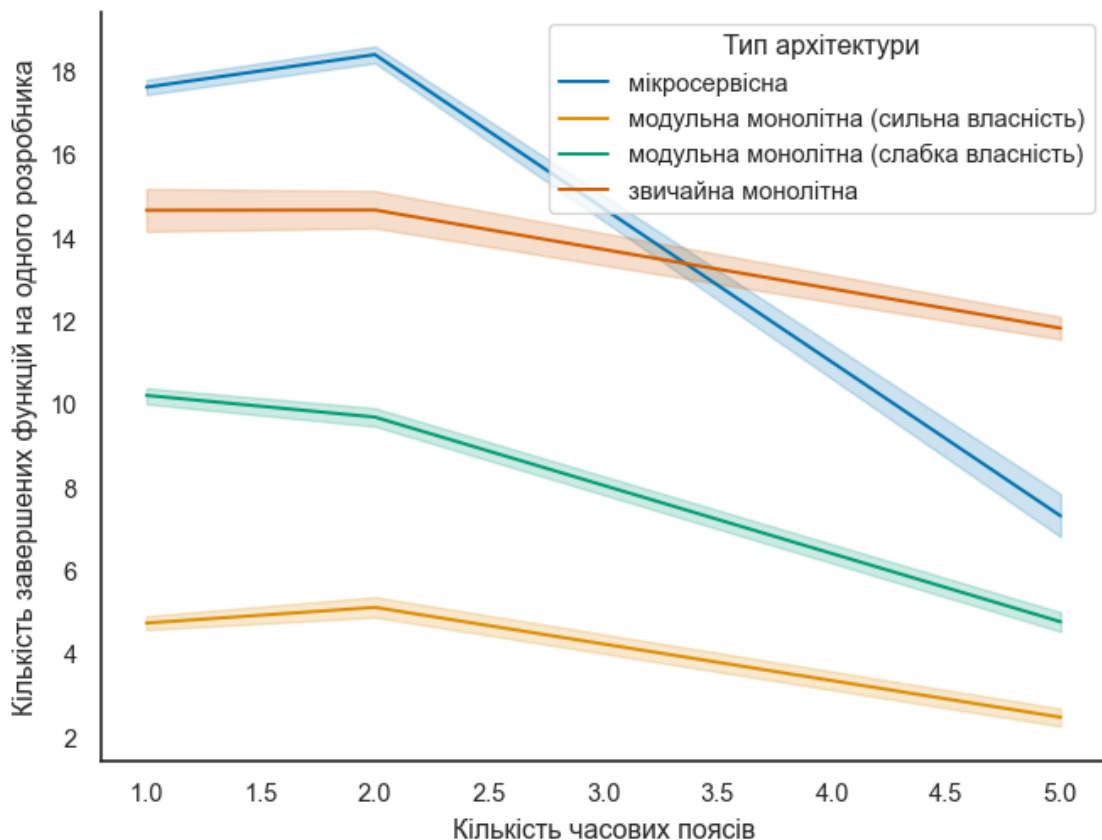


Рис. 3.5. Продуктивність розробки в залежності від кількості різних часових поясів

Перехід від 1 до 2 часових поясів дає несподіваний результат: для мікросервісної архітектури продуктивність розробки навіть незначно зростає (+3,6%, з 17,9 до 18,5 функцій на розробника), як і для модульної монолітної архітектури з сильною власністю коду (+3,3%). У випадку звичайної монолітної програмної системи, суттєвого приросту продуктивності не спостерігається (лише +0,8%), а от при слабкій власності коду в модульній монолітній

архітектурі, навпаки, спостерігається помірне зниження продуктивності процесу розробки (-5,0%). Це може свідчити про те, що розподіл по двох часових поясах при застосуванні сильної власності коду створює ефект «подовженого робочого дня», де команди частково перекриваються у часі, забезпечуючи більш тривалий щоденний робочий цикл без критичних затримок у комунікації. Натомість перехід до 5 часових поясів є катастрофічним для більшості моделей. Спад продуктивності розробки у відсотках між 5 та 1 часовими поясами для кожної архітектурної моделі приведено в таблиці 3.5.

Таблиця 3.5.

Спад продуктивності розробки для різних архітектурних моделей при розподілі команд розробки між 5 часовими поясами

Архітектурна модель	Продуктивність при 1 часовому поясі (функцій на розробника)	Продуктивність при 5 часових поясах (функцій на розробника)	Відсоток падіння продуктивності
Мікросервісна	17,9	8,8	51%
Модульна монолітна з сильною власністю програмного коду	5,0	2,9	43%
Модульна монолітна зі слабкою власністю програмного коду	10,3	5,4	47%
Звичайна монолітна	14,4	12,0	16%

Примітним винятком є звичайна монолітна архітектура, яка втрачає лише 16% продуктивності при розподілі команд розробки між 5 часовими поясами. З одного боку всі команди розробки є максимально незалежними одна від одної та можуть впроваджувати зміни до програмного коду будь-якої частини програмної системи, а з іншого – різні складові частини однієї нової функції можуть паралельно розроблятися різними командами розробки.

Ще одним фактором, що по-різному впливає на продуктивність розробки в контексті різних архітектурних моделей є кількість модулів програмного коду при фіксованому розмірі всієї кодової бази. Так як звичайна монолітна архітектура не має модульної структури, то відповідні симуляції виключені з поточного аналізу. Графіки залежності продуктивності розробки від кількості програмних модулів зображено на рис. 3.6.

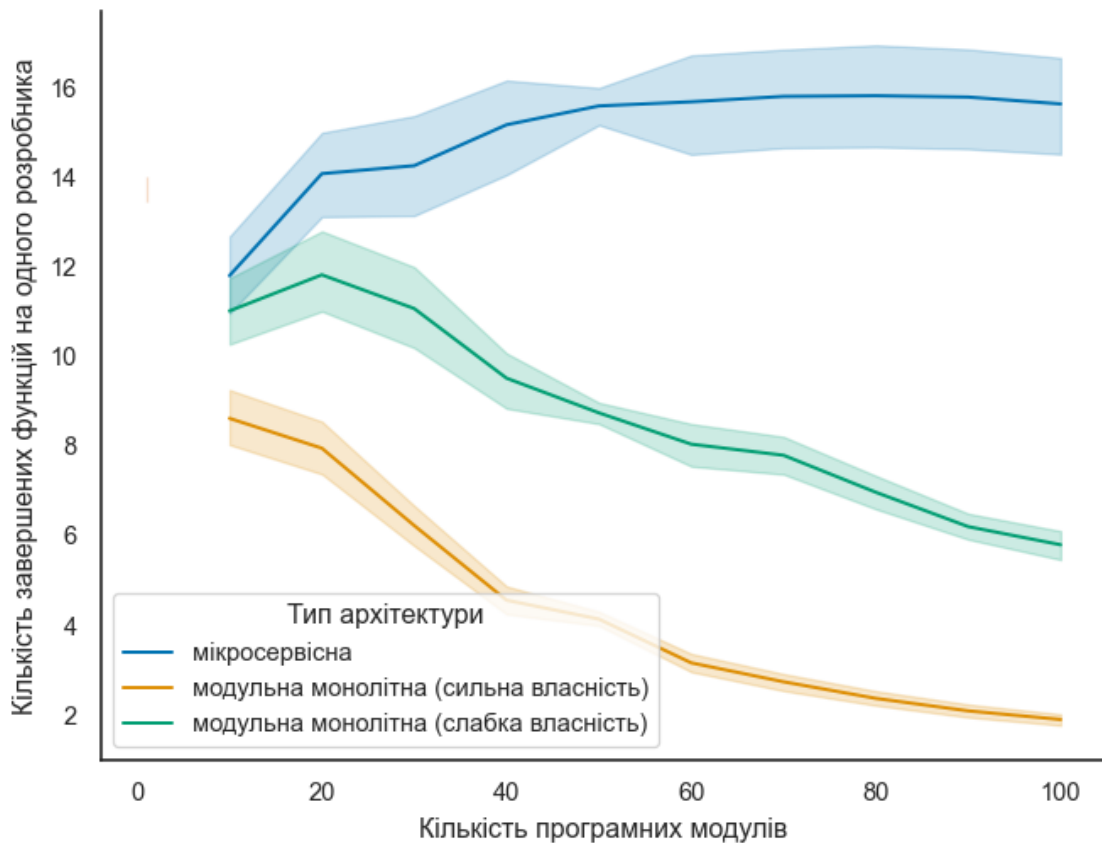


Рис. 3.6. Залежність продуктивності розробки від кількості програмних модулів

При використанні мікросервісної архітектури збільшення кількості модулів з 10 до 100 за фіксованого обсягу коду призводить до помірного зростання продуктивності на 15%. Це пояснюється тим, що більша кількість модулів означає менший розмір кожного сервісу, що спрощує їх розробку та тестування. Проте спостерігається цікавий ефект взаємодії з часовими поясами: при 1–2 часових поясах кореляція між кількістю модулів та продуктивністю розробки є позитивною, тоді як при 5 часових поясах вона стає негативною. Це

свідчить про те, що більша кількість модулів збільшує потребу у координації, яка стає особливо дорогою при значному часовому розподілі.

В контексті модульної монолітної архітектури з сильною власністю коду ефект є протилежним і надзвичайно вираженим. Збільшення модулів з 10 до 100 спричиняє колапс продуктивності з 8,0 до 1,7 функцій на одного розробника, що дорівнює зниженню продуктивності розробки на 78%. Кореляція є сильною та стабільною незалежно від кількості різних часових поясів. Механізм цього колапсу розкривається через аналіз витрат на тестування, які зростають практично лінійно з кількістю модулів: від 1,4 разів при 10 модулях до 10,4 разів при 100 модулях. Кожен додатковий модуль у монолітній архітектурі з сильною власністю програмного коду створює додаткові точки інтеграційного тестування, які зростають лінійно, а не логарифмічно, як можна було б очікувати при ефективній модуляризації.

При збільшенні кількості модулів в модульній монолітній архітектурі, але зі слабкою власністю коду, спостерігається аналогічна, але менш виражена тенденція зниження продуктивності процесу розробки з 9,3 до 5,5 функцій на одного розробника при збільшенні кількості модулів з 10 до 100, що є рівнозначним спаду продуктивності на 41%. Витрати на тестування також зростають лінійно з коефіцієнтом кореляції 0,95, але їхній абсолютний рівень нижчий, ніж у випадку модульної монолітної архітектури з сильною власністю коду.

Вплив загального обсягу програмного коду на результуючу продуктивність розробки проявляє різну динаміку залежно від архітектурної моделі. Графіки таких залежностей зображені на рис. 3.7.

Так як основною метою мікросервісної архітектури є саме розділення великої програмної системи на менші складові, то і результати симуляцій підтверджують, що вплив обсягу кодової бази на продуктивність розробки є статистично незначущим. Продуктивність практично ідентична при 50000 та 1000000 рядків коду.

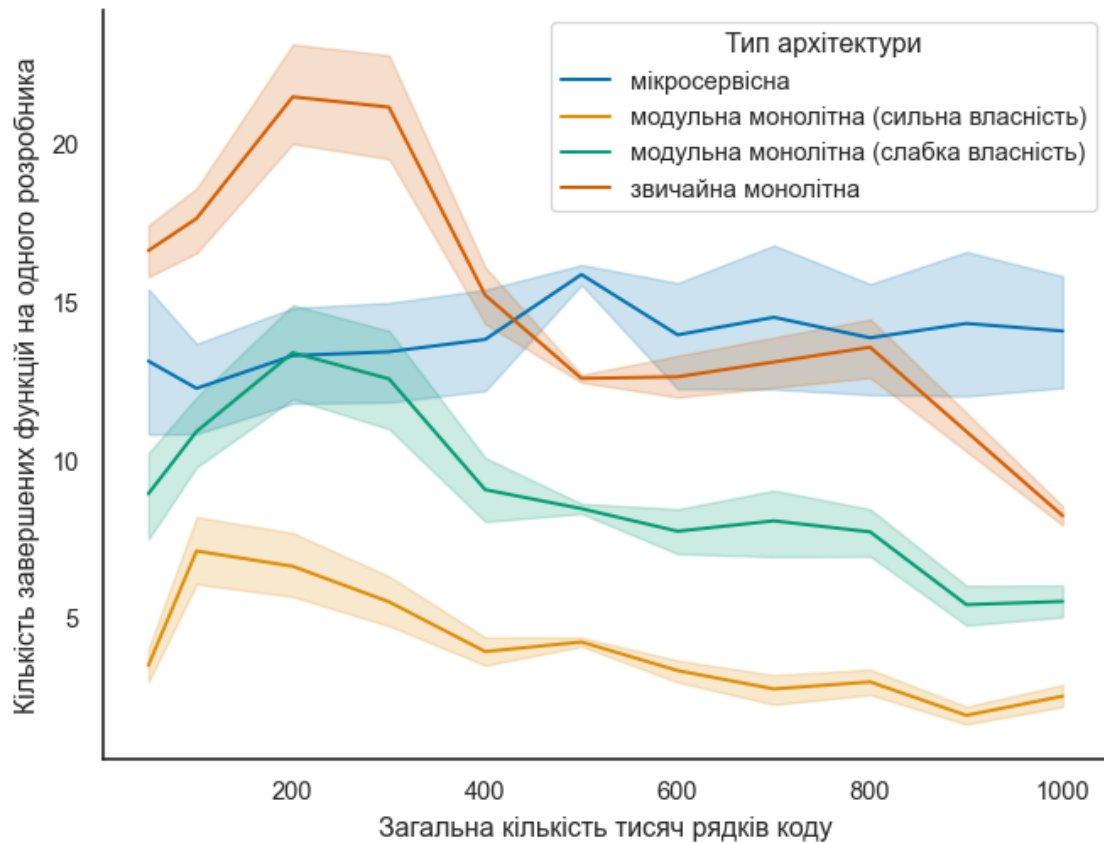


Рис. 3.7. Залежність продуктивності розробки від об'єму програмного коду

Зовсім інша картина постає у випадку використання звичайної немодульної монолітної архітектури, де спостерігається виражена нелінійна залежність з чітким оптимумом. Зі збільшенням кодової бази продуктивність розробки спочатку зростає до свого піку при 200000 рядків програмного коду, далі тримається майже на тому ж рівні до 300000 рядків коду, після чого знижується з таким самим темпом, як зростала на початку.

У випадках модульної організації монолітної кодової бази вплив загальної кількості рядків коду на продуктивність розробки є помірно негативним, без виражених нелінійних ефектів та не залежить від виду практики власності коду.

Симуляції процесів розробки програмного забезпечення з варіаціями розміру команд розробки та різним співвідношенням кількості тестувальників до кількості розробників підтвердили певні вже відомі закономірності, а також виявили деякі нові. Графіки залежності продуктивності розробки від кількості

розробників та тестувальників в командах зображені на рис. 3.7 та рис. 3.8 відповідно.

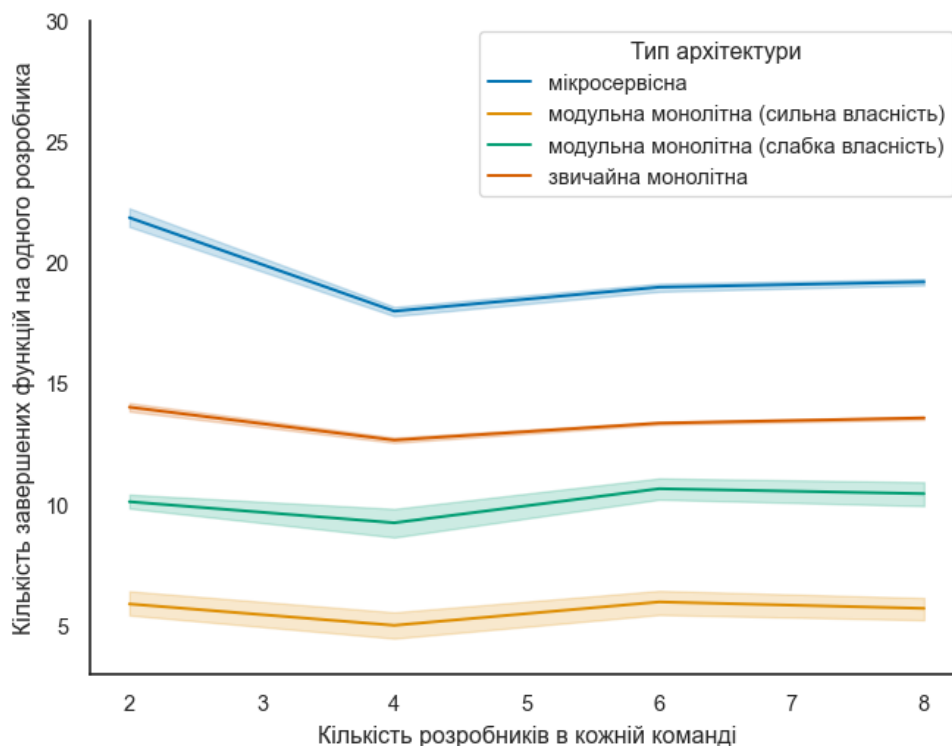


Рис. 3.7. Залежність продуктивності розробки від кількості розробників в кожній команді

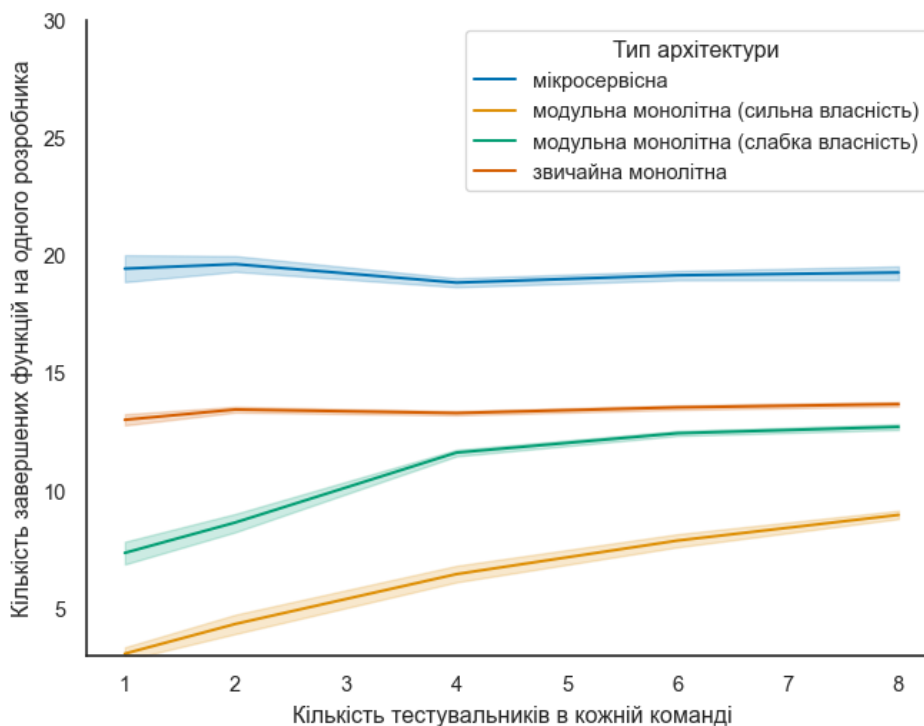


Рис. 3.8. Залежність продуктивності розробки від кількості тестувальників в кожній команді

В контексті модульної монолітної архітектури як із сильним типом власності програмного коду, так і зі слабким, кількість тестувальників в команді має вагому позитивну кореляцію з продуктивністю розробки, що зумовлене великими накладними витратами на тестування, тоді як кількість розробників має позитивний, але несуттєвий вплив. При 8 розробниках на команду, перехід від 1 до 8 тестувальників підвищує продуктивність розробки в контексті сильної власності коду в 4 рази, а при слабкій власності коду приблизно вдвічі.

У випадку ж мікросервісної архітектури ситуація парадоксально протилежна: кількість розробників має негативну кореляцію з продуктивністю розробки ($r = -0,33$), а тестувальників – слабо негативну ($r = -0,10$). Найвища продуктивність досягається у командах із 2 розробниками та 2 тестувальниками. Це пояснюється тим, що у мікросервісній архітектурі кожна команда працює над ізольованим сервісом, і менший розмір команди зменшує внутрішні координаційні витрати без втрати ефективності.

В сценаріях з немодульною монолітною архітектурою розмір команди має мінімальний вплив, що свідчить про відносну інваріантність цієї моделі до конфігурації команди.

Варіація симуляційних сценаріїв з різною кількістю команд розробки дозволяє оцінити ефективність горизонтального масштабування проєкту за рахунок додавання нових команд. Графіки зростання загальної продуктивності розробки при збільшенні кількості команд розробки фіксованого розміру (4 розробника, 2 тестувальника) для кожної архітектурної моделі зображені на рис. 3.9.

Мікросервісна архітектура демонструє найбільшу загальну пропускну здатність при масштабуванні, але продуктивність на одного розробника знижується приблизно на 37%. Маржинальний аналіз показує, що перші три команди додають по 73–82 додаткові завершені програмні функції кожна, четверта та п'ята – лише по 20–28, після чого спостерігається відновлення маржинальної віддачі (50–63 для шостої-восьмої команд), яке поступово знижується до 36 для десятої команди.

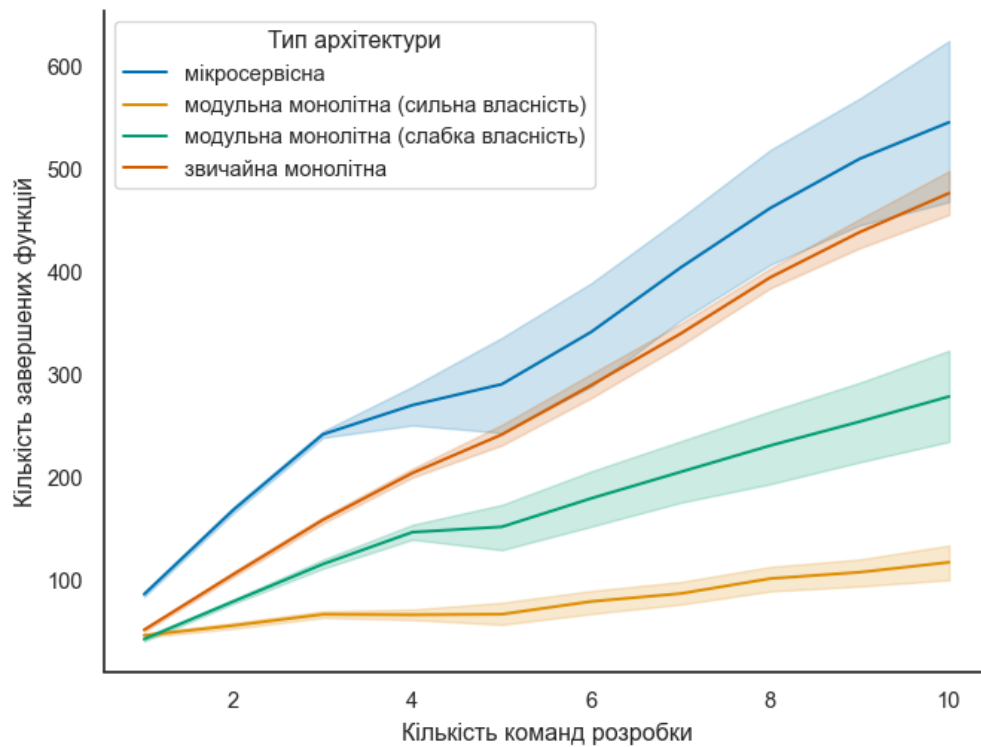


Рис. 3.9. Залежність продуктивності від кількості команд розробки фіксованого розміру (4 розробника, 2 тестувальника)

Ця нелінійність може свідчити про поріг, після якого кількість модулів стає достатньою для ефективного паралелізму, так як базова конфігурація містить 50 модулів на 5 команд, тобто по 10 модулів на кожну команду.

Модульна монолітна архітектура з сильною власністю коду показує найгірше масштабування. Продуктивність на розробника падає на -75%, а маржинальна віддача четвертої команди навіть від'ємна. Загальна продуктивність при 10 командах лише в 2,5 рази перевищує показник однієї команди при 10-кратному збільшенні ресурсів.

В контексті звичайної немодульної монолітної архітектури з колективною власністю програмного коду спостерігається найбільш стабільне масштабування, де продуктивність на одного розробника знижується лише на 8% при збільшенні кількості команд розробки з 1 до 10, а маржинальна віддача залишається відносно стабільною. Загальна кількість завершених програмних функцій зростає у 9,2 рази при збільшенні кількості команд розробки в 10 разів, що є найближчим до лінійного масштабування серед усіх моделей.

Модульна монолітна архітектура зі слабкою власністю програмного коду займає проміжну позицію, в якій при масштабуванні ресурсів розробки продуктивність на одного розробника знижується на 35%, а маржинальна віддача демонструє різкий спад при 5-й команді з подальшим відновленням до 24–28 для наступних команд.

Ще одним проведеним експериментом є варіація розподілу між кількістю та розміром команд розробки при фіксованій загальній кількості розробників. Дана варіація дозволяє дослідити та порівняти продуктивність великих команд проти більшої кількості малих команд розробки. Графіки залежності кількості завершених нових програмних функцій від кількості команд зображений на рис. 3.10.

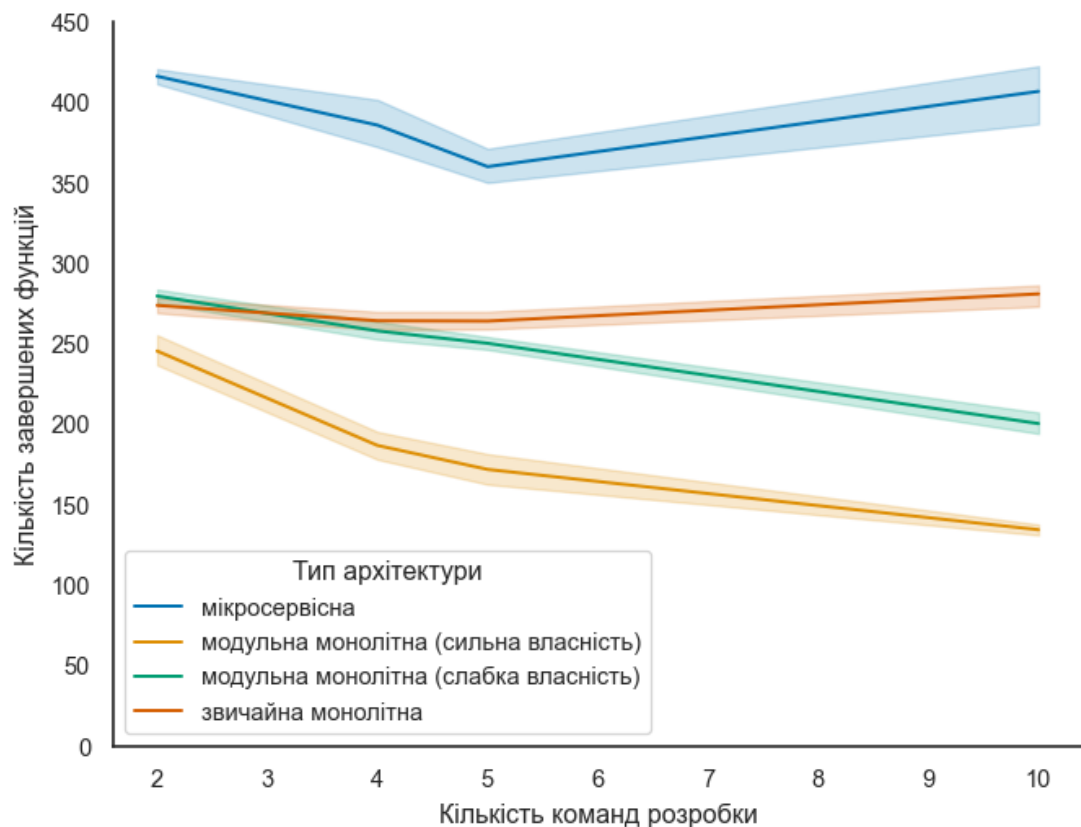


Рис. 3.10. Залежність продуктивності від кількості команд розробки при фіксованій загальній кількості розробників

При використанні модульної монолітної архітектури з сильною власністю програмного коду 2 команди по 10 розробників та 10 тестувальників показують

максимальну продуктивність розробки, що наближається до показників немодульної монолітної архітектури, а при 10 командах по 2 розробники відносна продуктивність падає майже вдвічі. В сценаріях зі слабкою власністю коду продуктивність розробки також знижується при збільшенні кількості команд через зв'язність модулів та взаємозалежність команд одна від одної. У випадку з мікросервісною архітектурою відмінність в продуктивності при різній кількості команд мінімальна з цікавим відхиленням, коли 10 малих команд досягають майже такого ж рівня продуктивності, як і 2 великі команди. Це підтверджує, що мікросервісна архітектура ефективно компенсує організаційну фрагментацію. Вплив розміру команд на продуктивність розробки в контексті звичайної немодульної монолітної архітектури є несуттєвим.

За замовчуванням, базові сценарії симуляції обмежують максимальну кількість активних задач в роботі в 3 задачі на кожного розробника. Навіть якщо задачу була переведена на етап рецензування або тестування, поки відповідні зміни програмного коду не будуть успішно розгорнуті, дані задачі вважаються активними для розробника, оскільки можуть вимагати додаткової роботи по виправленню дефектів тощо. Варіація симуляцій з різним обмеженням активних задач може виявити потенційні шляхи підвищення продуктивності розробки, проте симуляції не враховують когнітивне навантаження на розробників при роботі з занадто великою кількістю паралельних задач. Графік залежності продуктивності від обмеження максимальної кількості активних задач на одного розробника зображено на рис. 3.11.

З рис. 3.11 видно, що замале обмеження максимальної кількості активних задач призводить до зниження загальної продуктивності розробки для всіх архітектурних моделей, але після досягнення достатнього значення, подальший приріст продуктивності не спостерігається. Для досягнення максимальної продуктивності розробки в контексті мікросервісної архітектури достатньо обмеження в 4 активних задачі, для модульної монолітної з сильною власністю коду достатньо 2 паралельних задач, для модульної монолітної зі слабкою власністю програмного коду оптимальним є значення в 5 активних задач.

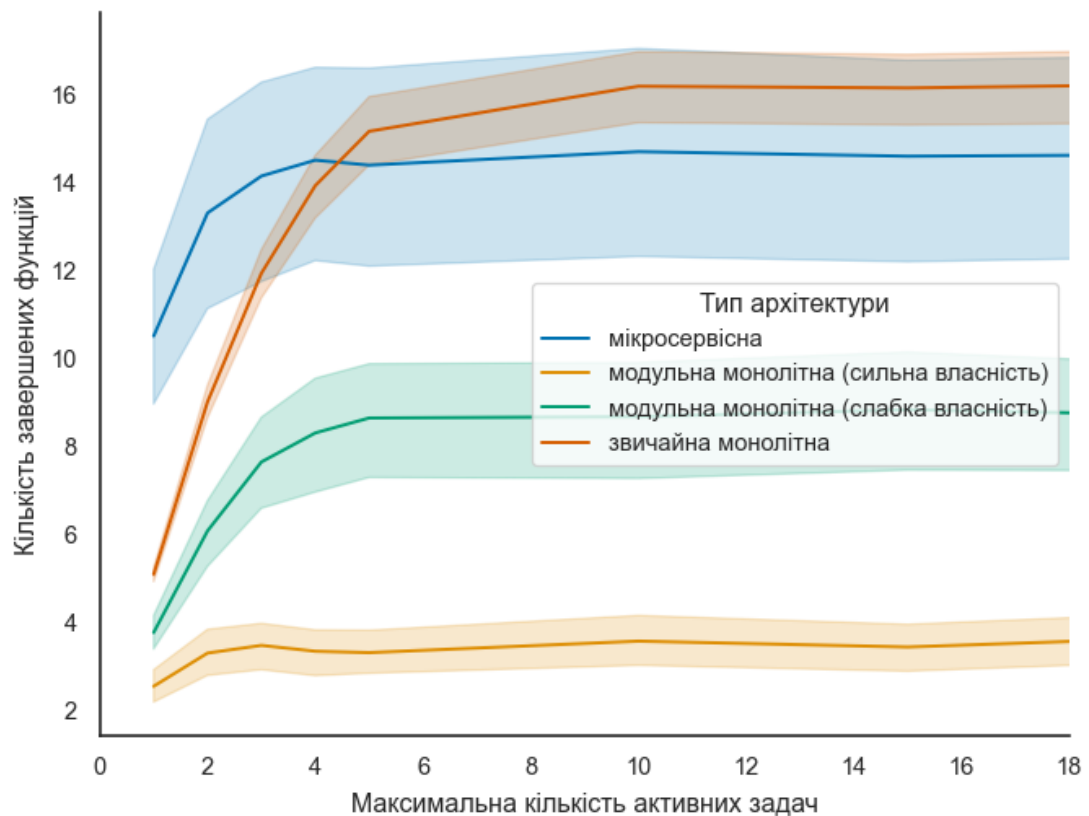


Рис. 3.11. Залежність продуктивності від обмеження максимальної кількості активних задач на одного розробника

Для звичайної ж немодульної монолітної архітектури суттєвий приріст продуктивності спостерігається вже при 5 паралельних задачах та досягає максимуму при 10, проте, так як когнітивне навантаження на розробників не враховується в симуляційній моделі, дані значення мають бути перевірені експериментальним чином в реальних умовах розробки програмного забезпечення.

Відповідно до виявлених закономірностей, можна вивести оптимальні рекомендації щодо різних вхідних параметрів для кожної окремої архітектурної моделі для максимізації продуктивності розробки програмного забезпечення. Оскільки різні вхідні параметри мають різний як якісний, так і кількісний вплив на ефективність процесу розробки ПЗ в контексті різних типів архітектури програмних систем, то і рекомендації наводяться окремо для кожної архітектурної моделі та мають розглядатись лише у відповідному контексті та не

виноситись за його межі. В таблиці 3.6 приведені рекомендовані значення або обмеження всіх вхідних параметрів, що були враховані в симуляційній моделі. Повний набір вихідних даних всіх сценаріїв та ітерацій проведених симуляцій доступний для завантаження на платформі Zenodo [101].

Таблиця 3.6.

Рекомендовані параметри для максимізації ефективності розробки для різних архітектурних моделей

Параметр	Мікросервіси	Немодульний моноліт	Модульний моноліт (сильна власність)	Модульний моноліт (слабка власність)
Часові пояси	1-2 (уникати більшого)	1-2 (більша к-сть не критична)	Мінімум	Мінімум
Розмір команди	2 розробники + 2 тестувальники	4 розробники + 2 тестувальники	Паритет розробників і тестувальників	Співвідношення тестувальників до розробників (1:1, мін. 1:2)
Кількість команд	1-3 (інд. прод.) або 6-10 (заг. пропускна здатність)	Не визначено	Мінімальна	Не визначено
Кодова база	Не є обмежуючим фактором	200-300 тис. рядків коду	Не визначено	~20-30 модулів по ~10000 рядків коду
Кількість модулів	Не менше 5 на команду	-	Не більше 20	20-30
Масштабованість	Найвища	Обмежена обсягом кодової бази	Найнижча	Низька

Для максимізації продуктивності на розробника в контексті мікросервісної архітектури рекомендується 1-2 часових пояси (уникати більшого географічного розподілу), команди з 2 розробників та 2 тестувальників, 1-3 команди для максимальної індивідуальної продуктивності або 6-10 для максимальної загальної пропускної здатності, не менше 5 модулів на команду та уникати жорсткого обмеження кількості активних задач на одного розробника. Обсяг кодової бази та загальна кількість модулів не є значущими обмежуючими

факторами, що робить мікросервісну архітектуру найбільш масштабованою моделлю.

В умовах немодульної монолітної архітектури найвища продуктивність розробки досягається при 200000-300000 рядків програмного коду, 1–2 часових поясах, проте більша їх кількість не є критичною, при структурі команд з 4 розробників та 2 тестувальників і відсутності жорсткого обмеження кількості активних задач на одного розробника. Дана архітектурна модель показує найвищу продуктивність розробки для глобально розподілених команд. Головним обмеженням є необхідність контролю обсягу кодової бази.

Модульна монолітна архітектура з сильним типом власності програмного коду є найменш продуктивною за всіх конфігурацій. Для мінімізації втрат рекомендується мати не більше 20 програмних модулів, паритетне співвідношення розробників та тестувальників, мінімально можливу кількість команд розробки та мінімально можливий розподіл команд поміж часових поясів, але основною рекомендацією є перехід до альтернативної архітектурної моделі або хоча б зміни виду практики власності програмного коду.

Модульна монолітна архітектура зі слабким типом власності програмного коду показує несуттєво вищу продуктивність, ніж у випадку з сильною власністю коду, але не демонструє високі абсолютні значення. Для максимізації продуктивності рекомендується максимально обмежити географічний розподіл команд розробки, оптимальна структура кодової бази має складатись з 20-30 програмних модулів приблизно по 10000 рядків коду кожен та високе співвідношення тестувальників до розробників (ідеально 1:1, мінімум 1:2). Для даного типу архітектури також рекомендується розглянути альтернативні моделі.

3.3. Регресійні моделі оцінки ефективності процесу розробки для різних типів архітектури програмних систем

Даний підрозділ присвячено побудові та порівняльному аналізу математичних моделей оцінки продуктивності розробки програмного забезпечення, що вимірюється кількістю завершених повноцінних функцій програмної системи на одного розробника. Моделювання здійснюється окремо для кожної архітектурної моделі в комбінації з різними видами практики власності програмного коду, що дозволяє врахувати специфіку кожного архітектурного підходу та виявити унікальні закономірності, властиві різним типам організації програмних систем.

Набір даних, отриманих шляхом симуляції процесів розробки програмного забезпечення в попередньому підрозділі, містить 8080 спостережень, розподілених між чотирма архітектурними моделями: мікросервісна (2270 спостережень), звичайна монолітна (1270 спостережень), модульна монолітна з сильним типом власності програмного коду (2270 спостережень) та модульна монолітна зі слабким типом власності коду (2270 спостережень). Кожне спостереження представляє результат імітаційного моделювання процесу розробки програмного забезпечення з різними конфігураціями параметрів.

Метрикою ефективності процесу розробки програмного забезпечення виступає кількість повністю завершених функцій програмної системи на одного розробника (P), а в якості незалежних змінних або предикторів використовуються сім вхідних параметрів, що характеризують технічну складність проєкту та організаційну структуру команд розробки:

- M – загальна кількість модулів у системі, що відображає рівень декомпозиції архітектури
- L – загальна кількість рядків програмного коду, що характеризує об'єм кодової бази проєкту
- T – кількість команд, задіяних у розробці
- D – кількість розробників у кожній команді
- Q – кількість тестувальників у кожній команді

- Z – кількість часових поясів, у яких працюють команди, що відображає географічний розподіл команд
- W – максимальна дозволена кількість незавершених задач на одного розробника

Для кожної архітектурної моделі будується п'ять типів регресійних моделей: множинна лінійна регресія, поліноміальна регресія другого ступеня, лог-лінійна модель, а також дві ансамблеві моделі: Random Forest та Gradient Boosting.

Множинна лінійна регресія це класична параметрична модель, яка встановлює лінійний зв'язок між вхідними параметрами та залежною змінною. Перевагою цієї моделі є простота інтерпретації: кожен коефіцієнт показує, на скільки одиниць змінюється оцінка продуктивності розробки при збільшенні відповідного параметра на одиницю за умови незмінності решти факторів. Рівняння лінійної моделі наведено у формулі (3.2):

$$P = \beta_0 + \sum_1^N \beta_i X_i + \varepsilon, \quad (3.2)$$

де β_0 – вільний член, β_i – коефіцієнти регресії, X_i – вхідні змінні, ε – випадкова похибка.

Поліноміальна регресія другого ступеня являє собою розширення лінійної моделі за рахунок додавання квадратичних членів та взаємодій між предикторами. Модель враховує не лише лінійні ефекти кожної змінної, а й їхні квадрати та попарні добутки. Це дозволяє описати нелінійні та синергетичні залежності. При 7 вхідних змінних кількість поліноміальних ознак складає 35 (7 лінійних, 7 квадратичних та 21 парної взаємодії). Загальний вигляд рівняння даної моделі зображений у формулі (3.3):

$$P = \beta_0 + \sum \beta_i X_i + \sum \beta_k X_k^2 + \sum \beta_{ij} X_i X_j + \varepsilon, \quad (3.3)$$

де β_0 – вільний член, β_i – коефіцієнти регресії, X_i – вхідні змінні, ε – випадкова похибка.

У лог-лінійній моделі логарифмуються як залежна змінна, так і всі значення вхідних параметрів. Вигляд рівняння даної моделі наведено у формулі (3.4):

$$\ln(P + 1) = \beta_0 + \sum_1^N \beta_i \ln(X_i + 1) + \varepsilon, \quad (3.4)$$

де β_0 – вільний член, β_i – коефіцієнти регресії, X_i – вхідні змінні, ε – випадкова похибка. Дана регресійна модель дозволяє моделювати мультиплікативні залежності та інтерпретувати коефіцієнти регресії як еластичності: коефіцієнт β_i показує, на скільки відсотків зміниться залежна змінна при зміні i -го предиктора на 1%. Модель добре описує мультиплікативні залежності та відносні ефекти, але зворотне перетворення з логарифмічного простору може спотворювати абсолютні похибки. Додавання одиниці перед логарифмуванням запобігає невизначеності при нульових значеннях.

Ансамблеві моделі представлені двома алгоритмами. Random Forest (випадковий ліс) будує 200 дерев рішень з максимальною глибиною 10 на випадкових підвбірках даних та ознак, усереднюючи їх результати. Це забезпечує стійкість до шуму та автоматично враховує нелінійні залежності та взаємодії між вхідними змінними. Gradient Boosting (градієнтне посилення) послідовно будує 200 дерев глибиною до 5, кожне з яких виправляє помилки попередніх, із кроком навчання 0.1. Обидва алгоритми надають оцінки важливості ознак, але не дають аналітичного рівняння регресії, що ускладнює їх інтерпретацію.

Перевірка усіх моделей здійснюється методом 10-кратного перехресного затвердження з перемішуванням та з фіксованим випадковим початковим значенням рівним 42, що забезпечує відтворюваність результатів. Метрики якості включають коефіцієнт детермінації R^2 (частка поясненої дисперсії), середнє абсолютне відхилення прогнозу від фактичного значення MAE (Mean Absolute Error) та корінь із середньоквадратичної похибки RMSE (Root Mean Square Error), що є індикатором чутливості до великих відхилень. Для всіх метрик наводяться середнє значення та стандартне відхилення серед 10 підвбірок перехресного затвердження (Cross-validation).

Тренування всіх моделей, оцінка коефіцієнтів детермінації та перехресне затвердження здійснюються із використанням бібліотеки «scikit-learn» для середовища програмування на мові Python [102].

Побудова математичних моделей оцінки ефективності розробки програмного забезпечення для **мікросервісної архітектури** базується на основі 2270 спостережень із середнім значенням продуктивності 15.62 функцій системи на одного розробника та стандартним відхиленням 5.68.

Методом множинної лінійної регресії було отримане рівняння, зображене у формулі (3.5):

$$P = 21.3847 + 0.026391 \cdot M + 0 \cdot L - 0.664931 \cdot T + 0.212397 \cdot D + 0.537838 \cdot Q - 2.481142 \cdot Z - 0.000343 \cdot W, \quad (3.5)$$

де P – кількість повністю завершених функцій на одного розробника, M – загальна кількість модулів у системі, L – кількість рядків програмного коду, T – кількість команд розробки, D – кількість розробників у кожній команді, Q – кількість тестувальників у кожній команді, Z – кількість різних часових поясів, W – максимальна кількість активних задач на одного розробника.

Кожен коефіцієнт регресії показує, на скільки одиниць зміниться очікувана продуктивність розробки при збільшенні відповідного параметра на одиницю, за умови фіксації решти параметрів. Коефіцієнт детермінації на навчальній вибірці R^2 становить 0.5529, що свідчить про те, що лінійна модель пояснює приблизно 55.3% дисперсії залежної змінної. Результати перехресного затвердження наведені в таблиці 3.7.

Таблиця 3.7.

Результати перехресного затвердження множинної лінійної регресії для мікросервісної архітектури

Метрика	Середнє значення	Стандартне відхилення
R^2	0.5462	0.0544
MAE	2.8113	0.1493
RMSE	3.8112	0.2110

Різниця між R^2 на навчальній вибірці та середнім R^2 за перехресним затвердженням є незначною, що свідчить про відсутність перенавчання моделі.

Методом поліноміальної регресії другого ступеня було отримано рівняння, наведене у формулі (3.6):

$$\begin{aligned}
 P = & -10.776622 + 2.861218 \cdot Z - 0.874211 \cdot Z^2 + 0.756078 \cdot D^2 - \\
 & 0.480143 \cdot D \cdot Q + 0.479851 \cdot T \cdot D + 0.353525 \cdot Q \cdot Z + 0.317787 \cdot D \cdot \\
 & Z - 0.237768 \cdot T \cdot Z - 0.152356 \cdot T \cdot Q + 0.142780 \cdot T^2 + 0.134784 \cdot \\
 & Q^2 - 0.019466 \cdot M \cdot Z - 0.015318 \cdot Z \cdot W + 0.014573 \cdot M \cdot D + \\
 & 0.010930 \cdot M \cdot W + 0.009343 \cdot M \cdot T + 0.007287 \cdot M \cdot Q + 0.003643 \cdot \\
 & M - 0.001438 \cdot M^2 + 1.2515 \times 10^{-4} \cdot L + 2.8560 \times 10^{-5} \cdot W^2 - \\
 & 2.1383 \times 10^{-5} \cdot L \cdot D - 8.1676 \times 10^{-6} \cdot L \cdot T + 3.8474 \times 10^{-6} \cdot L \cdot Q - \\
 & 1.0221 \times 10^{-6} \cdot L \cdot W + 2.6740 \times 10^{-7} \cdot L \cdot Z + 1.2312 \times 10^{-7} \cdot M \cdot L - \\
 & 7.5691 \times 10^{-9} \cdot T \cdot W - 6.1444 \times 10^{-9} \cdot D \cdot W - 2.9850 \times 10^{-9} \cdot Q \cdot \\
 & W - 1.2024 \times 10^{-9} \cdot T - 9.2623 \times 10^{-10} \cdot D - 8.0229 \times 10^{-10} \cdot W - \\
 & 2.7876 \times 10^{-10} \cdot Q,
 \end{aligned} \tag{3.6}$$

де P – кількість повністю завершених функцій на одного розробника, M – загальна кількість модулів у системі, L – кількість рядків програмного коду, T – кількість команд розробки, D – кількість розробників у кожній команді, Q – кількість тестувальників у кожній команді, Z – кількість різних часових поясів, W – максимальна кількість активних задач на одного розробника.

Коефіцієнт детермінації R^2 на навчальній вибірці становить 0.7577, що є суттєвим покращенням точності прогнозування порівняно з лінійною моделлю, де $R^2 = 0.5529$. Приріст пояснювальної здатності складає 20.5%, що свідчить про наявність значних нелінійних залежностей та ефектів взаємодії, які не можуть бути адекватно враховані лінійною моделлю. Результати перехресного затвердження даної регресійної моделі наведені в таблиці 3.8.

Таблиця 3.8.

Результати 10-кратного перехресного затвердження поліноміальної регресії для мікросервісної архітектури

Метрика	Середнє значення	Стандартне відхилення
R^2	0.7492	0.0344
MAE	2.0233	0.1157
RMSE	2.8316	0.1814

Поліноміальна модель демонструє помітне покращення прогнозних характеристик, так як середнє значення MAE знизилося з 2.8113 до 2.0233, а RMSE з 3.8112 до 2.8316. Різниця між R^2 на навчанні та середнім R^2 за перехресним затверджуванням складає 0.0085, що свідчить про відсутність суттєвого перенавчання.

За допомогою лог-лінійного регресійного аналізу даних для мікросервісної архітектури було отримано рівняння, представлене формулою (3.7):

$$\ln(P + 1) = 3.0471 + 0.105796 \cdot \ln(M + 1) + 0.033609 \cdot \ln(L + 1) - 0.338987 \cdot \ln(T + 1) + 0.003916 \cdot \ln(D + 1) + 0.241925 \cdot \ln(Q + 1) - 0.778518 \cdot \ln(Z + 1) + 0.004455 \cdot \ln(W + 1), \quad (3.7)$$

де P – кількість повністю завершених функцій на одного розробника, M – загальна кількість модулів у системі, L – кількість рядків програмного коду, T – кількість команд розробки, D – кількість розробників у кожній команді, Q – кількість тестувальників у кожній команді, Z – кількість різних часових поясів, W – максимальна кількість активних задач на одного розробника.

У логарифмічному просторі R^2 складає 0.5406, тоді як при зворотному перетворенні на оригінальну шкалу R^2 становить 0.3686. Результати перехресного затверджування лог-лінійної регресійної моделі наведені в таблиці 3.9.

Таблиця 3.9.

Результати 10-кратного перехресного затверджування лог-лінійної регресії для мікросервісної архітектури

Метрика	Середнє значення	Стандартне відхилення
R^2	0.3560	0.0779
MAE	3.5315	0.1681
RMSE	4.5397	0.2477

Лог-лінійна модель демонструє нижчу прогнозну здатність порівняно з лінійною регресією (0.3560 проти 0.5462). Зниження якості передбачення пов'язане з тим, що зворотне перетворення вносить систематичне зміщення, яке

особливо помітне при великому розкиді значень залежної змінної, крім того, мультиплікативна структура залежності, що лежить в основі лог-лінійної моделі, може не повністю відповідати природі досліджуваних даних.

В результаті тренування ансамблевих моделей Random Forest та Gradient Boosting були отримані наступні оцінки важливості входних параметрів на передбачення продуктивності розробки в контексті мікросервісної архітектури, що наведені в таблиці 3.10.

Таблиця 3.10.

Оцінки важливості входних параметрів за ансамблевими моделями для мікросервісної архітектури

Параметр	Random Forest	Gradient Boosting
Кількість модулів	4.62%	4.87%
Кількість рядків коду	0.27%	0.67%
Кількість команд розробки	11.78%	11.26%
Кількість розробників в команді	19.89%	19.25%
Кількість тестувальників в команді	4.84%	5.18%
Кількість різних часових поясів	57.75%	57.77%
Максимальна кількість активних задач	0.85%	1.00%

Random Forest демонструє коефіцієнт детермінації R^2 на навчальній вибірці 0.9180, а Gradient Boosting – 0.9227. Результати перехресного затвердження ансамблевих моделей наведені в таблиці 3.11. Обидві ансамблеві моделі демонструють суттєво вищу прогнозну здатність порівняно з параметричними моделями. Gradient Boosting показує найкращі результати з $R^2 = 0.9113$ та MAE = 1.0415. У випадку обох методів, різниця між коефіцієнтами детермінації на навчальній вибірці та отриманими в результаті перехресного затвердження свідчить про контрольоване перенавчання, яке не критично впливає на узагальнювальну здатність даних моделей.

Таблиця 3.11.

**Результати 10-кратного перехресного затвердження ансамблевих
моделей для мікросервісної архітектури**

	Random Forest		Gradient Boosting	
Метрика	Середнє значення	Стандартне відхилення	Середнє значення	Стандартне відхилення
R^2	0.9059	0.0321	0.9113	0.0292
MAE	1.0650	0.1175	1.0415	0.1248
RMSE	1.7142	0.2976	1.6649	0.2801

Монолітна немодульна архітектура характеризується єдиною кодовою базою, де будь-який розробник може вносити зміни у будь-яку частину системи. Оскільки в даному типі архітектури відсутня модульна організація програмного коду, то вхідний параметр кількості модулів був виключений зі всіх регресійних моделей. Аналіз симуляційних даних для даного типу архітектури проведено на основі 1270 спостережень із середнім значенням продуктивності 13.66 функцій системи на одного розробника та стандартним відхиленням 3.29.

Отримане рівняння множинної лінійної регресії наведене у формулі (3.8):

$$P = 21.5950 + 0.000012 \cdot L - 0.159599 \cdot T + 0.021585 \cdot D + 0.008386 \cdot Q - 0.637508 \cdot Z - 0.045359 \cdot W, \quad (3.8)$$

де P – кількість повністю завершених функцій на одного розробника, L – кількість рядків програмного коду, T – кількість команд розробки, D – кількість розробників у кожній команді, Q – кількість тестувальників у кожній команді, Z – кількість різних часових поясів, W – максимальна кількість активних задач на одного розробника.

Коефіцієнт детермінації R^2 на навчальній вибірці становить 0.4111, що свідчить про те, що лінійна модель пояснює 41.1% дисперсії залежної змінної, що є ще нижчим показником, ніж при застосуванні даної моделі в контексті мікросервісної архітектури. Результати перехресного затвердження наведені в таблиці 3.12.

Таблиця 3.12.

Результати 10-кратного перехресного затверджування множинної лінійної регресії для немодульної монолітної архітектури

Метрика	Середнє значення	Стандартне відхилення
R^2	0.4038	0.0539
MAE	1.7418	0.1484
RMSE	2.5223	0.2361

Різниця між значеннями коефіцієнтів детермінації на навчальній вибірці та середнім за перехресним затверджуванням є незначною, що свідчить про відсутність факту перенавчання моделі. Проте, враховуючи їх досить низьке абсолютне значення, доцільно дослідити альтернативні методи регресії.

Оскільки в звичайній монолітній архітектурі з колективною власністю програмного коду модульна організація коду не застосовується, параметр, що відповідає за загальну кількість модулів, виключено з поліноміальної моделі, тому дана модель побудована лише на 6 вхідних змінних замість. Загальна кількість ознак після поліноміального перетворення становить 27, що складається з 6 лінійних членів, 6 квадратичних членів та 15 парних взаємодій. Методом поліноміальної регресії другого ступеня було отримано рівняння, представлене у формулі (3.9):

$$\begin{aligned}
 P = & 21.628509 - 0.216900 \cdot Z^2 - 0.203329 \cdot Z + 0.105458 \cdot D^2 + \\
 & 0.088938 \cdot Q \cdot Z - 0.076347 \cdot T \cdot Z + 0.069065 \cdot D \cdot Z + 0.037969 \cdot T \cdot \\
 & D - 0.034348 \cdot Q^2 - 0.032334 \cdot T \cdot Q - 0.019394 \cdot T^2 - 0.009256 \cdot Z \cdot \\
 & W + 0.007887 \cdot D \cdot Q - 0.001791 \cdot W^2 - 1.5488 \times 10^{-5} \cdot L - \\
 & 3.0727 \times 10^{-6} \cdot L \cdot D + 1.7289 \times 10^{-6} \cdot L \cdot Z + 4.9459 \times 10^{-7} \cdot L \cdot Q + \\
 & 4.7564 \times 10^{-7} \cdot L \cdot W + 3.5052 \times 10^{-7} \cdot L \cdot T + 2.2558 \times 10^{-9} \cdot T + \\
 & 9.2371 \times 10^{-10} \cdot T \cdot W + 7.1885 \times 10^{-10} \cdot D \cdot W + 3.7161 \times 10^{-10} \cdot Q \cdot \\
 & W + 1.1319 \times 10^{-10} \cdot D,
 \end{aligned} \tag{3.9}$$

де P – кількість повністю завершених функцій на одного розробника, L – кількість рядків програмного коду, T – кількість команд розробки, D – кількість розробників у кожній команді, Q – кількість тестувальників у кожній команді, Z

– кількість різних часових поясів, W – максимальна кількість активних задач на одного розробника.

Коефіцієнт детермінації на навчальній вибірці становить 0.5220, що є помітним покращенням порівняно з лінійною моделлю (0.4111) та відповідає приросту пояснювальної здатності на 11.1%. Як і у випадку з мікросервісною архітектурою, це вказує на присутність значних нелінійних залежностей між вхідними змінними та ефектів їх взаємодії між собою, але дана модель все ще пояснює менший відсоток дисперсії, ніж будь-яка модель регресії, застосована для мікросервісної архітектури. Результати перехресного затвердження поліноміальної регресії другого ступеня наведені в таблиці 3.13.

Таблиця 3.13.

Результати 10-кратного перехресного затвердження поліноміальної регресії для немодульної монолітної архітектури

Метрика	Середнє значення	Стандартне відхилення
R^2	0.5099	0.0456
MAE	1.5002	0.1246
RMSE	2.2854	0.1984

Поліноміальна модель демонструє помітне покращення прогнозних характеристик порівняно з лінійною моделлю, оскільки MAE знизилося з 1.7418 до 1.5002, а RMSE з 2.5223 до 2.2854. Різниця між значеннями R^2 на навчанні та перехресного затвердження складає 0.0121, що свідчить про відсутність суттєвого перенавчання. Загалом, поліноміальна регресія підтверджує наявність суттєвих нелінійних залежностей та ефектів взаємодії між параметрами, але все ще не пояснює вагому частину дисперсії.

Лог-лінійний регресійний аналіз даних для монолітної архітектури вивів рівняння, наведене у формулі (3.10):

$$\ln(P + 1) = 5.4193 - 0.212041 \cdot \ln(L + 1) - 0.044656 \cdot \ln(T + 1) - 0.018523 \cdot \ln(D + 1) + 0.067309 \cdot \ln(Q + 1) - 0.152138 \cdot \ln(Z + 1) + 0.130743 \cdot \ln(W + 1), \quad (3.10)$$

де P – кількість повністю завершених функцій на одного розробника, L – кількість рядків програмного коду, T – кількість команд розробки, D – кількість розробників у кожній команді, Q – кількість тестувальників у кожній команді, Z – кількість різних часових поясів, W – максимальна кількість активних задач на одного розробника.

Результати перехресного затвердження наведені в таблиці 3.14.

Таблиця 3.14.

**Результати перехресного затвердження лог-лінійної моделі для
немодульної монолітної архітектури**

Метрика	Середнє значення	Стандартне відхилення
R^2	0.3781	0.0462
MAE	1.7531	0.1480
RMSE	2.5770	0.2342

Лог-лінійна модель демонструє найнижчу прогнозну здатність порівняно з обома попередніми моделями, навіть до зворотного перетворення. По результатам навчання, у логарифмічному просторі коефіцієнт детермінації R^2 складає 0.4037, тоді як після зворотного перетворення на оригінальну шкалу – 0.3861. Після ж перехресного затвердження коефіцієнт детермінації виявився ще нижчим.

Ансамблеві моделі Random Forest та Gradient Boosting виявили наступні оцінки важливості входних ознак, що наведені в таблиці 3.15.

Таблиця 3.15.

Оцінки важливості входних ознак для немодульної монолітної архітектури

Параметр	Random Forest	Gradient Boosting
Кількість рядків коду	43.55%	43.11%
Кількість команд розробки	1.26%	1.27%
Кількість розробників в команді	2.61%	1.86%
Кількість тестувальників в команді	0.47%	0.41%
Кількість різних часових поясів	20.31%	22.27%
Максимальна кількість активних задач	31.79%	31.09%

Random Forest демонструє коефіцієнт детермінації R^2 на навчальній вибірці 0.9617, а Gradient Boosting – 0.9657. Результати перехресного затвердження ансамблевих моделей наведені в таблиці 3.16.

Таблиця 3.16.

Результати 10-кратного перехресного затвердження ансамблевих моделей для немодульної монолітної архітектури

	Random Forest		Gradient Boosting	
Метрика	Середнє значення	Стандартне відхилення	Середнє значення	Стандартне відхилення
R^2	0.9549	0.0110	0.9571	0.0112
MAE	0.4995	0.0400	0.4792	0.0374
RMSE	0.6853	0.0586	0.6678	0.0616

Обидві ансамблеві моделі демонструють значно вищу прогнозну здатність порівняно з параметричними моделями. Gradient Boosting показує найкращі результати, але якість передбачення методом Random Forest майже не відстає. Структура важливості ознак є узгодженою між обома моделями, що підтверджує стійкість виявлених закономірностей. Ключова різниця полягає у принципі побудови: Random Forest використовує паралельну побудову незалежних дерев із усередненням, тоді як Gradient Boosting застосовує послідовну корекцію помилок, що дозволяє краще апроксимувати складні залежності за рахунок більшого ризику перенавчання, проте різниця між значеннями коефіцієнтів детермінації на навчальній вибірці та в результаті перехресного затвердження є несуттєвою, що свідчить про контрольоване перенавчання, яке не критично впливає на якість оцінки ефективності розробки програмного забезпечення.

Монолітна модульна архітектура із сильним типом власності програмного коду передбачає єдину кодову базу з чітким розподілом відповідальності між командами за конкретні програмні модулі. Для побудови регресійних моделей оцінки продуктивності розробки для даного типу архітектури проведено аналіз на основі 2270 спостережень із середнім значенням

продуктивності у 4.47 функцій системи на одного розробника та стандартним відхиленням рівним 2.72.

Методом множинної лінійної регресії було отримане рівняння, зображене у формулі (3.11):

$$P = 13.0210 - 0.072810 \cdot M - 0.000001 \cdot L - 0.469510 \cdot T - 0.815223 \cdot D + 1.425188 \cdot Q - 0.637539 \cdot Z - 0.011220 \cdot W, \quad (3.11)$$

де P – кількість повністю завершених функцій на одного розробника, M – загальна кількість модулів у системі, L – кількість рядків програмного коду, T – кількість команд розробки, D – кількість розробників у кожній команді, Q – кількість тестувальників у кожній команді, Z – кількість різних часових поясів, W – максимальна кількість активних задач на одного розробника.

Коефіцієнт детермінації R^2 пояснює 69% дисперсії вихідного значення продуктивності розробки з отриманим значенням на навчальній вибірці даних 0.6901, що є найкращим значенням лінійної регресії серед усіх попередніх типів архітектури. Результати перехресного затвердження наведені в таблиці 3.17.

Таблиця 3.17.

Результати перехресного затвердження множинної лінійної регресії для модульної монолітної архітектури з сильною власністю коду

Метрика	Середнє значення	Стандартне відхилення
R^2	0.6851	0.0185
MAE	1.1263	0.0614
RMSE	1.5184	0.1281

Відмінність значень R^2 отриманих на навчальній вибірці та шляхом перехресного затвердження є незначною, що вказує на відсутність істотного перенавчання моделі.

Для дослідження нелінійних залежностей та синергетичних ефектів між вхідними параметрами була також створена поліноміальна регресійна модель другого ступеня. Так само, як і для мікросервісної архітектури, загальна кількість ознак після поліноміального перетворення становить 35, що складається з 7

лінійних членів, 7 квадратичних членів та 21 парних взаємодій. В результаті аналізу було отримано рівняння, наведене у формулі (3.12):

$$\begin{aligned}
 P = & 9.141936 - 0.280458 \cdot D \cdot Q - 0.278020 \cdot Z + 0.226032 \cdot D^2 - \\
 & 0.205837 \cdot Z^2 + 0.144769 \cdot T^2 + 0.100330 \cdot Q \cdot Z - 0.076156 \cdot T \cdot Q + \\
 & 0.061322 \cdot D \cdot Z + 0.058198 \cdot T \cdot D - 0.036247 \cdot M \cdot D - 0.029449 \cdot T \cdot \\
 & Z - 0.027185 \cdot M \cdot W - 0.018123 \cdot M \cdot Q + 0.009402 \cdot M \cdot T - \\
 & 0.009248 \cdot Q^2 - 0.009062 \cdot M + 0.008059 \cdot M \cdot Z - 0.003505 \cdot Z \cdot W + \\
 & 7.4337 \times 10^{-4} \cdot M^2 + 8.7592 \times 10^{-5} \cdot W^2 + 8.8713 \times 10^{-6} \cdot L \cdot Q + \quad (3.12) \\
 & 5.8056 \times 10^{-6} \cdot L - 5.6910 \times 10^{-6} \cdot L \cdot T + 2.7183 \times 10^{-6} \cdot L \cdot W - \\
 & 2.2200 \times 10^{-6} \cdot L \cdot D + 4.2605 \times 10^{-7} \cdot L \cdot Z + 8.9672 \times 10^{-8} \cdot M \cdot L - \\
 & 3.5492 \times 10^{-10} \cdot T \cdot W - 2.6994 \times 10^{-10} \cdot D \cdot W - 7.5787 \times 10^{-11} \cdot \\
 & T - 7.5082 \times 10^{-11} \cdot Q \cdot W - 6.2173 \times 10^{-11} \cdot D - 2.6338 \times 10^{-11} \cdot \\
 & Q - 2.4620 \times 10^{-11} \cdot W - 1.4712 \times 10^{-12} \cdot L^2,
 \end{aligned}$$

де P – кількість повністю завершених функцій на одного розробника, M – загальна кількість модулів у системі, L – кількість рядків програмного коду, T – кількість команд розробки, D – кількість розробників у кожній команді, Q – кількість тестувальників у кожній команді, Z – кількість різних часових поясів, W – максимальна кількість активних задач на одного розробника.

Коефіцієнт детермінації R^2 на навчальній вибірці становить 0.8518, що є помітним покращенням порівняно з множинною лінійною моделлю ($R^2 = 0.6901$), при чому значення MAE знизилося з 1.1263 до 0.7786, а RMSE з 1.5184 до 1.0640. Приріст пояснювальної здатності у 16.2% вказує на наявність істотних нелінійних залежностей та синергетичних ефектів між вхідними параметрами, які не враховуються лінійною моделлю. Різниця між коефіцієнтами детермінації, отриманими в процесі навчання та при перехресному затвердженні складає лише 0.0077, що вказує на відсутність значного перенавчання регресійної моделі. Результати перехресного затвердження поліноміальної моделі наведені в таблиці 3.18.

Таблиця 3.18.

Результати 10-кратного перехресного затвердження поліноміальної регресії для модульної монолітної архітектури з сильною власністю коду

Метрика	Середнє значення	Стандартне відхилення
R ²	0.8441	0.0203
MAE	0.7786	0.0424
RMSE	1.0640	0.0805

Лог-лінійний регресійний аналіз вибірки даних для даної комбінації архітектури та виду практики власності програмного коду дає наступне рівняння, представлене у формулі (3.13):

$$\ln(P + 1) = 6.0488 - 0.565736 \cdot \ln(M + 1) - 0.040986 \cdot \ln(L + 1) - 0.615315 \cdot \ln(T + 1) - 0.856353 \cdot \ln(D + 1) + 1.186367 \cdot \ln(Q + 1) - 0.511616 \cdot \ln(Z + 1) - 0.006711 \cdot \ln(W + 1), \quad (3.13)$$

де P – кількість повністю завершених функцій на одного розробника, M – загальна кількість модулів у системі, L – кількість рядків програмного коду, T – кількість команд розробки, D – кількість розробників у кожній команді, Q – кількість тестувальників у кожній команді, Z – кількість різних часових поясів, W – максимальна кількість активних задач на одного розробника.

У логарифмічному просторі R² на навчальній вибірці складає 0.7518, тоді як при зворотному перетворенні на оригінальну шкалу стає рівне 0.6823, що демонструє суттєво нижчу прогностичну здатність в порівнянні з поліноміальною моделлю та навіть нижчу порівняно з лінійною регресією. Результати перехресного затвердження наведені в таблиці 3.19.

Таблиця 3.19.

Результати перехресного затвердження лог-лінійної регресії для модульної монолітної архітектури з сильною власністю коду

Метрика	Середнє значення	Стандартне відхилення
R ²	0.8441	0.0203
MAE	0.7786	0.0424
RMSE	1.0640	0.0805

В результаті тренування моделей Random Forest та Gradient Boosting були виявлені наступні оцінки важливості вхідних ознак, що наведені в таблиці 3.20.

Таблиця 3.20.

Оцінки важливості вхідних ознак для модульної монолітної архітектури з сильною власністю коду

Параметр	Random Forest	Gradient Boosting
Кількість модулів	29.51%	29.31%
Кількість рядків коду	4.24%	4.58%
Кількість команд розробки	20.61%	19.67%
Кількість розробників в команді	6.81%	6.30%
Кількість тестувальників в команді	24.16%	24.01%
Кількість різних часових поясів	14.62%	15.87%
Максимальна кількість активних задач	0.05%	0.26%

Random Forest демонструє коефіцієнт детермінації R^2 на навчальній вибірці 0.9288, а Gradient Boosting – 0.9350. Результати перехресного затвердження ансамблевих моделей наведені в таблиці 3.21.

Таблиця 3.21.

Результати перехресного затвердження ансамблевих моделей для модульної монолітної архітектури з сильною власністю коду

Метрика	Random Forest		Gradient Boosting	
	Середнє значення	Стандартне відхилення	Середнє значення	Стандартне відхилення
R^2	0.9175	0.0157	0.9230	0.0147
MAE	0.5271	0.0401	0.5043	0.0398
RMSE	0.7719	0.0817	0.7460	0.0804

Обидва ансамблеві методи суттєво перевершують параметричні моделі, а Gradient Boosting має незначну перевагу над Random Forest, як в більшій

спроможності пояснення дисперсії залежної змінної, так і в меншій похибці передбачення значень залежної змінної. Виявлена важливість вхідних ознак узгоджена між обома ансамблевими моделями, що доводить актуальність виявлених закономірностей. Розрив між значеннями коефіцієнтів детермінації R^2 , отриманими в результаті тренування та в результаті перехресного затвердження, є характерним для деревоподібних ансамблевих моделей і свідчить про контрольоване перенавчання.

Модульна монолітна архітектура зі слабкою власністю програмного коду передбачає єдину модульну кодову базу з чітким розподілом відповідальності, де команди розробки мають проводити рецензування змін до програмного коду модулів, які знаходяться під їх власністю, але кожна команда вільна вносити необхідні зміни в будь-який програмний модуль, незалежно від його власності. Процес побудови регресійних моделей передбачення продуктивності розробки в контексті даної архітектури базується на основі 2270 спостережень із середнім значенням продуктивності розробки у 8.85 функцій системи на одного розробника та стандартним відхиленням 3.50.

Отримане рівняння множинної лінійної регресійної моделі зображене у формулі (3.14):

$$P = 18.3029 - 0.056186 \cdot M - 0.000004 \cdot L - 0.355136 \cdot T - 0.544970 \cdot D + 1.212406 \cdot Q - 1.307230 \cdot Z + 0.006701 \cdot W, \quad (3.14)$$

де P – кількість повністю завершених функцій на одного розробника, M – загальна кількість модулів у системі, L – кількість рядків програмного коду, T – кількість команд розробки, D – кількість розробників у кожній команді, Q – кількість тестувальників у кожній команді, Z – кількість різних часових поясів, W – максимальна кількість активних задач на одного розробника.

Отримана модель пояснює 64.3% дисперсії продуктивності розробки, так як отриманий в процесі навчання коефіцієнт детермінації R^2 дорівнює 0.6428, що несуттєво відрізняється від середнього значення, отриманого в результаті перехресного затвердження. Кожен коефіцієнт виведеного рівняння показує, на скільки одиниць зміниться очікувана продуктивність розробки на одного

розробника при збільшенні відповідного параметра на одиницю, за умови, що значення решти параметрів залишаються незмінними. Результати перехресного затвердження наведені в таблиці 3.22.

Таблиця 3.22.

Результати перехресного затвердження множинної лінійної регресії для модульної монолітної архітектури зі слабкою власністю коду

Метрика	Середнє значення	Стандартне відхилення
R ²	0.6382	0.0282
MAE	1.5186	0.0886
RMSE	2.0982	0.1453

Повне отримане рівняння поліноміальної регресії другого ступеня для модульної монолітної архітектури зі слабким типом власності коду наведено у формулі (3.15):

$$\begin{aligned}
 P = & 17.351485 - 2.025140 \cdot Z - 0.318788 \cdot Q^2 - 0.297644 \cdot T \cdot Q + \\
 & 0.268584 \cdot T \cdot D + 0.210971 \cdot Q \cdot Z - 0.168734 \cdot T \cdot Z + 0.163603 \cdot D \cdot \\
 & Q + 0.153311 \cdot D \cdot Z - 0.145048 \cdot Z^2 + 0.128244 \cdot D^2 - 0.018757 \cdot M \cdot \\
 & D + 0.016646 \cdot T^2 - 0.014067 \cdot M \cdot W + 0.010912 \cdot M \cdot Z - 0.009378 \cdot \\
 & M \cdot Q + 0.006859 \cdot M \cdot T - 0.004689 \cdot M - 0.004170 \cdot Z \cdot W - \\
 & 4.3158 \times 10^{-4} \cdot M^2 - 3.4727 \times 10^{-4} \cdot W^2 + 8.7432 \times 10^{-6} \cdot L \cdot Q - \quad (3.15) \\
 & 6.4762 \times 10^{-6} \cdot L \cdot D - 2.1451 \times 10^{-6} \cdot L \cdot T - 1.8694 \times 10^{-6} \cdot L + \\
 & 1.5316 \times 10^{-6} \cdot L \cdot Z + 1.5102 \times 10^{-6} \cdot L \cdot W + 1.3819 \times 10^{-7} \cdot M \cdot L - \\
 & 1.2662 \times 10^{-10} \cdot Q + 1.1512 \times 10^{-10} \cdot T \cdot W + 1.0366 \times 10^{-10} \cdot Q \cdot \\
 & W - 8.1173 \times 10^{-11} \cdot D + 6.3538 \times 10^{-11} \cdot D \cdot W + 4.8975 \times 10^{-11} \cdot \\
 & W - 3.1425 \times 10^{-11} \cdot T + 4.5642 \times 10^{-13} \cdot L^2,
 \end{aligned}$$

де P – кількість повністю завершених функцій на одного розробника, M – загальна кількість модулів у системі, L – кількість рядків програмного коду, T – кількість команд розробки, D – кількість розробників у кожній команді, Q – кількість тестувальників у кожній команді, Z – кількість різних часових поясів, W – максимальна кількість активних задач на одного розробника.

В процесі навчання отриманий коефіцієнт детермінації R^2 дорівнює 0.7593, що є кращим показником відносно лінійної моделі (0.6428) та виступає індикатором присутності вагомих нелінійних залежностей між вхідними параметрами та ефектів їх взаємодії, але все ще не пояснює майже 25% дисперсії залежної змінної. Різниця між R^2 , отриманим в процесі тренування моделі, та середнім значенням з перехресного затвердження складає 0.0076, що підтверджує відсутність факту перенавчання моделі. Результати перехресного затвердження поліноміальної моделі наведені в таблиці 3.23.

Таблиця 3.23.

Результати 10-кратного перехресного затвердження поліноміальної регресії для модульної монолітної архітектури зі слабкою власністю коду

Метрика	Середнє значення	Стандартне відхилення
R^2	0.8441	0.0203
MAE	0.7786	0.0424
RMSE	1.0640	0.0805

Лог-лінійна регресивна модель оцінки ефективності процесу розробки програмного забезпечення з відповідними отриманими коефіцієнтами представлена рівнянням у формулі (3.16):

$$\begin{aligned} \ln(P + 1) = & 5.6150 - 0.215705 \cdot \ln(M + 1) - 0.125201 \cdot \ln(L + 1) - \\ & 0.283049 \cdot \ln(T + 1) - 0.389094 \cdot \ln(D + 1) + 0.681650 \cdot \ln(Q + 1) - \\ & 0.597260 \cdot \ln(Z + 1) + 0.063600 \cdot \ln(W + 1), \end{aligned} \quad (3.16)$$

де P – кількість повністю завершених функцій на одного розробника, M – загальна кількість модулів у системі, L – кількість рядків програмного коду, T – кількість команд розробки, D – кількість розробників у кожній команді, Q – кількість тестувальників у кожній команді, Z – кількість різних часових поясів, W – максимальна кількість активних задач на одного розробника.

У логарифмічному просторі коефіцієнт детермінації R^2 на навчальній вибірці складає 0.6370, а після зворотного перетворення на оригінальну шкалу падає до 0.5397, що демонструє суттєво нижчу прогнозну здатність в порівнянні

з іншими побудованими моделями для даного виду архітектури. Результати перехресного затвердження лог-лінійної моделі наведені в таблиці 3.24.

Таблиця 3.24.

**Результати перехресного затвердження лог-лінійної регресії для
модульної монолітної архітектури зі слабкою власністю коду**

Метрика	Середнє значення	Стандартне відхилення
R^2	0.5312	0.0373
MAE	1.7625	0.1134
RMSE	2.3871	0.1463

Ансамблеві моделі Random Forest та Gradient Boosting виявили наступні оцінки важливості входних параметрів при передбаченні продуктивності процесу розробки програмного забезпечення в контексті поточної модульної монолітної архітектурної моделі зі слабкою власністю програмного коду, що наведені в таблиці 3.25.

Таблиця 3.25.

**Оцінки важливості входних ознак для модульної монолітної архітектури зі
слабкою власністю коду**

Параметр	Random Forest	Gradient Boosting
Кількість модулів	12.71%	12.64%
Кількість рядків коду	17.47%	17.49%
Кількість команд розробки	4.82%	5.09%
Кількість розробників в команді	5.60%	5.29%
Кількість тестувальників в команді	15.57%	15.64%
Кількість різних часових поясів	39.67%	39.70%
Максимальна кількість активних задач	4.15%	4.15%

В результаті тренування моделей на навчальній вибірці отримані наступні значення коефіцієнтів детермінації R^2 : Random Forest – 0.9293, Gradient Boosting

– 0.9429. В контексті даної архітектурної моделі Gradient Boosting так само показує найбільшу здатність до оцінки продуктивності розробки в порівнянні зі всіма іншими методами, але перевага перед методом Random Forest незначна. Різниця між значеннями коефіцієнтів детермінації обох моделей, отриманими в процесі навчання та в результаті перехресного затвердження, складає менше 1%, що говорить про контрольоване перенавчання, яке критично не впливає на здатність даних моделей до передбачення значень залежної змінної. Результати 10-кратного перехресного затвердження обох ансамблевих моделей наведені в таблиці 3.26.

Таблиця 3.26.

Результати 10-кратного перехресного затвердження ансамблевих моделей для модульної монолітної архітектури зі слабкою власністю коду

	Random Forest		Gradient Boosting	
Метрика	Середнє значення	Стандартне відхилення	Середнє значення	Стандартне відхилення
R^2	0.9206	0.0147	0.9340	0.0134
MAE	0.6770	0.0499	0.6066	0.0429
RMSE	0.9782	0.0943	0.8909	0.0961

Аналіз отриманих результатів. Всі значення коефіцієнтів детермінації R^2 по всіх побудованих регресійним моделям для всіх досліджуваних архітектурних моделей та типів власності коду наведена у зведеній таблиці 3.27. Як видно із зведеної таблиці 3.27, Gradient Boosting стабільно посідає перше місце за для всіх чотирьох архітектур. Слідом іде Random Forest, який показує результати, схожі до Gradient Boosting з невеликим відставанням. Серед параметричних моделей, поліноміальна регресія другого ступеня є найкращою, проте все одно погано пояснює дисперсію продуктивності розробки в контексті звичайної монолітної архітектури. Так як лінійна регресія суттєво поступається поліноміальній, це підтверджує наявність значних нелінійних ефектів. Лог-

лінійна модель показала найменшу точність оцінки ефективності розробки в контексті всіх типів архітектури, що досліджуються.

Таблиця 3.27.

Зведена таблиця коефіцієнтів детермінації для всіх регресійних моделей та типів архітектури з різними видами власності коду

Архітектура	Власність коду	Лінійна регресія	Поліноміальна другого ступеня	Лог-лінійна	Random Forest	Gradient Boosting
Мікросервісна	Сильна	0.5529	0.7577	0.3686	0.9180	0.9227
Монолітна	Колективна	0.4111	0.5220	0.3861	0.9617	0.9657
Модульна монолітна	Сильна	0.6901	0.8518	0.6823	0.9288	0.9350
Модульна монолітна	Слабка	0.6428	0.7593	0.5397	0.9293	0.9429

Найвищу загальну точність оцінки продуктивності демонструє монолітна архітектура з колективної власністю програмного коду, що обґрунтовується максимальною незалежністю команд розробки в порівнянні з іншими архітектурними стилями. Найнижчу загальну точність серед ансамблевих моделей має мікросервісна архітектура, що свідчить про більшу стохастичність процесу розробки та можливу наявність неврахованих факторів в симуляційній моделі.

Аналіз важливості ознак ансамблевих моделей виявив суттєві відмінності у ключових драйверах продуктивності залежно від архітектурної моделі. Для мікросервісної архітектури домінуючими факторами є кількість часових поясів, кількість розробників у команді та загальна кількість команд розробки, що вказує на критичний вплив географічного розподілу команд на продуктивність розробки через постійну необхідність узгодження дій між різними командами. У випадку немодульної монолітної архітектури основними факторами є обсяг коду та максимальна дозволена кількість незавершених задач на одного розробника,

тоді як кількість різних часових поясів відіграє менш значущу, але все ще важливу роль. Для модульної монолітної архітектури із сильною власністю коду вплив розподілений більш рівномірно між кількістю програмних модулів, кількістю тестувальників в кожній команді, кількістю команд розробки та їх розподілу між часовими поясами. В умовах монолітної архітектури зі слабкою власністю програмного коду найбільш впливовим параметром є кількість часових поясів, за яким слідують обсяг коду, кількість тестувальників та кількість програмних модулів системи.

Для практичного застосування регресійних моделей з метою оцінки ефективності процесу розробки програмного забезпечення рекомендується використовувати ансамблеві моделі (Gradient Boosting або Random Forest), оскільки вони забезпечують найвищу точність. Водночас для аналітичного розуміння факторів впливу доцільно поєднувати ансамблеві моделі з множинною лінійною регресією, коефіцієнти якої мають чітку інтерпретацію. Поліноміальна регресія другого ступеня є хорошим компромісом між можливістю інтерпретації та прогнозною здатністю, особливо коли важливо враховувати ефекти взаємодії.

Результати дослідження підтверджують, що вибір архітектурної моделі суттєво впливає не лише на абсолютний рівень продуктивності, але й на відносну значущість окремих організаційних та технічних параметрів. Це має бути враховано при прийнятті управлінських рішень щодо структури команд, географічного розташування та обмежень паралельної роботи.

Обидві ансамблеві моделі, натреновані на симуляційних даних, для всіх типів архітектури доступні для завантаження на платформі Zenodo [101]. Усі 8 моделей збережено у форматі «Joblib» для зручного завантаження та використання в середовищі програмування Python. Кожен файл містить словник із ключами «model» (навчена модель), «features» (список 7 вхідних ознак) та «arch_own» (назва типу архітектури). Всі моделі прогнозують очікувану кількість завершених функцій на одного розробника протягом 60 робочих днів.

Оцінку загальної кількості функцій можна отримати шляхом множення отриманого значення на загальну кількість розробників по всім командам.

3.4. Валідація симуляційної моделі процесу розробки ПЗ методом експертних оцінок

В процесі тренування регресійних моделей оцінки ефективності процесу розробки програмного забезпечення вже була проведена валідація даних математичних моделей методом 10-кратного перехресного затверджування для перевірки якості регресійного аналізу. Проте всі вони були натреновані на основі експериментальних даних, отриманих в результаті багаторазових симуляцій різних процесів розробки ПЗ, тому валідація самої симуляційної моделі є необхідною для підтвердження її відповідності реалістичним процесам розробки в індустрії програмного забезпечення.

Валідація симуляційної моделі оцінки ефективності процесу розробки програмного забезпечення для різних видів програмної архітектури була здійснена за допомогою методу експертних оцінок. Для опитування було залучено 17 експертів-професіоналів з індустрії ПЗ, серед яких 9 досвідчених розробників, 5 архітекторів програмного забезпечення та 3 проєктних менеджера.

Респондентам був наданий стислий опис ключових особливостей симуляційної моделі, таких як перелік вхідних параметрів моделі, відмінності процесів розробки для різних типів архітектури та видів власності коду, деталі реалізації організаційних аспектів процесу розробки ПЗ та симуляції людських факторів. Додатково до даного опису були надані посилання на відповідні наукові статті, в яких викладено алгоритмічні основи симуляційної моделі, та анкета для опитування, що наведена в додатку Г.

Анкета містить перелік з 20 тверджень-показників, для кожного з яких опитувані мають вказати ступінь їх згоди з даними твердженнями, вказавши оцінку за 4-бальною шкалою: 0 балів – абсолютно не згодні, 1 бал – скоріше не

згодні, ніж згодні, 2 бали – скоріше згодні, ніж не згодні, 3 бали – абсолютно згодні. Показник вважається позитивним, якщо його середнє арифметичне значення (коефіцієнт показника) складає не менше 1,5.

Всі твердження-показники об'єднані у відповідні групи-критерії, за якими оцінюється валідність симуляційної моделі. Критерій вважається не достатньо проявленим, якщо менше половини його показників є позитивними; критичний прояв критерію – 50%-55%; критерій достатньо проявлений при 56%-75%; високий прояв критерію – 76%-100%.

Для валідації симуляційної моделі процесу розробки програмного забезпечення для оцінки його ефективності в контексті різних типів архітектури програмних систем були виділені наступні критерії оцінки її валідності.

Структурний критерій характеризує ступінь адекватності внутрішньої будови моделі відносно досліджуваної предметної області. Він спрямований на оцінювання того, наскільки повно та коректно модель відтворює склад елементів, зв'язки між ними та загальну організацію процесу розробки програмного забезпечення.

Показник *«достатність вхідних параметрів»* визначає, чи охоплює множина вхідних змінних моделі всі суттєві фактори, що впливають на перебіг процесу розробки в реальних умовах. Експерти оцінюють, чи не було пропущено критично важливих параметрів і чи не містить модель надлишкових змінних, які не мають практичного значення для досліджуваних сценаріїв.

Показник *«актуальність типів архітектури»* відображає відповідність представлених у моделі архітектурних підходів сучасному стану індустрії розробки програмного забезпечення. Респонденти оцінюють, чи відповідає перелік реалізованих архітектурних типів тим, що переважають у сучасній практиці проєктування програмних систем, та чи є цей перелік достатньо репрезентативним для цілей дослідження.

Показник *«відповідність видів власності програмного коду типам архітектури»* фіксує ступінь узгодженості між реалізованими в моделі формами власності програмного коду та обраними архітектурними підходами та дозволяє

встановити, чи коректно модель відтворює залежність між архітектурними рішеннями та організаційними практиками щодо розподілу відповідальності за компоненти програмного коду.

Показник *«коректність організаційного процесу розробки»* характеризує ступінь відповідності послідовності етапів, ролей та взаємодій, реалізованих у моделі, реальним організаційним процесам розробки програмного забезпечення. Експерти оцінюють, чи адекватно відтворено життєвий цикл задач, розподіл обов'язків між учасниками та координацію діяльності всередині команд.

Показник *«реалістичність поведінки агентів-розробників»* визначає, наскільки достовірно модель імітує дії, рішення та обмеження, притаманні реальним учасникам процесу розробки в контексті роботи над власними задачами, рецензування програмного коду інших розробників та перемикавання на попередні задачі, що потребують додаткової уваги або доопрацювання.

Показник *«доцільність моделювання втоми від частого рецензування коду»* оцінює обґрунтованість включення до моделі механізму зниження продуктивності розробників унаслідок надмірного навантаження, пов'язаного з процесом рецензування програмного коду. Експерти визначають, чи є цей ефект достатньо значущим у реальній практиці, щоб його врахування підвищувало достовірність результатів моделювання.

Проміжні дані опитування та результати їх опрацювання наведено в таблиці Д.1 додатку Д. Основні дані містить таблиця 3.28.

Таблиця 3.28.

Показники валідності симуляційної моделі за структурним критерієм

№	Показники критерію	Коефіцієнт показника	Проявлення критерію
1.	Достатність вхідних параметрів	1,41	67%
2.	Актуальність типів архітектури	2,82	
3.	Відповідність видів власності програмного коду типам архітектури	1,88	
4.	Коректність організаційного процесу розробки	2,94	
5.	Реалістичність поведінки агентів-розробників	1,47	
6.	Доцільність моделювання втоми від частого рецензування коду	2,59	

Алгоритмічний критерій спрямований на оцінювання якості та обґрунтованості обчислювальних процедур, покладених в основу функціонування симуляційної моделі, що дозволяє встановити, наскільки коректно формалізовано ключові процеси та чи забезпечують застосовані алгоритми адекватне відтворення динаміки досліджуваної системи.

Показник *«обґрунтованість алгоритму розподілу складності задач»* відображає ступінь відповідності механізму генерації задач різної складності реальним закономірностям, які спостерігаються в програмних проєктах.

Показник *«обґрунтованість алгоритму моделювання зв'язності модулів»* характеризує адекватність формалізації взаємозалежностей між модулями програмної системи в контексті мікросервісної та модульної монолітної архітектур. Оцінюється, чи коректно алгоритм моделює ступінь зв'язності модулів при моделюванні внесення в них змін на основі їх незалежних імовірностей модифікації.

Показник *«коректність алгоритму розподілу задач між командами»* визначає, наскільки адекватно модель відтворює процес розподілу задач в залежності від заданого виду власності програмного коду та архітектури програмної системи, де експерти оцінюють відповідність даного алгоритму реальним практикам управління проєктами.

Показник *«реалістичність алгоритму переходу задач між різними станами»* фіксує ступінь правдоподібності послідовності зміни станів задач протягом їхнього життєвого циклу, де експерти оцінюють адекватність відтворення таких переходів, а також реалістичність процесу повернення задачі на попередні етапи.

Проміжні дані опитування та результати їх опрацювання наведено в таблиці Д.2 додатку Д. Основні дані містить таблиця 3.29.

Таблиця 3.29.

Показники валідності симуляційної моделі за алгоритмічним критерієм

№	Показники критерію	Коефіцієнт показника	Проявлення критерію
1.	Обґрунтованість алгоритму розподілу складності задач	2,76	100%
2.	Обґрунтованість алгоритму моделювання зв'язності модулів	1,82	
3.	Коректність алгоритму розподілу задач між командами	2,35	
4.	Реалістичність алгоритму переходу задач між різними станами	2,94	

Концептуальний критерій оцінює адекватність теоретичних припущень та причинно-наслідкових зв'язків, покладених в основу моделі. Він визначає, наскільки коректно модель відтворює фундаментальні закономірності впливу архітектурних, організаційних та географічних факторів на процес розробки.

Показник *«коректність впливу архітектури на організаційні аспекти процесу розробки»* характеризує ступінь достовірності, з якою модель відтворює залежність організаційної структури, комунікаційних шаблонів та процесів координації від обраного типу архітектури програмної системи.

Показник *«коректність впливу власності програмного коду на організаційні аспекти процесу розробки»* визначає, наскільки адекватно модель відтворює вплив різних форм власності програмного коду на процеси його рецензування, координації між командами та розподілу відповідальності між учасниками розробки. Оцінюється правдоподібність модельованих відмінностей між колективною, сильною та слабкою формами власності коду.

Показник *«реалістичність впливу географічного розподілу команд розробки на організаційні аспекти процесу розробки»* відображає ступінь достовірності відтворення ефектів, які виникають внаслідок географічного розподілу команд розробки серед різних часових поясів.

Показник *«актуальність модульної монолітної архітектури»* визначає обґрунтованість включення до моделі цього архітектурного підходу як окремого

типу. Експерти оцінюють, чи є модульний моноліт достатньо поширеним та значущим у сучасній практиці розробки, щоб його виокремлення як самостійного варіанта архітектури було виправданим з наукової та практичної точок зору.

Проміжні дані опитування та результати їх опрацювання наведено в таблиці Д.3 додатку Д. Основні дані містить таблиця 3.30.

Таблиця 3.30.

Показники валідності симуляційної моделі за концептуальним критерієм

№	Показники критерію	Коефіцієнт показника	Проявлення критерію
1.	Коректність впливу архітектури на організаційні аспекти процесу розробки	3	100%
2.	Коректність впливу власності програмного коду на організаційні аспекти процесу розробки	2,47	
3.	Реалістичність впливу географічного розподілу команд розробки на організаційні аспекти процесу розробки	2,88	
4.	Актуальність модульної монолітної архітектури	2,29	

Прогностичний критерій спрямований на оцінювання здатності моделі генерувати достовірні оцінки щодо впливу різних факторів на ефективність процесу розробки програмного забезпечення та визначає, наскільки реалістичними є вихідні дані моделі при варіюванні ключових вхідних параметрів.

Показник *«реалістичність впливу практики власності коду на ефективність процесу розробки»* характеризує ступінь правдоподібності прогнозованих моделлю змін у продуктивності при переході між різними формами власності програмного коду, при якому експерти оцінюють, чи відповідають отримані результати емпіричним даним та їхньому професійному досвіду.

Показник *«реалістичність впливу вибору архітектури на ефективність розробки»* визначає, наскільки достовірно модель прогнозує зміну ключових

показників ефективності залежно від обраного архітектурного підходу, де респонденти порівнюють результати, отримані шляхом симуляційного експерименту, з власним попереднім досвідом.

Показник *«реалістичність впливу географічного розподілу команд на ефективність процесу розробки»* відображає ступінь достовірності прогнозів моделі щодо зміни показників ефективності розробки в умовах географічно розподілених команд розробки поміж різних часових поясів. Експерти оцінюють реалістичність відтворення ефектів розподіленої розробки та її відповідність минулим спостереженням з реальних проєктів.

Показник *«реалістичність впливу архітектури на масштабування розробки»* оцінює здатність моделі адекватно прогнозувати зміну характеристик процесу розробки при збільшенні кількості команд розробки, розробників або модулів системи. Респонденти визначають, чи коректно модель відтворює обмеження масштабування, притаманні різним архітектурним підходам, та чи відповідають ці результати відомим закономірностям.

Проміжні дані опитування та результати їх опрацювання наведено в таблиці Д.4 додатку Д. Основні дані містить таблиця 3.31.

Таблиця 3.31.

Показники валідності симуляційної моделі за прогностичним критерієм

№	Показники критерію	Коефіцієнт показника	Проявлення критерію
1.	Реалістичність впливу практики власності коду на ефективність процесу розробки	1,71	83%
2.	Реалістичність впливу вибору архітектури на ефективність розробки	2,65	
3.	Реалістичність впливу географічного розподілу команд на ефективність процесу розробки	2,82	
4.	Реалістичність впливу архітектури на масштабування розробки	1,41	

Прикладний критерій оцінює практичну цінність та доцільність використання розробленої симуляційної моделі для вирішення реальних завдань,

пов'язаних з управлінням процесом розробки програмного забезпечення. Цей критерій дозволяє встановити, чи може модель бути корисним інструментом для підтримки прийняття рішень у реальних умовах.

Показник *«доцільність використання симуляційної моделі для дослідження ефективності процесу розробки»* визначає обґрунтованість застосування саме симуляційного підходу для аналізу впливу архітектурних та організаційних факторів на ефективність розробки. Опитувані оцінюють, чи має розроблена модель достатній рівень достовірності та деталізації для проведення обґрунтованих досліджень, які дозволяють отримати нове знання про досліджувану предметну область.

Показник *«актуальність результатів симуляцій для прийняття практичних рішень»* характеризує ступінь придатності вихідних даних симуляційної моделі для використання як аналітичної основи при прийнятті майбутніх управлінських рішень щодо вибору архітектури, організації команд та визначення моделі власності коду в реальних програмних проєктах. Експерти визначають, чи можуть результати моделювання слугувати надійною доказовою базою для обґрунтування проєктних та організаційних рішень.

Проміжні дані опитування та результати їх опрацювання наведено в таблиці Д.5 додатку Д. Основні дані містить таблиця 3.32.

Таблиця 3.32.

Показники валідності симуляційної моделі за прикладним критерієм

№	Показники критерію	Коефіцієнт показника	Проявлення критерію
1.	Доцільність використання симуляційної моделі для дослідження ефективності процесу розробки	2,47	100%
2.	Актуальність результатів симуляцій для прийняття практичних рішень	2,82	

Аналіз наведених результатів дозволяє встановити такі ступені проявлення критеріїв валідності симуляційної моделі процесу розробки програмного забезпечення: високий – алгоритмічний, концептуальний, прогностичний та прикладний критерії; достатній – структурний.

На підставі проведеного експертного опитування можна зробити висновок, що розроблена агентно-орієнтована симуляційна модель процесу розробки програмного забезпечення для різних типів архітектури серверних програмних систем є валідною. Модель адекватно відображає структуру процесу розробки, коректно моделює внутрішні механізми процесу розробки ПЗ, базується на обґрунтованих концептуальних припущеннях та алгоритмічних основах, а також демонструє прийнятну прогностичну здатність.

Найнижчі експертні оцінки отримали структурні показники достатності варіативних вхідних параметрів та реалістичності поведінки агентів-розробників, що може бути враховане в подальших похідних дослідженнях.

Результати моделювання можуть бути використані для прийняття практичних рішень щодо вибору програмної архітектури, організації команд розробки та оптимізації процесу розробки програмного забезпечення.

Висновки до розділу 3

У третьому розділі розроблено симуляційну модель процесу розробки програмного забезпечення на основі агентного моделювання з використанням фреймворку MARS LIFE, проведено комплексний аналіз результатів 8080 симуляцій, побудовано регресійні моделі оцінки ефективності розробки в контексті різних типів архітектури серверних програмних систем та проведено валідацію симуляційної моделі методом експертних оцінок.

Розроблена симуляційна модель охоплює повний життєвий цикл задач від програмування до розгортання, включаючи етапи рецензування коду та тестування. Дана модель дозволяє варіювати архітектурні параметри (монолітна, модульна монолітна та мікросервісна архітектура), організаційні характеристики (кількість та розмір команд, географічний розподіл), параметри кодової бази (обсяг коду, кількість модулів) та практики управління (вид власності коду, обмеження кількості паралельних задач).

Встановлено, що вибір архітектурної моделі є визначальним фактором продуктивності розробки. Мікросервісна архітектура демонструє найвищу

середню продуктивність, тоді як модульна монолітна архітектура з сильною власністю коду є найменш продуктивною. Різниця між найкращою та найгіршою моделями складає 3.5 рази, що підтверджує критичну роль архітектурних рішень.

Виявлено, що географічний розподіл команд між часовими поясами є найбільш універсально значущим фактором зниження продуктивності для всіх архітектурних моделей, але з різною інтенсивністю. Мікросервісна архітектура втрачає 51% продуктивності при розподілі команд між 5 часовими поясами, тоді як звичайна монолітна архітектура – лише 16% завдяки максимальній незалежності команд. Водночас перехід від 1 до 2 часових поясів може навіть незначно підвищувати продуктивність за рахунок ефекту подовженого робочого дня.

Доведено, що збільшення кількості модулів у модульній монолітній архітектурі з сильною власністю коду призводить до колапсу продуктивності на 78% при зростанні кількості модулів з 10 до 100, що зумовлено лінійним зростанням накладних витрат на інтеграційне тестування. Для мікросервісної архітектури збільшення кількості модулів при сталому загальному об'ємі програмного коду, навпаки, підвищує продуктивність на 15% за рахунок зменшення розміру окремих сервісів. Також встановлено, що мікросервісна архітектура є найбільш масштабованою моделлю, де обсяг кодової бази не має статистично значущого впливу на продуктивність, тоді як для звичайної монолітної архітектури виявлено нелінійну залежність з оптимумом при 200000-300000 рядках коду, після чого продуктивність знижується.

Визначено, що характер впливу розміру команд суттєво залежить від архітектурної моделі. Для мікросервісної архітектури оптимальними є малі команди з 2 розробників та 2 тестувальників, тоді як для модульної монолітної архітектури критичним є паритетне співвідношення тестувальників до розробників, оскільки тестування є головним вузьким місцем процесу розробки. Звичайна монолітна архітектура демонструє найбільшу інваріантність до розміру команд та найкраще лінійне масштабування при додаванні нових команд

– зростання загальної продуктивності у 9.2 рази при 10-кратному збільшенні кількості команд.

Побудовано та порівняно п'ять типів регресійних моделей оцінки ефективності розробки для кожної архітектурної моделі. Ансамблеві моделі (Gradient Boosting та Random Forest) стабільно забезпечують найвищу точність передбачення значень вихідної змінної з коефіцієнтом детермінації R^2 від 0.92 до 0.97. Найвищу загальну точність демонструє монолітна архітектура з колективною власністю коду ($R^2 = 0.97$), що пояснюється максимальною незалежністю команд, а найнижчу точність серед ансамблевих моделей має мікросервісна архітектура ($R^2 = 0.92$), що свідчить про більшу стохастичність процесу розробки. Аналіз важливості ознак ансамблевих моделей виявив, що ключові фактори продуктивності суттєво відрізняються залежно від архітектурної моделі: для мікросервісної архітектури домінуючими є кількість часових поясів та кількість розробників у команді, а для монолітної – обсяг коду та обмеження паралельних задач, для модульної монолітної з сильною власністю коду вплив розподілений більш рівномірно між кількістю модулів, тестувальників, команд та часових поясів.

На основі проведеного аналізу сформульовано практичні рекомендації щодо оптимальних параметрів для кожної архітектурної моделі та обґрунтовано доцільність використання ансамблевих моделей для оцінки ефективності процесів розробки у поєднанні з множинною лінійною регресією для аналітичного розуміння факторів впливу.

Симуляційна модель процесу розробки програмного забезпечення для різних типів архітектури програмних систем була провалідована методом експертних оцінок, а побудовані регресійні математичні моделі після тренування були перевірені методом 10-кратного перехресного затвердження.

Матеріали розділу 3 подано в таких публікаціях автора [101].

Висновки

Відповідно до поставленої мети та завдань дослідження з проблеми моделювання ефективності розробки програмного забезпечення в залежності від типу архітектури серверних програмних систем отримано такі основні **результати**: проведено аналіз останніх досліджень у галузі ефективності розробки програмного забезпечення; досліджено архітектурні фактори впливу на продуктивність розробки програмного забезпечення та метрики оцінки ефективності процесу розробки; розроблено алгоритми імовірнісної генерації потоку вхідних задач різної складності та моделювання зв'язності модулів програмної системи на основі історичних даних проєктів з відкритим вихідним кодом; проведено симуляції процесу розробки програмного забезпечення в контексті різних архітектурних моделей та визначено значення параметрів максимальної ефективності розробки окремо для кожного типу архітектури; побудовано регресійні моделі оцінки ефективності процесу розробки для різних типів архітектури програмних систем на основі результатів симуляцій.

Результати дослідження дали підстави для таких **висновків**:

1. Аналіз останніх досліджень у галузі ефективності розробки програмного забезпечення засвідчив, що симуляційне моделювання процесів розробки програмного забезпечення залишається активною та динамічною областю наукових досліджень, а агентно-орієнтований підхід набуває дедалі більшої популярності завдяки можливості відтворювати індивідуальну поведінку розробників, враховувати неоднорідність команд, вплив архітектурного стилю системи та моделювати стохастичні комунікаційні взаємодії. В результаті дослідження архітектурних факторів впливу на продуктивність розробки програмного забезпечення та метрик оцінки ефективності процесу розробки було виявлено, що істотний вплив на ефективність процесу розробки програмного забезпечення має організаційна структура та географічний розподіл команд розробки, розмір кодової бази та зв'язність модулів коду між собою, вид власності програмного коду та процес

рецензування коду, що суттєво залежить від застосованого виду практики власності програмного коду.

2. На основі аналізу історичних даних проєктів з відкритим вихідним кодом, що входять до фонду «Apache Software Foundation», проаналізовано розподіл складності задач розробки програмного забезпечення та розроблено алгоритм розподілу імовірностей для генерації потоку вхідних задач різної складності в програмних симуляціях. Дослідження історії внесення змін до кодової бази 8 модульних проєктів з відкритим вихідним кодом, як в монолітній архітектурі, так і у мікросервісному виконанні, дозволило розробити алгоритм моделювання зв'язності між програмними модулями шляхом розрахунку розподілів незалежних імовірностей зміни модулів для кодових баз з різною загальною їх кількістю. Розроблені алгоритми дозволяють наблизити симуляційну модель процесу розробки програмного забезпечення до більш реалістичних умов, що підвищує її достовірність.

3. Проведено 8080 симуляцій процесу розробки програмного забезпечення в контексті різних архітектурних моделей. Симуляційна модель охоплює повний життєвий цикл задач від програмування до розгортання, включаючи етапи рецензування коду та тестування. Дана модель дозволяє варіювати архітектурні параметри (монолітна, модульна монолітна та мікросервісна архітектура в поєднанні з відповідними видами власності коду), організаційні характеристики (кількість та розмір команд, географічний розподіл) та параметри кодової бази (обсяг коду, кількість модулів). Симуляційна модель процесу розробки програмного забезпечення була провалідована методом експертних оцінок. Створений програмний комплекс симуляції процесів розробки програмного забезпечення дозволяє проводити детальне дослідження перебігу процесу розробки з метою ідентифікації його сповільнюючих факторів та подальшої оптимізації.

4. На основі результатів симуляцій були визначені значення параметрів максимальної ефективності розробки окремо для кожного типу архітектури, а також виявлено, що: мікросервісна архітектура демонструє найвищу середню

продуктивність та є найбільш масштабованою моделлю, але водночас є найбільш чутливою до географічного розподілу команд; монолітна архітектура з колективною власністю коду забезпечує найкраще лінійне масштабування при додаванні нових команд та найбільшу стійкість до географічного розподілу, а також демонструє найвищу загальну точність оцінки ефективності розробки серед усіх архітектурних моделей, що пояснюється максимальною незалежністю команд; модульна монолітна архітектура з сильною власністю коду виявилась найменш продуктивною серед досліджених моделей та продемонструвала колапс продуктивності при зростанні кількості модулів, що зумовлено лінійним зростанням накладних витрат на інтеграційне тестування, крім того критичним для цієї архітектури є паритетне співвідношення тестувальників до розробників, оскільки тестування є головним вузьким місцем процесу розробки. Географічний розподіл команд між часовими поясами є найбільш універсально значущим фактором зниження продуктивності для всіх архітектурних моделей, хоча інтенсивність цього впливу суттєво варіюється залежно від обраної архітектури. Водночас перехід від 1 до 2 часових поясів може навіть незначно підвищувати продуктивність за рахунок ефекту подовженого робочого дня. Ключові фактори продуктивності суттєво відрізняються залежно від архітектурної моделі: для мікросервісної архітектури домінуючими є кількість часових поясів та кількість розробників у команді, для монолітної – обсяг коду та обмеження паралельних задач, а для модульної монолітної архітектури з сильною власністю коду вплив розподілений більш рівномірно між кількістю модулів, команд, тестувальників в командах та географічним розподілом команд розробки. Ця диференціація підтверджує необхідність індивідуального підходу до оптимізації процесів розробки залежно від обраного архітектурного рішення.

5. Побудовано та порівняно п'ять типів регресійних моделей оцінки ефективності розробки для кожної архітектурної моделі. Ансамблеві моделі (Gradient Boosting та Random Forest) стабільно забезпечують найвищу точність передбачення значень вихідної змінної, тоді як множинна лінійна регресія надає можливість аналітичного розуміння факторів впливу. Аналіз важливості ознак

ансамблевих моделей виявив, що ключові фактори продуктивності суттєво відрізняються залежно від архітектурної моделі. Побудовані математичні моделі оцінки ефективності розробки програмного забезпечення можуть бути використані для додаткової аргументації при первинному виборі між сервісно-орієнтованим та модульним монолітним архітектурними підходами для майбутньої серверної програмної системи, а також при оцінці доцільності переходу з поточного архітектурного рішення на альтернативне.

Виконане дослідження не вичерпує всіх аспектів поставленої проблеми. Продовження наукових досліджень за даною проблематикою доцільно у таких напрямках:

- Врахування більшої кількості соціальних аспектів процесу розробки програмного забезпечення, таких як наради, задачі, які безпосередньо не пов'язані з роботою над поточною програмною системою, парне програмування, пізні зміни пріоритетів задач, раптове перемикання на вирішення екстрених проблем тощо;
- Моделювання процесу розробки програмної системи з врахуванням еволюційного розвитку програмного коду, його модульної структури та характеристик якості новостворених функцій програмної системи, що, як відомо, відрізняються між різними стилями архітектури програмних систем;
- Врахування сучасного тренду розробки з використанням технологій штучного інтелекту в симуляційній моделі процесу розробки ПЗ, так як в даному контексті основний фокус зміщується зі швидкого написання якісного програмного коду до рецензування програмного коду, написаного штучним інтелектом;
- Моделювання процесу розробки програмної системи в контексті різних стратегій забезпечення якості програмного продукту, оскільки окрема роль тестувальників в певній ІТ-компанії може бути відсутня взагалі та компенсована за рахунок автоматизованих тестів або штучного інтелекту.

Список використаних джерел

1. Mallampati B. Demystifying cloud-native microservices architecture for scalable applications. *World Journal of Advanced Engineering Technology and Sciences*. 2025. Vol. 15, № 1. P. 1806–1817. DOI: 10.30574/wjaets.2025.15.1.0422.
2. Su R., Li X. Modular Monolith: Is This the Trend in Software Architecture?. *SATrends '24: 1st International Workshop on New Trends in Software Architecture*, Lisbon Portugal. New York, NY, USA, 2024. P. 10–13. DOI: 10.1145/3643657.3643911.
3. Boehm B. W. Software engineering economics. Englewood Cliffs : Prentice Hall, 1981. 767 p.
4. Boehm B. W., Abts C., Brown A. W., Chulani S., Clark B. K., Horowitz E., Madachy R., Reifer D. J., Steece B. Software cost estimation with COCOMO II. Upper Saddle River : Prentice Hall, 2000. 544 p.
5. Baxter G., Sommerville I. Socio-technical systems: From design methods to systems engineering. *Interacting with Computers*. 2011. Vol. 23, no. 1. P. 4–17. DOI: 10.1016/j.intcom.2010.07.003.
6. Software Market Size, Share and Trends 2025 to 2034. Precedence Research, 2025. URL: <https://www.precedenceresearch.com/software-market> (дата звернення: 05.03.2025).
7. Bloch M., Blumberg S., Laartz J. Delivering large-scale IT projects on time, on budget, and on value. *Harvard Business Review*. 2012. P. 2–7.
8. Duarte C. H. C. Software Productivity in Practice: A Systematic Mapping Study. *Software*. 2022. Vol. 1, No. 2. P. 164–214. DOI: 10.3390/software1020008.
9. Pasuksmit J., Thongtanunam P., Karunasekera S. A Systematic Literature Review on Reasons and Approaches for Accurate Effort Estimations in Agile. *ACM Computing Surveys*. 2024. Vol. 56, No. 11. P. 1–37. DOI: 10.1145/3663365.
10. García-García J. A., Enríquez J. G., Ruiz M., Arévalo C., Jiménez-Ramírez A. Software process simulation modeling: Systematic literature review. *Computer Standards & Interfaces*. 2020. Vol. 70. P. 103425. DOI: 10.1016/j.csi.2020.103425.

11. Li Y., Zhang H., Liu B., Dong L., Gong H., Rong G. Verification and validation of software process simulation models: A systematic mapping study. *Journal of Software: Evolution and Process*. 2024. Vol. 36, no. 6. P. e2612. DOI: 10.1002/smr.2612.
12. O'Reilly Media. Microservices Adoption in 2020. O'Reilly Media, 2020. URL: <https://www.oreilly.com/radar/microservices-adoption-in-2020/> (дата звернення: 30.06.2025).
13. Microservices vs. Monolithic Architecture: A Decision Framework with Real-World Examples. Offshore Development Company & Staff Augmentation in the Philippines - Full Scale. 2025. URL: <https://fullscale.io/blog/microservices-vs-monolithic-architecture/> (дата звернення: 15.07.2025).
14. Van Hai V., Tin N. T., Prokopova Z. Software Architecture Influence on Software Effort Estimation. *System Design in Software Engineering. CoMeSySo 2024 : Lecture Notes in Networks and Systems*. Vol. 1471 / eds. R. Silhavy, P. Silhavy. Cham : Springer, 2025. P. 321–331. DOI: 10.1007/978-3-031-94770-4_25.
15. Abdel-Hamid T. K., Madnick S. E. Software project dynamics: An integrated approach. Englewood Cliffs : Prentice Hall, 1991. 264 p.
16. Kellner M. I., Madachy R. J., Raffo D. M. Software process simulation modeling: Why? What? How?. *Journal of Systems and Software*. 1999. Vol. 46, no. 2-3. P. 91–105. DOI: 10.1016/S0164-1212(99)00003-5.
17. Alturki S., E-amin F. Comprehensive Analysis of Software Effort Estimation Techniques: Evolving Trends, Key Challenges, and Prospective Directions. *International Journal of Computer Applications*. 2025. Vol. 186, No. 68. P. 42–48. DOI: 10.5120/ijca2025924523.
18. Zhang H., Kitchenham B., Pfahl D. Reflections on 10 years of software process simulation modeling: A systematic review. Making globally distributed software development a success story. ICSP 2008 / eds. Q. Wang, D. Pfahl, D. M. Raffo. *Lecture Notes in Computer Science*. Vol. 5007. Berlin : Springer, 2008. P. 345–365. DOI: 10.1007/978-3-540-79588-9_30.

19. Liu B., Zhang H., Dong L., Wang Q., Li S. Metrics for software process simulation modeling. *Journal of Software: Evolution and Process*. 2024. Vol. 36, no. 12. P. e2676. DOI: 10.1002/smr.2676.
20. Antelmi A., Cordasco G., D'Ambrosio G., De Vinco D., Spagnuolo C. Experimenting with Agent-Based Model Simulation Tools. *Applied Sciences*. 2023. Vol. 13, No. 1. Art. 13. DOI: 10.3390/app13010013.
21. Cherif R., Davidsson P. Software development process simulation: Multi agent-based simulation versus system dynamics. Multi-agent-based simulation X. MABS 2009 / eds. G. Di Tosto, H. Van Dyke Parunak. *Lecture Notes in Computer Science*. Vol. 5683. Berlin : Springer, 2010. P. 73–85. DOI: 10.1007/978-3-642-13553-8_7.
22. Spasic B., Onggo B. S. S. Agent-based simulation of the software development process: A case study at AVL. *Proceedings of the 2012 Winter Simulation Conference (WSC)*. Piscataway : IEEE, 2012. P. 1–11. DOI: 10.1109/wsc.2012.6465117.
23. Smith N., Capiluppi A., Fernández-Ramil J. Users and developers: An agent-based simulation of open source software evolution. Software process change. SPW 2006 / eds. Q. Wang, D. Pfahl, D. M. Raffo, P. Wernick. *Lecture Notes in Computer Science*. Vol. 3966. Berlin : Springer, 2006. P. 286–296. DOI: 10.1007/11754305_31.
24. Barde K. Modular Monoliths: Revolutionizing Software Architecture for Efficient Payment Systems in Fintech. *International Journal of Computer Trends and Technology*. 2023. Vol. 71, no. 10. P. 20–27. DOI: 10.14445/22312803/IJCTT-V71I10P103.
25. Antelmi A., Caramante P., Cordasco G., D'Ambrosio G., De Vinco D., Foglia F., Postiglione L., Spagnuolo C. Reliable and Efficient Agent-Based Modeling and Simulation. *Journal of Artificial Societies and Social Simulation*. 2024. Vol. 27, No. 2. Art. 4. DOI: 10.18564/jasss.5300.
26. Brooks F. P., Jr. The mythical man-month: Essays on software engineering. Anniversary ed. Reading : Addison-Wesley, 1995. 322 p.

27. Gunther N. J. *Guerrilla capacity planning: A tactical approach to planning for highly scalable applications and services*. Berlin : Springer, 2007. 253 p. DOI: 10.1007/978-3-540-31010-5.
28. DORA Team. *Accelerate State of DevOps Report 2024*. Google Cloud, 2024. URL: <https://dora.dev/research/2024/dora-report/> (дата звернення: 13.07.2025).
29. DORA Team. *2025 State of AI-assisted software development report*. Google Cloud, 2025. URL: <https://dora.dev/research/2025/dora-report/> (дата звернення: 25.01.2026).
30. Blinowski G., Ojdowska A., Przybyłek A. Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation. *IEEE Access*. 2022. Vol. 10. P. 20357–20374. DOI: 10.1109/ACCESS.2022.3152803.
31. Zhong C., Huang H., Zhang H., Li S. Impacts, Causes, and Solutions of Architectural Smells in Microservices: An Industrial Investigation. *Software: Practice and Experience*. 2022. Vol. 52, No. 12. P. 2574–2597. DOI: 10.1002/spe.3138.
32. Software AG. Software AG and Vanson Bourne survey: organizations combining API management with microservices will have competitive advantage, say 93% of global IT leaders. *BusinessWire*. 2021. URL: <https://www.businesswire.com/news/home/20210511005178/en/> (дата звернення: 30.06.2025).
33. Ayas H. M., Leitner P., Hebig R. An Empirical Study of the Systemic and Technical Migration towards Microservices. *Empirical Software Engineering*. 2023. Vol. 28, No. 4. Art. 85. DOI: 10.1007/s10664-023-10308-9.
34. Wang Y., Kadiyala H., Rubin J. Promises and Challenges of Microservices: An Exploratory Study. *Empirical Software Engineering*. 2021. Vol. 26, No. 4. Art. 63. DOI: 10.1007/s10664-020-09910-y.
35. Assunção W. K., Krüger J., Mosser S., Selaoui S. How Do Microservices Evolve? An Empirical Analysis of Changes in Open-Source Microservice Repositories. *Journal of Systems and Software*. 2023. Vol. 204. Art. 111788. DOI: 10.1016/j.jss.2023.111788.

36. Tapia F., Mora M. Á., Fuertes W., Aules H., Flores E., Toulkeridis T. From monolithic systems to microservices: A comparative study of performance. *Applied Sciences*. 2020. Vol. 10, № 17. P. 5797. DOI: 10.3390/app10175797.
37. Al-Qora'n L. F., Al-Said Ahmad A. Modular Monolith Architecture in Cloud Environments: A Systematic Literature Review. *Future Internet*. 2025. Vol. 17, no. 11. P. 496. DOI: 10.3390/fi17110496.
38. Worklytics. Benchmarking 2025: What's a 'good' employee productivity score for software engineering teams? Worklytics Research, 2025. URL: <https://www.worklytics.co/resources/software-engineering-productivity-benchmarks-2025-good-scores> (дата звернення: 15.08.2025).
39. Russo D., Hanel P. H. P., Altnickel S., van Berkel N. Predictors of Well-Being and Productivity among Software Professionals during the COVID-19 Pandemic — a Longitudinal Study. *Empirical Software Engineering*. 2021. Vol. 26, No. 4. Art. 62. DOI: 10.1007/s10664-021-09945-9.
40. Šmite D., Moe N. B., Klotins E., Gonzalez-Huerta J. From Forced Working-from-Home to Voluntary Working-from-Anywhere: Two Revolutions in Telework. *Journal of Systems and Software*. 2023. Vol. 195. Art. 111509. DOI: 10.1016/j.jss.2022.111509.
41. Müller K., Koch C., Riehle D., et al. Challenges of Working from Home in Software Development during COVID-19 Lockdowns. *ACM Transactions on Software Engineering and Methodology*. 2023. Vol. 32, No. 5. Art. 111. P. 1–41. DOI: 10.1145/3579636.
42. Smite D., Christensen E. L., Tell P., Russo D. The Future Workplace: Characterizing the Spectrum of Hybrid Work Arrangements for Software Teams. *IEEE Software*. 2023. Vol. 40, No. 2. P. 34–41. DOI: 10.1109/MS.2022.3230289.
43. MacCormack A., Rusnak J., Baldwin C. Y. Exploring the structure of complex software designs: *An empirical study of open source and proprietary code*. *Management Science*. 2006. Vol. 52, № 7. P. 1015–1030. DOI: 10.1287/mnsc.1060.0552.

44. Russo D., Hanel P., Altnickel S., van Berkel N. Satisfaction and Performance of Software Developers during Enforced Work from Home in the COVID-19 Pandemic. *Empirical Software Engineering*. 2023. Vol. 28, No. 2. Art. 44. DOI: 10.1007/s10664-023-10293-z.
45. Nagappan N., Murphy B., Basili V. The influence of organizational structure on software quality: An empirical case study. *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. New York : ACM, 2008. P. 521–530. DOI: 10.1145/1368088.1368160.
46. Jadhav A., Shandilya S. K. Reliable Machine Learning Models for Estimating Effective Software Development Efforts: A Comparative Analysis. *Journal of Engineering Research (Kuwait)*. 2023. Vol. 11, No. 4. P. 362–276. DOI: 10.1016/j.jer.2023.100150.
47. Alsheikh N. M., Munassar N. M. Improving Software Effort Estimation Models Using Grey Wolf Optimization Algorithm. *IEEE Access*. 2023. Vol. 11. P. 143549–143579. DOI: 10.1109/ACCESS.2023.3340140.
48. Rankovic N., Rankovic D., Ivanovic M., Kaljevic J. Interpretable Software Estimation with Graph Neural Networks and Orthogonal Array Tunning Method. *Information Processing & Management*. 2024. Vol. 61, No. 5. Art. 103778. DOI: 10.1016/j.ipm.2024.103778.
49. Saravanos A., Curinga M. X. Simulating the software development lifecycle: The waterfall model. *Applied System Innovation*. 2023. Vol. 6, № 6. Art. 108. DOI: 10.3390/asi6060108.
50. Lunesu M. I., Tonelli R., Marchesi L., Marchesi M. Assessing the risk of software development in agile methodologies using simulation. *IEEE Access*. 2021. Vol. 9. P. 134240–134258. DOI: 10.1109/ACCESS.2021.3115941.
51. Joshi B. R. Factors affecting the software developer's performance. *Journal of Administrative and Business Studies*. 2018. Vol. 4, № 6. P. 288–296. DOI: 10.20474/jabs-4.6.3.
52. Canedo E. D., Santos G. A. Factors affecting software development productivity: An empirical study. *Proceedings of the XXXIII Brazilian Symposium on Software*

- Engineering (SBES '19)*. New York : ACM, 2019. P. 307–316. DOI: 10.1145/3350768.3352491.
53. Zhan J., Zhou X., Zhao J. Impact of software complexity on development productivity. *International Journal of Software Engineering and Knowledge Engineering*. 2012. Vol. 22, № 8. P. 1103–1122. DOI: 10.1142/S0218194012500301.
 54. Trendowicz A., Münch J. Factors influencing software development productivity – state-of-the-art and industrial experiences. *Advances in Computers*. 2009. Vol. 77. P. 185–241. DOI: 10.1016/S0065-2458(09)01206-6.
 55. Koppel J., Jackson D. Demystifying dependence. *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. New York : ACM, 2020. P. 48–64. DOI: 10.1145/3426428.3426916.
 56. Panichella S., Rahman M., Taibi D. Structural Coupling for Microservices. *Proceedings of the 11th International Conference on Cloud Computing and Services Science*. 2021. Vol. 1. P. 280–287. DOI: 10.5220/0010481902800287.
 57. Apolinário D. R., de França B. B. A method for monitoring the coupling evolution of microservice-based architectures. *Journal of the Brazilian Computer Society*. 2021. Vol. 27. Article 17. DOI: 10.1186/s13173-021-00120-y.
 58. Greiler M., Herzig K., Czerwonka J. Code ownership and software quality: A replication study. *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. Piscataway : IEEE, 2015. P. 2–12. DOI: 10.1109/MSR.2015.8.
 59. Thongtanunam P., McIntosh S., Hassan A. E., Iida H. Revisiting code ownership and its relationship with software quality in the scope of modern code review. *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. New York : ACM, 2016. P. 1039–1050. DOI: 10.1145/2884781.2884852.
 60. Faragó C., Hegedűs P., Ferenc R. Code ownership: Impact on maintainability. *Computational Science and Its Applications – ICCSA 2015. Lecture Notes in Computer Science*. Vol. 9159. Cham : Springer, 2015. P. 3–19. DOI: 10.1007/978-3-319-21413-9_1.

61. Sedano T., Ralph P., Péraire C. Practice and perception of team code ownership. *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering (EASE '16)*. New York : ACM, 2016. Article 36. P. 1–6. DOI: 10.1145/2915970.2916002.
62. Badampudi D., Unterkalmsteiner M., Britto R. Modern code reviews – a survey of literature and practice. *ACM Transactions on Software Engineering and Methodology*. 2023. Vol. 32, № 4. P. 1–41. DOI: 10.1145/3585004.
63. Yang L., Liu B., Jia J. et al. Prioritizing code review requests to improve review efficiency: a simulation study. *Empirical Software Engineering*. 2024. Vol. 30, № 1. DOI: 10.1007/s10664-024-10575-0.
64. Rigby P. C. et al. Improving code reviewer recommendation: Accuracy, latency, workload, and bystanders. *ACM Transactions on Software Engineering and Methodology*. 2025. Vol. 35, № 1. Article 24. DOI: 10.1145/3736405.
65. Pandya P., Tiwari S. CORMS: a GitHub and Gerrit based hybrid code reviewer recommendation approach for modern code review. *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22)*. New York : ACM, 2022. P. 546–557. DOI: 10.1145/3540250.3549115.
66. Kudrjavets G. et al. Mining code review data to understand waiting times between acceptance and merging. *Proceedings of the 19th International Conference on Mining Software Repositories (MSR '22)*. New York : ACM, 2022. DOI: 10.1145/3524842.3528432.
67. Fregnan E., Petrulio F., Bacchelli A. The evolution of the code during review: an investigation on review changes. *Empirical Software Engineering*. 2022. Vol. 27, № 7. P. 579–590. DOI: 10.1007/s10664-022-10205-7.
68. Chouchen M. et al. Learning to predict code review completion time in modern code review. *Empirical Software Engineering*. 2023. Vol. 28, № 4. DOI: 10.1007/s10664-023-10300-3.
69. Mykhailo Kotenko, Dmytro Moskalyk, Valeriia Kovach, Viacheslav Osadchyi. Navigating the challenges and best practices in securing microservices

- architecture. *Proceedings of the Cybersecurity Providing in Information and Telecommunication Systems II*, Kyiv, Ukraine, October 26, 2024 (online). Edited by Volodymyr Sokolov, Vasyl Ustimenko, Tamara Radivilova, Mariya Nazarkevych. CEUR Workshop Proceedings, Vol. 3826, 2024. PP.1–16. URL: <https://ceur-ws.org/Vol-3826/paper1.pdf>
70. Москалик Д. О., Антонюк Д. С. Вплив архітектури програмної системи на продуктивність розробки програмного забезпечення. *Інформаційні технології в освіті та науці* : зб. наук. пр. III Міжнар. наук.-практ. конф., м. Мелітополь, 25–26 трав. 2023 р. Мелітополь : МДПУ ім. Б. Хмельницького, 2023. С. 491–495.
 71. Москалик Д. О., Антонюк Д. С. Вплив різновидів практики власності коду на процес розробки програмного забезпечення. *Комп'ютерні технології: інновації, проблеми, рішення* : тези доп. V Всеукр. наук.-техн. конф., м. Житомир, 01–02 груд. 2022 р. Житомир : Держ. ун-т «Житомирська політехніка», 2022. С. 32–33.
 72. Ярош І. В., Павловський Є. В., Назарова І. А. Математичне моделювання процесу прогнозування витрат часу на розв'язання типової задачі з розробки програмного забезпечення. *Наукові праці ДонНТУ. Серія: Інформатика, кібернетика та обчислювальна техніка*. 2023. № 35–36. С. 79–84.
 73. Chiang A. J., Jeang A. Stochastic project management via computer simulation and optimisation method. *International Journal of Systems Science: Operations & Logistics*. 2015. Vol. 2, № 4. P. 211–230. DOI: 10.1080/23302674.2015.1025889.
 74. Bocharov P. O., Schwarz T. Evaluation of mean and variance approximations in three point estimation of task completion times using the beta and the Kumaraswamy distribution. *International Journal of Information Technology and Management*. 2019. Vol. 18, № 4. P. 389–406. DOI: 10.1504/ijitm.2019.103053.
 75. Diamantopoulos T., Nastos D.-N., Symeonidis A. Apache Jira Issue Tracking Dataset. Zenodo. 2023. DOI: 10.5281/zenodo.7740379.

76. Kline D. E., Bender D. A. Maximum likelihood estimation for shifted Weibull and lognormal distributions. *Transactions of the ASAE*. 1990. Vol. 33, № 1. P. 330–335. DOI: 10.13031/2013.31334.
77. Knop R. Remark on algorithm 334 [G5]: normal random deviates. *Communications of the ACM*. 1969. Vol. 12, № 5. P. 281. DOI: 10.1145/362946.362996.
78. Giner G., Smyth G. K. Statmod: Probability calculations for the inverse Gaussian distribution. *The R Journal*. 2016. Vol. 8, № 1. P. 339–351. DOI: 10.32614/rj-2016-024.
79. Folks J. L., Chhikara R. S. The inverse Gaussian distribution and its statistical application – a review. *Journal of the Royal Statistical Society: Series B (Methodological)*. 1978. Vol. 40, № 3. P. 263–275. DOI: 10.1111/j.2517-6161.1978.tb01039.x.
80. Michael J. R., Schucany W. R., Haas R. W. Generating random variates using transformations with multiple roots. *The American Statistician*. 1976. Vol. 30, № 2. P. 88–90. DOI: 10.1080/00031305.1976.10479147.
81. Vural H., Koyuncu M. Does domain-driven design lead to finding the optimal modularity of a microservice? *IEEE Access*. 2021. Vol. 9. P. 32721–32733. DOI: 10.1109/ACCESS.2021.3060895.
82. Rolfsnes T. et al. Generalizing the analysis of evolutionary coupling for software change impact analysis. *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Piscataway : IEEE, 2016. P. 201–212. DOI: 10.1109/SANER.2016.101.
83. Mondal M. et al. Historank: History-based ranking of co-change candidates. *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Piscataway : IEEE, 2020. P. 240–250. DOI: 10.1109/SANER48275.2020.9054869.
84. Шматковська Т., Коробчук Т. Сучасні інформаційні та комунікаційні технології в моделюванні бізнес-процесів. *Економічний форум*. 2023. № 1(3). С. 156–161. DOI: 10.36910/6775-2308-8559-2023-3-20.

85. Кордунова Ю. С., Фелтіновскі М., Придатко О. В., Смотр О. О. Математичне моделювання процесу розробки спеціалізованих програмних систем безпеко-орієнтованого спрямування. *Вісник Львівського державного університету безпеки життєдіяльності*. 2023. № 27. С. 23–31. DOI: 10.32447/20784643.27.2023.03.
86. Приходько С. Б., Приходько Н. В., Книрик К. О. Математичне моделювання трудомісткості розробки мобільних застосунків у фазі планування із врахуванням викидів. *Комп'ютерне моделювання та оптимізація складних систем* : матеріали V Міжнар. наук.-техн. конф., м. Дніпро, 6–8 листоп. 2019 р. Дніпро : УДХТУ, 2019. С. 50.
87. Li Y., Zhang H., Dong L., Liu B., Ma J. Constructing a hybrid software process simulation model in practice: An exemplar from industry. *Proceedings of the International Conference on Software and System Processes*. New York : ACM, 2020. P. 135–144. DOI: 10.1145/3379177.3388906.
88. Helal M., Rabelo L., Sepúlveda J., Jones A. A methodology for integrating and synchronizing the system dynamics and discrete event simulation paradigms. *Proceedings of the 25th International Conference of the System Dynamics Society*. Vol. 3, № 3. Albany : System Dynamics Society, 2007. P. 1–24.
89. Огінський Є., Антонюк Д., Вакалюк Т., Москалик Д., Василенко В. Аналіз алгоритмічного і математичного апарату для систем побудови та аналізу інструментів управління персональними фінансами. *Інформаційні технології та суспільство*. 2022. № 3 (5). С. 29–40. DOI: 10.32689/maup.it.2022.3.4.
90. Multi-Agent Research and Simulation Framework: документація. URL: <https://www.mars-group.org/docs/tutorial/intro> (дата звернення: 10.12.2025).
91. MathNet.Numerics: документація. URL: <https://numerics.mathdotnet.com> (дата звернення: 10.12.2025).
92. Olewicki D., Habchi S., Adams B. An empirical study on code review activity prediction and its impact in practice. *Proceedings of the ACM on Software Engineering*. 2024. Vol. 1, FSE. P. 2238–2260. DOI: 10.1145/3660806.

93. Dong T. The impact of PR size on code review quality: What data tells us. URL: <https://www.propelcode.ai/blog/pr-size-impact-code-review-quality-data-study> (дата звернення: 12.12.2025).
94. Maginnis B. What the data tells us about teams' code review practices in the real world. URL: <https://www.sourcery.ai/blog/ideals-vs-reality/> (дата звернення: 17.10.2025).
95. Москалик Д. О., Антонюк Д. С. Моделювання зв'язності шляхом імовірнісного розподілу частот внесення змін до програмних модулів при симуляції процесу розробки програмного забезпечення. *Вісник Херсонського національного технічного університету*. 2025. Т. 2, № 2 (93). С. 238–245. DOI: 10.35546/kntu2078-4481.2025.2.2.28.
96. Москалик Д. О., Антонюк Д. С. Імовірнісний розподіл складності вхідних задач для дискретно-подійного та агентно-орієнтованого моделювання процесу розробки програмного забезпечення. *Вчені записки Таврійського національного університету імені В. І. Вернадського. Серія: Технічні науки*. 2024. Т. 35 (74), № 3, ч. 1. С. 152–159. DOI: 10.32782/2663-5941/2024.3.1/23.
97. Москалик Д. О., Антонюк Д. С. Моделювання процесу рецензування коду в агентно-орієнтованих симуляціях розробки модульного програмного забезпечення. *Актуальні проблеми автоматизації та інформаційних технологій*. 2025. Т. 29. С. 306-317. DOI: 10.15421/432528.
98. Москалик Д. О., Антонюк Д. С. Аналіз розподілу складності задач при розробці програмного забезпечення з відкритим вихідним кодом. *Сучасні комп'ютерні системи та мережі в управлінні* : матеріали VI Всеукр. наук.-практ. інтернет-конф. здобувачів вищої освіти та молодих вчених, м. Хмельницький, м. Херсон, 30 листоп. 2023 р. Херсон : ХНТУ, 2023. С. 31–32.
99. Москалик Д. О., Антонюк Д. С. Підхід до моделювання процесу розробки програмного забезпечення для дослідження впливу архітектури на продуктивність розробки. *Тенденції розвитку технологій в автоматизації, електроенергетиці, приладобудуванні та робототехніці* : матеріали

- Всеукр. наук.-практ. on-line конф. здобувачів вищої освіти і молодих учених, присвяч. Дню науки, м. Житомир, 13 трав. 2024 р. Житомир : Держ. ун-т «Житомирська політехніка», 2024. С. 81.
100. Москалик Д. О., Антонюк Д. С. Моделювання зв'язності вихідного коду в симуляціях процесу розробки програмного забезпечення. *Сучасні інформаційні технології та системи в управлінні* : зб. матеріалів VI Міжнар. наук.-практ. конф. молодих вчених, аспірантів і студентів, м. Київ, 10–11 квіт. 2025 р. Київ : КНЕУ, 2025. С. 314–315.
101. Москалик Д. Агентно-орієнтовані симуляції процесу розробки програмного забезпечення для різних типів архітектури: набір даних. Версія 1.0.0. Zenodo, 2026. DOI: 10.5281/zenodo.19206776.
102. scikit-learn: machine learning in Python. scikit-learn. URL: <https://scikit-learn.org/> (дата звернення: 18.12.2025).

Додатки

Додаток А

Повний перелік подій, що генеруються програмним комплексом
симуляції процесу розробки програмного забезпечення

Таблиця А.1.

Події, що генеруються в процесі симуляції

Подія	Джерело	Опис
DevRested	Developer	Розробник завершив перерву
StartedNewDevelopmentTask	Developer	Розробник взяв нове завдання з черги та розпочав розробку
ContinueDevelopmentOnTask	Developer	Розробник продовжив роботу над завданням, яке вже було в процесі
SwitchedBackToPostponedDevTask	Developer	Розробник повернувся до завдання, яке було тимчасово відкладено
DevFinished	Developer	Розробник завершив розробку завдання
StartedRework	Developer	Розробник розпочав доопрацювання після відхилення на запиті на злиття
ReworkFinished	Developer	Розробник завершив доопрацювання за результатами рецензування коду
StartedReview	Developer	Розробник розпочав рецензування коду
ReviewFinished	Developer	Розробник завершив поточний етап рецензування коду
StartedBugfixing	Developer	Розробник розпочав виправлення помилки, знайденої під час тестування
BugfixingFinished	Developer	Розробник завершив виправлення помилки, про яку повідомив тестувальник
StartedFixingFailedDeployment	Developer	Розробник розпочав виправлення коду, який привів до невдалого розгортання
DeploymentFixingFinished	Developer	Розробник завершив усунення причин невдалого розгортання

TaskSentToTesting	Developer	Задача пройшла рецензування коду та передана до черги тестування
TaskSentToDeployment	Developer	Завдання пройшло тестування та передане до конвеєру розгортання
TaskCompleted	Developer	Завдання успішно пройшло розгортання та вважається повністю завершеним
TesterRested	Tester	Тестувальник завершив перерву
TestingStarted	Tester	Тестувальник взяв завдання та розпочав нову ітерацію тестування
TestingFinished	Tester	Тестувальник завершив ітерацію тестування та зафіксував результат
ReviewStatusChanged	Code Review	Усі рецензенти подали свої вердикти та визначено підсумок ітерації рецензування коду
EpicCreated	Backlog	Згенеровано нову узагальнену користувацьку історію та додано її складові до черги завдань
NewBug	Backlog	Знайдена нова помилка у вже розгорнутій версії системи
EpicFinished	Backlog	Узагальнену користувацьку історію повністю завершено
DeploymentFailed	Deployment	Спроба розгортання завершилася невдачею, завдання повернуто розробнику для виправлення
DeploymentSucceeded	Deployment	Спроба розгортання завершилася успішно

Вихідний код симуляційного програмного комплексу

Program.cs

```
using System.Diagnostics;
using Mars.Common.Core.Random;
using Mars.Components.Starter;
using Mars.Core.Simulation;
using MarsSim.Layers;
using MarsSim.Utilities;

var outputRoot = Environment.GetEnvironmentVariable("SIM_OUTPUT_ROOT") ?? "runs";
var runId = Environment.GetEnvironmentVariable("SIM_OUTPUT_RUNID") ??
    $"{DateTime.Now:yy-MM-dd-HH-mm-ss}";

var durationInDays = 60;
var avgOfN = 10;
var seeds = NewSeeds(avgOfN);
// seeds =
[442349467, 787221927, 256991735, 1605299526, 1082074708, 2069385717, 1635796219, 977294177, 35
9361641, 625675846];
string[] projectNames = [
    "Microservices_50_500k",
    "ModularMonolith_Strong_50_500k",
    "ModularMonolith_Weak_50_500k",
    "Monolith_500k",
];
string[] teamTopologyNames = [
    "5x4-2_UTC",
    "5x4-2_2TZ",
    "5x4-2_TeamPerTZ",
];
string[] variationNames = [
    "ModuleCount",
    "ModuleSize",
    "LOC",
    "TeamsCount",
    "TeamsSize",
    "WIP",
    "ModuleVsTeamCount",
    "TeamsRatio",
];

foreach (var arg in args)
{
    if (arg.StartsWith("--projects=", StringComparison.Ordinal))
        projectNames = arg[11..].Split(',', StringSplitOptions.RemoveEmptyEntries);
    else if (arg.StartsWith("--teams=", StringComparison.Ordinal))
        teamTopologyNames = arg[8..].Split(',', StringSplitOptions.RemoveEmptyEntries);
    else if (arg.StartsWith("--variation=", StringComparison.Ordinal))
        variationNames = arg[12..].Split(',', StringSplitOptions.RemoveEmptyEntries);
    else if (arg.StartsWith("--duration=", StringComparison.Ordinal))
        durationInDays = Convert.ToInt32(arg[11..]);
    else if (arg.StartsWith("--avg-of=", StringComparison.Ordinal))
        avgOfN = Convert.ToInt32(arg[9..]);
    else

```

```

    {
        Console.Error.WriteLine("Unknown argument: " + arg);
        return;
    }
}

var scenarios = Scenario.Load(projectNames, teamTopologyNames).ToArray();
if (scenarios.Length == 0)
{
    Console.Error.WriteLine("No scenarios specified.");
    return;
}

var variations = Variation.Load(variationNames);
scenarios = Variation.Expand(scenarios, variations);

var runOutputRoot = Path.Combine(outputRoot, runId);
var results = new List<RunResults>(scenarios.Length);
Console.WriteLine($"Using seeds: {string.Join(", ", seeds)}");
var globalStart = Stopwatch.GetTimestamp();
for (var scenarioIndex = 0; scenarioIndex < scenarios.Length; scenarioIndex++)
{
    results[scenarioIndex] = new List<RunResults>(avgOfN);
    var scenario = scenarios[scenarioIndex];

    Console.WriteLine($"=== Scenario[{(scenarioIndex + 1):00}/{scenarios.Length}]:
{scenario.Name} ===");
    scenario.WriteTo(Path.Combine(runOutputRoot, scenario.VariationName,
scenario.Name));

    for (var i = 1; i <= avgOfN; i++)
    {
        Generator.Seed = seeds[i - 1];
        Console.WriteLine($"Running iteration {i} with seed {Generator.Seed}");

        var outputPath = Path.Combine(runOutputRoot, scenario.VariationName,
scenario.Name, $"{i:0000}");
        Directory.CreateDirectory(outputPath);

        var startMoment = Stopwatch.GetTimestamp();
        var r = RunScenario(scenario, durationInDays, outputPath);
        results[scenarioIndex].Add(r);
        Console.WriteLine($"Finished iteration {i} in
{Stopwatch.GetElapsedTime(startMoment)}");
    }
    Console.WriteLine();
    var globalElapsed = Stopwatch.GetElapsedTime(globalStart);
    var globalRemaining = globalElapsed * (scenarios.Length - scenarioIndex - 1) /
(scenarioIndex + 1);
    Console.WriteLine($"Running for {globalElapsed}; remaining {globalRemaining};
estimated completion at {DateTime.Now + globalRemaining}");
    Console.WriteLine();
}

var summaryPath = Path.Combine(runOutputRoot, "summary.txt");
using (var summaryFile = File.AppendText(summaryPath))
{
    for (var i = 0; i < scenarios.Length; i++)
    {
        var scenario = scenarios[i];
        summaryFile.WriteLine($"=== Scenario: {scenario.Name} ===");
    }
}

```

```

        RunResults.Avg(results[i]).Print(summaryFile);
    }
}
Console.WriteLine();
Console.WriteLine("=====");
Console.WriteLine("===== SUMMARY =====");
Console.WriteLine("=====");
Console.WriteLine(File.ReadAllText(summaryPath));

static RunResults RunScenario(Scenario scenario, int durationInDays, string?
outputPath)
{
    var builder = new ModelBuilder(
        durationInDays: durationInDays,
        outputPath: outputPath);

    var model = builder.CreateModel();
    var config = builder.CreateConfig(scenario);
    using var app = SimulationStarter.BuildApplication(model, config);
    var simulation = app.Resolve<ISimulation>();

    var watch = Stopwatch.StartNew();
    var state = simulation.StartSimulation();
    var backlog = state.Model.Layers.Values.OfType<Backlog>().First();
    watch.Stop();

    return new RunResults(
        state.Iterations, watch.Elapsed,
        backlog.StartedTaskCount, backlog.CompletedTaskCount,
        backlog.StartedEpicCount, backlog.CompletedEpicCount);
}

static int[] NewSeeds(int count)
{
    var seeds = new int[count];
    var rnd = Random.Shared;
    var seed = 0;
    for (var i = 0; i < count; i++)
    {
        while (seed == 0 || seeds.Contains(seed))
            seed = rnd.Next();
        seeds[i] = seed;
    }
    return seeds;
}

sealed record RunResults(
    long Iterations,
    TimeSpan Elapsed,
    long StartedTaskCount,
    long CompletedTaskCount,
    long StartedEpicCount,
    long CompletedEpicCount)
{
    public void Print(TextWriter output)
    {
        output.WriteLine($"Executed iterations {Iterations} lasted {Elapsed}");
        output.WriteLine($"Tasks: started={StartedTaskCount},
finished={CompletedTaskCount}");
        output.WriteLine($"Epics: started={StartedEpicCount},
finished={CompletedEpicCount}");
    }
}

```

```

        output.WriteLine();
    }

    public static RunResults Avg(List<RunResults> collection)
    {
        long it = 0, ts = 0, tc = 0, es = 0, ec = 0;
        var e = TimeSpan.Zero;
        foreach (var item in collection)
        {
            e += item.Elapsed;
            it += item.Iterations;
            ts += item.StartedTaskCount;
            tc += item.CompletedTaskCount;
            es += item.StartedEpicCount;
            ec += item.CompletedEpicCount;
        }
        return new RunResults(
            it / collection.Count,
            e / collection.Count,
            ts / collection.Count,
            tc / collection.Count,
            es / collection.Count,
            ec / collection.Count);
    }
}

```

Layers/Telemetry.cs

```

using System.Threading.Tasks.Dataflow;
using Mars.Components.Layers;
using Mars.Interfaces.Annotations;
using Mars.Interfaces.Data;
using Mars.Interfaces.Layers;
using MarsSim.Entities;

namespace MarsSim.Layers;

public sealed class Telemetry : AbstractLayer, IDisposable
{
    ActionBlock<string>? queue;
    StreamWriter? outputWriter;

    [PropertyDescription]
    public string? OutputPath { get; set; }

    public override bool InitLayer(LayerInitData layerInitData, RegisterAgent?
registerAgentHandle = null,
    UnregisterAgent? unregisterAgent = null)
    {
        base.InitLayer(layerInitData, registerAgentHandle, unregisterAgent);

        if (OutputPath is null)
            return true;

        if (!Directory.Exists(OutputPath))
            Directory.CreateDirectory(OutputPath);

        outputWriter = new StreamWriter(Path.Combine(OutputPath, "Events.csv"));
    }
}

```

```

outputWriter.WriteLine($"Tick,Type,Actor,Task,Epic,Iteration,Spent,Outcome,AddedEffort,
ModuleCount,LocPercent");
    queue ??= new ActionBlock<string>(outputWriter.WriteLine, new
ExecutionDataflowBlockOptions
    {
        MaxDegreeOfParallelism = 1,
        EnsureOrdered = true,
    });

    return true;
}

public void Track(string eventType,
    Guid? actor = null,
    BacklogTask? task = null,
    Guid? epic = null,
    int? iteration = null,
    int? spent = null,
    int? moduleCount = null,
    double? locPercent = null,
    string? outcome = null,
    int? addedEffort = null)
{
    if (queue is null)
        return;

    epic ??= task?.Epic?.ID;

    queue.Post($"{GetCurrentTick()}, {eventType}, {actor}, {task?.ID}, {epic}, {iteration}, {spen
t}, {outcome}, {addedEffort}, {moduleCount}, {locPercent}");
}

public void Dispose()
{
    if (queue is not null)
    {
        queue.Complete();
        queue.Completion.Wait();
        queue = null;
    }

    outputWriter?.Dispose();
    outputWriter = null;
}
}

```

Layers/Backlog.cs

```

using Mars.Common.Core.Random;
using Mars.Components.Layers;
using Mars.Interfaces.Annotations;
using MarsSim.Agents;
using MarsSim.Entities;
using MarsSim.Utilities;

namespace MarsSim.Layers;

public sealed class Backlog : AbstractLayer
{
    const double PostReleaseBugProbability = 0.1;

```

```

const double PostReleaseBugAfterTaskCount = 1.0 / PostReleaseBugProbability;

[PropertyDescription]
public SoftwareProduct Product { get; set; } = null!;

Telemetry Telemetry => Product.Telemetry;

readonly EffortGenerator effortGenerator = new(Generator.Seed!.Value, maxMinutes:
960 /* 2 days */);
readonly List<BacklogTask> backlog = [];

int startedTasks;
int completedTasks;
int startedEpics;
int completedEpics;

public int StartedTaskCount => startedTasks;
public int CompletedTaskCount => completedTasks;
public int StartedEpicCount => startedEpics;
public int CompletedEpicCount => completedEpics;

public BacklogTask GetNextTask(Developer developer)
{
    Interlocked.Increment(ref startedTasks);
    var task = GetNextUnassignedTask(developer.Team);
    task.AssignedTo = developer;
    return task;
}

public void CompleteTask(BacklogTask task)
{
    var newBugEveryNTasks = Product.Codebase.OwnershipType is
CodeOwnershipType.Collective
        ? PostReleaseBugAfterTaskCount / 2
        : PostReleaseBugAfterTaskCount;

    if (Interlocked.Increment(ref completedTasks) % newBugEveryNTasks is 0)
    {
        lock (backlog)
        {
            var bug = new BacklogTask
            {
                ID = Guid.NewGuid(),
                Epic = task.Epic,
                Type = BacklogTaskType.Bug,
                Effort = effortGenerator.Next(),
                AffectedModules = task.AffectedModules,
                OwningTeam = task.OwningTeam,
            };
            backlog.Add(bug);
            Telemetry.Track("NewBug", task: bug, addedEffort: bug.Effort);
        }
    }

    if (task.Epic is null)
        return;

    lock (task.Epic)
    {
        if (task.Epic.Tasks.Contains(task) && task.Epic.Tasks.All(static t =>

```

```

        t.DeploymentRequest?.LastIteration.Outcome is
TaskValidationOutcome.Approved))
    {
        Interlocked.Increment(ref completedEpics);
        Telemetry.Track("EpicFinished", epic: task.Epic.ID);
    }
}

BacklogTask GetNextUnassignedTask(Team team)
{
    lock (backlog)
    {
        var i = 0;
        while (true)
        {
            for (; i < backlog.Count; i++)
            {
                var task = backlog[i];
                if (task.OwningTeam is not null && task.OwningTeam != team)
                    continue;

                if (task.AssignedTo is null)
                {
                    backlog.RemoveAt(i);
                    return task;
                }
            }
            AddNewEpic();
            Interlocked.Increment(ref startedEpics);
        }
    }
}

void AddNewEpic()
{
    var epic = new BacklogEpic { ID = Guid.NewGuid() };
    var taskCount = Generator.Random.Next(2, 10);
    if (Product.Codebase.Modules.Count is 1)
    {
        for (var i = 0; i < taskCount; i++)
        {
            epic.Tasks.Add(CreateFeatureTask(epic, effortModifier: 1.0,
Product.Codebase.Modules, owningTeam: null));
        }
    }
    else
    {
        var affectedModules = Product.Codebase.PickChangedModules();
        if (Product.Codebase.OwnershipType is CodeOwnershipType.Strong)
        {
            var effortModifier = Product.Codebase.Architecture is
ArchitectureType.Microservices
                ? 1.0 : 0.8;
            var groups = GroupModulesByTeam(affectedModules);
            while (epic.Tasks.Count < taskCount)
            {
                foreach (var group in groups.OrderByDescending(static g =>
g.Value.Count))
                {

```

```

        epic.Tasks.Add(CreateFeatureTask(epic, effortModifier,
group.Value, group.Key));
        if (epic.Tasks.Count == taskCount)
            break;
    }
}
else
{
    for (var i = 0; i < taskCount; i++)
    {
        epic.Tasks.Add(CreateFeatureTask(epic, effortModifier: 0.9,
affectedModules, owningTeam: null));
    }
}
}
backlog.AddRange(epic.Tasks);
Telemetry.Track("EpicCreated", epic: epic.ID);
}

BacklogTask CreateFeatureTask(
    BacklogEpic epic,
    double effortModifier,
    List<CodebaseModule> modules,
    Team? owningTeam) => new()
{
    ID = Guid.NewGuid(),
    Epic = epic,
    Type = BacklogTaskType.Feature,
    Effort = (int)Math.Ceiling(effortGenerator.Next() * effortModifier),
    AffectedModules = modules,
    OwningTeam = owningTeam,
};

static Dictionary<Team, List<CodebaseModule>>
GroupModulesByTeam(List<CodebaseModule> codebaseModules)
{
    var dic = new Dictionary<Team, List<CodebaseModule>>();
    foreach (var module in codebaseModules)
    {
        if (!dic.TryGetValue(module.OwningTeam!, out var list))
            dic.Add(module.OwningTeam!, list = []);

        list.Add(module);
    }
    return dic;
}
}

```

Layers/CodeReview.cs

```

using System.Diagnostics.CodeAnalysis;
using Mars.Common.Core.Random;
using Mars.Components.Layers;
using Mars.Interfaces.Annotations;
using MarsSim.Agents;
using MarsSim.Entities;
using MarsSim.Utilities;

namespace MarsSim.Layers;

```

```

public sealed class CodeReview : AbstractLayer
{
    const double ProbabilityOfInitialRework = 0.6;
    const double ProbabilityOfReworkDecrement = 0.2;
    const int DefaultMinApprovals = 2;

    [PropertyDescription]
    public Telemetry Telemetry { get; set; } = null!;

    readonly List<BacklogTask> underReview = [];

    public void Send(BacklogTask task)
    {
        var request = CodeReviewRequest.Create(task);
        lock (underReview)
        {
            task.CodeReviewRequest = request;
            underReview.Add(task);
        }
    }

    public void Return(BacklogTask task)
    {
        lock (underReview)
        {
            var newIteration = new CodeReviewIteration
            {
                Request = task.CodeReviewRequest!,
            };
            foreach (var lastArtifact in
task.CodeReviewRequest!.LastIteration.Artifacts)
            {
                CodeReviewArtifact? newArtifact = null;
                foreach (var lastAssignment in lastArtifact.Assignments)
                {
                    if (lastAssignment.Status is not
CodeReviewAssignmentStatus.ReworkNeeded)
                        continue;

                    newArtifact ??= new CodeReviewArtifact
                    {
                        Iteration = newIteration,
                        Module = lastArtifact.Module,
                        ReviewEffort =
(int)Math.Ceiling(((double)lastArtifact.ReviewEffort /
task.CodeReviewRequest.Iterations.Count),
                    );
                    newArtifact.Assignments.Add(new CodeReviewAssignment
                    {
                        Artifact = newArtifact,
                        Developer = lastAssignment.Developer,
                    });
                }
                if (newArtifact is not null)
                    newIteration.Artifacts.Add(newArtifact);
            }
            task.CodeReviewRequest.Iterations.Add(newIteration);
        }
    }

    List<CodeReviewAssignment> assignmentsForReview = new();

```

```

    public bool TryGetNextReviewTask(Developer developer, [MaybeNullWhen(false)] out
CodeReviewAssignment[] newAssignments)
    {
        lock (underReview)
        {
            foreach (var task in underReview)
            {
                var lastIteration = task.CodeReviewRequest!.LastIteration;
                if (lastIteration.Outcome is not TaskValidationOutcome.Unknown)
                    continue;

                assignmentsForReview.Clear();
                foreach (var ar in lastIteration.Artifacts)
                {
                    var startedCount = ar.Assignments.Count(a => a.Status is not
CodeReviewAssignmentStatus.NotStarted);
                    if (startedCount >= Math.Min(DefaultMinApprovals,
ar.Assignments.Count))
                        continue;

                    foreach (var a in ar.Assignments)
                    {
                        if (a.Developer == developer && a.Status is
CodeReviewAssignmentStatus.NotStarted)
                            assignmentsForReview.Add(a);
                    }

                    if (assignmentsForReview.Count > 0)
                    {
                        newAssignments = assignmentsForReview.ToArray();
                        foreach (var a in newAssignments)
                            a.Status = CodeReviewAssignmentStatus.Started;
                        return true;
                    }
                }
            }
        }

        newAssignments = null;
        return false;
    }

    public void FinishReview(CodeReviewAssignment assignment)
    {
        lock (underReview)
        {
            assignment.Status = ShouldApprove(assignment)
                ? CodeReviewAssignmentStatus.Approved
                : CodeReviewAssignmentStatus.ReworkNeeded;

            UpdateIterationOutcome(assignment.Artifact.Iteration);
        }

        if (assignment.Artifact.Iteration.Outcome is not TaskValidationOutcome.Unknown)
        {
            Telemetry.Track("ReviewStatusChanged",
                task: assignment.Artifact.Iteration.Request.SourceTask,
                iteration: assignment.Artifact.Iteration.Request.Iterations.Count,
                outcome: assignment.Artifact.Iteration.Outcome.ToString(),
                addedEffort: assignment.Artifact.Iteration.ReworkEffort
            );
        }
    }

```

```

    });
}

void UpdateIterationOutcome(CodeReviewIteration iteration)
{
    var reworkNeeded = false;
    foreach (var artifact in iteration.Artifacts)
    {
        var approves = 0;
        var rejections = 0;
        foreach (var assignment in artifact.Assignments)
        {
            if (assignment.Status is CodeReviewAssignmentStatus.Started)
                return;

            if (assignment.Status is CodeReviewAssignmentStatus.ReworkNeeded)
                rejections++;
            else if (assignment.Status is CodeReviewAssignmentStatus.Approved)
                approves++;
        }

        if (approves + rejections < Math.Min(DefaultMinApprovals,
artifact.Assignments.Count))
            return;

        if (rejections > 0)
            reworkNeeded = true;
    }

    if (reworkNeeded)
    {
        iteration.Outcome = TaskValidationOutcome.ReworkNeeded;
        var request = iteration.Request;
        iteration.ReworkEffort = ReworkDistribution.Sample(
            request.SourceTask.Effort,
            request.Iterations.Count);

        var developerTeam = request.SourceTask.AssignedTo!.Team;
        var ownModulesRatio = iteration.Artifacts.Count(a => a.Module.OwningTeam ==
developerTeam)
                                / (double)iteration.Artifacts.Count;
        iteration.ReworkEffort -= (int)(0.2 * ownModulesRatio *
iteration.ReworkEffort);
    }
    else
    {
        iteration.Outcome = TaskValidationOutcome.Approved;
        underReview.Remove(iteration.Request.SourceTask);
    }
}

static bool ShouldApprove(CodeReviewAssignment assignment)
{
    var iteration = assignment.Artifact.Iteration;
    var decisionsCount = iteration.Artifacts.Count * 2;

    var probabilityOfOverallRework = Math.Max(0,
        ProbabilityOfInitialRework - (iteration.Request.Iterations.Count - 1) *
ProbabilityOfReworkDecrement);

```

```

        var probabilityOfModuleRework = 1.0 - Math.Pow(1.0 -
probabilityOfOverallRework, 1d / decisionsCount);

        var developer = assignment.Artifact.Iteration.Request.SourceTask.AssignedTo!;
        var reviewer = assignment.Developer;
        var owningTeam = assignment.Artifact.Module.OwningTeam;
        if (reviewer.Team != owningTeam || developer.Team == owningTeam)
            probabilityOfModuleRework *= 0.8;

        var requiresRework = Generator.Random.NextDouble() <=
probabilityOfModuleRework;
        return !requiresRework;
    }
}

```

Layers/Deployment.cs

```

using Mars.Common.Core.Random;
using Mars.Components.Layers;
using Mars.Interfaces.Annotations;
using MarsSim.Entities;
using MarsSim.Utilities;

namespace MarsSim.Layers;

public sealed class Deployment : AbstractActiveLayer
{
    const double EffortPerLineOfCode = 0.003d;
    const double ProbabilityOfFailure = 0.1;

    readonly List<BacklogTask> pendingTasks = [];

    [PropertyDescription]
    public Telemetry Telemetry { get; set; } = null!;

    public void Send(BacklogTask task)
    {
        var codebase = task.AffectedModules[0].Codebase;
        var linesOfCodeToDeploy = codebase.Architecture is ArchitectureType.Monolith
            ? codebase.LinesOfCode
            : codebase.LinesOfCode * task.AffectedModules.Max(static m =>
m.SizePercent);

        var request = new DeploymentRequest
        {
            SourceTask = task,
            DeploymentEffort = (int)Math.Ceiling(EffortPerLineOfCode *
linesOfCodeToDeploy),
        };
        var iteration = new DeploymentIteration
        {
            Request = request,
        };
        request.Iterations.Add(iteration);

        lock (pendingTasks)
        {
            task.DeploymentRequest = request;
            pendingTasks.Add(task);
        }
    }
}

```



```

using MarsSim.Agents;
using MarsSim.Entities;
using MarsSim.Utilities;

namespace MarsSim.Layers;

public sealed class Testing : AbstractLayer
{
    const double TestingEffortPercentage = 0.3;
    const double ProbabilityOfInitialRework = 0.1;
    const double ProbabilityOfReworkDecrement = 0.05;

    readonly List<BacklogTask> pendingTasks = [];

    public void Send(BacklogTask task)
    {
        var request = new TestingRequest
        {
            SourceTask = task,
        };
        var iteration = new TestingIteration
        {
            Request = request,
            TestingEffort = CalcInitialTestingEffort(task),
        };
        request.AddIteration(iteration);
        task.TestingRequest = request;

        lock (pendingTasks)
            pendingTasks.Add(task);
    }

    public void Return(BacklogTask task)
    {
        var lastIteration = task.TestingRequest!.LastIteration;
        var iteration = new TestingIteration
        {
            Request = task.TestingRequest,
            TestingEffort = (int)Math.Ceiling(((double)lastIteration.TestingEffort /
task.TestingRequest.Iterations.Count),
        };
        task.TestingRequest.AddIteration(iteration);
    }

    public BacklogTask? GetNextTaskForTesting(Tester tester)
    {
        lock (pendingTasks)
        {
            foreach (var pendingTask in pendingTasks)
            {
                if (pendingTask.AssignedTo!.Team != tester.Team)
                    continue;

                if (pendingTask.TestingRequest!.AssignedTo is null)
                {
                    pendingTask.TestingRequest.AssignedTo = tester;
                    return pendingTask;
                }
            }
        }
    }
}

```

```

        return null;
    }

    public void FinalizeTesting(BacklogTask task)
    {
        var iteration = task.TestingRequest!.LastIteration;
        if (ShouldApprove(iteration))
        {
            iteration.Outcome = TaskValidationOutcome.Approved;
            lock (pendingTasks)
                pendingTasks.Remove(task);
        }
        else
        {
            iteration.Outcome = TaskValidationOutcome.ReworkNeeded;
            iteration.ReworkEffort = ReworkDistribution.Sample(task.Effort,
task.TestingRequest.Iterations.Count);

            var developerTeam = task.AssignedTo!.Team;
            var ownModulesRatio = task.AffectedModules.Count(m => m.OwningTeam ==
developerTeam)
                                / (double)task.AffectedModules.Count;
            iteration.ReworkEffort -= (int)(0.2 * ownModulesRatio *
iteration.ReworkEffort);
        }
    }

    static bool ShouldApprove(TestingIteration iteration)
    {
        var probabilityOfRework = Math.Max(0,
            ProbabilityOfInitialRework - (iteration.Request.Iterations.Count - 1) *
ProbabilityOfReworkDecrement);

        var requiresRework = Generator.Random.NextDouble() <= probabilityOfRework;
        return !requiresRework;
    }

    static int CalcInitialTestingEffort(BacklogTask task)
    {
        var baseEffort = (int)Math.Ceiling(task.Effort * TestingEffortPercentage);
        if (task.AffectedModules[0].Codebase.Architecture is
ArchitectureType.Microservices)
            return baseEffort;

        var totalBaseSize = task.AffectedModules.Sum(static m => m.SizePercent);
        var totalDependantSize = task.AffectedModules.SelectMany(static m =>
m.Dependants)
            .Distinct().Sum(static m => m.SizePercent);

        return baseEffort + (int)Math.Ceiling(baseEffort * totalDependantSize /
totalBaseSize);
    }
}

```

Layers/SoftwareProduct.cs

```

using Mars.Components.Layers;
using Mars.Interfaces.Annotations;
using Mars.Interfaces.Data;
using Mars.Interfaces.Layers;
using MarsSim.Entities;

```

```

using MarsSim.Utilities;

namespace MarsSim.Layers;

public sealed class SoftwareProduct : AbstractLayer
{
    public List<Team> Teams { get; private set; } = null!;

    public Codebase Codebase { get; private set; } = null!;

    [PropertyDescription]
    public WorkSchedule WorkSchedule { get; set; } = null!;

    [PropertyDescription]
    public Backlog Backlog { get; set; } = null!;

    [PropertyDescription]
    public CodeReview Review { get; set; } = null!;

    [PropertyDescription]
    public Testing Testing { get; set; } = null!;

    [PropertyDescription]
    public Deployment Deployment { get; set; } = null!;

    [PropertyDescription]
    public Scenario Scenario { get; set; } = null!;

    [PropertyDescription]
    public Telemetry Telemetry { get; set; } = null!;

    [PropertyDescription]
    public string? OutputPath { get; set; }

    public override bool InitLayer(LayerInitData layerInitData, RegisterAgent?
registerAgentHandle = null,
    UnregisterAgent? unregisterAgent = null)
    {
        base.InitLayer(layerInitData, registerAgentHandle, unregisterAgent);

        Teams = Scenario.Teams
            .Select(static t => Team.Create(t.DevelopersCount, t.TestersCount,
t.DevelopersTimeZones, t.TestersTimeZones))
            .ToList();

        Codebase = Codebase.Create(
            Scenario.Project.Architecture,
            Scenario.Project.CodeOwnership,
            Scenario.Project.TotalLinesOfCode,
            Scenario.Project.ModulesCount,
            Teams);

        foreach (var team in Teams)
        {
            foreach (var dev in team.Developers)
            {
                dev.Init(this);
                RegisterAgent!(this, dev);
            }

            foreach (var t in team.Testers)

```

```

        {
            t.Init(this);
            RegisterAgent!(this, t);
        }
    }

    if (OutputPath is not null)
    {
        if (!Directory.Exists(OutputPath))
            Directory.CreateDirectory(OutputPath);

        using var outputWriter = new StreamWriter(Path.Combine(OutputPath,
"Modules.csv"));
        outputWriter.WriteLine("Index,SizePercent,ChangeProbability,OwningTeam");
        foreach (var module in Codebase.Modules)

outputWriter.WriteLine($"{module.Index},{module.SizePercent},{module.ChangeProbability}
,{module.OwningTeam?.ID}");
    }

    return true;
}
}

```

Layers/WorkSchedule.cs

```

using Mars.Components.Layers;
using Mars.Interfaces.Data;
using Mars.Interfaces.Layers;
using MarsSim.Entities;

namespace MarsSim.Layers;

public sealed class WorkSchedule : AbstractActiveLayer
{
    int hour = 0;
    int minute = 0;

    public TimePhase GetTimePhase(int timeZone)
    {
        var localHour = hour + timeZone;
        return (localHour, minute) switch
        {
            (8, 0) => TimePhase.WorkDayStart,
            (12, 0) => TimePhase.LunchStart,
            (12, _) => TimePhase.LunchTime,
            (13, 0) => TimePhase.AfterLunchStart,
            (17, 0) => TimePhase.EndOfDay,
            (>= 17 or < 8, _) => TimePhase.NonWorkingHours,
            _ => TimePhase.WorkingHours,
        };
    }

    public override void Tick()
    {
        minute++;
        if (minute is 60)
        {
            minute = 0;
            hour++;
            if (hour is 24)

```

```

        hour = 0;
    }
}

```

Agents/Tester.cs

```

using System.Diagnostics.CodeAnalysis;
using Mars.Components.Agents;
using Mars.Interfaces.Agents;
using Mars.Interfaces.Annotations;
using MarsSim.Entities;
using MarsSim.Layers;

namespace MarsSim.Agents;

public sealed class Tester : AbstractAgent, IAgent<SoftwareProduct>
{
    const int MaxIdleDuration = 15;

    public Team Team { get; init; } = null!;

    [PropertyDescription]
    public int TimeZone { get; init; } = 0;

    [PropertyDescription]
    public Guid TeamId => Team.ID;

    [PropertyDescription]
    public bool IsIdle => activeTask is null;

    [PropertyDescription]
    public int WaitingTaskCount => pendingTasks.Count;

    SoftwareProduct Product = null!;
    WorkSchedule WorkSchedule => Product.WorkSchedule;
    Testing Testing => Product.Testing;
    Telemetry Telemetry => Product.Telemetry;

    readonly HashSet<BacklogTask> pendingTasks = new();
    BacklogTask? activeTask;
    int? idleDuration;

    public void Init(SoftwareProduct layer) => Product = layer;

    public override void Tick()
    {
        var timePhase = WorkSchedule.GetTimePhase(TimeZone);
        if (timePhase is TimePhase.NonWorkingHours or TimePhase.LunchTime)
            return;

        if (timePhase is TimePhase.EndOfDay or TimePhase.LunchStart)
        {
            if (idleDuration.HasValue)
                Telemetry.Track("TesterRested", actor: ID, spent: idleDuration.Value);
            idleDuration = null;
            return;
        }

        if (activeTask is not null)
        {

```

```

var testingIteration = activeTask.TestingRequest!.LastIteration;
testingIteration.TestingTimeSpent++;
if (testingIteration.TestingTimeSpent >= testingIteration.TestingEffort)
{
    Testing.FinalizeTesting(activeTask);
    Telemetry.Track("TestingFinished",
        actor: ID,
        task: activeTask,
        iteration: activeTask.TestingRequest.Iterations.Count,
        outcome: testingIteration.Outcome.ToString(),
        spent: testingIteration.TestingTimeSpent);

    if (testingIteration.Outcome is TaskValidationOutcome.ReworkNeeded)
        pendingTasks.Add(activeTask);
    else
        pendingTasks.Remove(activeTask);

    activeTask = null;
    idleDuration = 0;
}
}
else
{
    if (idleDuration.HasValue && ++idleDuration < MaxIdleDuration)
        return;

    if (!TryPickForReTest(out activeTask))
        activeTask = Testing.GetNextTaskForTesting(this);

    if (activeTask is not null)
    {
        if (idleDuration.HasValue)
        {
            Telemetry.Track("TesterRested", actor: ID, spent:
idleDuration.Value);
            idleDuration = null;
        }
        Telemetry.Track("TestingStarted",
            actor: ID,
            task: activeTask,
            iteration: activeTask.TestingRequest!.Iterations.Count
        );
    }
}
}

bool TryPickForReTest([MaybeNullWhen(false)] out BacklogTask task)
{
    foreach (var pendingTask in pendingTasks)
    {
        if (pendingTask.TestingRequest is null)
            continue;

        if (pendingTask.TestingRequest.LastIteration.Outcome is
TaskValidationOutcome.Unknown)
        {
            task = pendingTask;
            return true;
        }
    }
}

```

```

        task = null;
        return false;
    }
}

```

Agents/Developer.cs

```

using System.Diagnostics.CodeAnalysis;
using Mars.Components.Agents;
using Mars.Interfaces.Agents;
using Mars.Interfaces.Annotations;
using MarsSim.Entities;
using MarsSim.Layers;
using MarsSim.Utilities;

namespace MarsSim.Agents;

public sealed class Developer : AbstractAgent, IAgent<SoftwareProduct>
{
    const int MaxIdleDuration = 15;

    public Team Team { get; init; } = null!;

    [PropertyDescription]
    public int TimeZone { get; init; } = 0;

    [PropertyDescription]
    public int Fatigue { get; set; } = 0;

    [PropertyDescription]
    public DeveloperActivity Activity => activity;

    [PropertyDescription]
    public Guid TeamId => Team.ID;

    SoftwareProduct Product = null!;
    WorkSchedule WorkSchedule => Product.WorkSchedule;
    Backlog Backlog => Product.Backlog;
    CodeReview Review => Product.Review;
    Testing Testing => Product.Testing;
    Deployment Deployment => Product.Deployment;
    Telemetry Telemetry => Product.Telemetry;
    int maxWip;

    DeveloperActivity activity;
    int? idleDuration;
    BacklogTask? activeTask;
    CodeReviewAssignment[]? activeReview;

    readonly List<BacklogTask> pendingTasks = new();

    public void Init(SoftwareProduct layer)
    {
        Product = layer;
        maxWip = layer.Scenario.Project.MaxDevWorkInProgress;
    }

    public override void Tick()
    {
        var timePhase = WorkSchedule.GetTimePhase(TimeZone);
        if (timePhase is TimePhase.EndOfDay or TimePhase.LunchStart)

```

```

{
    activity = DeveloperActivity.Idle;
    activeTask = null;
    if (idleDuration.HasValue)
    {
        Telemetry.Track("DevRested", actor: ID, spent: idleDuration.Value);
        idleDuration = null;
    }
    return;
}

if (timePhase is TimePhase.NonWorkingHours or TimePhase.LunchTime)
    return;

if (timePhase is TimePhase.WorkDayStart)
    Fatigue = 0;
else if (timePhase is TimePhase.AfterLunchStart)
    Fatigue /= 2;

if (activity is DeveloperActivity.Development)
{
    Fatigue += FatigueEvaluator.DefaultIncrement;
    activeTask!.DevelopmentTimeSpent++;
    if (activeTask.DevelopmentTimeSpent >= activeTask.Effort)
    {
        Telemetry.Track("DevFinished", actor: ID, task: activeTask, spent:
activeTask.DevelopmentTimeSpent);
        Review.Send(activeTask);
        activeTask = null;
        activity = DeveloperActivity.Idle;
    }
}
else if (activity is DeveloperActivity.Refactoring)
{
    Fatigue += FatigueEvaluator.DefaultIncrement;
    var lastIteration = activeTask!.CodeReviewRequest!.LastIteration;
    lastIteration.ReworkTimeSpent++;
    if (lastIteration.ReworkTimeSpent >= lastIteration.ReworkEffort)
    {
        Telemetry.Track("ReworkFinished",
            actor: ID,
            task: activeTask,
            spent: lastIteration.ReworkTimeSpent,
            iteration: activeTask.CodeReviewRequest.Iterations.Count);

        Review.Return(activeTask);
        activeTask = null;
        activity = DeveloperActivity.Idle;
    }
}
else if (activity is DeveloperActivity.Review)
{
    Fatigue += FatigueEvaluator.IntenseIncrement;
    foreach (var a in activeReview!)
    {
        if (a.Status is not CodeReviewAssignmentStatus.Started)
            continue;

        a.TimeSpent++;
        if (a.TimeSpent >= a.Artifact.ReviewEffort)
            Review.FinishReview(a);
    }
}

```

```

        break;
    }

    if (activeReview[^1].Status is not CodeReviewAssignmentStatus.Started)
    {
        Telemetry.Track("ReviewFinished",
            actor: ID,
            task: activeReview[0].Artifact.Iteration.Request.SourceTask,
            outcome: activeReview.Any(a => a.Status ==
CodeReviewAssignmentStatus.ReworkNeeded)
                ? "ReworkNeeded"
                : "Approved",
            iteration:
activeReview[0].Artifact.Iteration.Request.Iterations.Count,
            spent: activeReview.Sum(a => a.TimeSpent));

        activeReview = null;
        activity = DeveloperActivity.Idle;
    }
}
else if (activity is DeveloperActivity.Bugfixing)
{
    Fatigue += FatigueEvaluator.DefaultIncrement;
    var iteration = activeTask!.TestingRequest!.LastIteration;
    iteration.ReworkTimeSpent++;
    if (iteration.ReworkTimeSpent >= iteration.ReworkEffort)
    {
        Telemetry.Track("BugfixingFinished",
            actor: ID,
            task: activeTask,
            spent: iteration.ReworkTimeSpent,
            iteration: activeTask.TestingRequest.Iterations.Count);

        Testing.Return(activeTask);
        activeTask = null;
        activity = DeveloperActivity.Idle;
    }
}
else if (activity is DeveloperActivity.FixingDeployment)
{
    Fatigue += FatigueEvaluator.DefaultIncrement;
    var lastIteration = activeTask!.DeploymentRequest!.LastIteration;
    lastIteration.ReworkTimeSpent++;
    if (lastIteration.ReworkTimeSpent >= lastIteration.ReworkEffort)
    {
        Telemetry.Track("DeploymentFixingFinished",
            actor: ID,
            task: activeTask,
            spent: lastIteration.ReworkTimeSpent,
            iteration: activeTask.DeploymentRequest.Iterations.Count);

        Deployment.Return(activeTask);
        activeTask = null;
        activity = DeveloperActivity.Idle;
    }
}
else if (activity is DeveloperActivity.Idle)
{
    PushPendingTasksToNextStage();

    if (idleDuration.HasValue && ++idleDuration < MaxIdleDuration)

```

```

        return;

    if (TryPickFailedDeployment(out activeTask))
    {
        activity = DeveloperActivity.FixingDeployment;
        Telemetry.Track("StartedFixingFailedDeployment",
            actor: ID,
            task: activeTask,
            iteration: activeTask.DeploymentRequest!.Iterations.Count);
    }
    else if (activeReview is not null ||
        (ShouldPickAnyReviews() && Review.TryGetNextReviewTask(this, out
activeReview)))
    {
        activity = DeveloperActivity.Review;
        Telemetry.Track("StartedReview",
            actor: ID,
            task: activeReview[0].Artifact.Iteration.Request.SourceTask,
            iteration:
activeReview[0].Artifact.Iteration.Request.Iterations.Count);
    }
    else if (activeTask is not null)
    {
        activity = DeveloperActivity.Development;
        Telemetry.Track("ContinueDevelopmentOnTask",
            actor: ID,
            task: activeTask);
    }
    else if (TryPickForRework(out activeTask))
    {
        activity = DeveloperActivity.Refactoring;
        Telemetry.Track("StartedRework",
            actor: ID,
            task: activeTask,
            iteration: activeTask.CodeReviewRequest!.Iterations.Count);
    }
    else if (TryPickForBugfixing(out activeTask))
    {
        activity = DeveloperActivity.Bugfixing;
        Telemetry.Track("StartedBugfixing",
            actor: ID,
            task: activeTask,
            iteration: activeTask.TestingRequest!.Iterations.Count);
    }
    else if (TryContinueDevelopment(out activeTask))
    {
        activity = DeveloperActivity.Development;
        Telemetry.Track("SwitchedBackToPostponedDevTask",
            actor: ID,
            task: activeTask);
    }
    else if (pendingTasks.Count < maxWip)
    {
        activeTask = Backlog.GetNextTask(this);
        pendingTasks.Add(activeTask);
        activity = DeveloperActivity.Development;
        Telemetry.Track("StartedNewDevelopmentTask",
            actor: ID,
            task: activeTask!,
            moduleCount: activeTask.AffectedModules.Count,

```

```

        locPercent: activeTask.AffectedModules.Sum(static m =>
m.SizePercent));
    }

    if (activity is not DeveloperActivity.Idle && idleDuration.HasValue)
    {
        Telemetry.Track("DevRested", actor: ID, spent: idleDuration.Value);
        idleDuration = null;
    }
}

// just became idle
if (activity is DeveloperActivity.Idle && !idleDuration.HasValue)
    idleDuration = 0;
}

bool ShouldPickAnyReviews() => FatigueEvaluator.Eval(Fatigue);

bool TryPickForRework([MaybeNullWhen(false)] out BacklogTask task)
{
    foreach (var pendingTask in pendingTasks)
    {
        if (pendingTask.CodeReviewRequest is null)
            continue;

        if (pendingTask.CodeReviewRequest.LastIteration.Outcome is
TaskValidationOutcome.ReworkNeeded)
        {
            task = pendingTask;
            return true;
        }
    }

    task = null;
    return false;
}

bool TryPickForBugfixing([MaybeNullWhen(false)] out BacklogTask task)
{
    foreach (var pendingTask in pendingTasks)
    {
        if (pendingTask.TestingRequest is null)
            continue;

        if (pendingTask.TestingRequest.LastIteration.Outcome is
TaskValidationOutcome.ReworkNeeded)
        {
            task = pendingTask;
            return true;
        }
    }

    task = null;
    return false;
}

bool TryPickFailedDeployment([MaybeNullWhen(false)] out BacklogTask task)
{
    foreach (var pendingTask in pendingTasks)
    {
        if (pendingTask.DeploymentRequest is null)

```

```

        continue;

        if (pendingTask.DeploymentRequest.LastIteration.Outcome is
TaskValidationOutcome.ReworkNeeded)
        {
            task = pendingTask;
            return true;
        }
    }

    task = null;
    return false;
}

bool TryContinueDevelopment([MaybeNullWhen(false)] out BacklogTask task)
{
    foreach (var pendingTask in pendingTasks)
    {
        if (pendingTask.CodeReviewRequest is null)
        {
            task = pendingTask;
            return true;
        }
    }

    task = null;
    return false;
}

void PushPendingTasksToNextStage()
{
    for (var i = pendingTasks.Count - 1; i >= 0; i--)
    {
        var pendingTask = pendingTasks[i];
        if (pendingTask.DeploymentRequest?.LastIteration.Outcome is
TaskValidationOutcome.Approved)
        {
            pendingTasks.RemoveAt(i);
            Backlog.CompleteTask(pendingTask);
            Telemetry.Track("TaskCompleted",
                actor: ID,
                task: pendingTask);
        }
        else if (pendingTask.TestingRequest?.LastIteration.Outcome is
TaskValidationOutcome.Approved)
        {
            if (pendingTask.DeploymentRequest is null)
            {
                Deployment.Send(pendingTask);
                Telemetry.Track("TaskSentToDeployment",
                    actor: ID,
                    task: pendingTask);
            }
        }
        else if (pendingTask.CodeReviewRequest?.LastIteration.Outcome is
TaskValidationOutcome.Approved)
        {
            if (pendingTask.TestingRequest is null)
            {
                Testing.Send(pendingTask);
                Telemetry.Track("TaskSentToTesting",

```



```

        description.AddEntity<BacklogTask>();

        description.AddAgent<Developer, SoftwareProduct>();
        description.AddAgent<Tester, SoftwareProduct>();

        return description;
    }

    public SimulationConfig CreateConfig(Scenario scenario)
    {
        return new SimulationConfig
        {
            SimulationIdentifier = "sdlc",
            Globals =
            {
                Steps = durationInDays * MinutesPerDay,
                // OutputTarget = outputPath is not null ? OutputTargetType.Csv :
                OutputTargetType.None,
                // CsvOptions = { OutputPath = outputPath },
            },
            LayerMappings =
            {
                new LayerMapping
                {
                    Name = nameof(Telemetry),
                    IndividualMapping =
                    {
                        new IndividualMapping { ParameterName = "OutputPath", Value =
                        outputPath },
                    },
                },
                new LayerMapping
                {
                    Name = nameof(SoftwareProduct),
                    IndividualMapping =
                    {
                        new IndividualMapping { ParameterName = "Scenario", Value =
                        scenario },
                        new IndividualMapping { ParameterName = "OutputPath", Value =
                        outputPath },
                    },
                },
            },
        };
    }
}

```

Utilities/TeamTopology.cs

```

using System.Text.Json;

namespace MarsSim.Utilities;

public sealed class TeamTopology
{
    public Scenario.TeamStructure[] Teams { get; init; } = null!;

    public static TeamTopology Load(string name)
    {
        using var s = File.OpenRead(Path.Combine("Scenarios", "TeamTopologies",
        $"{name}.json"));
    }
}

```

```

        return JsonSerializer.Deserialize<TeamTopology>(s) ?? throw new
        ArgumentException("Scenario not found");
    }
}

```

Utilities/EffortGenerator.cs

```

using MathNet.Numerics.Distributions;

namespace MarsSim.Utilities;

sealed class EffortGenerator(int seed, int maxMinutes)
{
    const double ProbabilityAdjustmentFor20MinutesEffort = 0.119844;

    readonly InverseGaussian dist = new(112.8197, 49.1561, new Random(seed));

    public int Next()
    {
        while (true)
        {
            while (dist.RandomSource.NextDouble() <=
ProbabilityAdjustmentFor20MinutesEffort)
                return 20;

            var v = (int)Math.Ceiling(dist.Sample() / 10) * 10;
            if (v <= maxMinutes) // eliminate long distribution tail
                return v;
        }
    }
}

```

Utilities/ReworkDistribution.cs

```

using Mars.Common.Core.Random;
using MathNet.Numerics.Distributions;

namespace MarsSim.Utilities;

static class ReworkDistribution
{
    const double LowerBound = 0.1;
    const double UpperBound = 0.3;

    static readonly Normal Distribution = new(
        mean: LowerBound + UpperBound / 2,
        stddev: (UpperBound - LowerBound) / 6,
        randomSource: new Random(Generator.Seed!.Value));

    public static int Sample(int initialEffort, int iterationNumber)
    {
        var percent = 0d;
        while (percent is < LowerBound or > UpperBound)
            percent = Distribution.Sample();

        return (int)Math.Ceiling(initialEffort * percent / iterationNumber);
    }
}

```

Utilities/CouplingGenerator.cs

```
namespace MarsSim.Utilities;

static class CouplingGenerator
{
    public static double[] GetChangeProbabilities(int moduleCount, int seed)
    {
        var countLog = Math.Log(moduleCount);
        var m = -1.2624653134178174 * countLog + 1.407336741730882;
        var s = Math.Sqrt(0.24806030790958875 * countLog + 0.13184253926824063);
        var dist = MathNet.Numerics.Distributions.LogNormal.WithMuSigma(m, s, new
Random(seed));
        var probabilities = new double[moduleCount];
        dist.Samples(probabilities);
        DecreaseNoneProbability(probabilities, 0.0001);
        return probabilities;
    }

    static void DecreaseNoneProbability(double[] probabilities, double
maxNoneProbability)
    {
        var noneProbability = probabilities.Select(v => 1.0 - v).Aggregate(1.0, (a, v)
=> a * v);
        if (noneProbability < maxNoneProbability)
            return;

        var k = Math.Pow(maxNoneProbability / noneProbability, 1.0 /
probabilities.Length);
        for (var i = 0; i < probabilities.Length; i++)
            probabilities[i] = 1.0 - k * (1 - probabilities[i]);
    }
}
```

Utilities/ProjectConfig.cs

```
using System.Text.Json;

namespace MarsSim.Utilities;

public sealed class ProjectConfig
{
    public Scenario.ProjectInfo Project { get; init; } = null!;

    public static ProjectConfig Load(string name)
    {
        using var s = File.OpenRead(Path.Combine("Scenarios", "Projects",
$"{name}.json"));
        return JsonSerializer.Deserialize<ProjectConfig>(s)
            ?? throw new ArgumentException($"Project {name} not found");
    }
}
```

Utilities/Scenario.cs

```
using System.Text.Json;
using System.Text.Json.Serialization;
using MarsSim.Entities;
```

```

namespace MarsSim.Utilities;

public sealed class Scenario
{
    public required string Name { get; init; }
    public required string VariationName { get; init; }

    public required string ProjectName { get; init; }
    public required ProjectInfo Project { get; init; }

    public required string TeamTopologyName { get; init; }
    public required TeamStructure[] Teams { get; init; }

    public sealed class ProjectInfo
    {
        [JsonConverter(typeof(JsonStringEnumConverter))]
        public required ArchitectureType Architecture { get; init; }

        public required int ModulesCount { get; init; }

        [JsonConverter(typeof(JsonStringEnumConverter))]
        public required CodeOwnershipType CodeOwnership { get; init; }

        public required int TotalLinesOfCode { get; init; }

        public int MaxDevWorkInProgress { get; init; } = 3;
    }

    public sealed class TeamStructure
    {
        public int DevelopersCount { get; init; } = 4;
        public int[] DevelopersTimeZones { get; init; } = [0];

        public int TestersCount { get; init; } = 2;
        public int[] TestersTimeZones { get; init; } = [0];
    }

    public static IEnumerable<Scenario> Load(string[] projectNames, string[]
teamTopologyNames)
    {
        return
            from p in projectNames.Select(static n => (n, p:
ProjectConfig.Load(n).Project))
            from t in teamTopologyNames.Select(static n => (n, t:
TeamTopology.Load(n).Teams))
            select new Scenario
            {
                Name = $"{p.n}__{t.n}",
                VariationName = "NOVAR",
                ProjectName = p.n, Project = p.p,
                TeamTopologyName = t.n, Teams = t.t,
            };
    }

    public void WriteTo(string outputDir)
    {
        Directory.CreateDirectory(outputDir);
        var fi = new FileInfo(Path.Combine(outputDir, $"Scenario.json"));
        if (fi.Exists)
            fi.Delete();
        using var s = fi.OpenWrite();
    }
}

```

```

        JsonSerializer.Serialize(s, this);
    }

    internal Scenario[] ExpandWith(Variation variation)
    {
        var result = new Scenario[variation.VariationsCount];
        for (var i = 0; i < variation.VariationsCount; i++)
        {
            result[i] = new Scenario
            {
                Name = $"{Name}__VAR-{variation.Name}-{i + 1:000}",
                VariationName = variation.Name,
                ProjectName = ProjectName,
                TeamTopologyName = TeamTopologyName,
                Teams = variation is not { TeamsCount: null, DevsPerTeam: null,
TestersPerTeam: null }
                    ? ExpandTeams(Teams, variation.TeamsCount?[i],
variation.DevsPerTeam?[i], variation.TestersPerTeam?[i])
                    : Teams,
                Project = new ProjectInfo
                {
                    Architecture = Project.Architecture,
                    CodeOwnership = Project.CodeOwnership,
                    TotalLinesOfCode = variation.TotalLinesOfCode?[i] ??
Project.TotalLinesOfCode,
                    ModulesCount = variation.ModulesCount?[i] ?? Project.ModulesCount,
                    MaxDevWorkInProgress = variation.MaxDevWorkInProgress?[i] ??
Project.MaxDevWorkInProgress,
                },
            };
        }
        return result;
    }

    static TeamStructure[] ExpandTeams(TeamStructure[] teams, int? count, int? devs,
int? testers)
    {
        var teamsCount = count ?? teams.Length;
        var result = new TeamStructure[teamsCount];
        for (var i = 0; i < teamsCount; i++)
        {
            var team = teams[i % teams.Length];
            result[i] = devs.HasValue || testers.HasValue ? new TeamStructure
            {
                DevelopersCount = devs ?? team.DevelopersCount,
                TestersCount = testers ?? team.TestersCount,
            } : team;
        }
        return result;
    }
}

```

Utilities/Variation.cs

```

using System.Text.Json;

namespace MarsSim.Utilities;

sealed class Variation
{
    public string Name { get; private set; } = "";
}

```

```

public int[]? TotalLinesOfCode { get; init; }
public int[]? ModulesCount { get; init; }
public int[]? MaxDevWorkInProgress { get; init; }
public int[]? TeamsCount { get; init; }
public int[]? DevsPerTeam { get; init; }
public int[]? TestersPerTeam { get; init; }

public int VariationsCount { get; private set; }

public static Variation[] Load(string[] names)
{
    if (names.Length == 0)
        return [];

    var result = new Variation[names.Length];
    for (var i = 0; i < names.Length; i++)
        result[i] = Load(names[i]);
    return result;
}

static Variation Load(string name)
{
    using var s = File.OpenRead(Path.Combine("Scenarios", "Variations",
        $"{name}.json"));
    var result = JsonSerializer.Deserialize<Variation>(s)
        ?? throw new ArgumentException($"Variation {name} not found");

    var locCount = result.TotalLinesOfCode?.Length;
    var moduleCount = result.ModulesCount?.Length;
    var maxWip = result.MaxDevWorkInProgress?.Length;
    var teamsCount = result.TeamsCount?.Length;
    var devsPerTeam = result.DevsPerTeam?.Length;
    var testersPerTeam = result.TestersPerTeam?.Length;
    result.VariationsCount = locCount ?? moduleCount ?? maxWip ?? teamsCount ??
    devsPerTeam ?? testersPerTeam ?? 0;
    result.Name = name;

    if (result.VariationsCount is 0)
        throw new FormatException("Variation must have at least one variable with
at least one value");

    if (locCount.HasValue && locCount.Value != result.VariationsCount
        || moduleCount.HasValue && moduleCount.Value != result.VariationsCount
        || maxWip.HasValue && maxWip.Value != result.VariationsCount
        || teamsCount.HasValue && teamsCount.Value != result.VariationsCount
        || devsPerTeam.HasValue && devsPerTeam.Value != result.VariationsCount
        || testersPerTeam.HasValue && testersPerTeam.Value !=
result.VariationsCount)
        throw new FormatException("The number of variation values must be equal for
all variation parameters");

    return result;
}

public static Scenario[] Expand(Scenario[] scenarios, Variation[] variations)
{
    if (variations.Length == 0)
        return scenarios;

    var expanded = new Scenario[scenarios.Length * variations.Sum(static v =>
v.VariationsCount)];

```

```

        var index = 0;
        foreach (var variation in variations)
        {
            foreach (var scenario in scenarios)
            {
                var subs = scenario.ExpandWith(variation);
                subs.CopyTo(expanded, index);
                index += subs.Length;
            }
        }
        return expanded;
    }
}

```

Entities/DeploymentRequest.cs

```

namespace MarsSim.Entities;

public sealed class DeploymentRequest
{
    public required BacklogTask SourceTask { get; init; }
    public required int DeploymentEffort { get; set; }

    public List<DeploymentIteration> Iterations { get; } = new();
    public DeploymentIteration LastIteration => Iterations[^1];
}

```

Entities/ArchitectureType.cs

```

namespace MarsSim.Entities;

public enum ArchitectureType
{
    Monolith,
    Microservices,
}

```

Entities/CodeOwnershipType.cs

```

namespace MarsSim.Entities;

public enum CodeOwnershipType
{
    Collective,
    Weak,
    Strong,
}

```

Entities/CodeReviewIteration.cs

```

namespace MarsSim.Entities;

public sealed class CodeReviewIteration
{
    public required CodeReviewRequest Request { get; init; }
    public List<CodeReviewArtifact> Artifacts { get; } = new();

    public TaskValidationOutcome Outcome { get; set; }
    public int ReworkEffort { get; set; }
}

```

```

        public int ReworkTimeSpent { get; set; }
    }

```

Entities/CodeReviewRequest.cs

```

namespace MarsSim.Entities;

```

```

public sealed class CodeReviewRequest
{

```

```

    const double CodingMinutesPerLineOfCode = 1.0;
    const double ReviewMinutesPerLineOfCode = .25;

```

```

    public required BacklogTask SourceTask { get; init; }

```

```

    public List<CodeReviewIteration> Iterations { get; } = new();
    public CodeReviewIteration LastIteration => Iterations[^1];

```

```

    private CodeReviewRequest()
    {
    }

```

```

    public static CodeReviewRequest Create(BacklogTask task)
    {

```

```

        ArgumentNullException.ThrowIfNull(task.AffectedModules);
        var request = new CodeReviewRequest
        {
            SourceTask = task,
        };
        var iteration = new CodeReviewIteration { Request = request };
        request.Iterations.Add(iteration);

```

```

        var totalSizePercent = task.AffectedModules.Sum(static m => m.SizePercent);
        var changedLinesOfCode = task.Effort / CodingMinutesPerLineOfCode;
        var artifacts = task.AffectedModules.Select(m => new CodeReviewArtifact
        {
            Iteration = iteration,
            Module = m,
            ReviewEffort = (int)(
                m.SizePercent / totalSizePercent * changedLinesOfCode *
ReviewMinutesPerLineOfCode
            ),
        });

```

```

        iteration.Artifacts.AddRange(artifacts);

```

```

        foreach (var artifact in iteration.Artifacts)
        {

```

```

            if (artifact.Module.OwningTeam is not null)
            {

```

```

                foreach (var developer in artifact.Module.OwningTeam.Developers)
                {
                    if (developer == task.AssignedTo)
                        continue;

```

```

                    artifact.Assignments.Add(new CodeReviewAssignment
                    {
                        Artifact = artifact,
                        Developer = developer,
                    });
                }

```

```

            }
            else // collective ownership

```

```

        {
            foreach (var developer in task.AssignedTo!.Team.Developers)
            {
                if (developer == task.AssignedTo)
                    continue;

                artifact.Assignments.Add(new CodeReviewAssignment
                {
                    Artifact = artifact,
                    Developer = developer,
                });
            }
        }
    }

    return request;
}

```

Entities/CodebaseModule.cs

```

using Mars.Components.Agents;

namespace MarsSim.Entities;

public sealed class CodebaseModule : AbstractEntity
{
    public required Codebase Codebase { get; init; }
    public Team? OwningTeam { get; init; }
    public required int Index { get; init; }
    public double ChangeProbability { get; init; }
    public double SizePercent { get; init; }

    public IEnumerable<CodebaseModule> DependsOn => Codebase.Modules.Take(Index);
    public IEnumerable<CodebaseModule> Dependants => Codebase.Modules.Skip(Index + 1);
}

```

Entities/Codebase.cs

```

using Mars.Common.Core.Random;
using Mars.Components.Agents;
using MarsSim.Utilities;

namespace MarsSim.Entities;

public sealed class Codebase : AbstractEntity
{
    public required ArchitectureType Architecture { get; init; }
    public required CodeOwnershipType OwnershipType { get; init; }
    public required int LinesOfCode { get; init; }
    public List<CodebaseModule> Modules { get; private set; } = null!;

    public List<CodebaseModule> PickChangedModules()
    {
        var result = new List<CodebaseModule>();
        while (result.Count is 0)
        {
            foreach (var module in Modules)
            {
                if (Generator.Random.NextDouble() <= module.ChangeProbability)

```

```

        result.Add(module);
    }
}
return result;
}

public static Codebase Create(
    ArchitectureType architecture, CodeOwnershipType ownershipType,
    int linesOfCode, int moduleCount,
    IReadOnlyList<Team> teams)
{
    var changeProbabilities = CouplingGenerator.GetChangeProbabilities(moduleCount,
Generator.Seed!.Value);
    Array.Sort(changeProbabilities);
    Array.Reverse(changeProbabilities);

    var codebase = new Codebase
    {
        Architecture = architecture,
        OwnershipType = ownershipType,
        LinesOfCode = linesOfCode,
    };

    var sumOfProbabilities = changeProbabilities.Sum();
    var sortedTeams = teams.Select(static t => (t, 0.0)).ToArray();
    var modules = new List<CodebaseModule>(moduleCount);
    for (var i = 0; i < moduleCount; i++)
    {
        var owningTeam = ownershipType is not CodeOwnershipType.Collective
            ? Assign(sortedTeams, changeProbabilities[i])
            : null;
        var module = new CodebaseModule
        {
            Codebase = codebase,
            OwningTeam = owningTeam,
            Index = i,
            ChangeProbability = changeProbabilities[i],
            SizePercent = changeProbabilities[i] / sumOfProbabilities,
        };
        modules.Add(module);
        owningTeam?.OwnedModules.Add(module);
    }

    codebase.Modules = modules;
    return codebase;
}

static Team Assign((Team Team, double Sum)[] array, double changeProbability)
{
    var (team, sum) = array[0];
    sum += changeProbability;
    array[0] = (team, sum);
    for (var i = 1; i < array.Length; i++)
    {
        if (sum < array[i].Sum)
            break;

        array[i - 1] = array[i];
        array[i] = (team, sum);
    }
    return team;
}

```

```
    }
}
```

Entities/TimePhase.cs

```
namespace MarsSim.Entities;
```

```
public enum TimePhase
{
    NonWorkingHours,
    WorkDayStart,
    WorkingHours,
    LunchStart,
    LunchTime,
    AfterLunchStart,
    EndOfDay,
}
```

Entities/Team.cs

```
using Mars.Components.Agents;
using MarsSim.Agents;
```

```
namespace MarsSim.Entities;
```

```
public sealed class Team : AbstractEntity
{
    public List<Developer> Developers { get; } = [];
    public List<Tester> Testers { get; } = [];
    public List<CodebaseModule> OwnedModules { get; } = [];

    public static Team Create(
        int developerCount, int testerCount,
        int[] developerTimeZones, int[] testerTimeZones)
    {
        var team = new Team { ID = Guid.NewGuid() };
        team.Developers.AddRange(Enumerable.Range(0, developerCount).Select(i => new
Developer
        {
            ID = Guid.NewGuid(),
            Team = team,
            TimeZone = developerTimeZones[i % developerTimeZones.Length],
        }));
        team.Testers.AddRange(Enumerable.Range(0, testerCount).Select(i => new Tester
        {
            ID = Guid.NewGuid(),
            Team = team,
            TimeZone = testerTimeZones[i % testerTimeZones.Length],
        }));
        return team;
    }
}
```

Entities/CodeReviewAssignmentStatus.cs

```
namespace MarsSim.Entities;
```

```
public enum CodeReviewAssignmentStatus
{
}
```

```

        NotStarted,
        Started,
        Approved,
        ReworkNeeded,
    }

```

Entities/BacklogTaskType.cs

```

namespace MarsSim.Entities;

public enum BacklogTaskType
{
    Feature,
    Bug,
}

```

Entities/CodeReviewAssignment.cs

```

using MarsSim.Agents;

namespace MarsSim.Entities;

public sealed class CodeReviewAssignment
{
    public required CodeReviewArtifact Artifact { get; init; }
    public required Developer Developer { get; init; }

    public int TimeSpent { get; set; }
    public CodeReviewAssignmentStatus Status { get; set; }
}

```

Entities/BacklogTask.cs

```

using Mars.Components.Agents;
using MarsSim.Agents;

namespace MarsSim.Entities;

public sealed class BacklogTask : AbstractEntity
{
    public BacklogEpic? Epic { get; init; }
    public BacklogTaskType Type { get; init; }
    public Team? OwningTeam { get; init; }
    public int Effort { get; init; }
    public List<CodebaseModule> AffectedModules { get; init; } = null!;

    public Developer? AssignedTo { get; set; }

    public int DevelopmentTimeSpent { get; set; }

    public CodeReviewRequest? CodeReviewRequest { get; set; }
    public TestingRequest? TestingRequest { get; set; }
    public DeploymentRequest? DeploymentRequest { get; set; }
}

```

Entities/DeploymentIteration.cs

```

namespace MarsSim.Entities;

```

```

public sealed class DeploymentIteration
{
    public required DeploymentRequest Request { get; init; }

    public int TimeSpent { get; set; }

    public TaskValidationOutcome Outcome { get; set; }
    public int ReworkEffort { get; set; }
    public int ReworkTimeSpent { get; set; }
}

```

Entities/CodeReviewArtifact.cs

```

namespace MarsSim.Entities;

public sealed class CodeReviewArtifact
{
    public required CodeReviewIteration Iteration { get; init; }
    public required CodebaseModule Module { get; init; }
    public required int ReviewEffort { get; init; }

    public List<CodeReviewAssignment> Assignments { get; } = new();
}

```

Entities/BacklogEpic.cs

```

using Mars.Components.Agents;

namespace MarsSim.Entities;

public sealed class BacklogEpic : AbstractEntity
{
    public List<BacklogTask> Tasks { get; } = new();
}

```

Entities/TestingIteration.cs

```

namespace MarsSim.Entities;

public sealed class TestingIteration
{
    public required TestingRequest Request { get; init; }
    public required int TestingEffort { get; set; }
    public int TestingTimeSpent { get; set; }

    public TaskValidationOutcome Outcome { get; set; }
    public int ReworkEffort { get; set; }
    public int ReworkTimeSpent { get; set; }
}

```

Entities/DeveloperActivity.cs

```

namespace MarsSim.Entities;

public enum DeveloperActivity
{
    Idle,
    Development,
    Refactoring,
}

```

```

        Review,
        Bugfixing,
        FixingDeployment,
    }

```

Entities/TaskValidationOutcome.cs

```

namespace MarsSim.Entities;

public enum TaskValidationOutcome
{
    Unknown,
    Approved,
    ReworkNeeded,
}

```

Entities/TestingRequest.cs

```

using MarsSim.Agents;

namespace MarsSim.Entities;

public sealed class TestingRequest
{
    public required BacklogTask SourceTask { get; init; }

    public Tester? AssignedTo { get; set; }

    readonly List<TestingIteration> iterations = new();
    public IReadOnlyList<TestingIteration> Iterations => iterations;

    public TestingIteration LastIteration
    {
        get
        {
            lock (iterations)
                return Iterations[^1];
        }
    }

    public void AddIteration(TestingIteration iteration)
    {
        lock (iterations)
            iterations.Add(iteration);
    }
}

```

Варіації базових сценаріїв симуляції процесу розробки програмного
забезпечення

Таблиця В.1.

Варіації базових симуляційних сценаріїв

Варіація	Назви параметрів	Варіативні значення
Кількість рядків коду	TotalLinesOfCode	50000, 100000, 200000, 300000, 400000, 500000, 600000, 800000, 1000000
Кількість програмних модулів	ModulesCount	10, 20, 30, 40, 50, 60, 70, 80, 90, 100
Кількість модулів фіксованого розміру	(ModulesCount; TotalLinesOfCode)	(10; 100000), (20; 200000), (30; 300000), (40; 400000), (50; 500000), (60; 600000), (70; 700000), (80; 800000), (90; 900000), (100; 1000000)
Кількість команд розробки	TeamsCount	1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Розмір команд розробки	(DevsPerTeam; TestersPerTeam)	(2; 1), (2; 2), (4; 1), (4; 2), (4; 4), (6; 2), (6; 4), (6; 6), (8; 2), (8; 4), (8; 6), (8; 8)
Максимальна кількість активних задач на одного розробника	MaxDevWorkInProgress	1, 2, 3, 4, 5, 10, 15, 20, 50, 100
Кількість та розмір команд при фіксованій кількості розробників	(TeamsCount; DevsPerTeam; TestersPerTeam)	(2; 10; 10), (4; 5; 5), (5; 4; 4), (10; 2; 2)
Співвідношення кількості команд до кількості програмних модулів	(ModulesCount; TeamsCount)	(10; 4), (20; 4), (30; 4), (40; 4), (50; 4), (60; 4), (70; 4), (80; 4), (90; 4), (100; 4), (10; 6), (20; 6), (30; 6), (40; 6), (50; 6), (60; 6), (70; 6), (80; 6), (90; 6), (100; 6), (10; 8), (20; 8), (30; 8), (40; 8), (50; 8), (60; 8), (70; 8), (80; 8), (90; 8), (100; 8), (10; 10), (20; 10), (30; 10), (40; 10), (50; 10), (60; 10), (70; 10), (80; 10), (90; 10), (100; 10)

Матеріали для опитування щодо валідності симуляційної моделі процесу
розробки програмного забезпечення

Анкета для опитування
щодо визначення валідності агентно-орієнтованої симуляційної
моделі процесу розробки програмного забезпечення

Оцініть за 4-бальною шкалою, наскільки Ви згодні з наведеними твердженнями: 0 балів – абсолютно не згодні, 1 бал – скоріше не згодні, ніж згодні, 2 бали – скоріше згодні, ніж не згодні, 3 бали – абсолютно згодні.

№	Твердження	Оцінка
1.1.	Набір вхідних параметрів моделі є достатнім для моделювання продуктивності розробки.	
1.2.	Вибір типів архітектури серверних програмних систем є релевантним та охоплює найпоширеніші архітектурні підходи в сучасній розробці.	
1.3.	Моделювання залежності типу володіння кодом від програмної архітектури відповідає реальній практиці.	
1.4.	Послідовність етапів роботи на задачами (розробка, рецензування коду, тестування, розгортання) адекватно відображає реальний процес.	
1.5.	Логіка прийняття рішень агентом-розробником є реалістичною моделлю поведінки реального розробника.	
1.6.	Врахування втоми розробника від рецензування програмного коду є важливим фактором, який впливає на продуктивність і має бути присутнім у моделі.	
2.1.	Алгоритм генерації функціональних вимог на основі історичних даних реальних проєктів є обґрунтованим способом моделювання потоку вхідних задач.	
2.2.	Визначення списку програмних модулів, які зачіпає нова функціональна вимога, на основі незалежної ймовірності їх модифікації є доцільним механізмом моделювання модульної зв'язності.	
2.3.	Модель розподілу задач між командами в контексті різних видів власності програмного коду відповідає реальним практикам.	

2.4.	Механізм повернення задачі на доопрацювання при виявленні проблем на будь-якому з етапів роботи над задачею (рецензування коду, тестування, розгортання) відповідає реальному процесу.	
3.1.	В реальних проєктах мікросервісна архітектура дозволяє командам працювати більш незалежно одна від одної порівняно з монолітною архітектурою.	
3.2.	Сильний тип власності програмного коду зменшує кількість конфліктів та потребу в координації між командами.	
3.3.	Збільшення кількості часових поясів, у яких розподілені команди, суттєво ускладнює процес огляду коду та координацію.	
3.4.	Модульна монолітна архітектура є розумним компромісом між звичайним монолітом та мікросервісами з точки зору балансу між незалежністю команд та складністю інфраструктури.	
4.1.	Застосування практики власності програмного коду до монолітного типу програмної архітектури позитивно впливає на якість програмного коду, проте знижує продуктивність процесу розробки.	
4.2.	Проєкти на мікросервісній архітектурі демонструють найвищу ефективність процесу розробки порівняно з монолітним типом архітектури.	
4.3.	Географічний розподіл команд розробки по більшій кількості часових поясів негативно впливає на ефективність процесу розробки через підвищення складності координації команд.	
4.4.	Монолітна архітектура з колективною власністю коду забезпечує найкраще лінійне масштабування процесу розробки при додаванні нових команд.	
5.1.	В цілому, представлена симуляційна модель є адекватним інструментом для дослідження ефективності процесу розробки ПЗ.	
5.2.	Результати такої моделі можуть бути корисними для прийняття практичних рішень щодо організації процесу розробки, вибору архітектури, формування команд тощо.	

Таблиці проміжних і допоміжних даних для аналізу результатів експертного оцінювання симуляційної моделі процесу розробки ПЗ

Таблиця Д.1.

Результати оцінювання симуляційної моделі за структурним критерієм

№ респондента	Показник №					
	1	2	3	4	5	6
1	2	3	2	3	2	3
2	3	3	2	3	1	2
3	1	3	2	3	2	3
4	0	2	1	3	1	2
5	2	3	2	3	2	3
6	1	3	2	3	2	3
7	2	3	2	3	1	2
8	2	3	2	3	1	3
9	1	3	3	3	1	3
10	0	2	1	2	1	2
11	2	3	2	3	2	3
12	2	3	2	3	1	2
13	1	3	2	3	2	3
14	1	2	1	3	1	2
15	2	3	2	3	2	3
16	1	3	2	3	1	2
17	1	3	2	3	2	3
Сер. арифм.	1,41	2,82	1,88	2,94	1,47	2,59

Джерело: опрацьовано автором.

Таблиця Д.2.

Результати оцінювання симуляційної моделі за алгоритмічним критерієм

№ респондента	Показник №			
	1	2	3	4
1	3	2	2	3
2	3	2	2	3
3	3	2	3	3
4	2	1	2	3
5	3	2	2	3
6	3	2	3	3
7	2	1	2	3
8	3	2	2	3
9	3	2	3	3
10	2	1	2	2
11	3	2	2	3
12	3	2	3	3
13	3	3	3	3
14	2	1	2	3
15	3	2	2	3
16	3	2	2	3
17	3	2	3	3
Сер. арифм.	2,76	1,82	2,35	2,94

Джерело: опрацьовано автором.

Таблиця Д.3.

Результати оцінювання симуляційної моделі за концептуальним критерієм

№ респондента	Показник №			
	1	2	3	4
1	3	3	3	2
2	3	2	3	2
3	3	3	3	3
4	3	2	3	2
5	3	2	3	2
6	3	3	3	3
7	3	2	3	2
8	3	3	3	2
9	3	3	3	3
10	3	2	2	1
11	3	3	3	2
12	3	2	3	3
13	3	3	3	3
14	3	2	2	2
15	3	2	3	2
16	3	2	3	2
17	3	3	3	3
Сер. арифм.	3	2,47	2,88	2,29

Джерело: опрацьовано автором.

Результати оцінювання симуляційної моделі за прогностичним критерієм

№ респондента	Показник №			
	1	2	3	4
1	2	3	3	1
2	1	2	3	2
3	2	3	3	2
4	1	2	2	1
5	2	3	3	1
6	2	3	3	2
7	1	2	3	1
8	2	3	3	2
9	2	3	3	2
10	1	2	2	0
11	2	3	3	1
12	2	3	3	2
13	2	3	3	2
14	1	2	2	1
15	2	2	3	1
16	2	3	3	1
17	2	3	3	2
Сер. арифм.	1,71	2,65	2,82	1,41

Джерело: опрацьовано автором.

Таблиця Д.5.

Результати оцінювання симуляційної моделі за прикладним критерієм

№ респондента	Показник №	
	1	2
1	3	3
2	2	3
3	3	3
4	2	2
5	2	3
6	3	3
7	2	3
8	3	3
9	3	3
10	2	2
11	3	3
12	2	3
13	3	3
14	2	2
15	2	3
16	2	3
17	3	3
Сер. арифм.	2,47	2,82

Джерело: опрацьовано автором.

Список публікацій здобувача за темою дисертації

Статті у фахових виданнях, включених до переліку наукових фахових видань України з присвоєнням категорії «Б»:

1. Москалик Д.О., Антонюк Д.С. Імовірнісний розподіл складності вхідних задач для дискретно-подійного та агентно-орієнтованого моделювання процесу розробки програмного забезпечення. Вчені записки Таврійського національного університету імені В.І. Вернадського. Серія: Технічні науки. Том 35 (74) № 3, Частина 1. 2024. С. 152-159. DOI: 10.32782/2663-5941/2024.3.1/23
2. Москалик Д.О., Антонюк Д.С. Моделювання зв'язності шляхом імовірнісного розподілу частот внесення змін до програмних модулів при симуляції процесу розробки програмного забезпечення. Вісник Херсонського національного технічного університету. Том 2 № 2 (93). 2025. С. 238-245. DOI: 10.35546/kntu2078-4481.2025.2.2.28
3. Москалик Д.О., Антонюк Д.С. Моделювання процесу рецензування коду в агентно-орієнтованих симуляціях розробки модульного програмного забезпечення. Актуальні проблеми автоматизації та інформаційних технологій. Том 29. 2025. С. 306-317. DOI: 10.15421/432528

Публікація у матеріалах міжнародної наукової конференції, що проіндексована у базі даних Scopus:

4. Mykhailo Kotenko, Dmytro Moskalyk, Valeriia Kovach, Viacheslav Osadchyi. Navigating the challenges and best practices in securing microservices architecture. Proceedings of the Cybersecurity Providing in Information and Telecommunication Systems II, Kyiv, Ukraine, October 26, 2024 (online). Edited by Volodymyr Sokolov, Vasyl Ustimenko, Tamara Radivilova, Mariya Nazarkevych. CEUR Workshop Proceedings, Vol. 3826, 2024. PP.1–16. URL: <https://ceur-ws.org/Vol-3826/paper1.pdf>

Публікації в матеріалах наукових конференцій:

5. Москалик Д.О., Антонюк Д.С. Вплив архітектури програмної системи на продуктивність розробки програмного забезпечення. *Інформаційні технології в освіті та науці* : Збірник наукових праць III Міжнародної науково-практичної конференції, м. Мелітополь, 25-26 травня 2023 р. / Мелітопольський державний педагогічний університет ім. Богдана Хмельницького. Мелітополь, 2023. С. 491-495. URL: <https://drive.google.com/file/d/1wcYwBHxWECzv9TT3-SIDBYUR4Z8eVURI/view>
6. Москалик Д.О., Антонюк Д.С. Вплив різновидів практики власності коду на процес розробки програмного забезпечення. *Комп'ютерні технології: інновації, проблеми, рішення* : тези доп. V Всеукраїнської науково-технічної конференції, м. Житомир, 01-02 грудня 2022 р. / Державний університет «Житомирська політехніка». Житомир, 2022. С. 32-33. URL: <https://conf.ztu.edu.ua/wp-content/uploads/2023/02/32.pdf>
7. Москалик Д.О., Антонюк Д.С. Підхід до моделювання процесу розробки програмного забезпечення для дослідження впливу архітектури на продуктивність розробки. *Тенденції розвитку технологій в автоматизації, електроенергетиці, приладобудуванні та робототехніці* : матеріали Всеукраїнської науково-практичної on-line конференції здобувачів вищої освіти і молодих учених, присвяченої Дню науки, м. Житомир, 13 травня 2024 р. / Державний університет «Житомирська політехніка». Житомир, 2024. С. 81. URL: <https://conf.ztu.edu.ua/wp-content/uploads/2024/06/sekcija-4.pdf>
8. Москалик Д.О., Антонюк Д.С. Моделювання зв'язності вихідного коду в симуляціях процесу розробки програмного забезпечення. *Сучасні інформаційні технології та системи в управлінні* [Електронний ресурс] : збірник матеріалів VI Міжнародної науково-практичної конференції молодих вчених, аспірантів і студентів, м. Київ, 10-11 квітня 2025 р. – Київ: КНЕУ, 2025. С. 314-315. URL: https://kneu.edu.ua/userfiles//Faculty_of_Information_Systems_and_Technology/kaf%20ise/kmv/ZbD196rnik_tez_2025.pdf

9. Москалик Д.О., Антонюк Д.С. Аналіз розподілу складності задач при розробці програмного забезпечення з відкритим вихідним кодом. *Сучасні комп'ютерні системи та мережі в управлінні* : матеріали VI Всеукраїнської наук.-практ. Інтернет-конф. здобувачів вищої освіти та молодих вчених, м. Хмельницький, м. Херсон, 30 листопада 2023 р. / Херсонський національний технічний університет. Херсон, 2023. С. 31-32. URL: https://kntu.net.ua/ukr/content/download/109941/620832/file/CICT_2023.pdf

Відомості про апробацію результатів дисертації

Масові науково-практичні заходи міжнародного рівня:

1. III Міжнародна науково-практична конференція "Інформаційні технології в освіті та науці" (Україна, м. Мелітополь, Мелітопольський державний педагогічний університет ім. Богдана Хмельницького, 25-26 травня 2023 р.) Форма участі – виступ на пленарному засіданні, публікація матеріалів на тему «Вплив архітектури програмної системи на продуктивність розробки програмного забезпечення».

2. Cybersecurity Providing in Information and Telecommunication Systems II (Україна, м. Київ, Київський столичний університет імені Бориса Грінченка, 26 жовтня 2024 р.) Форма участі – публікація матеріалів на тему «Navigating the challenges and best practices in securing microservices architecture».

3. VI Міжнародна науково-практична конференція молодих вчених, аспірантів і студентів "Сучасні інформаційні технології та системи в управлінні" (Україна, м. Київ, Київський національний економічний університет імені Вадима Гетьмана, 10-11 квітня 2025 р.) Форма участі – публікація матеріалів на тему «Моделювання зв'язності вихідного коду в симуляціях процесу розробки програмного забезпечення».

Масові науково-практичні заходи всеукраїнського рівня:

1. V Всеукраїнська науково-технічна конференція "Комп'ютерні технології: інновації, проблеми, рішення" (Україна, м. Житомир, Державний університет "Житомирська політехніка", 01-02 грудня 2022 р.) Форма участі – публікація матеріалів на тему «Вплив різновидів практики власності коду на процес розробки програмного забезпечення».

2. Всеукраїнська науково-практична інтернет-конференція здобувачів вищої освіти та молодих вчених "Сучасні комп'ютерні системи та мережі в

управлінні" (Україна, м. Херсон, Херсонський національний технічний університет, 30 листопада 2023 р.) Форма участі – публікація матеріалів на тему «Аналіз розподілу складності задач при розробці програмного забезпечення з відкритим вихідним кодом».

3. Всеукраїнська науково-практична on-line конференція здобувачів вищої освіти і молодих учених, присвячена Дню науки, "Тенденції розвитку технологій в автоматизації, електроенергетиці, приладобудуванні та робототехніці" (Україна, м. Житомир, Державний університет "Житомирська політехніка", 13 травня 2024 р.) Форма участі – публікація матеріалів на тему «Підхід до моделювання процесу розробки програмного забезпечення для дослідження впливу архітектури на продуктивність розробки».

Масові науково-практичні заходи регіонального рівня:

1. Засідання міжкафедрального науково-методологічного семінару для молодих учених Державного університету "Житомирська політехніка" (Україна, м. Житомир, Державний університет "Житомирська політехніка", 21 січня 2026 р.). Форма участі – виступ з презентацією за результатами дисертаційного дослідження на тему: «Моделювання ефективності розробки програмного забезпечення в залежності від типу архітектури серверних програмних систем».

Довідки про впровадження результатів дисертації



VISEVEN EUROPE LLC
Tax Number: 38428784
Peremohy St.99, Zhytomyr 10003, Ukraine
+380 412 425 904
finance@viseven.com

24-Є/26

ДОВІДКА

про впровадження результатів наукового дослідження

Москалика Дмитра Олександровича

на тему «Моделювання ефективності розробки програмного забезпечення в залежності від типу

архітектури серверних програмних систем»

на здобуття наукового ступеня доктора філософії

за спеціальністю 121 Інженерія програмного забезпечення

Дана довідка підтверджує, що науково-теоретичні та практичні результати дисертаційного дослідження на здобуття наукового ступеня доктора філософії за спеціальністю 121 Інженерія програмного забезпечення МОСКАЛИКА Д.О. проходили апробацію в роботі ТОВ «Візевен Європа». Знайдені оптимальні параметри процесів розробки було апробовано при плануванні та організації процесу розробки програмної системи компанії, що дозволило підвищити ефективність розробки шляхом максимізації продуктивності команд. Програмні симуляції, розроблені в рамках дисертаційного дослідження, було використано для моделювання процесів розробки в контексті поточної програмної архітектури з метою ідентифікації вузьких місць у виробничому процесі. Застосування запропонованих математичних моделей прогнозування ефективності розробки отримало позитивні відгуки від технічних керівників та архітекторів програмного забезпечення компанії.

Зростання складності сучасних серверних програмних систем створює виклики перед ІТ-компаніями щодо обґрунтованого вибору архітектурних рішень та оптимізації процесів розробки, що вимагає застосування нових підходів до моделювання та прогнозування ефективності. Тому запропоновані в роботі МОСКАЛИКА Д.О. математичні моделі та програмні засоби дозволяють частково вирішити дану проблему та підтверджують важливість отриманих наукових результатів та їх практичність.

Заступник директора з операційної діяльності
ТОВ «Візевен Європа»



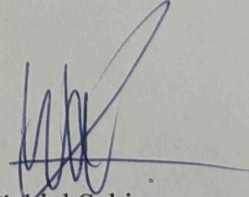
Василенко В'ячеслав

CERTIFICATE
of implementation of research results by
Moskalyk Dmytro Oleksandrovych
on the topic "Modelling software development efficiency based on the server software
architecture types"
submitted for the degree of Doctor of Philosophy (PhD)
in the field of 121 Software Engineering

This certificate confirms that the theoretical and practical outcomes of the dissertation by Moskalyk Dmytro Oleksandrovych, entitled "Modelling software development efficiency based on the server software architecture types" were validated at Sana Commerce. The mathematical models developed for predicting software development efficiency were applied to support decision-making when choosing between service-oriented and modular monolithic architectural approaches for our new server-side software project. The use of these models provided additional evidence-based justification for the selected architectural solution and enabled an assessment of whether transitioning from the existing architecture to an alternative would be worthwhile. Additionally, software-based development process simulations created as part of the research were used to conduct an in-depth analysis of the development workflow, identify bottlenecks, and further improve the development team's productivity.

The significance of the research results obtained by Moskalyk D.O. stems from the growing need to enhance the efficiency of server software development and to ensure that architectural decisions are grounded in quantitative evidence. Accordingly, the models, algorithms, and software tools proposed by the candidate are highly relevant, timely, and practically valuable.

Managing Partner
Sana Commerce


Michiel Schipperus

Звіт подібності дисертації

Москалика Дмитра Олександровича

«Моделювання ефективності розробки програмного забезпечення в залежності
від типу архітектури серверних програмних систем»

Метадані

ДОКУМЕНТ
Заголовок
Москалик_Дисертація_20260407

Автор
Дмитро Москалик

Науковий керівник / Експерт
Zhytomyr Polytechnic

ІД документу
333547197

ОРГАНІЗАЦІЯ
Назва організації
State University "Zhytomyr Polytechnic"

підрозділ
State University "Zhytomyr Polytechnic"

ЗВІТ
Дата звіту
4/7/2026

Дата редагування
4/7/2026

Обсяг знайдених подібностей

Коефіцієнт подібності визначає, який відсоток тексту по відношенню до загального обсягу тексту було знайдено в різних джерелах. Зверніть увагу, що високі значення коефіцієнта не автоматично означають плагіат. Звіт має аналізувати компетентна / уповноважена особа.

0.89%

4.95%

КП 1

25

Довжина фрази для коефіцієнта подібності 2

0.00%

4.01%

КП 2

32456

Кількість слів

256413

Кількість символів

Тривога

У цьому розділі ви знайдете інформацію щодо текстових спотворень. Ці спотворення в тексті можуть говорити про **МОЖЛИВІ** маніпуляції в тексті. Спотворення в тексті можуть мати навмисний характер, але частіше характер технічних помилок при конвертації документа та його збереженні, тому ми рекомендуємо вам підходити до аналізу цього модуля відповідально. У разі виникнення запитань, просимо звертатися до нашої служби підтримки.

Заміна букв		7
Інтервали		0
Мікропробіли		37
Білі знаки		0
Парафрази (SmartMarks)		31

Джерела

Нижче наведений список джерел. В цьому списку є джерела із різних баз даних. Колір тексту означає в якому джерелі він був знайдений. Ці джерела і значення Коефіцієнту Подібності не відображають прямого плагіату. Необхідно відкрити кожне джерело і проаналізувати зміст і правильність оформлення джерела.

10 найдовших фраз

Колір тексту

#	НАЗВА ТА АДРЕСА ДЖЕРЕЛА URL (НАЗВА БАЗИ)	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)
1	https://actualproblems.dp.ua/index.php/APAIT/article/download/317/288	349 (1.08 %)

2	https://actualproblems.dp.ua/index.php/APAIT/article/download/317/288	175 (0.54 %)
3	https://actualproblems.dp.ua/index.php/APAIT/article/download/317/288	111 (0.34 %)
4	https://actualproblems.dp.ua/index.php/APAIT/article/download/317/288	109 (0.34 %)
5	https://actualproblems.dp.ua/index.php/APAIT/article/download/317/288	100 (0.31 %)
6	https://actualproblems.dp.ua/index.php/APAIT/article/download/317/288	96 (0.30 %)
7	https://actualproblems.dp.ua/index.php/APAIT/article/download/317/288	92 (0.28 %)
8	https://actualproblems.dp.ua/index.php/APAIT/article/download/317/288	91 (0.28 %)
9	https://actualproblems.dp.ua/index.php/APAIT/article/download/317/288	57 (0.18 %)
10	https://actualproblems.dp.ua/index.php/APAIT/article/download/317/288	56 (0.17 %)
Програма обміну базами даних (0.00 %)		
#	ЗАГОЛОВОК	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)
Інтернет (4.95 %)		
#	ДЖЕРЕЛО URL	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)
1	https://actualproblems.dp.ua/index.php/APAIT/article/download/317/288	1319 (15) (4.06 %)
2	http://lib.iitta.gov.ua/715709/1/dis_15.pdf	168 (14) (0.52 %)
3	https://conf.ztu.edu.ua/wp-content/uploads/2024/02/zhytomyrska-politehnika_konferenciyi-molodyh-uchenyh_-2024.pdf	24 (2) (0.07 %)
4	https://eir.kntu.net.ua/jspui/bitstream/123456789/1683/1/%D0%A8%D0%B0%D0%BB%D1%8C%20%D0%84.%20%D0%A1.%20%D0%9F%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D1%8F%20%D1%82%D0%B0%20%D0%BA%D1%80%D0%B8%D1%82%D0%B8%D1%87%D0%BD%D0%B8%D0%B9%20%D0%B0%D0%BD%D0%B0%D0%BB%D1%96%D0%B7%20%D1%96%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%86%D1%96%D0%B9%D0%BD%D0%BE%D1%97%20%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B8%20%D0%B7%20%D0%BC%D0%B5%D1%82%D0%BE%D1%8F%20%D0%BF%D1%96%D0%B4%D0%B2%D0%B8%D1%89%D0%B5%D0%BD%D0%BD%D1%8F%20%D0%B5%D1%84%D0%B5%D0%BA%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D1%81%D1%82%D1%96%20%D1%83%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D1%96%D0%BD%D0%BD%D1%8F%20%20...pdf	23 (1) (0.07 %)
5	https://mydisser.com/ru/catalog/view/14627.html	22 (2) (0.07 %)
6	https://openedu.kubg.edu.ua/journal/index.php/openedu/article/download/211/pdf/	16 (1) (0.05 %)
7	https://svr.pnu.edu.ua/wp-content/uploads/sites/5/2020/11/%D0%9F%D0%BE%D0%B3%D1%83%D1%82%D1%8C_%D0%9A%D0%B0%D0%BB%D0%B0%D1%83%D1%80.pdf	14 (1) (0.04 %)
8	https://lib.iitta.gov.ua/id/eprint/710849/1/aref_Antoniuk_Printed.pdf	10 (1) (0.03 %)
9	https://referatu.net.ua/referats/7569/170028	7 (1) (0.02 %)
10	http://lib.pnu.edu.ua:8080/bitstream/123456789/16962/1/%D0%90%D0%B2%D1%82%D0%BE%D1%80%D0%B5%D1%84%D0%B5%D1%80%D0%B0%D1%82_%D0%9A%D0%BE%D1%81%D1%82%D1%8E%D0%BA%20%D0%9C.%D0%9C..pdf	5 (1) (0.02 %)
 Список прийнятих фрагментів		

#	ЗМІСТ	КІЛЬКІСТЬ ОДНАКОВИХ СЛІВ (ФРАГМЕНТІВ)
	https://actualproblems.dp.ua/index.php/APAIT/art... 	1319 (4.06%)