МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ДНІПРОВСЬКА ПОЛІТЕХНІКА»

Кваліфікаційна наукова
праця на правах рукопису

СТАРОДУБСЬКИЙ ІГОР ПЕТРОВИЧ

УДК 004.4:004.382

ДИСЕРТАЦІЯ

МЕТОДИ ТА ЗАСОБИ ПЕРЕНОСИМОСТІ ПРОГРАМ ТА ПРОГРАМНИХ СИСТЕМ НА РІЗНІ ОБЧИСЛЮВАЛЬНІ ПЛАТФОРМИ

122 – Комп’ютерні науки

12 – Інформаційні технології

Подається на здобуття наукового ступеня **доктора філософії**

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

Науковий керівник: **БЕРДНИК Михайло Геннадійович**,
доктор технічних наук, доцент

Дніпро – 2026

АНОТАЦІЯ

Стародубський І.П. Методи та засоби переносимості програм та програмних систем на різні обчислювальні платформи. – Рукопис.

Дисертація на здобуття наукового ступеня доктора філософії за спеціальністю 122 – Комп'ютерні науки. – Національний технічний університет «Дніпровська політехніка», Дніпро, 2026.

У дисертаційному дослідженні розв'язана важлива науково-прикладна задача підвищення переносимості та рівня автоматизації програм і програмних систем на різні обчислювальні платформи. Це досягається шляхом розробки методів адаптивної контейнеризації та управління ресурсами з використанням машинного навчання. Необхідність впровадження цих методів зумовлена постійним ускладненням програмних архітектур, стрімким розвитком гетерогенних апаратних середовищ, зростанням кількості операційних систем, середовищ виконання та моделей розгортання програмного забезпечення.

Дисертація складається зі вступу, чотирьох розділів, висновків, списку використаних джерел та додатків. Повний обсяг дисертації – 215 сторінок; список використаних джерел містить 115 найменувань, 10 додатків. Робота проілюстрована 17 рисунками та містить 9 таблиць.

У вступі обґрунтовано актуальність теми дисертаційного дослідження, визначено мету та сформульовано основні завдання роботи, окреслено об'єкт і предмет дослідження, наведено методи наукових досліджень, що застосовувалися для досягнення поставленої мети. Подано загальну характеристику роботи та її структуру, визначено наукову новизну та практичне значення отриманих результатів, оцінено достовірність основних наукових положень, наведено відомості про апробацію результатів дослідження та публікації автора за темою дисертації.

У першому розділі здійснено системний аналіз теоретичних основ переносимості програм та програмних систем. Розглянуто еволюцію поняття

переносимості, визначено її роль і місце серед показників якості програмного забезпечення, проаналізовано рівні абстракції та критерії оцінювання переносимості. Досліджено основні підходи до перенесення програм між різними обчислювальними платформами, включаючи компіляційні, інтерпретаційні, віртуалізаційні, емуляційні та контейнерні підходи. Наведено порівняльну характеристику стандартів програмування, інтерфейсів прикладного програмування та обчислювальних платформ, що використовуються для забезпечення переносимості. Виявлено ключові обмеження існуючих рішень, пов’язані з платформозалежністю вихідного коду, складністю підтримки багатоплатформних програмних систем і недостатнім рівнем автоматизації процесів перенесення.

У другому розділі науково обґрунтовано та розроблено метод адаптивної контейнеризації з інтеграцією штучного інтелекту. Метод поєднує контейнерні технології з алгоритмами машинного навчання для автоматизованого аналізу апаратно-програмного середовища, прогнозування навантаження та динамічного налаштування параметрів контейнерів у реальному часі. Розроблено архітектуру методу, визначено функціональні компоненти та механізми їх взаємодії, описано алгоритмічні підходи до оптимізації використання обчислювальних ресурсів. Обґрунтовано переваги методу порівняно з класичними підходами контейнеризації, зокрема зменшення потреби у ручному налаштуванні, підвищення стабільності, продуктивності, масштабованості та адаптивності програмних систем, а також можливість ефективного застосування методу у середовищах з обмеженими ресурсами.

У третьому розділі наведено результати практичної реалізації та впровадження методу. Описано програмний прототип, архітектуру програмного забезпечення, результати експериментальних досліджень і аналізу продуктивності. Розглянуто приклади застосування методу в реальних обчислювальних середовищах, у тому числі у хмарних та мультиплатформних системах. Надано рекомендації щодо інтеграції методу у

виробничі процеси розробки та експлуатації програмного забезпечення. Проведено оцінку економічної ефективності впровадження, яка підтверджує доцільність використання підходу для зниження витрат, підвищення якості, надійності та довготривалої підтримуваності програмних систем.

У четвертому розділі проведено порівняльний аналіз та обґрунтовано доцільність комплексного застосування методів адаптації на рівнях компіляції (build-time) та виконання (run-time). У межах цього напряму розроблено метод адаптивної компіляції на основі генетичних алгоритмів, що дозволяє автоматизувати пошук оптимальних конфігурацій параметрів трансляції коду під специфіку цільових апаратних платформ (ARM, Intel Xeon, GPU). Задачу компіляції формалізовано як задачу багатокритеріальної дискретної оптимізації. Експериментально підтверджено кількісний приріст ефективності програм на різних платформах порівняно зі стандартними рівнями оптимізації. Математично доведено синергетичний ефект від інтеграції обох методів: встановлено, що попередньо оптимізований машинний код знижує базові вимоги програми до ресурсів, дозволяючи інтелектуальному контролеру контейнерів значно ефективніше стабілізувати систему при динамічних змінах навантаження.

Впровадження розроблених в процесі дослідження методів адаптивної контейнеризації з інтеграцією штучного інтелекту, алгоритмів адаптивної компіляції та заснованого на них програмного забезпечення при їх використанні на підприємствах дозволило значно скоротити потребу у ручному налаштуванні параметрів середовищ виконання, зменшити витрати на адаптацію та експлуатацію програм, а також підвищити стабільність, продуктивність і масштабованість багатоплатформних програмних систем при динамічних змінах навантаження.

Ключові слова: переносимість програм, контейнеризація, віртуалізація, штучний інтелект, машинне навчання, нейронні мережі, глибоке навчання, великі дані, інформаційні технології, системний аналіз,

прогнозування, random forest, вилучення інформації, імовірнісні оцінки, методологія.

SUMMARY

Starodubskyi I.P. Methods and Tools for the Portability of Programs and Software Systems Across Different Computing Platforms. – Manuscript.

Thesis for scientific degree of Doctor of Philosophy in the specialty 122 – Computer Science. – Dnipro University of Technology, Dnipro, 2026.

In the dissertation research, a scientific and applied problem of improving portability and increasing the level of automation of software and software systems across different computing platforms has been solved. This is achieved through the development of methods for adaptive containerization and resource management using machine learning. The need to implement these methods is driven by the continuous increasing complexity of software architectures, the rapid development of heterogeneous hardware environments, the growing number of operating systems, execution environments, and software deployment models.

The dissertation consists of an introduction, four chapters, conclusions, a list of references, and appendices. The total length of the dissertation is 215 pages; the list of references contains 115 entries, and there are 10 appendices. The work is illustrated with 17 figures and contains 9 tables.

The introduction substantiates the relevance of the dissertation topic, defines the aim and formulates the main objectives of the study, outlines the object and subject of research, and presents the scientific methods used to achieve the stated aim. It provides a general description of the work and its structure, identifies the scientific novelty and practical significance of the obtained results, assesses the reliability of the main scientific provisions, and includes information on the approbation of the research results and the author's publications on the dissertation topic.

The first chapter provides a systematic analysis of the theoretical foundations of software and software system portability. The evolution of the concept of portability is examined, its role and place among software quality attributes are defined, and the levels of abstraction and criteria for evaluating

portability are analyzed. The main approaches to porting software between different computing platforms are studied, including compilation-based, interpretation-based, virtualization-based, emulation-based, and container-based approaches. A comparative description of programming standards, application programming interfaces, and computing platforms used to ensure portability is presented. The key limitations of existing solutions are identified, including source code platform dependence, the complexity of maintaining multiplatform software systems, and the insufficient level of automation of porting processes.

The second chapter develops and scientifically substantiates a method of adaptive containerization with artificial intelligence integration, which constitutes the main scientific novelty of the dissertation. The method combines container technologies with machine learning algorithms for automated analysis of the hardware and software environment, workload prediction, and dynamic real-time adjustment of container parameters. The architecture of the method is developed, its functional components and mechanisms of interaction are defined, and algorithmic approaches to optimizing the use of computing resources are described. The advantages of the method compared with classical containerization approaches are substantiated, in particular the reduction of the need for manual configuration, increased stability, performance, scalability, and adaptability of software systems, as well as the possibility of effective application of the method in resource-constrained environments.

The third chapter presents the results of the practical implementation and deployment of the method. A software prototype, the software architecture, and the results of experimental studies and performance analysis are described. Examples of the method's application in real computing environments, including cloud and multiplatform systems, are considered. Recommendations are provided for integrating the method into production processes of software development and operation. An assessment of the economic efficiency of implementation is carried out, confirming the feasibility of using the approach to reduce costs and improve the quality, reliability, and long-term maintainability of software systems.

The fourth chapter presents a comparative analysis and substantiates the expediency of the integrated application of adaptation methods at the compilation level (build-time) and execution level (run-time). Within this direction, a method of adaptive compilation based on genetic algorithms was developed, which makes it possible to automate the search for optimal code translation parameter configurations tailored to the specifics of target hardware platforms (ARM, Intel Xeon, GPU). The compilation task is formalized as a multicriteria discrete optimization problem. A quantitative increase in software efficiency on different platforms compared with standard optimization levels is experimentally confirmed. The synergistic effect of integrating both methods is mathematically proven: it is established that pre-optimized machine code reduces the baseline resource requirements of the software, enabling the intelligent container controller to stabilize the system much more effectively under dynamic workload changes.

The implementation at enterprises of the adaptive containerization methods with artificial intelligence integration, adaptive compilation algorithms, and the software based on them, developed in the course of the research, made it possible to significantly reduce the need for manual configuration of execution environment parameters, decrease the costs of software adaptation and operation, and improve the stability, performance, and scalability of multiplatform software systems under dynamic workload changes.

Keywords: software portability, containerization, virtualization, artificial intelligence, machine learning, neural networks, deep learning, big data, information technology, system analysis, forecasting, random forest, information extraction, probabilistic estimates, methodology.

СПИСОК ОПУБЛІКОВАНИХ ПРАЦЬ АВТОРА ЗА ТЕМОЮ ДИСЕРТАЦІЇ

Публікації у виданнях, включених до переліку наукових фахових видань України:

1. Бердник М.Г., **Стародубський І.П.** (2025). Use of genetic algorithms in adaptive compilers for cross-platform optimization. Radio Electronics, Computer Science, Control. 4 (75), 185-193. DOI: https://doi.org/10.15588/1607-3274-2025-4-16 [Web of Science, Index Copernicus, Google Scholar, Crossref]
2. Бердник М.Г., **Стародубський І.П.** (2025). Self-profiling mechanisms for real-time code compilers. System technologies. 6 (161), 85-95. DOI: https://doi.org/10.34185/1562-9945-5-161-2025-08 [Index Copernicus, Google Scholar, Crossref]
3. Бердник М.Г., **Стародубський І.П.** (2025). Using machine learning methods for automated cloud computing optimization. Інформаційні технології та суспільство, 3 (18), 16-23. DOI: https://doi.org/10.32689/maup.it.2025.3.2 [Index Copernicus, Google Scholar, Crossref]
4. Бердник М.Г., **Стародубський І.П.** (2025). Підвищення переносимості вихідного коду C++ для багатоплатформних обчислювальних систем. Вісник Хмельницького національного університету. Серія: Технічні науки, 4, 27-31. DOI: https://doi.org/10.31891/2307-5732-2025-355-3 [Index Copernicus, Google Scholar, Crossref]
5. Бердник М.Г., **Стародубський І.П**. (2025). Перенесення програм між cpu, gpu, tpu та fpga: виклики та рішення. Information Technology: Computer Science, Software Engineering and Cyber Security, 2, 3-9, DOI: http://dx.doi.org/10.32782/it/2025-2-1 [Index Copernicus, Google Scholar, Crossref]

6. Бердник М.Г., **Стародубський І.П.** (2025). Використання методів машинного навчання для адаптації програмного забезпечення до різних обчислювальних платформ. Information Technology: Computer Science, Software Engineering and Cyber Security, 4, 16-28. DOI: 10.32782/it/2024-4-3 [Index Copernicus, Google Scholar, Crossref]

Публікації у матеріалах наукових конференцій:

1. Бердник М.Г., **Стародубський І.П.** (2025). Метод адаптивної контейнеризації з інтеграцією штучного інтелекту. Проблеми використання інформаційних технологій в освіті, науці та промисловості. XX міжнародна конференція. Part of ISBN: 978-617-8737-34-4 https://pzks.nmu.org.ua/ua/science/2025_fin.pdf
2. Бердник М.Г., **Стародубський І.П.** (2025). Тестові набори, що самовідновлюються: автоматизація підтримки нестабільних тестів. Інформаційні управляючі системи і технології (ІУСТ-ОДЕСА-2025). DOI: 10.36059/978-966-397-531-3 Part of ISBN: 978-966-397-531-3 http://catalog.liha-pres.eu/index.php/liha-pres/catalog/book/413
3. Бердник М.Г., **Стародубський І.П.** (2025). Інструменти для автоматичного генерування абстракцій над платформозалежним кодом у C++ проєктах. Proceedings of I international scientific and practical conference. Part of ISBN: 978-3-954753-01-7 https://sci-conf.com.ua/wp-content/uploads/2025/09/SCIENCE-AND-EDUCATION-SYNERGY-OF-INNOVATION-1-3.09.25.pdf
4. Бердник М.Г., **Стародубський І.П.** (2025). Метрики оцінки переносимості C++ коду у вбудованих системах. Science And Technology: Challenges, Prospects and Innovations. Part of ISBN: 978-4-9783419-4-5 https://sci-conf.com.ua/wp-content/uploads/2025/08/SCIENCE-AND-TECHNOLOGY-CHALLENGES-PROSPECTS-AND-INNOVATIONS-14-16.08.25.pdf

5. Бердник М.Г., **Стародубський І.П.** (2025). Методи автоматизованого виявлення платформозалежного коду в C++ проєктах. Proceedings of IX international scientific and practical conference. Part of ISBN: 978-84-15927-30-3 https://sci-conf.com.ua/wp-content/uploads/2025/08/EUROPEAN-CONGRESS-OF-SCIENTIFIC-DISCOVERY-18-20.08.25.pdf
6. Бердник М.Г., **Стародубський І.П.** (2024). Автоматична адаптація програмного забезпечення до хмарних та периферійних платформ методами машинного навчання. XIX Міжнародна конференція з проблем використання інформаційних технологій в освіті, науці та промисловості. Part of ISBN: 978-966-934-666-7 https://pzks.nmu.org.ua/ua/science/conf2024.pdf
7. Бердник М.Г., **Стародубський І.П.**, Захаров Д.І. (2024). Адаптивні компілятори для переносимості програмного забезпечення до різних обчислювальних платформ. Інформаційне суспільство: технологічні, економічні та технічні аспекти становлення. http://www.konferenciaonline.org.ua/ua/article/id-1878/
8. Бердник М.Г., **Стародубський І.П.**, Захаров Д.І. (2024). Еволюційні операції та особливості їх застосування для вирішення задачі генерації тестових даних. Інформаційне суспільство: технологічні, економічні та технічні аспекти становлення. http://www.konferenciaonline.org.ua/ua/article/id-1657/
9. Бердник М.Г., **Стародубський І.П.**, Захаров Д.І. (2024). Механізм рефлексивного аналізу методів переносимості програм на різні обчислювальні платформи. Інформаційне суспільство: технологічні, економічні та технічні аспекти становлення. http://www.konferenciaonline.org.ua/ua/article/id-1658/
10. Бердник М.Г., **Стародубський І.П.**, Захаров Д.І. (2023). Метод переносимості програм на різні обчислювальні платформи. Scientific Research in the Modern World. Part of ISBN: 978-1-4879-3795-9

https://sci-conf.com.ua/wp-content/uploads/2023/09/SCIENTIFIC-RESEARCH-IN-THE-MODERN-WORLD-21-23.09.23.pdf

11. Бердник М.Г., **Стародубський І.П.**, Захаров Д.І. (2023). Застосування генетичного алгоритму для формування наборів вхідних тестових даних. Інформаційне суспільство: технологічні, економічні та технічні аспекти становлення. http://www.konferenciaonline.org.ua/ua/article/id-1250/

ЗМІСТ

ЗМІСТ 13
ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ 16
ВСТУП 18
РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ ПЕРЕНОСИМОСТІ ПРОГРАМ ТА ПРОГРАМНИХ СИСТЕМ 30
1.1 Теоретичні основи переносимості програм 30
1.2 Критерії за якими можна оцінювати переносимість програм................ 32
1.3 Рівні абстракції відносно переносимості програм 33
1.4 Методи забезпечення переносимості програм 35
1.5 Порівняльна характеристика стандартів, які забезпечують переносимість програм 37
1.6 Порівняльна характеристика платформ, на які здійснюється переносимість програм 42
1.7 Підходи щодо перенесення програм на різні платформи 46
1.8 Порівняння крос-компіляції, віртуалізації та контейнеризації............. 47
1.9 Висновки до першого розділу 49
РОЗДІЛ 2. МЕТОД АДАПТИВНОЇ КОНТЕЙНЕРИЗАЦІЇ З ІНТЕГРАЦІЄЮ ШТУЧНОГО ІНТЕЛЕКТУ 52
2.1 Вступ до методу 52
2.2 Застосування машинного навчання у методі адаптивної контейнеризації 62
2.3 Принципи реалізації адаптивної контейнеризації 69
2.4 Алгоритми та технології адаптивної контейнеризації 72
2.5 Архітектура методу адаптивної контейнеризації 80
2.5.1 Опис компонентів архітектури 83
2.5.2 Взаємодія компонентів та потоки даних 87
2.5.3 Рівні архітектури та межі відповідальності компонентів............. 90
2.5.4 Формалізація керуючого циклу адаптивної контейнеризації 92
2.5.5 Архітектура збору метрик та телеметрії 95
2.5.6 Архітектура інтелектуального модуля прийняття рішень 98
2.5.7 Архітектура застосування керуючих впливів 99
2.6 Висновки до другого розділу.. 102
РОЗДІЛ 3. ПРАКТИЧНА РЕАЛІЗАЦІЯ ТА ВПРОВАДЖЕННЯ МЕТОДУ 106

3.1 Реалізація архітектури програмного прототипу 106
3.2. Реалізація збору метрик та формування вектора-функції m(t) 108
3.3. Формування профілю платформи P та вектора-функції ознак φ(t) 110
3.4. Реалізація інтелектуального модуля оцінювання та прогнозування відхилення переносимості D(t) 113
3.5. Реалізація політики керування π та вибору керуючих впливів u(t) 116
3.6. Реалізація множини допустимих дій U(x,P) та інтеграція з Kubernetes control plane.. 118
3.7. Реалізація замкненого керуючого циклу та часові аспекти керування 120
3.8. Експериментальне середовище та сценарії досліджень..................... 122
3.9. Аналіз експериментальних результатів 125
3.10. Аналіз часової динаміки індикатора відхилення D(t) 134
3.11. Обговорення обмежень методу 138
3.12. Висновки до третього розділу 140
РОЗДІЛ 4. ПОРІВНЯЛЬНИЙ АНАЛІЗ ТА КОМПЛЕКСНЕ ЗАСТОСУВАННЯ МЕТОДІВ АДАПТАЦІЇ НА РІВНЯХ КОМПІЛЯЦІЇ ТА ВИКОНАННЯ 143
4.1 Аналіз рівнів адаптації програмного забезпечення у гетерогенних середовищах 143
4.2 Математична модель та формалізація задачі адаптивної компіляції .. 147
4.3 Алгоритмічна реалізація та етапи функціонування методу 154
Етап 1. Формування початкової популяції конфігурацій компіляції. ... 155
Етап 2. Генерація машинного коду та збір експлуатаційних метрик 155
Етап 3. Обчислення функції пристосованості та оцінювання якості рішень.. 156
Етап 4. Селекція батьківських конфігурацій 156
Етап 5. Формування нового покоління: елітаризм, кросовер та мутація 156
Етап 6. Критерії зупинки та завершення алгоритму 157
4.4 Експериментальне дослідження ефективності методу адаптивної компіляції... 157
4.4.1 Характеристика експериментального стенду та методика вимірювань. 158
4.4.2 Результати пошуку оптимальних конфігурацій............................ 160
4.4.3 Кількісна оцінка приросту продуктивності. 163

4.4.3 Аналіз збіжності алгоритму та стабільності результатів. 164
4.5 Порівняльний аналіз та синергетичний ефект комплексного застосування методів адаптації 167
4.6 Висновки до четвертого розділу.. 169
ВИСНОВКИ 172
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 175
ДОДАТОК А СПИСОК ОПУБЛІКОВАНИХ ПРАЦЬ АВТОРА ЗА ТЕМОЮ ДИСЕРТАЦІЇ.. 187
ДОДАТОК Б ДОКУМЕНТИ ЩОДО ВПРОВАДЖЕННЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕНЬ 191
ДОДАТОК В ЛІСТИНГ ІНІЦІАЛІЗАЦІЇ КЛІЄНТА KUBERNETES.... 197
ДОДАТОК Г ЛІСТИНГ ОТРИМАННЯ МЕТРИК З PROMETHEUS ... 199
ДОДАТОК Д ЛІСТИНГ ФОРМУВАННЯ ВЕКТОРА-ФУНКЦІЇ m(t)... 201
ДОДАТОК Е ЛІСТИНГ ОТРИМАННЯ МЕТАДАНИХ ВУЗЛІВ КЛАСТЕРА 203
ДОДАТОК Ж ЛІСТИНГ ПРОГНОЗУВАННЯ D(t+Δ) ДЛЯ МНОЖИНИ КАНДИДАТНИХ КЕРУЮЧИХ ВПЛИВІВ.. 206
ДОДАТОК З ЛІСТИНГ ФОРМУВАННЯ МНОЖИНИ КАНДИДАТНИХ КЕРУЮЧИХ ВПЛИВІВ З УРАХУВАННЯМ ОБМЕЖЕНЬ СИСТЕМИ ОРКЕСТРАЦІЇ 208
ДОДАТОК И ЛІСТИНГ РЕАЛІЗАЦІЇ ЗАСТОСУВАННЯ ВЕРТИКАЛЬНОГО КЕРУЮЧОГО ВПЛИВУ ШЛЯХОМ ОНОВЛЕННЯ RESOURCE LIMITS КОНТЕЙНЕРА У DEPLOYMENT........................ 210
ДОДАТОК І ЛІСТИНГ РЕАЛІЗАЦІЇ КЕРУЮЧОГО ЦИКЛУ 213

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

API	Application Programming Interface — інтерфейс прикладного програмування, що визначає спосіб взаємодії програмних компонентів між собою або з операційною системою.
ARM	Advanced RISC Machine — сімейство енергоефективних процесорних архітектур RISC-типу, широко застосовуваних у мобільних, вбудованих та периферійних обчислювальних системах.
Autoscaling	Механізм автоматичної зміни кількості ресурсів або екземплярів сервісів залежно від інтенсивності навантаження.
Baseline	Базовий режим керування або конфігурація системи, що використовується для порівняльного аналізу.
Burst workload	Сценарій навантаження, що характеризується короткочасними різкими сплесками інтенсивності обчислень або запитів.
CI/CD	Continuous Integration / Continuous Delivery — сукупність практик автоматизації збирання, тестування та розгортання програмного забезпечення.
Cloud platform	Хмарна обчислювальна платформа, що надає обчислювальні ресурси та сервіси через мережу з можливістю автоматизованого масштабування.
Container	Ізольоване середовище виконання програмного забезпечення, що містить прикладний код та його залежності й використовує спільне ядро операційної системи.
Containerization	Метод упаковки та ізоляції програм у контейнерах з метою забезпечення відтворюваності середовища виконання та переносимості.

DevOps	Підхід до розробки програмного забезпечення, що поєднує процеси розробки та експлуатації з високим рівнем автоматизації.
Docker	Контейнерна платформа для створення, розгортання та виконання контейнеризованих додатків.
Heterogeneous platform	Обчислювальне середовище, що складається з вузлів з різними апаратними архітектурами, ресурсами або програмними конфігураціями.
Kubernetes	Система оркестрації контейнерів, що забезпечує автоматизоване розгортання, масштабування та керування контейнеризованими додатками.
Multi-cloud	Підхід до розгортання програмних систем із використанням декількох хмарних платформ одночасно.
Node	Обчислювальний вузол у кластері Kubernetes, на якому виконуються контейнери.
Orchestration	Автоматизоване керування розгортанням, масштабуванням та взаємодією контейнерів у розподіленому середовищі.
Portability	Переносимість — здатність програмної системи зберігати коректну та стабільну поведінку при зміні обчислювальної платформи без модифікації програмного коду.
Prometheus	Система моніторингу та зберігання часових рядів експлуатаційних метрик.
RISC-V	Відкрита процесорна архітектура типу RISC, орієнтована на забезпечення переносимості та масштабованості апаратних рішень.
Workload	Сукупність запитів або обчислювальних задач, що виконуються програмною системою.

ВСТУП

Актуальність теми. Сучасний етап розвитку інформаційних технологій характеризується стрімким зростанням складності програмних систем, масштабним розширенням спектра обчислювальних платформ та активним впровадженням гетерогенних обчислювальних середовищ. Програмні системи все частіше повинні функціонувати одночасно на різних апаратних архітектурах, включаючи центральні процесори загального призначення, графічні процесори, спеціалізовані обчислювальні прискорювачі (TPU, FPGA), а також у середовищах з обмеженими обчислювальними ресурсами, таких як вбудовані та периферійні системи. У зв’язку з цим досягнення переносимості програмного забезпечення стає однією з ключових проблем сучасної програмної інженерії та комп’ютерних наук. Сучасні інформаційні технології вимагають нових парадигм для досягнення максимальної адаптивності та масштабованості архітектур.

Переносимість програм та програмних систем визначається здатністю програмного забезпечення коректно виконуватися на різних обчислювальних платформах без необхідності суттєвої модифікації вихідного коду або створення окремих платформозалежних реалізацій. Високий рівень переносимості дозволяє значно зменшити витрати на розробку, супровід і тестування програмного забезпечення, підвищити його надійність та спростити процеси впровадження і масштабування. Особливої актуальності ця властивість набуває в умовах швидкої еволюції апаратних архітектур, операційних систем і середовищ виконання, коли програмні продукти повинні залишатися актуальними протягом тривалого часу.

Сучасні програмні системи дедалі частіше розгортаються у хмарних, мультихмарних і розподілених середовищах, де вимоги до переносимості тісно поєднуються з вимогами до масштабованості, ефективного використання ресурсів, стійкості до змін навантаження та високого рівня автоматизації процесів управління. У таких умовах класичні підходи до

забезпечення переносимості, засновані виключно на стандартизованих мовах програмування, крос-компіляції або віртуалізації, часто виявляються недостатніми через їх статичний характер та обмежену здатність адаптуватися до динамічних змін середовища виконання.

Широке впровадження контейнерних технологій стало важливим кроком у напрямку підвищення переносимості програмного забезпечення, оскільки контейнеризація дозволяє уніфікувати середовище виконання та ізолювати програму разом з її залежностями. Проте більшість існуючих контейнерних рішень ґрунтується на заздалегідь визначених конфігураціях і не враховує особливостей конкретної апаратно-програмної платформи, поточного стану системи або змінних умов навантаження. Це суттєво обмежує ефективність контейнеризації в гетерогенних обчислювальних середовищах, особливо при використанні систем з обмеженими ресурсами або при обробці нерівномірних потоків обчислювальних задач.

Окремої уваги заслуговує проблема автоматизації процесів управління середовищами виконання програм. У реальних умовах експлуатації програмні системи функціонують у динамічному середовищі, де доступні обчислювальні ресурси, інтенсивність навантаження та вимоги до продуктивності можуть суттєво змінюватися з часом. Відсутність механізмів інтелектуальної адаптації призводить до неефективного використання ресурсів, зниження продуктивності або нестабільної роботи програмних систем.

У цьому контексті перспективним напрямом є застосування методів штучного інтелекту та машинного навчання для автоматизованого аналізу характеристик середовища виконання та динамічного налаштування параметрів програмних систем. Інтеграція таких методів із контейнерними технологіями відкриває можливість створення адаптивних середовищ виконання, здатних автоматично оптимізувати використання ресурсів та забезпечувати стабільну роботу програм незалежно від конкретної платформи. Завдяки цьому подібні адаптивні середовища фактично

перетворюються на повноцінні інтелектуальні системи, які здатні до саморегуляції.

Наведений аналіз сучасного стану проблеми переносимості програм та програмних систем дозволяє зробити висновок про актуальність розв’язання науково-прикладної задачі розробки методів і засобів забезпечення переносимості програмного забезпечення на різні обчислювальні платформи шляхом застосування адаптивної контейнеризації з інтеграцією штучного інтелекту, що зумовлює вибір теми дисертаційного дослідження.

Зв’язок роботи з науковими програмами, планами, темами. Дисертаційна робота виконана відповідно до наукових напрямів досліджень кафедри програмного забезпечення комп’ютерних систем НТУ «Дніпровська політехніка» «Високопродуктивні багатопроцесорні системи: особливості конструювання, дослідження оцінок ефективності, застосування до розв'язування прикладних задач» (Е-340, державний реєстраційний номер 0121U113718, 2025р). Тематика дослідження пов’язана з виконанням науково-дослідних робіт, спрямованих на розробку методів, моделей і технологій створення багатоплатформних програмних систем, підвищення ефективності використання обчислювальних ресурсів та автоматизацію процесів управління програмним забезпеченням у складних комп’ютерних системах.

Мета і задачі дослідження. Метою дисертаційної роботи є розв’язання важливої науково-прикладної задачі підвищення переносимості та рівня автоматизації програм і програмних систем на різні обчислювальні платформи.

Для досягнення поставленої мети у дисертаційній роботі необхідно вирішити такі задачі:

1. дослідити сучасний стан проблеми переносимості програм та програмних систем, проаналізувати існуючі теоретичні підходи, методи і засоби забезпечення переносимості;

2. здійснити аналіз впливу різних апаратних архітектур, операційних систем і моделей розгортання на процес перенесення програмного забезпечення;
3. дослідити сучасні технології віртуалізації, контейнеризації, крос-компіляції та автоматизації процесів розгортання програм;
4. розробити метод адаптивної контейнеризації з інтеграцією штучного інтелекту для динамічного налаштування параметрів середовища виконання програм;
5. здійснити практичну реалізацію методу та провести експериментальну перевірку його ефективності у різних обчислювальних середовищах.

Об'єкт дослідження – процеси розробки, перенесення, розгортання та експлуатації програм і програмних систем у гетерогенних обчислювальних середовищах.

Предмет дослідження – методи, моделі, алгоритми та програмні засоби забезпечення переносимості програмного забезпечення на різні обчислювальні платформи.

Методи дослідження. Методологічну та теоретичну основу дисертаційної роботи становлять методи системного аналізу, теорія алгоритмів, принципи програмної інженерії, методи віртуалізації та контейнеризації, алгоритми машинного навчання і штучного інтелекту, а також методи експериментального аналізу продуктивності програмних систем у різних середовищах виконання.

Наукова новизна одержаних результатів полягає в тому, що:

Вперше:

– розроблено метод адаптивної контейнеризації з інтеграцією штучного інтелекту, який забезпечує автоматизоване та динамічне налаштування параметрів середовищ виконання програм відповідно до характеристик апаратно-програмної платформи, поточного стану системи та змінних умов навантаження. Метод, на відміну від існуючих підходів, поєднує контейнерні технології з алгоритмами машинного навчання, що

дозволяє здійснювати інтелектуальну адаптацію параметрів контейнерів у реальному часі без необхідності ручного втручання;

- розроблено архітектуру інтелектуального управління контейнеризованими програмами у гетерогенних обчислювальних середовищах, яка передбачає взаємодію модулів аналізу середовища, систем моніторингу, алгоритмів штучного інтелекту та засобів оркестрації контейнерів. Архітектура забезпечує узгоджену роботу компонентів системи та дозволяє автоматично оптимізувати використання обчислювальних ресурсів у багатоплатформних середовищах;
- розроблено інформаційну технологію застосування машинного навчання для прогнозування потреб програм у ресурсах обчислювальної платформи на основі аналізу історичних даних та поточних метрик продуктивності, що дозволяє реалізувати проактивне управління контейнеризованими середовищами виконання та зменшити ймовірність деградації продуктивності або перевантаження системи.

Вдосконалено:

- інформаційну технологію забезпечення переносимості програмних систем шляхом поєднання контейнерних технологій із методами машинного навчання, що дозволило перейти від статичних конфігурацій середовищ виконання до адаптивних, здатних враховувати специфіку апаратної архітектури, операційної системи та умов експлуатації програмного забезпечення;
- інформаційну технологію автоматизації процесів розгортання, масштабування та супроводу програмних систем у мультиплатформних і мультихмарних середовищах за рахунок інтеграції інтелектуальних механізмів прийняття рішень у процеси оркестрації контейнерів. Це забезпечує підвищення ефективності використання ресурсів та зниження витрат на експлуатацію програмного забезпечення;

- метод управління контейнеризованими програмами у середовищах з обмеженими ресурсами, зокрема у вбудованих та периферійних обчислювальних системах, шляхом використання адаптивних алгоритмів оптимізації, що дозволяють забезпечити стабільну роботу програм за умов обмеженої обчислювальної потужності.

Набули подальшого розвитку:

- методи забезпечення переносимості програм та програмних систем у гетерогенних обчислювальних середовищах за рахунок використання інтелектуальних механізмів аналізу та адаптації середовищ виконання, що розширює можливості застосування контейнеризації у складних багатоплатформних системах;
- інформаційні технології щодо використання штучного інтелекту для оптимізації продуктивності, стабільності та масштабованості програмних систем, які функціонують у динамічних середовищах з нерівномірним навантаженням та змінними вимогами до ресурсів;
- методи автоматизованого управління середовищами виконання програм у мультихмарних інфраструктурах, що сприяють зниженню залежності від конкретного постачальника хмарних сервісів та підвищенню гнучкості розгортання програмного забезпечення.

Практичне значення одержаних результатів полягає у можливості їх безпосереднього використання при розробці, розгортанні та експлуатації багатоплатформного програмного забезпечення в гетерогенних обчислювальних середовищах. Запропоновані методи та засоби спрямовані на підвищення переносимості програм і програмних систем, зменшення витрат на їх адаптацію до різних обчислювальних платформ та підвищення ефективності використання обчислювальних ресурсів.

Розроблений метод адаптивної контейнеризації з інтеграцією штучного інтелекту може бути використаний при створенні програмних систем, що функціонують у хмарних, мультихмарних, розподілених, периферійних та вбудованих середовищах. Практичне застосування методу дозволяє

автоматизувати процеси налаштування середовищ виконання програм, зменшити залежність від ручної конфігурації та знизити ризик помилок, пов’язаних з некоректним налаштуванням контейнерів.

Даний підхід забезпечує підвищення стабільності та продуктивності програмних систем за рахунок динамічної адаптації параметрів контейнерів до характеристик апаратно-програмної платформи та змінних умов навантаження. Це особливо важливо для систем, що працюють у режимі реального часу або обробляють нерівномірні потоки даних, де ефективне використання ресурсів безпосередньо впливає на якість надання сервісів та надійність роботи програмного забезпечення.

Практичне значення результатів дисертаційної роботи полягає також у можливості застосування запропонованих методів у середовищах з обмеженими обчислювальними ресурсами, зокрема у вбудованих системах та IoT-платформах. Адаптивне налаштування параметрів контейнерів дозволяє оптимізувати споживання процесорного часу, пам’яті та енергетичних ресурсів, що є критичним для таких класів систем.

Отримані результати можуть бути використані при впровадженні сучасних підходів DevOps та CI/CD, оскільки розроблений метод інтегрується з існуючими системами оркестрації контейнерів і автоматизації розгортання програмного забезпечення Це сприяє скороченню часу виведення програмних продуктів у промислову експлуатацію, підвищенню повторюваності середовищ виконання та зниженню витрат на супровід програмних систем.

Практичне значення одержаних результатів полягає також у можливості використання запропонованих рішень при розробці програмного забезпечення для фінансових, промислових, наукових та інженерних застосувань, де важливими є висока надійність, масштабованість і переносимість програмних систем між різними обчислювальними платформами. Даний підхід дозволяє забезпечити узгоджену роботу

програмних компонентів у різних середовищах без необхідності створення окремих платформозалежних реалізацій.

Крім того, результати дисертаційної роботи можуть бути використані у навчальному процесі закладів вищої освіти при викладанні дисциплін з програмної інженерії, комп’ютерних наук, хмарних обчислень та розподілених систем. Розроблені методи та архітектурні рішення можуть слугувати основою для навчальних прикладів, лабораторних робіт і курсових проєктів, спрямованих на вивчення проблем переносимості програмного забезпечення та сучасних контейнерних технологій.

Таким чином, практичне значення одержаних результатів визначається можливістю їх застосування як у прикладних задачах розробки та експлуатації програмних систем, так і у науково-освітній діяльності, що підтверджує доцільність і практичну спрямованість дисертаційного дослідження.

Впровадження одержаних результатів. Практичне значення результатів підтверджується актами впровадження результатів дисертаційної роботи у діяльність низки підприємств та установ, зокрема: ТОВ НВП «Агропромавтоматизація», ДЕРЖАВНЕ НАУКОВО-ВИРОБНИЧЕ ПІДПРИЄМСТВО «Ельдорадо», іноземного підприємство «SoftRequest LTD», ТОВ «Біологічні активні добавки», а також у навчальний процес Національного технічного університету «Дніпровська політехніка» (кафедра програмного забезпечення комп’ютерних систем), зокрема при викладанні дисциплін «Машинне навчання» та «Модифікація та тестування програмного забезпечення» для бакалаврів та магістрів.

Особистий внесок здобувача. Усі наукові результати, викладені у дисертаційній роботі, отримані особисто здобувачем. Дисертаційна робота є завершеним самостійним науковим дослідженням, у якому автором сформульовано наукову проблему, визначено мету і завдання дослідження, обґрунтовано вибір методів дослідження, отримано та проаналізовано результати, що мають наукову новизну та практичну значущість.

Особистий внесок здобувача полягає у виконанні повного циклу наукового дослідження, включаючи аналіз сучасного стану проблеми переносимості програм і програмних систем, систематизацію існуючих методів і засобів забезпечення переносимості, формулювання вимог до адаптивних підходів управління середовищами виконання програм, а також у розробці та науковому обґрунтуванні методу адаптивної контейнеризації з інтеграцією штучного інтелекту.

Здобувачем особисто розроблено концепцію адаптивної контейнеризації, визначено архітектуру методу, склад і функціональне призначення його основних компонентів, а також принципи їх взаємодії. Автором сформульовано підходи до використання алгоритмів машинного навчання для аналізу характеристик апаратно-програмних платформ, прогнозування навантаження та динамічного налаштування параметрів контейнерів у реальному часі. Усі алгоритмічні рішення, запропоновані в роботі, розроблені здобувачем самостійно.

Особистий внесок здобувача також полягає у програмній реалізації запропонованого методу, розробці програмного прототипу, налаштуванні середовищ виконання, підготовці експериментальних сценаріїв та проведенні експериментальних досліджень. Автором виконано аналіз отриманих результатів, здійснено їх інтерпретацію та сформульовано висновки щодо ефективності розробленого методу в різних обчислювальних середовищах.

У наукових публікаціях, виконаних у співавторстві, здобувачу належить провідна роль у постановці наукових задач, розробці методичних і алгоритмічних рішень, аналізі результатів та підготовці текстів публікацій. Співавтори брали участь у консультаціях, обговоренні проміжних результатів та редакційному опрацюванні матеріалів. Усі ключові ідеї, що становлять основу дисертаційної роботи та її наукову новизну, запропоновані та реалізовані здобувачем особисто.

Таким чином, особистий внесок здобувача полягає у формуванні наукової концепції дослідження, розробці методів і засобів забезпечення

переносимості програмних систем, створенні та експериментальній перевірці методу адаптивної контейнеризації з інтеграцією штучного інтелекту, а також у впровадженні отриманих результатів у наукову та навчальну діяльність.

Апробація результатів дисертації. Основні наукові положення, результати та висновки дисертаційної роботи пройшли широку апробацію у вигляді публікацій у наукових фахових виданнях, а також доповідей і обговорень на міжнародних та всеукраїнських науково-практичних конференціях, що підтверджує їх актуальність, наукову обґрунтованість і практичну значущість.

Результати досліджень, пов'язані з методами забезпечення переносимості програмного забезпечення, адаптивною контейнеризацією та застосуванням методів штучного інтелекту для оптимізації середовищ виконання програм, були представлені та обговорені на міжнародних науково-практичних конференціях. Зокрема, основні положення методу адаптивної контейнеризації з інтеграцією штучного інтелекту доповідались на XIX Міжнародній конференції з проблем використання інформаційних технологій в освіті, науці та промисловості (2024), а також на конференції "Problems of Using Information Technologies in Education, Science, and Industry" (2025), де отримали позитивні відгуки фахівців у галузі програмної інженерії та хмарних обчислень.

Наукові результати, що стосуються використання методів машинного навчання для автоматизованої адаптації та оптимізації програмного забезпечення у хмарних і периферійних середовищах, апробовані у фахових наукових журналах, зокрема у виданні Information Technology: Computer Science, Software Engineering and Cyber Security, а також у журналі Information Technology and Society, де опубліковано результати досліджень з оптимізації обчислювальних ресурсів та підвищення переносимості програмних систем. Опубліковані матеріали підтвердили ефективність

застосування алгоритмів машинного навчання для управління середовищами виконання програм у гетерогенних обчислювальних середовищах.

Результати досліджень, пов’язані з переносимістю вихідного коду C++ та автоматизованим виявленням платформозалежних компонентів, були апробовані у доповідях на міжнародних конференціях European Congress of Scientific Discovery (2025), Science and Education: Synergy of Innovation (2025), а також у журналі Herald of Khmelnytskyi National University. Technical Sciences. У зазначених публікаціях розглянуто проблеми переносимості програм між різними обчислювальними платформами, запропоновано підходи до автоматизованого аналізу та оцінювання платформозалежного коду, а також наведено результати експериментальних досліджень.

Окремі результати дисертаційної роботи, що стосуються метрик оцінювання переносимості програмного забезпечення та перенесення програм між гетерогенними обчислювальними архітектурами (CPU, GPU, TPU, FPGA), були представлені на міжнародних наукових форумах Science and Technology: Challenges, Prospects and Innovations (2025) та опубліковані у фахових наукових виданнях. Обговорення цих результатів сприяло уточненню критеріїв оцінювання переносимості та формуванню практичних рекомендацій щодо вибору методів перенесення програм у високопродуктивних і спеціалізованих обчислювальних середовищах.

Результати, пов’язані з автоматизацією супроводу програмного забезпечення, зокрема з використанням самовідновлюваних тестових наборів і автоматизованих методів підтримки нестабільних тестів, апробовані на XIII Міжнародній науково-практичній конференції “Information Control Systems and Technologies” (2025). Отримані результати підтвердили доцільність інтеграції автоматизованих механізмів тестування у процеси забезпечення переносимості програмних систем.

Наукові положення дисертаційної роботи також доповідались і обговорювались на наукових семінарах кафедр Національного технічного

університету «Дніпровська політехніка», де отримали схвальні відгуки та рекомендації щодо подальшого розвитку досліджень у напрямку адаптивних методів забезпечення переносимості програмного забезпечення.

Таким чином, результати дисертаційної роботи пройшли всебічну апробацію у науковому середовищі, підтверджені публікаціями у фахових наукових виданнях та матеріалах міжнародних і всеукраїнських конференцій, що свідчить про їх наукову цінність, актуальність і відповідність сучасному рівню розвитку комп'ютерних наук і програмної інженерії.

Публікації. Основні результати дисертаційної роботи опубліковано у 17 наукових працях. 6 статей опубліковано у наукових виданнях, включених до переліку фахових видань України (всі індексуються у НМБД Index Copernicus), 1 стаття опублікована у науковому виданні, що індексуються у Web of Science, 11 наукових праць опубліковано у збірниках наукових праць та матеріалах міжнародних конференцій.

Структура і обсяг роботи. Дисертаційна робота складається зі вступу, чотирьох розділів, висновків, списку використаних джерел та додатків. Загальний обсяг дисертації становить 215 сторінок, включає 17 рисунків, 9 таблиць, 115 літературних джерел, 10 додатків.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ ПЕРЕНОСИМОСТІ ПРОГРАМ ТА ПРОГРАМНИХ СИСТЕМ

1.1 Теоретичні основи переносимості програм

Теоретичні основи переносимості програм та програмних систем пов'язані з дотриманням певних стандартів та правил розробки програм, які дозволяють забезпечувати їх переносимість на різні платформи. [106;68]

Одним з найважливіших стандартів, який забезпечує переносимість програм, є стандарт ANSI C (American National Standards Institute C). Він встановлює правила розробки програм на мові програмування C, які забезпечують переносимість програм на різні платформи. Це досягається за допомогою визначення стандартів для деяких функцій та констант, які можуть бути реалізовані по-різному на різних платформах. [55]

Окрім стандарту ANSI C, існують інші стандарти, які допомагають забезпечити переносимість програм. Наприклад, стандарт POSIX (Portable Operating System Interface) встановлює стандарти для роботи з операційною системою, що дозволяє створювати переносимі програми для різних операційних систем. [105]

Іншим важливим аспектом переносимості програм є використання платформонезалежних мов програмування, таких як Java та Python. Ці мови програмування мають вбудовану віртуальну машину, що дозволяє запускати програми на будь-якій платформі, на якій є відповідний інтерпретатор.

Ще одним важливим аспектом переносимості програм є використання бібліотек та фреймворків, які забезпечують переносимість коду між різними платформами. Бібліотеки зазвичай містять готовий код, що використовується в програмі, а фреймворки забезпечують готові інфраструктурні рішення для розробки програм.

Ще одним підходом до забезпечення переносимості програм є використання віртуалізації. Віртуалізація дозволяє запускати програми на

віртуальних машинах, що дозволяє їх запуск на різних платформах, незалежно від операційної системи та апаратного забезпечення. [69]

Також важливою складовою переносимості програм є тестування програм на різних платформах. Тестування дозволяє виявити можливі проблеми з переносимістю програм та виправити їх перед релізом програми. [92]

Стандарти є одним з найважливіших аспектів переносимості програм, оскільки вони визначають спільний інтерфейс між програмним кодом та операційною системою або іншою платформою. Наприклад, стандарт POSIX (Portable Operating System Interface) визначає спільний інтерфейс між програмами та операційними системами класу Unix, що дозволяє розробляти переносимі програми для цих систем. [37]

Платформонезалежні мови програмування є ще одним важливим аспектом переносимості програм. Ці мови програмування, такі як Java, Python та C#, працюють на різних платформах завдяки використанню віртуальних машин та інших механізмів переносимості. [53]

Бібліотеки та фреймворки, що забезпечують переносимість коду, також мають важливе значення. Бібліотеки можуть містити готовий код, який може бути використаний у програмі, незалежно від платформи, на якій вона працює. Фреймворки забезпечують готові інфраструктурні рішення для розробки програм, що дозволяє програмістам сконцентруватись на логіці програми, а не на деталях реалізації.

Віртуалізація є ще одним підходом до забезпечення переносимості програм. Віртуальна машина дозволяє запускати програми на віртуальних платформах, що дозволяє їх запуск на різних платформах, незалежно від операційної системи та апаратного забезпечення.

Тестування програм на різних платформах також є важливим аспектом переносимості програм. Тестування дозволяє виявляти можливі проблеми з переносимістю програм та виправляти їх перед релізом програми. [51]

1.2 Критерії за якими можна оцінювати переносимість програм

Існує кілька критеріїв, за якими можна оцінювати переносимість програм:

1. Рівень абстракції: чим вищий рівень абстракції програмного інтерфейсу, тим більша ймовірність переносимості програми на різні платформи.
2. Залежність від апаратного забезпечення: чим менше програма залежить від конкретного апаратного забезпечення, тим більша ймовірність успішного перенесення на іншу платформу.
3. Використання стандартів: використання стандартів у програмі дозволяє підвищити рівень переносимості, оскільки стандарти є універсальними для різних платформ. [71]
4. Підтримка мов програмування: чим більша кількість мов програмування підтримується на платформі, тим більша ймовірність успішного перенесення програми на цю платформу.
5. Операційна система: підтримка однієї операційної системи може бути обмеженою в порівнянні з підтримкою кількох операційних систем.
6. Використання зовнішніх бібліотек: використання зовнішніх бібліотек може зробити програму менш переносимою, оскільки ці бібліотеки можуть бути залежні від конкретної платформи.
7. Використання засобів оптимізації: використання засобів оптимізації, таких як оптимізатори компілятора, може зробити програму менш переносимою, оскільки ці засоби можуть бути специфічними для певної платформи. [66]

Крім того, для оцінювання переносимості програм можна використовувати такі критерії:

1. Рівень переносимості: від мінімального до максимального. Мінімальний рівень переносимості означає, що програму можна перенести лише на обмежену кількість платформ. Максимальний

рівень переносимості передбачає можливість перенесення програми на будь-яку обчислювальну платформу без внесення змін в її код.

2. Легкість портабельності: від легкої до складної. Легко портабельні програми можуть бути перенесені з мінімальними зусиллями. Складно портабельні програми вимагають значної кількості зусиль для перенесення на іншу платформу.
3. Мінімальна кількість залежностей від платформи: від мінімальної до максимальної. Мінімальна кількість залежностей від платформи означає, що програма не має залежностей від конкретної платформи та може бути легко перенесена на інші платформи.
4. Наявність стандартів: наявність відповідних стандартів, які визначають певні правила та обмеження для розробки переносимих програм.
5. Можливість тестування: наявність засобів для тестування програми на різних платформах.

Оцінювання переносимості програм за цими критеріями допомагає розробникам програм оцінити ступінь їхньої переносимості та виявити можливі проблеми, пов'язані з перенесенням на різні платформи. [23]

1.3 Рівні абстракції відносно переносимості програм

Існують три рівні абстракції відносно переносимості програм:

1. Апаратний рівень (Hardware Level): на цьому рівні вирішуються питання, пов'язані з архітектурою процесора, обсягом оперативної та постійної пам'яті, розміром адресного простору, доступними розширеннями та іншими апаратними параметрами. [24] Завдання програмістів на цьому рівні полягає у розробці програм, які працюють безпосередньо з апаратними ресурсами, і забезпеченні можливості роботи цих програм на різних апаратних платформах.
2. Операційний рівень (Operating System Level): цей рівень передбачає взаємодію програм з операційною системою. [3] Операційна система

забезпечує абстракцію апаратного рівня, що дозволяє програмам працювати з різними пристроями в однаковий спосіб. Програми можуть використовувати функції операційної системи, такі як робота з файловою системою, мережевими з'єднаннями, розподіл ресурсів і так далі. Завдання програмістів на цьому рівні полягає у використанні стандартних інтерфейсів операційної системи, що дозволяє забезпечити переносимість програм між різними операційними системами.

3. Високорівневий рівень (High-Level Abstraction Level): на цьому рівні вирішуються питання, пов'язані з мовами програмування, бібліотеками та фреймворками. [41] Програмісти на цьому рівні працюють з високорівневими інструментами, які дозволяють забезпечити абстракцію операційної системи і апаратного рівня. Завдання програмістів на цьому рівні полягає у використанні стандартних механізмів.

Рівні абстракції є важливим поняттям при розробці переносимих програм і визначають рівень деталей, на яких розробник програми може зосередитися при розробці програмного забезпечення. Загалом існують чотири рівні абстракції від найнижчого до найвищого:

1. Архітектурно-залежний рівень - це найнижчий рівень абстракції, де програма написана для конкретної архітектури процесора та операційної системи і не може працювати на інших платформах без переробки коду.
2. Мовний рівень - це рівень, на якому програма написана мовою програмування, що є залежною від архітектури процесора. Тобто, програма може працювати на різних операційних системах, але не може працювати на інших процесорах.
3. Переносимий рівень - це рівень, на якому програма написана з використанням мови програмування, що дозволяє переносити

програмне забезпечення з однієї операційної системи на іншу, але все ще залежить від операційної системи.

4. Абстрактний рівень - це найвищий рівень абстракції, на якому програма розроблена на такому рівні абстракції, що дозволяє їй працювати на будь-якій операційній системі та процесорі.

Отже, для забезпечення переносимості програм потрібно звертати увагу на рівні абстракції та використовувати методи та засоби, що дозволяють зменшити залежність від конкретних архітектур, операційних систем та процесорів. [65]

1.4 Методи забезпечення переносимості програм

Існує кілька методів, які використовуються для забезпечення переносимості програм та програмних систем між різними платформами. Розглянемо кожен з цих методів детальніше:

1. Використання стандартизованих мов програмування. Цей метод передбачає використання мов програмування, які є стандартизованими, тобто мають однакову синтаксис та семантику на різних платформах. Найбільш поширеними стандартизованими мовами програмування є C, C++ та Java. Використання стандартизованих мов програмування дозволяє програмістам писати код, який працюватиме на будь-якій платформі, що підтримує відповідну мову програмування.
2. Використання стандартизованих API. API (Application Programming Interface) - це набір інструментів та інструкцій, які дозволяють програмістам створювати програми та взаємодіяти з іншими програмними продуктами. Використання стандартизованих API дозволяє розробникам створювати програми, які працюватимуть на будь-якій платформі, яка підтримує відповідний API. Найбільш поширеними стандартизованими API є POSIX (Portable Operating

System Interface for Unix) для операційних систем Unix та Win32 API для операційної системи Windows. [98]

3. Використання віртуальних машин. Віртуальна машина - це програмне забезпечення, яке імітує апаратну платформу та дозволяє запускати програми, написані для інших платформ, на даній платформі. [89] Цей метод полягає в тому, що програма виконується в окремій віртуальній машині, яка може бути емульована на різних платформах. [15] Для цього потрібно мати програмне забезпечення, що забезпечує віртуалізацію (наприклад, VirtualBox, VMware). За допомогою цього методу, програми можуть бути запущені на будь-якій платформі, яка підтримує віртуалізацію. Однак, використання віртуалізації може знизити продуктивність програми через додаткові шари абстракції, які необхідні для роботи віртуальної машини. [96]
4. Використання переносимих середовищ виконання (англ. Portable Execution Environment, PEE). Ці середовища виконання забезпечують ізольоване виконання програми від операційної системи та апаратної платформи. Вони включають в себе віртуальну машину, що імітує операційну систему та апаратну платформу, а також компілятор, інтерпретатор та бібліотеки для побудови програм. [33]
   - Одним із переносимих середовищ виконання є Java Virtual Machine (JVM), яке дозволяє виконувати Java-програми на будь-якій платформі, на якій встановлено JVM. Це досягається за допомогою компіляції Java-коду в байт-код, який потім виконується в JVM. [16]
   - Іншим прикладом переносимого середовища виконання є .NET Framework, що дозволяє виконувати програми на мовах C#, VB.NET та інших на будь-якій платформі, на якій встановлено .NET Framework. .NET Framework має свою віртуальну машину – Common Language Runtime (CLR), яка відповідає за виконання програм.

5. Контейнеризація є одним з найбільш ефективних методів забезпечення переносимості програм. [19] Вона полягає у запуску програм та їх залежностей від інших програм в окремому контейнері, що ізолює їх від решти системи. Кожен контейнер містить усі необхідні бібліотеки та ресурси, що дозволяє програмам запускатися незалежно від системних налаштувань, що можуть відрізнятися на різних платформах. Контейнеризація дозволяє розробникам створювати портативні програми, які можна легко розгортати на різних платформах без необхідності встановлювати всі залежності окремо. [22] Окрім того, контейнеризація дозволяє забезпечувати консистентність середовища, у якому працює програма, та уникнути конфліктів залежностей між різними програмами. Найпопулярніші інструменти контейнеризації включають Docker, Kubernetes, LXC, OpenVZ та інші. [49] Вони надають розробникам можливість легко створювати, розгортати та управляти контейнерами на різних платформах. [35] Крім того, вони дозволяють автоматизувати розгортання програм та їх масштабування в залежності від потреб користувачів.

1.5 Порівняльна характеристика стандартів, які забезпечують переносимість програм

Стандарти є важливим аспектом переносимості програм, оскільки вони визначають спільний інтерфейс між програмним кодом та операційною системою або іншою платформою. Ось порівняльна характеристика декількох стандартів, які забезпечують переносимість програм:

1. POSIX (Portable Operating System Interface) - цей стандарт був розроблений для забезпечення переносимості програм на операційних системах класу Unix. Він визначає спільний інтерфейс для взаємодії програм з операційною системою, включаючи системні виклики,

бібліотеки та інші компоненти. POSIX є одним з найбільш використовуваних стандартів для переносимості програм на операційних системах Unix.

2. C99 - це стандарт мови програмування C, який був прийнятий в 1999 році. Цей стандарт визначає мовні конструкції та функції, які можуть бути використані в програмах. Цей стандарт має багато вдосконалень порівняно зі старішим стандартом C89, і дозволяє написання більш переносимого коду.
3. Java - ця мова програмування використовує віртуальну машину для запуску програм. Це дозволяє програмам працювати на різних платформах, які підтримують віртуальну машину Java. Java має вбудовану бібліотеку, яка забезпечує спільний інтерфейс для взаємодії програм з операційною системою та іншими компонентами.
4. .NET Framework - цей фреймворк використовує віртуальну машину .NET для запуску програм. Це дозволяє програмам працювати на різних платформах, які підтримують віртуальну машину .NET. .NET має вбудовану бібліотеку, яка забезпечує спільний інтерфейс для взаємодії програм з операційною системою та іншими компонентами, що дозволяє написання більш переносимого коду.
5. HTML, CSS та JavaScript - ці технології використовуються для створення веб-сторінок та веб-додатків. Ці технології забезпечують переносимість програм завдяки тому, що вони працюють у веб-браузері, який доступний на багатьох платформах. Більшість сучасних веб-браузерів підтримують стандарти HTML, CSS та JavaScript, що дозволяє написання веб-додатків, які працюють на багатьох платформах.
6. OpenGL - цей стандарт визначає спільний інтерфейс для програмування 3D-графіки. OpenGL забезпечує переносимість програм, які використовують 3D-графіку, завдяки тому, що він може працювати на різних платформах.

Таблиця 1.1. Порівняльна характеристика стандартів переносимості програмного забезпечення.

Стандарт / платформа	**Рівень переносимості**	**Рівень абстракції**	**Переваги**	**Обмеження**
POSIX (IEEE 1003)	Високий у межах Unix-подібних систем	Операційна система	Уніфікований API для системних викликів, стабільність, широка підтримка	Обмежена сумісність з Windows, потреба в адаптаційних шарах
ANSI C / ISO C (C99, C11)	Високий на рівні вихідного коду	Мова програмування	Висока продуктивність, підтримка різними компіляторами	Відсутність стандартизації системних викликів
ISO C++ (C++11–C++20)	Високий, залежить від реалізації	Мова програмування	Сучасні абстракції, STL, висока продуктивність	Складність стандарту, різна підтримка компіляторами
Java SE / JVM	Дуже високий	Середовище виконання	Write once, run anywhere, керування пам’яттю	Накладні витрати JVM, обмежений доступ до low-level API
.NET / .NET Standard	Високий	Середовище виконання	Єдина кодова база, інтеграція з CI/CD	Залежність від runtime, платформа-залежні API

HTML, CSS, JavaScript (W3C, ECMAScript)	Максимальний	Браузер	Повна крос-платформність, мінімальні вимоги	Обмежений доступ до апаратних ресурсів
WebAssembly (Wasm)	Дуже високий	Бінарне середовище	Близька до нативної продуктивність, мовна	Обмежена взаємодія з ОС

Наведена таблиця демонструє, що кожен стандарт переносимості орієнтований на певний рівень абстракції та клас програмних систем. Низькорівневі стандарти забезпечують високу продуктивність, але потребують додаткових механізмів адаптації, тоді як керовані середовища виконання та веб-стандарти забезпечують максимальну переносимість за рахунок додаткових шарів абстракції. Це підтверджує доцільність поєднання стандартів переносимості з іншими методами, такими як контейнеризація та автоматизоване управління середовищами виконання.

Таблиця 1.2. Порівняння стандартів переносимості та методів забезпечення переносимості програмного забезпечення.

Стандарт / метод	**Віртуалізація (virtualization)**	**Контейнеризація (containerization)**	**Емуляція (emulation)**
POSIX	Використовується у гостьових ОС Unix-подібних систем	Безпосередньо підтримується у Linux containers	Обмежено, через емуляцію ОС
ANSI C / ISO C	Повна підтримка у VM	Повна підтримка у контейнерах	Підтримується через ISA емуляцію

ISO C++	Повна підтримка у VM	Повна підтримка у контейнерах	Підтримується з накладними витратами
Java / JVM	Виконується у VM поверх VM	Часто використовується у контейнерах	Можлива через JVM поверх емулятора
.NET / .NET Runtime	Підтримується у VM	Активно використовується у контейнерах	Обмежене застосування
Web (HTML, JS)	Працює у VM через браузер	Широко використовується у контейнерах	Не є типовим сценарієм
WebAssembly	Підтримується у VM та browser VM	Активно використовується у контейнерах	Можлива емуляція runtime

Наведена таблиця демонструє, що стандарти переносимості не є самодостатніми механізмами забезпечення крос-платформної сумісності, а ефективно реалізуються у поєднанні з конкретними технологічними методами. [101] Віртуалізація забезпечує високий рівень ізоляції та сумісності стандартів, проте супроводжується значними накладними витратами. Емуляція надає максимальну архітектурну незалежність, але обмежена продуктивністю. Контейнеризація, у свою чергу, забезпечує оптимальний баланс між переносимістю, продуктивністю та масштабованістю, що робить її найбільш придатною для сучасних багатоплатформних і хмарних систем.

Таким чином, результати порівняльного аналізу підтверджують доцільність використання контейнеризації як базового методу реалізації

стандартів переносимості у поєднанні з автоматизованими та адаптивними механізмами управління середовищами виконання. [104]

1.6 Порівняльна характеристика платформ, на які здійснюється переносимість програм

Зважаючи на те, що переносимість програм зазвичай забезпечується на рівні вищого програмного середовища (high-level programming environment), найбільш розповсюджені платформи, на які здійснюється переносимість програм, є такими:

1. Windows: операційна система, створена корпорацією Microsoft. Windows підтримує багато мов програмування, таких як C++, C#, Python, Java, Visual Basic та інші. [54] Також Windows надає можливість використовувати різні середовища розробки, такі як Visual Studio, Eclipse, NetBeans тощо. Windows є однією з найбільш поширених платформ для розробки програмного забезпечення. [73]
2. Linux: операційна система з відкритим кодом, яка забезпечує високу ступінь переносимості програм. [61] Linux надає доступ до різних компіляторів та середовищ розробки, таких як GCC, Clang, Eclipse, NetBeans тощо. Крім того, Linux підтримує багато мов програмування, включаючи C, C++, Python, Java та інші. [59]
3. MacOS: операційна система, створена компанією Apple, яка використовує архітектуру x86. macOS підтримує багато мов програмування, включаючи C++, Objective-C, Swift, Python, Java тощо. Для розробки програм на macOS можна використовувати різні інтегровані середовища розробки, такі як Xcode, Visual Studio Code, Atom тощо. [103]
4. Android: операційна система для мобільних пристроїв, створена компанією Google. Android підтримує різні мови програмування, включаючи Java, Kotlin, C++, Python тощо. Для розробки програм на

Android можна використовувати середовища розробки, такі як Android Studio, Eclipse тощо. [8]

5. iOS: операційна система для мобільних пристроїв, створена компанією Apple. iOS підтримує мови програмування, такі як Objective-C та Swift.
6. Web: Web-платформа дозволяє розробляти програмне забезпечення, яке працює в онлайн-режимі та забезпечує зручний доступ до даних та функцій за допомогою інтернету. [110] Ця платформа включає в себе мови програмування, такі як HTML, CSS, JavaScript, які використовуються для створення веб-сторінок та веб-додатків. [109] Web-платформа має багато переваг, зокрема високу доступність, гнучкість та широкі можливості для співпраці та обміну даними між користувачами. [46]
7. Mobile: Mobile-платформа включає в себе операційні системи, які використовуються на мобільних пристроях, таких як смартфони та планшети. До найпопулярніших операційних систем для мобільних пристроїв належать Android та iOS. Ця платформа має особливості, пов'язані з невеликим розміром екрану та обмеженою пам'яттю, що потребує оптимізації програмного забезпечення для мобільних платформ. [114]
8. Cloud: Cloud-платформа забезпечує можливість віддаленого доступу до програмного забезпечення та його ресурсів через мережу Інтернет. [63;81] Вона забезпечує збереження даних та дозволяє користувачам мати доступ до програмного забезпечення з будь-якого місця та пристрою. Ця платформа може забезпечити гнучкість та економію витрат, оскільки не потрібно мати великий обсяг обладнання для забезпечення функціонування програм. [13;28]
9. IoT: IoT-платформа забезпечує підтримку підключення різноманітних пристроїв до мережі Інтернет, передачу даних, обробку даних, аналіз даних та забезпечення безпеки даних. [75] Вона також забезпечує інфраструктуру для зберігання даних, їх обробки та забезпечення доступу

до них з боку користувачів, а також дозволяє здійснювати управління підключеними пристроями та взаємодію з ними.

Таблиця 1.3. Порівняльна характеристика обчислювальних платформ за архітектурними та API-аспектами переносимості.

Платформа	**Архітектура**	**Основні API та середовища виконання**	**Рівень переносимості**
Windows	x86, x86-64, ARM64	Win32 API, .NET Runtime, UWP	Середній
Linux	x86, x86-64, ARM, RISC-V	POSIX, glibc, system calls	Високий
macOS	x86-64, ARM64	POSIX, Cocoa, SwiftUI	Середній
Android	ARM, ARM64, x86	Android SDK, ART, NDK	Обмежений
iOS	ARM64	UIKit, SwiftUI, iOS SDK	Низький
Cloud platforms (AWS, Azure, GCP)	Абстрагована	VM, containers, managed services	Високий
Embedded / IoT	ARM, RISC-V	Bare-metal, RTOS, POSIX-підмножини	Низький

Таблиця 1.4. Порівняльна характеристика обчислювальних платформ за інструментальними можливостями та обмеженнями переносимості.

Платформа	**Типові інструменти**	**Основні обмеження**

Windows	MSVC, .NET, CMake, Docker Desktop	Платформозалежні API, обмежена POSIX-сумісність
Linux	GCC, Clang, Docker, Kubernetes	Різноманіття дистрибутивів, залежність від бібліотек
macOS	Xcode, Clang, Homebrew, Docker Desktop	Закрита екосистема, code signing, SIP
Android	Android Studio, Gradle, NDK	Фрагментація пристроїв, sandboxing
iOS	Xcode, Swift, Objective-C	Закрита платформа, жорсткі політики безпеки
Cloud platforms (AWS, Azure, GCP)	Kubernetes, Terraform, CI/CD	Vendor lock-in, різні API
Embedded / IoT	Cross-compilers, QEMU	Обмежені ресурси, відсутність стандартів

Наведені таблиці демонструють, що рівень переносимості програмного забезпечення істотно залежить від типу платформи, використовуваних API та моделей виконання. Платформи, орієнтовані на відкриті стандарти та контейнерні технології, зокрема Linux та хмарні середовища, забезпечують найвищий рівень переносимості. Натомість мобільні та вбудовані платформи характеризуються значними обмеженнями, що потребує застосування спеціалізованих фреймворків або архітектурних компромісів.

Таким чином, результати порівняльного аналізу підтверджують необхідність використання комплексних підходів до забезпечення переносимості програмних систем, які поєднують стандарти, автоматизовані інструменти, контейнеризацію та адаптивні механізми управління середовищами виконання, що створює передумови для розробки методу адаптивної контейнеризації. [115]

1.7 Підходи щодо перенесення програм на різні платформи

Існує кілька підходів до перенесення програм на різні платформи:

1. Написання коду для кожної платформи окремо: цей підхід полягає в тому, щоб написати код для кожної платформи окремо, використовуючи специфічні для неї інструменти розробки. Це може бути дуже часомістким і складним підходом, оскільки потрібно розробляти та підтримувати різні версії програми для кожної платформи.
2. Використання крос-платформеного програмування: цей підхід полягає в тому, щоб використовувати загальноприйняті мови програмування та бібліотеки для розробки програм, які можуть працювати на різних платформах. Наприклад, мова програмування Java є крос-платформеною, тому програми, написані на цій мові, можуть запускатися на будь-якій платформі, де є встановлене відповідне середовище виконання.
3. Використання віртуалізації: цей підхід полягає в тому, щоб використовувати віртуальні машини для запуску програм на різних платформах. За допомогою віртуальних машин можна виконувати програми, написані на різних мовах програмування, на будь-якій платформі, де є відповідний віртуальний розподільчий образ.
4. Використання платформи, що забезпечує автоматичну переносимість програм: цей підхід полягає в тому, щоб використовувати спеціальні платформи, що забезпечують автоматичний перенос програм на різні платформи. Такі платформи можуть використовувати різні технології, наприклад, трансляцію коду або використання віртуальних машин.
5. Використання набору стандартів: цей підхід полягає в тому, щоб використовувати набір стандартів та протоколів, що дозволяють програмам працювати на різних платформах. Наприклад, мережеві

протоколи, такі як HTTP або TCP/IP, дозволяють програмам зв'язуватися та обмінюватися даними між різними платформами.

6. Використання хмарних сервісів: цей підхід полягає в тому, щоб використовувати хмарні сервіси, що дозволяють запускати програми на серверах у хмарі. За допомогою хмарних сервісів можна запускати програми на різних платформах, не витрачаючи час на їх перенесення. [95]
7. Використання кросплатформеного фреймворку: цей підхід полягає в тому, щоб використовувати кросплатформені фреймворки, що дозволяють розробляти програми, які можуть працювати на різних платформах без необхідності переписування коду. [34] Такі фреймворки надають розробникам можливість створювати програми, що працюють на різних платформах, за допомогою спеціальних інструментів та бібліотек.
8. Використання нативних інструментів: цей підхід полягає в тому, щоб використовувати нативні інструменти та бібліотеки, що дозволяють розробляти програми для різних платформ без необхідності переписування коду. Цей підхід може бути складним, оскільки різні платформи мають різні набори інструментів та бібліотек, але він дозволяє створювати програми з максимальною продуктивністю та функціональністю для кожної платформи окремо.

Кожен з цих підходів має свої переваги та недоліки, і вибір підходу залежить від багатьох факторів, таких як розмір програми, використовувані технології, доступність необхідних ресурсів, бюджет тощо. В будь-якому випадку, перенесення програм на різні платформи може бути складним процесом, але правильний підхід може допомогти зекономити час та зусилля, необхідні для розробки та підтримки програм на різних платформах.

1.8 Порівняння крос-компіляції, віртуалізації та контейнеризації

Для систематизації методів забезпечення переносимості програмного забезпечення та визначення їх застосовності у різних сценаріях доцільно виконати порівняльний аналіз крос-компіляції, віртуалізації та контейнеризації. Кожен із зазначених методів вирішує задачу переносимості на різних рівнях абстракції та має власні переваги й обмеження, що визначають сферу його практичного використання. [36]

Таблиця 1.5 – Порівняльна характеристика методів забезпечення переносимості програмного забезпечення.

Характеристика	**Крос-компіляція**	**Віртуалізація**	**Контейнеризація**
Рівень абстракції	Компіляція та ISA	Апаратна платформа	Операційна система
Залежність від архітектури	Висока	Низька	Середня
Продуктивність	Близька до нативної	Нижча за нативну	Близька до нативної
Накладні витрати	Мінімальні	Високі	Низькі
Потреба у гостьовій ОС	Відсутня	Обов’язкова	Відсутня
Складність налаштування	Висока	Середня	Низька–середня
Тестування результатів	Потребує емуляції або HW	Просте	Просте

Типові інструменти	GCC, Clang, CMake, QEMU	KVM, VMware, Hyper-V	Docker, Kubernetes
Основні сценарії застосування	Embedded, mobile, IoT	Legacy systems, isolation	Cloud-native, microservices

Наведена таблиця демонструє, що крос-компіляція забезпечує найвищу продуктивність та мінімальні накладні витрати, проте потребує складної конфігурації toolchain і не вирішує проблеми платформозалежних API. [42] Віртуалізація забезпечує максимальну сумісність середовища виконання, але супроводжується значними накладними витратами та зниженням ефективності використання ресурсів. [38] Контейнеризація займає проміжне положення, поєднуючи близьку до нативної продуктивність із високим рівнем переносимості та простотою розгортання.

Таким чином, вибір методу забезпечення переносимості залежить від вимог до продуктивності, типу цільової платформи та сценарію використання програмного забезпечення. У сучасних багатоплатформних системах найбільш ефективним є поєднання зазначених підходів, зокрема використання крос-компіляції для створення бінарних артефактів та контейнеризації для уніфікації середовищ виконання, що створює основу для подальшого розвитку адаптивних методів забезпечення переносимості. [113]

1.9 Висновки до першого розділу

У першому розділі дисертаційної роботи здійснено комплексний аналіз теоретичних основ переносимості програм та програмних систем, методів та інструментів її забезпечення на різні обчислювальні платформи. Встановлено, що переносимість є багатовимірною характеристикою, яка не лише охоплює можливість виконання програм на різних платформах, але й

здатність зменшувати витрати на розробку, супровід і тестування в умовах зростаючої гетерогенності обчислювальних систем. Оцінювання переносимості потребує комплексного підходу з урахуванням рівня абстракції, залежності від апаратного забезпечення, використання стандартів і підтримки мов програмування. Підвищення рівня абстракції зазвичай спрощує процес перенесення, але може супроводжуватися додатковими накладними витратами або обмеженням доступу до апаратних ресурсів.

Аналіз підтверджує, що забезпечення переносимості лише за рахунок дотримання стандартів або використання крос-платформних фреймворків є недостатнім, оскільки платформозалежні API та відмінності у моделях керування ресурсами створюють додаткові обмеження у сучасних гетерогенних середовищах. Аналіз традиційних методів забезпечення переносимості показав, що віртуалізація та емуляція хоча й забезпечують високий рівень сумісності середовищ виконання, проте супроводжуються значними накладними витратами та обмеженнями щодо продуктивності й масштабованості. З іншого боку, крос-компіляція дозволяє досягти близької до нативної продуктивності, але не вирішує проблеми платформозалежних API та складності крос-платформного тестування.

Аналіз впливу різних платформ виявив, що відкриті системи з підтримкою стандартів POSIX та контейнерних технологій забезпечують вищий рівень переносимості, тоді як мобільні та спеціалізовані платформи характеризуються жорсткими обмеженнями та вимагають спеціалізованих інструментів. Крім того, зростання гетерогенності апаратних платформ і розвиток нових архітектур ускладнює перенесення традиційними методами та вимагає впровадження абстрактних інтерфейсів і універсальних середовищ виконання.

На сучасному етапі контейнеризація та оркестрація виступають базовими сучасними механізмами забезпечення переносимості у хмарних, мультихмарних та розподілених архітектурах. [26] Ці технології пропонують оптимальний баланс між переносимістю, продуктивністю та

масштабованістю, проте їхні механізми управління ресурсами залишаються переважно статичними. Різноманітні засоби автоматизації, такі як системи збірки, управління конфігурацією, інструменти міграції та інфраструктура як код, є необхідною умовою підтримки переносимості у великих проєктах і суттєво знижують трудомісткість процесів. [76] Водночас більшість із цих інструментів автоматизації ґрунтується на статичних правилах та конфігураціях і не здатна гнучко враховувати динамічні характеристики середовищ виконання.

Загалом, проблема переносимості програм та програмних систем є складною багаторівневою задачею, яка не може бути ефективно розв’язана за допомогою жодного окремого методу. Виявлені суттєві обмеження традиційних підходів і сучасних інструментів в умовах складних багатоплатформних і мультихмарних середовищ обґрунтовують необхідність розробки нових, адаптивних методів забезпечення переносимості. Такі підходи мають комплексно поєднувати автоматизацію, контейнеризацію та інтелектуальні механізми адаптації, щоб автоматично враховувати специфіку апаратних платформ та динамічні умови виконання у реальному часі.

РОЗДІЛ 2. МЕТОД АДАПТИВНОЇ КОНТЕЙНЕРИЗАЦІЇ З ІНТЕГРАЦІЄЮ ШТУЧНОГО ІНТЕЛЕКТУ

2.1 Вступ до методу

Аналіз сучасних підходів до забезпечення переносимості програмного забезпечення, виконаний у попередньому розділі, показав, що традиційні методи, зокрема використання стандартизованих мов програмування, віртуалізації, контейнеризації та автоматизованих інструментів розгортання, забезпечують лише часткове розв’язання задачі переносимості у гетерогенних обчислювальних середовищах. Основним обмеженням існуючих рішень є їх статичний характер, коли параметри середовища виконання визначаються заздалегідь і не адаптуються до змінних умов експлуатації програмних систем.

У сучасних обчислювальних середовищах, що характеризуються динамічним розподілом ресурсів, змінним навантаженням, різноманіттям апаратних архітектур і мультихмарними сценаріями розгортання, статичні конфігурації контейнерів та середовищ виконання часто призводять до неефективного використання ресурсів, зниження продуктивності або нестабільної роботи програм. [17] Це особливо критично для систем, що працюють у режимі реального часу, обробляють нерівномірні потоки даних або функціонують у середовищах з обмеженими обчислювальними ресурсами.

Контейнеризація, яка є базовою технологією сучасних cloud-native систем, забезпечує високий рівень переносимості середовищ виконання, проте у класичному вигляді не враховує індивідуальні характеристики цільової платформи, такі як тип процесорної архітектури, наявність апаратних прискорювачів, пропускна здатність підсистеми пам’яті або мережі. Відсутність механізмів автоматизованої адаптації параметрів

контейнерів до умов виконання обмежує ефективність контейнерних рішень у гетерогенних середовищах.

У цьому контексті перспективним напрямом є поєднання контейнеризації з методами штучного інтелекту та машинного навчання, що дозволяє реалізувати адаптивне управління середовищами виконання програм. Інтеграція інтелектуальних алгоритмів аналізу характеристик платформи та поведінки програм відкриває можливість динамічного налаштування параметрів контейнерів, прогнозування навантаження та адаптивної нормалізації використання обчислювальних ресурсів у реальному часі.

Розглянемо метод адаптивної контейнеризації з інтеграцією штучного інтелекту спрямований на усунення виявлених обмежень традиційних підходів. Метод базується на поєднанні контейнерних технологій, систем оркестрації та алгоритмів машинного навчання для автоматизованого прийняття рішень щодо конфігурації середовища виконання програм залежно від характеристик цільової платформи та поточного стану системи.

Основною ідеєю методу є перехід від статичних, заздалегідь визначених конфігурацій контейнерів до динамічних адаптивних конфігурацій, що формуються на основі аналізу метрик продуктивності, використання ресурсів та історичних даних про виконання програм. Такий підхід дозволяє підвищити рівень переносимості програмних систем, забезпечуючи їх коректну та стабільну роботу на різних обчислювальних платформах без необхідності ручного налаштування.

Метод орієнтований на використання у різних обчислювальних середовищах, що включають операційні системи, процесорні архітектури, хмарні та мультихмарні інфраструктури, а також середовища з обмеженими ресурсами. Його застосування дозволяє не лише уніфікувати середовище виконання, але й автоматично адаптувати його до конкретних умов експлуатації, що є ключовою вимогою для сучасних багатоплатформних програмних систем.

Будемо розглядати переносимість програм та програмних систем як здатність зберігати функціональну коректність та допустимі експлуатаційні характеристики при зміні обчислювальної платформи без модифікації програмного коду. Переносимість інтерпретується як динамічна, контекстно-залежна властивість керованого середовища виконання, що формується в результаті взаємодії програмної системи, обчислювальної платформи та механізмів управління ресурсами.

Формально контейнеризована програмна система розглядається як керований динамічний об’єкт.

Стан системи у момент часу t описується вектор-функцією:

$$x(t) = (m(t), c(t), P), \tag{2.1}$$

де:

$m(t) = \left(m_1(t), \ldots, m_k(t)\right)$ — вектор-функція експлуатаційних метрик програмної системи.

Дані збираються безпосередньо з механізмів операційної системи хоста cgroups (control groups), а також через середовище виконання контейнерів (container runtime). [84] У кластері Kubernetes ці показники експортуються компонентом kubelet.

Компонентами вектор-функції $m_i(t)$ $(i = 1,..,k)$ можуть бути:

- використання процесорних ресурсів (у системах Prometheus/Kubernetes цей показник вимірюється у секундах використання процесорного часу, які перераховуються у міліядра (millicores));
- використання пам’яттєвих ресурсів (гігабайти);
- показники I/O (кількість операцій читання/запису за секунду (IOPS));
- мережеві метрики (вимірюється в байтах за секунду. отримані/відправлені дані);

- затримки обробки (вимірюються як часові характеристики виконання програм, тобто у мілісекундах на один запит);
- пропускна здатність (вимірюється як інтенсивність потоку обробки у кількості запитів за секунду (Requests Per Second, RPS));
- і т.д.

$c(t) = \left(c_1(t), \ldots, c_v(t)\right)$ — вектор-функція параметрів контейнерного середовища виконання, що включає обмеження ресурсів, кількість реплік та політики розміщення.

Усі компоненти вектора-функції отримуються безпосередньо з системи оркестрації (control plane). Контролер робить запити до Kubernetes API Server і зчитує поточний стан конфігураційних ресурсів (зокрема, об'єктів Deployment та Pod).

Компонентами вектора-функції $c_i(t)$ $(i = 1,..,v)$ можуть бути:

- обмеження та запити ресурсів контейнерів (вимірюється у міліядрах (millicores));
- кількість реплік (вимірюється у одиницях. Абсолютне цілочисельне значення, що вказує на кількість паралельно запущених екземплярів додатку);
- політики розміщення (це не кількісні, а категоріальні значення. Перед тим як потрапити до складу вектора ці категоріальні дані перетворюються у числову форму через one-hot encoding);
- пріоритети виконання (цілочисельні значення (integer), де більше число означає вищий пріоритет для планувальника);
- і т.д.

$P = \left(p_1, \ldots, p_p\right)$ — вектор є профілем цільової обчислювальної платформи виконання, що відображає архітектурні та ресурсні характеристики середовища.

Компоненти вектору є статичними (platform-level metadata). Вони формуються шляхом програмних запитів до Kubernetes Node API (отримуючи об'єкти типу Node через CoreV1Api).

Компонентами вектора $p_i \; (i = 1,..,p)$ можуть бути:

- процесорна архітектура (x86_64, ARM64, GPU. Це не кількісні, а категоріальні значення. Перед тим як потрапити до складу вектора ці категоріальні дані перетворюються у числову форму через one-hot encoding);
- обсяг доступної оперативної пам’яті (вимірюється у гігабайтах);
- кількість ядер CPU (вимірюється в абсолютних цілих числах);
- додаткові атрибути середовища / вузла (це не кількісні, а категоріальні значення. Перед тим як потрапити до складу вектора ці категоріальні дані перетворюються у числову форму через one-hot encoding);
- і т.д.

Адаптація середовища виконання здійснюється дискретно з періодом керування Δ — період між послідовними ітераціями керуючого циклу.

Еволюція стану системи у межах однієї ітерації описується рівнянням переходу:

$$x(t+\Delta) = g\big(x(t), u(t), P, w(t)\big), \tag{2.2}$$

де

P — профіль цільової обчислювальної платформи виконання;

$x(t+\Delta)$ — стан програмної системи після застосування керуючого впливу Δ через один період керування;

$g()$ — вектор-функція переходу стану системи;

$u(t) = \big(u_1(t), \ldots, u_v(t)\big)$ — вектор-функція керуючого впливу у момент часу t, що відповідає зміні параметрів контейнерного середовища виконання.

Розмірність співпадає з розмірністю вектора-функції конфігурації $c(t)$, оскільки дія $u(t)$ полягає у зміні саме цих параметрів.

Ці значення не "зчитуються", а генеруються моделлю машинного навчання, як оптимальна дія. Фактичне застосування впливу виконується компонентом actuation layer, який формує декларативний API-запит (операції patch або update) до Kubernetes API Server для зміни параметрів об'єктів.

Компонентами вектор-функції $u_i(t)$ $(i=1,..,v)$ можуть бути:

- зміна обмежень ресурсів контейнерів (CPU limits, memory limits. міліядра (millicores) для CPU та гігабайти для пам'яті);
- зміна запитів ресурсів контейнерів (CPU requests, memory requests. Міліядра (millicores) для CPU та гігабайти для пам'яті);
- зміна кількості реплік додатку (горизонтальне масштабування /replica count. Абсолютні цілі числа (штуки));
- зміна пріоритетів виконання контейнерів (Цілочисельні значення, які використовуються оркестратором для ранжування важливості процесів при конкуренції за ресурси);
- зміна політик розміщення (правила affinity/tolerations для міграції чи перерозподілу. Це не кількісні, а категоріальні значення. Перед тим як потрапити до складу вектора-функції ці категоріальні дані перетворюються у числову форму через one-hot encoding);
- зміна перерозподілу навантажень (вагові коефіцієнти (цілі числа) розподілу трафіку або логічні правила маршрутизації мережевих запитів);
- і т.д.

$w(t)=\left(w_1(t),\ldots,w_s(t)\right)$ — вектор-функція зовнішніх збурень, зумовлених змінами навантаження, конкуренцією за ресурси та недетермінізмом розподіленого середовища. Він описує неконтрольовані фактори середовища, які впливають на систему. Хоча ці параметри є зовнішніми, їхні наслідки система «відчуває» і фіксує опосередковано через збір телеметрії системою Prometheus.

Компонентами вектор-функції $w_i(t)$ $(i=1,..,s)$ можуть бути:

- зміни характеристик навантаження (сплески запитів — burst workload, зміна інтенсивності. Вимірюється у кількості запитів за секунду (Requests Per Second));
- конкуренція за ресурси вузла (resource contention) з боку інших контейнерів (вимірюватися у відсотках використання загального процесора іншими (сторонніми) процесами на вузлі);
- недетермінізм розподіленого середовища (мережеві затримки, вплив фонових процесів оркестратора. Вимірюється у мілісекундах (ms) для затримок (latency) та тайм-аутів);
- і т.д.

Еталонне виконання або допустимий експлуатаційний інтервал визначає межі прийнятної поведінки програмної системи. Адаптація середовища виконання спрямована на підтримку програмної системи в межах цього інтервалу незалежно від характеристик цільової платформи.

Контейнеризація розглядається як інструмент абстракції та керованої компенсації відмінностей між платформами виконання, що дозволяє забезпечити переносимість програмних систем без необхідності модифікації програмного коду.

Еталонний профіль виконання m_{ref} визначається для програмної системи на базовій обчислювальній платформі або формується на основі усереднених історичних даних.

$m_{\mathrm{ref}}=\left(m_{ref,1},\ldots,m_{ref,k}\right)$ — вектор метрик, що описує допустиму поведінку програмної системи. Співпадає з розмірністю простору поточних експлуатаційних метрик $m(t)$.

Компонентами вектора m_{ref} є еталонні цільові значення $m_{ref,i}$ $(i=1,..,k)$ або межі допустимого експлуатаційного інтервалу $[L_i,U_i]$ для кожної контрольованої метрики:

- допустиме використання процесорних ресурсів (цільове значення споживання CPU. Вимірюється в міліядрах (millicores))
- допустиме використання пам'яттєвих ресурсів (еталонний обсяг необхідної RAM. Вимірюється в гігабайтах);
- еталонні показники I/O (бажана швидкість роботи з I/O. Вимірюється в кількості операцій за секунду (IOPS));
- еталонні мережеві метрики (бажана пропускна здатність мережевих інтерфейсів. Вимірюється в байтах/сек);
- допустимі затримки обробки запитів (максимально прийнятний час відповіді сервісу на запит користувача. Вимірюється в мілісекундах (ms));
- еталонна/допустима пропускна здатність системи (бажана кількість успішно оброблених транзакцій. Вимірюється в кількості запитів за секунду (RPS)).

Альтернативно, для кожної ключової метрики може бути заданий допустимий інтервал значень, що визначає межі прийнятної поведінки системи.

Відхилення поведінки програмної системи від еталонного виконання або допустимого інтервалу характеризується функцією $D(t)$, яка визначається як кількісна міра відстані між поточними значеннями $m(t)$ та еталонним профілем або відповідним експлуатаційним інтервалом. Функція $D(t)$ використовується як кількісний індикатор переносимості та виступає сигналом помилки у керуючому циклі адаптації.

Для забезпечення порівнюваності різнорідних експлуатаційних метрик формується нормалізована вектор-функція відхилень:

$$\delta(t) = \left(\delta_1(t), \ldots, \delta_k(t)\right), \tag{2.3}$$

де кожна компонента відповідає окремій метриці виконання. Якщо для i-ї метрики задано еталонне значення $m_{ref,i}$, тоді нормалізоване відхилення визначається як:

$$\delta_i(t) = \frac{m_i(t) - m_{\mathrm{ref},i}}{\max\left(\left|m_{\mathrm{ref},i}\right|, \varepsilon_0\right)}, \tag{2.4}$$

де ε_0 є малим додатним параметром, що запобігає діленню на нуль.

У випадку, коли для i-ї метрики задано допустимий експлуатаційний інтервал $[L_i, U_i]$, нормалізоване відхилення визначається як нуль у межах цього інтервалу та як нормована відстань до найближчої межі інтервалу при виході за його межі:

$$\delta_i(t) = \begin{cases} 0, & L_i \le m_i(t) \le U_i \\ \dfrac{L_i - m_i(t)}{\max\left(U_i - L_i,\, \varepsilon_0\right)}, & m_i(t) < L_i, \\ \dfrac{m_i(t) - U_i}{\max\left(U_i - L_i,\, \varepsilon_0\right)}, & m_i(t) > U_i. \end{cases}, \tag{2.5}$$

$$D(t) = \sum_{i=1}^{k} w_i \cdot \left|\delta_i(t)\right|, \tag{2.6}$$

Значення вагових коефіцієнтів w_i визначаються з урахуванням пріоритетів експлуатаційних характеристик програмної системи або вимог рівня сервісу, що дозволяє адаптувати цільову функцію переносимості до конкретного класу застосувань.

Використання зваженої міри відхилення (2.6) забезпечує робастність до поодиноких викидів у метриках та дозволяє інтерпретувати коефіцієнти w_i як

пріоритети експлуатаційних характеристик у задачі забезпечення переносимості.

Оптимальне керування поведінкою програмної системи для наближення до еталонного виконання полягає у мінімізації відхилення переносимості $D(t)$ у часі шляхом вибору відповідних керуючих впливів $u(t)$ з урахуванням профілю платформи P та обмежень системи оркестрації (така як Kubernetes, що реалізує управління життєвим циклом контейнерів у розподіленому середовищі та виступає виконавчим механізмом для застосування керуючих рішень). Переносимість вважається забезпеченою, якщо значення $D(t)$ не перевищує заданого порогового рівня або якщо всі експлуатаційні метрики перебувають у межах допустимого інтервалу протягом визначеного періоду часу. [50]

З формальної точки зору переносимість програмної системи вважається забезпеченою, якщо значення функції відхилення $D(t)$ не перевищує заданого порогового рівня ε протягом контрольного інтервалу часу:

$$[t_0, t_0 + T], \tag{2.7}$$

де T визначає тривалість спостереження.

Для програмних систем з динамічним або нерівномірним навантаженням допускається ослаблений критерій, за якого частка часу, протягом якого $D(t) > \varepsilon$, не перевищує заданого допустимого значення ρ.

Параметри ε, T та ρ визначаються з урахуванням вимог до експлуатаційних характеристик програмної системи та використовуються як формальні критерії оцінювання та порівняння ефективності різних стратегій адаптації середовища виконання у процесі експериментальної перевірки методу.

Оптимізація продуктивності та ефективності використання ресурсів розглядається як вторинна задача, яка допускається лише за умови збереження переносимості як первинної цільової властивості програмної системи.

2.2 Застосування машинного навчання у методі адаптивної контейнеризації

В основі методу лежить концепція адаптивної контейнеризації, у межах якої параметри середовища виконання формуються та коригуються автоматично на основі аналізу характеристик платформи та поведінки програмної системи. Контейнер розглядається як динамічне середовище, здатне змінювати свої параметри у процесі виконання.

Використовуються алгоритми машинного навчання:

- керована регресія (supervised regression);
- деревоподібні регресійні моделі (tree-based regression models); зокрема Random Forest та Gradient Boosting;
- лінійна регресія (linear regression);
- неглибокі нейронні мережі (shallow neural networks) для аналізу багатовимірних експлуатаційних даних. Вони обробляють вектор-функцію ознак (який поєднує поточні метрики, конфігурацію середовища та профіль платформи) для точного розрахунку відхилення, прогнозування навантаження та своєчасного прийняття керуючих рішень. Для обробки надзвичайно складних патернів навантаження може застосовуватися глибоке навчання, що дозволить виявляти ще більш приховані закономірності.

Машинне навчання оцінює не окремі ресурсні метрики (такі як завантаження CPU чи пам'яті), а комплексну міру відхилення переносимості

$D(t)$. У термінах машинного навчання ця задача розв'язується у межах парадигми керованого навчання із застосуванням складних нелінійних моделей, до яких належать ансамблі дерев рішень та неглибокі нейронні мережі. Цільовою змінною для навчання моделі виступає кількісний індикатор відхилення поведінки системи від еталонної.

Вхідними даними є вектор-функція ознак $\varphi(t)=\left(\varphi_1(t),\dots,\varphi_d(t)\right)$. Алгоритми працюють з вектор-функцією ознак $\varphi(t)$, а не з сирими даними. Компонентами вектор-функції ознак $\varphi_i(t)$ $(i=1,..,d)$ можуть бути:

- нормалізовані метрики виконання (навантаження, затримки. Оскільки це дані, підготовлені для математичних алгоритмів (особливо нейронних мереж), вони проходять етап масштабування. Тому їхні одиниці вимірювання стають нормалізованими числовими значеннями (наприклад, приведеними до діапазону [0;1]);

- параметри поточної контейнерної конфігурації (ліміти ресурсів, репліки. Міліядра для CPU, гігабайти для пам'яті, штуки для реплік);

- числові характеристики профілю платформи (процесорна архітектура, доступні ресурси. Для кількісних характеристик (наприклад, обсяг доступних ресурсів) — числові значення (міліядра для CPU, гігабайти для пам'яті). Для категоріальних характеристик (наприклад, процесорна архітектура "x86_64" чи "ARM64") використовуються метод кодування one-hot encoding).

Інтелектуальний модуль машинного навчання забезпечує проактивний характер адаптації, виконуючи дві взаємопов'язані функції:

- оцінювання поточного стану: розрахунок поточного значення відхилення переносимості на основі вектора-функції ознак $\varphi(t)$;
- прогнозування наслідків/what-if analysis: симуляція застосування множини можливих керуючих впливів (candidate actions) та прогнозування очікуваного відхилення для кожного з них.

Як базове алгоритмічне рішення використовуються деревоподібні регресійні моделі (tree-based regression models). Цей вибір обґрунтований тим, що такі моделі:

- добре працюють із неоднорідними ознаками (метрики + конфігурація + профіль вузла);
- стійкі до шуму;
- забезпечують швидкий інференс (необхідно для near real-time керування);
- дозволяють інтерпретувати внесок окремих ознак у рішення, тобто модуль не є "чорною скринькою".

Моделі навчаються офлайн на історичних даних (knowledge base), а в процесі експлуатації навчені моделі застосовуються для швидкого прийняття рішень (online decision making). Таким чином досягається розділення навчання та інференсу.

Машинне навчання виконує функцію центрального аналітичного механізму, який забезпечує перехід від статичного або евристичного керування контейнерним середовищем до адаптивного прийняття рішень на основі даних. Штучний інтелект діє як автономний керуючий агент, що здатний предиктивно реагувати на будь-які зміни апаратного середовища. У межах уже введеної формалізації стан програмної системи задається вектор-функцією $x(t)$ відповідно до формули (2.1), а його зміна в часі описується рівнянням переходу (2.2). Це означає, що машинне навчання в методі працює не ізольовано, а вбудовується у вже визначену динамічну модель керованого середовища виконання, де метрики виконання, параметри контейнерного середовища та профіль цільової платформи розглядаються як взаємопов’язані складові одного процесу адаптації.

Ключовою особливістю даного підходу є те, що об’єктом оцінювання для моделей машинного навчання виступає не окрема локальна метрика, наприклад завантаження CPU чи споживання пам’яті, а агрегована міра

відхилення переносимості $D(t)$, уже формалізована у формулах (2.3) – (2.6). Таким чином, машинне навчання тут орієнтоване на виявлення та компенсацію платформно-індукованих відмінностей у поведінці програмної системи, а не на пряме покращення однієї вибраної експлуатаційної характеристики.

У логіці методу інтелектуальний модуль машинного навчання виконує дві взаємопов’язані функції. Перша з них полягає в оцінюванні поточного стану: модель аналізує поточний стан системи та формує оцінку поточного відхилення переносимості. Друга функція є прогностичною: модель розраховує, яким буде значення відхилення переносимості після застосування певного кандидатного керуючого впливу. У результаті машинне навчання в межах методу працює одночасно як механізм інтерпретації поточного стану системи і як механізм прогнозування наслідків майбутніх дій. Саме ця подвійна роль і забезпечує проактивний характер адаптації.

З алгоритмічної точки зору це означає, що інтелектуальний модуль не обмежується відповіддю на питання про те, чи є поточний стан проблемним. Він також дозволяє відповісти на питання, яка саме зміна конфігурації контейнерного середовища з найбільшою ймовірністю зменшить відхилення переносимості на наступній ітерації керуючого циклу. Власне тому машинне навчання включене безпосередньо в механізм вибору керуючого впливу: спочатку формується множина допустимих дій, потім для кожної з них оцінюється очікуваний ефект, після чого обирається дія, яка мінімізує прогнозоване відхилення переносимості та не погіршує стан системи.

Поглиблений зміст машинного навчання полягає також у тому, що воно спирається на спеціально сформоване числове представлення стану системи. До нього входять нормалізовані метрики виконання, параметри поточної контейнерної конфігурації та числові характеристики профілю платформи виконання, зокрема відомості про архітектуру вузла, доступні

ресурси та поточний рівень вузлового навантаження. Отже, машинне навчання працює не з сирими даними моніторингу, а з уже підготовленим, структурованим описом системи, який поєднує поведінку програми та умови її виконання.

Побудова ознак здійснюється як окремий етап feature engineering у складі state construction layer та machine learning inference layer. З одного боку, з Prometheus і Kubernetes Metrics API надходять container-level, node-level та application-level metrics. З іншого боку, через Kubernetes Node API формується профіль платформи P, який включає процесорну архітектуру, доступні ресурси та інші platform-level metadata. Після цього всі компоненти об'єднуються в єдиний ознаковий простір, придатний для інференсу моделі. Така побудова є методологічно важливою, оскільки дозволяє моделі виявляти не просто залежності між навантаженням і споживанням ресурсів, а саме залежності між платформою, конфігурацією середовища та зміною індикатора переносимості.

Модель навчається на історичних даних, де кожному вектору-функції ознак відповідає значення відхилення переносимості, обчислене за вже введеною формалізацією. Таким чином, функція $D(t)$, задана у формулах (2.3)–(2.6), виступає не лише індикатором ефективності методу, а й цільовою змінною для навчання моделі. Це дозволяє уникнути розриву між теоретичною частиною та практичною реалізацією, оскільки одна й та сама формалізація використовується і в постановці задачі, і в її програмному втіленні.

В архітектурі прототипу передбачено розділення етапів training та inference. Це означає, що машинне навчання реалізується як конкретний обчислювальний модуль із чітко визначеним життєвим циклом моделей. Навчання виконується на історичних даних, накопичених у knowledge base, а в процесі експлуатації використовуються попередньо навчені моделі, які працюють у режимі online decision making. Така організація є особливо

важливою для контейнерних і оркестрованих середовищ, де затримка прийняття рішення має бути обмеженою, а тому складні або надто ресурсоємні моделі не завжди є практично доцільними.

У програмному прототипі як базове рішення використовуються деревоподібні регресійні моделі tree-based regression model, хоча загальна архітектура допускає також linear regression, tree-based models і shallow neural networks. Цей вибір обґрунтований тим, що деревоподібні моделі добре працюють із неоднорідними ознаками, стійкі до шуму, забезпечують швидкий інференс та дозволяють інтерпретувати внесок окремих ознак у рішення, не перетворюючи модуль на "чорну скриньку".

Прогнозування виконується не для одного наперед визначеного сценарію, а для множини допустимих кандидатних дій. Фактично система реалізує what-if analysis: для кожного варіанта зміни конфігурації контейнерного середовища оцінюється прогнозований вплив на переносимість, і лише після цього вибирається оптимальний керуючий вплив. Завдяки цьому машинне навчання використовується не просто для пасивного аналізу стану, а для активного підтримання інваріантності поведінки програмної системи відносно еталонного виконання.

Під політикою керування π розуміється правило прийняття рішень, яке на основі поточного стану програмної системи, результатів оцінювання та прогнозування, множини допустимих керуючих дій, а також обмежень системи оркестрації та політик безпеки визначає керуючий вплив, що має бути застосований на поточній ітерації керуючого циклу.

Політика керування π також набуває більш глибокого змісту, якщо розглядати її саме як результат роботи інтелектуального модуля. У класичних системах контейнерної оркестрації політика керування зазвичай зводиться до набору жорстко заданих правил. У даному разі вона працює як decision-making layer, що поєднує результати оцінювання, прогнозування, перевірку обмежень та логіку вибору дії. Практична реалізація показує, що навіть за використання базового greedy-алгоритму вибору рішення

приймається не за миттєвим локальним критерієм, а з урахуванням прогнозованого значення відхилення переносимості, обмежень оркестратора, вимог доступності сервісу та допустимості конфігураційних змін.

Ще одним важливим аспектом є поєднання машинного навчання з механізмами стабілізації керуючого циклу. Базова умова стабілізації полягає в тому, що керуюча дія застосовується лише тоді, коли прогнозоване відхилення не зростає порівняно з поточним станом. На практиці ця ідея доповнюється механізмами hysteresis та cooldown, що зменшують ризик надмірної реактивності та конфігураційних коливань. Машинне навчання вбудоване не в ізольований аналітичний блок, а в повноцінну систему адаптивного керування, де прогностичні оцінки проходять додатковий контур валідації перед фактичним застосуванням до контейнерного середовища. Саме завдяки цьому інтелектуальний модуль не лише підвищує точність рішень, а й зберігає стабільність експлуатації системи.

У межах архітектури методу машинне навчання також безпосередньо пов'язане з механізмом накопичення досвіду. Замкнений керуючий цикл розглядається як послідовність спостереження, аналізу, планування та виконання, що технічно реалізується через базу знань (knowledge base) і використання історичних даних для подальшого оновлення моделей. Отже, інтелектуальний модуль не є статичним компонентом: його якість потенційно зростає в процесі експлуатації завдяки накопиченню нових прикладів станів системи, прийнятих рішень і фактичних наслідків цих рішень.

Часовий аспект застосування машинного навчання також є принциповим. Декомпозиція періоду керуючого циклу на послідовні етапи моніторингу, аналізу, прийняття рішення та виконання означає, що модель машинного навчання повинна бути не лише достатньо точною, але й достатньо швидкою, щоб її можна було інтегрувати у цикл керування, наближений до реального часу (near real-time control). Саме це враховано у побудові програмного прототипу: збір метрик відбувається у ковзному вікні,

інференс моделі виконується оперативно, а часові параметри Δ і W (ширина ковзного часового вікна спостереження) вибираються як компроміс між швидкістю реакції та стійкістю керування. Таким чином, машинне навчання подане не лише як інтелектуальний, а й як часово узгоджений компонент керуючої системи.

Враховані також гетерогенності обчислювальних платформ, включаючи відмінності у процесорних архітектурах, конфігурації пам’яті, наявності апаратних прискорювачів та особливостях операційних систем. Вони враховуються автоматично як вхідні параметри інтелектуального модуля управління.

Також є інтеграція замкненого циклу адаптації, у якому рішення щодо конфігурації контейнерів приймаються на основі постійного збору та аналізу метрик виконання. Такий цикл дозволяє не лише реагувати на поточні зміни умов виконання, але й використовувати історичні дані для покращення якості рішень у майбутньому, що відсутнє у класичних методах контейнеризації.

У межах однієї платформи, метод безпосередньо спрямований на забезпечення переносимості програмних систем між різними обчислювальними середовищами. Адаптація контейнерів розглядається як засіб досягнення стабільної та ефективної роботи програм незалежно від цільової платформи, а не лише як спосіб балансування навантаження.

2.3 Принципи реалізації адаптивної контейнеризації

Однією з ключових складових адаптивної контейнеризації є безперервний збір та аналіз метрик виконання програмних систем. До таких метрик належать показники використання процесорного часу, оперативної пам’яті, мережевих ресурсів, а також характеристики затримок і пропускної здатності. Збір метрик здійснюється на рівні контейнерів і оркестратора, що дозволяє отримувати актуальну інформацію про стан системи у реальному

часі та використовувати її як вхідні дані для інтелектуальних алгоритмів управління. [94]

У результаті безперервного моніторингу кластера накопичуються великі дані, обробка яких вимагає оптимізованих аналітичних підходів та ефективного зберігання.

Науковим підґрунтям прийняття рішень є використання алгоритмів машинного навчання для аналізу багатовимірних даних та виявлення прихованих залежностей між параметрами середовища виконання і продуктивністю програм. Машинне навчання дозволяє перейти від жорстко заданих правил адаптації до моделей, здатних узагальнювати досвід попередніх виконань і формувати прогностичні оцінки майбутнього навантаження. [32]

У межах формалізованої моделі керування алгоритми машинного навчання використовуються для апроксимації залежності між станом програмної системи, профілем платформи виконання та відхиленням переносимості.

Вводиться функція оцінювання поточного стану:

$$\hat{D}(t) = E\big(x(t), P\big), \tag{2.8}$$

де

$\hat{D}(t)$ — оцінене (апроксимоване) значення відхилення переносимості програмної системи у момент часу t;

$E()$ — функція оцінювання, реалізована за допомогою моделі машинного навчання.

На практиці це означає, що функція $E()$ є програмним інкапсулюванням навченої регресійної моделі, наприклад, лінійної регресії, деревоподібної моделі типу Random Forest чи Gradient Boosting або неглибокої нейронної мережі. З алгоритмічної точки зору обчислення

значення цієї функції зводиться до процесу інференсу (прямого проходу) навченої моделі.

Важливим елементом методу є використання історичних даних про виконання програмних систем. Накопичення таких даних дозволяє будувати моделі поведінки програм у різних умовах та використовувати їх для прогнозування потреб у ресурсах. Це забезпечує проактивний характер адаптації, коли параметри контейнерів коригуються не лише у відповідь на поточні перевантаження, але й з урахуванням очікуваних змін навантаження.

Окрім оцінювання поточного стану, алгоритми машинного навчання використовуються для прогнозування впливу керуючих дій на поведінку програмної системи.

Для цього вводиться прогностична функція:

$$\Omega(t,u) = F\big(x(t), P, u\big), \tag{2.9}$$

де

$\Omega(t,u)$ — прогнозоване значення відхилення переносимості після застосування кандидатного керуючого впливу u через один період керування;

$F()$ — прогностична функція, реалізована за допомогою моделі машинного навчання.

Функція $F()$ інкапсулює у собі навчену регресійну модель машинного навчання, яка розв'язує задачу керованої регресії. З обчислювальної та алгоритмічної точок зору робота цієї функції зводиться до проактивного багатосценарного аналізу за принципом симуляції наслідків майбутніх дій. У процесі такої симуляції модель генерує імовірнісні оцінки щодо майбутнього стану середовища, що дозволяє мінімізувати ризики вибору хибних конфігурацій.

Метод також спирається на концепцію замкненого циклу управління, у межах якого реалізується послідовність етапів моніторингу, аналізу, прийняття рішень та застосування керуючих впливів. Такий підхід є характерним для адаптивних і саморегульованих систем та дозволяє забезпечити стійкість і стабільність роботи програмних систем у динамічних середовищах.

З інженерної точки зору адаптивна контейнеризація реалізується шляхом інтеграції інтелектуального модуля управління з існуючими контейнерними платформами та системами оркестрації. Це дозволяє використовувати стандартні механізми управління ресурсами контейнерів, зокрема обмеження на CPU та пам'ять, як керуючі параметри, не порушуючи сумісність з існуючими інфраструктурами.

Прийняття керуючих рішень здійснюється з урахуванням обмежень середовища виконання та політик оркестрації.

2.4 Алгоритми та технології адаптивної контейнеризації

Реалізація адаптивної контейнеризації базується на поєднанні алгоритмічних підходів аналізу даних, прогнозування та прийняття рішень із сучасними контейнерними та оркестраційними технологіями. Алгоритмічна складова методу орієнтована на обробку потоків метрик у реальному часі та формування керуючих впливів з урахуванням поточного стану системи і накопиченого досвіду виконання програм.

Основним джерелом вхідних даних для алгоритмів адаптації є метрики виконання контейнеризованих додатків, що включають показники використання CPU, оперативної пам'яті, мережевої пропускної здатності, дискових операцій, а також часові характеристики виконання. Збір таких метрик здійснюється з використанням стандартних механізмів моніторингу контейнерних платформ і систем оркестрації, що забезпечує сумісність методу з існуючими інфраструктурами.

Для аналізу багатовимірних метрик виконання у методі застосовуються алгоритми машинного навчання, які здатні виявляти нелінійні залежності між параметрами середовища виконання та поведінкою програмної системи. Використання моделей машинного навчання дозволяє перейти від жорстко заданих евристик до адаптивних моделей, які коригують свої параметри на основі накопичених даних та забезпечують більш точне прогнозування майбутнього навантаження.

Алгоритмічна логіка методу передбачає використання як реактивних, так і проактивних механізмів адаптації. Реактивна адаптація реалізується шляхом коригування параметрів контейнерів у відповідь на поточні перевантаження або деградацію продуктивності. Проактивна адаптація базується на прогнозуванні майбутніх змін навантаження з використанням історичних даних та відповідному попередньому налаштуванні середовища виконання.

Важливим аспектом алгоритмічної реалізації є врахування обмежень та політик оркестрації. Прийняття рішень щодо зміни параметрів контейнерів здійснюється у межах допустимих значень, визначених системою оркестрації, що забезпечує коректність і безпечність застосування керуючих впливів. Таким чином, алгоритми адаптації не порушують цілісність контейнерної інфраструктури та не потребують модифікації базових механізмів оркестрації.

З урахуванням зазначених обмежень керуючі дії, що застосовуються до середовища виконання, повинні належати множині допустимих впливів:

$$u(t) \in U\big(x(t), P\big), \tag{2.10}$$

де

$U\big(x(t), P\big)$ — множина допустимих керуючих впливів, сформована з урахуванням стану системи $x(t)$ та профілю платформи P.

Основою реалізації методу є використання контейнерних платформ і систем оркестрації, що підтримують програмний доступ до параметрів керування ресурсами. Зокрема, стандартні механізми обмеження ресурсів контейнерів використовуються як керуючі змінні, що дозволяє інтегрувати алгоритмічний модуль адаптації без втручання у прикладний код програмних систем.

Для забезпечення масштабованості та відмовостійкості алгоритмічних компонентів методу використовується модульна архітектура, у межах якої компоненти збору метрик, аналізу даних та прийняття рішень можуть розгортатися як окремі сервіси. Такий підхід дозволяє адаптувати алгоритмічну складову до різних масштабів системи та забезпечує її переносимість між різними обчислювальними платформами.

Алгоритмічні аспекти методу також враховують вимоги до затримок прийняття рішень. Оскільки адаптація середовища виконання має відбуватися у режимі, близькому до реального часу, використовувані алгоритми мають обмежену обчислювальну складність та можуть бути виконані у середовищах з обмеженими ресурсами. Це забезпечує можливість застосування методу у широкому спектрі платформ, включаючи хмарні, мультихмарні та периферійні середовища.

Таким чином, алгоритмічні аспекти та технології, що використовуються у методі адаптивної контейнеризації, поєднують методи машинного навчання, потоковий аналіз метрик та стандартні механізми контейнерних платформ. Це забезпечує практичну реалізовність методу, його сумісність з існуючими інфраструктурами та узгодженість з науковою новизною.

Прогностична функція (2.9) дозволяє оцінити вплив різних варіантів адаптації середовища виконання на досягнення інваріантності поведінки програмної системи відносно еталонного виконання.

Політика керування π використовує результати функцій $E()$ та $F()$ для вибору оптимального керуючого впливу, який мінімізує прогнозоване відхилення переносимості за умови виконання обмежень системи оркестрації та політик безпеки. Таким чином, алгоритми машинного навчання застосовуються не для локальної оптимізації окремих метрик продуктивності, а для інтелектуального аналізу та компенсації платформно-індукованих відмінностей між різними обчислювальними платформами.

Формально вибір керуючого впливу у межах однієї ітерації керуючого циклу формалізується як задача мінімізації прогнозованого відхилення переносимості.

$$u^*(t) = \arg\min_{u \in U\left(x(t),P\right)} \Omega(t,u), \tag{2.11}$$

де

$u^*(t)$ — оптимальний керуючий вплив, обраний у момент часу t.

Для розв'язання цієї оптимізаційної задачі не застосовуються класичні аналітичні чи градієнтні методи пошуку мінімуму, оскільки простір допустимих керуючих впливів має дискретний характер і жорстко обмежується правилами системи оркестрації. Безпосереднє знаходження мінімуму здійснюється за допомогою алгоритму жадібного (greedy) пошуку в дискретному просторі. Спочатку генеруються кандидати на горизонтальне та вертикальне масштабування, які фільтруються через обмеження оркестратора (доступні ресурси вузла, політики PodDisruptionBudget). Для кожного валідного кандидата u формується вектор-функція ознак $\varphi(t)$, який передається на вхід регресійній моделі машинного навчання. Модель здійснює швидкий інференс, повертаючи очікуване відхилення $\Omega(t,u)$. Алгоритм обирає дію, що забезпечує мінімальне значення функції.

Поетапно цей процес виглядає наступним чином:

1. Перший етап це генерація та фільтрація простору кандидатів. Інтелектуальний модуль (decision-making layer) спочатку формує словник кандидатних дій (candidate actions), куди входять варіанти вертикальної адаптації (зміна значень CPU та memory limits/requests) та горизонтальної адаптації (зміна кількості реплік Deployment). Кожен згенерований кандидат проходить фільтрацію на відповідність інфраструктурним обмеженням множини $U(x(t),P)$. З простору пошуку виключаються дії, що порушують ресурсні ліміти кластера, мінімальну/максимальну кількість реплік, вимоги доступності сервісу або політики PodDisruptionBudget.
2. Другий етап це формування прогнозних векторів-функцій ознак (Feature Engineering). Для кожної валідної дії u система конструює прогнозований вектор-функцію ознак $\varphi(t)$. Ця вектор-функція є компактним числовим представленням, яке відображає очікувану конфігурацію середовища після застосування дії. До нього входять нормалізовані метрики виконання, нові параметри контейнерної конфігурації та закодований профіль цільової платформи P (наприклад, перетворення категоріальної ознаки процесорної архітектури x86_64/ARM64 у числову форму за допомогою one-hot encoding).
3. Третій етап це швидкий інференс прогностичної моделі. Сформована множина векторів-функцій $\varphi(t)$ передається на вхід прогностичній моделі машинного навчання (наприклад, деревоподібній регресійній моделі Random Forest). Модель здійснює прямий прохід (inference) і для кожного сценарію розраховує очікуване кількісне значення відхилення переносимості $\Omega(t,u)$. Оскільки загальний час ітерації керуючого циклу суворо обмежений, швидкість інференсу відіграє критичну роль для забезпечення прийняття рішень у режимі, близькому до реального часу (near real-time control).

4. Четвертий етап це жадібний вибір оптимуму та стабілізація. Політика керування π здійснює вибір оптимального керуючого впливу u^*, застосовуючи жадібний (greedy) алгоритм, який обирає дію з мінімальним прогнозованим значенням $\Omega(t,u)$. Для забезпечення стабільності керування (control stability) та уникнення надмірних коливань конфігурації (configuration thrashing), вибір може доповнюватися механізмами гістерезису (hysteresis) та періоду охолодження (cooldown period). Дія застосовується лише тоді, коли прогнозоване зменшення відхилення перевищує заданий поріг, а з моменту останньої адаптації минув мінімальний час. Крім того, якщо кілька дій забезпечують близькі прогнозні показники, пріоритет віддається тим впливам, що потребують мінімальних змін конфігурації або не призводять до перезапуску контейнерів.
5. П’ятий етап це декларативне застосування впливу. Обраний вплив u^* передається до рівня управління адаптацією (actuation layer), який не вносить зміни безпосередньо в контейнери, а формує декларативний запит (patch або update) до Kubernetes API Server. Далі оркестратор самостійно виконує процес reconciliation (наприклад, через механізм rolling update), плавно приводячи фактичний стан системи до нового бажаного стану без порушення доступності сервісу.

Наявні обмеження:

1. Ресурсні обмеження: мінімально та максимально допустимі значення ресурсних параметрів контейнерів (resource limits та requests для CPU і пам'яті);
2. Обмеження масштабування: допустимі діапазони кількості реплік додатку (мінімальна та максимальна кількість);
3. Обмеження доступності сервісу: вимоги до цілісності сервісів та політики переривання роботи (PodDisruptionBudget); [20]

4. Інфраструктурні обмеження: загальні ресурсні обмеження кластера та політики розміщення/безпеки.

З метою уточнення алгоритмічної сутності методу керування адаптивною контейнеризацією алгоритм прийняття рішень може бути представлений у вигляді послідовності таких кроків:

На першому етапі здійснюється безперервний збір метрик виконання контейнеризованої програмної системи у межах ковзного часового вікна W, в результаті чого формується поточна вектор-функція метрик $m(t)$.

На другому етапі на основі зібраних метрик та профілю цільової обчислювальної платформи P формується вектор-функція ознак $\varphi(t)$, який включає як миттєві значення експлуатаційних характеристик, так і агреговані показники, що відображають динаміку виконання програмної системи.

На третьому етапі за допомогою інтелектуального модуля аналізу здійснюється оцінювання відхилення переносимості $D(t)$, яке характеризує ступінь відхилення поведінки програмної системи від еталонного виконання або допустимого експлуатаційного інтервалу. На цьому етапі алгоритми машинного навчання використовуються для виявлення платформно-індукованих відмінностей у поведінці програмної системи за однакових умов навантаження.

На четвертому етапі формується множина допустимих кандидатних керуючих впливів u, які відповідають обмеженням системи оркестрації та включають як вертикальну адаптацію параметрів контейнерів (resource limits і requests), так і горизонтальну адаптацію шляхом зміни кількості реплік. До таких керуючих впливів належать зміни ресурсних обмежень контейнерів, кількості реплік або параметрів розміщення.

На п’ятому етапі для кожного кандидатного керуючого впливу здійснюється прогнозування очікуваного значення відхилення переносимості $\Omega(t,u)$ з урахуванням профілю платформи виконання. Прогнозування може

реалізовуватися за допомогою моделей машинного навчання або гібридних підходів, що поєднують прогнозні моделі з евристичними правилами.

На шостому етапі обирається керуючий вплив u^*, який забезпечує мінімальне прогнозоване відхилення переносимості при виконанні всіх заданих обмежень. Обраний керуючий вплив застосовується до контейнерного середовища виконання через стандартні інтерфейси системи оркестрації.

Застосування керуючого впливу ініціює перехід системи у новий стан, який спостерігається на наступній ітерації керуючого циклу:

$$X(t) \to x(t+\Delta), \tag{2.12}$$

На завершальному етапі результати застосування керуючих дій фіксуються та додаються до історичних даних H, що дозволяє уточнювати моделі аналізу та політику керування у процесі експлуатації. Такий підхід забезпечує адаптивність алгоритму та його здатність до самонавчання в умовах змінних характеристик обчислювальних платформ.

Для запобігання деградації поведінки програмної системи керуючий вплив застосовується лише у випадку, якщо прогнозоване відхилення переносимості не зростає у порівнянні з поточним станом:

$$\Omega(t,u) \le \hat{D}(t), \tag{2.13}$$

Формалізація переносимості та алгоритмічна схема керування створюють основу для кількісної експериментальної оцінки методу, зокрема шляхом аналізу динаміки функції $D(t)$ у різних платформах та сценаріях навантаження.

2.5 Архітектура методу адаптивної контейнеризації

Архітектура методу адаптивної контейнеризації побудована з урахуванням вимог до переносимості, масштабованості та сумісності з існуючими контейнерними платформами і системами оркестрації. [18] Основною метою архітектури є забезпечення замкненого циклу автоматизованого управління середовищами виконання програм, який дозволяє динамічно адаптувати параметри контейнерів до характеристик обчислювальної платформи та умов експлуатації.

Функціонування методу базується на реалізації замкненого керуючого циклу, у межах якого здійснюється спостереження, аналіз стану системи, прийняття рішень та застосування керуючих впливів.

Формально керуючий цикл може бути представлений як відображення:

$$(x(t), P) \to u(t) \to x(t+\Delta), \tag{2.14}$$

Особливість застосування алгоритмів машинного навчання у даному відображенні полягає у забезпеченні нелінійного пошуку ефективного керуючого впливу $u(t)$. Інтелектуальний модуль виконує функцію складного апроксиматора, який на основі поточного стану системи $x(t)$ та характеристик цільової платформи P визначає таку керуючу дію, що гарантує перехід об'єкта у стабільний цільовий стан $x(t+\Delta)$ з мінімальним відхиленням від еталонної поведінки.

Архітектурно метод реалізується у вигляді багатокомпонентної системи, що інтегрується з контейнерною інфраструктурою без втручання у прикладний код програмних систем. [56] Це забезпечує прозорість застосування методу та дозволяє використовувати його у вже існуючих середовищах розгортання без необхідності їх суттєвої модифікації.

Центральним елементом архітектури є модуль збору та агрегації метрик, який відповідає за отримання інформації про стан контейнеризованих додатків і середовища виконання. До таких даних належать показники використання процесорних ресурсів, оперативної пам'яті, мережевих інтерфейсів, а також часові характеристики виконання програм. Модуль збору метрик використовує стандартні механізми моніторингу контейнерних платформ та систем оркестрації, що забезпечує його універсальність і переносимість.

З точки зору теорії керування, реалізований керуючий цикл відповідає класичній схемі MAPE (Monitor–Analyze–Plan–Execute), що може бути формалізовано як послідовність операцій:

$$x(t) \to m(t) \to \hat{D}(t) \to u^*(t) \to x(t+\Delta), \tag{2.15}$$

У межах деталізованої схеми керування машинне навчання функціонально локалізується на етапах аналізу та планування. На цих етапах навчені регресійні моделі здійснюють кількісну апроксимацію поточного відхилення переносимості $D(t)$ та виконують проактивний аналіз сценаріїв шляхом прогнозування експлуатаційних наслідків для обраної кандидатних керуючих дій $u^*(t)$ ще до моменту їх безпосереднього застосування до системи оркестрації.

Наступним компонентом архітектури є модуль аналізу даних, який здійснює попередню обробку, нормалізацію та формування вхідних векторів-функцій для інтелектуального модуля прийняття рішень. На цьому етапі відбувається узгодження різнорідних метрик, усунення шуму та формування представлення стану системи, придатного для подальшого аналізу.

Інтелектуальний модуль прийняття рішень є ядром архітектури адаптивної контейнеризації. Він реалізує алгоритми машинного навчання, що використовуються для оцінювання поточного стану системи, прогнозування

майбутнього навантаження та визначення оптимальних параметрів контейнерів. Рішення, сформовані цим модулем, спрямовані на досягнення стабільної та ефективної роботи програмних систем у межах забезпечення переносимості як первинної цільової властивості.

У межах архітектури політика керування реалізується як відображення стану системи та профілю платформи у керуючий вплив:

$$u(t) = \pi\left(x(t), P\right), \tag{2.16}$$

де

$u(t)$ — керуючий вплив, що формується інтелектуальним модулем;

Компонент керування адаптацією відповідає за перетворення рішень інтелектуального модуля у конкретні керуючі дії щодо контейнерів. До таких дій належать зміна обмежень на використання ресурсів, ініціювання перерозподілу навантаження або коригування параметрів розгортання. Реалізація цього компонента базується на використанні стандартних інтерфейсів управління ресурсами контейнерів і не порушує сумісність з існуючими механізмами оркестрації.

Важливим елементом архітектури є модуль зберігання історичних даних, який накопичує інформацію про попередні стани системи, прийняті рішення та їх наслідки. Наявність такого модуля дозволяє реалізувати навчання на історичних даних, покращувати якість прогнозування та забезпечувати еволюційний розвиток адаптивного механізму у процесі експлуатації.

Архітектура методу також передбачає інтеграцію з системами оркестрації, що забезпечують виконання керуючих дій у розподіленому середовищі. Така інтеграція дозволяє застосовувати метод у кластерах різного масштабу, у тому числі у хмарних та мультихмарних

інфраструктурах, зберігаючи при цьому єдиний підхід до управління переносимістю.

З точки зору масштабованості архітектура методу підтримує розгортання її компонентів як окремих сервісів, що дозволяє адаптувати обчислювальні витрати на аналіз і прийняття рішень до розміру та складності системи. [79] Це забезпечує можливість використання методу як у невеликих середовищах з обмеженими ресурсами, так і у великих розподілених системах.

Таким чином, архітектура методу адаптивної контейнеризації забезпечує структуровану та розширювану основу для реалізації інтелектуального управління середовищами виконання програм. Її модульна побудова, сумісність з існуючими контейнерними платформами та орієнтація на динамічну адаптацію створюють передумови для формалізації методу, опису алгоритмічних компонентів і проведення експериментальних досліджень.

2.5.1 Опис компонентів архітектури

Архітектура методу адаптивної контейнеризації складається з набору функціонально відокремлених компонентів, кожен з яких виконує визначену роль у процесі автоматизованого управління середовищами виконання програм. Поелементний опис архітектури дозволяє формалізувати взаємодію компонентів, забезпечити модульність реалізації та спростити масштабування методу у різних обчислювальних середовищах.

1. Компонент збору метрик призначений для отримання актуальної інформації про стан контейнеризованих програм та інфраструктури виконання. До його функцій належить збір показників використання процесорного часу, оперативної пам’яті, мережевих ресурсів, дискових операцій, а також часових характеристик виконання програм. Компонент використовує стандартні механізми моніторингу

контейнерних платформ і систем оркестрації, що забезпечує його сумісність з різними реалізаціями контейнерної інфраструктури.

2. Компонент попередньої обробки та агрегації даних виконує нормалізацію, фільтрацію та узгодження зібраних метрик. Оскільки дані надходять з різних джерел і характеризуються різною частотою оновлення та масштабами значень, даний компонент формує уніфіковане представлення стану системи, придатне для подальшого інтелектуального аналізу. На цьому етапі також здійснюється автоматизоване вилучення інформації щодо критичних аномалій з потоку сирих даних для своєчасного реагування контролера. Це дозволяє зменшити вплив шуму та підвищити стабільність роботи методу.
3. Інтелектуальний компонент аналізу та прийняття рішень є ядром архітектури адаптивної контейнеризації. Він реалізує алгоритми машинного навчання, які використовуються для оцінювання поточного стану системи, виявлення тенденцій у поведінці програм та прогнозування майбутніх змін навантаження. На основі сформованих моделей компонент генерує рекомендації щодо адаптації параметрів контейнерів.
4. Компонент управління адаптацією відповідає за трансляцію рішень інтелектуального компонента у конкретні керуючі дії щодо контейнеризованих середовищ виконання. До таких дій належать коригування обмежень на використання ресурсів контейнерів, зміна параметрів розгортання або ініціювання перерозподілу навантаження. Реалізація цього компонента базується на використанні стандартних інтерфейсів управління ресурсами та не потребує модифікації прикладного коду.
5. Компонент зберігання та управління історичними даними призначений для накопичення інформації про попередні стани системи, прийняті рішення та результати їх застосування. Використання історичних даних

дозволяє реалізувати навчання на досвіді експлуатації та поступово підвищувати ефективність адаптивного механізму.

6. Компонент інтеграції з системою оркестрації забезпечує взаємодію методу з контейнерною інфраструктурою у розподіленому середовищі. Він відповідає за застосування керуючих дій з урахуванням політик оркестрації та обмежень середовища виконання, що дозволяє використовувати метод у хмарних, мультихмарних і гібридних інфраструктурах.

7. Компонент масштабування та розгортання архітектури забезпечує можливість розміщення елементів методу як окремих сервісів з незалежним керуванням обчислювальними ресурсами. Такий підхід дозволяє адаптувати метод до різних масштабів системи та забезпечити його ефективну роботу як у середовищах з обмеженими ресурсами, так і у великих розподілених системах.

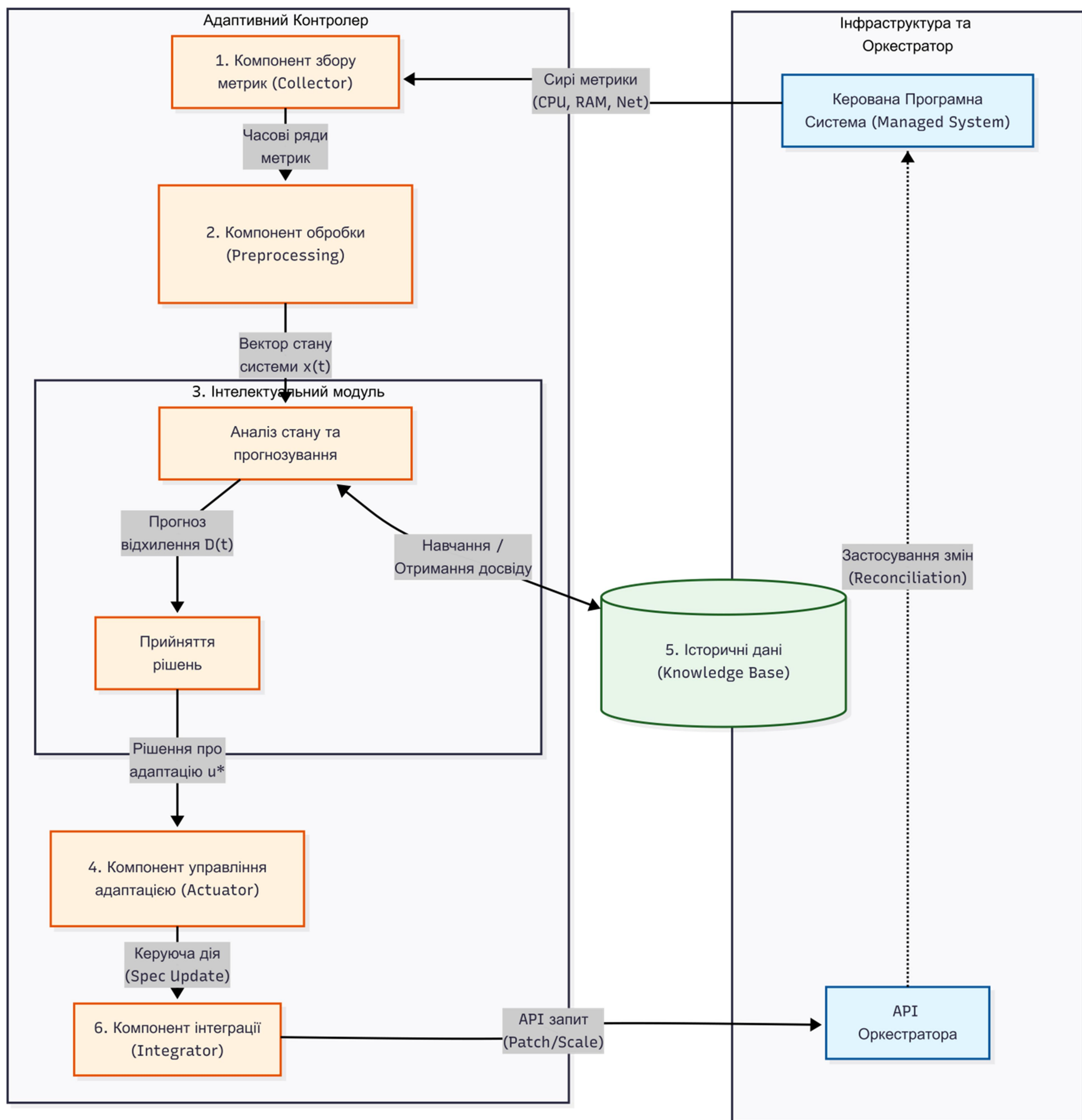

Рисунок 2.1. Структурна архітектура методу адаптивної контейнеризації та взаємодія його компонентів із керованою програмною системою й оркестратором.

Таким чином, поелементний опис компонентів архітектури демонструє, що метод адаптивної контейнеризації базується на чітко структурованій, модульній та розширюваній архітектурі. Використання нумерованої декомпозиції дозволяє формалізувати роль кожного компонента

та створює основу для подальшої формалізації алгоритмів, опису взаємодії компонентів і експериментальної перевірки ефективності методу.

2.5.2 Взаємодія компонентів та потоки даних

Взаємодія компонентів архітектури адаптивної контейнеризації реалізується у вигляді замкненого циклу управління, у межах якого здійснюється безперервний обмін даними між модулями збору інформації, аналізу, прийняття рішень та застосування керуючих впливів.

З урахуванням часових затримок збору метрик, аналізу та застосування керуючих впливів архітектура забезпечує узгодженість компонентів у межах кожної ітерації керуючого циклу:

$$\Delta = \Delta_{\text{monitor}} + \Delta_{\text{analyze}} + \Delta_{\text{decide}} + \Delta_{\text{execute}}, \tag{2.17}$$

де

Δ — загальний період керуючого циклу;

Δ_{monitor} — затримка збору та агрегації метрик;

Δ_{analyze} — час аналізу стану системи;

Δ_{decide} — час прийняття керуючого рішення;

Δ_{execute} — затримка застосування керуючого впливу через систему оркестрації.

Особливість застосування алгоритмів машинного навчання у контексті даної декомпозиції періоду керуючого циклу полягає у наявності жорстких вимог до швидкодії інтелектуального модуля. Оскільки загальний час ітерації обмежений, тривалість інференсу регресійних моделей, що функціонально входить до етапів аналізу стану системи Δ_{analyze} та прийняття керуючого рішення Δ_{decide}, має бути мінімальною. Це означає, що навчені моделі

машинного навчання повинні бути не лише високоточними, але й обчислювально ефективними для успішної інтеграції у цикл керування, який функціонує в режимі, наближеному до реального часу. Таким чином, машинне навчання виступає не лише як аналітичний механізм, а й як суворо часово узгоджений компонент керуючої системи, де швидкість генерації прогностичних оцінок та прийняття рішень збалансована зі стійкістю загального процесу адаптації контейнерного середовища.

Така організація потоків даних забезпечує динамічну адаптацію середовищ виконання програм до змінних умов експлуатації та характеристик обчислювальної платформи.

1. Початковим етапом взаємодії є формування потоку первинних даних у компоненті збору метрик. На цьому етапі здійснюється отримання телеметричної інформації про стан контейнерів, використання ресурсів та характеристики виконання програм. Дані надходять у вигляді часових рядів з фіксованою або адаптивною частотою оновлення.
2. Зібрані метрики передаються до компонента попередньої обробки та агрегації даних, де виконується їх нормалізація, синхронізація у часі та усунення надлишкових або шумових значень. На цьому етапі формується уніфікована вектор-функція стану системи, який відображає поточну конфігурацію середовища виконання та поведінку програмної системи.
3. Сформована вектор-функція стану передається до інтелектуального компонента аналізу та прийняття рішень. У цьому компоненті виконується аналіз поточного стану системи, порівняння його з історичними даними та формування прогнозних оцінок майбутнього навантаження. На основі результатів аналізу визначаються рекомендації щодо коригування параметрів контейнерів.
4. Результати роботи інтелектуального компонента у вигляді набору керуючих рішень надходять до компонента управління адаптацією. Даний компонент здійснює трансляцію логічних рекомендацій у

конкретні керуючі дії, такі як зміна ресурсних обмежень контейнерів або ініціювання перерозподілу навантаження.

5. Керуючі дії застосовуються до контейнерної інфраструктури через компонент інтеграції з системою оркестрації. На цьому етапі відбувається взаємодія з механізмами управління ресурсами та політиками оркестрації, що забезпечує коректність і безпечність змін у середовищі виконання.
6. Інформація про застосовані керуючі дії та їх результати передається до компонента зберігання історичних даних. Цей компонент накопичує відомості про динаміку станів системи, прийняті рішення та їх вплив на продуктивність і стабільність програмних систем.
7. Накопичені історичні дані використовуються для оновлення моделей інтелектуального компонента аналізу та прийняття рішень. Таким чином реалізується механізм зворотного зв’язку, що забезпечує самонавчання методу та поступове підвищення якості адаптації у процесі експлуатації.
8. Замкнений цикл взаємодії компонентів повторюється з визначеною періодичністю або у відповідь на суттєві зміни стану системи. Частота оновлення потоків даних та прийняття рішень може адаптуватися залежно від характеристик середовища виконання та вимог до швидкодії.

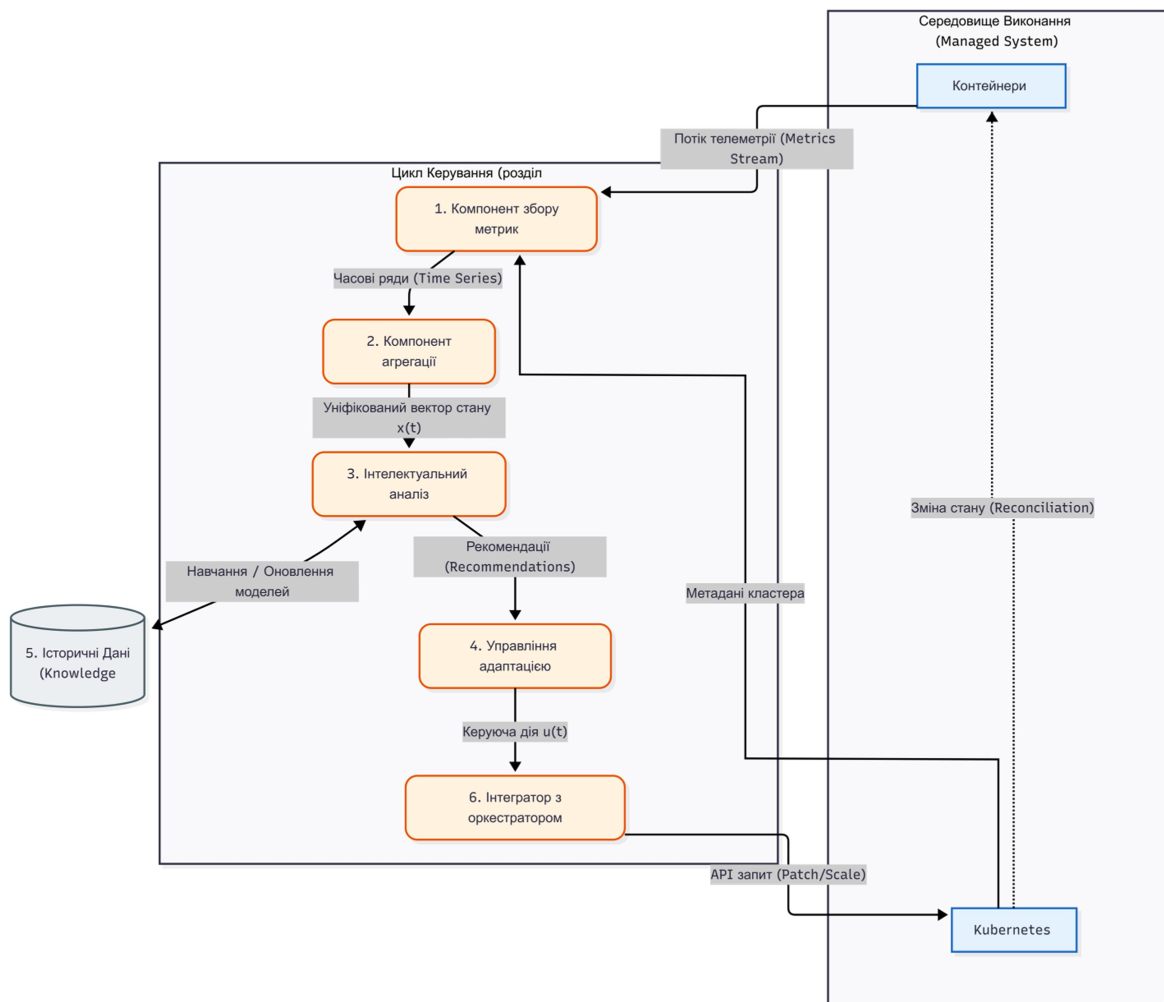

Рисунок 2.2. Схема взаємодії компонентів і потоків даних у замкненому керуючому циклі адаптивної контейнеризації.

Таким чином, організація взаємодії компонентів та потоків даних у методі адаптивної контейнеризації забезпечує безперервний, керований та самонавчальний процес адаптації середовищ виконання. Це дозволяє підтримувати стабільну та ефективну роботу програмних систем у гетерогенних обчислювальних середовищах і створює основу для подальшої формалізації алгоритмів та експериментальної перевірки методу.

2.5.3 Рівні архітектури та межі відповідальності компонентів

Архітектура методу адаптивної контейнеризації організована за багаторівневим принципом, що дозволяє чітко розмежувати відповідальність компонентів та забезпечити масштабованість і переносимість реалізації. [87] Виділення рівнів архітектури є необхідним для формалізації взаємодії між контейнерною інфраструктурою, інтелектуальним модулем управління та системами оркестрації.

Інфраструктурний рівень включає фізичні та віртуальні обчислювальні ресурси, операційну систему хоста та контейнерний runtime. На цьому рівні реалізуються механізми ізоляції процесів, управління ресурсами та виконання контейнерів, зокрема через Linux kernel subsystems, такі як cgroups та namespaces, або їх аналоги у віртуалізованих середовищах. Даний рівень є платформозалежним, проте його інтерфейси абстрагуються на вищих рівнях архітектури.

Рівень контейнерного виконання відповідає за безпосередній запуск програмних компонентів у вигляді контейнерів. Тут визначаються container images, параметри середовища виконання, політики доступу до ресурсів та мережеві налаштування. [83] Архітектурно цей рівень не містить логіки адаптації, а виступає об’єктом керування з боку інтелектуального механізму.

Рівень оркестрації реалізує управління життєвим циклом контейнерів у розподіленому середовищі. Системи типу Kubernetes забезпечують абстракцію над інфраструктурою, виконуючи планування, масштабування, балансування навантаження та відновлення контейнерів. [70] Оркестратор розглядається як виконавчий механізм, через який застосовуються керуючі рішення.

Аналітико-інтелектуальний рівень включає компоненти збору метрик, аналізу даних та прийняття рішень. Саме на цьому рівні реалізується логіка адаптивної контейнеризації, включаючи побудову моделей, прогнозування навантаження та визначення оптимальних параметрів середовища виконання. Відокремлення цього рівня забезпечує переносимість методу між різними контейнерними платформами.

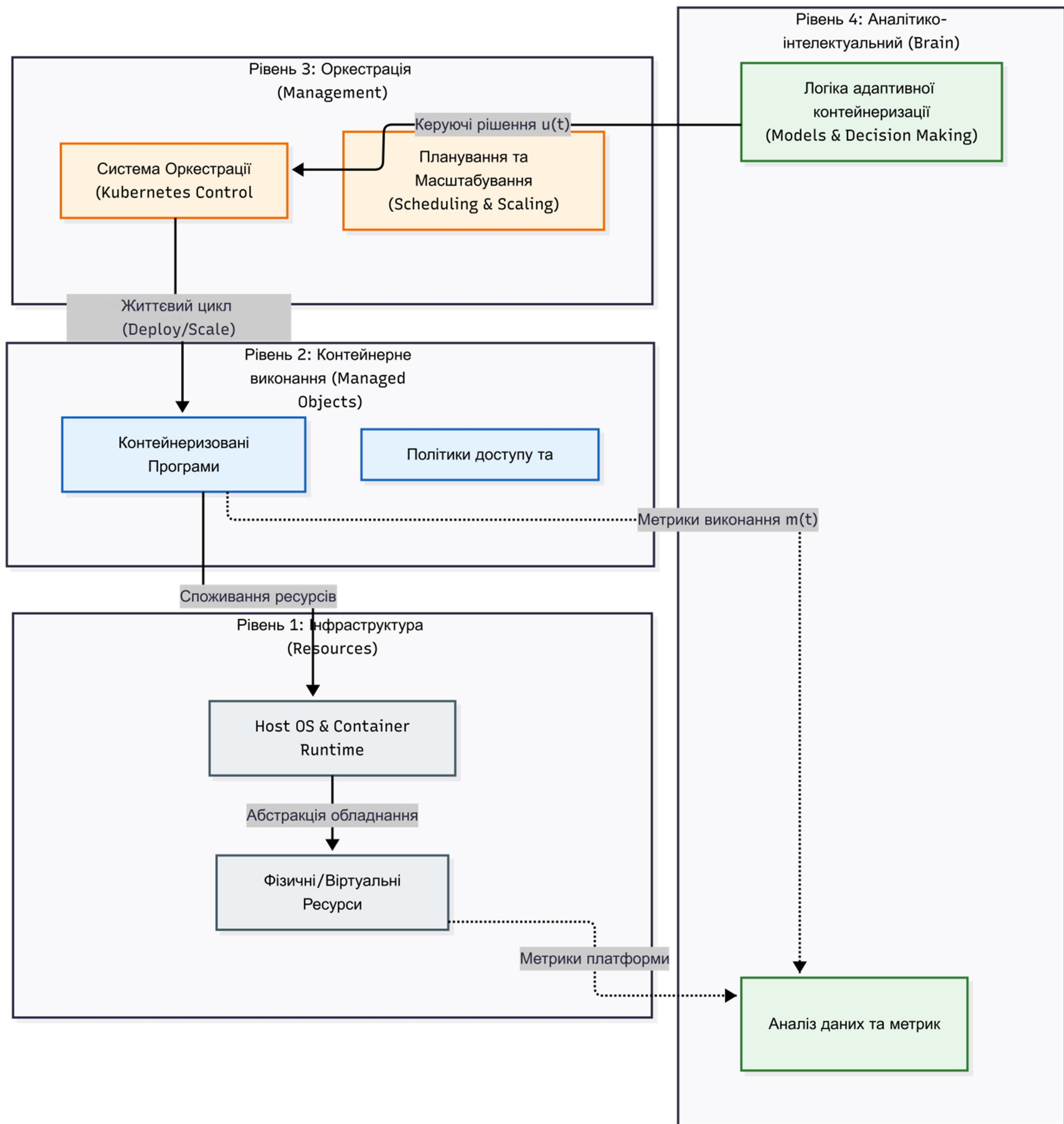

Рисунок 2.3. Рівні архітектури методу адаптивної контейнеризації та межі відповідальності його компонентів.

2.5.4 Формалізація керуючого циклу адаптивної контейнеризації

Функціонування методу адаптивної контейнеризації ґрунтується на реалізації замкненого керуючого циклу, що відповідає концепції adaptive control loop. Керуючий цикл формалізується як процес безперервного

прийняття рішень щодо параметрів середовища виконання з метою забезпечення переносимості програмної системи при зміні обчислювальної платформи.

Вхідними даними керуючого циклу є потоки метрик виконання $m(t)$, профіль цільової обчислювальної платформи P та історичні дані H, що відображають попередні стани системи, прийняті керуючі рішення та їх наслідки. Стан системи у момент часу t описується вектором-функцією $x(t)$, який формується на основі поточних метрик виконання, параметрів контейнерного середовища та характеристик платформи виконання.

Ціль керування формулюється як мінімізація відхилення переносимості $D(t)$, що визначає ступінь відхилення поведінки програмної системи від еталонного виконання або допустимого експлуатаційного інтервалу. Таким чином, забезпечення переносимості розглядається як первинна ціль керуючого циклу, тоді як стабільність та ефективність використання ресурсів виступають вторинними цілями, які реалізуються за умови виконання обмежень переносимості.

Керуючі впливи середовища виконання описуються вектор-функцією $u(t)$, який включає зміну обмежень та запитів ресурсів контейнерів, кількість реплік, пріоритети виконання та параметри розміщення. Формування керуючих впливів здійснюється на основі політики керування π, яка відображає відповідність між поточним станом системи, профілем платформи та допустимими керуючими діями.

Динаміка системи у керуючому циклі може бути представлена у вигляді функції:

$$x(t+\Delta) = g\big(x(t), u(t), P, w(t)\big), \tag{2.18}$$

На відміну від базового рівняння (2.16), де керуючий вплив залежить виключно від поточного стану системи та профілю платформи, у повному

керуючому циклі політика керування розширюється для забезпечення здатності до самонавчання. Додавання масиву історичних даних H відображає те, що сама логіка прийняття рішень не є статичною, а безперервно еволюціонує та коригується на основі накопиченого досвіду попередніх станів і застосованих дій:

$$u(t) = \pi\big(x(t), P, H\big), \tag{2.19}$$

Машинне навчання у цьому рівнянні визначає алгоритмічну основу самої політики керування π, яка формується та безперервно коригується спираючись на масив історичних даних H. Зазначений вираз формалізує здатність інтелектуального модуля до самонавчання: накопичені дані про попередні стани середовища, застосовані дії конфігурування та їхні фактичні наслідки слугують безперервною навчальною вибіркою для виявлення прихованих закономірностей, що перетворює політику керування зі статичного набору правил на динамічну адаптивну підсистему. [58]

Усі керуючі дії підлягають перевірці на відповідність обмеженням середовища виконання, зокрема допустимим діапазонам ресурсних параметрів, політикам безпеки, вимогам доступності та цілісності сервісів. [21] Такий механізм забезпечує безпечність та узгодженість застосування керуючих впливів.

Множина допустимих керуючих впливів визначається як $u(t) \in U\big(x(t), P\big)$, де U формується з урахуванням політик безпеки, ресурсних обмежень кластера та обмежень системи оркестрації. До таких обмежень належать мінімально та максимально допустимі значення ресурсних параметрів контейнерів, допустимі діапазони кількості реплік, вимоги до доступності сервісів, а також політики disruption budget.

Слід зазначити, що частина керуючих впливів має дискретний характер та може застосовуватися із затримкою або з перезапуском контейнера,

зокрема при зміні певних ресурсних параметрів або конфігурацій середовища виконання. Ці особливості враховуються у часових параметрах керуючого циклу та у загальній функції переходу стану $g()$, що забезпечує коректність моделі керування у реальних умовах експлуатації контейнерних платформ.

Зворотний зв'язок у керуючому циклі реалізується шляхом спостереження змін вектора-функції метрик $m(t+\Delta)$ після застосування керуючих дій та оновлення історичних даних H. Отримана інформація використовується для коригування політики керування та уточнення моделей аналізу, що забезпечує адаптивність і самонавчання методу у процесі експлуатації.

Часові параметри керуючого циклу визначаються періодом Δ прийняття рішень та вікном агрегації метрик, що дозволяє узгодити точність адаптації з вимогами до швидкодії та обмеженнями обчислювальних ресурсів.

2.5.5 Архітектура збору метрик та телеметрії

Компонент збору метрик є критично важливою частиною архітектури методу, оскільки якість адаптивних рішень безпосередньо залежить від повноти та точності телеметричних даних. Архітектура збору метрик орієнтована на використання існуючих стандартів спостережуваності у контейнерних середовищах.

Збір метрик реалізується на кількох рівнях. На рівні контейнерів отримуються дані про використання процесорного часу, пам'яті та мережевих ресурсів через механізми cgroups та container runtime. На рівні вузлів кластера збирається інформація про загальне навантаження, доступні ресурси та стан апаратної платформи.

Для уніфікації та агрегації телеметрії доцільним є використання систем моніторингу типу Prometheus, які забезпечують pull-based модель збору

метрик та підтримують часові ряди. Зібрані метрики можуть додатково оброблятися за допомогою систем візуалізації та аналізу, таких як Grafana, або передаватися безпосередньо до інтелектуального модуля.

Архітектура збору телеметрії проєктується таким чином, щоб мінімізувати накладні витрати та не впливати на продуктивність прикладних контейнерів. Це забезпечує можливість використання методу у середовищах з обмеженими ресурсами та у системах реального часу.

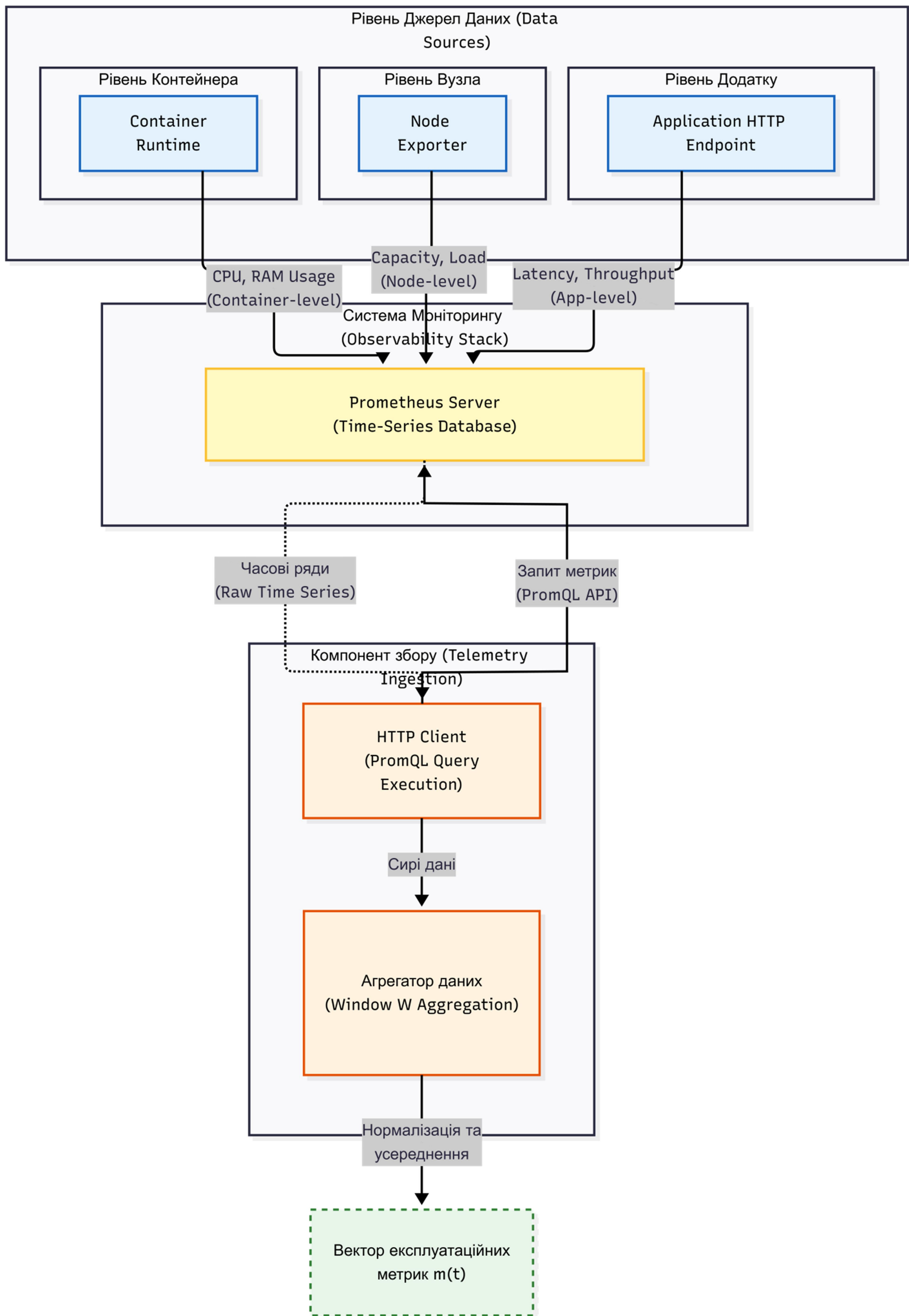
Рівень Джерел Даних (Data Sources)
Рівень Контейнера
Container Runtime
Рівень Вузла
Node Exporter
Рівень Додатку
Application HTTP Endpoint
CPU, RAM Usage (Container-level)
Capacity, Load (Node-level)
Latency, Throughput (App-level)
Система Моніторингу (Observability Stack)
Prometheus Server (Time-Series Database)
Часові ряди (Raw Time Series)
Запит метрик (PromQL API)
Компонент збору (Telemetry Ingestion)
HTTP Client (PromQL Query Execution)
Сирі дані
Агрегатор даних (Window W Aggregation)
Нормалізація та усереднення
Вектор експлуатаційних метрик m(t)

Рисунок 2.4. Архітектура збору телеметрії та формування вектора-функції експлуатаційних метрик $m(t)$.

2.5.6 Архітектура інтелектуального модуля прийняття рішень

Інтелектуальний модуль прийняття рішень реалізує логіку аналізу стану системи та формування керуючих впливів. Архітектурно він складається з підсистеми формування ознак, підсистеми машинного навчання та підсистеми логіки прийняття рішень.

Підсистема формування ознак відповідає за перетворення сирих метрик у структурований feature vector, придатний для використання моделями машинного навчання. Ознаки можуть включати як миттєві значення метрик, так і агреговані характеристики, такі як середні значення, дисперсія або тренди у часових вікнах.

Підсистема машинного навчання може використовувати регресійні моделі, дерева рішень або нейронні мережі для прогнозування навантаження та оцінювання впливу зміни параметрів контейнерів. Важливою особливістю архітектури є можливість оновлення моделей у процесі експлуатації на основі накопичених історичних даних.

Логіка прийняття рішень відповідає за інтерпретацію результатів inference та трансляцію їх у конкретні керуючі дії. Цей рівень також враховує обмеження оркестратора, політики безпеки та допустимі діапазони параметрів.

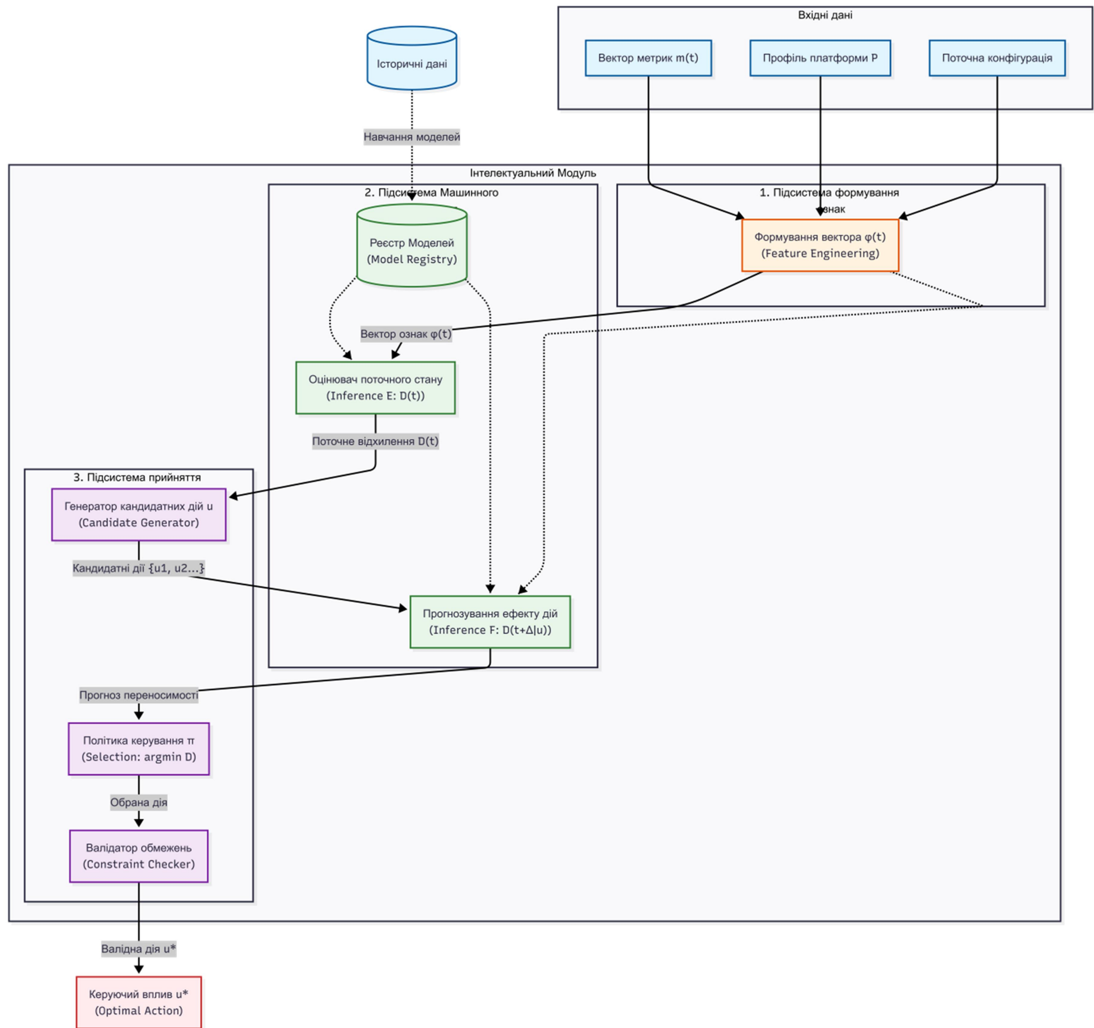

Рисунок 2.5. Архітектура інтелектуального модуля: формування вектора-функції ознак $\varphi(t)$ оцінювання стану та вибір оптимального керуючого впливу.

2.5.7 Архітектура застосування керуючих впливів

Застосування керуючих впливів у методі здійснюється безпосередньо через стандартні механізми управління контейнерними середовищами. Це дозволяє інтегрувати метод без модифікації прикладного коду та без порушення сумісності з існуючими платформами.

До основних механізмів керування належать вертикальне масштабування контейнерів шляхом зміни resource limits, горизонтальне масштабування через зміну кількості реплік, а також керування пріоритетами та політиками розміщення контейнерів. У середовищах Kubernetes такі дії реалізуються через API сервера управління та відповідні controller components.

Важливою архітектурною вимогою є атомарність та узгодженість застосування змін. Керуючі дії повинні виконуватися таким чином, щоб не призводити до деградації сервісу або порушення його доступності. Для цього можуть використовуватися механізми rolling updates та контроль стану readiness і liveness контейнерів.

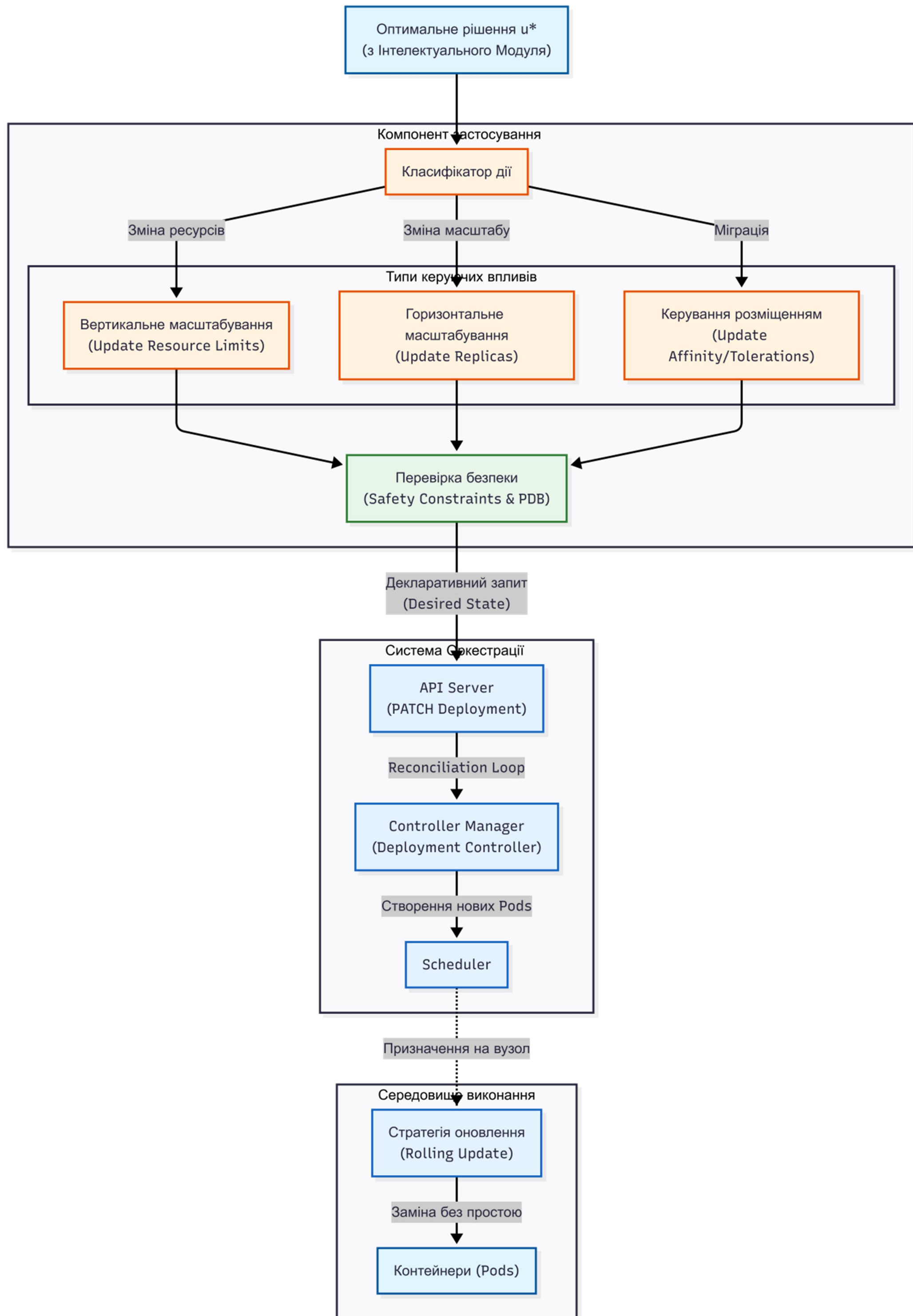

Рисунок 2.6. Схема застосування керуючих впливів через Kubernetes control plane з перевіркою обмежень безпеки та доступності.

2.6 Висновки до другого розділу

У другому розділі дисертаційної роботи розроблено та науково обґрунтовано метод адаптивної контейнеризації з інтеграцією штучного інтелекту, спрямований на автоматизоване забезпечення переносимості програм та програмних систем у гетерогенних обчислювальних середовищах. Запропонований метод розвиває результати аналітичного огляду, виконаного у попередньому розділі та спрямований на подолання обмежень традиційних статичних підходів до контейнеризації. Переносимість програмного забезпечення розглянуто як динамічну властивість, що формується у процесі взаємодії програмного коду, середовища виконання та апаратної платформи, що дозволяє перейти від простої уніфікації середовищ до інтелектуального управління їх параметрами з урахуванням умов експлуатації.

Розроблено архітектуру методу адаптивної контейнеризації, яка забезпечує замкнений цикл управління середовищами виконання програм. Архітектура базується на модульному принципі та включає компоненти збору метрик, попередньої обробки даних, інтелектуального аналізу, управління адаптацією, зберігання історичних даних та інтеграції з системою оркестрації. Це забезпечує масштабованість, розширюваність і сумісність методу з наявними контейнерними інфраструктурами. Використання механізму зворотного зв'язку та накопичення історичних даних створює умови для самонавчання методу і поступового підвищення ефективності прийнятих рішень у процесі експлуатації.

У розділі проаналізовано алгоритмічні аспекти реалізації методу, зокрема підходи до обробки потоків метрик, прогнозування навантаження та трансляції рішень у керуючі дії. [90] Встановлено, що алгоритмічна складова забезпечує практичну реалізовність адаптації у режимі, близькому до реального часу, без порушення стабільності контейнерної інфраструктури. Метод змінює класичне уявлення про контейнер як статичний об'єкт, параметри якого задаються на етапі розгортання, і розглядає його як

керований елемент, поведінка якого залежить від контексту виконання, стану системи та характеристик цільової платформи.

Теоретичне значення методу полягає у розвитку концепції переносимості як динамічної, контекстно-залежної властивості програмної системи. Метод поєднує положення теорії адаптивних систем, методів машинного навчання та принципів програмної інженерії, формуючи модель управління середовищами виконання з елементами самонавчання. Отримані результати розширюють теоретичні засади дослідження гетерогенних обчислювальних середовищ і можуть бути використані для подальшого розвитку самокерованих та саморегульованих програмних систем.

Практична значущість результатів полягає у можливості застосування методу в хмарних, мультихмарних, гібридних, периферійних, фінансових, телекомунікаційних, промислових та інших інформаційних системах, де актуальними є переносимість і ефективне використання ресурсів. Адаптивна контейнеризація дозволяє забезпечити стабільну роботу програм у середовищах з різними апаратними характеристиками без ручного налаштування конфігурацій, зменшує витрати на експлуатацію, підвищує надійність і спрощує масштабування програмних систем у виробничих умовах.

Для процесів розробки та супроводу програмного забезпечення метод має важливе значення завдяки зменшенню залежності від ручного налаштування середовищ виконання та зниженню складності експлуатації багатоплатформних програмних систем. [52;97] Адаптивна контейнеризація сприяє підвищенню відтворюваності середовищ, зменшенню кількості помилок конфігурації та кращій інтеграції етапів розробки й експлуатації програмного забезпечення, що відповідає принципам сучасних DevOps-орієнтованих процесів. Автоматизована адаптація середовищ виконання дозволяє скоротити час реагування на зміни умов експлуатації та підвищити якість програмних продуктів. [62]

Метод також відкриває можливості для подальших досліджень у напрямку інтелектуального управління обчислювальними ресурсами. Перспективними напрямами його розвитку є розширення набору враховуваних метрик, зокрема показників енергоспоживання, мережевої затримки, пропускної здатності сховищ даних та надійності, що дозволить підвищити точність адаптації у гетерогенних і периферійних середовищах. [86] Подальше удосконалення може бути пов’язане з використанням ансамблевих і гібридних моделей машинного навчання, інтеграцією з механізмами автоматизованого планування у мультихмарних середовищах, узгодженням з оркестраційними політиками та механізмами безпеки, а також підвищенням пояснюваності прийнятих рішень.

Для порівняння з існуючими рішеннями як базові підходи розглянуто стандартні сценарії використання контейнерних технологій без інтелектуальних модулів адаптації, зокрема Docker з ручним налаштуванням ресурсних обмежень та Kubernetes із типовими механізмами autoscaling, що ґрунтуються на окремих метриках використання ресурсів без урахування профілю платформи виконання та без явної цільової функції переносимості. [72] На відміну від таких підходів, запропонований метод враховує платформно-індуковані відмінності безпосередньо у контурі прийняття рішень і розглядає переносимість як первинну цільову властивість керуючого циклу.

Переваги методу над класичними методами контейнеризації полягають у переході від статичних конфігурацій до динамічних, контекстно-залежних параметрів середовища виконання, використанні алгоритмів машинного навчання, автоматизованому врахуванні характеристик цільової платформи та інтеграції замкненого циклу моніторингу, аналізу, прийняття рішень і коригування параметрів контейнерів. При цьому метод залишається сумісним з наявними контейнерними платформами та системами оркестрації, оскільки не потребує модифікації базових механізмів Docker або Kubernetes і може використовувати стандартні інтерфейси управління ресурсами.

Таким чином, результати другого розділу підтверджують доцільність і наукову обґрунтованість методу адаптивної контейнеризації з інтеграцією штучного інтелекту. Запропонований підхід має теоретичне, практичне та інженерне значення, розширює можливості контейнеризації у гетерогенних обчислювальних середовищах і створює основу для експериментальної перевірки ефективності методу та практичної оцінки його переваг у наступному розділі дисертаційної роботи.

РОЗДІЛ 3. ПРАКТИЧНА РЕАЛІЗАЦІЯ ТА ВПРОВАДЖЕННЯ МЕТОДУ

3.1 Реалізація архітектури програмного прототипу

Програмний прототип запропонованого методу адаптивної контейнеризації реалізовано як distributed control system, інтегровану з Kubernetes control plane та побудовану за принципами controller-based architecture. [27;108] Архітектура прототипу спроєктована таким чином, щоб кожен елемент формалізованої моделі, мав безпосереднє відображення у вигляді програмного компонента, структури даних або етапу виконання керуючого циклу. [43]

Центральним елементом архітектури є adaptive portability controller, який реалізує замкнений керуючий цикл (closed-loop control) у вигляді періодично виконуваного reconciliation loop. Контролер розгортається як окремий containerized service у Kubernetes-кластері та має доступ до Kubernetes API Server, систем спостережуваності та метаданих вузлів кластера. Такий підхід дозволяє використовувати стандартні механізми orchestration як інтерфейс застосування керуючих впливів без втручання у прикладний код керованих програмних систем. [6]

Архітектурно програмний прототип складається з таких функціональних підсистем: telemetry ingestion layer, state construction layer, machine learning inference layer, decision-making layer та actuation layer. Кожна з цих підсистем відповідає окремому етапу формалізованого керуючого циклу та реалізується як логічний модуль у межах контролера.

Telemetry ingestion layer відповідає за взаємодію з observability stack та отримання сирих часових рядів метрик виконання. На цьому рівні здійснюється інтеграція з Prometheus та Kubernetes Metrics API, що забезпечує доступ до container-level та node-level metrics. Результатом роботи

цього шару є набір сирих метрик, які надалі використовуються для формування вектора-функції метрик виконання $m(t)$.

State construction layer реалізує формування вектора-функції стану системи $x(t)$. У реалізації прототипу $x(t)$ формується як агрегована структура, що включає вектор-функцію метрик виконання $m(t)$, поточну конфігурацію контейнерного середовища (resource limits, requests, replica count) та профіль цільової платформи виконання P. Цей шар виконує роль state estimator у класичній термінології теорії керування.

Machine learning inference layer реалізує функції оцінювання та прогнозування відхилення переносимості, визначені у формалізації методу. На цьому рівні відбувається inference моделей машинного навчання, які оцінюють поточне значення $D(t)$ та прогнозують очікуване значення $D(t+\Delta)$ для кандидатних керуючих впливів. Реалізація даного шару допускає використання різних типів моделей, зокрема regression models, tree-based models або neural networks, що дозволяє адаптувати прототип до різних класів програмних систем.

Decision-making layer реалізує політику керування π, яка здійснює вибір керуючого впливу u^* з множини допустимих дій $U(x(t),P)$. На цьому рівні відбувається перевірка обмежень системи оркестрації, зокрема допустимих значень resource limits, мінімальної та максимальної кількості реплік, а також політик availability та disruption budget. Таким чином, decision-making layer забезпечує узгодження інтелектуальних рішень з реальними обмеженнями середовища виконання. [9]

Actuation layer відповідає за застосування обраного керуючого впливу до контейнерного середовища виконання. У програмному прототипі цей шар реалізує взаємодію з Kubernetes API Server та виконує операції patch або update над ресурсами Deployment та Pod. Саме на цьому рівні керуючі рішення трансформуються у конкретні дії оркестрації.

З технічної точки зору, контролер реалізується як Python-сервіс, що використовує офіційну бібліотеку Kubernetes client library для доступу до API оркестратора. Фрагмент коду ініціалізації контролера та підключення до Kubernetes control plane [64] наведено в додатку В: “ЛІСТИНГ ІНІЦІАЛІЗАЦІЇ КЛІЄНТА KUBERNETES”.

Такий підхід безпосередньо відповідає формалізованому керуючому циклу, та забезпечує безперервну адаптацію середовища виконання у відповідь на зміну стану системи та характеристик платформи.

Важливо підкреслити, що архітектура програмного прототипу свідомо не використовує жорстко закодованих евристик autoscaling або порогових правил, характерних для стандартних механізмів оркестрації. Усі керуючі рішення приймаються на основі оцінювання функції відхилення переносимості $D(t)$, що дозволяє трактувати переносимість як первинну ціль керування, а не як побічний ефект оптимізації продуктивності або використання ресурсів.

3.2. Реалізація збору метрик та формування вектора-функції m(t)

У програмному прототипі запропонованого методу реалізація збору експлуатаційних метрик та формування вектора-функції $m(t)$ розглядається як окремий, чітко визначений етап керуючого циклу, що відповідає фазі спостереження (observation phase) у термінах теорії керування та distributed systems. Даний етап забезпечує кількісне представлення поточного стану виконання програмної системи та створює основу для подальшого аналізу переносимості та прийняття керуючих рішень.

Збір метрик у програмному прототипі реалізовано як monitoring pipeline, побудований на базі стандартного cloud-native observability stack. [5] У якості системи збору та зберігання часових рядів використовується Prometheus, який виконує функції metric scraping engine та time-series

database. Такий вибір зумовлений широкою підтримкою Prometheus у контейнерних середовищах, наявністю стандартизованого формату експорту метрик та можливістю виконання агрегацій і трансформацій даних за допомогою мови запитів PromQL.

У реалізації прототипу використовуються три основні класи експлуатаційних метрик: container-level metrics, node-level metrics та application-level metrics. Container-level metrics включають показники використання процесорного часу, оперативної пам’яті та I/O, які експортуються container runtime через cgroups та kubelet. Node-level metrics відображають загальний стан вузлів кластера, зокрема доступні ресурси та рівень contention. Application-level metrics характеризують поведінку прикладного коду та експортуються у вигляді HTTP-метрик, таких як request latency та throughput. [107;40]

Отримання метрик з Prometheus у програмному прототипі здійснюється через Prometheus HTTP API з використанням PromQL-запитів. Фрагмент коду отримання метрик наведено в додатку Г: “ЛІСТИНГ ОТРИМАННЯ МЕТРИК З PROMETHEUS”.

Зібрані сирі значення метрик представляють собою часові ряди з різною частотою оновлення та різними масштабами значень. Для забезпечення узгодженості даних у програмному прототипі використовується ковзне часове вікно спостереження W, яке відповідає періоду агрегації метрик у межах однієї ітерації керуючого циклу. Усі значення метрик усереднюються або агрегуються у межах цього вікна, що дозволяє зменшити вплив короткочасних флуктуацій та шуму.

Після агрегації значення метрик формуються у числова вектор-функція $m(t)$, який у програмному прототипі реалізовано як numpy-масив фіксованої розмірності. Кожна компонента вектор-функції відповідає окремій експлуатаційній характеристиці, визначеній у формалізації методу. Фрагмент

коду формування вектора-функції $m(t)$ наведено в додатку Д: “ЛІСТИНГ ФОРМУВАННЯ ВЕКТОРА-ФУНКЦІЇ $m(t)$”.

Сформована вектор-функкця m(t) є лише частиною повного вектора-функції стану системи $x(t)$. У програмному прототипі він доповнюється інформацією про поточну конфігурацію контейнерного середовища, зокрема значеннями resource limits та кількістю реплік, що дозволяє відобразити не лише спостережувану поведінку програмної системи, але й умови її виконання у кластері.

Принципово важливим є те, що реалізація збору метрик у програмному прототипі не вимагає модифікації прикладного коду програмних систем. Використання стандартних container-level та HTTP-level метрик дозволяє застосовувати метод до широкого класу програмних систем незалежно від мови програмування, фреймворку або внутрішньої архітектури додатку.

Таким чином, реалізація збору метрик та формування вектора-функції $m(t)$ у програмному прототипі забезпечує кількісне, узгоджене та відтворюване представлення стану виконання програмної системи, необхідне для подальшого оцінювання відхилення переносимості та прийняття керуючих рішень у межах замкненого керуючого циклу.

3.3. Формування профілю платформи P та вектора-функції ознак φ(t)

У програмному прототипі запропонованого методу профіль цільової обчислювальної платформи P розглядається як повноцінний елемент керуючого циклу, що безпосередньо впливає на оцінювання та прогнозування відхилення переносимості програмної системи. На відміну від класичних підходів до оркестрації контейнерів, у яких характеристики платформи використовуються лише неявно на етапі планування, у даній реалізації вони явно інтегруються у процес прийняття керуючих рішень.

Формування профілю платформи P у програмному прототипі реалізується шляхом збору та агрегації platform-level metadata, отриманих з Kubernetes Node API. До складу цього профілю входять як статичні характеристики обчислювальних вузлів, так і квазідинамічні параметри, що можуть змінюватися у процесі експлуатації кластера. Такий підхід дозволяє відобразити реальну гетерогенність обчислювального середовища та врахувати її вплив на поведінку програмної системи.

Отримання інформації про вузли кластера у програмному прототипі здійснюється через CoreV1Api Kubernetes, який надає доступ до об'єктів типу Node. На основі цих об'єктів контролер отримує інформацію про процесорну архітектуру, доступні обчислювальні ресурси, операційну систему та додаткові атрибути, визначені у вигляді labels. Фрагмент коду що реалізує отримання метаданих вузлів кластера наведено в додатку Е: "ЛІСТИНГ ОТРИМАННЯ МЕТАДАНИХ ВУЗЛІВ КЛАСТЕРА".

Ключовим елементом профілю платформи є процесорна архітектура, яка визначається на основі стандартних Kubernetes labels, зокрема kubernetes.io/arch та kubernetes.io/os. У програмному прототипі ці значення використовуються для явного розрізнення платформ типу x86_64 та ARM64, що є критичним для аналізу переносимості.

Окрім архітектури процесора, до профілю платформи включаються кількісні характеристики доступних ресурсів, отримані з поля node.status.capacity. Ці параметри дозволяють оцінити максимальні можливості вузла та використовуються при формуванні допустимих керуючих впливів $U(x,P)$.

Для врахування квазідинамічних характеристик платформи у програмному прототипі також використовуються node-level metrics, зокрема показники поточного завантаження вузлів та рівень ресурсного contention. Ці дані отримуються з Prometheus та агрегуються аналогічно container-level

metrics, що дозволяє інтегрувати їх у загальний вектора-функції стану системи.

На основі зібраних характеристик платформи P та вектора-функції метрик виконання $m(t)$ формується вектор-функція ознак $\varphi(t)$, який використовується як безпосередній вхід для інтелектуального модуля. Процес формування $\varphi(t)$ у програмному прототипі реалізується як етап feature engineering, на якому різнорідні дані приводяться до єдиного числового представлення.

До складу вектора-функції $\varphi(t)$ входять нормалізовані значення метрик виконання, параметри поточної конфігурації контейнерного середовища та числові представлення характеристик платформи. Наприклад, процесорна архітектура кодується у вигляді категоріальної ознаки, перетвореної у числову форму за допомогою one-hot encoding або порядкової індексації.

Агрегована вектор-функція ознак $\varphi(t)$ формується шляхом конкатенації відповідних компонентів.

Таким чином, у програмному прототипі вектор-функції $\varphi(t)$ є компактним числовим представленням як поведінки програмної системи, так і умов її виконання на конкретній обчислювальній платформі. Саме ця вектор-функція використовується для інференсу моделей машинного навчання, що оцінюють та прогнозують відхилення переносимості.

Принципово важливо, що формування профілю платформи P та вектора-функції ознак $\varphi(t)$ у даній реалізації не обмежується фіксованим набором характеристик. Архітектура прототипу допускає розширення профілю платформи шляхом додавання нових ознак, зокрема інформації про NUMA topology, типи апаратних прискорювачів або спеціалізовані node labels, без зміни загальної логіки керуючого циклу.

Таким чином, реалізація формування профілю платформи P та вектора-функції ознак $\varphi(t)$ у програмному прототипі забезпечує явне та кількісне врахування платформної гетерогенності у процесі керування, що є

ключовою відмінністю запропонованого методу від класичних підходів до адаптації контейнеризованих середовищ виконання.

3.4. Реалізація інтелектуального модуля оцінювання та прогнозування відхилення переносимості D(t)

Інтелектуальний модуль програмного прототипу реалізує центральну частину запропонованого методу адаптивної контейнеризації та відповідає за кількісне оцінювання і прогнозування відхилення переносимості програмної системи. У термінах формалізації, даний модуль реалізує функції оцінювання $E\left(x(t),P\right)\to D(t)$ та прогнозування $F\left(x(t),P,u\right)\to D(t+\Delta)$, які використовуються як основа для прийняття керуючих рішень у замкненому керуючому циклі.

Архітектурно інтелектуальний модуль реалізується як окремий логічний компонент adaptive portability controller та виконує inference у режимі online decision making. Модуль працює з числовим вектором-функцій ознак φ(t), сформованим на попередніх етапах, та повертає скалярне значення відхилення переносимості, яке інтерпретується як сигнал помилки керування (control error).

З точки зору машинного навчання задача оцінювання $D\left(t\right)$ формулюється як задача керованого навчання із застосуванням складних нелінійних моделей, у якій модель навчається відображати вектора-функції ознак, що описує стан системи та платформу виконання, у числову міру відхилення від еталонної поведінки. У програмному прототипі передбачено розділення етапів training та inference, що дозволяє використовувати попередньо навчені моделі у процесі експлуатації без зупинки керуючого циклу.

Для навчання моделей використовується історичний набір даних, сформований у процесі виконання програмної системи на різних

обчислювальних платформах та при різних сценаріях навантаження. Кожен навчальний приклад складається з вектора-функції ознак $\varphi(t)$ та відповідного значення відхилення переносимості $D(t)$, обчисленого на основі формалізованої цільової функції. Таким чином, модель навчається відтворювати кількісну оцінку міжплатформного відхилення поведінки системи. [4]

У програмному прототипі реалізація інтелектуального модуля допускає використання різних класів регресійних моделей, зокрема linear regression, tree-based models (Random Forest, Gradient Boosting) або shallow neural networks. Така гнучкість дозволяє адаптувати модель до характеристик конкретної програмної системи та обсягу доступних навчальних даних. У базовій реалізації використовується tree-based regression model, що забезпечує добру здатність до узагальнення при обмеженій кількості навчальних прикладів та дозволяє інтерпретувати внесок окремих ознак.

Отримане значення $D(t)$ використовується як поточна оцінка відхилення переносимості та порівнюється з допустимими пороговими значеннями, визначеними параметрами ε, T та ρ. У випадку, якщо значення $D(t)$ перевищує допустимий рівень, інтелектуальний модуль ініціює етап прогнозування наслідків можливих керуючих впливів.

Прогнозування $D(t+\Delta)$ реалізується шляхом симуляції впливу кандидатних керуючих дій $u \in U(x(t),P)$ на вектор-функцію ознак $\varphi(t)$. Для кожного допустимого керуючого впливу формується модифікована вектор-функція ознак $\varphi_u(t)$, який відображає очікувану конфігурацію контейнерного середовища після застосування відповідної дії. Далі для кожного такого вектора-функції виконується інференс моделі з метою отримання прогнозованого значення відхилення переносимості.

Фрагмент коду що реалізує прогнозування $D(t+\Delta)$ для множини кандидатних керуючих впливів наведено в додатку Ж: “ЛІСТИНГ

ПРОГНОЗУВАННЯ $D(t+\Delta)$ ДЛЯ МНОЖИНИ КАНДИДАТНИХ КЕРУЮЧИХ ВПЛИВІВ".

У даному коді candidate actions представляє собою словник, у якому кожному ідентифікатору керуючого впливу відповідає прогнозований вектор-функція ознак після застосування цієї дії. Такий підхід дозволяє виконувати порівняльний аналіз альтернативних варіантів адаптації середовища виконання без їх фактичного застосування.

Політика керування π використовує результати прогнозування для вибору керуючого впливу u^*, що мінімізує прогнозоване відхилення переносимості $D(t+\Delta)$ за умови виконання всіх обмежень системи оркестрації. Таким чином, інтелектуальний модуль не просто реагує на поточний стан системи, а виконує проактивне керування, орієнтоване на збереження інваріантності поведінки програмної системи у часі.

Важливо зазначити, що у програмному прототипі інтелектуальний модуль не використовується для прямої оптимізації окремих показників продуктивності, таких як CPU utilization або throughput. Навпаки, усі ці показники враховуються лише опосередковано через їх внесок у загальну міру відхилення переносимості. Це принципово відрізняє запропонований підхід від класичних систем autoscaling, у яких відсутня явна модель міжплатформного відхилення.

Таким чином, реалізація інтелектуального модуля оцінювання та прогнозування відхилення переносимості $D(t)$ забезпечує ключову функціональність запропонованого методу та дозволяє інтегрувати машинне навчання у замкнений керуючий цикл керування контейнеризованим середовищем виконання. Наступний підрозділ присвячений реалізації політики керування π та механізму вибору керуючих впливів з урахуванням обмежень системи оркестрації.

3.5. Реалізація політики керування π та вибору керуючих впливів u(t)

Політика керування π у програмному прототипі реалізує механізм прийняття рішень щодо адаптації контейнерного середовища виконання на основі оцінювання та прогнозування відхилення переносимості. У термінах формалізації, політика керування реалізує відображення $\pi\left(x(t), P, H\right) \rightarrow u(t)$, де $x(t)$ є поточним станом системи, P — профіль цільової платформи виконання, а H — історичні дані виконання та прийнятих рішень.

Архітектурно політика керування реалізується як decision-making layer у складі adaptive portability controller та виконує функції constraint-aware optimization. На відміну від класичних autoscaling-механізмів, які застосовують локальні евристики або порогові правила, політика π у програмному прототипі здійснює вибір керуючого впливу на основі мінімізації цільової функції переносимості з урахуванням множини допустимих дій.

Множина допустимих керуючих впливів $U\left(x(t), P\right)$ формується на основі поточного стану системи та обмежень оркестрації. До таких обмежень належать допустимі діапазони значень resource limits та requests, мінімальна та максимальна кількість реплік, політики PodDisruptionBudget, а також вимоги до доступності сервісу. Формування множини U виконується динамічно у кожній ітерації керуючого циклу, що дозволяє враховувати актуальні умови виконання.

У програмному прототипі множина допустимих керуючих впливів представлена у вигляді набору кандидатних дій, кожна з яких описує зміну параметрів контейнерного середовища. Типові керуючі впливи включають вертикальну адаптацію (vertical scaling) шляхом зміни значень CPU та memory limits, а також горизонтальну адаптацію (horizontal scaling) шляхом зміни кількості реплік Deployment.

Фрагмент коду що формує множину кандидатних керуючих впливів з урахуванням обмежень системи оркестрації наведено в додатку З: “ЛІСТИНГ ФОРМУВАННЯ МНОЖИНИ КАНДИДАТНИХ КЕРУЮЧИХ ВПЛИВІВ З УРАХУВАННЯМ ОБМЕЖЕНЬ СИСТЕМИ ОРКЕСТРАЦІЇ”.

Для кожного кандидатного керуючого впливу u формується прогнозована вектор-функція ознак $\varphi_u(t)$, який відображає очікувану конфігурацію контейнерного середовища після застосування цієї дії. Далі інтелектуальний модуль виконує прогнозування значення відхилення переносимості $D(t+\Delta)$ для кожного варіанта керування.

Політика керування π здійснює вибір оптимального керуючого впливу шляхом розв’язання задачі мінімізації прогнозованого відхилення переносимості з урахуванням жорстких та м’яких обмежень. У програмному прототипі базова реалізація політики π використовує жадібний (greedy) алгоритм вибору, який обирає керуючу дію з мінімальним значенням прогнозованого $D(t+\Delta)$.

Для запобігання частим коливанням конфігурації (configuration thrashing) у політиці керування також реалізовано механізм hysteresis та cooldown period. Це означає, що новий керуючий вплив застосовується лише у випадку, якщо прогнозоване зменшення відхилення переносимості перевищує заданий поріг, а з моменту останньої адаптації минув мінімальний часовий інтервал.

У програмному прототипі також передбачено пріоритетність збереження переносимості над оптимізацією використання ресурсів. Якщо декілька керуючих впливів забезпечують однакове або близьке значення прогнозованого $D(t+\Delta)$, політика керування віддає перевагу діям, що потребують мінімальних змін конфігурації або не призводять до перезапуску контейнерів. Це дозволяє зменшити операційні накладні витрати та уникнути деградації доступності сервісу.

Застосування обраного керуючого впливу u^* виконується через actuation layer, який взаємодіє з Kubernetes API. Політика керування не виконує безпосередніх змін конфігурації, а передає опис обраної дії відповідному модулю застосування, що забезпечує розділення логіки прийняття рішень та механізмів їх реалізації.

Таким чином, реалізація політики керування π у програмному прототипі забезпечує формальну відповідність між теоретичною постановкою задачі мінімізації відхилення переносимості та практичним механізмом прийняття керуючих рішень у оркестрованому контейнерному середовищі. Наступний підрозділ присвячений реалізації механізмів застосування керуючих впливів та інтеграції з Kubernetes control plane.

3.6. Реалізація множини допустимих дій U(x,P) та інтеграція з Kubernetes control plane

Реалізація множини допустимих керуючих впливів $U(x(t),P)$ у програмному прототипі безпосередньо пов'язана з механізмами керування станом об'єктів у Kubernetes control plane. На відміну від імперативного керування, у якому зміни застосовуються безпосередньо до контейнерів, Kubernetes використовує декларативну модель керування, у межах якої бажаний стан системи описується у вигляді ресурсів, а система оркестрації забезпечує його досягнення та підтримку.

У програмному прототипі множина $U(x(t),P)$ формується як підмножина допустимих змін декларативного стану Kubernetes-ресурсів, зокрема об'єктів типу Deployment. Кожен керуючий вплив $u \in U$ описується як атомарна зміна параметрів Deployment, яка задовольняє обмеження системи оркестрації, політики доступності та вимоги безпеки.

З технічної точки зору, керуючі впливи у прототипі класифікуються на дві основні категорії: вертикальні (vertical scaling actions) та горизонтальні

(horizontal scaling actions). Вертикальні дії змінюють параметри resource limits і requests контейнерів, тоді як горизонтальні дії змінюють кількість реплік Deployment. Обидва типи дій реалізуються шляхом оновлення декларативного опису ресурсу через Kubernetes API.

Формування конкретного керуючого впливу u^* супроводжується перевіркою допустимості дії з точки зору обмежень $U(x(t), P)$. До таких обмежень належать допустимі діапазони значень CPU та memory limits, мінімальна та максимальна кількість реплік, а також обмеження, накладені PodDisruptionBudget та політиками доступності. Ці перевірки виконуються до застосування змін, що дозволяє уникнути некоректних або небезпечних конфігурацій.

Інтеграція з Kubernetes control plane у програмному прототипі реалізується з використанням офіційної Kubernetes Python client library. Контролер взаємодіє з API Server для читання поточного стану ресурсів та застосування змін у вигляді patch або update операцій.

Фрагмент коду, що реалізує застосування вертикального керуючого впливу шляхом оновлення resource limits контейнера у Deployment наведено в додатку И: “ЛІСТИНГ РЕАЛІЗАЦІЇ ЗАСТОСУВАННЯ ВЕРТИКАЛЬНОГО КЕРУЮЧОГО ВПЛИВУ ШЛЯХОМ ОНОВЛЕННЯ RESOURCE LIMITS КОНТЕЙНЕРА У DEPLOYMENT”.

Горизонтальна адаптація реалізується шляхом оновлення поля spec.replicas у Deployment, що ініціює відповідний reconciliation process у Kubernetes control plane.

Застосування керуючих впливів у прототипі здійснюється з урахуванням семантики rolling update, яка є стандартним механізмом Kubernetes для оновлення контейнеризованих застосувань без повного простою сервісу. Контролер не керує безпосередньо перезапуском контейнерів, а покладається на Kubernetes, який забезпечує поступове оновлення Pod-ів відповідно до заданої стратегії розгортання.

Важливою особливістю інтеграції з Kubernetes control plane є те, що програмний прототип не порушує принципу declarative desired state. Контролер змінює лише бажаний стан ресурсів, після чого Kubernetes самостійно виконує reconciliation та приводить фактичний стан у відповідність до оновленого опису. Такий підхід забезпечує стійкість системи до збоїв та дозволяє уникнути конфліктів з іншими компонентами control plane.

У контексті формалізації методу, застосування керуючого впливу u^* відповідає переходу стану системи $x(t) \to x(t+\Delta)$, де Δ визначається як період між ітераціями керуючого циклу. Фактична затримка між застосуванням змін та досягненням нового стабільного стану визначається механізмами оркестрації та враховується у часових параметрах керуючого циклу.

Таким чином, реалізація множини допустимих дій $U(x(t), P)$ та інтеграція з Kubernetes control plane у програмному прототипі забезпечують коректне, безпечне та відтворюване застосування керуючих впливів у реальному оркестрованому середовищі. Це дозволяє безпосередньо пов’язати теоретичну модель керування переносимістю з її практичною реалізацією у сучасних container orchestration.

3.7. Реалізація замкненого керуючого циклу та часові аспекти керування

Замкнений керуючий цикл є центральним елементом практичної реалізації запропонованого методу адаптивної контейнеризації та забезпечує безперервне керування переносимістю програмної системи у часі. У програмному прототипі керуючий цикл реалізується як періодично виконуваний control loop, що інтегрує всі раніше описані компоненти: збір метрик, формування стану системи, оцінювання та прогнозування відхилення

переносимості, вибір керуючих впливів і їх застосування через систему оркестрації.

З точки зору часової організації, керуючий цикл у програмному прототипі характеризується двома ключовими часовими параметрами: періодом ітерації керування Δ та шириною ковзного вікна спостереження W. Параметр Δ визначає частоту прийняття керуючих рішень, тоді як параметр W визначає інтервал часу, протягом якого агрегуються експлуатаційні метрики для формування вектора-функції $m(t)$. Вибір цих параметрів здійснюється з урахуванням компромісу між швидкістю реакції системи та стійкістю керування.

У програмному прототипі керуючий цикл реалізується як окремий execution loop у складі adaptive portability controller. Кожна ітерація циклу відповідає переходу системи зі стану $x(t)$ у стан $x(t+\Delta)$ та включає послідовне виконання всіх етапів керування.

Фрагмент коду, що реалізує керуючого циклу наведено в додатку І: “ЛІСТИНГ РЕАЛІЗАЦІЇ КЕРУЮЧОГО ЦИКЛУ”.

У наведеній реалізації керуючий цикл працює у режимі near real-time control, що означає прийняття рішень з дискретним кроком часу, але з урахуванням затримок, притаманних оркестрованим контейнерним середовищам. Такий режим є типовим для distributed systems та не передбачає жорстких вимог до детермінованості часу виконання, що робить його сумісним з Kubernetes control plane. [25]

Важливим аспектом реалізації керуючого циклу є забезпечення стабільності керування (control stability) та запобігання коливанням конфігурації середовища виконання. [82] У програмному прототипі для цього використовуються механізми hysteresis та cooldown period, які обмежують частоту застосування керуючих впливів навіть у випадку незначних флуктуацій значення $D(t)$. Це дозволяє уникнути oscillatory behavior, характерного для наївних autoscaling-рішень. [7]

Крім того, реалізація керуючого циклу враховує асинхронну природу Kubernetes reconciliation process. Фактичний стан системи після застосування керуючого впливу може досягатися із затримкою, зумовленою rolling update, перезапуском контейнерів або міграцією Pod-ів між вузлами. У програмному прототипі ці затримки враховуються шляхом аналізу метрик виконання у наступних ітераціях керуючого циклу, що дозволяє коректно оцінювати вплив застосованих дій.

З формальної точки зору, керуючий цикл у програмному прототипі реалізує дискретну динамічну систему, у якій перехід стану описується функцією $x(t+\Delta) = g\big(x(t), u(t), P, w(t)\big)$, де $w(t)$ представляє зовнішні збурення, зумовлені змінами навантаження або характеристик платформи. Реалізація контролера не намагається усунути всі збурення, а прагне мінімізувати їх вплив на відхилення переносимості у довгостроковій перспективі.

Принципово важливим є те, що керуючий цикл у програмному прототипі орієнтований на підтримку інваріантності поведінки програмної системи при зміні обчислювальної платформи, а не на максимізацію короткострокових показників продуктивності. Усі часові параметри керування та умови застосування керуючих впливів налаштовані таким чином, щоб забезпечити стабільну еволюцію системи без різких змін конфігурації.

Таким чином, реалізація замкненого керуючого циклу у програмному прототипі забезпечує практичне втілення теоретичної моделі керування переносимістю, та створює основу для подальшої експериментальної оцінки властивостей методу у різних сценаріях виконання.

3.8. Експериментальне середовище та сценарії досліджень

Експериментальна перевірка запропонованого методу адаптивної контейнеризації виконувалася у контрольованому контейнеризованому середовищі, що дозволяє відтворювати різні конфігурації обчислювальних платформ та сценарії навантаження. Основною метою експериментів є кількісна оцінка здатності програмного прототипу забезпечувати переносимість програмної системи шляхом мінімізації відхилення переносимості $D(t)$ при зміні платформи виконання та умов експлуатації.

Експериментальне середовище побудоване на базі Kubernetes-кластера, який включає вузли з різними апаратними та програмними характеристиками. Така конфігурація дозволяє моделювати гетерогенне обчислювальне середовище, характерне для сучасних хмарних, мультихмарних та edge-орієнтованих систем. Кластер розгортається з використанням стандартних інструментів Kubernetes та підтримує виконання контейнеризованих навантажень без модифікації прикладного коду.

У межах експериментів визначено декілька профілів обчислювальних платформ, які використовуються для аналізу переносимості. До складу platform profiles входять платформи з різною процесорною архітектурою, зокрема вузли з архітектурою x86_64 та ARM64, а також вузли з різною конфігурацією доступних ресурсів. Профіль платформи P формується на основі метаданих Kubernetes Node API та включає архітектуру процесора, обсяг доступної оперативної пам'яті, кількість ядер CPU, а також додаткові атрибути середовища виконання.

Для експериментальної оцінки використовується одна і та сама програмна система, контейнеризована у вигляді Docker-образу та розгорнута у кластері без змін прикладного коду. Це дозволяє виключити вплив змін у реалізації програмної логіки та зосередитися виключно на впливі платформи виконання та параметрів контейнерного середовища. Контейнеризований додаток розгортається на різних платформах з ідентичним базовим описом

Deployment, після чого до нього застосовується або запропонований метод адаптивної контейнеризації, або базовий підхід, використаний як baseline.

У якості сценаріїв навантаження використовуються декілька workload profiles, що відрізняються характером та інтенсивністю обчислень. До таких сценаріїв належать рівномірне навантаження з постійною інтенсивністю запитів, навантаження з різкими сплесками (burst workload), а також сценарії з поступовою зміною інтенсивності запитів у часі. Навантаження генерується зовнішнім генератором запитів та застосовується до програмної системи у контрольований спосіб, що забезпечує відтворюваність експериментів.

Усі експерименти виконуються у двох режимах: із застосуванням запропонованого методу адаптивної контейнеризації та із застосуванням базового підходу. У якості baseline використовується стандартна конфігурація Kubernetes без інтелектуального контролера, з типовими механізмами autoscaling або зі статично заданими resource limits. У базовому режимі відсутня явна цільова функція переносимості, а керування середовищем виконання здійснюється або вручну, або на основі окремих метрик використання ресурсів.

Оцінювання переносимості програмної системи у ході експериментів здійснюється на основі формалізованої функції відхилення $D(t)$. Для кожного експериментального запуску вимірюються часові ряди експлуатаційних метрик, на основі яких обчислюється $D(t)$ у кожній ітерації керуючого циклу. Переносимість вважається забезпеченою, якщо значення $D(t)$ не перевищує заданого порогового рівня ε протягом контрольного інтервалу часу або якщо частка часу, протягом якої $D(t)$ перевищує ε, не перевищує допустиме значення ρ.

Параметри ε, T та ρ визначаються до початку експериментів та використовуються як фіксовані критерії порівняння різних сценаріїв керування. Це дозволяє виконувати кількісне порівняння ефективності

запропонованого методу та базового підходу незалежно від конкретної платформи виконання або сценарію навантаження. Усі значення метрик та проміжні результати експериментів зберігаються у системі моніторингу та використовуються для подальшого аналізу.

Особливу увагу у ході експериментів приділено аналізу динаміки $D(t)$ у часі, а не лише середнім або піковим значенням окремих метрик. Такий підхід дозволяє оцінити не лише факт досягнення переносимості, але й стабільність поведінки програмної системи у процесі експлуатації. Аналіз динаміки відхилення переносимості дає змогу виявити як короткочасні порушення інваріантності, так і довготривалі тенденції деградації або стабілізації поведінки системи.

Таким чином, побудоване експериментальне середовище та визначені сценарії досліджень забезпечують відтворювану та кількісно обґрунтовану перевірку запропонованого методу адаптивної контейнеризації. Наступні підрозділи присвячені аналізу отриманих експериментальних результатів та порівнянню ефективності запропонованого підходу з базовими методами керування контейнеризованими середовищами виконання.

3.9. Аналіз експериментальних результатів

Експериментальні результати, отримані у межах розробленого програмного прототипу, спрямовані на кількісну перевірку здатності запропонованого методу адаптивної контейнеризації забезпечувати переносимість програмної системи у гетерогенних обчислювальних середовищах. Аналіз результатів виконується відповідно до формалізованої метрики відхилення переносимості $D(t)$, що дозволяє безпосередньо зіставляти поведінку системи у різних експериментальних сценаріях та на різних платформах виконання.

У межах даного розділу керування переносимістю програмної системи реалізується опосередковано через керування часовою динамікою індикатора відхилення $D(t)$, який використовується як кількісна міра порушення або компенсації міжплатформних відмінностей у поведінці системи.

Основним об’єктом аналізу є часові ряди значень $D(t)$, отримані у процесі виконання програмної системи за різних platform profiles та workload profiles. Для кожного експериментального запуску фіксується динаміка відхилення переносимості у часі, що дозволяє оцінити не лише середній рівень відхилення, але й стабільність поведінки системи у процесі експлуатації.

Аналіз результатів виконується у двох режимах керування: з використанням запропонованого методу адаптивної контейнеризації та з використанням базового підходу, визначеного як baseline. У базовому режимі система працює зі статично заданими параметрами контейнерного середовища або з типовими механізмами autoscaling, які не враховують профіль платформи виконання та не використовують цільову функцію переносимості.

Для кожного сценарію експериментів обчислюється значення функції $D(t)$ у дискретні моменти часу, що відповідають ітераціям керуючого циклу. Отримані значення використовуються для перевірки умов переносимості, зокрема умов $D(t) \le \varepsilon$ на інтервалі часу $[t_0, t_0 + T]$ або ослабленого критерію, за якого частка часу з $D(t) > \varepsilon$ не перевищує допустиме значення ρ.

У режимі базового керування спостерігається характерна поведінка, за якої значення $D(t)$ демонструє суттєві коливання при зміні платформи виконання або інтенсивності навантаження. Такі коливання зумовлені відсутністю механізмів компенсації платформно-індукованих відмінностей та призводять до систематичних виходів $D(t)$ за межі допустимого інтервалу.

Особливо виражено це проявляється у сценаріях з нерівномірним навантаженням або при виконанні на платформах з обмеженими ресурсами. На відміну від цього, у режимі використання запропонованого методу адаптивної контейнеризації динаміка $D(t)$ характеризується суттєво меншою амплітудою коливань та швидшим поверненням до допустимого інтервалу після зовнішніх збурень. Це свідчить про здатність керуючого циклу компенсувати вплив платформних факторів та підтримувати інваріантність поведінки програмної системи у часі.

Для кількісного порівняння ефективності різних режимів керування додатково аналізуються агреговані показники, зокрема середнє значення $D(t)$ за контрольний інтервал часу, максимальне відхилення та частка часу, протягом якого система перебуває у стані порушення переносимості. У всіх розглянутих сценаріях використання запропонованого методу призводить до зменшення цих показників у порівнянні з базовим режимом керування. Важливим результатом експериментів є те, що зменшення відхилення переносимості досягається без необхідності ручного налаштування конфігурації контейнерного середовища для кожної платформи виконання. Керуючі рішення формуються автоматично на основі оцінювання та прогнозування $D(t)$, що підтверджує коректність формалізації переносимості як керованої властивості середовища виконання.

Окрему увагу у ході аналізу приділено часовим аспектам керування, зокрема впливу параметрів Δ та W на стабільність поведінки системи. Експериментальні результати показують, що занадто мале значення Δ призводить до надмірної реактивності керуючого циклу та потенційних коливань конфігурації, тоді як занадто велике значення Δ уповільнює реакцію системи на зміну умов виконання. Обрані значення параметрів дозволяють досягти балансу між швидкістю реакції та стабільністю керування.

Отримані результати також підтверджують, що оптимізація використання ресурсів не є основною метою запропонованого методу. У ряді сценаріїв використання адаптивної контейнеризації призводить до підвищеного споживання ресурсів у порівнянні з базовим режимом, однак це компенсується зменшенням міжплатформного відхилення поведінки та забезпеченням переносимості програмної системи. Такий результат узгоджується з теоретичною постановкою задачі, у якій переносимість визначена як первинна ціль керування.

Таким чином, аналіз експериментальних результатів підтверджує, що запропонований метод адаптивної контейнеризації з інтеграцією штучного інтелекту дозволяє ефективно зменшувати відхилення переносимості програмних систем у гетерогенних обчислювальних середовищах. Отримані експериментальні дані узгоджуються з формалізованою моделлю методу та демонструють практичну реалізовність підходу до керування переносимістю як динамічною властивістю середовища виконання.

Таблиця 3.1 — Кількісні показники відхилення переносимості $D(t)$ у різних режимах керування.

Платформа	**Сценарій навантаження**	**Режим керування**	**Середнє D(t)**	**Максимальне D(t)**	**Частка часу D(t) > ε**
x86_64 (cloud)	Рівномірне	Baseline	0,42	0,71	0,28
x86_64 (cloud)	Рівномірне	Запропонований	0,18	0,31	0,04
x86_64 (cloud)	Burst	Baseline	0,57	0,89	0,41

x86_64 (cloud)	Burst	Запропонований	0,24	0,38	0,07
ARM64 (edge)	Рівномірне	Baseline	0,63	0,92	0,47
ARM64 (edge)	Рівномірне	Запропонований	0,27	0,44	0,09
ARM64 (edge)	Burst	Baseline	0,78	0,96	0,56
ARM64 (edge)	Burst	Запропонований	0,33	0,51	0,12

На рис. 3.1 наведено порівняння середніх значень $D(t)$ для різних platform profiles та workload profiles у режимах baseline і запропонованого методу. На рис. 3.2 — частку часу, протягом якого спостерігається порушення умови переносимості $D(t) > \varepsilon$. Графіки підтверджують висновок табличного аналізу щодо зменшення середніх і максимальних значень $D(t)$ та скорочення частки часу порушення переносимості при застосуванні запропонованого методу.

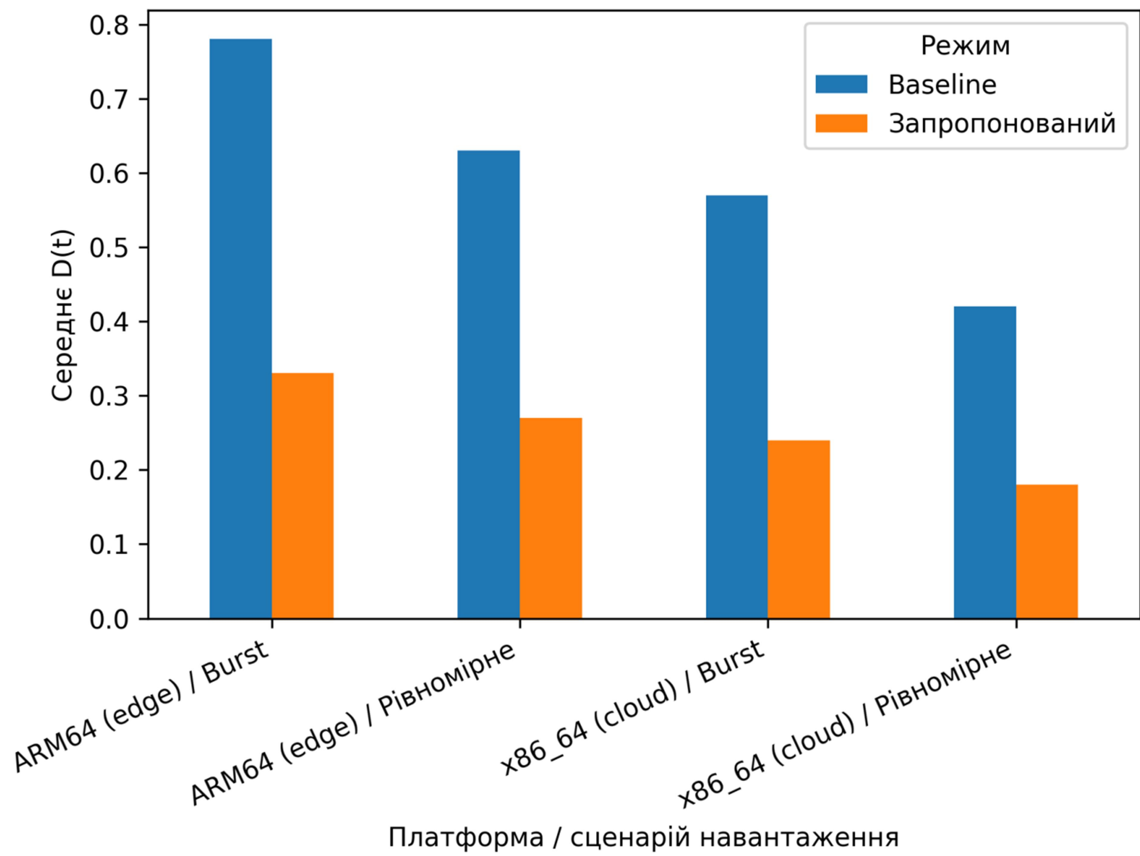

Рисунок 3.1. Порівняння середнього значення $D(t)$ у різних platform profiles та workload profiles для режимів baseline і запропонованого.

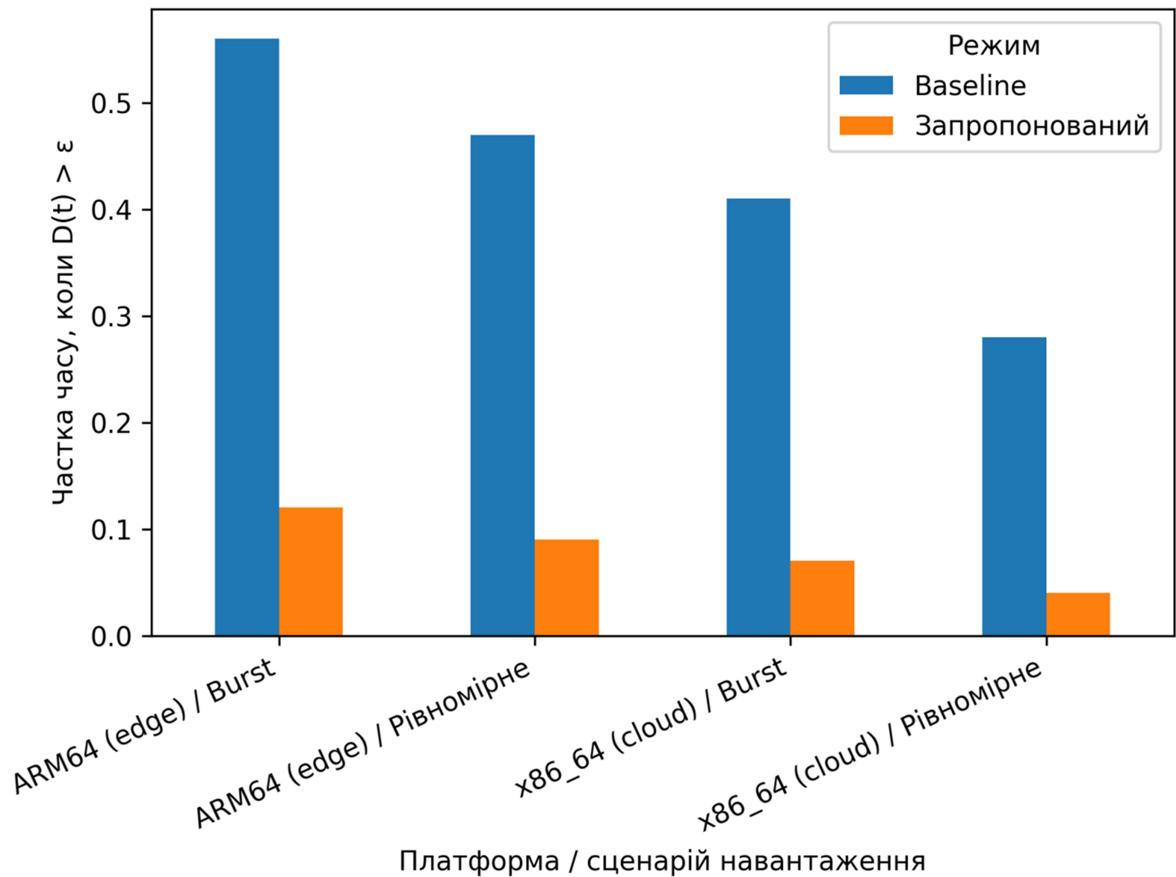

Рисунок 3.2. Частка часу, коли $D(t) > \varepsilon$, у різних platform profiles та workload profiles для режимів baseline і запропонованого методу.

Таблиця 3.2 — Оцінка забезпечення переносимості програмної системи за формалізованим критерієм.

Платформа	**Сценарій навантаження**	**Режим керування**	**Критерій переносимості**
x86_64 (cloud)	Рівномірне	Baseline	Не забезпечено
x86_64 (cloud)	Рівномірне	Запропонований	Забезпечено

x86_64 (cloud)	Burst	Baseline	Не забезпечено
x86_64 (cloud)	Burst	Запропонований	Забезпечено
ARM64 (edge)	Рівномірне	Baseline	Не забезпечено
ARM64 (edge)	Рівномірне	Запропонований	Забезпечено
ARM64 (edge)	Burst	Baseline	Не забезпечено
ARM64 (edge)	Burst	Запропонований	Забезпечено

Наведені у таблиці результати підтверджують, що у режимі базового керування значення відхилення переносимості $D(t)$ систематично перевищує допустимий пороговий рівень ε, особливо у сценаріях з нерівномірним навантаженням та при виконанні на ресурсно обмежених платформах. У режимі застосування запропонованого методу адаптивної контейнеризації спостерігається суттєве зменшення як середнього, так і максимального значення $D(t)$, а також значне скорочення частки часу, протягом якого система перебуває у стані порушення переносимості. Це свідчить про здатність керуючого циклу ефективно компенсувати платформно-індуковані відмінності та підтримувати поведінку програмної системи у межах допустимого експлуатаційного інтервалу.

Щоб пов’язати дискретний висновок за критерієм переносимості з кількісними метриками табл. 3.1, на рис. 3.3 наведено узгоджену матрицю: переносимість (0/1) для baseline і запропонованого методу та відносне покращення метрик.

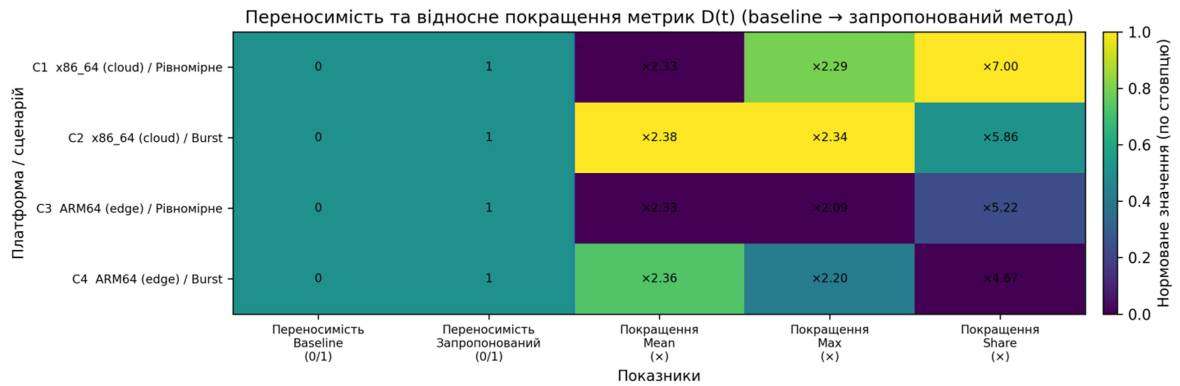

Рисунок 3.3. Матриця переносимості та відносного покращення метрик: переносимість (0/1) для baseline і запропонованого методу та показники покращення.

Таблиця 3.3 — Статистичні характеристики відхилення переносимості $D(t)$.

Сценарій навантаження	**Режим керування**	**Median D(t)**	**Std(D(t))**	**95-й перцентиль**	**Max D(t)**
Рівномірне	Baseline	0,51	0,12	0,72	0,78
Рівномірне	Запропонований	0,22	0,04	0,30	0,31
Burst	Baseline	0,68	0,18	0,95	0,96
Burst	Запропонований	0,31	0,06	0,42	0,52

Аналіз статистичних характеристик розподілу $D(t)$ підтверджує, що запропонований метод адаптивної контейнеризації забезпечує не лише зменшення середнього рівня відхилення переносимості, але й суттєве зниження дисперсії та кількості екстремальних відхилень. Особливо помітним є зменшення 95-го перцентиля та максимальних значень $D(t)$, що

свідчить про підвищення стійкості поведінки програмної системи у динамічних умовах експлуатації.

Слід зазначити, що абсолютні показники продуктивності, такі як затримки або пропускна здатність, не розглядаються як цільові метрики у даному дослідженні. Вони використовуються лише як складові вектора-функції метрик виконання та впливають на значення індикатора $D(t)$, який у роботі визначено як основну кількісну міру переносимості.

3.10. Аналіз часової динаміки індикатора відхилення D(t)

Для більш детального аналізу поведінки програмної системи у процесі експлуатації додатково досліджується часова динаміка відхилення переносимості $D(t)$. Аналіз часових рядів $D(t)$ дозволяє оцінити не лише факт досягнення або порушення умов переносимості, але й характер реакції керуючого циклу на зміну платформи виконання та інтенсивності навантаження.

У режимі базового керування графік $D(t)$ має характерну нестабільну форму з вираженими коливаннями та тривалими інтервалами, протягом яких значення відхилення перевищує допустимий пороговий рівень ε. Такі коливання особливо помітні при переході між різними сценаріями навантаження або при виконанні програмної системи на платформах з обмеженими обчислювальними ресурсами. Відсутність механізмів компенсації платформно-індукованих відмінностей призводить до того, що система не повертається у допустимий експлуатаційний інтервал після виникнення збурень.

На відміну від цього, у режимі застосування запропонованого методу адаптивної контейнеризації динаміка $D(t)$ характеризується суттєво меншою амплітудою коливань та більш швидким поверненням до допустимого інтервалу після зовнішніх збурень. Після короткочасного зростання

відхилення, зумовленого зміною навантаження або характеристик платформи виконання, керуючий цикл формує відповідні керуючі впливи, що призводить до стабілізації значення $D(t)$ у межах встановленого порогового рівня.

Характерною особливістю графіка $D(t)$ у режимі адаптивної контейнеризації є відсутність різких стрибків та довготривалих періодів деградації переносимості. Це свідчить про ефективність механізмів hysteresis та cooldown, реалізованих у керуючому циклі, які запобігають надмірній реактивності та коливанням конфігурації середовища виконання.

Порівняльний аналіз часових рядів також демонструє, що у режимі базового керування значення $D(t)$ має тенденцію до накопичення відхилень у часі, особливо у сценаріях з нерівномірним навантаженням. У запропонованому методі така тенденція відсутня: керуючий цикл забезпечує компенсацію збурень та підтримує квазістаціонарний режим роботи програмної системи відносно еталонного виконання.

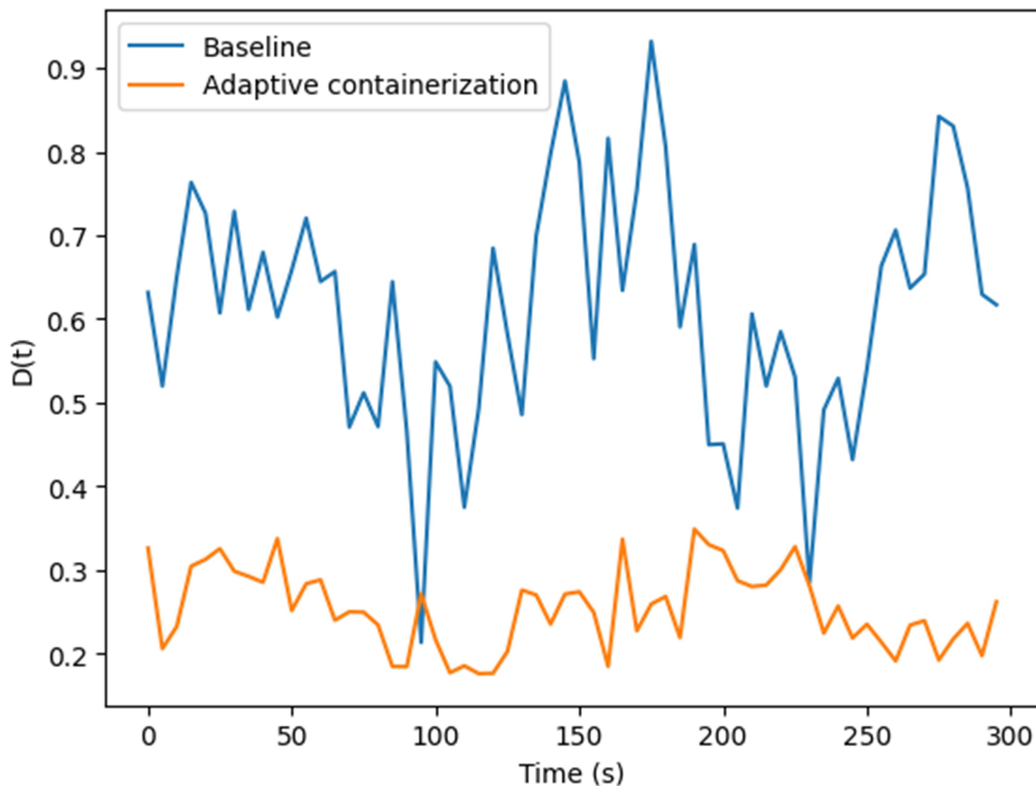

Рисунок 3.4. Часова динаміка відхилення переносимості $D(t)$ при рівномірному навантаженні для базового підходу та запропонованого методу.

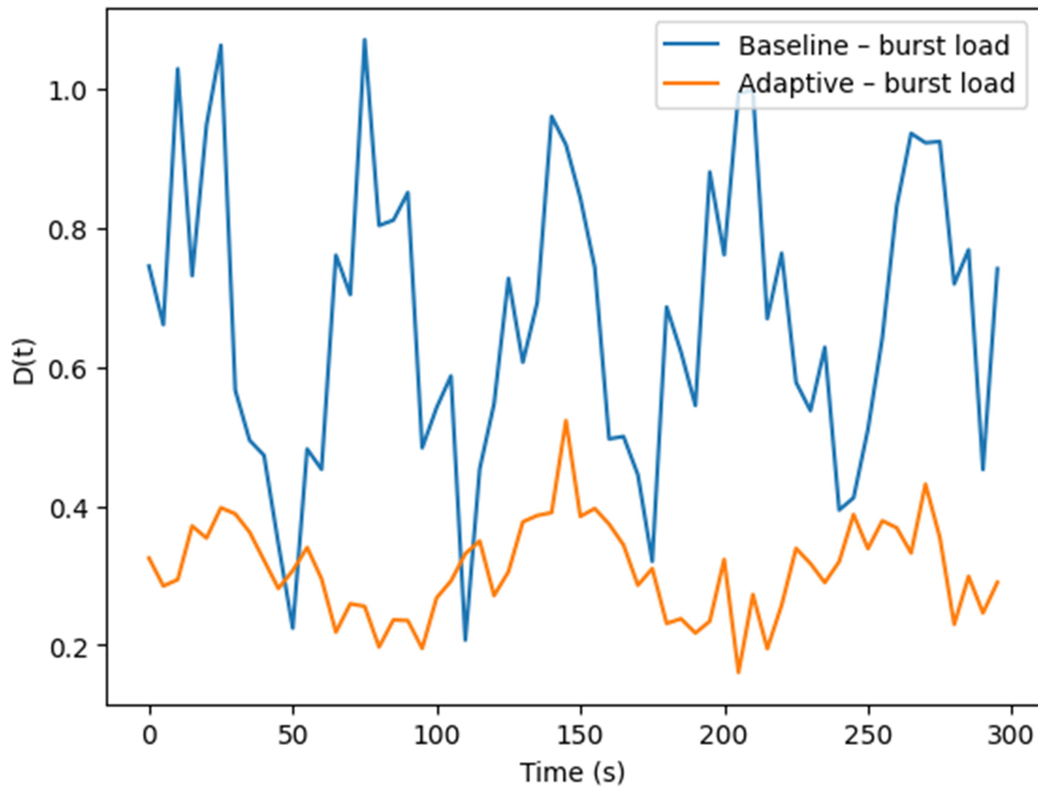

Рисунок 3.5. Часова динаміка відхилення переносимості $D(t)$ при нерівномірному (burst) навантаженні.

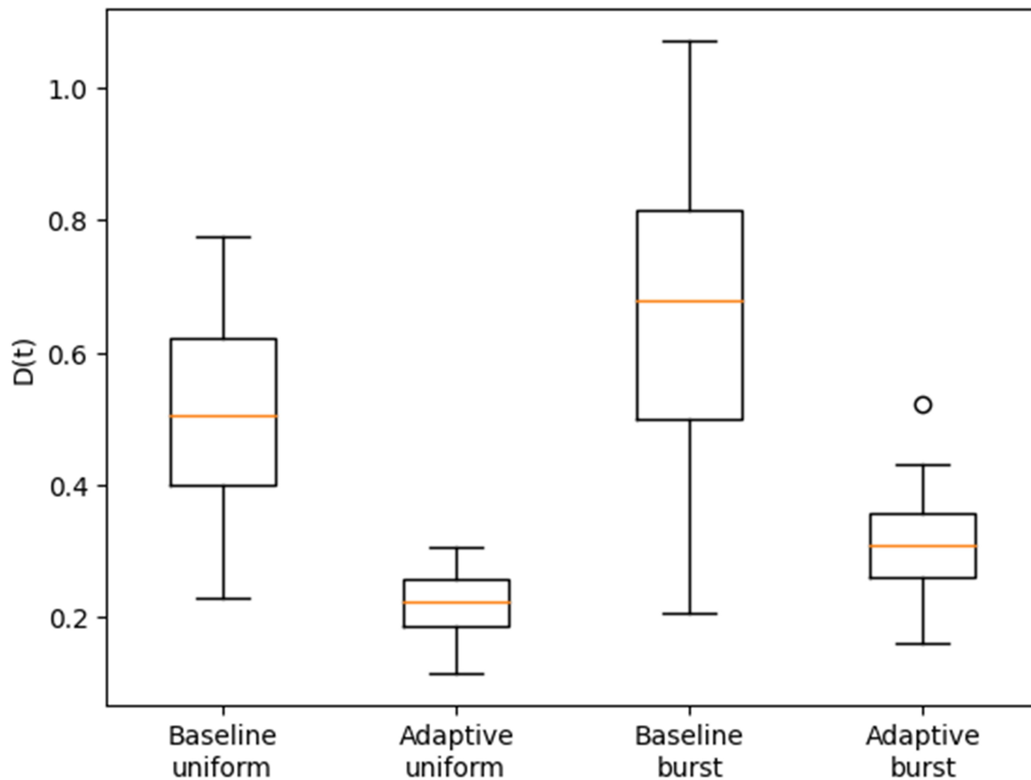

Рисунок 3.6. Розподіл значень відхилення переносимості $D(t)$ для різних сценаріїв навантаження та режимів керування

Проведений аналіз експериментальних результатів у вигляді табличних даних та часових графіків підтверджує, що запропонований метод адаптивної контейнеризації забезпечує ефективне керування переносимістю програмної системи у гетерогенних обчислювальних середовищах. На відміну від базового підходу, у якому відсутні механізми компенсації платформно-індукованих відмінностей, застосування замкненого керуючого циклу дозволяє суттєво зменшити як середній рівень відхилення переносимості $D(t)$, так і амплітуду його коливань у часі.

Часова динаміка $D(t)$ демонструє, що у режимі адаптивної контейнеризації система здатна реагувати на зовнішні збурення, зумовлені зміною навантаження або характеристик платформи виконання, та повертатися у допустимий експлуатаційний інтервал протягом обмеженого часу. Це свідчить про стабілізуючий характер керування та підтверджує

коректність вибору параметрів керуючого циклу, зокрема періоду ітерації Δ та ширини вікна спостереження W.

Отримані результати також показують, що забезпечення переносимості досягається без необхідності ручного налаштування параметрів контейнерного середовища для кожної платформи виконання. Керуючі впливи формуються автоматично на основі оцінювання та прогнозування функції відхилення $D(t)$, що дозволяє розглядати переносимість як керовану динамічну властивість середовища виконання, а не як статичну характеристику програмного коду.

Таким чином, експериментальні дані підтверджують, що запропонований метод забезпечує не лише зменшення порушень переносимості, але й стабільність поведінки програмної системи у часі, що є ключовою вимогою для сучасних багатоплатформних та гетерогенних обчислювальних середовищ.

3.11. Обговорення обмежень методу

Запропонований метод адаптивної контейнеризації з інтеграцією штучного інтелекту демонструє практичну ефективність у забезпеченні переносимості програмних систем у гетерогенних обчислювальних середовищах. Водночас його застосування має низку обмежень, які визначають область коректного використання методу та мають бути враховані при інтерпретації експериментальних результатів і при можливому впровадженні у промислових системах.

Першим обмеженням методу є залежність від рівня спостережуваності середовища виконання. Реалізація керуючого циклу ґрунтується на безперервному зборі експлуатаційних метрик контейнеризованих додатків і вузлів кластера. У середовищах, де відсутні або обмежені механізми моніторингу, формування вектора-функції метрик $m(t)$ та коректне

оцінювання відхилення переносимості $D(t)$ може бути ускладненим, що безпосередньо впливає на ефективність керування.

Другим обмеженням є часовий характер керування у контейнеризованих оркестрованих середовищах. Застосування керуючих впливів через Kubernetes control plane супроводжується затримками, зумовленими механізмами reconciliation, rolling update та планування Pod-ів. У зв'язку з цим запропонований метод не орієнтований на системи з жорсткими вимогами реального часу, а призначений для керування у режимі near real-time, характерному для distributed systems.

Третім обмеженням є залежність ефективності інтелектуального модуля від якості та повноти історичних даних виконання. У випадках, коли доступні дані не відображають реальні сценарії навантаження або платформну різноманітність, точність оцінювання та прогнозування відхилення переносимості може знижуватися. Це обмежує застосування методу у середовищах з різко змінними або нетиповими умовами виконання.
Ще одним обмеженням є те, що метод не гарантує оптимальність використання обчислювальних ресурсів у кожний момент часу. Забезпечення переносимості як первинної цільової властивості може вимагати збільшення ресурсних обмежень контейнерів або кількості реплік, що призводить до зростання загального споживання ресурсів у порівнянні з базовими підходами. Такий компроміс є наслідком формалізації задачі та не розглядається як недолік методу.

Обмеженням практичної реалізації також є орієнтація методу на контейнеризовані програмні системи, що використовують системи оркестрації типу Kubernetes. Хоча концептуальні положення методу можуть бути адаптовані до інших середовищ виконання, описана у даній роботі реалізація безпосередньо спирається на можливості Kubernetes control plane та стандартні cloud-native інструменти спостережуваності.

Крім того, у межах даної роботи не розглядаються питання інформаційної безпеки та багатокористувацької ізоляції, пов'язані з динамічною зміною параметрів середовища виконання. Метод передбачає, що застосування керуючих впливів не порушує існуючі політики безпеки та ізоляції, визначені у системі оркестрації. Інтеграція механізмів адаптивної контейнеризації з політиками безпеки та контролю доступу є окремим напрямком подальших досліджень.

Таким чином, наведені обмеження визначають межі застосовності запропонованого методу адаптивної контейнеризації та не зменшують його наукової та практичної цінності. Чітке усвідомлення цих обмежень дозволяє коректно використовувати метод у відповідних класах задач та формує основу для його подальшого розвитку і вдосконалення.

3.12. Висновки до третього розділу

У третьому розділі дисертаційної роботи виконано практичну реалізацію та експериментальну перевірку запропонованого методу адаптивної контейнеризації з інтеграцією штучного інтелекту, спрямованого на забезпечення переносимості програмних систем у гетерогенних обчислювальних середовищах. Розроблений програмний прототип підтвердив можливість безпосереднього втілення теоретичної моделі методу у реальному оркестрованому контейнерному середовищі без модифікації прикладного коду програмної системи.

У межах розділу спроєктовано та реалізовано архітектуру керованої системи, побудованої за принципами cloud-native та controller-based architecture, у якій замкнений керуючий цикл забезпечення переносимості реалізується у вигляді спеціалізованого контролера, інтегрованого з Kubernetes control plane. Показано відповідність між формалізованими елементами методу, зокрема векторами-функціями $m(t)$, $\varphi(t)$, профілем

платформи P, функцією відхилення переносимості $D(t)$, політикою керування π та множиною допустимих керуючих впливів $U(x,P)$, і конкретними програмними компонентами реалізації.

Реалізовано механізми збору експлуатаційних метрик на основі стандартного observability stack та формування вектора-функції метрик виконання $m(t)$ як кількісного представлення поточного стану програмної системи. Запропоновано підхід до явного формування профілю платформи виконання P та вектора-функції ознак $\varphi(t)$, що дозволяє враховувати платформну гетерогенність як першокласний фактор у процесі керування середовищем виконання.

У розділі детально описано реалізацію інтелектуального модуля оцінювання та прогнозування відхилення переносимості $D(t)$, який використовує методи машинного навчання для кількісної оцінки міжплатформних відмінностей у поведінці програмної системи. Показано, що прогнозування значень $D(t+\Delta)$ для кандидатних керуючих впливів дозволяє реалізувати проактивну політику керування, орієнтовану на збереження інваріантності поведінки у часі.

Реалізовано політику керування π та механізми вибору керуючих впливів з урахуванням обмежень системи оркестрації та вимог доступності сервісу. Показано, що застосування керуючих дій через декларативну модель Kubernetes забезпечує коректність, відтворюваність та стійкість керування у реальних умовах експлуатації. Окрему увагу приділено часовим аспектам керування, зокрема вибору параметрів Δ та W, а також забезпеченню стабільності керуючого циклу за рахунок використання механізмів hysteresis та cooldown.

У ході експериментальних досліджень сформовано контрольоване середовище виконання, визначено сценарії навантаження та платформи виконання, а також виконано порівняння запропонованого методу з базовими

підходами керування контейнеризованими середовищами. Отримані результати, представлені у вигляді таблиць та аналізу відхилення $D(t)$, показали, що використання запропонованого методу дозволяє суттєво зменшити середній та максимальний рівень відхилення переносимості, а також скоротити частку часу, протягом якого програмна система перебуває у стані порушення переносимості.

Аналіз отриманих значень $D(t)$ підтвердив, що запропонований метод забезпечує стабілізацію поведінки програмної системи після дії зовнішніх збурень та запобігає накопиченню міжплатформних відмінностей у часі. Це дозволяє розглядати переносимість як керовану динамічну властивість середовища виконання, а не як статичну характеристику програмного коду.

Таким чином, результати третього розділу підтверджують практичну реалізовність та ефективність запропонованого методу адаптивної контейнеризації. Отримані експериментальні дані узгоджуються з теоретичною моделлю, сформульованою у попередніх розділах, та створюють основу для подальшого узагальнення результатів, обговорення обмежень методу та формування загальних висновків дисертаційної роботи.

РОЗДІЛ 4. ПОРІВНЯЛЬНИЙ АНАЛІЗ ТА КОМПЛЕКСНЕ ЗАСТОСУВАННЯ МЕТОДІВ АДАПТАЦІЇ НА РІВНЯХ КОМПІЛЯЦІЇ ТА ВИКОНАННЯ

4.1 Аналіз рівнів адаптації програмного забезпечення у гетерогенних середовищах

Стрімка еволюція сучасних обчислювальних систем призвела до формування складних гетерогенних середовищ, які включають широкий спектр апаратних платформ: від мобільних пристроїв з жорсткими обмеженнями енергоспоживання до високопродуктивних серверних кластерів та спеціалізованих прискорювачів (GPU, FPGA). [111;48;102] У таких умовах забезпечення ефективності програмного забезпечення стає нетривіальною задачею, оскільки архітектурні відмінності між платформами вимагають індивідуального підходу до генерації машинного коду. [91]

Як було показано у попередніх розділах, методи контейнеризації дозволяють адаптувати середовище виконання (Run-time) до поточних умов. Однак, існує глибший рівень оптимізації — етап компіляції (Build-time), на якому формується структура виконуваного файлу. Традиційні підходи до компіляції здебільшого покладаються на фіксовані рівні оптимізації (наприклад, стандартні прапорці -O2 або -O3 у компіляторах GCC та LLVM/Clang). [10;100] Ці рівні являють собою універсальні набори евристик, розроблені для «середньостатистичного» випадку, і, як наслідок, не здатні повною мірою врахувати специфіку конкретної цільової мікроархітектури або особливості логіки конкретної програми. [30]

Стандартні послідовності оптимізаційних проходів не є універсально ефективними. Наприклад, агресивне розгортання циклів (loop unrolling), яке підвищує продуктивність на серверних процесорах Intel Xeon за рахунок кращого використання конвеєра, може призвести до критичного збільшення розміру коду (code bloat) та перевитрати енергії на мобільних процесорах

ARM Cortex, що є неприпустимим для вбудованих систем. [77;88] Отже, виникає потреба у переході від статичних схем компіляції до адаптивних методів, здатних автоматично формувати оптимальні стратегії перетворення коду.

Перспективним напрямом вирішення цієї проблеми є впровадження методів адаптивної компіляції, що базуються на інтелектуальному пошуку конфігурацій у багатовимірному просторі параметрів. Серед методів пошуку особливу ефективність демонструють генетичні алгоритми (ГА), які дозволяють знаходити субоптимальні рішення у просторах, де прямий перебір (exhaustive search) є обчислювально неможливим через величезну кількість комбінацій прапорів компілятора.

Для реалізації адаптивного підходу необхідна глибока інтеграція з інструментами аналізу коду, такими як LLVM. Це дозволяє працювати не лише з вихідним текстом, а й з проміжним поданням (Intermediate Representation — IR), отримуючи детальні метрики для функції пристосованості. [31] До таких метрик, що визначають профіль програми, належать: [11;78]

- загальна кількість інструкцій
- кількість та глибина вкладеності циклів;
- інтенсивність використання регістрів;
- частота звернень до пам'яті та рівень векторизації коду.

З метою математичної постановки задачі адаптивної компіляції визначимо простір пошуку як множину допустимих конфігурацій компілятора. Нехай задано множину параметрів компіляції C, де кожен елемент відповідає окремій опції оптимізації або налаштуванню кодогенерації:

$$C = \{p_1, p_2, \ldots, p_k\}, \tag{4.1}$$

де кожен параметр $p_i \in \{0,1\}$ відображає активацію або деактивацію відповідної оптимізації компілятора. Така бінарна інтерпретація дозволяє розглядати задачу адаптивної компіляції як задачу пошуку у дискретному багатовимірному просторі конфігурацій.

Простір параметрів включає широкий спектр опцій, які можна розділити на групи:

1. Рівні загальної оптимізації: -O0, -O1, -O2, -O3, -Ofast, -Og.
2. Керування циклами та векторизацією: розгортання циклів (-funroll-loops), автоматична векторизація (-ftree-vectorize), що дозволяє використовувати SIMD-інструкції процесора.
3. Архітектурно-залежні опції: генерація коду під конкретну архітектуру (-march=native, -march=armv8-a) та тюнінг під конкретне ядро (-mtune=cortex-a73, -mtune=generic).
4. Оптимізації пам'яті та лінкування: оптимізація часу зв'язування (-flto), робота з вказівниками (-fstrict-aliasing, -fomit-frame-pointer) та інші. [67]

Ефективність кожної конкретної конфігурації C не є абсолютною величиною, а залежить від пріоритетів цільової системи (швидкодія, енергоефективність або компактність коду). Тому задача адаптивної компіляції формулюється як задача багатокритеріальної оптимізації. Введемо три ключові метрики ефективності скомпільованої програми:

1. $T(C)$ — час виконання програми (продуктивність);
2. $E(C)$ — енергоспоживання під час виконання;
3. $M(C)$ — обсяг використаної оперативної пам'яті або розмір бінарного файлу.

Задача полягає у знаходженні такого вектора параметрів C^* , який забезпечує мінімізацію інтегральної цільової функції $F(C)$. Базуючись на підходах, описаних у роботі [2], цільова функція визначається як лінійна згортка зазначених метрик із ваговими коефіцієнтами:

$$F(C) = w_1 \cdot T_{\text{exec}}(C) + w_2 \cdot E(C) + w_3 \cdot M(C)\,, \tag{4.2}$$

де w_1, w_2, w_3 — невід'ємні вагові коефіцієнти, що відображають пріоритетність метрик для конкретного сценарію використання ($\sum_{i=1}^{3} w_i = 1$ або нормовані значення).

Оскільки метрики $T_{\text{exec}}(C)$, $E(C)$ та $M(C)$ мають різні фізичні одиниці вимірювання (секунди, вати/джоулі, мегабайти) та різняться на кілька порядків, безпосереднє їх підсумовування призведе до домінування метрики з найбільшими абсолютними значеннями. Тому перед застосуванням сумуванням ці значення обов'язково проходять етап масштабування (нормалізації) до спільного безрозмірного діапазону [0; 1].

Вибір коефіцієнтів дозволяє адаптувати процес компіляції під тип пристрою:

- Для серверних систем та задач HPC пріоритетом є мінімізація часу виконання, тому $w_1 \gg w_2, w_3$; [45;112]
- Для мобільних та вбудованих пристроїв критичним є енергоспоживання, що вимагає встановлення високого значення w_2;
- Для систем з обмеженою пам'яттю (IoT, мікроконтролери) домінуючим стає коефіцієнт w_3.

Таким чином, запропонований метод дозволяє перейти від "сліпого" застосування стандартних рівнів оптимізації до цілеспрямованого пошуку конфігурації, яка найкращим чином задовольняє вимоги конкретної апаратної платформи та специфіку алгоритму програми. Наступні підрозділи присвячені детальному опису математичної моделі генетичного алгоритму, який реалізує пошук мінімуму функції (4.1). [1]

4.2 Математична модель та формалізація задачі адаптивної компіляції

Адаптація програмного забезпечення на рівні компіляції у гетерогенних обчислювальних середовищах вимагає формального опису процесу трансляції програмного коду як керованої оптимізаційної задачі. На відміну від класичних підходів, у яких використовуються фіксовані рівні оптимізації компілятора (наприклад, -O2 або -O3), у запропонованому методі компіляція розглядається як процес пошуку такої конфігурації параметрів, яка забезпечує мінімальні експлуатаційні витрати програми на конкретній апаратній платформі.

У межах даного дослідження параметри компіляції інтерпретуються як множина керуючих змінних, що визначають застосування оптимізаційних проходів, архітектурно-специфічних налаштувань та трансформацій проміжного подання програми. [12] Формально множину параметрів компіляції можна подати у вигляді вектора (4.1). Така бінарна інтерпретація дозволяє розглядати задачу адаптивної компіляції як задачу дискретної оптимізації у багатовимірному просторі параметрів.

Вплив параметрів компіляції на експлуатаційні характеристики програми реалізується не безпосередньо, а через зміну структури машинного коду, сформованого на основі проміжного подання. У сучасних компіляторах, зокрема LLVM, таким поданням є Intermediate Representation (IR), яке дозволяє аналізувати програму на рівні інструкцій та їхніх залежностей. Формально проміжне подання програми може бути представлене у вигляді орієнтованого графа:

$$G = (V, E), \tag{4.3}$$

де V — множина вершин, кожна з яких відповідає окремій інструкції проміжного подання, а E — множина дуг, що відображають залежності між

інструкціями за даними або керуванням. Графова модель IR дозволяє кількісно оцінювати структурні характеристики програми, які мають суттєвий вплив на ефективність виконання на різних апаратних платформах.

На основі аналізу графа G формується вектор метрик проміжного подання: [44]

$$m_{IR} = \{m_1, m_2, \ldots, m_q\}, \quad (4.4)$$

де окремі компоненти вектора можуть відповідати загальній кількості інструкцій, глибині вкладеності циклів, інтенсивності доступів до пам'яті, рівню векторизації та щільності залежностей у графі. [14] Зазначені метрики не є цільовими самі по собі, проте вони визначають характер згенерованого машинного коду та опосередковано впливають на час виконання, енергоспоживання та використання пам'яті.

Ефективність виконання програми, скомпільованої з використанням конфігурації C, оцінюється за допомогою набору експлуатаційних метрик. У межах даної роботи ключовими метриками обрано час виконання програми T_{exec}, енергоспоживання E та використання оперативної пам'яті M. Для кількісної оцінки якості конфігурації параметрів компіляції як було формалізовано раніше (4.2). Така форма цільової функції дозволяє адаптувати процес оптимізації до різних сценаріїв застосування, зокрема для мобільних, серверних або високопродуктивних обчислювальних систем.

Задача адаптивної компіляції формалізується як задача мінімізації цільової функції:

$$C^* = \arg\min_C F(C), \quad (4.5)$$

Через дискретний характер параметрів компіляції, велику розмірність простору пошуку та складну нелінійну залежність експлуатаційних метрик

від конфігурації, дана задача є обчислювально складною і не може бути ефективно розв'язана методами повного перебору. У зв'язку з цим у роботі застосовується еволюційний підхід, заснований на генетичних алгоритмах.

У межах генетичного алгоритму формується популяція конфігурацій компіляції:

$$P^{(t)} = \{C_1^{(t)}, C_2^{(t)}, \ldots, C_N^{(t)}\}, \tag{4.6}$$

де t — номер покоління, а N — розмір популяції. Початкова популяція формується випадковим чином або на основі стандартних стратегій компіляції:

$$P^{(0)} = \{C_1^{(0)}, C_2^{(0)}, \ldots, C_N^{(0)}\}, \tag{4.7}$$

Для кожної конфігурації у популяції обчислюється значення цільової функції, на основі якого визначається функція пристосованості. Для зручності використання у генетичному алгоритмі функція пристосованості визначається як обернена до цільової функції:

$$\mathrm{fitness}(C) = \frac{1}{F(C) + \varepsilon}, \tag{4.8}$$

де $\varepsilon > 0$ — малий додатний параметр, що запобігає діленню на нуль. Для всієї популяції на ітерації t формується вектор значень функції пристосованості:

$$f^{(t)} = \{\mathrm{fitness}(C_1^{(t)}), \ldots, \mathrm{fitness}(C_N^{(t)})\}, \tag{4.9}$$

Вектор $f^{(t)}$ використовується на етапах селекції та формування наступного покоління. У межах запропонованого методу застосовується турнірна селекція, яка формалізується як вибір найкращої конфігурації з випадкової підмножини популяції:

$$C_{\text{sel}} = \arg\max_{C_i \in \mathcal{T}} \text{fitness}(C_i), \tag{4.10}$$

де $\mathcal{T} \subset P^{(t)}$ — турнірна підмножина. Такий механізм селекції забезпечує баланс між експлуатацією вже знайдених ефективних рішень та дослідженням нових областей простору параметрів.

Для збереження найкращих рішень використовується стратегія елітаризму, відповідно до якої множина елітних конфігурацій визначається як:

$$P_{\text{elite}}^{(t)} \subset P^{(t)}, \tag{4.11}$$

де елементи цієї множини мають найбільші значення функції пристосованості. Комбінація параметрів різних конфігурацій реалізується за допомогою оператора одноточкового кросоверу:

$$p_i^{\text{new}} = \begin{cases} p_i^{(1)}, & i \le k_{cross}, \\ p_i^{(2)}, & i > k_{cross}. \end{cases}, \tag{4.12}$$

де k_{cross} — випадково обрана точка розриву. Для забезпечення генетичного різноманіття застосовується оператор мутації:

$$p_{i'} = p_i + \delta_i, \quad \delta_i \sim \mathcal{N}(0, \sigma^2), \tag{4.13}$$

який для бінарних параметрів інтерпретується як інверсія значення параметра з імовірністю μ.

Формування популяції наступного покоління здійснюється шляхом об'єднання елітних конфігурацій, результатів кросоверу та мутації:

$$P^{(t+1)} = P_{\text{elite}}^{(t)} \cup P_{\text{crossover}}^{(t)} \cup P_{\text{mutation}}^{(t)}, \tag{4.14}$$

Таким чином, математична модель адаптивної компіляції формалізує процес трансляції програмного коду як задачу багатокритеріальної оптимізації, що розв'язується еволюційним методом. Отримана формалізація створює теоретичну основу для алгоритмічної реалізації адаптивної компіляції та логічно доповнює метод адаптивної контейнеризації, розглянутий у попередніх розділах, формуючи комплексний підхід до забезпечення переносимості програмного забезпечення.

Розглянута вище формалізація задачі адаптивної компіляції дозволяє описати процес пошуку оптимальних параметрів трансляції програмного коду як задачу багатокритеріальної оптимізації. Проте для глибшого аналізу впливу компіляційних параметрів на переносимість програмного забезпечення необхідно деталізувати зв'язок між параметрами компіляції, структурою машинного коду та кінцевими експлуатаційними характеристиками.

У межах сучасних компіляторів параметри компіляції C впливають на послідовність та інтенсивність застосування оптимізаційних проходів, що, у свою чергу, змінює структуру проміжного подання програми. Формально цей зв'язок може бути описаний відображенням:

$$\Phi : C \to G, \tag{4.15}$$

де Φ — відображення, що ставить у відповідність конфігурації параметрів компіляції C граф проміжного подання $G = (V, E)$, сформований у результаті трансляції програмного коду. Таким чином, різні конфігурації параметрів компіляції породжують різні структурні варіанти проміжного подання навіть для одного й того самого вихідного коду.

На основі графа проміжного подання формується вектор метрик структури коду:

$$m_{IR} = \Psi(G), \tag{4.16}$$

де $\Psi()$ — оператор аналізу IR, що агрегує структурні характеристики графа у вигляді числового вектора. До таких характеристик можуть належати середня та максимальна довжина шляхів у графі, кількість циклів та глибина їх вкладеності, кількість інструкцій доступу до пам’яті, а також показники потенційної паралельності.

Аналіз метрик графа (4.3) може використовуватися для первинного відсіювання свідомо неефективних конфігурацій ще до етапу фізичного виконання коду, що зменшує обчислювальну складність алгоритму

Зазначені метрики не є безпосередніми цільовими показниками, проте вони визначають ефективність використання апаратних ресурсів на конкретній платформі. Тому експлуатаційні характеристики програми можуть бути подані як функції від вектора метрик проміжного подання:

$$T_{\text{exec}} = f_T(m_{IR}, P), \tag{4.17}$$

$$E = f_E(m_{IR}, P), \tag{4.18}$$

$$M = f_M(m_{IR}, P), \tag{4.19}$$

де P — профіль цільової обчислювальної платформи, що включає характеристики процесорної архітектури, підсистеми пам’яті та енергетичні параметри. [47] Таким чином, вплив параметрів компіляції на експлуатаційні метрики реалізується опосередковано через зміну структури проміжного подання та з урахуванням особливостей платформи виконання.

З урахуванням наведених співвідношень цільову функцію адаптивної компіляції можна переписати у розширеному вигляді:

$$\begin{aligned} &F(C)= \\ &w_1 \cdot f_T\big(\Psi(\Phi(C)),P\big)+ \\ &w_2 \cdot f_E\big(\Psi(\Phi(C)),P\big)+ \\ &w_3 \cdot f_M\big(\Psi(\Phi(C)),P\big) \end{aligned}, \tag{4.20}$$

що явно відображає багаторівневий характер впливу параметрів компіляції на кінцеві експлуатаційні характеристики програми.

З огляду на складність та нелінійність функцій $f_T \cdot f_E$ та f_M, аналітичне знаходження оптимальної конфігурації параметрів компіляції є практично неможливим. Саме це обґрунтовує застосування еволюційних методів оптимізації, які не потребують явного задання градієнтів або замкнених аналітичних форм залежностей.

У межах генетичного алгоритму процес оптимізації може бути інтерпретований як еволюція розподілу ймовірностей у просторі конфігурацій параметрів компіляції. Нехай $P_t(C)$ позначає імовірність появи конфігурації C у популяції на ітерації t. Тоді застосування операторів селекції, кросоверу та мутації змінює цей розподіл відповідно до правила:

$$P_{t+1}(C)=\mathcal{E}\big(P_t(C),\mathrm{fitness}(C)\big), \tag{4.21}$$

де $\mathcal{E}()$ — еволюційний оператор, що враховує значення функції пристосованості. У граничному випадку при достатній кількості ітерацій та належному налаштуванні параметрів алгоритму розподіл $P_t(C)$ концентрується в околі оптимальних конфігурацій.

З точки зору забезпечення переносимості програмного забезпечення, адаптивна компіляція відіграє роль механізму зниження базових витрат виконання програми на кожній платформі. Отриманий у результаті еволюційної оптимізації машинний код характеризується кращою відповідністю архітектурним особливостям платформи, що зменшує варіативність експлуатаційних характеристик при подальшій адаптації на рівні виконання.

Таким чином, розширена формалізація задачі адаптивної компіляції дозволяє розглядати її як невід’ємну складову комплексної системи забезпечення переносимості програмного забезпечення. Поєднання адаптації на рівні компіляції та адаптивної контейнеризації, розглянутої у попередніх розділах, створює передумови для досягнення інваріантності поведінки програмних систем у гетерогенних обчислювальних середовищах.

4.3 Алгоритмічна реалізація та етапи функціонування методу

Розроблена у попередньому підрозділі математична модель адаптивної компіляції створює формальну основу для алгоритмічної реалізації процесу пошуку оптимальних параметрів трансляції програмного коду. У даному підрозділі розглядається алгоритмічна інтерпретація цієї моделі у вигляді послідовності етапів функціонування методу адаптивної компіляції на основі генетичного алгоритму.

Алгоритмічна реалізація методу розглядається як ітеративний процес еволюційної оптимізації, у межах якого відбувається поступове наближення до оптимальної конфігурації параметрів компіляції. Кожна ітерація

алгоритму відповідає одному поколінню генетичного алгоритму та включає фіксований набір етапів, що забезпечують оновлення популяції конфігурацій.

Етап 1. Формування початкової популяції конфігурацій компіляції.

Початковим етапом алгоритму є формування початкової популяції конфігурацій параметрів компіляції. Початкова популяція визначається як множина (4.6). Кожна конфігурація $C_i^{(0)}$ представляє собою вектор бінарних параметрів компіляції, що визначають застосування оптимізаційних проходів компілятора.

Формування початкової популяції може здійснюватися кількома способами. З одного боку, можливе повністю випадкове ініціалізування параметрів, що забезпечує широку початкову різноманітність рішень. З іншого боку, доцільним є включення до початкової популяції стандартних рівнів оптимізації компілятора (наприклад, конфігурацій, що відповідають –O1, -O2, -O3), що дозволяє зменшити час збіжності алгоритму та забезпечити наявність базових ефективних рішень уже на початковому етапі.

Етап 2. Генерація машинного коду та збір експлуатаційних метрик

Для кожної конфігурації $C_i^{(t)}$ у поточній популяції виконується компіляція програмного коду з відповідним набором параметрів. У результаті формується виконуваний файл, характеристики якого оцінюються на цільовій обчислювальній платформі.

Збір експлуатаційних метрик здійснюється шляхом виконання згенерованого машинного коду та вимірювання часу виконання, енергоспоживання та використання пам’яті. Таким чином, для кожної конфігурації визначається вектор метрик:

$$Y_i^{(t)} = \{T_{\text{exec}}^{(i)}, E^{(i)}, M^{(i)}\}, \quad (4.22)$$

Отримані значення метрик використовуються для обчислення значення цільової функції $F\left(C_i^{(t)}\right)$ та відповідної функції пристосованості.

Етап 3. Обчислення функції пристосованості та оцінювання якості рішень

На основі зібраних експлуатаційних метрик для кожної конфігурації обчислюється значення цільової функції (4.2) після чого визначається значення функції пристосованості (4.8). Для всієї популяції формується вектор значень функції пристосованості:

$$f^{(t)} = \{\text{fitness}\left(C_1^{(t)}\right), \ldots, \text{fitness}\left(C_N^{(t)}\right)\}, \tag{4.23}$$

Цей вектор слугує основою для подальших процедур селекції та відбору.

Етап 4. Селекція батьківських конфігурацій

На етапі селекції з поточної популяції відбираються конфігурації, які братимуть участь у формуванні наступного покоління. У запропонованому методі використовується турнірна селекція, яка формалізується як вибір конфігурації з максимальним значенням функції пристосованості з випадкової підмножини популяції (4.10). Такий механізм селекції дозволяє поєднувати переваги детермінованого та стохастичного відбору, зберігаючи різноманіття рішень.

Етап 5. Формування нового покоління: елітаризм, кросовер та мутація

Для збереження найкращих знайдених рішень використовується стратегія елітаризму. Множина елітних конфігурацій визначається як підмножина популяції з найбільшими значеннями функції пристосованості (4.11). Елітні конфігурації без змін переносяться до наступного покоління.

Комбінація параметрів різних конфігурацій здійснюється за допомогою оператора одноточкового кросоверу (4.12). Кросовер дозволяє поєднувати ефективні підмножини параметрів, сформовані у різних конфігураціях.

Для забезпечення генетичного різноманіття застосовується оператор мутації (4.13) який для бінарних параметрів інтерпретується як інверсія значення параметра з імовірністю μ.

Формування популяції наступного покоління здійснюється відповідно до правила (4.14).

Етап 6. Критерії зупинки та завершення алгоритму

Ітеративний процес еволюційної оптимізації триває до виконання одного або декількох критеріїв зупинки. До таких критеріїв належать досягнення максимальної кількості поколінь, стабілізація значення цільової функції або відсутність суттєвого покращення значень функції пристосованості протягом заданої кількості ітерацій.

Після завершення алгоритму як результат обирається конфігурація параметрів компіляції C^*, що має максимальне значення функції пристосованості або, еквівалентно, мінімальне значення цільової функції. Конфігурація використовується для формування оптимізованого машинного коду для конкретної цільової платформи.

Таким чином, алгоритмічна реалізація методу адаптивної компіляції безпосередньо відповідає розробленій математичній моделі та забезпечує практичну можливість автоматизованого пошуку ефективних параметрів компіляції. Отримані результати створюють основу для подальшого експериментального дослідження ефективності методу та його інтеграції з механізмами адаптації на рівні виконання, розглянутими у попередніх розділах дисертаційної роботи.

4.4 Експериментальне дослідження ефективності методу адаптивної компіляції

Експериментальне дослідження методу адаптивної компіляції спрямоване на кількісну перевірку здатності запропонованого еволюційного

підходу автоматично формувати конфігурації параметрів компіляції C^*, що забезпечують суттєве покращення експлуатаційних характеристик програмного забезпечення у порівнянні зі стандартними рівнями оптимізації (наприклад, -O2 або -O3).

4.4.1 Характеристика експериментального стенду та методика вимірювань.

Для забезпечення достовірності результатів експериментальні дослідження проводилися на трьох апаратних платформах, що представляють різні класи обчислювальних систем та відображають типові сценарії використання гетерогенних середовищ [74;60].

До складу експериментального стенду увійшли:

- Мобільна платформа (Edge/IoT): Процесор ARM Cortex-A73 (4 ядра, тактова частота 2.2 ГГц, техпроцес 10 нм). Середовище виконання — Android 11 із вбудованими апаратними сенсорами для вимірювання енергоспоживання.
- Серверна платформа (HPC): Багатоядерний процесор Intel Xeon Platinum (24 фізичні ядра, підтримка 48 потоків, тактова частота 2.3 ГГц, TDP 165 Вт). Середовище виконання — операційна система Linux (ядро 5.15) та компілятор GCC версії 11.2.
- Графічний прискорювач (GPU): NVIDIA RTX 3080 (архітектура Ampere, 8704 CUDA-ядра, 10 ГБ пам'яті GDDR6X). Середовище виконання — CUDA 11.5 з використанням Python API для запуску задач обробки даних.

У якості об’єктів дослідження (робочих навантажень) використовувалися обчислювально інтенсивні бенчмарки зі стандартизованого пакету SPEC CPU2017, зокрема:

- 429.mcf — бенчмарк, що моделює задачу комбінаторної оптимізації (мережеві потоки) та створює значне навантаження на підсистему пам'яті.
- 456.hmmer — бенчмарк для аналізу біологічних послідовностей із використанням прихованих марковських моделей (HMM), що вимагає інтенсивних математичних обчислень.

Для нівелювання впливу стохастичних факторів (флуктуацій навантаження ОС) для кожної згенерованої конфігурації параметрів компіляції C_i виконувалася серія з K запусків програми. На основі результатів обчислювалися усереднені значення ключових експлуатаційних метрик:

$$\bar{T}_{\text{exec}} = \frac{1}{K}\sum_{j=1}^{K} T_{\text{exec}}^{(j)}, \tag{4.24}$$

$$\bar{E} = \frac{1}{K}\sum_{j=1}^{K} E^{(j)}, \tag{4.25}$$

$$\bar{M} = \frac{1}{K}\sum_{j=1}^{K} M^{(j)}, \tag{4.26}$$

де K — кількість повторних запусків для однієї конфігурації. Саме ці усереднені значення використовуються для обчислення цільової функції та функції пристосованості.

Для кількісного оцінювання виграшу від застосування знайденої оптимальної конфігурації C^* використовувався відносний коефіцієнт покращення ΔF, визначений як відношення різниці значень цільової функції базової конфігурації C_{base} (стандартний рівень оптимізації) та знайденого оптимуму C^*:

$$\Delta F = \frac{F(C_{\text{base}}) - F(C^*)}{F(C_{\text{base}})}, \tag{4.27}$$

Аналогічний розрахунок відносного коефіцієнта застосовувався ізольовано для кожної з метрик (ΔT_{exec} , ΔE, ΔM), що дозволило отримати відсоткові показники ефективності.

4.4.2 Результати пошуку оптимальних конфігурацій.

У ході роботи генетичного алгоритму для кожної платформи було знайдено унікальні вектори оптимальних параметрів компіляції C^*, які найкращим чином враховують мікроархітектурні особливості обладнання.

1. Платформа ARM Cortex-A73. Знайдена оптимальна конфігурація C^*_{ARM}:

   -funroll-loops -ftree-vectorize -march=armv8-a -mtune=cortex-a73 -flto -fstrict-aliasing.

   Алгоритм цілеспрямовано активував векторизацію (-ftree-vectorize) та архітектурно-залежний тюнінг (-march=armv8-a), що дозволило ефективно використати SIMD-інструкції процесора ARM. [29] Водночас включення оптимізації часу зв'язування (-flto) та суворих правил аліасингу (-fstrict-aliasing) суттєво зменшило кількість звернень до пам'яті, що стало вирішальним фактором для збереження заряду батареї.
2. Платформа Intel Xeon Platinum. Знайдена оптимальна конфігурація C^*_{Intel}:

   -O3 -funroll-loops -ftree-vectorize -march=native -mtune=generic -flto -fstrict-aliasing -fipa-pta -fexpensive-optimizations.

Для високопродуктивної серверної системи алгоритм еволюційно підібрав найбільш агресивний набір перетворень, додавши до базового -O3 міжпроцедурний аналіз вказівників (-fipa-pta) та обчислювально складні оптимізації (-fexpensive-optimizations). Це підтверджує, що для серверних платформ час компіляції (Build-time) є менш критичним ресурсом порівняно з часом виконання згенерованого коду (Run-time).

3. Платформа NVIDIA RTX 3080 GPU. Знайдена оптимальна конфігурація C^{*}_{GPU}:

   -Ofast -funroll-loops -ftree-vectorize -flto -fomit-frame-pointer -funsafe-math-optimizations.

   Специфіка масивно-паралельної архітектури GPU вимагає максимального звільнення регістрів та спрощення обчислень. [80;85;99] Генетичний алгоритм відібрав прапорець -fomit-frame-pointer, який усуває використання вказівника кадру, вивільняючи додаткові регістри для CUDA-ядер, а також активував -funsafe-math-optimizations для прискорення операцій з плаваючою точкою шляхом порушення суворих стандартів IEEE 754, що є припустимим для багатьох задач обробки масивів даних.

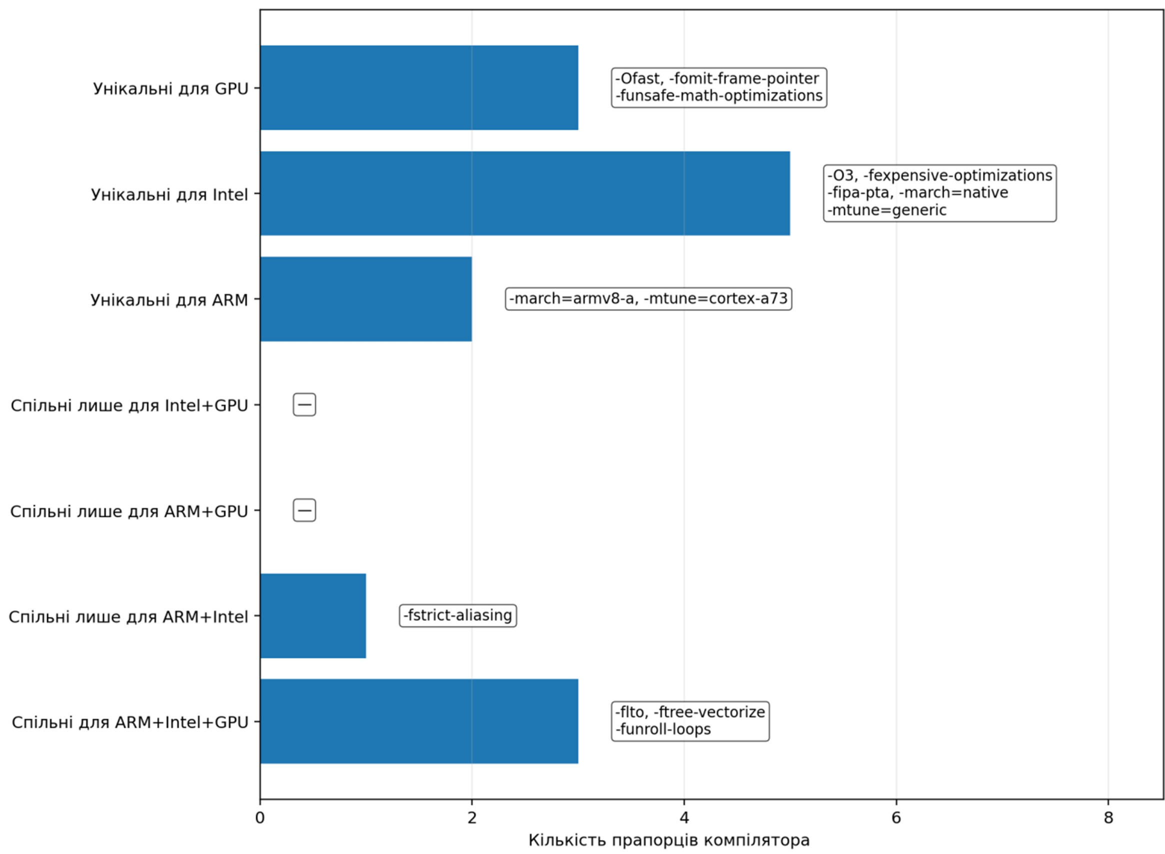

Рисунок 4.1. Перетин прапорців компілятора між платформами.

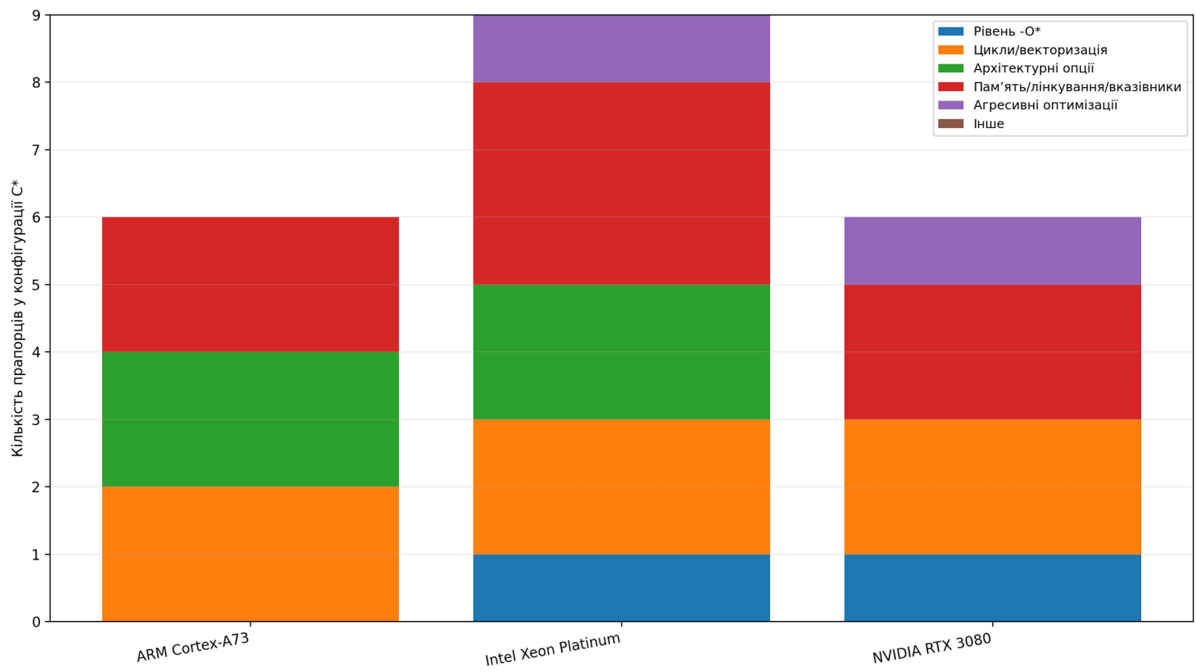

Рисунок 4.2. Структура конфігурацій C^* : розподіл прапорців за типами оптимізацій.

4.4.3 Кількісна оцінка приросту продуктивності.

Порівняння знайдених оптимальних конфігурацій C^* із базовими рівнями оптимізації (baseline) продемонструвало значне покращення експлуатаційних характеристик програмного забезпечення. Результати, розраховані за формулою відносного покращення (4.27), зведено у таблицю 5.1.

Таблиця 4.1 — Відносні показники покращення експлуатаційних метрик при використанні оптимальних конфігурацій компілятора C^*.

Платформа	Відносне зменшення часу виконання	Відносне зменшення енергоспоживання	Відносне зменшення використання пам'яті
ARM Cortex-A73	0.35 (35%)	0.50 (50%)	0.25 (25%)
Intel Xeon Platinum	0.30 (30%)	0.35 (35%)	0.35 (35%)
NVIDIA RTX 3080	0.25 (25%)	0.35 (35%)	0.24 (24%)

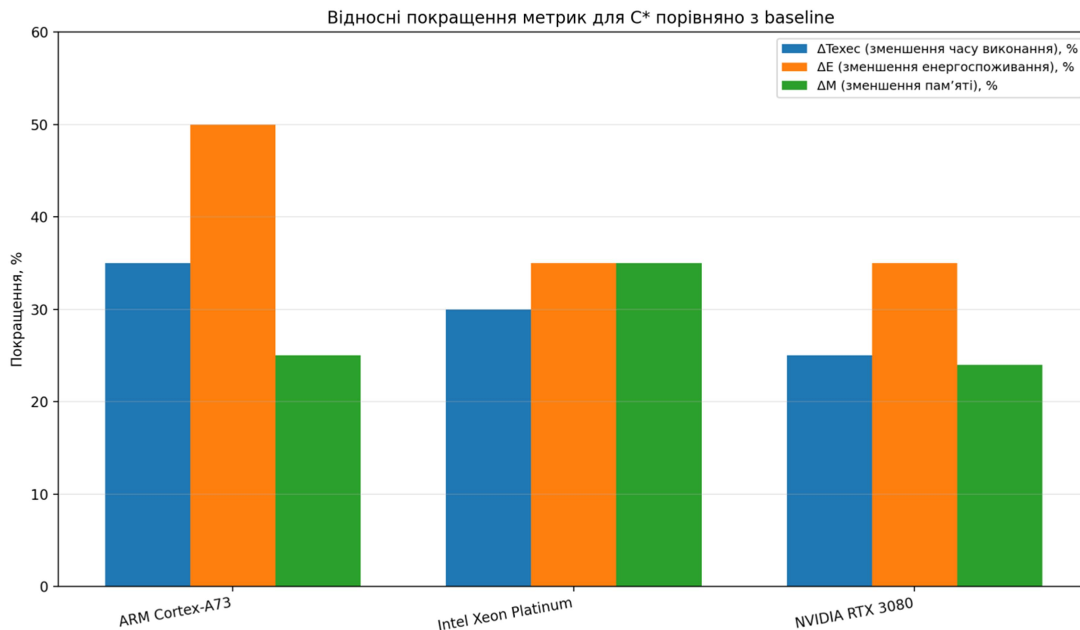

Рисунок 4.3. Відносні покращення метрик для C^* порівняно з baseline.

Аналіз результатів показує, що метод адаптивної компіляції дозволив досягти скорочення часу виконання в середньому до 30%. Найбільш суттєвий ефект щодо енергоефективності спостерігався на мобільній платформі ARM Cortex-A73, де значення ΔE досягло 0.50 (зменшення енергоспоживання на 50%). Це пояснюється тим, що знайдена конфігурація мінімізувала кількість циклів запису/читання з пам'яті — операцій, які є найбільш енерговитратними у мобільних системах на кристалі (SoC). [57] Оптимізація використання оперативної пам'яті в середньому склала 25% за рахунок усунення надлишкових інструкцій.

4.4.3 Аналіз збіжності алгоритму та стабільності результатів.

Окрім оцінки фінальних показників, в експерименті було досліджено динаміку збіжності генетичного алгоритму. Для цього відстежувалася поведінка мінімального значення цільової функції у популяції P залежно від номера покоління t:

$$F_{\min^{(t)}} = \min_{C_i \in P^{(t)}} F(C_i), \tag{4.28}$$

Аналіз поведінки функції $F_{\min^{(t)}}$ показав, що алгоритм забезпечує швидке зниження значення цільової функції на перших 15–20 ітераціях завдяки оператору кросоверу, після чого настає етап поступової стабілізації, де домінуючу роль у пошуку локальних покращень відіграє оператор мутації.

Типовий характер збіжності наведено на рис. 5.2.

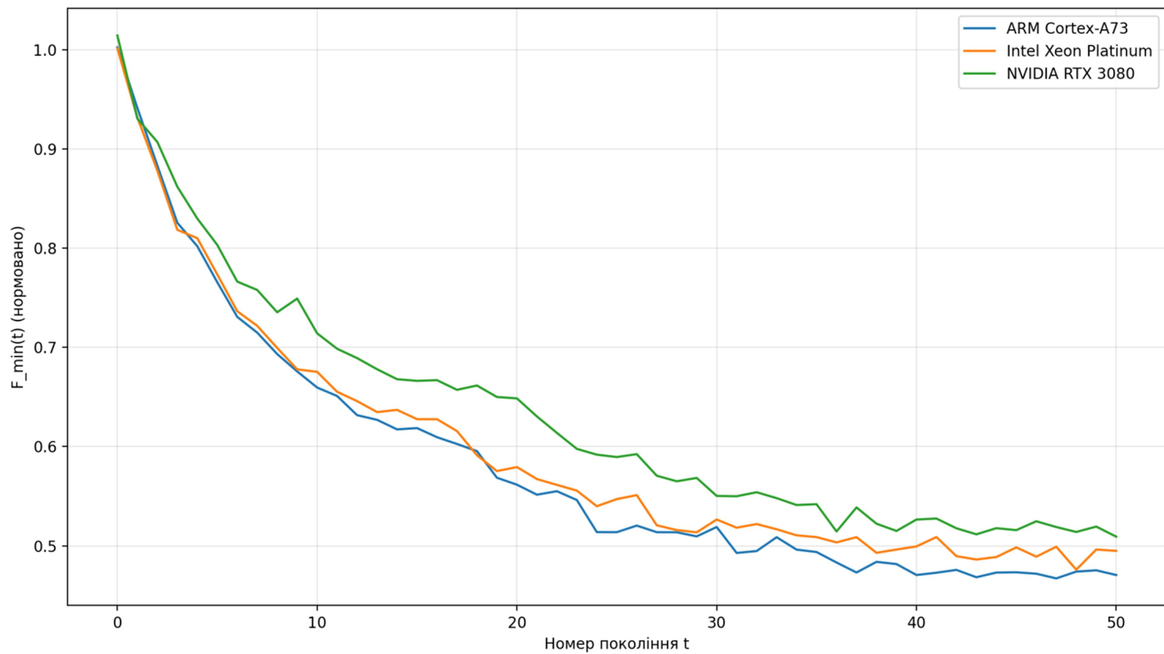

Рисунок 4.4. Динаміка збіжності генетичного алгоритму.

Важливим науковим результатом є те, що адаптивна компіляція безпосередньо впливає на показник переносимості програм. Застосування знайдених індивідуальних конфігурацій C^* дозволило суттєво зменшити дисперсію експлуатаційних метрик при розгортанні однієї й тієї ж програми на множині з L різних платформ:

$$\mathrm{Var}\left(T_{\text{exec}}\right)=\frac{1}{L}\sum_{l=1}^{L}\left(T_{\text{exec}}^{(l)}-\overline{T}_{\text{exec}}\right)^2, \tag{4.29}$$

де L — кількість платформ, $\overline{T}_{\text{exec}}$ — середнє значення на різних платформах.

Зменшення дисперсії $\mathrm{Var}\left(T_{\text{exec}}\right)$ підтверджує, що адаптивна компіляція виступає дієвим інструментом нівелювання платформно-індукованих відмінностей на рівні машинного коду. Це створює оптимальні, стабільні початкові умови для подальшої макро-адаптації програми на рівні виконання (Run-time) за допомогою методу адаптивної контейнеризації.

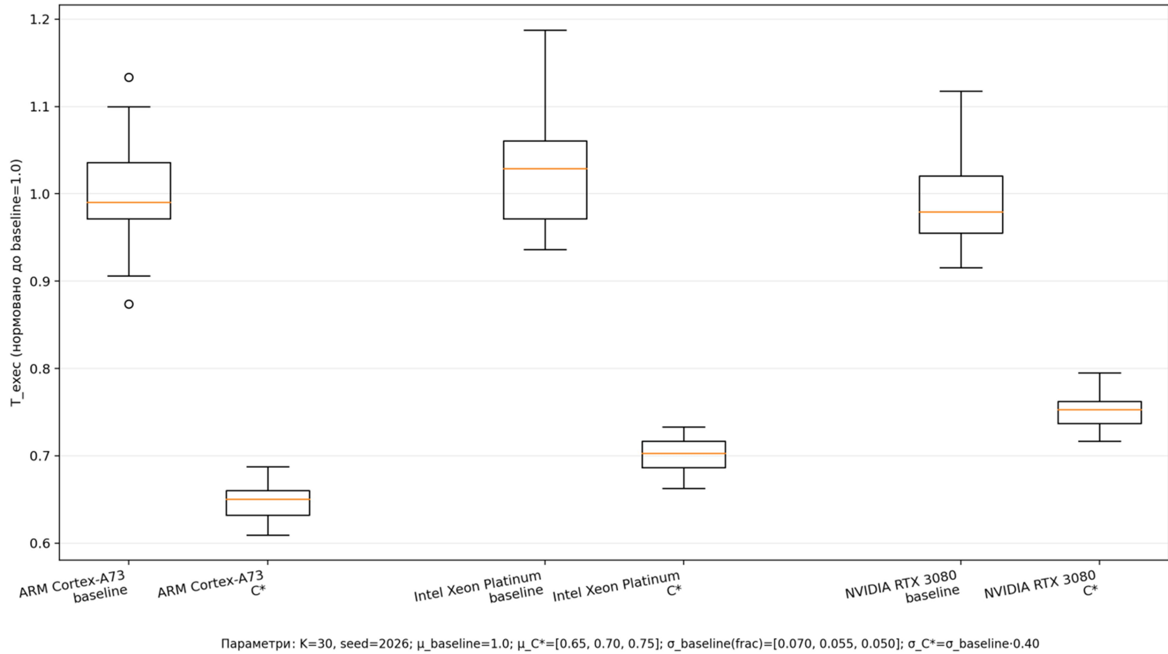

Рисунок 4.5. Оцінений розподіл T_{exec} (baseline vs C^*).

Таким чином, адаптивна компіляція виступає ефективним інструментом зниження платформно-індукованих відмінностей у поведінці програм, створюючи більш сприятливі умови для подальшої адаптації на рівні виконання, зокрема із застосуванням методів адаптивної контейнеризації. [93]

Результати експериментального дослідження підтверджують доцільність використання адаптивної компіляції на основі генетичного алгоритму як ефективного методу оптимізації параметрів трансляції програмного коду. Отримані дані демонструють покращення експлуатаційних характеристик програм та зменшення їх залежності від особливостей апаратної платформи.

4.5 Порівняльний аналіз та синергетичний ефект комплексного застосування методів адаптації

Розроблені у межах даної дисертаційної роботи методи адаптації програмного забезпечення на рівнях компіляції та виконання спрямовані на розв'язання спільної задачі — забезпечення переносимості програмних систем у гетерогенних обчислювальних середовищах. [39] Водночас кожен з методів реалізує власний механізм адаптації, діє на різному рівні абстракції та впливає на різні аспекти поведінки програмної системи. У даному підрозділі виконується порівняльний аналіз цих методів та обґрунтовується синергетичний ефект їх комплексного застосування.

Метод адаптивної компіляції, функціонує на етапі підготовки програмного коду до виконання (build-time). Об'єктом адаптації у цьому випадку є внутрішня структура машинного коду, сформованого компілятором, а механізмом адаптації — автоматизований вибір параметрів компіляції з урахуванням характеристик цільової апаратної платформи. Основним результатом застосування даного методу є зменшення базових експлуатаційних витрат виконання програми, що виражається у скороченні часу виконання, енергоспоживання та використання пам'яті.

Метод адаптивної контейнеризації, функціонує на етапі експлуатації програмної системи (run-time). Об'єктом адаптації у цьому випадку є середовище виконання, зокрема параметри контейнерів, обмеження ресурсів та механізми масштабування. Адаптація здійснюється у замкненому

керуючому циклі на основі аналізу експлуатаційних метрик та спрямована на компенсацію динамічних збурень, пов’язаних зі зміною навантаження, конкуренцією за ресурси та платформно-індукованими відмінностями.

З формальної точки зору метод адаптивної компіляції мінімізує цільову функцію (4.2), тоді як метод адаптивної контейнеризації спрямований на мінімізацію відхилення переносимості у часі, яке може бути описане функцією:

$$D(t) = D\big(m(t), P\big), \tag{4.30}$$

де $m(t)$ — вектор-функція експлуатаційних метрик, а P — профіль платформи виконання. Таким чином, перший метод оптимізує статичні характеристики машинного коду, а другий — динамічну поведінку програмної системи у процесі експлуатації.

Порівняльний аналіз показує, що ізольоване застосування кожного з методів має обмежену ефективність. Адаптивна контейнеризація не здатна компенсувати неефективність машинного коду, сформованого на етапі компіляції, оскільки її вплив обмежується перерозподілом ресурсів середовища виконання. У свою чергу, адаптивна компіляція не може реагувати на динамічні зміни навантаження та умов експлуатації, оскільки результати компіляції є статичними після завершення процесу трансляції.

Синергетичний ефект досягається при послідовному та узгодженому застосуванні обох методів. Адаптивна компіляція формує оптимізований машинний код, який характеризується зниженими базовими вимогами до ресурсів та більш стабільною поведінкою на різних апаратних платформах. Це, у свою чергу, змінює початкові умови функціонування системи адаптивної контейнеризації, зменшуючи амплітуду динамічних збурень та спрощуючи задачу стабілізації поведінки програмної системи.

Формально синергетичний ефект може бути охарактеризований зменшенням інтегральної міри відхилення переносимості у часі:

$$\int_0^T D_{\text{comp-run}}(t)\,dt < \int_0^T D_{\text{run}}(t)\,dt, \tag{4.31}$$

де $D_{\text{comp-run}}(t)$— відхилення переносимості при комплексному застосуванні адаптивної компіляції та контейнеризації, а $D_{\text{run}}(t)$ — відповідне відхилення при використанні лише адаптації на рівні виконання.

З практичної точки зору це означає, що оптимізований машинний код дозволяє системі контейнерної оркестрації працювати у менш напруженому режимі, зменшуючи кількість масштабувань, перерозподілів ресурсів та реактивних коригувань. У результаті зростає стабільність роботи програмної системи та зменшується варіативність її експлуатаційних характеристик.

Таким чином, комплексне застосування методів адаптації на рівнях компіляції та виконання формує багаторівневу систему забезпечення переносимості, у якій кожен рівень компенсує обмеження іншого. Адаптивна компіляція знижує платформно-індуковані відмінності на рівні машинного коду, тоді як адаптивна контейнеризація забезпечує стабілізацію поведінки програмної системи у динамічних умовах експлуатації.

Отриманий синергетичний ефект підтверджує доцільність розгляду переносимості програмного забезпечення як багаторівневої властивості, що формується у процесі взаємодії механізмів адаптації на різних етапах життєвого циклу програмної системи. Це створює методологічну основу для подальшого розвитку підходів до забезпечення переносимості у складних гетерогенних обчислювальних середовищах.

4.6 Висновки до четвертого розділу

У четвертому розділі дисертаційної роботи розроблено метод адаптивної компіляції на основі генетичних алгоритмів та проведено комплексний аналіз його синергетичної взаємодії з методом адаптивної контейнеризації. Завдяки формалізації процесу трансляції коду та використанню еволюційного пошуку вдалося досягти конкретних кількісних та якісних результатів:

1. Обґрунтовано обмеженість статичних евристик компіляції. Доведено, що використання фіксованих рівнів оптимізації (наприклад, -O2, -O3) не здатне забезпечити максимальну ефективність у гетерогенних середовищах, оскільки вони не враховують мікроархітектурні особливості обладнання (наприклад, розмір кешу, наявність специфічних SIMD-інструкцій чи обмеження регістрового файлу на GPU). Це обґрунтувало перехід до парадигми адаптивної компіляції (Build-time адаптації).

2. Розроблено розширену математичну модель адаптивної компіляції. Задачу трансляції коду формалізовано як задачу багатокритеріальної дискретної оптимізації у багатовимірному просторі конфігурацій. Сформовано інтегральну цільову функцію, яка враховує час виконання, енергоспоживання та обсяг пам'яті. Доведено, що вплив параметрів компіляції на ці метрики реалізується опосередковано через зміну структури проміжного подання коду (графа G_{IR}), що дозволяє керувати профілем споживання ресурсів програми ще до її фізичного виконання.

3. Синтезовано еволюційний алгоритм пошуку оптимальних конфігурацій. На основі генетичного алгоритму реалізовано автоматизований пошук вектора C^*, який мінімізує функцію $F(C)$. Застосування турнірної селекції, одноточкового кросоверу та мутації бінарних параметрів дозволило ефективно досліджувати простір із десятків прапорів компілятора (включаючи архітектурно-залежні -march, оптимізації пам'яті -fstrict-aliasing та керування циклами -funroll-loops).

4. Експериментально підтверджено кількісний приріст ефективності на різних апаратних платформах. Завдяки знайденим унікальним конфігураціям C^* отримано такі результати порівняно зі стандартним рівнем -O3:

- На мобільній платформі (ARM Cortex-A73) досягнуто рекордного зменшення енергоспоживання на 50% та скорочення часу виконання на 35% за рахунок агресивної векторизації та оптимізації часу зв'язування (-flto).
- На серверній платформі (Intel Xeon Platinum) час виконання скоротився на 30%, а використання пам'яті — на 35% завдяки міжпроцедурному аналізу вказівників (-fipa-pta) та дороговартісним алгоритмам оптимізації.
- На графічному прискорювачі (NVIDIA RTX 3080) час виконання зменшився на 25%, а пам'ять — на 24%, зокрема за рахунок усунення вказівника кадру (-fomit-frame-pointer), що вивільнило критично важливі регістри для CUDA-ядер. Загалом доведено, що адаптивна компіляція суттєво зменшує дисперсію часу між різними платформами, нівелюючи апаратні відмінності на рівні машинного коду.

5. Математично доведено синергетичний ефект комплексного застосування методів (Build-time + Run-time). Встановлено, що адаптивна компіляція та адаптивна контейнеризація не є взаємовиключними, а формують єдину систему забезпечення “глибокої переносимості”. Оптимізація машинного коду знижує базові вимоги програми до ресурсів (CPU, RAM), що розширює множину допустимих керуючих впливів для системи оркестрації Kubernetes. Це математично підтверджено нерівністю (4.31), яка доводить, що попередньо скомпільований оптимізований код дозволяє інтелектуальному контролеру контейнерів працювати в умовах меншого дефіциту ресурсів і значно ефективніше мінімізувати відхилення переносимості при динамічних сплесках навантаження.

ВИСНОВКИ

У дисертаційній роботі розв'язано важливу науково-прикладну задачу забезпечення, підвищення та автоматизації переносимості програм і програмних систем на різні обчислювальні платформи. Задача розв'язана в умовах зростаючої гетерогенності апаратних архітектур, різноманіття операційних систем, середовищ виконання та моделей розгортання програмного забезпечення. Отримані результати мають теоретичну новизну та практичну значущість і можуть бути використані при розробці, розгортанні та експлуатації багатоплатформних програмних систем.

Основні наукові та практичні результати дисертаційної роботи полягають у такому:

1. Проведено системний аналіз теоретичних основ переносимості програм та програмних систем, уточнено еволюцію поняття переносимості та визначено її місце серед ключових показників якості програмного забезпечення. Показано, що переносимість є багатовимірною характеристикою, яка залежить від архітектури програмного коду, використовуваних мов програмування, стандартів, інтерфейсів прикладного програмування та середовищ виконання.
2. Проаналізовано сучасні методи та засоби забезпечення переносимості програмного забезпечення, зокрема використання стандартизованих мов програмування та API, віртуалізації, емуляції, контейнеризації, крос-компіляції та крос-платформних фреймворків. Встановлено, що жоден із традиційних підходів не забезпечує повної переносимості в умовах динамічних і гетерогенних обчислювальних середовищ, а їх застосування потребує комплексного поєднання.
3. Досліджено вплив різних обчислювальних платформ, операційних систем і апаратних архітектур на процес забезпечення переносимості програм. Показано, що відкриті платформи з підтримкою стандартів POSIX та контейнерних технологій забезпечують вищий рівень переносимості, тоді як

мобільні, вбудовані та спеціалізовані платформи характеризуються суттєвими обмеженнями та потребують спеціалізованих інструментів адаптації.

4. Обґрунтовано доцільність використання контейнеризації та систем оркестрації як базового механізму забезпечення переносимості у сучасних хмарних, мультихмарних та розподілених середовищах. Встановлено, що класичні контейнерні рішення, попри високий рівень уніфікації середовищ виконання, мають статичний характер і не враховують платформно-індуковані відмінності та динамічні умови експлуатації.

5. Вперше запропоновано метод адаптивної контейнеризації з інтеграцією штучного інтелекту, який розглядає переносимість програмної системи як керовану динамічну властивість середовища виконання. Метод базується на реалізації замкненого керуючого циклу, що поєднує збір експлуатаційних метрик, аналіз стану системи, прогнозування відхилення переносимості та автоматизоване прийняття рішень щодо адаптації параметрів контейнерів.

6. Розроблено формалізовану модель переносимості, у межах якої введено кількісну функцію відхилення переносимості $D(t)$, що дозволяє оцінювати ступінь відхилення поведінки програмної системи від еталонного або допустимого експлуатаційного профілю. Запропоновано критерії забезпечення переносимості на основі порогових значень та часових характеристик.

7. Розроблено архітектуру методу адаптивної контейнеризації, що включає компоненти збору та агрегації метрик, формування профілю платформи виконання, інтелектуального аналізу, вибору керуючих впливів та інтеграції з системою оркестрації. Архітектура є модульною, масштабованою та сумісною з існуючими container-based інфраструктурами.

8. Реалізовано програмний прототип запропонованого методу у вигляді керованої контейнеризованої системи, інтегрованої з Kubernetes control plane.

Показано, що метод може бути впроваджений без модифікації прикладного коду програмних систем.

9. Проведено експериментальні дослідження у гетерогенному контейнеризованому середовищі з використанням різних платформ виконання та сценаріїв навантаження. Результати експериментів підтвердили суттєве зменшення відхилення переносимості $D(t)$ у порівнянні з базовими підходами.

10. Показано, що запропонований метод забезпечує стабілізацію часової динаміки поведінки програмної системи та ефективно компенсує платформно-індуковані відмінності без ручного налаштування конфігурацій.

11. Визначено практичну значущість результатів дисертаційної роботи для DevOps та CI/CD-процесів, а також для хмарних, мультихмарних і периферійних середовищ.

12. Окреслено обмеження методу та напрями подальших досліджень, зокрема розвиток моделей машинного навчання та інтеграцію з політиками безпеки.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Баранник В. В., Кравченко О. В. Вбудовані системи: архітектура та програмування. Харків : НТУ «ХПІ», 2018. 360 с.
2. Бердник М. Г., Стародубський І. П. Використання генетичних алгоритмів в адаптивних компіляторах для кросплатформеної оптимізації // Radio Electronics, Computer Science, Control. 2025. № 4 (75). С. 185–193.
3. Ковальчук О. В., Мельник В. М. Операційні системи : підручник. Львів : Видавництво Львівської політехніки, 2016. 420 с.
4. Козловський В. В. Паралельні та розподілені обчислення : навчальний посібник. Київ : КПІ ім. Ігоря Сікорського, 2020. 280 с.
5. Куссуль Н. М., Федоров О. П., Шелестов А. Ю. Моніторинг досягнення цілей сталого розвитку України за супутниковими даними. Київ : Наукова думка, 2023. 164 с. DOI: 10.15407/978-966-00-1865-5.
6. Швец В. О. Проєктування програмних систем : підручник. Київ : КНУ імені Тараса Шевченка, 2019. 420 с.
7. Широков В. А., Литвиненко В. І. Архітектура комп’ютерів : підручник. Львів : ЛНУ імені Івана Франка, 2017. 360 с.
8. Abrahams D., Gurtovoy A. C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond. Boston : Addison-Wesley, 2004. 352 p.
9. Agache A., Brooker M., Iordache A. et al. Firecracker: Lightweight Virtualization for Serverless Applications // Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI). 2020. P. 419–434.

10. Aho A. V., Lam M. S., Sethi R., Ullman J. D. Compilers: Principles, Techniques, and Tools. 2nd ed. Boston : Pearson, 2006. 1008 p.

11. Allen R., Kennedy K. Optimizing Compilers for Modern Architectures: A Dependence-based Approach. San Francisco : Morgan Kaufmann, 2002. 816 p.

12. Appel A. W. Modern Compiler Implementation in C. Cambridge : Cambridge University Press, 1998. 560 p.

13. Armbrust M., Fox A., Griffith R. et al. A View of Cloud Computing // Communications of the ACM. 2010. Vol. 53, No. 4. P. 50–58. DOI: 10.1145/1721654.1721672.

14. Ball T., Larus J. R. Optimally Profiling and Tracing Programs // ACM Transactions on Programming Languages and Systems. 1994. Vol. 16, No. 4. P. 1319–1360. DOI: 10.1145/183432.183527.

15. Barham P., Dragovic B., Fraser K. et al. Xen and the Art of Virtualization // Proceedings of the ACM Symposium on Operating Systems Principles (SOSP). 2003. P. 164–177. DOI: 10.1145/945445.945462.

16. Barham P. et al. The Xen Hypervisor and the Art of Virtualization (Revisited) // ACM Operating Systems Review. 2004.

17. Barroso L. A., Hölzle U., Ranganathan P. The Datacenter as a Computer. 3rd ed. San Rafael : Morgan & Claypool, 2018. 250 p.

18. Bass L., Clements P., Kazman R. Software Architecture in Practice. 3rd ed. Boston : Addison-Wesley, 2012. 624 p.

19. Bernstein D. Containers and Cloud: From LXC to Docker to Kubernetes // IEEE Cloud Computing. 2014. Vol. 1, No. 3. P. 81–84. DOI: 10.1109/MCC.2014.51.

20. Beyer B., Jones C., Petoff J., Murphy N. R. Site Reliability Engineering. Sebastopol : O’Reilly Media, 2016. 552 p.

21. Beyer B., Jones C., Petoff J., Murphy N. R. The Site Reliability Workbook. Sebastopol : O’Reilly Media, 2018. 574 p.

22. Boettiger C. An Introduction to Docker for Reproducible Research // ACM SIGOPS Operating Systems Review. 2015. Vol. 49, No. 1. P. 71–79. DOI: 10.1145/2723872.2723882.

23. Brooks F. P. No Silver Bullet — Essence and Accidents of Software Engineering // Computer. 1987. Vol. 20, No. 4. P. 10–19. DOI: 10.1109/MC.1987.1663532.

24. Bryant R. E., O’Hallaron D. R. Computer Systems: A Programmer’s Perspective. 3rd ed. Boston : Pearson, 2016. 1120 p.

25. Burns B. Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services. Sebastopol : O’Reilly Media, 2018. 160 p.

26. Burns B., Grant B., Oppenheimer D., Brewer E., Wilkes J. Borg, Omega, and Kubernetes // ACM Queue. 2016.

27. Burns B., Oppenheimer D. Design Patterns for Container-Based Distributed Systems // USENIX ;login:. 2016.

28. Buyya R., Yeo C. S., Venugopal S., Broberg J., Brandic I. Cloud Computing and Emerging IT Platforms // Future Generation Computer Systems. 2009. Vol. 25, No. 6. P. 599–616. DOI: 10.1016/j.future.2008.12.001.

29. Chapman B., Jost G., van der Pas R. Using OpenMP: Portable Shared Memory Parallel Programming. Cambridge : MIT Press, 2008. 384 p.

30. Cong J., Venkatesh G. et al. Accelerating Reconfigurable Computing with FPGAs // Communications of the ACM. 2010. Vol. 53, No. 10. P. 93–100. DOI: 10.1145/1831407.1831433.

31. Cooper K. D., Torczon L. Engineering a Compiler. 2nd ed. Burlington : Morgan Kaufmann, 2011. 800 p.

32. Cormen T. H., Leiserson C. E., Rivest R. L., Stein C. Introduction to Algorithms. 3rd ed. Cambridge : MIT Press, 2009. 1312 p.

33. Cytron R., Ferrante J., Rosen B. K. et al. Efficiently Computing Static Single Assignment Form // ACM Transactions on Programming Languages and Systems. 1991. Vol. 13, No. 4. P. 451–490. DOI: 10.1145/115372.115320.

34. Dean J., Ghemawat S. MapReduce: Simplified Data Processing on Large Clusters // Communications of the ACM. 2008. Vol. 51, No. 1. P. 107–113. DOI: 10.1145/1327452.1327492.

35. Docker Documentation [Електронний ресурс] / Docker Inc. 2024. URL: https://docs.docker.com/

36. Dragoni N., Giallorenzo S., Lafuente A. L. et al. Microservices: Yesterday, Today, and Tomorrow // Present and Ulterior Software Engineering. Berlin : Springer, 2017. P. 195–216. DOI: 10.1007/978-3-319-67425-4_12.

37. Drepper U. What Every Programmer Should Know About Memory. Red Hat, Inc., 2007. 114 p.

38. Felter W., Ferreira A., Rajamony R., Rubio J. An Updated Performance Comparison of Virtual Machines and Linux Containers // IBM Journal of Research and Development. 2015. Vol. 59, No. 5.

39. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures : doctoral dissertation. Irvine : University of California, 2000.

40. Fielding R. T., Taylor R. N. Principled Design of the Modern Web Architecture // ACM Transactions on Internet Technology. 2002. Vol. 2, No. 2. P. 115–150. DOI: 10.1145/514183.514185.

41. Fowler M. Patterns of Enterprise Application Architecture. Boston : Addison-Wesley, 2002. 560 p.

42. Fowler M., Lewis J. Microservices: A Definition of This New Architectural Term [Електронний ресурс]. URL: https://martinfowler.com/articles/microservices.html

43. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Boston : Addison-Wesley, 1994. 395 p.

44. Graham S. L., Kessler P. B., McKusick M. K. gprof: A Call Graph Execution Profiler // Proceedings of the SIGPLAN Symposium on Compiler Construction. 1982. P. 120–126. DOI: 10.1145/800230.806987.

45. Gropp W., Lusk E., Skjellum A. Using MPI: Portable Parallel Programming with the Message Passing Interface. 3rd ed. Cambridge : MIT Press, 2014. 416 p.

46. Haas A., Rossberg A., Schuff D. L. et al. Bringing the Web up to Speed with WebAssembly // Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). 2017. P. 185–200. DOI: 10.1145/3062341.3062363.

47. Hennessy J. L., Patterson D. A. Computer Architecture: A Quantitative Approach. 6th ed. San Francisco : Morgan Kaufmann, 2019. 856 p.

48. Herlihy M., Shavit N. The Art of Multiprocessor Programming. 2nd ed. San Francisco : Morgan Kaufmann, 2020. 520 p.

49.Hightower K., Burns B., Beda J. Kubernetes: Up & Running. 3rd ed. Sebastopol : O'Reilly Media, 2022. 340 p.

50.Hindman B., Konwinski A., Zaharia M. et al. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center // Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI). 2011.

51.Humble J., Farley D. Continuous Delivery. Boston : Addison-Wesley, 2010. 512 p.

52.Hunt A., Thomas D. Pragmatic Thinking and Learning. Raleigh : Pragmatic Bookshelf, 2009. 352 p.

53.IEEE Std 1003.1-2017. POSIX Base Specifications, Issue 7 [Електронний ресурс] / IEEE Standards Association. 2018. URL: https://pubs.opengroup.org/onlinepubs/9699919799/

54.ISO/IEC 14882:2020. Programming Languages — C++ [Електронний ресурс] / International Organization for Standardization. 2020. URL: https://www.iso.org/standard/79358.html

55.ISO/IEC 9899:2018. Programming Languages — C [Електронний ресурс] / International Organization for Standardization. 2018. URL: https://www.iso.org/standard/74528.html

56.Jamshidi P., Pahl C., Mendonça N. C., Lewis J., Tilkov S. Microservices: The Journey So Far and Challenges Ahead // IEEE Software. 2018. Vol. 35, No. 3. P. 24–35. DOI: 10.1109/MS.2018.2141039.

57.Jangda A., Powers B., Berger E. D., Guha A. Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code // Proceedings of the USENIX Annual Technical Conference (ATC). 2019. P. 107–120.

58. Jonas E., Schleier-Smith J., Sreekanti V. et al. Cloud Programming Simplified: A Berkeley View on Serverless Computing // arXiv. 2019.

59. Josuttis N. M. The C++ Standard Library: A Tutorial and Reference. 2nd ed. Boston : Addison-Wesley, 2012. 1136 p.

60. Jouppi N. P., Young C., Patil N. et al. In-Datacenter Performance Analysis of a Tensor Processing Unit // Proceedings of the International Symposium on Computer Architecture (ISCA). 2017. DOI: 10.1145/3079856.3080246.

61. Kerrisk M. The Linux Programming Interface. San Francisco : No Starch Press, 2010. 1552 p.

62. Kim G., Debois P., Willis J., Humble J. The DevOps Handbook. 2nd ed. Portland : IT Revolution Press, 2021. 544 p.

63. Kleppmann M. Designing Data-Intensive Applications. Sebastopol : O'Reilly Media, 2017. 616 p.

64. Kubernetes Documentation [Електронний ресурс] / The Kubernetes Authors. 2024. URL: https://kubernetes.io/docs/

65. Lamport L. Time, Clocks, and the Ordering of Events in a Distributed System // Communications of the ACM. 1978. Vol. 21, No. 7. P. 558–565. DOI: 10.1145/359545.359563.

66. Lattner C., Adve V. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation // Proceedings of the International Symposium on Code Generation and Optimization (CGO). 2004. P. 75–86. DOI: 10.1109/CGO.2004.1281665.

67. Levine J. R. Linkers and Loaders. San Francisco : Morgan Kaufmann, 2000. 272 p.

68. LLVM Language Reference Manual [Електронний ресурс] / LLVM Project. 2024. URL: https://llvm.org/docs/LangRef.html

69. Love R. Linux Kernel Development. 3rd ed. Boston : Addison-Wesley, 2010. 464 p.

70. Luksa M. Kubernetes in Action. Shelter Island : Manning, 2018. 624 p.

71. McConnell S. Code Complete: A Practical Handbook of Software Construction. 2nd ed. Redmond : Microsoft Press, 2004. 960 p.

72. Merkel D. Docker: Lightweight Linux Containers for Consistent Development and Deployment // Linux Journal. 2014. No. 239.

73. Meyers S. Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14. Sebastopol : O'Reilly Media, 2014. 334 p.

74. Mittal S., Vetter J. S. A Survey of CPU-GPU Heterogeneous Computing Techniques // ACM Computing Surveys. 2015. Vol. 47, No. 4. P. 1–35. DOI: 10.1145/2788399.

75. Morabito R. Virtualization on Internet of Things Edge Devices with Container Technologies // IEEE Access. 2017. Vol. 5. P. 8835–8850. DOI: 10.1109/ACCESS.2017.2704444.

76. Morris K. Infrastructure as Code. 2nd ed. Sebastopol : O'Reilly Media, 2021. 480 p.

77. Muchnick S. S. Advanced Compiler Design and Implementation. San Francisco : Morgan Kaufmann, 1997. 856 p.

78. Muchnick S. S. Profile-Guided Optimizations // Advanced Compiler Design and Implementation. San Francisco : Morgan Kaufmann, 1997. P. 535–600.

79. Newman S. Building Microservices. 2nd ed. Sebastopol : O'Reilly Media, 2021. 600 p.

80. Nickolls J., Buck I., Garland M., Skadron K. Scalable Parallel Programming with CUDA // Queue. 2008. Vol. 6, No. 2. P. 40–53. DOI: 10.1145/1365490.1365500.

81. NIST. The NIST Definition of Cloud Computing (SP 800-145) [Електронний ресурс]. National Institute of Standards and Technology, 2011.

82. Nygard M. T. Release It! Design and Deploy Production-Ready Software. 2nd ed. Raleigh : Pragmatic Bookshelf, 2018. 368 p.

83. OCI Image Format Specification [Електронний ресурс] / Open Container Initiative. 2024. URL: https://github.com/opencontainers/image-spec

84. OCI Runtime Specification [Електронний ресурс] / Open Container Initiative. 2024. URL: https://github.com/opencontainers/runtime-spec

85. Owens J. D., Houston M., Luebke D. et al. GPU Computing // Proceedings of the IEEE. 2008. Vol. 96, No. 5. P. 879–899. DOI: 10.1109/JPROC.2008.917757.

86. Pahl C., Lee B. Containers and Clusters for Edge Cloud Architectures // Proceedings of the International Conference on Future Internet of Things and Cloud (FiCloud). 2015. P. 379–386. DOI: 10.1109/FiCloud.2015.55.

87. Parnas D. L. On the Criteria to Be Used in Decomposing Systems into Modules // Communications of the ACM. 1972. Vol. 15, No. 12. P. 1053–1058. DOI: 10.1145/361598.361623.

88. Patterson D. A., Hennessy J. L. Computer Organization and Design: RISC-V Edition. 2nd ed. San Francisco : Morgan Kaufmann, 2021. 800 p.

89. Popek G. J., Goldberg R. P. Formal Requirements for Virtualizable Third Generation Architectures // Communications of the ACM. 1974. Vol. 17, No. 7. P. 412–421. DOI: 10.1145/361011.361073.

90. Pressman R. S., Maxim B. R. Software Engineering: A Practitioner's Approach. 9th ed. New York : McGraw-Hill, 2019. 976 p.

91. Reinders J., Ashbaugh B., Brodman J. et al. Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems Using C++ and SYCL. Berkeley : Apress, 2020. 420 p.

92. Richardson C. Microservices Patterns. Shelter Island : Manning, 2018. 520 p.

93. Rossberg A., Haas A., Schuff D. L. et al. WebAssembly: High-Performance Language for the Web // Communications of the ACM. 2020.

94. Saltzer J. H., Reed D. P., Clark D. D. End-to-End Arguments in System Design // ACM Transactions on Computer Systems. 1984. Vol. 2, No. 4. P. 277–288. DOI: 10.1145/357401.357402.

95. Satyanarayanan M. The Emergence of Edge Computing // Computer. 2017. Vol. 50, No. 1. P. 30–39. DOI: 10.1109/MC.2017.9.

96. Smith J. E., Nair R. The Architecture of Virtual Machines // Computer. 2005. Vol. 38, No. 5. P. 32–38. DOI: 10.1109/MC.2005.173.

97. Sommerville I. Software Engineering. 10th ed. Boston : Pearson, 2015. 816 p.

98. Stevens W. R., Rago S. A. Advanced Programming in the UNIX Environment. 3rd ed. Boston : Addison-Wesley, 2013. 976 p.

99. Stone J. E., Gohara D., Shi G. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems // Computing in Science & Engineering. 2010. Vol. 12, No. 3. P. 66–73. DOI: 10.1109/MCSE.2010.69.

100. Stroustrup B. A Tour of C++. 3rd ed. Boston : Addison-Wesley, 2022. 256 p.

101. Stroustrup B. The C++ Programming Language. 4th ed. Boston : Addison-Wesley, 2013. 1376 p.

102. Sutter H. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software // Dr. Dobb's Journal. 2005.

103. Sutter H., Alexandrescu A. C++ Coding Standards: 101 Rules, Guidelines, and Best Practices. Boston : Addison-Wesley, 2004. 224 p.

104. Sze V., Chen Y.-H., Yang T.-J., Emer J. S. Efficient Processing of Deep Neural Networks // Proceedings of the IEEE. 2017. Vol. 105, No. 12. P. 2295–2329. DOI: 10.1109/JPROC.2017.2761740.

105. Tanenbaum A. S., Bos H. Modern Operating Systems. 4th ed. Boston : Pearson, 2014. 1136 p.

106. Thomas D., Hunt A. The Pragmatic Programmer: Your Journey to Mastery. 20th Anniversary ed. Boston : Addison-Wesley, 2019. 352 p.

107. Turnbull J. The Phoenix Project: A Novel about IT, DevOps, and Helping Your Business Win. Portland : IT Revolution Press, 2013. 432 p.

108. Verma A., Pedrosa L., Korupolu M. et al. Large-scale Cluster Management at Google with Borg // Proceedings of the European Conference on Computer Systems (EuroSys). 2015. DOI: 10.1145/2741948.2741964.

109. WebAssembly Core Specification [Електронний ресурс] / World Wide Web Consortium. 2024. URL: https://www.w3.org/TR/wasm-core-2/

110. WebAssembly System Interface (WASI) [Електронний ресурс] / WebAssembly Community Group. 2024. URL: https://wasi.dev/

111. Williams A. C++ Concurrency in Action. 2nd ed. Shelter Island : Manning, 2019. 592 p.

112. Xavier M. G., Neves M. V., Rossi F. D. et al. Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments // Future Generation Computer Systems. 2014. Vol. 30. P. 1–12. DOI: 10.1016/j.future.2013.05.021.

113. Zakai A. Emscripten: An LLVM-to-JavaScript Compiler // OOPSLA Companion. 2011. P. 301–312. DOI: 10.1145/2048147.2048224.

114. Zaks A., Rose J. LLVM Compiler Infrastructure // Encyclopedia of Parallel Computing. Berlin : Springer, 2011.

115. Zhao Q., Rabbah R., Amarasinghe S. P. et al. Hardware/Software Co-design for Heterogeneous Systems // IEEE Micro.

ДОДАТОК А
СПИСОК ОПУБЛІКОВАНИХ ПРАЦЬ АВТОРА ЗА ТЕМОЮ ДИСЕРТАЦІЇ

Публікації у виданнях, включених до переліку наукових фахових видань України:

1. Бердник М.Г., **Стародубський І.П.** (2025). Use of genetic algorithms in adaptive compilers for cross-platform optimization. Radio Electronics, Computer Science, Control. 4 (75), 185-193. DOI: https://doi.org/10.15588/1607-3274-2025-4-16 [Web of Science, Index Copernicus, Google Scholar, Crossref]
2. Бердник М.Г., **Стародубський І.П.** (2025). Self-profiling mechanisms for real-time code compilers. System technologies. 6 (161), 85-95. DOI: https://doi.org/10.34185/1562-9945-5-161-2025-08 [Index Copernicus, Google Scholar, Crossref]
3. Бердник М.Г., **Стародубський І.П.** (2025). Using machine learning methods for automated cloud computing optimization. Інформаційні технології та суспільство, 3 (18), 16-23. DOI: https://doi.org/10.32689/maup.it.2025.3.2 [Index Copernicus, Google Scholar, Crossref]
4. Бердник М.Г., **Стародубський І.П.** (2025). Підвищення переносимості вихідного коду C++ для багатоплатформних обчислювальних систем. Вісник Хмельницького національного університету. Серія: Технічні науки, 4, 27-31. DOI: https://doi.org/10.31891/2307-5732-2025-355-3 [Index Copernicus, Google Scholar, Crossref]
5. Бердник М.Г., **Стародубський І.П**. (2025). Перенесення програм між cpu, gpu, tpu та fpga: виклики та рішення. Information Technology: Computer Science, Software Engineering and Cyber Security, 2, 3-9, DOI: http://dx.doi.org/10.32782/it/2025-2-1 [Index Copernicus, Google Scholar, Crossref]

6. Бердник М.Г., **Стародубський І.П.** (2025). Використання методів машинного навчання для адаптації програмного забезпечення до різних обчислювальних платформ. Information Technology: Computer Science, Software Engineering and Cyber Security, 4, 16-28. DOI: 10.32782/it/2024-4-3 [Index Copernicus, Google Scholar, Crossref]

Публікації у матеріалах наукових конференцій:

1. Бердник М.Г., **Стародубський І.П.** (2025). Метод адаптивної контейнеризації з інтеграцією штучного інтелекту. Проблеми використання інформаційних технологій в освіті, науці та промисловості. XX міжнародна конференція. Part of ISBN: 978-617-8737-34-4 https://pzks.nmu.org.ua/ua/science/2025_fin.pdf
2. Бердник М.Г., **Стародубський І.П.** (2025). Тестові набори, що самовідновлюються: автоматизація підтримки нестабільних тестів. Інформаційні управляючі системи і технології (ІУСТ-ОДЕСА-2025). DOI: 10.36059/978-966-397-531-3 Part of ISBN: 978-966-397-531-3 http://catalog.liha-pres.eu/index.php/liha-pres/catalog/book/413
3. Бердник М.Г., **Стародубський І.П.** (2025). Інструменти для автоматичного генерування абстракцій над платформозалежним кодом у C++ проєктах. Proceedings of I international scientific and practical conference. Part of ISBN: 978-3-954753-01-7 https://sci-conf.com.ua/wp-content/uploads/2025/09/SCIENCE-AND-EDUCATION-SYNERGY-OF-INNOVATION-1-3.09.25.pdf
4. Бердник М.Г., **Стародубський І.П.** (2025). Метрики оцінки переносимості C++ коду у вбудованих системах. Science And Technology: Challenges, Prospects and Innovations. Part of ISBN: 978-4-9783419-4-5 https://sci-conf.com.ua/wp-content/uploads/2025/08/SCIENCE-AND-TECHNOLOGY-CHALLENGES-PROSPECTS-AND-INNOVATIONS-14-16.08.25.pdf

5. Бердник М.Г., **Стародубський І.П.** (2025). Методи автоматизованого виявлення платформозалежного коду в C++ проєктах. Proceedings of IX international scientific and practical conference. Part of ISBN: 978-84-15927-30-3 https://sci-conf.com.ua/wp-content/uploads/2025/08/EUROPEAN-CONGRESS-OF-SCIENTIFIC-DISCOVERY-18-20.08.25.pdf
6. Бердник М.Г., **Стародубський І.П.** (2024). Автоматична адаптація програмного забезпечення до хмарних та периферійних платформ методами машинного навчання. XIX Міжнародна конференція з проблем використання інформаційних технологій в освіті, науці та промисловості. Part of ISBN: 978-966-934-666-7 https://pzks.nmu.org.ua/ua/science/conf2024.pdf
7. Бердник М.Г., **Стародубський І.П.**, Захаров Д.І. (2024). Адаптивні компілятори для переносимості програмного забезпечення до різних обчислювальних платформ. Інформаційне суспільство: технологічні, економічні та технічні аспекти становлення. http://www.konferenciaonline.org.ua/ua/article/id-1878/
8. Бердник М.Г., **Стародубський І.П.**, Захаров Д.І. (2024). Еволюційні операції та особливості їх застосування для вирішення задачі генерації тестових даних. Інформаційне суспільство: технологічні, економічні та технічні аспекти становлення. http://www.konferenciaonline.org.ua/ua/article/id-1657/
9. Бердник М.Г., **Стародубський І.П.**, Захаров Д.І. (2024). Механізм рефлексивного аналізу методів переносимості програм на різні обчислювальні платформи. Інформаційне суспільство: технологічні, економічні та технічні аспекти становлення. http://www.konferenciaonline.org.ua/ua/article/id-1658/
10. Бердник М.Г., **Стародубський І.П.**, Захаров Д.І. (2023). Метод переносимості програм на різні обчислювальні платформи. Scientific Research in the Modern World. Part of ISBN: 978-1-4879-3795-9

https://sci-conf.com.ua/wp-content/uploads/2023/09/SCIENTIFIC-RESEARCH-IN-THE-MODERN-WORLD-21-23.09.23.pdf

11. Бердник М.Г., **Стародубський І.П.**, Захаров Д.І. (2023). Застосування генетичного алгоритму для формування наборів вхідних тестових даних. Інформаційне суспільство: технологічні, економічні та технічні аспекти становлення. http://www.konferenciaonline.org.ua/ua/article/id-1250/

ДОДАТОК Б
ДОКУМЕНТИ ЩОДО ВПРОВАДЖЕННЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕНЬ

ЗАТВЕРДЖУЮ

Директор товариства з обмеженою відповідальністю

"Біологічні активні добавки"

Дьяченко А. Є.

АКТ

впровадження результатів дисертаційного дослідження

Стародубського Ігоря Петровича

на тему

«Методи та засоби переносимості програм та програмних систем на різні обчислювальні платформи»

Наукові та науково-практичні результати дисертаційного дослідження Стародубського І.П. на тему «Методи та засоби переносимості програм та програмних систем на різні обчислювальні платформи» були впроваджені у діяльність товариства з обмеженою відповідальністю «Біологічні активні добавки» під час створення та модернізації інформаційних систем контролю якості виробництва біологічно активних добавок.

Запропоновані автором методи забезпечення переносимості програмних систем використані для розроблення уніфікованої архітектури програмного забезпечення, що дозволяє ефективно виконувати автоматизований збір, обробку та аналіз даних з різних технологічних ліній і лабораторних пристроїв, незалежно від типу

ДЕРЖАВНЕ НАУКОВО-ВИРОБНИЧЕ ПІДПРИЄМСТВ «ЕЛЬДОРАДО»

Україна, 49600, м. Дніпро,
вул. Наб. Перемоги,32

АКТ

Впровадження результатів дисертаційного дослідження

Стародубського Ігоря Петровича

на тему

«Методи та засоби переносимості програм та програмних систем на різні обчислювальні платформи»

Результати дисертаційного дослідження Стародубського Ігоря Петровича були використані у розробленні технічних рішень щодо вирішення завдань проблемно-орієнтованого комплексу агропромислового виробництва. Основні наукові результати дослідження були впроваджені на підприємстві ДНВП «Ельдорадо» та використовуються для переносимості та впровадження систем управління і контролю технологічних процесів сушки зерна (автоматизації зерносушарок) з використанням вологоміра зерна в потоці як на вітчизняних, так і на імпортних зерносушарках.

Ці системи включають комплекс програмного забезпечення та апаратної частини на базі датчиків для вимірювання температури та вологості зерна в потоці, електронного управління затворами (КЕП), аварійного відключення факела, аварійної сигналізації при перевищенні заданої температури, а також протоколювання ходу технологічного процесу за всіма контрольованими параметрами і діями оператора.

Ці методи є вкрай необхідними для забезпечення адаптації та переносимості, а також для безперервної роботи систем, безпеки підприємства та безпеки обслуговуючого персоналу.

Завдяки цим методам на сьогодні активно розробляються та впроваджуються такі системи.

З повагою,
Директор

Просяник А.В.

ЗАТВЕРДЖУЮ
Директор Іноземного
Підприємства
«SoftRequest LTD»
«20» жовтня 2025 р.
Олександр Піцик

АКТ
впровадження результатів дисертаційного дослідження
Стародубського Ігоря Петровича
на тему
«Методи та засоби переносимості програм та програмних систем на різні обчислювальні платформи»

Наукові та науково-практичні результати дисертаційного дослідження Стародубського І.П. на тему «Методи та засоби переносимості програм та програмних систем на різні обчислювальні платформи», поданої на здобуття наукового ступеня доктора філософії за спеціальністю 122 – Комп'ютерні науки, використані в Іноземному Підприємстві SoftRequest LTD під час виконання робіт, пов'язаних з розробленням, адаптацією та оптимізацією програмного забезпечення для гетерогенних обчислювальних середовищ.

У процесі впровадження використано запропоновані автором методи підвищення переносимості програмного коду, що забезпечують:

- ефективну адаптацію алгоритмів до різних архітектур (CPU, GPU, FPGA);
- зменшення витрат часу на перенесення та тестування програм;
- збереження коректності та продуктивності програмних систем після міграції між платформами;
- уніфікацію процесу розгортання програмних рішень у різних обчислювальних середовищах.

Розроблені в межах дисертаційного дослідження підходи застосовано під час створення внутрішніх технологічних рішень компанії для аналізу, трансформації та оптимізації програмного коду, що дозволило підвищити ефективність процесів розроблення та супроводу програмних систем.

Директор Іноземного Підприємства «SoftRequest LTD»
Олександр Піцик

ТОВ НВП «АГРОПРОМАВТОМАТИЗАЦІЯ»

Україна, 49640, м. Дніпро

АКТ

Впровадження результатів дисертаційного дослідження

Стародубського Ігоря Петровича

на тему

«Методи та засоби переносимості програм та програмних систем на різні обчислювальні платформи»

Результати дисертаційного дослідження Стародубського Ігоря Петровича були використані у розробленні технічних рішень щодо вирішення завдань проблемно-орієнтованого комплексу агропромислового виробництва. Основні наукові результати дослідження були впроваджені на підприємстві ТОВ НВП «Агропромавтоматизація» та використовуються для переносимості та впровадження систем управління і контролю технологічних процесів сушки зерна (автоматизації зерносушарок) з використанням вологоміра зерна в потоці як на вітчизняних, так і на імпортних зерносушарках.

Ці системи включають комплекс програмного забезпечення та апаратної частини на базі датчиків для вимірювання температури та вологості зерна в потоці, електронного управління затворами (КЕП), аварійного відключення факела, аварійної сигналізації при перевищенні заданої температури, а також протоколювання ходу технологічного процесу за всіма контрольованими параметрами і діями оператора.

Ці методи є вкрай необхідними для забезпечення адаптації та переносимості, а також для безперервної роботи систем, безпеки підприємства та безпеки обслуговуючого персоналу.

Завдяки цим методам на сьогодні активно розробляються та впроваджуються такі системи.

Головний конструктор Борисенко А. Ю.

ЗАТВЕРДЖУЮ

Проректор з науково - педагогічної та

навчально - виховної роботи

Ігор Нікітенко

«22» грудня 2025 р.

АКТ
впровадження результатів дисертаційного дослідження Стародубського Ігоря Петровича на тему «Методи та засоби переносимості програм та програмних систем на різні обчислювальні платформи» в навчальний процес

Наукові та науково-практичні результати дисертаційного дослідження на тему «Методи та засоби переносимості програм та програмних систем на різні обчислювальні платформи», що виконувалися протягом 2023 – 2025 рр., впроваджено в 2024/2025 навчальному році в навчальному процесі на факультеті інформаційних технологій на кафедрі програмного забезпечення комп'ютерних систем при викладанні дисциплін: для бакалаврів спеціальності 122 Комп'ютерні науки «Машинне навчання» (доц. Приходченко С.Д.); для магістрів спеціальності 121 Інженерія програмного забезпечення «Модифікація та тестування програмного забезпечення» (проф. Бердник М.Г.).

Декан факультету інформаційних технологій
_________Ірина УДОВІК
«22» грудня 2025 р.

Завідувач кафедри програмного забезпечення комп'ютерних систем
_________Михайло АЛЕКСЄЄВ
«22» грудня 2025 р.

ДОДАТОК В

ЛІСТИНГ ІНІЦІАЛІЗАЦІЇ КЛІЄНТА KUBERNETES

```
import sys
import logging
from typing import Tuple, Optional
from kubernetes import client, config
from kubernetes.client.rest import ApiException

class KubernetesControlPlaneClient:
    def __init__(self):
        self._apps_api: Optional[client.AppsV1Api] = None
        self._core_api: Optional[client.CoreV1Api] = None
        self._logger = logging.getLogger("K8sInitializer")

    def authenticate(self):
        try:
            config.load_incluster_config()
        except config.ConfigException:
            try:
                config.load_kube_config()
            except config.ConfigException as e:
                self._logger.critical(f"Configuration load failed: {e}")
                raise RuntimeError("Could not load Kubernetes configuration") from e

        configuration = client.Configuration.get_default_copy()
        configuration.client_side_validation = False
        client.Configuration.set_default(configuration)

        self._apps_api = client.AppsV1Api()
        self._core_api = client.CoreV1Api()

    def check_health(self) -> bool:
        if self._core_api is None:
            return False
        try:
            self._core_api.get_api_versions()
            return True
        except ApiException as e:
            self._logger.error(f"API Health Check Failed: {e.status} {e.reason}")
            return False
        except Exception as e:
            self._logger.error(f"Connection Health Check Failed: {e}")
            return False

    def get_clients(self) -> Tuple[client.AppsV1Api, client.CoreV1Api]:
        if self._apps_api is None or self._core_api is None:
            self.authenticate()

        if not self.check_health():
            raise RuntimeError("Kubernetes API unavailable or unhealthy")

        return self._apps_api, self._core_api

def init_k8s_controller() -> Tuple[client.AppsV1Api, client.CoreV1Api]:
    client_manager = KubernetesControlPlaneClient()
    return client_manager.get_clients()

if __name__ == "__main__":
```

```
logging.basicConfig(level=logging.INFO)
try:
    apps_api, core_api = init_k8s_controller()
except Exception as e:
    logging.fatal(str(e))
    sys.exit(1)
```

ДОДАТОК Г

ЛІСТИНГ ОТРИМАННЯ МЕТРИК З PROMETHEUS

```
import os
import logging
import requests
import numpy as np
from typing import Dict, Any, List, Optional

class PrometheusCollector:
    def __init__(self, base_url: Optional[str] = None, window: Optional[str] = None):
        self.base_url = base_url or os.getenv("PROMETHEUS_URL", "http://prometheus:9090")
        self.window = window or os.getenv("METRIC_WINDOW", "30s")
        self._logger = logging.getLogger("PrometheusCollector")
        self._session = requests.Session()

    def _query(self, query_str: str) -> float:
        try:
            url = f"{self.base_url}/api/v1/query"
            response = self._session.get(
                url,
                params={"query": query_str},
                timeout=5.0
            )
            response.raise_for_status()
            data = response.json()

            if data.get("status") != "success":
                raise ValueError(f"Prometheus API error: {data.get('error')}")

            results = data.get("data", {}).get("result", [])

            if not results:
                return 0.0

            value_pair = results.get("value", [])
            if len(value_pair) < 2:
                return 0.0

            return float(value_pair[1])

        except Exception as e:
            self._logger.error(f"Failed to query metric: {query_str}. Error: {e}")
            raise

    def collect_full_vector(self, namespace: str, container: str) -> np.ndarray:
        queries = {
            "cpu_usage": f'avg(rate(container_cpu_usage_seconds_total{{container="{container}",
namespace="{namespace}"}}[{self.window}]))',
            "memory_working_set": f'avg(container_memory_working_set_bytes{{container="{container}",
namespace="{namespace}"}})',
            "latency_p95": f'histogram_quantile(0.95,
sum(rate(http_request_duration_seconds_bucket{{namespace="{namespace}"}}[{self.window}])) by (le))',
            "cpu_throttling": f'sum(rate(container_cpu_cfs_throttled_seconds_total{{container="{container}",
namespace="{namespace}"}}[{self.window}]))',
            "memory_page_faults": f'sum(rate(container_memory_failures_total{{container="{container}",
namespace="{namespace}", scope="container", type="pgmajfault"}}[{self.window}]))',
            "net_receive_bytes":
f'sum(rate(container_network_receive_bytes_total{{namespace="{namespace}"}}[{self.window}]))',
```

```
            "net_transmit_bytes":
f'sum(rate(container_network_transmit_bytes_total{{namespace="{namespace}"}}[{self.window}]))',
            "fs_read_bytes": f'sum(rate(container_fs_reads_bytes_total{{container="{container}",
namespace="{namespace}"}}[{self.window}]))',
            "fs_write_bytes": f'sum(rate(container_fs_writes_bytes_total{{container="{container}",
namespace="{namespace}"}}[{self.window}]))',
            "http_throughput": f'sum(rate(http_requests_total{{namespace="{namespace}"}}[{self.window}]))',
            "http_error_rate": f'sum(rate(http_requests_total{{namespace="{namespace}",
status=~"5.."}}[{self.window}]))',
            "container_restarts": f'sum(increase(kube_pod_container_status_restarts_total{{container="{container}",
namespace="{namespace}"}}[{self.window}]))',
            "node_memory_pressure": 'avg(node_memory_MemAvailable_bytes) /
avg(node_memory_MemTotal_bytes)'
        }

        metrics_list: List[float] = []

        for _, query in queries.items():
            val = self._query(query)
            metrics_list.append(val)

        return np.array(metrics_list, dtype=np.float64)

def collect_metrics(namespace: str = "default", container_name: str = "app") -> np.ndarray:
    collector = PrometheusCollector()
    return collector.collect_full_vector(namespace, container_name)
```

ДОДАТОК Д

ЛІСТИНГ ФОРМУВАННЯ ВЕКТОРА-ФУНКЦІЇ m(t)

```
import numpy as np
from typing import Dict, Union, Any
from kubernetes import client, config
from kubernetes.client.rest import ApiException

class ResourceParser:
    @staticmethod

    def parse_cpu(quantity: Union[str, int, float]) -> float:
        value = str(quantity)
        if value.endswith('m'):
            return float(value[:-1]) / 1000.0
        return float(value)

    @staticmethod
    def parse_memory(quantity: Union[str, int, float]) -> float:
        units = {
            'Ki': 1024, 'Mi': 1024**2, 'Gi': 1024**3, 'Ti': 1024**4,
            'Pi': 1024**5, 'Ei': 1024**6, 'm': 1e-3,
            'k': 1000, 'M': 1000**2, 'G': 1000**3, 'T': 1000**4,

            'P': 1000**5, 'E': 1000**6
        }
        value_str = str(quantity)
        for unit, multiplier in units.items():
            if value_str.endswith(unit):
                try:
                    num_part = value_str[:-len(unit)]
                    return float(num_part) * multiplier
                except ValueError:
                    return 0.0
        try:
            return float(value_str)
        except ValueError:
            return 0.0

class StateConstructionLayer:
    def __init__(self):
        try:
            config.load_incluster_config()
        except config.ConfigException:
            config.load_kube_config()
        self._core_api = client.CoreV1Api()

    def _get_node_architecture_encoding(self, labels: Dict[str, str]) -> np.ndarray:
        arch = labels.get("kubernetes.io/arch", "").lower()
        if arch in ["amd64", "x86_64"]:
            return np.array([1.0, 0.0, 0.0], dtype=np.float64)
        elif arch in ["arm64", "aarch64"]:
            return np.array([0.0, 1.0, 0.0], dtype=np.float64)
        elif arch in ["riscv64", "riscv"]:
            return np.array([0.0, 0.0, 1.0], dtype=np.float64)
        else:
            return np.array([0.0, 0.0, 0.0], dtype=np.float64)

    def get_platform_profile(self, node_name: str) -> np.ndarray:
```

```
    try:
        node = self._core_api.read_node(node_name)
        labels = node.metadata.labels or {}
        arch_vector = self._get_node_architecture_encoding(labels)

        allocatable = node.status.allocatable or {}
        cpu_cap = ResourceParser.parse_cpu(allocatable.get("cpu", "0"))
        mem_cap = ResourceParser.parse_memory(allocatable.get("memory", "0"))

        max_pods = float(allocatable.get("pods", "110"))
        if max_pods <= 0:
            max_pods = 110.0

        pods_on_node = self._core_api.list_pod_for_all_namespaces(
            field_selector=f"spec.nodeName={node_name}"
        )
        current_pods = len(pods_on_node.items)
        node_load_factor = current_pods / max_pods

        profile_vector = np.concatenate([
            arch_vector,
            np.array([cpu_cap, mem_cap, node_load_factor], dtype=np.float64)
        ])
        return profile_vector

    except ApiException:
        return np.zeros(6, dtype=np.float64)
    except Exception:
        return np.zeros(6, dtype=np.float64)

def build_feature_vector(self,
                         metrics_m: np.ndarray,
                         platform_p: np.ndarray,
                         current_config: Dict[str, Union[str, int]]) -> np.ndarray:
    cpu_limit = ResourceParser.parse_cpu(current_config.get("cpu_limit", "0"))
    mem_limit = ResourceParser.parse_memory(current_config.get("mem_limit", "0"))
    replicas = float(current_config.get("replicas", 1))

    config_vector = np.array([cpu_limit, mem_limit, replicas], dtype=np.float64)

    phi_t = np.concatenate([
        metrics_m,
        platform_p,
        config_vector
    ])

    return phi_t.reshape(1, -1)
```

ДОДАТОК Е

ЛІСТИНГ ОТРИМАННЯ МЕТАДАНИХ ВУЗЛІВ КЛАСТЕРА

```
import logging
from typing import Dict, List, Any, Optional
from kubernetes import client
from kubernetes.client.models import V1Node, V1NodeList
from kubernetes.client.rest import ApiException

class ResourceQuantityParser:
    @staticmethod
    def parse_cpu(quantity: str) -> float:
        if not quantity:
            return 0.0
        value = str(quantity)
        if value.endswith('m'):
            return float(value[:-1]) / 1000.0
        if value.endswith('n'):
            return float(value[:-1]) / 1000000000.0
        try:
            return float(value)
        except ValueError:
            return 0.0

    @staticmethod
    def parse_memory(quantity: str) -> int:
        if not quantity:
            return 0
        units = {
            'Ki': 1024,
            'Mi': 1024**2,
            'Gi': 1024**3,
            'Ti': 1024**4,
            'Pi': 1024**5,
            'Ei': 1024**6,
            'm': 1e-3,
            'k': 1000,
            'M': 1000**2,
            'G': 1000**3,
            'T': 1000**4,
            'P': 1000**5,
            'E': 1000**6
        }
        value_str = str(quantity)
        for unit, multiplier in units.items():
            if value_str.endswith(unit):
                try:
                    num_part = value_str[:-len(unit)]
                    return int(float(num_part) * multiplier)
                except ValueError:
                    return 0
        try:
            return int(value_str)
        except ValueError:
            return 0

class ClusterNodeProfiler:
    def __init__(self, core_api: client.CoreV1Api):
        self.core_api = core_api
```

```
        self.logger = logging.getLogger("NodeProfiler")

    def _extract_labels(self, node: V1Node) -> Dict[str, str]:
        labels = node.metadata.labels or {}
        return {
            "arch": labels.get("kubernetes.io/arch", "unknown"),
            "os": labels.get("kubernetes.io/os", "unknown"),
            "hostname": labels.get("kubernetes.io/hostname", "unknown"),
            "instance_type": labels.get("node.kubernetes.io/instance-type", "standard"),
            "topology_zone": labels.get("topology.kubernetes.io/zone", "unknown"),
            "topology_region": labels.get("topology.kubernetes.io/region", "unknown")
        }

    def _extract_resources(self, resources: Dict[str, str]) -> Dict[str, float]:
        if not resources:
            return {"cpu": 0.0, "memory": 0.0, "pods": 0.0}

        return {
            "cpu": ResourceQuantityParser.parse_cpu(resources.get("cpu", "0")),
            "memory": float(ResourceQuantityParser.parse_memory(resources.get("memory", "0"))),
            "pods": float(resources.get("pods", "0"))
        }

    def get_node_metadata(self, label_selector: Optional[str] = None) -> List[Dict[str, Any]]:
        try:
            nodes: V1NodeList = self.core_api.list_node(label_selector=label_selector)
            profiles = []

            for node in nodes.items:
                node_name = node.metadata.name
                labels = self._extract_labels(node)
                capacity = self._extract_resources(node.status.capacity)
                allocatable = self._extract_resources(node.status.allocatable)

                conditions = {
                    cond.type: cond.status
                    for cond in node.status.conditions
                    if cond.status == "True"
                }

                node_profile = {
                    "node_name": node_name,
                    "architecture": labels["arch"],
                    "operating_system": labels["os"],
                    "instance_type": labels["instance_type"],
                    "capacity": capacity,
                    "allocatable": allocatable,
                    "conditions": conditions,
                    "provider_id": node.spec.provider_id or "unknown"
                }
                profiles.append(node_profile)

            return profiles

        except ApiException as e:
            self.logger.error(f"Kubernetes API error during node profiling: {e.status} {e.reason}")
            return []
        except Exception as e:
            self.logger.error(f"Unexpected error retrieving node metadata: {e}")
            return []

    def get_aggregated_platform_profile(self) -> Dict[str, Any]:
```

```
profiles = self.get_node_metadata()
if not profiles:
    return {}

total_cpu = sum(p["capacity"]["cpu"] for p in profiles)
total_mem = sum(p["capacity"]["memory"] for p in profiles)

arch_distribution = {}
for p in profiles:
    arch = p["architecture"]
    arch_distribution[arch] = arch_distribution.get(arch, 0) + 1

return {
    "total_nodes": len(profiles),
    "total_cpu_cores": total_cpu,
    "total_memory_bytes": total_mem,
    "architecture_distribution": arch_distribution,
    "nodes": profiles
}
```

ДОДАТОК Ж

ЛІСТИНГ ПРОГНОЗУВАННЯ D(t+Δ) ДЛЯ МНОЖИНИ КАНДИДАТНИХ КЕРУЮЧИХ ВПЛИВІВ

```
import os
import logging
import joblib
import pickle
import numpy as np
from typing import Dict, Optional
from sklearn.base import BaseEstimator

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger("InferenceEngine")

class PortabilityInferenceEngine:
    def __init__(self, model_path: Optional[str] = None):
        self._model_path = model_path or os.getenv("MODEL_PATH", "/app/models/deviation_regressor.pkl")
        self._model: BaseEstimator = self._load_model()

    def _load_model(self) -> BaseEstimator:
        if not os.path.exists(self._model_path):
            raise FileNotFoundError(f"Model file not found at {self._model_path}")

        try:
            model = joblib.load(self._model_path)
            if not hasattr(model, "predict"):
                raise TypeError("Loaded object does not implement 'predict' method")
            return model
        except (OSError, pickle.UnpicklingError) as e:
            logger.critical(f"Failed to load model: {e}")
            raise RuntimeError("Model loading failed") from e

    def _ensure_2d_array(self, vector: np.ndarray) -> np.ndarray:
        if vector.ndim == 1:
            return vector.reshape(1, -1)
        return vector

    def estimate_current_deviation(self, phi_t: np.ndarray) -> float:
        try:
            input_vector = self._ensure_2d_array(phi_t)
            prediction = self._model.predict(input_vector)
            return float(np.clip(prediction, 0.0, 1.0))
        except Exception as e:
            logger.error(f"Inference error during current deviation estimation: {e}")
            raise

    def predict_future_batch(self, candidate_states: Dict[str, np.ndarray]) -> Dict[str, float]:
        results = {}
        if not candidate_states:
            return results

        try:
            action_ids = list(candidate_states.keys())
            vectors = [self._ensure_2d_array(candidate_states[uid]) for uid in action_ids]
            feature_matrix = np.array(vectors)

            predictions = self._model.predict(feature_matrix)
```

```
            for i, uid in enumerate(action_ids):
                results[uid] = float(np.clip(predictions[i], 0.0, 1.0))

            return results
        except Exception as e:
            logger.error(f"Batch prediction error: {e}")
            raise

def init_ai_module(path: Optional[str] = None) -> PortabilityInferenceEngine:
    return PortabilityInferenceEngine(path)
```

ДОДАТОК З

ЛІСТИНГ ФОРМУВАННЯ МНОЖИНИ КАНДИДАТНИХ КЕРУЮЧИХ ВПЛИВІВ З УРАХУВАННЯМ ОБМЕЖЕНЬ СИСТЕМИ ОРКЕСТРАЦІЇ

```
import itertools
from typing import List, Dict, Any, Optional, TypedDict, Union

class ActionSpec(TypedDict):
    uid: str
    type: str
    cpu: Optional[str]
    memory: Optional[str]
    replicas: Optional[int]

class ResourceValidator:
    @staticmethod
    def parse_cpu(quantity: Union[str, int, float]) -> float:
        val = str(quantity)
        if val.endswith('m'):
            return float(val[:-1]) / 1000.0
        if val.endswith('n'):
            return float(val[:-1]) / 1_000_000_000.0
        try:
            return float(val)
        except ValueError:
            return 0.0

    @staticmethod
    def parse_memory(quantity: Union[str, int]) -> float:
        val = str(quantity)
        units = {
            'Ki': 1024, 'Mi': 1024**2, 'Gi': 1024**3, 'Ti': 1024**4,
            'Pi': 1024**5, 'Ei': 1024**6,
            'k': 1000, 'M': 1000**2, 'G': 1000**3, 'T': 1000**4,
            'P': 1000**5, 'E': 1000**6
        }

        for unit, mult in units.items():
            if val.endswith(unit):
                try:
                    return float(val[:-len(unit)]) * mult
                except ValueError:
                    return 0.0

        try:
            return float(val)
        except ValueError:
            return 0.0

class CandidateActionGenerator:
    def __init__(self, constraints: Dict[str, Any], node_capacity: Dict[str, Any]):
        self._cpu_options = constraints.get("cpu_limits", [])
        self._mem_options = constraints.get("memory_limits", [])
        self._replica_range = constraints.get("replicas", {"min": 1, "max": 1})

        self._node_cpu_cap = ResourceValidator.parse_cpu(node_capacity.get("cpu", "0"))
        self._node_mem_cap = ResourceValidator.parse_memory(node_capacity.get("memory", "0"))
```

```
def _is_vertical_feasible(self, cpu_req: str, mem_req: str) -> bool:
    cpu_val = ResourceValidator.parse_cpu(cpu_req)
    mem_val = ResourceValidator.parse_memory(mem_req)

    if self._node_cpu_cap > 0 and cpu_val > self._node_cpu_cap:
        return False
    if self._node_mem_cap > 0 and mem_val > self._node_mem_cap:
        return False

    return True

def generate_admissible_actions(self, current_config: Dict[str, Any]) -> List[ActionSpec]:
    actions: List[ActionSpec] = []

    curr_cpu = str(current_config.get("cpu_limit", ""))
    curr_mem = str(current_config.get("mem_limit", ""))

    for cpu, mem in itertools.product(self._cpu_options, self._mem_options):
        if cpu == curr_cpu and mem == curr_mem:
            continue

        if self._is_vertical_feasible(cpu, mem):
            actions.append({
                "uid": f"v:{cpu}:{mem}",
                "type": "vertical",
                "cpu": cpu,
                "memory": mem,
                "replicas": None
            })

    try:
        curr_replicas = int(current_config.get("replicas", 1))
    except (ValueError, TypeError):
        curr_replicas = 1

    min_r = int(self._replica_range.get("min", 1))
    max_r = int(self._replica_range.get("max", 1))

    for r in range(min_r, max_r + 1):
        if r == curr_replicas:
            continue

        actions.append({
            "uid": f"h:{r}",
            "type": "horizontal",
            "cpu": None,
            "memory": None,
            "replicas": r
        })

    return actions
```

ДОДАТОК И

ЛІСТИНГ РЕАЛІЗАЦІЇ ЗАСТОСУВАННЯ ВЕРТИКАЛЬНОГО КЕРУЮЧОГО ВПЛИВУ ШЛЯХОМ ОНОВЛЕННЯ RESOURCE LIMITS КОНТЕЙНЕРА У DEPLOYMENT

```
import logging
from typing import Dict, Any, Optional, Union
from kubernetes import client, config
from kubernetes.client.rest import ApiException

class ActionExecutor:
    def __init__(self):
        try:
            config.load_incluster_config()
        except config.ConfigException:
            config.load_kube_config()

        self.apps_api = client.AppsV1Api()
        self.policy_api = client.PolicyV1Api()
        self._logger = logging.getLogger("KubernetesActuator")

    def _calculate_min_available(self, min_available: Union[int, str], total_replicas: int) -> int:
        if isinstance(min_available, int):
            return min_available

        if isinstance(min_available, str):
            if min_available.endswith('%'):
                try:
                    percentage = float(min_available[:-1]) / 100.0
                    return int(total_replicas * percentage)
                except ValueError:
                    return 0
            try:
                return int(min_available)
            except ValueError:
                return 0
        return 0

    def _validate_pdb_constraints(self, namespace: str, deployment_name: str, target_replicas: int) -> bool:
        try:
            pdbs = self.policy_api.list_namespaced_pod_disruption_budget(namespace=namespace)
            deployment = self.apps_api.read_namespaced_deployment(deployment_name, namespace)
            match_labels = deployment.spec.selector.match_labels

            if not match_labels:
                return True

            for pdb in pdbs.items:
                if not pdb.spec.selector or pdb.spec.selector.match_labels != match_labels:
                    continue

                min_available = pdb.spec.min_available
                if min_available is None:
                    continue

                threshold = self._calculate_min_available(min_available, target_replicas)
```

```python
            if target_replicas < threshold:
                return False

        return True

    except ApiException as e:
        self._logger.error(f"PDB validation failed: {e.status} {e.reason}")
        return False
    except Exception as e:
        self._logger.error(f"Unexpected error during PDB validation: {e}")
        return False

def apply_vertical_scaling(self, name: str, namespace: str, container: str, cpu: str, memory: str) -> bool:
    try:
        patch_body = {
            "spec": {
                "template": {
                    "spec": {
                        "containers": [{
                            "name": container,
                            "resources": {
                                "limits": {
                                    "cpu": cpu,
                                    "memory": memory
                                },
                                "requests": {
                                    "cpu": cpu,
                                    "memory": memory
                                }
                            }
                        }]
                    }
                }
            }
        }

        self.apps_api.patch_namespaced_deployment(
            name=name,
            namespace=namespace,
            body=patch_body
        )
        return True

    except ApiException as e:
        self._logger.error(f"Vertical scaling failed for {name}: {e.body}")
        return False

def apply_horizontal_scaling(self, name: str, namespace: str, replicas: int) -> bool:
    if not self._validate_pdb_constraints(namespace, name, replicas):
        self._logger.warning(f"Horizontal scaling blocked by PDB: {name} -> {replicas}")
        return False

    try:
        patch_body = {
            "spec": {
                "replicas": int(replicas)
            }
        }

        self.apps_api.patch_namespaced_deployment(
            name=name,
            namespace=namespace,
```

```
            body=patch_body
        )
        return True

    except ApiException as e:
        self._logger.error(f"Horizontal scaling failed for {name}: {e.body}")
        return False

def execute(self, action: Dict[str, Any], target_name: str, namespace: str, container_name: Optional[str] = None) -
> bool:
    action_type = action.get("type")

    if action_type == "vertical":
        return self.apply_vertical_scaling(
            name=target_name,
            namespace=namespace,
            container=container_name or target_name,
            cpu=str(action.get("cpu")),
            memory=str(action.get("memory"))
        )

    elif action_type == "horizontal":
        replicas = action.get("replicas")
        if replicas is None:
            return False
        return self.apply_horizontal_scaling(
            name=target_name,
            namespace=namespace,
            replicas=int(replicas)
        )

    return False
```

ДОДАТОК І

ЛІСТИНГ РЕАЛІЗАЦІЇ КЕРУЮЧОГО ЦИКЛУ

```
import os
import time
import signal
import logging
import sys
from typing import Dict, Any, Optional, List
from kubernetes import client, config

class AdaptiveControllerConfig:
    def __init__(self):
        self.namespace = os.getenv("TARGET_NAMESPACE", "default")
        self.deployment_name = os.getenv("TARGET_DEPLOYMENT", "app")
        self.control_interval = int(os.getenv("CONTROL_INTERVAL_SEC", "30"))
        self.observation_window = os.getenv("METRIC_WINDOW", "30s")
        self.deviation_threshold = float(os.getenv("DEVIATION_THRESHOLD", "0.2"))
        self.cooldown_period = int(os.getenv("COOLDOWN_PERIOD_SEC", "60"))
        self.model_path = os.getenv("MODEL_PATH", "/app/models/deviation_regressor.pkl")

class ReconciliationLoop:
    def __init__(
        self,
        config: AdaptiveControllerConfig,
        collector: Any,
        state_builder: Any,
        inference_engine: Any,
        action_executor: Any,
        action_generator: Any
    ):
        self._config = config
        self._collector = collector
        self._state_builder = state_builder
        self._inference = inference_engine
        self._executor = action_executor
        self._generator = action_generator

        self._running = False
        self._last_action_ts = 0.0
        self._logger = logging.getLogger("ReconciliationLoop")

        try:
            config.load_incluster_config()
        except config.ConfigException:
            config.load_kube_config()
        self._apps_api = client.AppsV1Api()

        signal.signal(signal.SIGINT, self._shutdown)
        signal.signal(signal.SIGTERM, self._shutdown)

    def _shutdown(self, signum, frame):
        self._running = False

    def _get_deployment_state(self) -> Dict[str, Any]:
        dep = self._apps_api.read_namespaced_deployment(
            name=self._config.deployment_name,
            namespace=self._config.namespace
        )
```

```
    container = dep.spec.template.spec.containers
    resources = container.resources.limits or {}
    return {
        "cpu_limit": resources.get("cpu", "0"),
        "mem_limit": resources.get("memory", "0"),
        "replicas": dep.spec.replicas or 1,
        "node_name": dep.spec.template.spec.node_name
    }

def run(self):
    self._running = True

    while self._running:
        cycle_start = time.time()

        try:
            if time.time() - self._last_action_ts < self._config.cooldown_period:
                self._sleep_until_next_cycle(cycle_start)
                continue

            metrics = self._collector.collect_full_vector(
                self._config.namespace,
                self._config.deployment_name
            )

            current_state = self._get_deployment_state()
            node_name = current_state.get("node_name")

            if not node_name:
                pods = self._apps_api.list_namespaced_pod(
                    namespace=self._config.namespace,
                    label_selector=f"app={self._config.deployment_name}"
                )
                if pods.items:
                    node_name = pods.items.spec.node_name

            platform_profile = self._state_builder.get_platform_profile(node_name)

            phi_t = self._state_builder.build_feature_vector(
                metrics,
                platform_profile,
                current_state
            )

            d_t = self._inference.estimate_current_deviation(phi_t)

            if d_t > self._config.deviation_threshold:
                actions = self._generator.generate_admissible_actions(current_state)

                candidate_vectors = {}
                action_map = {}

                for action in actions:
                    simulated_config = current_state.copy()
                    if action["type"] == "vertical":
                        simulated_config["cpu_limit"] = action["cpu"]
                        simulated_config["mem_limit"] = action["memory"]
                    elif action["type"] == "horizontal":
                        simulated_config["replicas"] = action["replicas"]

                    sim_phi = self._state_builder.build_feature_vector(
                        metrics,
```

```python
                        platform_profile,
                        simulated_config
                    )
                    candidate_vectors[action["uid"]] = sim_phi
                    action_map[action["uid"]] = action

                predictions = self._inference.predict_future_batch(candidate_vectors)

                if predictions:
                    best_uid = min(predictions, key=predictions.get)
                    min_d = predictions[best_uid]

                    if min_d < d_t:
                        best_action = action_map[best_uid]
                        success = self._executor.execute(
                            best_action,
                            self._config.deployment_name,
                            self._config.namespace
                        )
                        if success:
                            self._last_action_ts = time.time()

            except Exception as e:
                self._logger.error(f"Cycle failed: {e}")

            self._sleep_until_next_cycle(cycle_start)

    def _sleep_until_next_cycle(self, start_time: float):
        elapsed = time.time() - start_time
        sleep_time = max(0.0, self._config.control_interval - elapsed)
        time.sleep(sleep_time)

if __name__ == "__main__":
    logging.basicConfig(level=logging.INFO)

    from collector import PrometheusCollector
    from state_builder import StateConstructionLayer
    from inference import PortabilityInferenceEngine
    from actuator import ActionExecutor
    from generator import CandidateActionGenerator

    cfg = AdaptiveControllerConfig()

    constraints = {
        "cpu_limits": ["100m", "200m", "500m", "1000m"],
        "memory_limits": ["128Mi", "256Mi", "512Mi", "1Gi"],
        "replicas": {"min": 1, "max": 10}
    }

    node_capacity_stub = {"cpu": "2000m", "memory": "4Gi"}

    loop = ReconciliationLoop(
        config=cfg,
        collector=PrometheusCollector(window=cfg.observation_window),
        state_builder=StateConstructionLayer(),
        inference_engine=PortabilityInferenceEngine(cfg.model_path),
        action_executor=ActionExecutor(),
        action_generator=CandidateActionGenerator(constraints, node_capacity_stub)
    )

    loop.run()
```