

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КИЇВСЬКИЙ
ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»
МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КИЇВСЬКИЙ
ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»
МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Кваліфікаційна наукова
праця на правах рукопису

РУСІНОВ ВОЛОДИМИР ВОЛОДИМИРОВИЧ

УДК 004.75

ДИСЕРТАЦІЯ

МЕТОД ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ РОЗГОРТАННЯ
КОМПОНЕНТІВ ПЛАТФОРМИ ВБУДОВАНИХ СИСТЕМ

123 Комп'ютерна інженерія

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

Науковий керівник: Стіренко Сергій Григорович, д.т.н., професор

Київ – 2025

АНОТАЦІЯ

Русінов В.В. Метод підвищення ефективності розгортання компонентів платформи вбудованих систем. – Кваліфікаційна наукова праця на правах рукопису.

Дисертація на здобуття наукового ступеня доктора філософії за спеціальністю 123 – Комп'ютерна інженерія з галузі знань 12 – Інформаційні технології. – Національний Технічний Університет України «Київський Політехнічний Інститут імені Ігоря Сікорського», Київ, 2025.

Дисертаційна робота присвячена розробці методу розгортання компонент платформи для вбудованих систем. По результатах роботи було розроблено метод розгортання компонент ШІ-платформи на основі онтологій процесів, які включають в себе програмно-апаратні рішення на основі поєднання можливостей вбудованих систем та хмарних систем, та процес розгортання та моніторингу компонент платформи вбудованих, який використовує модифікований алгоритм із застосуванням *CI/CD* конвеєра для постійної підтримки працездатності платформи з урахуванням конкурентної зборки та прунінгу контейнерів.

Сучасний прогрес в сфері розробки вбудованих пристроїв зумовлює попит на їх широке впровадження в різних сферах, наприклад в системах відеоспостереження, автономних автомобілях, розумний інтелект в безпілотних апаратах, тощо. Це зумовлює попит на розробку методу розгортання компонент платформи вбудованих пристроїв, який дозволяє швидко проводити синхронізацію компонент платформи, враховуючи особливості задач ШІ, такі як необхідність оновлення моделі з появою нових більш точних моделей та високої потреби в ресурсах системи.

Використано процес *MLOps* в якості основи методу розгортання компонент платформи вбудованих систем. Цей підхід використовується для інтеграції

машинного навчання в цикли розробки програмного забезпечення та забезпечення ефективного управління процесами, що включають навчання моделей, їх тестування, розгортання та моніторинг. Впровадження *MLOps* дозволяє оптимізувати взаємодію між різними етапами життєвого циклу машинного навчання, що, в свою чергу, забезпечує високий рівень автоматизації та знижує ймовірність помилок. Основою цієї методології є забезпечення безперервної інтеграції та безперервне постачання (*CI/CD*) моделей ШІ, а також можливість їх швидкої адаптації до змінюваних умов та вимог. Проаналізовано, що більшості сучасних систем та рішень на основі *MLOps* притаманне використання ресурсно-інтенсивних архітектурних рішень, як використання *Cloud* або спеціалізованого обладнання.

Розроблено метод розгортання компонент платформи вбудованих системи на основі онтологій процесів. Онтології процесів, представлені через графові структури, такі як дерева Бема, дозволяють формалізувати концепти і їх взаємодії в рамках систем, що складають платформу, що, у свою чергу, сприяє точному відображенню семантики всіх системних складових. За допомогою таксономій і λ -термів, що реалізують семантичну зв'язність, можна детально описати взаємодії між компонентами, що дозволяє не тільки знижувати ймовірність помилок, але й підвищувати ефективність процесів, що відбуваються в межах компонентів платформи вбудованих систем. На основі запропонованого методу можна розгорнути компоненти тестової платформу, готові до практичного використання та використання з відповідним апаратним та програмним забезпеченні, у відповідності з описаними вимогами.

Удосконалено конвеєр *CI/CD* під процес *MLOps* для запропонованого методу розгортання компонент платформи на основі вбудованих систем. Використання конвеєру *CI/CD* для розгортання компонент, відповідальних за задачі ШІ, на основі вбудованих систем мають особливості які вимагають модифікацію

процесу, серед іншого, децентралізовану зборку в умовах гетерогенної природи системи, автоматизоване тестування точності моделі, моніторинг вбудованих пристроїв і аналіз та верифікацію продуктивності моделі. Використання конвеєру *CI/CD* у сукупності з методами оптимізації зборки контейнеру, дозволяють зменшити час розгортання системи в порівнянні з системами аналогічними підходами висвітленими в літературі, визначеними як базові.

На основі запропонованих підходів з використання прунінгу та конкурентної зборки, було підвищено ефективність процесу розгортання компонентів платформи за рахунок зменшення часу зборки контейнерів. За допомогою процесу прунінгу, можливо досягнути меншого розміру контейнеру при більшій швидкості зборки, який включає в себе видалення додаткових залежностей на етапі зборки контейнеру та на етапі розгортання. Даний метод показує прискорення зборки 5.79%, в порівнянні з запропонованими рішеннями. Прискорено час зборки контейнеру до 16.24%, порівняно з існуючими рішеннями, при зменшенні використання пам'яті на 11.15%.

Розроблено програмний додаток для перевірки та демонстрації роботи комплексного методу для розгортання системи на прикладі деревовидної структури з дебруйнівськими зв'язками. Програмний додаток дає можливість перевірити роботу платформи від початку до кінця та протестувати можливості роботи в умовах високої завантаженості системи. За допомогою даного підходу, платформа дозволяє виконувати переналаштування у разі відмови певного вузлу, тим самим перенаправивши потік запитів на інший вузол, без повної зупинки системи на переналаштування. В ході тестування було використано декілька різних моделей штучної нейронної мережі (ШНМ): *MobileNet*, *ResNet* та *InceptionNet* на датасеті *ANIMAL10N*. Проведене тестування підтвердило очікуване функціонування системи та тестування навантаження встановило можливість використання системи в реальному часі при високому вхідному

потоків запитів. Аналіз результатів показує, що платформа здатна швидко адаптуватись до зміни моделі ШНМ та продовжити роботу без відмови.

Ключові слова: *MLOps*, машинне навчання, конвеєр *CI/CD*, відмовостійкість, вбудовані пристрої, *Cloud*, топологічна організація, штучний інтелект, онтології.

ABSTRACT

Rusinov V.V. Method increasing the efficiency of component deployment for embedded systems. – Qualified scientific work as a manuscript.

Dissertation for the degree of Doctor of Philosophy in specialty 123 – Computer engineering in the field of knowledge 12 – Information technology. – National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", Kyiv, 2025.

The dissertation is devoted to the development of a method for deploying platform components for embedded systems. Based on the results of the work, a method was developed based on process ontologies, which include software and hardware solutions based on a combination of embedded and cloud systems capabilities, and a platform deployment plan that uses a modified algorithm using the CI/CD pipeline to constantly maintain the platform's performance, taking into account competitive assembly and pruning of containers.

Modern progress in the field of developing embedded devices determines the demand for their widespread implementation in various areas, for example, in video surveillance systems, autonomous cars, swarm intelligence in unmanned vehicles, etc. This determines the demand for the development of a method for deploying platform components based on embedded devices, which allows for rapid synchronization of platform components, taking into account the specifics of AI tasks, such as the need to update the model with the emergence of new, more accurate models and high demand for system resources.

The MLOps process was used as the basis for the method of deploying components of the embedded systems platform. This approach is used to integrate machine learning into software development cycles and ensure effective management of processes that include model training, testing, deployment, and monitoring. The implementation of MLOps allows you to optimize the interaction between different stages of the machine learning life cycle, which, in turn, provides a high level of automation and reduces the

likelihood of errors. The basis of this methodology is to ensure continuous integration and continuous delivery (CI/CD) of AI models, as well as the ability to quickly adapt them to changing conditions and requirements. It was analyzed that most modern systems and solutions based on MLOps are characterized by the use of resource-intensive architectural solutions, such as the use of Cloud or specialized equipment.

A method of deploying components of the embedded systems platform based on process ontologies was developed. Process ontologies, represented through graph structures such as Bohm trees, allow us to formalize concepts and their interactions within the systems that make up the platform, which, in turn, contributes to the accurate reflection of the semantics of all system components. Using taxonomies and λ -terms that implement semantic connectivity, we can describe in detail the interactions between components, which allows us not only to reduce the probability of errors, but also to increase the efficiency of processes occurring within the components of the embedded systems platform. Based on the proposed method, we can deploy test platform components that are ready for practical use and use with appropriate hardware and software, in accordance with the described requirements.

The CI/CD pipeline for the MLOps process has been improved for the proposed method of deploying platform components based on embedded systems. The use of the CI/CD pipeline for deploying components responsible for AI tasks based on embedded systems has features that require process modification, including decentralized assembly in the conditions of the heterogeneous nature of the system, automated testing of model accuracy, monitoring of embedded devices, and analysis and verification of model performance. The use of the CI/CD pipeline in conjunction with container assembly optimization methods allows reducing the system deployment time compared to systems with similar approaches highlighted in the literature, defined as baseline.

Based on the proposed approaches using pruning and competitive assembly, the efficiency of the platform component deployment process was increased by reducing the container assembly time. Using the pruning process, it is possible to achieve a smaller container size with a higher assembly speed, which includes removing

additional dependencies at the container assembly stage and at the deployment stage. This method shows a 14.52% assembly speedup while reducing the image size by 11.15%, in contrast to the basic approach shown in the literature. The use of competitive assembly shows a 14.7% speedup, reaching a 19.21% speedup in combination with pruning.

A software application has been developed to verify and demonstrate the operation of the integrated method for system deployment on the example of a tree structure with de Bruijn connections. The software application makes it possible to verify the platform from start to finish and test the capabilities of operation under high system load conditions. Using this approach, the platform allows for reconfiguration in the event of a failure of a particular node, thereby redirecting the flow of requests to another node, without completely stopping the system for reconfiguration. During the testing, several different artificial neural network (ANN) models were used: MobileNet, ResNet, and InceptionNet on the ANIMAL10N dataset. The testing confirmed the expected operation of the system and the load testing established the possibility of using the system in real time with a high incoming flow of requests. The analysis of the results shows that the platform is able to quickly adapt to changes in the ANN model and continue to work without failure.

Keywords: MLOps, machine learning, deep learning, CI/CD pipeline, fault tolerance, embedded devices, Cloud, topological organization, artificial intelligence, ontologies.

СПИСОК ПУБЛІКАЦІЙ ЗДОБУВАЧА

Наукові праці, в яких опубліковано основні наукові результати дисертації:

1. Волокита А., Русінов В., Мугуєв К. Дослідження відмовостійкості для топології Де Бруйна на основі коефіцієнта посередництва // Технічні науки та технології. – 2021. – Вип. 1, № 23. – С. 69–80. – doi: 10.25140/2411-5363-2021-1(23)-69-80.
2. Korenko D., Cherevatenko O., Rusinov V., Kulakov Y. Creation of the method of multipath routing using known paths in software-defined networks // Technology Audit and Production Reserves. – 2022. – Vol. 4, No. 2(66). – P. 19–24. – doi: 10.15587/2706-5448.2022.262787.
3. Rusinov V., Honcharenko O., Volokyta A., Loutskii H., Pustovit O., Kyrianov A. Methods of topological organization synthesis based on tree and dragonfly combinations // Lecture Notes on Data Engineering and Communications Technologies. – 2023. – P. 472–485. – doi: 10.1007/978-3-031-36118-0_43.
4. Volokyta A., Loutskii H., Honcharenko O., Cherevatenko O., Rusinov V., Kulakov Y., Tsybulia S. Fault Tolerance Exploration and SDN Implementation for de Bruijn Topology based on betweenness Coefficient // International Journal of Computer Network and Information Security. – 2024. – Vol. 16, No. 1. – P. 97–112. – doi: 10.5815/ijcnis.2024.01.08.
5. Русінов, В. (2025). Спосіб розгортання AI платформи з використанням методології MLOps. // Смарт технології: промислова та цивільна інженерія, 3(16), 19-28. – doi: 10.32347/st.2025.3.1202.
6. Rusinov, V., & Basenko, N. (2025, April). Exploration of the Efficiency of SLM-Enabled Platforms for Everyday Tasks. In 13th International Conference on Applied Innovations in IT (p. 133). doi: 10.25673/119225

Апробація наукових результатів дисертації:

7. Rusinov V., Cherevatenko O. Method of neural network training for Edge

architecture // The International Conference on Security, Fault Tolerance, Intelligence. – Kyiv, Ukraine, June 30, 2022. – 2022. – [Electronic resource]. – Available: <http://icsfti.kpi.ua/proc/article/view/280999/291232>.

8. Rusinov V., Muhuiev K. Development of a scalable AI platform based on integration of Edge computing with Cloud technologies // The International Conference on Security, Fault Tolerance, Intelligence. – Kyiv, Ukraine, June 28, 2024. – 2024. – [Electronic resource]. – Available: <https://icsfti-proc.kpi.ua/article/view/298011>.

9. Rusinov V. Increasing the efficiency of AI task deployment for Cloud-Edge environments // The International Conference on Security, Fault Tolerance, Intelligence. – Kyiv, Ukraine, June 28, 2024. – 2024. – [Electronic resource]. – Available: <https://icsfti-proc.kpi.ua/article/view/307047>.

10. Rusinov V. Development of MLOps pipeline to support AI systems // The International Conference on Security, Fault Tolerance, Intelligence. – Kyiv, Ukraine, June 28, 2024. – 2024. – [Electronic resource]. – Available: <https://icsfti-proc.kpi.ua/article/view/307046>.

Праці, які додатково відображають результати дисертації:

11. Oleksandr Pustovit, Rusinov Volodymyr, Oleksii Cherevatenko, Leonid Pustovit, Artem Volokyta. Isoefficient calculation method for discrete Fourier transform. Information, Computing and Intelligent systems : International Conference ICSFTI2022, м. Київ, 30 черв. 2022 р. Київ, 2022.

12. Голованенко М.В., Русінов В.В. (2025). Цифровізація бізнес-процесів компанії із впровадженням AI-платформи. Теоретичні та прикладні питання економіки. Збірник наукових праць, 1(50).

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ	15
ВСТУП.....	16
РОЗДІЛ 1 ОГЛЯД СУЧАСНИХ ПІДХОДІВ РОЗГОРТАННЯ КОМПОНЕНТ ПЛАТФОРМ ВБУДОВАНИХ СИСТЕМ	21
1.1. Аналіз сучасних масштабованих платформ вбудованих систем	21
1.1.1. Основні поняття платформ вбудованих систем	24
1.1.2. Топологічні організації платформ вбудованих систем.....	26
1.1.3. Особливості апаратного забезпечення компонент платформ вбудованих систем.....	27
1.2. Порівняльний аналіз підходів до розгортання сучасних платформ вбудованих систем.....	30
1.2.1. Проблематика розгортання MLOps	30
1.2.2. Практики розгортання MLOps в хмарних системах	33
1.2.3. Практики розгортання MLOps в вбудованих системах	36
1.3. Особливості процесу розгортання компонентів платформ вбудованих систем.....	43
1.3.1. Багаторівнева практика розгортання MLOps.....	46
1.3.2. Конвеєр даних при розгортанні платформ вбудованих систем	49
1.4. Узагальнення проблематики розгортання компонент платформ вбудованих систем.....	53
Висновки до розділу 1	55
РОЗДІЛ 2 МЕТОД РОЗГОРТАННЯ КОМПОНЕНТІВ ПЛАТФОРМ ВБУДОВАНИХ СИСТЕМ ІЗ ЗАСТОСУВАННЯМ ТЕХНОЛОГІЇ MLOPS	57

2.1. Розробка архітектури платформи вбудованих систем для розпізнавання образів	57
2.1.1. Особливості впровадження гібридної Edge-Cloud архітектури в платформу вбудованих систем.....	59
2.2. Особливості розгортання компонент платформи вбудованих систем на основі контейнеризації.....	62
2.2.1. Застосування планувальника задач для балансування навантаження контейнерів платформи.....	65
2.2.2. Опис моделі глибокої нейронної мережі для платформи вбудованих систем для розпізнавання образів	68
2.3. Застосування CI/CD конвеєру для розгортання платформи вбудованих систем.....	72
2.4. Метод розгортання компонент ІІІ-платформи на основі онтології процесів	79
2.5. Процес розгортання та моніторингу компонент платформи вбудованих систем з можливостями ІІІ	86
2.5.1. Шаблони розгортання моделі ІІІ для платформ вбудованих систем	87
2.5.2. Метрики оцінки відмовостійкості компонентів платформи	89
Висновки до розділу 2.....	94
РОЗДІЛ 3 ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ РОЗГОРТАННЯ КОМПОНЕНТІВ ПЛАТФОРМИ ВБУДОВАНИХ СИСТЕМ	96
3.1. Застосування технології Infrastructure as code	96
3.2. Оцінка відмовостійкості на основі коефіцієнту посередності	101
3.2.1. Міграція контейнеру на інший вузол після відмови.....	104

3.3. Підвищення ефективності розгортання контейнерів для компонентів на основі вбудованих систем.....	106
3.3.1. Застосування процесу конвеєрної зборки контейнера для компонентів платформи вбудованих систем.....	112
3.3.2. Архітектурні особливості розгортання контейнерів на Cloud-Edge архітектурі.....	117
3.4. Обробка залежностей в рамках розгортання платформи	119
3.5. Розробка схеми розгортання платформи вбудованих систем	123
Висновки до розділу 3.....	126
РОЗДІЛ 4 МОДЕЛЮВАННЯ МЕТОДУ РОЗГОРТАННЯ КОМПОНЕНТ ПЛАТФОРМИ ВБУДОВАНИХ СИСТЕМ ТА АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ.....	128
4.1. Застосування логування в рамках CI/CD процесу.....	128
4.2. Обґрунтування вибору програмного забезпечення для компонент вбудованих систем.....	130
4.2.1. Архітектура тестової системи	130
4.2.2. Технологічний стек системи	133
4.2.3. Вимоги до тестового середовища	135
4.3. Проведення експериментів розгортання та використання компоненти плаформи вбудованих систем.....	137
4.3.1. Експерименти з масштабування платформи.....	150
4.3.2. Порівняння з аналогічними платформами.....	155
4.4. Аналіз результатів моделювання методу розгортання компонент платформи вбудованих систем.....	161
Висновки до розділу 4.....	165

	14
ВИСНОВКИ	167
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	169
ДОДАТОК А	180
ДОДАТОК Б.....	207
ДОДАТОК В	209
ДОДАТОК Г	211
ДОДАТОК Д	213

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

AI – artificial intelligence, штучний інтелект.

CI/CD – Continuous Integraion, Continuous Deployment, постійна інтеграція, постійне розгортання.

DevOps – методологія, що поєднує розробку (Development) і операції (Operations).

ETL – extract transform load.

IoT – internet of things, інтернет речей.

ML – machine learning, машинне навчання.

MLOps – machine learning operations, підхід, що поєднує практики машинного навчання з операційними процесами.

SLA – service license agreement, договір між постачальником послуг і клієнтом.

ВСТУП

Актуальність теми. Актуальність даної тематики зумовлена зростаючим попитом на вбудовані системи в різних галузях, таких як медицина, автопромисловість, енергетика та інші. Штучний інтелект швидко розвивається та змінює спосіб роботи бізнесу, від автоматизації процесів до когнітивного розширення завдань та інтелектуальної аналітики процесів/даних. Разом з цим, еволюціонує і процес навчання моделей, і системи підтримки цього процесу. Головним завданням для користувачів системи ШІ є розуміння та довіра до результату алгоритмів і методів ШІ. З часом, будь-яка система зазнає значних втрат в точності та в продуктивності, а, отже потребує налагоджений алгоритм підтримки. Ці системи стають все більш складними та вимагають високого рівня інтелекту та автономності для ефективної роботи з даними.

Наукова та інженерна література показує, що використання методів *Machine Learning Operations* і *Machine Learning Pipeline (MLOps)* у створенні ШІ-платформ для вбудованих систем дозволяє покращити ефективність розробки та впровадження штучного інтелекту. *MLOps* надає можливість автоматизувати процеси розробки, навчання та експлуатації моделей машинного навчання, забезпечуючи швидку реакцію на зміни в середовищі або вхідних даних. Це сприяє зниженню часу на розгортання та підтримку системи в цілому.

Що важливо, традиційному хмарному застосунку, з використанням *DNN*, значною мірою заважає значна затримка глобальної мережі, що призводить до низької продуктивності в реальному часі, а також низької якості взаємодії з користувачем. Традиційний підхід обмежує можливості для збору даних, саме через це обмеження, та накладає додаткові обмеження на *Cloud*-застосунок.

Зв'язок роботи з науковими програмами, планами, темами. Тема дисертаційної роботи входить в план наукової роботи кафедри обчислювальної техніки КПІ ім. Ігоря Сікорського і виконана в рамках наступних пошукових

досліджень (ініціативних тематик): «Високопродуктивні комп'ютерні системи та мережі: теорія, методи і засоби апаратної та програмної реалізації» (факультет інформатики та обчислювальної техніки – керівник: доц. А. М. Волокита), № договору: Д/р №0121U108261, дата реєстрації: 11.02.2021.

Мета і завдання дослідження. Метою дисертаційного дослідження є підвищення ефективності розгортання компонент платформи вбудованих систем та обробки даних за допомогою методів *MLOps* для вбудованих пристроїв.

Для досягнення мети дисертаційної роботи, були поставлені завдання:

Для досягнення мети потрібно виконати наступне:

- Провести аналіз аспектів розгортання компонентів платформ вбудованих систем виходячи з існуючих технологій та процесів, з метою розробки власного методу, що дозволяє систематизувати процес побудови контейнерів та їх пересилку підвищуючи загальну ефективність платформи;
- Підвищити відмовостійкість компонентів платформи, використовуючи спеціалізовані топології на основі кодових перетворень ДеБруйна, який дає змогу побудувати додаткові маршрути для пересилки даних в платформі за рахунок дебруйнівських зв'язків;
- Розробити метод із розгортанням компонент платформи вбудованих систем для вбудованих пристроїв, на основі онтологій процесів, який передбачає інтеграцію архітектурних рішень і технологій розгортання та обробки даних;
- Вдосконалити процес розгортання компонентів платформи вбудованих систем використовуючи прунінг контейнерів та конвеєрні зборки контейнерів, з метою зменшення розміру контейнера та зменшення часу його пересилки від головного вузла до вузлів-посередників та вбудованих систем;

- Розробити інструментальний засіб, з метою реалізації запропонованого методі із застосуванням топологій на дебруйнівських зв'язках таких як *Tree-DeBrujin* та *Dragonfly-DeBrujin* для ефективного управління процесом розгортання компонентів платформи;
- Провести оцінку та аналіз отриманих результатів роботи системи, яка засновується на розробленій системі, показати її ефективність у реальних умовах, порівнюючи отримані результати з існуючими рішеннями, та рішеннями на основі існуючих підходів.

Об'єкт дослідження. Об'єктом дослідження дисертаційної роботи є процес розгортання платформ вбудованих систем на основі технологій *Edge* та *Cloud* із залученням технології *MLOps*.

Предмет дослідження. Предметом дослідження дисертаційної роботи є метод підвищення ефективності розгортання компонентів платформи шляхом групування *Edge* пристроїв та *Cloud* при впровадженні *CI/CD* конвеєру.

Методи дослідження. Методами дослідження дисертаційної роботи є пошук сучасної літератури на тематику розгортання компонент платформ вбудованих систем, розробка власного методу та його моделювання.

В ході опрацювання дослідження були використані методи статистичного аналізу, порівняльного аналізу, метрики швидкості та кількості передачі даних, моделювання пристроїв, емуляція апаратного середовища, профілювання продуктивності, моніторинг використання пам'яті, тестування на навантаження.

Наукова новизна отриманих результатів. Науковою новизною отриманих в процесі дисертаційного дослідження результатів є:

- Вперше було розроблено метод розгортання компонент ШІ-платформи, яка об'єднана у ієрархічну, гетерогенну структуру із застосуванням модифікованого способу *MLOps* що, на відміну від існуючих підходів, впроваджує конвеєр постійного обслуговування, що оновлює модель

штучної нейронної мережі, без необхідності в реконфігурації системи за рахунок модифікованого плану розгортання.

- Запропоновано метод розгортання системи із застосуванням модифікованого процесу *MLOps* на основі онтології процесів, який, на відміну від існуючих рішень, знизив час відгуку платформи при високому навантаженні, за рахунок застосування хмарних технологій в системі для більш трудомістких завдань.
- Запропоновано підхід до розгортання системи, який, на відміну від існуючих підходів, дозволяє заощадити час розгортання компонент платформи та зменшує використання ресурсів системи, забезпечуючи можливість більш швидкого відновлення компонент платформи, за рахунок конвеєрної зборки та прунінгу контейнерів.

Практичне значення отриманих результатів. Результати дисертаційної роботи дають реалізувати готове рішення для підтримки сучасних систем із застосуванням технології штучного інтелекту, де важливими факторами є постійна обробка інформації, донавчання моделі, виходячи з досвіду і метрик при використанні моделі. Запропонована платформа дозволяє розробляти та підтримувати системи штучного інтелекту з використанням великої кількості джерел даних з граничних (*Edge*) пристроїв. Постійний моніторинг стану та топологічна організація розробленої платформи дозволяє значно підвищити відмовостійкість системи та автоматизувати процес масштабування системи відповідно до задачі, при застосуванні *CI/CD* конвеєра, що робить її практично придатною для впровадження в системах, де потрібні високий рівень автономності, гнучкість та адаптивність у роботі з даними.

Особистий внесок здобувача. Дисертаційна робота є самостійним дослідженням автора. Основні результати та наукові положення дисертації отримані шляхом самостійної науково-дослідної роботи. В роботах, які були опубліковані в співавторстві, дисертанту належать: [1] - моделювання

масштабованих систем на основі запропонованої топології мережі та дослідження характеристик ШІ, [2] – моделювання масштабованих систем на основі запропонованої топології та топологічних показників, [3] - моделювання топології та топологічних показників, використання посередності та моделювання мережі з відмовами, [4] – дослідження *Edge* архітектури та навчання штучних нейронних мереж, [5] – робота виконана повністю автором дисертаційного дослідження, [6] – практичне застосування та аналіз використання платформ вбудованих систем для задач ШІ.

Апробація результатів дисертації. Основні результати роботи опубліковано та обговорено на міжнародних та всеукраїнських наукових конференціях, зокрема на: Information, Computing and Intelligent systems: International Conference ICSFTI2022, Information, Computing and Intelligent systems: International Conference ICSFTI2023 та Information, Computing and Intelligent systems: International Conference ICSFTI2024.

Публікації. За результатами дисертаційних досліджень у фахових українських та міжнародних виданнях опубліковано 6 наукових статей, з яких 3 входять до наукометричної бази даних з міжнародним індексом цитування *Scopus*.

Структура і обсяг роботи. Дисертаційна робота складається зі вступу, чотирьох розділів, загальних висновків, списку використаних джерел із 80 найменувань. Загальний обсяг дисертації становить 213 сторінок, з яких 147 сторінок основного тексту, 5 додатків на 33 сторінках, та містить 42 рисунки, 30 формул, 22 таблиці.

РОЗДІЛ 1

ОГЛЯД СУЧАСНИХ ПІДХОДІВ РОЗГОРТАННЯ КОМПОНЕНТ ПЛАТФОРМ ВБУДОВАНИХ СИСТЕМ

1.1. Аналіз сучасних масштабованих платформ вбудованих систем

Не зважаючи на те, що епоха спеціалізованих архітектурних рішень на основі топологічних організацій, для вирішення деяких, конкретних прикладних задач (під конкретний граф задачі) вже пройшла, досі існує необхідність в створенні складних систем, які вирішують задачі пошуку оптимального маршруту та відмовостійкості. В рамках запропонованої вбудованої платформи, необхідно враховувати, що надійність вбудованих систем значно нижча за *Cloud*, який має доступність до 99.9999% [35]. Для вирішення проблем відмовостійкості, запропоновано декілька різних топологічних організацій на основі існуючих рішень запропонованих в ході дослідження в рамках дисертаційної роботи [36-37].

Однією з основних причин, чому необхідно створити архітектуру і топологію системи ІІІ, є забезпечення відмовостійкості, масштабованості та гнучкості. У міру того, як додатки ІІІ ускладнюються і збільшуються в розмірах, важливо мати структуру, яка дозволяє легко розширювати і модифікувати їх. Окрім відмовостійкості, потрібно вирішити задачу створення конвеєра, який буде подавати дані, на конкретний вузол, в якому буде виконуватись процес навчання. Розуміючи взаємозв'язки між різними компонентами, розробники можуть виявити потенційні вузькі місця або сфери для вдосконалення [36]. Це дозволяє їм точно налаштувати систему і підвищити її ефективність, що призводить до кращої продуктивності і швидшого виконання.

Edge архітектура є новою галуззю високопродуктивних обчислень, яка прагне зайняти нішу між локальними вбудованими пристроями і комп'ютерами, та хмарою. *Edge* – це концепція обробки даних і запуску додатків фізично ближче

до джерела, замість того, щоб покладатися на централізовану хмарну інфраструктуру [42]. Це особливо вигідно, коли мова йде про системи *ML*, оскільки дозволяє приймати рішення в режимі реального часу і зменшує затримки. Розгортаючи моделі *ML* на периферії, організації можуть використовувати можливості ШІ в сценаріях, де низька затримка і висока надійність мають вирішальне значення, наприклад, в автономних транспортних засобах, розумних містах або промисловій автоматизації.

Edge архітектура є новою галуззю високопродуктивних обчислень, яка прагне зайняти нішу між локальними вбудованими пристроями і комп'ютерами, та хмарою. *Edge* – це концепція обробки даних і запуску додатків фізично ближче до джерела, замість того, щоб покладатися на централізовану хмарну інфраструктуру [42]. Це особливо вигідно, коли мова йде про системи *ML*, оскільки дозволяє приймати рішення в режимі реального часу і зменшує затримки. Розгортаючи моделі *ML* на периферії, організації можуть використовувати можливості ШІ в сценаріях, де низька затримка і висока надійність мають вирішальне значення, наприклад, в автономних транспортних засобах, розумних містах або промисловій автоматизації.

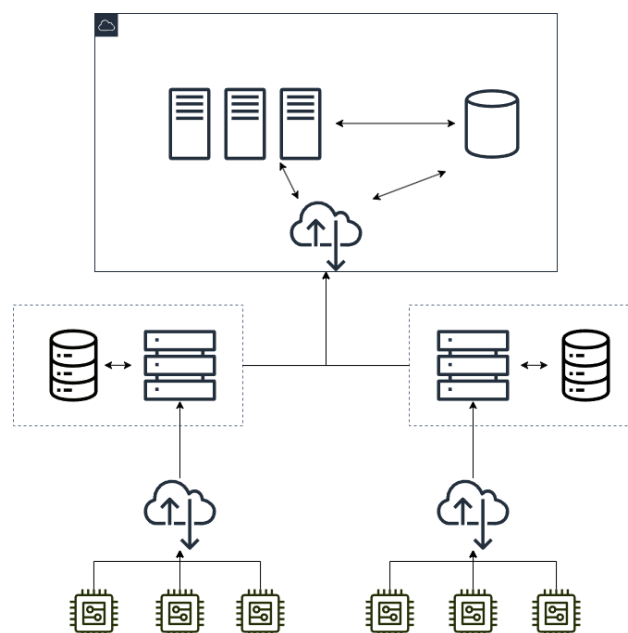


Рис. 1.1. Приклад деревоподібної платформи на основі архітектури *Edge-Cloud*

Іншим завданням може бути використання вбудованих систем для збору даних, та *Edge*-серверів для їх агрегації та попередньої обробки. Вбудовані пристрої генерують величезні обсяги даних, які можна використовувати для навчання і вдосконалення моделей *ML*. Поєднуючи периферійні обчислення з можливостями вбудованих систем, організації можуть обробляти і аналізувати ці дані локально, зменшуючи потребу в передачі великих обсягів даних в хмару для аналізу [43]. Це не тільки зменшує затримки, але й мінімізує вимоги до пропускної здатності, роблячи розгортання ШІ моделей в середовищах з вбудованих систем більш економічно ефективним і масштабованим.

Основними рушійними факторами дослідження є зростаюча популярність і повсюдне поширення пристроїв вбудованих систем, а також забезпечення додаткового рівня конфіденційності та безпеки. Ми вже можемо спостерігати високий ступінь проникнення технології вбудованих систем у різні аспекти нашого життя, лише дивлячись на список доступних пристроїв (*NVIDIA Jetson*, *Raspberry Pi*, смартфон, фітнес-трекери тощо) [44-45]. Можливість розгортання служб на межі дозволяє кінцевим користувачам використовувати відносно дешеві ресурси з додатковими перевагами низької затримки та збору даних, які просто недоступні основним постачальникам *Cloud*.

Впроваджуючи механізми розгортання для задач ШІ, які використовують периферійні обчислення та вбудованих систем, організації можуть гарантувати, що їхні моделі постійно вдосконалюються і надають точні прогнози. Збір даних, розглянутий раніше, має дуже важливе значення для задач машинного навчання, а тому необхідно враховувати особливості цієї архітектурної моделі при побудові конвеєра [43]. *MLOps* доповнює використання периферійних обчислень та Інтернету речей, надаючи необхідні інструменти та практики для управління моделями ШІ протягом усього їхнього життєвого циклу [46]. Це включає автоматизацію процесу навчання, управління версіями моделі, моніторинг

продуктивності моделі та розгортання моделей у відтворюваний і масштабований спосіб.

1.1.1. Основні поняття платформ вбудованих систем

Сучасні тенденції в розвитку вбудованих платформ зумовлені необхідністю інтеграції з новими технологіями, такими як Інтернет речей, штучний інтелект і автономні системи, що вимагають нових підходів до архітектури та розробки таких систем. Для поєднання цих технологій та впровадження конвеєру розгортання задач на вбудовані системи, пропонується застосування платформ вбудованих систем.

Платформи вбудованих систем є основою для реалізації спеціалізованої системи, що виконують функції ШІ в умовах обмежених ресурсів. Під платформою вбудованої системи розуміється апаратно-програмний комплекс, що складається з компонентів, які забезпечують необхідну середу для розробки, тестування та експлуатації додатків в умовах обмежених ресурсів вбудованої системи. Апаратна частина платформи складається з мікропроцесора, периферійних пристроїв, наприклад камер та датчиків, засобів комунікації та елементів керування, що взаємодіють із зовнішнім середовищем. Програмна частина включає операційну систему та програмні інтерфейси реалізовані за допомогою сервісних компонент, що забезпечують взаємодію між апаратними компонентами та програмним забезпеченням. Важливими характеристиками платформ вбудованих систем є низьке споживання енергії, низький рівень витрат пам'яті та обчислювальних ресурсів, а також можливість забезпечення високої надійності та реального часу в обробці даних.

Під компонентою платформи вбудованих систем мається на увазі окремий елемент платформи, що є частиною архітектури платформи та виконує специфічні функції для забезпечення її працездатності. Компоненти можуть бути як апаратними, так і програмними, і разом вони формують цілісну платформу, що

здатна виконувати завдання ІІІ. Апаратні компоненти включають процесорні блоки, мікроконтролери, датчики, виконавчі пристрої (наприклад, моторчики, реле), засоби зберігання даних (флеш-пам'ять, *EEPROM*) та інші периферійні пристрої, які взаємодіють із зовнішнім середовищем або іншими пристроями. В умовах обмеження ресурсів вбудованих систем, це накладає додаткові обмеження з точки зору програмних компонентів. Програмні компоненти включають бібліотеки та інтерфейси, що забезпечують взаємодію між різними модулями та керують обробкою даних і взаємодією з користувачами. Особливістю компонентного підходу до створення платформи є її модульність, що дозволяє замінювати або оновлювати окремі елементи платформи без значних змін у загальній системі, що сприяє відмовостійкості платформи загалом.

Використання часу розгортання платформи як критерію ефективності є важливим аспектом оцінки продуктивності та зручності технологічних систем. Час розгортання визначається як період, необхідний для повного встановлення, налаштування та активації платформи, що забезпечує її функціональність для кінцевих користувачів або організацій. Тривалість розгортання безпосередньо пов'язана з рядом факторів, таких як складність інфраструктури, рівень автоматизації процесів, якість попередньої підготовки та адаптації платформи, а також наявність необхідних ресурсів і кваліфікації персоналу. Цей параметр дозволяє оцінити швидкість впровадження технології в експлуатацію та впливає на загальну ефективність системи, оскільки чим коротший час розгортання, тим швидше система може бути інтегрована в робочі процеси і використовуватись для досягнення бізнес-цілей.

З погляду управління та розробки допоміжного програмного забезпечення для розгортання платформ, зменшення часу розгортання платформи може бути досягнуте через застосування нових методів автоматизації певних процесів у платформі, використання контейнеризації та мікросервісної архітектури. Ці технології значно прискорюють процеси налаштування та масштабування

інфраструктури, водночас зберігаючи високу надійність і знижуючи людський фактор.

1.1.2. Топологічні організації платформ вбудованих систем

Серед видів архітектури розподілених систем повнозв'язні топології представляють собою найбільш поширений спосіб взаємодії між компонентами обчислювальної інфраструктури, де кожен вузол безпосередньо з'єднаний з усіма іншими вузлами мережі. Незважаючи на теоретичну привабливість та практичну простоту налаштування такої мережі, вона демонструє значні обмеження в практичній реалізації інфраструктурних рішень для машинного навчання, що в свою чергу створює необхідність в додаткових ресурсах для використання підходу *MLOps*. Практичні обмеження повнозв'язних топологій особливо відчутні в контексті використання нейронних мереж, де необхідна висока пропускна здатність та мінімальні затримки при обміні проміжними результатами. Альтернативні топологічні рішення, такі як ієрархічні (деревовидні) або *mesh*-мережі (тороїди), демонструють значно вищу ефективність.

Топологія "жирне дерево" - це популярний дизайн для мереж центрів обробки даних, який має на меті забезпечити високу пропускну здатність мережі та низьку затримку. Ця топологія складається з декількох шарів, зазвичай трьох, і будується за допомогою графової структури, відомої як "жирне дерево" [38]. Однією з переваг топології "жирне дерево" є її рекурсивна масштабованість. Це означає, що в міру розширення мережі і додавання нових пристроїв топологія "жирне дерево" може легко задовольнити зростаючий попит без шкоди для продуктивності. Для подальшого підвищення продуктивності топологій жирних дерев дослідники запропонували такі вдосконалення, як циркулююче жирне дерево (*Circulant Fat-Tree, CFT*) [38]. Ця вдосконалена топологія вирішує деякі проблеми, з якими стикаються традиційні топології жирних дерев, і має на меті ще більше оптимізувати продуктивність мережі. Використовуючи ці досягнення

в дизайні топології "жирне дерево", центри обробки даних можуть продовжувати задовольняти зростаючі потреби у високошвидкісному підключенні та надійних мережевих сервісах.

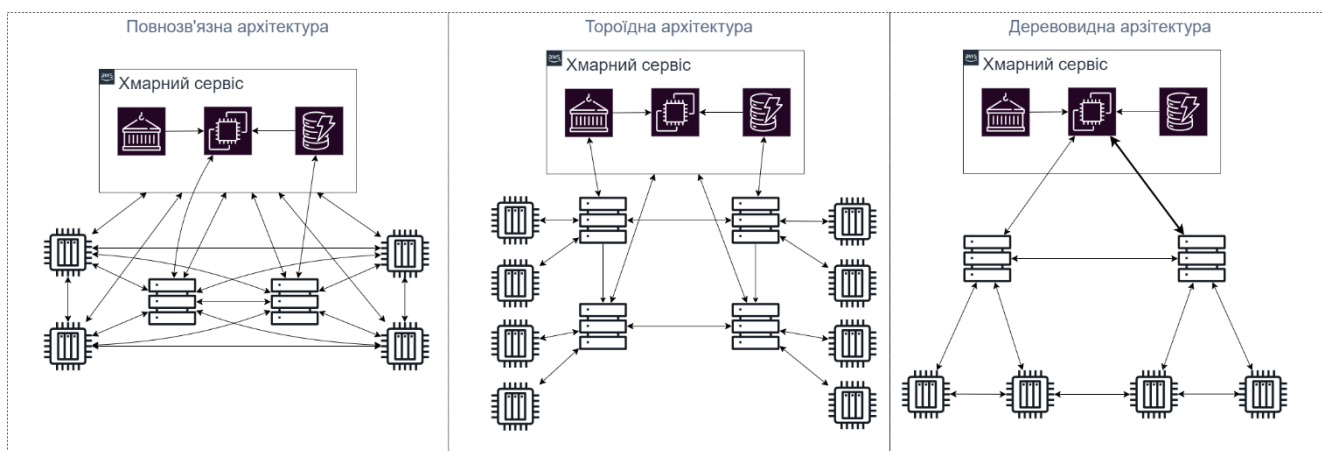


Рис. 1.2. Деякі топології платформ вбудованих систем

У даній статі [39], визначено тенденції використання тороїдних з'єднань в індустрії суперкомп'ютерів, і описано топологію мережі тороїдів (*TCT*) як рішення проблеми вартості мережі. Топологія *TCT* була формально визначена та теоретично та емпірично оцінена як така, що значно знижує вартість мережі порівняно зі звичайними підходами, такими як тороподібні цикли та топологія *Tofu*, яка використовується двома основними суперкомп'ютерами *Fujitsu K* та *Fugaku*, таким чином відкриваючи шлях для подальшого підвищення продуктивності.

1.1.3. Особливості апаратного забезпечення компонент платформ вбудованих систем

Вбудовані пристрої в контексті ШІ-платформ постають як високоінтегровані апаратно-програмні комплекси, що забезпечують принципово новий рівень обчислювальної ефективності. Концептуальною особливістю вбудованих технологій у штучному інтелекті є їх вузькоспеціалізована природа, що дозволяє реалізовувати надскладні обчислювальні завдання з високою точністю та швидкодією. Їхня архітектура характеризується унікальною здатністю

виконувати складні алгоритмічні перетворення з мінімальними апаратними та енергетичними витратами, що виокремлює їх від інших програмно-апаратних засобів. Такі системи принципово відрізняються від традиційних своєю компактністю, контекстною адаптивністю та здатністю до миттєвої реакції в режимі реального часу, завдяки своїй фізичній наближеності до досліджуваного об'єкту.

Архітектурні рішення вбудованих систем в платформах з можливостями ШІ передбачають багаторівневу оптимізацію обчислювальних процесів. Механізмами такої оптимізації на апаратному рівні виступають спеціалізовані процесорні модулі, інтелектуальні мікроконтролери та апаратні акселератори машинного навчання. За допомогою таких підходів, можливе паралельне виконання складних математичних операцій з мінімальними затримками та максимальною енергоефективністю. На програмному рівні, архітектура компонентів у платформах вбудованих систем має передбачати багаторівневу модульність та гнучку інтероперабельність.

Важливим аспектом є механізми інтеграції вбудованих систем у структуру штучного інтелекту. Йдеться про здатність не тільки виконувати обчислення, але й адаптивно реагувати на зміну вхідних параметрів, самонавчатися та оптимізувати власні алгоритми функціонування. Це питання значною мірою вирішується за допомогою підходу *MLOps*, але відкритим залишається питання проміжного рівня – гіпервізора. Заслужують уваги мікроархітектурні рішення, що уможливлюють високу швидкодію нейромережових перетворень. Завдяки своїй апаратній структурі та необхідності в розробці зв'язки з іншими пристроями або серверами, мікроархітектурний підхід є не лише доречним, але й найбільш природнім для вирішення задачі створення плану розгортання та комунікації між пристроями.

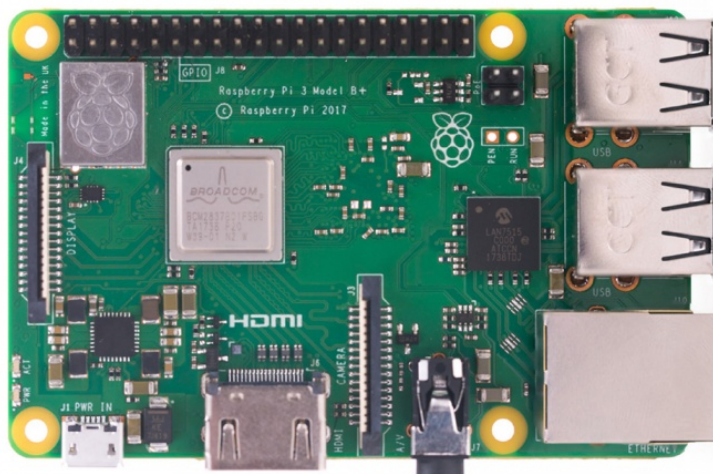


Рис. 1.3. Зображення мікрокомп'ютера Raspberry Pi 3 B+

Raspberry Pi 3 B+ є ефективним пристроєм для розгортання деяких компонентів ШІ-платформи завдяки поєднанню технічних характеристик, достатніх для виконання задач, пов'язаних з ШІ, та низької вартості. Мікрокомп'ютер обладнаний 64-бітним чотирьохядерним процесором *Cortex-A53* з тактовою частотою 1.4 ГГц, що забезпечує обчислювальну потужність для реалізації відносно нескладних нейромережових моделей та алгоритмів машинного навчання. Вбудована графічна система *VideoCore IV* дозволяє ефективно здійснювати обробку відеопотоків та графічних даних, а наявність 1 ГБ оперативної пам'яті *RAM* уможливорює запуск полегшених дистрибутивів штучного інтелекту, таких як *TensorFlow Lite* та *PyTorch*. Комунікаційні можливості впроваджені за допомогою низки інтерфейсів, таких як *Wi-Fi*, *Ethernet* та *Bluetooth*, а також наявність GPIO-роз'ємів для підключення периферійних пристроїв, що робить *Raspberry Pi 3 B+* універсальною платформою для прототипування та експериментальних досліджень в галузі вбудованого штучного інтелекту.

Емпіричні дослідження підтверджують, що впровадження та використання практик *MLOps* в вбудовані системи дозволяє отримати ефективну альтернативну, продуктивну та низьковартісну платформу порівняно з традиційними комп'ютерними архітектурами. Особливо виразно це проявляється у таких

напрямках, як комп'ютерне бачення, обробка потоку зображень та в автономних системах для захисту мережі. Архітектурні рішення розглянуті в літературі та сучасних рішеннях передбачають багаторівневий захист інформаційних потоків, криптографічний захист обчислювальних контурів та інтелектуальні механізми превентивного виявлення потенційних загроз.

1.2. Порівняльний аналіз підходів до розгортання сучасних платформ вбудованих систем

У світлі стрімкого розвитку галузі машинного навчання та штучного інтелекту, впровадження та управління моделями стає вирішальним елементом для підприємств у сучасному цифровому ландшафті. Концепція *MLOps*, або управління життєвим циклом моделей машинного навчання, виходить за межі простого розробки моделей, включаючи в себе автоматизацію, моніторинг та масштабування для забезпечення ефективності та надійності виробничих систем [4]. В наукових дослідженнях та статтях з даної теми фахівці вдосконалюють стратегії *MLOps*, спрямовані на оптимізацію життєвого циклу моделей, щоб вони відповідали вимогам швидкозмінюваного та конкурентного інформаційного середовища. Аналіз сучасних рішень та досліджень у цій області відкриває новаторські підходи до управління моделями машинного навчання, що забезпечують не тільки прогресивні технічні рішення, а й стратегії для підтримки стійкості та розвитку в контексті великих обсягів даних та вимог виробничих процесів.

1.2.1. Проблематика розгортання MLOps

Традиційно, розробка програмного забезпечення та його впровадження відбуваються відокремлено. Розробники створюють програмне забезпечення, а оператори займаються його впровадженням та підтримкою. Цей підхід призводить до недоліків, таких як відсутність спілкування, порушення вимог до забезпечення якості та затримки в часі [8].

Концепція *DevOps* сформувалась на межі 2008–2009 років як відповідь на необхідність тіснішої інтеграції процесів розробки програмного забезпечення та його операційного забезпечення, щоб забезпечити безперервну комунікацію та співпрацю між розробниками та операторами. *DevOps* — це більше, ніж чиста методологія, а радше представляє собою парадигму вирішення соціальних і технічних проблем в організаціях, які займаються розробкою програмного забезпечення [8-10].

Один з перших кроків до появи *DevOps* було впровадження методології *Agile*, яка спрямована на швидку зміну та вдосконалення. *Agile* дозволив розробникам швидко реагувати на змінні вимоги та змінювати програмне забезпечення в процесі розробки.

Згодом, з'явилися інші практики та методології, такі як *Continuous Integration (CI)* і *Continuous Delivery (CD)*. *CI* дозволяє розробникам інтегрувати свій код в основну гілку проекту регулярно, щоб виявляти й вирішувати конфлікти та проблеми з розробкою. *CD* надає можливість автоматичного впровадження програмного забезпечення в середовище виробництва за допомогою автоматизованих процесів [11].

Як показують емпіричні результати, *DevOps* забезпечує кращу якість реалізації програмного забезпечення [15]. Люди в галузі, а також науковці отримали багатий досвід розробки програмного забезпечення за допомогою *DevOps*. Зараз цей досвід використовується для автоматизації та введення в дію *ML*. Але, необхідно розуміти ряд відмінностей між програмним продуктом та ШІ-моделлю. Платформа на основі ШІ — це програмна система, тому традиційні методи застосовуються, щоб гарантувати, що кінцевий користувач може надійно створювати та використовувати системи ШІ у великих масштабах [16-17]. Однак системи ШІ мають відмінності від інших програмних систем за наступними етапами:

Командна взаємодія: у проєкті ШІ команда зазвичай включає вчених з обробки даних або дослідників ШІ, які зосереджуються на дослідницькому аналізі даних, розробці моделі та експериментуванні. Ці учасники можуть не бути досвідченими інженерами програмного забезпечення, які можуть створювати сервіси виробничого класу. Це може призвести до проблем з масштабуванням системи, її відмовостійкістю та продуктивністю [17-19].

Розробка: ШІ має експериментальний характер. Вчений має спробувати різні функції, алгоритми, методи моделювання та конфігурації параметрів, щоб якомога швидше знайти те, що найкраще підходить для вирішення проблеми [16-17]. Завдання полягає в тому, щоб відстежувати, що спрацювало, які показники такого алгоритму, який час займає навчання такої моделі, а що не спрацювало, або зіпсувало якість моделі, і підтримувати відтворюваність при максимальному повторному використанні коду.

Тестування: тестування системи ШІ є більш складною, ніж тестування інших програмних систем. На додаток до типових модульних та інтеграційних тестів розробнику потрібно пройти кроки перевірки даних, оцінки якості моделі, яка пройшла навчання, і перевірки моделі [16, 20].

Розгортання: у системах ШІ розгортання не таке просте, як, наприклад, розгортання офлайн-навченої моделі *ML* в якості служби прогнозування погоди. Системи ШІ можуть вимагати розгортання багатоетапного конвеєра для автоматичного перенавчання та розгортання моделі. Цей конвеєр додає складності та вимагає від вас автоматизації кроків, які виконуються вручну перед розгортанням спеціалістами з обробки даних для навчання та перевірки нових моделей [16, 21].

Випуск (*production*): моделі ШІ можуть мати знижену продуктивність не лише через неоптимальне кодування, але й через постійну еволюцію профілів даних. Іншими словами, моделі можуть занепадати більшою мірою, ніж звичайні програмні системи, і потрібно враховувати це погіршення [16, 21]. Тому

необхідно відстежувати підсумкову статистику вхідних та вихідних даних і відстежувати онлайн-продуктивність моделі, щоб надсилати сповіщення або повертатися, коли значення відхиляються від ваших очікувань.

Виходячи з вищенаведених проблем, необхідно мати рішення щодо впровадження процесу, який би включав особливості сучасних платформ та конфігурацію і розгортання її окремо виділених компонент. Описані вимоги та критерії поєднані в процес, який має назву операції машинного навчання.

1.2.2. Практики розгортання MLOps в хмарних системах

Не зважаючи на відносну новизну, практики *MLOps* вже знаходять свою імплементацію в сучасних інженерно-наукових рішеннях в *Cloud*-платформах, таких як *AWS*. Для розуміння стадії розвитку даної технології, є сенс подивитись на її реалізацію на практиці в подібних системах.

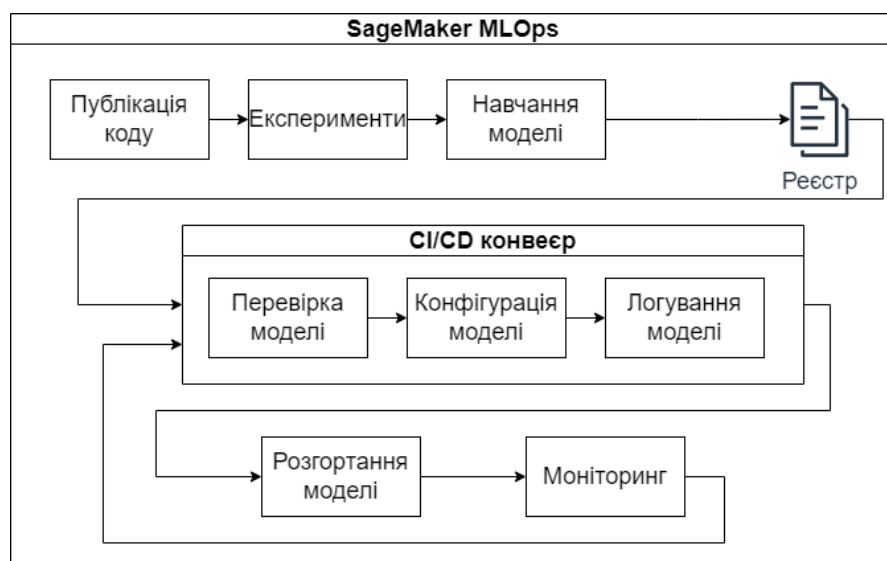


Рис. 1.4. Схема системи *AWS SageMaker Pipelines*

Перша розглянута система це *AWS SageMaker Pipelines*. *SageMaker Pipelines* — це перша спеціально створена служба безперервної інтеграції та безперервної доставки (*CI/CD*) для *ML*. За допомогою *SageMaker Pipelines* ви можете автоматизувати різні етапи робочого процесу машинного навчання, включаючи завантаження даних, перетворення даних, навчання, налаштування, оцінку та

розгортання. Реєстр моделей *SageMaker* дозволяє відстежувати версії моделей, їхні метадані, такі як групування випадків використання, і базові показники продуктивності моделі в центральному сховищі, де легко вибрати правильну модель для розгортання на основі ваших бізнес-вимог. *SageMaker Clarify* забезпечує кращий огляд ваших тренувальних даних і моделей, щоб ви могли визначити й обмежити упередження та пояснити прогнози. Дана технологія входить в стек технології *SageMaker*, які розроблені спеціально для розробки ШІ, інтеграція між якими може зробити простішим процес дослідження моделей. Однією з конкурентних переваг *SageMaker MLOps* є його здатність до масштабування, способом оперативно адаптувати ресурси для навчання та впровадження моделей відповідно до змінюваних вимог навантаження. Інтеграція з іншими сервісами *AWS*, такими як *S3*, *Lambda*, *Glue* та *Athena*, забезпечує додаткову гнучкість у побудові рішень, що дозволяє командам ефективно використовувати екосистему *AWS*.

SageMaker кодує конвеєр у спрямованому ациклічному графі (*DAG*), де кожен вузол представляє крок, а з'єднання між вузлами представляють залежності. Щоб перевірити конвеєр *DAG* з інтерфейсу *SageMaker Studio*, виберіть вкладку *SageMaker Resources* на лівій панелі, виберіть *Pipelines* зі спадного списку, а потім виберіть *FraudDetectXGBPipeline, Graph*. Ви бачите, що створені вами кроки конвеєра представлені вузлами на графіку, а зв'язки між вузлами були визначені *SageMaker* на основі вхідних і вихідних даних, наданих у визначеннях кроків.

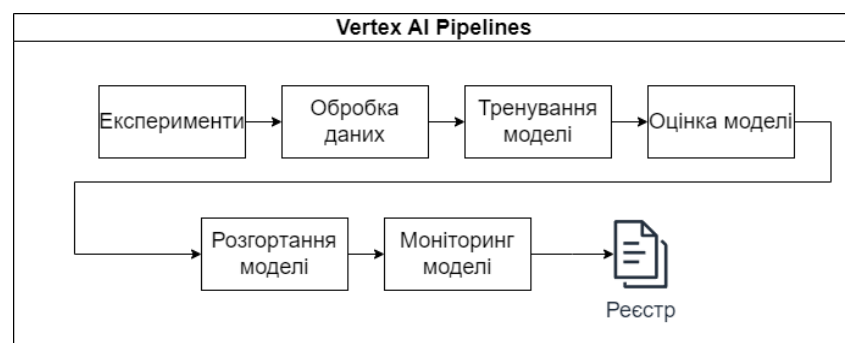


Рис. 1.5. Схема системи *Vertex AI*

Реалізація від *Google Cloud* відрізняється від *Amazon* тим, що вона більш розгалужена і спеціалізована на саме навчанні моделей. Один з сервісів, який імплементує цю технологію – це *Vertex AI*. *Vertex AI* — це платформа машинного навчання, яка дозволяє навчати та розгортати моделі *ML* і програми ШІ. *Vertex AI* поєднує в собі робочі процеси обробки даних, науки про дані та розробки машинного навчання, забезпечуючи командну співпрацю за допомогою спільного набору інструментів. Важливим компонентом є *Vertex AI*, який об'єднує різні функції для *MLOps*, включаючи автоматизоване навчання моделей та управління даними. *Vertex AI* надає можливості для імплементації *AutoML*, що спрощує процес налаштування моделей. Крім того, для обробки даних у реальному часі *GCP* пропонує сервіс *Dataflow*, який дозволяє створювати процеси *ETL* (*Extract, Transform, Load*), інтегровані з моделями машинного навчання.

Зберігання великих обсягів даних забезпечує *Cloud Storage*, що підтримує версіонування даних, що особливо важливо для управління різними версіями наборів даних. Сервіс *BigQuery*, у свою чергу, забезпечує можливість виконання *SQL*-запитів для аналізу даних, що можуть бути використані для навчання моделей. Для автоматизації процесів, таких як тригери для запуску моделей або обробки даних, *GCP* пропонує *Cloud Functions*, що використовують безсерверні функції. Інтеграція з інструментами *CI/CD*, такими як *Cloud Build* і *GitHub*, дозволяє автоматизувати процеси розгортання моделей і управління їх версіями. Моніторинг продуктивності моделей реалізується через *Google Cloud Monitoring* і *Logging*, що дозволяє відстежувати їх використання та виявляти аномалії. Окрім того, важливим аспектом є безпека, яку *GCP* забезпечує через механізми контролю доступу на основі ролей (*IAM*), що критично важливо для захисту даних і моделей.

1.2.3. Практики розгортання MLOps в вбудованих системах

Традиційний *ML* на краю також має деякі обмеження. Наприклад, машинне навчання моделі часто навчаються централізовано та розгортаються вручну та окремо на *Edge* вузлах. Цей процес розробки, розгортання та моніторингу моделей машинного навчання потрібно автоматизувати, щоб мати масштабованість. Спираючись на досвід побудови широкомасштабних конвеєрів машинного навчання в реальному світі з використанням операцій машинного навчання (*MLOps*) в статтях нову автоматизовану структуру для операцій машинного навчання на межі (*Edge MLOps*).

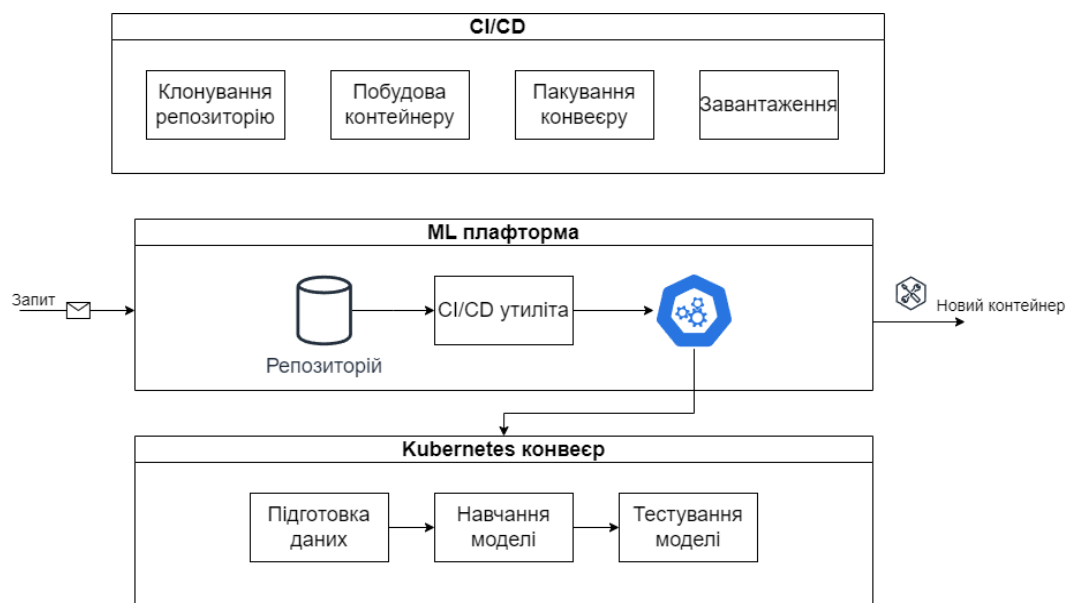


Рис. 1.6. Схема системи *Edge MLOps*

В статті [43] запропонований фреймворк, який фокусується на розробці конвеєру для вбудованих пристроїв. Структура фреймворку забезпечує виконання вищевказаних вимог і функцій модульним способом. На високому рівні існує два основних шари.

1) Рівень *Cloud Orchestration*, який працює на хмарній платформі, призначений для навчання моделям *ML*, перепідготовки, завдань *CI/CD*, централізованого зберігання даних, керування вихідним кодом, панелі аналітики, а також управління даними та моделлю.

2) Рівень *Edge Inference* призначений для оркестрування висновків *ML* і операцій для вбудованих пристроїв, таких як безперервна інтеграція та потокова передача даних із вбудованих пристроїв.

Перевагами такої структури є високий рівень роз'єднання через модульність між двома рівнями та використання сильних сторін обох платформ. Відмовостійкість такої системи була перевірена шляхом її реалізації. З 5-хвилинним інтервалом вбудовані пристрої без збоїв надсилали дані на периферійні пристрої. Протягом 45 днів експерименту тригер часу запускався щодня о 12:00 для моніторингу дрейфу моделі кожної розгорнутої моделі. Загалом 135 тригерів по часу було успішно виконано без жодних збоїв. На додаток до них, 27 ручних тригерів були виконані випадковим чином без будь-яких збоїв. На хмарному рівні 23 моделі *ML* були успішно перенавчені, коли їх продуктивність була знижена [43].

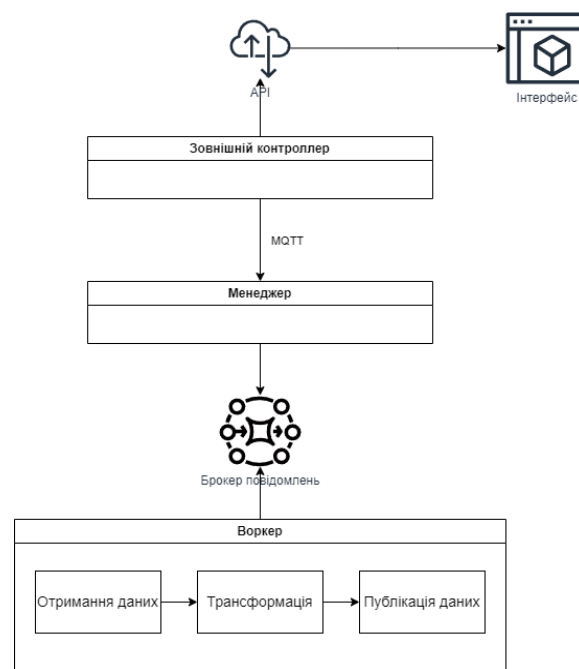


Рис. 1.7. Схема системи ERAIA

В іншій статі [51], запропонований підхід, на основі для вирішення задачі *MLOps* на *Edge*. ERAIA є реактивною системою і тому успадковує її характеристики: швидка відповідь, стійкість, еластичність і керованість

повідомленнями. Створений з використанням акторської моделі, він надає децентралізовану систему керування для розгортання високопродуктивних розподілених обчислень, які можна ефективно масштабувати за допомогою ресурсів і масштабувати за допомогою додавання нових вузлів [51]. На рисунку 1.7. зображено архітектуру рішення, розділене на чотири категорії компонентів:

Зовнішній компонент надає *API* для створення/декомпозиції конвеєра розвідки та даних на основі *ERAIA*, а також дозволяє зовнішнім програмам взаємодіяти з інфраструктурою [51].

Менеджер кластера створює і керує відмовостійким одноранговим кластером. Цей компонент бере на себе відповідальність за розподіл і розміщення робочих навантажень за допомогою планувальника та/або політик на основі розташування.

Робочий (*worker*) — це компонент, який виконує елементи конвеєра, надані через зовнішній *API*. Це компонент, відповідальний за площину даних, яка встановлює з'єднання з пристроями та/або посередниками повідомлень.

Інтерфейс користувача полегшує візуалізацію та керування системою, що відкривається через *API*. *ERAIA* [51] надає простий *API*, який дозволяє створювати конвеєри керування даними з підтримкою вилучення інформації та розвідувальних даних у реальному часі. Він пропонує можливість розподіляти обчислення на основі конкретних вимог додатків для вбудованих систем, таких як енергоспоживання, пропускна здатність, затримка, доступність обчислювальної потужності або спеціального обладнання, доступність джерела даних, безпека та конфіденційність даних тощо. Рішення інтегрує компоненти з відкритим кодом і розширює їх функціональність, дозволяючи виконання в акторах.

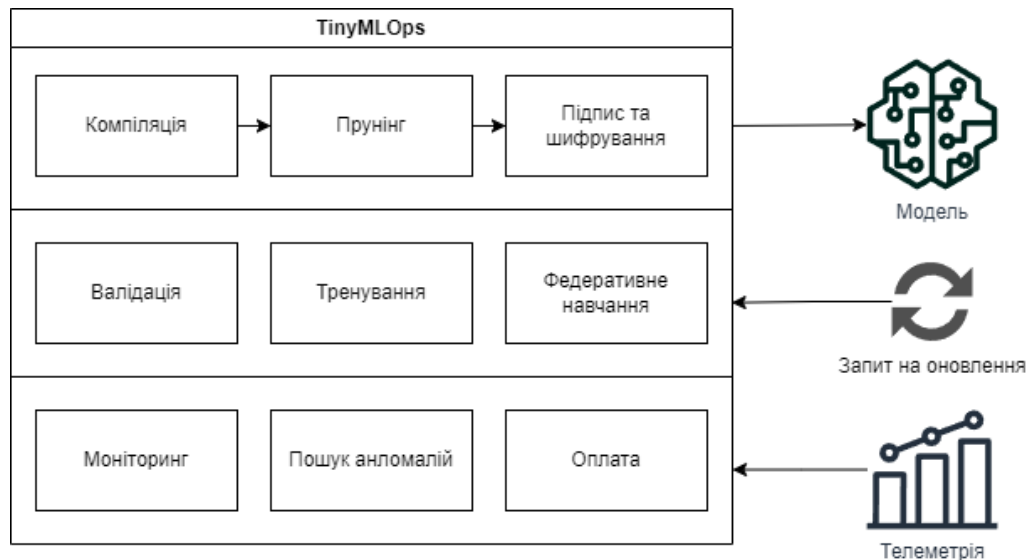


Рис. 1.8. Схема системи *Tiny ML Ops*

Ще одна модель, *TinyML* [52], стосується використання моделей машинного навчання на периферійних пристроях з обмеженими ресурсами, є привабливою парадигмою розгортання для різноманітних програм, таких як розумна побутова техніка, віртуальні помічники, автономні транспортні засоби та інтелектуальне спостереження. Для цих програм периферійне розгортання може забезпечити меншу затримку, підвищену надійність, масштабованість і конфіденційність порівняно зі сценаріями розгортання, де модель оцінюється в хмарній інфраструктурі. Більшість робіт у сфері *TinyML* зосереджено на покращенні ефективності моделей на пристроях з обмеженими ресурсами [52].

Незважаючи на коротку історію *TinyML*, уже існує величезна кількість літератури про підходи, які зменшують обчислювальні витрати, обсяг пам'яті та енергоспоживання моделей *ML*, націлених на периферійне розгортання. Ці методи включають скорочення, квантування моделі, дистиляцію знань, адаптивне обчислення або автоматизований пошук нейронної архітектури. Деякі з цих методів можуть повністю розкрити свій потенціал лише за умови підтримки апаратної платформи, на якій вони розгорнуті. Таким чином, існує величезна можливість для апаратно-програмного спільного проектування для досягнення максимально можливої ефективності [52]. Окрім оптимізації самої моделі, виконуються додаткові кроки по обробці даних. Багатообіцяючим підходом є

використання *WebAssembly* [53] для упаковки цих різних операцій у портативні та повторно використовувані модулі. На стадії випуску системи, існує кілька доступних рішень для спостереження *ML*, які зазвичай відстежують розподіл вхідних значень для виявлення дрейфу даних. Це дозволяє інженерам машинного навчання виявляти погіршення продуктивності моделі на ранній стадії.

Серед недоліків такої системи є спрощений спосіб оновлення моделі, який враховує можливості прунінгу, без чітко прописаного процесу розгортання його в системі. Такий підхід регламентує спосіб, який можна використовувати в *Cloud* системах, але передбачає лише лімітоване використання вбудованих пристроїв, для виконання найпростіших функцій. Дослідники [52] припускають, що запропонована їми модель знаходиться на ранньому етапі як і існуючі платформи та програмні засоби, тому логічним є подальше дослідження та масштабування.

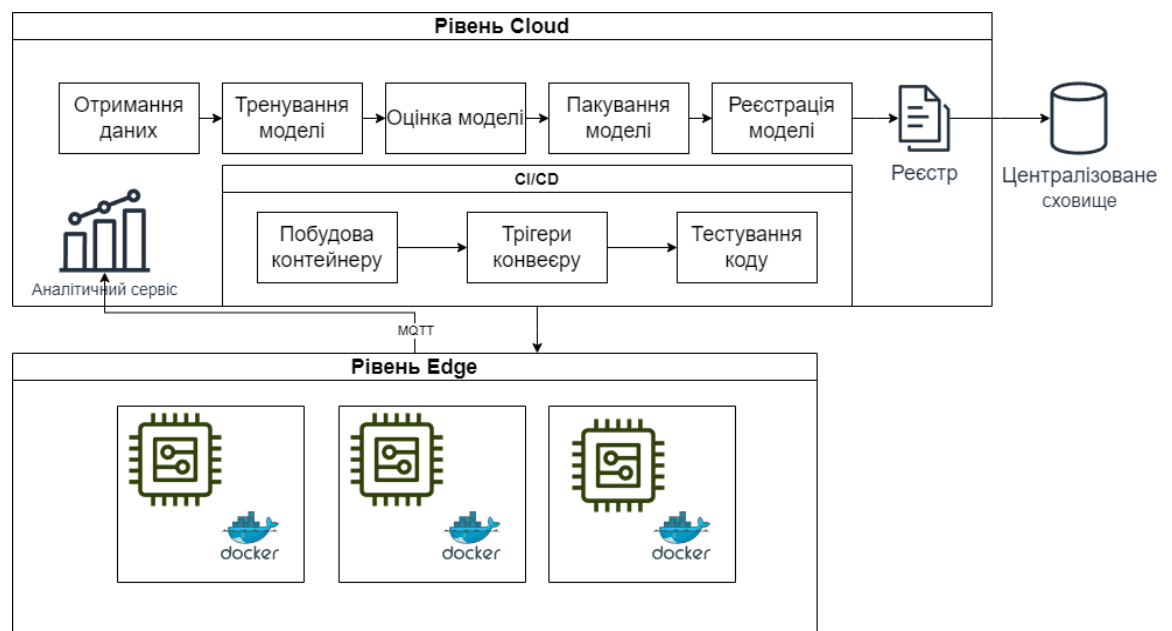


Рис. 1.9. Схема системи мікросервісного *Edge MLOps*

В останній статті пропонується модель *Edge MLOps* на основі мікросервісної архітектури. Навчена модель пропонує два способи надання результатів логічного висновку у відповідь на запити даних через сервер *API* у хмарному середовищі або функціонує в *Edge* середовищі як сервіс [54]. Зазначений метод можна вибрати відповідно до обчислювальних ресурсів, доступних у крайовому середовищі.

В якості *Edge* середовища, використовується шлюз, який був розроблений на базі *NVIDIA Jetson Xavier NX* з 48 тензорними ядрами для прискорення алгоритмів машинного навчання для аналізу даних [55], а сервери хмарного рівня з процесорами *Xeon* і подвійними графічними процесорами *NVIDIA A40* були використані для виконання ролі середовища хмарних обчислень.

Для перевірки платформи *MLOps*, розробленої та реалізованої для розробки та розгортання служб штучного інтелекту, що працюють у *Edge-Cloud* середовищі, використовується модель прогнозування даних часових рядів на основі глибокого навчання. Ця модель перевіряє дані датчиків, зібрані на межі, щоб оцінити стан стандартної роботи, одночасно виявляючи ненормальні дані за допомогою прогнозування даних, а потім швидко вживаючи заходів [54].

Що стосується програмного забезпечення, в статті використовується різноманітне програмне забезпечення з відкритим кодом для застосування платформи *MLOps*. В основу закладений *Kubeflow*, платформа штучного інтелекту, побудовану на *Kubernetes*, яка слугує інструментом оркестровки контейнерів, що використовується для ефективного керування програмним забезпеченням периферійних обчислень [56].

Таблиця 1.1

Порівняльний аналіз розглянутих рішень

Модель	Переваги	Недоліки
<i>SageMaker MLOps</i>	Інтеграція з <i>Cloud</i> рішеннями <i>AWS</i> , автоматичний менеджмент ресурсів	Немає підтримки вбудованих пристроїв та інших сторонніх пристроїв, обмежений набір функціоналу, несумісність зі сторонніми <i>CI/CD</i> рішеннями

Модель	Переваги	Недоліки
<i>Edge MLOps</i>	Імплементований підхід <i>MLOps</i> на вбудованих пристроях. Більш гнучкий в підборі апаратного забезпечення порівняно з <i>Cloud</i> -рішеннями.	Складна система зв'язку рівня вбудованих пристроїв із хмарою. Висока кількість помилок, які можуть призвести до відмов у реальних умовах. Низька швидкість розгортання.
<i>ERAIA</i>	Дозволяє вносити зміни за допомогою користувацького інтерфейсу. Додаткова відмостійкість за рахунок розділення менеджера кластера та зовнішнього керування.	Контролер має представлення воркерів як одного пристрою. Обробка помилок та відновлення ручне. Обмежене масштабування, як результат використання менеджера кластера.
<i>TinyML</i>	Навчання моделей виконується на рівні вбудованих пристроїв. Широко використовуються прийоми прунінгу, зокрема квантизація та автоматизований пошук ШНМ.	Лімітована кількість пристроїв здатна виконувати задачі навчання. Неописаний процес розгортання системи. Обмежена масштабованість за рахунок невизначеності архітектури системи.

Модель	Переваги	Недоліки
Мікросервісний <i>Edge MLOps</i>	Широка інтеграція <i>Cloud</i> з <i>Edge</i> рівнем. Надача <i>API</i> для використання системи ззовні. Використання мікросервісного підходу на <i>Edge</i> рівні ефективно використовує можливості вбудованих пристроїв. Можливість масштабування є найкращою серед інших варіантів.	Досить складний процес розгортання є затратним у часі. <i>Cloud</i> ресурси є високовартісними і потребують постійного моніторингу.

Дослідження виявило комплексні особливості різних моделей MLOps, які характеризуються специфічними архітектурними та функціональними обмеженнями. Кожна з проаналізованих платформ має унікальні переваги та притаманні їм недоліки, що впливає на ефективність впровадження машинного навчання в операційні процеси. *Cloud*-орієнтовані рішення, зокрема *SageMaker MLOps* та *GCP Vertex AI MLOps*, демонструють високий рівень інтеграції з відповідними хмарними екосистемами, проте мають суттєві обмеження щодо підтримки вбудованих пристроїв та сторонніх систем безперервної інтеграції та доставки. Мікросервісний *Edge MLOps* виокремлюється найбільш збалансованою архітектурою, що забезпечує широку інтеграцію *Cloud* та *Edge* рівнів, але вимагає значних часових та фінансових інвестицій для впровадження.

1.3. Особливості процесу розгортання компонентів платформ вбудованих систем

Створення надійної та ефективної інфраструктури для додатків штучного інтелекту має вирішальне значення як для організацій так і для груп дослідників, які прагнуть використовувати можливості ШІ ефективно. Ця інфраструктура

передбачає інтеграцію різних технологій і процесів, зокрема *MLOps*, щоб забезпечити безперебійну розробку, розгортання та управління моделями штучного інтелекту. *MLOps*, що розшифровується як *Machine Learning Operations*, зосереджується на застосуванні принципів *DevOps* до життєвого циклу машинного навчання. Поєднуючи практики *MLOps* і *DevOps*, організації можуть створити конвеєр (pipeline), який забезпечує безперервну інтеграцію, доставку та моніторинг додатків ШІ [1-3].

MLOps, або штучний інтелект для *IT*-операцій - це галузь, що швидко розвивається, яка поєднує штучний інтелект і методи машинного навчання з *IT*-операціями для підвищення ефективності та результативності *IT*-послуг. *MLOps* призначений для автоматизації та оптимізації різних завдань *IT*-операцій, таких як моніторинг, управління подіями, управління інцидентами та аналіз першопричин. Він використовує аналітику та алгоритми для аналізу великих обсягів даних з різних джерел, як логи, метрики та інструменти моніторингу, в режимі реального часу [4]. Таким чином, дана методологія допомагає організаціям покращити свої *IT*-операції, зменшити час простою та забезпечити краще надання послуг.

Актуальність розвитку науки в даній області, зумовлюють впровадження автоматизованих систем розгортання, зокрема є зростаюча складність цифрової інфраструктури та збільшення обсягу даних, що генеруються різними *IT*-системами. Традиційні інструменти управління інфраструктурними операціями не справляються з такою складністю та обсягом даних, що призводить до неефективності та затримок у вирішенні проблем. *MLOps* пропонується як інструмент вирішення даної проблеми, який дозволяє організаціям проактивно виявляти та вирішувати проблеми до того, як вони вплинуть на операції.

Одним з ключових компонентів побудови інфраструктури для додатків ШІ є створення надійної структури *MLOps*. Це передбачає впровадження процесів та інструментів, які забезпечують безперебійний потік даних і моделей на різних

етапах життєвого циклу машинного навчання. *MLOps* допомагає автоматизувати такі завдання, як збір даних, попередня обробка, навчання моделей і розгортання, що призводить до прискорення циклів розробки та підвищення ефективності. Впроваджуючи дану методологію, організації можуть гарантувати, що їхні програми штучного інтелекту побудовані на надійних і масштабованих фреймворках [5].

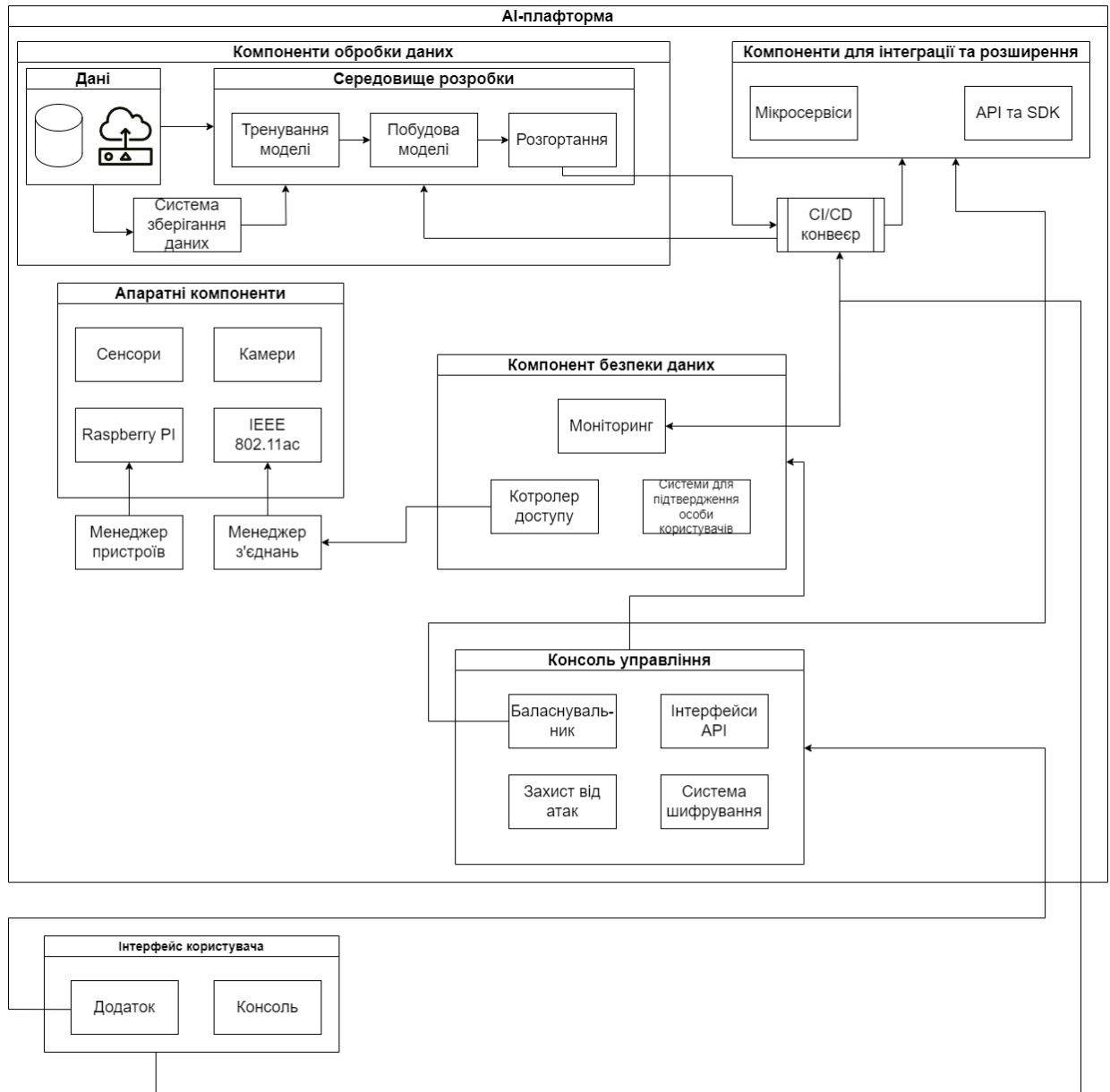


Рис. 1.10. Узагальнена схема взаємодії компонентів платформи на основі *MLOps* підходу

Незважаючи на свій багатообіцяючий потенціал та ріст популярності в останній роки, впровадження *MLOps* не позбавлене проблем. Однією з головних

проблем є інтеграція інструментів *MLOps* з існуючою *IT*-інфраструктурою та системами [4]. Багато організацій мають складні *IT*-середовища з безліччю застарілих систем та інструментів, які можуть бути несумісними з рішеннями *MLOps*. Ці середовища, як правило, складаються із компонентів, які дуже тісно пов'язані між собою і розв'язати питання *decoupling* дуже складно. Крім того, при використанні алгоритмів штучного інтелекту для аналізу конфіденційних *IT*-даних можуть виникати проблеми з конфіденційністю та безпекою даних.

Щоб ефективно побудувати інфраструктуру для додатків ШІ, необхідно враховувати різні фактори. Це включає вибір правильних інструментів і технологій, які підтримують практики *MLOps*. Наприклад, організації можуть використовувати хмарні платформи, такі як *Amazon Web Services (AWS)* [6] або *Microsoft Azure* [7], щоб створити масштабовану і гнучку інфраструктуру для запуску робочих навантажень ШІ. Ці платформи надають широкий спектр послуг, включаючи зберігання даних, обчислювальні ресурси та фреймворки машинного навчання, які можуть покращити розробку та розгортання додатків ШІ. Вибір правильних інструментів і технологій також має важливе значення для побудови масштабованої та гнучкої інфраструктури для робочих навантажень ШІ. Маючи добре продуману інфраструктуру, організації можуть використовувати весь потенціал штучного інтелекту в своїх додатках.

1.3.1. Багаторівнева практика розгортання *MLOps*

Значна частина наукової та інженерної літератури пропонує трьохрівневу модель *MLOps*, виходячи з рівня впровадження *MLOps* в платформі ШІ. *Google* пропонує три рівні зрілості. В першому (0) рівні, всі процеси проходять вручну, другий (1) рівень додає автоматичне тренування моделі і останній (2) рівень додає *CI/CD* конвеєр до системи. Розподіл на декілька рівнів дозволяє покроково розроблювати систему, особливо для вже існуючих рішень [47].

Рівень 0 називається процесом створення та розгортання моделей машинного навчання повністю вручну (а саме без *MLOps*) [49]. Це найпростіший рівень зрілості, і він поширений у багатьох проєктах, які починають застосовувати машинне навчання у своїх сценаріях використання. Цей ручний підхід може бути достатнім, коли моделі рідко змінюються або навчаються; на практиці моделі часто потребують доопрацювання, коли їх розгортають у реальному світі. За допомогою цього підходу виконання кожного кроку є ручним: включаючи аналіз даних, підготовку даних, навчання моделі та перевірку, а також перехід від одного кроку до іншого. Машинне навчання та операції повністю відключені, і дослідники даних передають навчену модель як артефакт групі інженерів для її розгортання. Ітерації випуску нечасті, і через тривалий час, щоб отримати модель у виробництві, можуть виникнути проблеми, пов'язані з перекосом у навчанні [48]. Безперервна інтеграція та безперервна доставка не прийняті, оскільки передбачається небагато змін. Цей підхід також може призвести до відсутності активного моніторингу продуктивності.

Рівень 1: автоматизація за допомогою створення конвеєра машинного навчання [49]. Основною метою рівня 1 є автоматизація процесу машинного навчання для безперервного навчання моделі. Оскільки процедури навчання та розробки моделі організовані, цей підхід дозволяє проводити швидкі випробування. Враховуючи що переходи між різними процесами автоматизовані, експерименти можна проводити швидко. Використовуючи нові дані на основі тригерів конвеєра в реальному часі (нові дані, що надходять, можуть почати новий запуск конвеєра навчання або також можна використовувати заплановані тригери), безперервне навчання дозволяє автоматично навчати модель у виробництві [48]. Симетрія між експериментальним і робочим середовищами — реалізація конвеєра, що використовується в середовищі розробки чи експерименту, також використовується в середовищі підготовки та виробництва — є ще одним важливим компонентом цього рівня автоматизації. Це важливий

компонент навчання *MLOps*. Компоненти для конвеєрів машинного навчання мають бути багаторазовими, модульними та, можливо, доступними для спільного використання в конвеєрі (або конвеєрах). На цьому етапі автоматизації модульність стає важливою, оскільки вона дає змогу відокремити середовище виконання для користувацького коду від середовища виконання та ізолювати окремі компоненти конвеєра, кожен з яких може бути реалізований за допомогою іншої мови та бібліотеки та мати власне середовище виконання. навколишнє середовище [50]. Навчена модель використовується як служба прогнозування на рівні 0; на рівні 1.

На рівні 2 *MLOps* вирішуються більш складні проблеми, пов'язані з управлінням життєвим циклом моделей машинного навчання (*MLOps*) [49]. Головною ідеєю на другому рівні є те, що система переходить до більшої ступені автоматизації та оптимізації, розширює свою інфраструктуру та вдосконалює процеси для ефективного впровадження та управління моделями [48, 50]. На цьому етапі важливо впроваджувати моніторинг у реальному часі, який дає можливість визначати витрати та продуктивність моделей виробництва. Для досягнення цього використовують автоматичні системи моніторингу та повідомлення, які можуть реагувати на непередбачені аномалії або зміни в середовищі даних.

Також, важливо мати систему автоматизованого управління версіями моделей, що дозволяє ефективно керувати розвитком архітектури, гіперпараметрів та інших аспектів моделей, що називається *CT* (*Continuous Training*) [21]. Це сприяє підтримці консистентності та можливості відкату до попередніх версій у випадку необхідності. Застосовуються стандартизовані процеси розробки для забезпечення єдиної методології та сприяння збільшенню ефективності команди. Поряд з цим, розглядаються питання оптимізації витрат, створення відмовостійкої архітектури із широким залученням усіх функціональних компонентів системи, використовуючи ресурси обчислювальних

платформ ефективно та економно [50]. Науковий підхід на рівні 2 *MLOps* полягає у впровадженні технологічних інновацій для автоматизації та оптимізації процесів, що покращує стабільність та продуктивність систем управління моделями машинного навчання в реальному виробничому середовищі.

1.3.2. Конвеєр даних при розгортанні платформ вбудованих систем

MLOps, або Операції машинного навчання - це парадигма, яка фокусується на ефективному та надійному розгортанні та підтримці моделей машинного навчання у виробництві. Вона поєднує в собі практики зі сфер машинного навчання, *DevOps* та інженерії даних, щоб оптимізувати весь життєвий цикл проектів машинного навчання – від інтеграції до розгортання та моніторингу [22]. Основна мета даного підходу – підвищити рівень автоматизації та покращити якість виробничих моделей з урахуванням бізнес та регуляторних вимог. Це незалежний підхід до управління життєвим циклом моделей машинного навчання і є підмножиною *MLOps*.

Для успішного впровадження *MLOps* організаціям необхідно дотримуватися процесу, який включає навчання моделей, їх пакування, валідацію, розгортання та моніторинг. Цей процес гарантує, що моделі ефективно навчаються, пакуються для розгортання, перевіряються на точність і продуктивність, розгортаються у виробничих середовищах і постійно відстежуються на наявність будь-яких проблем або відхилень. Дотримуючись цього процесу, організації можуть гарантувати, що їхні моделі машинного навчання є надійними, масштабованими та надійними.

Можна розбити процес *MLOps* на декілька основних етапів [16]:

- Розробка та експериментування ШІ-моделі;
- Створення конвеєру постійної інтеграції (*CI pipeline*);
- Створення конвеєру постійної доставки (*CD pipeline*);
- Автоматичний запуск, створення тригерів для запуску;

- Моделювання *CD*;
- Моніторинг.

Стосовно циклу розробки ШІ-моделі, можна виділити наступні етапи [23]:

- Збір даних: він зосереджений на отриманні зразків даних шляхом спостереження та вимірювання реальної системи, процесу чи явища, для яких потрібно побудувати модель *ML*.
- Аналіз даних: на цьому етапі формується схема даних і визначається необхідність підготовки та попередньої обробки даних.
- Попередня обробка даних: цей крок передбачає очищення даних, створення узгоджених даних для навчання та перевірки, а також інженерію фіч (*feature engineering*), які допоможуть в навчанні. Крім того, у випадку навчання під наглядом дані розмічені.
- Тренування моделі: передбачає вибір моделі залежно від типу проблеми (наприклад, класифікація чи регресія). Модель навчається з використанням навчальних даних, поки похибка не буде мінімізована шляхом надання іншого значення параметрів і гіперпараметрів. (Гіперпараметр регулює важливі властивості моделі *ML*, включаючи оверфітінг, андерфітінг і складність моделі).
- Перевірка моделі: продуктивність моделі оцінюється за перевірочним набором даних, щоб переконатися, що навчена модель добре працює на невидимих даних (перевіряється здатність до узагальнення).
- Розгортання моделі: підтверджена модель розгортається у виробництві та контролюється її продуктивність.

ШІ-платформа, як і будь-яка програмна система, потребує розробки окремої моделі *ML*, яка є по суті експериментальною. Типовий робочий процес розробки моделі машинного навчання починається зі збору додаткової інформації про дані та наявну задачу. Дані потрібно проаналізувати (*Exploratory Data Analysis (EDA)*), очистити та попередньо обробити (інженерія та вибір фіч), щоб основні

характеристики були витягнуті з необроблених даних [13]. Після цього відбувається формування набору даних для навчання, тестування та перевірки.

Навчання, валідація та тестування алгоритму машинного навчання є процесом, пов'язаним із складностями. Ітераційний процес включає підгонку гіперпараметрів алгоритму *ML* до навчального набору даних під час використання валідаційного набору, для перевірки продуктивності даних, які модель не бачила раніше, на кожній ітерації. Тут важливо підкреслити, що те що дані модель не бачила, не усуває похибок пов'язаних з новизною даних, для цього окремо модель тестується в пост-продакшені. Нарешті, тестовий набір даних використовується для оцінки кінцевої, неупередженої продуктивності моделі [33]. Набір даних перевірки є унікальним щодо тестового набору даних, оскільки він не залишається прихованим від підготовки моделі, однак натомість він використовується для надання адекватної продуктивності здатності останньої налаштованої моделі.

З самого початку ці три набори даних розділені, і їх не можна змішувати в будь-який момент часу. Обробка даних – невід'ємний етап у підготовці до навчання нейронної мережі, який визначає ефективність та точність моделі [24]. Цей процес включає в себе ряд кроків, спрямованих на оптимізацію та підготовку вхідних даних для навчання моделі. Оскільки специфічні для проекту дані можуть бути обмежені доменно-залежними вхідними даними, цілісністю атрибутів, валідністю історичних часових шкал або залежними від стану переходами [25], збереження загальної якості даних є непростим завданням. Крім того, життєвий цикл наборів даних пов'язаний з різними параметрами якості, як описано в [26]. Проблеми з якістю слід виявляти якомога раніше, щоб ізолювати та адаптувати несправні процеси, які впливають на подальше використання. Перевірка потенційних даних, також входить до сфери відповідальності експертних моделей або експертної оцінки дослідників даних. Залежно від структури організації існують додаткові суб'єкти та ролі для операцій з даними, наприклад, *Data Stewardship* [28], який відповідає за нагляд за даними на різних етапах проекту. З

фокусом на управлінні якістю даних (*DQM – Data Quality Management*) та керуванні даними, у ранніх підходах було визначено кілька пов'язаних з даними розпорядників ролей, які визначає структура управління даними в системі.

Коли набір даних готовий до навчання, наступним кроком є вибір алгоритму і, нарешті, навчання. Цей крок є ітераційним, коли кілька алгоритмів намагаються отримати навчену модель. Для кожного алгоритму потрібно оптимізувати його гіперпараметри на навчальному наборі та перевірити модель на перевірконому наборі, що є трудомістким завданням і потребує додаткових зусиль [4]. Щоб отримати найкращу модель, потрібні кілька ітерацій, і відстеження всіх ітерацій стає важким процесом (часто це робиться вручну в таблицях *Excel*). Для регенерації найкращих результатів необхідно використовувати точну конфігурацію: гіперпараметри, архітектуру моделі, набір даних тощо [33]. Далі цю навчену модель необхідно перевірити із використанням конкретних попередньо визначених критеріїв у тестовому наборі, і якщо продуктивність прийнятна, модель готова до розгортання. Після того, як модель буде готова, вона передається групі з експлуатації, яка керує процесом розгортання та моніторингу щоб можна було зробити висновки щодо моделі.

Визначення принципів підвищення можливості багаторазового використання наборів даних описує характеристики систем, які зосереджуються на цінних результатах досліджень [29]. Ця норма визначення принципів підвищення можливості багаторазового використання наборів даних є критичною для забезпечення ефективності та узгодженості у роботі з даними. Існуюча різноманітність інструментів у сфері обробки даних і навчання моделей може викликати непорозуміння, помилки та невідповідності між етапами обробки даних та навчання моделей [29]. Створення та фокусування на конкретних *KPI* дає змогу підвищити якість значно підвищити якість моделі, але для цього потрібно передбачити постійний процес оновлення та перевірки даних.

1.4. Узагальнення проблематики розгортання компонент платформ вбудованих систем

На сьогоднішній день, актуальною задачею є розгортання компонент платформ вбудованих платформ, та окремо виділених компонент, ріст числа яких залишається експоненційним і охоплюючим велику кількість сфер життя. Сучасний індустрійний уклад розвивається навколо ШІ завдяки тому що нейронні мережі та інші моделі штучного інтелекту довели свою ефективність в багатьох задачах. Завдяки можливостям нейронних мереж, багато задач, які раніше виконувались людьми, зараз можна або автоматизувати, або значно поліпшити результати роботи, прискоривши та піднявши точність її виконання.

Що не є достатньо розвиненим на сьогоднішній день – це задача автоматизації та підтримки ШІ-платформ. Питання застосування практик *DevOps* набуло особливої уваги в останні роки, особливо у сфері штучного інтелекту, де ця необхідність зростає з кількістю нових проєктів. Принципи *DevOps* підтвердили на практиці і в низці наукових статтях, що вони збільшують ефективність роботи системи за рахунок автоматизації процесів інтеграції системи та постійної її підтримки.

Практики *DevOps* необхідно суттєво модифікувати, перед тим як застосовувати у ШІ-платформах, оскільки специфіка використання штучного інтелекту замість алгоритму заданого кодом потребує постійного моніторингу не лише комп'ютерної системи, на якій ця модель застосовується, але й є необхідність в постійному моніторингу реакції моделі на нові дані, які не проходили через модель раніше. Тому, з'явилося поняття *MLOps*, яке засноване на *DevOps*, але враховує етапи збору та обробки даних для моделі, а також враховує моніторинг якості моделі та повторне тренування моделі.

Топологічна організація в ШІ-платформах із запровадженням *MLOps* відіграє ключову роль у забезпеченні ефективного управління життєвим циклом моделей машинного навчання. Топологія в даному контексті визначає спосіб і структуру

зв'язків між різними етапами та елементами системи, такими як розробка, навчання, тестування та впровадження моделей. У зв'язку з високою складністю і обчислювальною інтенсивністю завдань машинного навчання, оптимізована топологічна організація визначає структуру розподілених обчислювальних ресурсів, оптимізованих для обробки великого обсягу даних.

Ієрархічні топологічні організації, як товсті дерева та гібридні топологічні організації на основі дерев, зможуть значно поліпшити масштабуванню ІІІ-платформ та такі топології забезпечують ефективне використання обчислювальних ресурсів під час різних етапів роботи з моделями машинного навчання. Особливо важливим є необхідність в такій структурі з умовою, що різні елементи систем, на різних рівнях топологічної організації, мають різні обчислювальні можливості та займаються різними задачами в рамках *MLOps*.

Використання *Edge*-архітектури спрямоване на оптимізацію обробки даних та використання моделей ІІІ ближче до кінцевого користувача. *Edge* в контексті *MLOps* розглядається як розширення обчислювального середовища, де відбувається збір та обробка даних безпосередньо на пристрої, що має безпосереднє значення в процесах навчання моделі та постійної підтримки системи. Існує низка статей, які розглядають різні аспекти використання *Edge* архітектури в рамках *MLOps*, і як видно з огляду літератури, це питання є актуальним і є перспективним використання *Edge* пристроїв у ІІІ-платформу.

В розглянутих статтях було показано, що підхід розробки та інтеграції *Edge*-кластерів до системи *MLOps* є одночасно багатообіцяючим та потребує додаткових досліджень. В літературі сказано про шляхи оптимізації взаємодії із *Edge*-кластерами, проте недослідженим залишається питання відмовостійкості системи та можливість використання задач ІІІ, в рамках *MLOps*, на вбудованих пристроях, або проміжних вузлах-агрегаторах. Важливим та недостатньо дослідженим аспектом є збір метрик щодо використання таких гібридних систем, та порівняння з традиційними підходами.

Висновки до розділу 1

Аналіз існуючих рішень у наукових публікаціях показав, що наявні підходи мають низку обмежень, зокрема щодо забезпечення відмовостійкості системи, управління ресурсами в гетерогенному середовищі та підвищення ефективності розгортання компонент платформи вбудованих систем. Дослідження також вказують на необхідність розробки нових методів моніторингу й збору метрик для оцінювання продуктивності таких гібридних систем у порівнянні з традиційними централізованими підходами. Недостатньо вивченими залишаються питання автоматизації процесів оновлення моделей на пристроях з обмеженими обчислювальними ресурсами, а також адаптації алгоритмів навчання до специфіки *Edge*-середовища, що потребує подальших досліджень і розробки нових методологічних підходів. З огляду на швидкий розвиток штучного інтелекту та машинного навчання, підвищення ефективності впровадження таких компонент потребує застосування новітніх технологічних підходів, серед яких особливе місце займає *MLOps*. Цей підхід забезпечує автоматизацію процесів, оптимізацію робочих потоків і гарантування безпеки даних, що є необхідною умовою для ефективної інтеграції та функціонування компонент на основі ШІ в масштабах великих і розподілених систем.

Розгортання та інтеграція компонент платформ вимагає створення масштабованих і адаптивних рішень, які здатні ефективно підтримувати різноманітні вимоги до обробки великих обсягів даних і виконання складних обчислювальних задач. Впровадження таких рішень також має на меті забезпечення безперервної роботи, швидкості адаптації до змінних умов навантаження, а також забезпечення високої надійності й безпеки при обробці конфіденційних даних. У свою чергу, вбудовані системи мають специфічні обмеження, що накладають вимоги до автоматизації не лише процесів розробки та тестування, але й до безперервного моніторингу й перенавчання моделей. Це

забезпечує можливість підтримки високої продуктивності й ефективності роботи системи в умовах реального часу. Однією з основних задач є розробка масштабованих рішень для автоматизації процесів, що дозволяють знизити вплив людського фактору та помилки, пов'язані з ручним налаштуванням і перевіркою.

На основі цього поставлені наступні задачі дослідження:

- Провести аналіз аспектів розгортання компонентів платформ вбудованих систем виходячи з існуючих технологій та процесів, з метою розробки власного методу, що дозволяє систематизувати процес побудови контейнерів та їх пересилку підвищуючи загальну ефективність платформи;

- Підвищити відмовостійкість компонентів платформи, використовуючи спеціалізовані топології на основі кодових перетворень ДеБруйна, який дає змогу побудувати додаткові маршрути для пересилки даних в платформі за рахунок дебруйнівських зв'язків;

- Розробити метод із розгортанням компонент платформи вбудованих систем для вбудованих пристроїв, на основі онтологій процесів, який передбачає інтеграцію архітектурних рішень і технологій розгортання та обробки даних;

- Вдосконалити процес розгортання компонентів платформи вбудованих систем використовуючи прунінг контейнерів та конвеєрні зборки контейнерів, з метою зменшення розміру контейнера та зменшення часу його пересилки від головного вузла до вузлів-посередників та вбудованих систем;

- Розробити інструментальний засіб, з метою реалізації запропонованого методі із застосуванням топологій на дебруйнівських зв'язках таких як *Tree-DeBruijn* та *Dragonfly-DeBruijn* для ефективного управління процесом розгортання компонентів платформи;

- Провести оцінку та аналіз отриманих результатів роботи системи, яка засновується на розробленій системі, показати її ефективність у реальних умовах, порівнюючи отримані результати з існуючими рішеннями, та рішеннями на основі існуючих підходів.

РОЗДІЛ 2

МЕТОД РОЗГОРТАННЯ КОМПОНЕНІВ ПЛАТФОРМ ВБУДОВАНИХ СИСТЕМ ІЗ ЗАСТОСУВАННЯМ ТЕХНОЛОГІЇ MLOPS

2.1. Розробка архітектури платформи вбудованих систем для розпізнавання образів

На сьогоднішній день, актуальною задачею є створення систем ШІ, ріст якого залишається експоненційним і охоплюючим велику кількість сфер життя. Сучасний індустрійний уклад розвивається навколо ШІ завдяки тому що нейронні мережі та інші моделі штучного інтелекту довели свою ефективність в багатьох задачах [57]. Часто питання архітектури системи, на якій виконується така задача не розглядається, а тому є сенс, з наукової та інженерної точки зору, дослідити засоби та прийоми при розробці архітектури системи.

Під платформою на основі вбудованих систем будемо розглядати поєднання апаратних та програмних компонентів, що дозволяють забезпечити ефективну інтеграцію різних функцій та сервісів у межах однієї системи. Вбудовані системи часто характеризуються компактними розмірами, а також здатністю до адаптації під специфічні вимоги користувача або середовища експлуатації. У контексті таких платформ, ключову роль відіграє оптимізація програмного забезпечення для забезпечення стабільної роботи навіть за обмежених ресурсів. В рамках даного дослідження, програмне забезпечення використовується для моделей ШІ.

Задача полягає у розробці системи, здатної розпізнавати образи з фотознімків на основі даних з камери, яка емулюється використанням зображень з датасету *ANIMAL10N*, та обробки візуальної інформації для автоматичного виявлення та класифікації об'єктів у реальному часі. Для виконання даної задачі запропоновано використання платформ вбудованих систем. Використання датасету дозволяє створювати моделі, здатні розпізнавати різні види тварин у варіативних умовах знімання, що є важливим для адаптації алгоритмів до різних

сценаріїв відеоспостереження. При цьому, незважаючи на те, що дані з *ANIMALION* не є справжніми відео, даний підхід дозволяє моделювати та тренувати системи для подальшого застосування у відеоаналізі. Основною метою є підвищення точності та швидкості обробки зображень для ефективної інтеграції у реальні системи відеоспостереження, що включають в себе аналіз поведінки, виявлення аномалій та забезпечення безпеки у різних умовах.

Одним із ключових аспектів архітектури системи для ІІІ задач є масштабованість. Моделі ІІІ часто вимагають значних обчислювальних потужностей і ємності для зберігання даних. Тому розробка архітектури повинна передбачувати масштабованість системи, таким чином щоб існували можливості обробки зростаючих обсягів даних і обчислювальними вимогами в міру того, як ІІІ платформа буде ускладнюватись або завдання змінюватись, наприклад обробка зображень та обробка відео [58]. Крім того, враховуючи чутливий характер даних, що використовуються в ІІІ, необхідно впровадити надійні заходи безпеки для захисту від несанкціонованого доступу та витоку даних. Архітектура системи має також передбачувати необхідність в оркестровці вбудованих пристроїв, які, за дизайном, мають нижчу обчислювальну продуктивність.

При створенні високопродуктивної обчислювальної системи, якою є ІІІ-платформа, важливою складовою роботи є розробка топологічної організації. Цей крок дозволяє вчасно передбачити характеристики системи, не виконуючи моделювання системи з використанням реальних ресурсів, що є проблематичним з підвищенням ранку масштабування, і як результат дозволяє правильно підібрати обладнання та розробити алгоритми маршрутизації. Як відомо, більшість кластерів використовує центральні процесори разом з декількома графічними прискорювачами для вирішення обчислювальних задач. Зазвичай, значна частина ліній *PCI-Express* вже використовується, і тому важливо передчасно кількість зв'язків для кожного вузла, що буде впливати на чергу

повідомлень, яка буде утворюватися. Окремо слід виділити можливості *Edge* пристроїв до під'єднання до мережі, більшість таких пристроїв має, щонайменше, *Gigabit Ethernet*, і враховуючи ріст можливостей *Wi-Fi 6E*, можна вважати, що такі пристрої здатні підтримувати з'єднань з декількома пристроями в мережі [59].

Окрім цього, застосовуються різні методи синтезу нових топологій, в тому числі за допомогою двійкових перетворень, наприклад дебруйнівський ланцюг, та недвійкових кодових перетворень, з використанням таблиці, наприклад латинський квадрат, з використанням систем рівнянь. Запропонований метод буде використовувати метод об'єднання топологій двійкового дерева та дебруйнівських ланцюгів [60].

2.1.1. Особливості впровадження гібридної Edge-Cloud архітектури в платформу вбудованих систем

Метою розробки гібридної архітектури на основі *Edge* та *Cloud* є використання та подальше застосування сильних сторін обох архітектур.

Першим кроком у створенні архітектури *Edge-Cloud* є визначення конкретних завдань, які необхідно виконувати на периферії, і тих, які краще підходять для хмари. В рамках платформи вбудованих систем та архітектурного рішення з використанням дерева з дебруйнівськими зв'язками, можна застосувати ієрархічну природу такої топології для пристроїв на *Edge* та *Cloud* [61]. *Edge* пристрої можуть знаходитись на нижчих ярусах, коли *Cloud* знаходитиметься на корні дерева. З точки зору того, що це платформа, необхідна модель, яка буде впроваджена. Для цього необхідно проаналізувати вимоги задачі для моделі, враховуючи такі фактори, як затримка, пропускну здатність і обсяг даних. Наприклад, завдання, які вимагають обробки в режимі реального часу або низької затримки, такі як виявлення об'єктів в автономних транспортних засобах, краще підходять для периферійних обчислень. В рамках даних факторів

та їх похідних, необхідно розглядати різні метрики для постійної підтримки працездатності системи.

Створення ефективної ІІ-платформи передбачає створення *CI/CD* конвеєра. Від ефективності та відмовостійкості конвеєра залежить здатність системи швидко реагувати на зміни. Ці зміни можуть бути як і зовнішні та і внутрішні, наприклад, вихід зі строю одного пристрою може нанести суттєвий удар по працездатності системи. Враховуючи, що питання відмовостійкості вирішується за допомогою використання спеціальної топологічної організації, залишається питання впровадження змін до системи для відновлення працездатності. Також, можлива ситуація, коли точність моделі падає за рахунок використання даних, раніше не навчених моделлю. Крім того, необхідно мати механізми синхронізації цих даних та логування, щоб гарантувати, що оновлення, зроблені або отримані на периферії або в хмарі, були відкриті для використання в рамках системи.

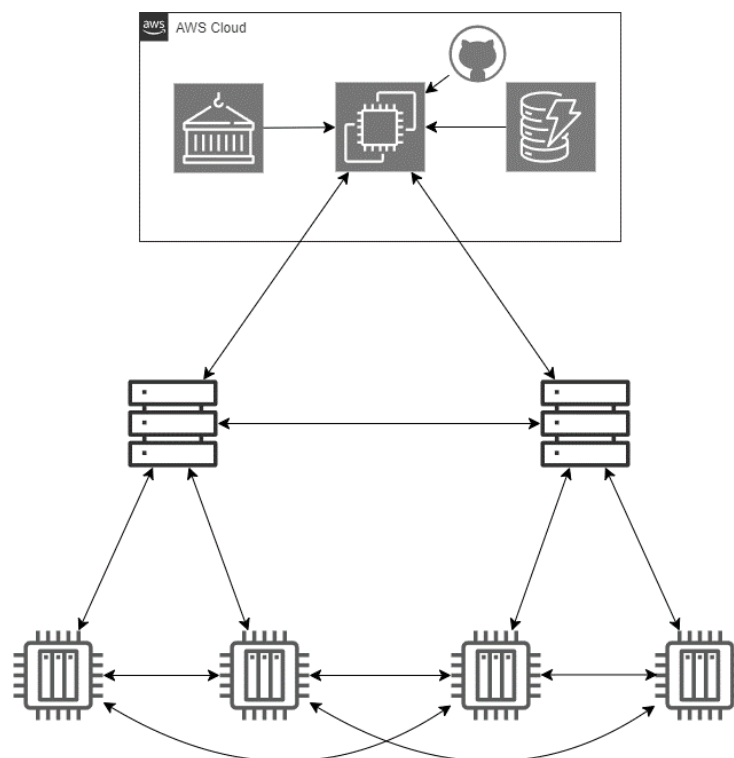


Рис. 2.1. Макет платформи вбудованих систем

На рисунку 2.1 наведений макет запропонованої архітектури системи. Можна побачити, що система гетерогенна та здатна до вертикального

масштабування на будь-якому з елементів. На корні дерева використовується наданий *Cloud* ресурси від *AWS*, в даному випадку, віртуальна машина, нереляційна база даних та місце зберігання контейнерів. Для того щоб підвищити ефективність виконання завдань на *Cloud*, можна підключати додаткові сервіси, або замовляти більш потужні віртуальні машини [6].

Враховуючи те, що *Edge* пристрої значно менш надійні ніж, наприклад, серверне обладнання, необхідно враховувати сценарій коли виходить з ладу один з пристроїв. В разі коли таке трапляється, завдання для цього пристрою автоматично переходять до під'єданого до нього пристрою. За допомогою топологічної організації, яка розрахована на відмовостійкість, задачі переходять до вузла під'єданого до несправного, після чого відправляється запит на корінь дерева, тобто *Cloud*. Коли під'єднана до мережі заміна пристрою, який відмовив, проходить робота конвеєру і новий пристрій отримує задачі минулого, в вигляді артефакту.

Важливо, щоби архітектура системи мала можливість відповідати на виклики поставлені під час виконання програмного забезпечення, в даному випадку, III моделі. Одним з найчастіших викликів є сценарій коли падає точність моделі під час її застосування на даних, які не були застосовані під час навчання. Враховуючи, що завжди існує доля проникнення даних з набору для тестування під час навчання, потрібно завчасно враховувати, що використання моделі поза її командою розробників може призвести до значного зменшення характеристик як точність. Для вирішення даної проблеми пропонується використовувати логування та прийом постійного моніторингу. Основна ідея полягає в тому, що нові дані будуть відправлятися на один вузли вищого рангу, або на корінь, і там буде прийматись рішення щодо донавчання моделі та розсилки задач на мережу.

2.2. Особливості розгортання компонент платформи вбудованих систем на основі контейнеризації

Розгортання компонент платформ вбудованих систем, як і інших систем, є складним завданням. Для його успішного виконання застосовується низка процедур, які в сукупності складають конвеєр. Весь конвеєр часто будується послідовно за допомогою інструментів, які важко інтегруються. Метою *MLOps* є автоматизація конвеєра *ML*. Його можна розглядати як перетин між машинним навчанням і практиками *DevOps*. По суті, це сукупність методів автоматизації, керування та прискорення тривалої експлуатації ШІ моделі виконане через тестування, контейнеризація та розгортання шляхом інтеграції практик *DevOps* у ШІ. Розглянуті методи не включатимуть постійну підтримку саме моделі ШІ, оскільки цей процес виконується в рамках дослідження саме структури та параметрів нейронної мережі, або будь-якого іншого методу машинного навчання.

В багатьох інженерних та наукових дослідженнях порівнювалась продуктивність віртуальних машин і контейнерів. Час виконання, використання процесору, пам'яті та енергоспоживання контейнерів кращі, ніж у віртуальних машин. Однак безпека та універсальність віртуальних машин кращі, ніж у контейнерів, завдяки додатковому шару ізоляції. Наприклад, віртуальна машина може запускати *Linux* і *FreeBSD* одночасно, але контейнер може виконувати лише один із них, оскільки контейнери можуть запускати лише різні ОС в одному ядрі (наприклад, *Debian*, *Red Hat*, *Ubuntu*) [62]. У таблиці 2.1. порівнюються контейнери та віртуальні машини. Обидві технології мають переваги та недоліки та підходять для різних застосувань.

Порівняння контейнерізованого ІІІ-застосунку та віртуалізованого

Характеристика	Контейнер	Віртуальна машина
Операційна система	Спільна з хост ОС	Незалежна від ОС
Архітектура CPU	Така сама що і у хост	Можлива емуляція
Портативність	Висока	Низька, але не залежить від апаратного забезпечення
Безпека	Всі процеси виконуються на одному ядрі	Емульована ОС працює окремо; високий рівень ізоляції від хосту
Доступ до апаратного забезпечення	Доступ через драйвери хосту	Необхідна ВМ і окремі драйвери; в деяких випадках, апаратне забезпечення має бути ізольоване від хосту
Оверхед оперативної пам'яті	Низький	Високий
Обмін файлами	Контейнери можуть обмінюватись файлами	Кожна ВМ має свою файлову систему, необхідно додаткове налаштування

MLOps має на меті об'єднати в один конвеєр та автоматизувати розробку і розгортання систем машинного навчання. Вона передбачає інтеграцію розробки машинного навчання з операціями для оптимізації робочих процесів і забезпечення ефективного та результативного розгортання моделей штучного інтелекту. У цьому контексті контейнери відіграють важливу роль в *MLOps*, долаючи розрив між різними середовищами розробки та виробничими реаліями.

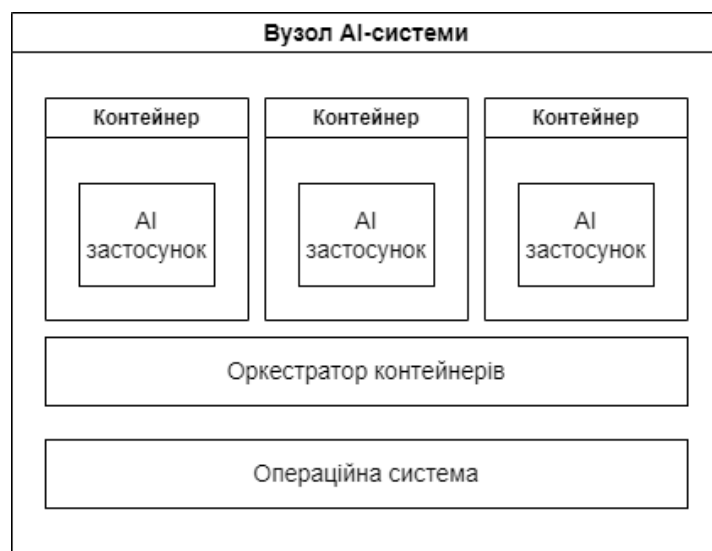


Рис. 2.2. Схема вузла з контейнерами

Контейнери – це модульні одиниці, які інкапсулюють середовище виконання програми. Вони забезпечують послідовне та ізольоване середовище для запуску моделей ШІ, гарантуючи стабільну продуктивність, відтворюваність і масштабованість. Пакуючи всі залежності та конфігурації, необхідні для запуску моделі ШІ, в контейнер, розробники можуть забезпечити безперебійну роботу моделі в різних середовищах - від розробки до виробництва. На рисунку 2.5. показано схематичне зображення вузла платформи вбудованих систем, де можна побачити, що контейнери виконують код незалежно один від іншого, маючи доступ до клієнтської ОС. Це може бути використано для того, щоб в подальшому вертикально масштабувати систему, додавши новий функціонал на вузли.

Однією з головних проблем при розгортанні моделей машинного навчання є управління залежностями. Моделі *ML* часто вимагають певних версій бібліотек і фреймворків, які є специфічними, тобто версія яких є встановленою, і може бути складно керувати в різних середовищах великою кількістю залежностей. Контейнеризація вирішує цю проблему, інкапсулюючи не тільки додаток, але і всі його залежності в єдиний пакет, відомий як контейнер. Це дозволяє легше керувати залежностями і гарантує, що модель може бути розгорнута без проблем із сумісністю [63].

Масштабованість – ще один важливий аспект розгортання ШІ. Контейнери забезпечують масштабоване рішення, дозволяючи розробникам легко реплікувати і розподіляти контейнери між декількома серверами або кластерами. Завдяки тому, що кожен контейнер фактично є копією, не тільки програмного коду, або програмного продукту, а, фактично, є копією середовища, програму можна дуже швидко масштабувати. Розгортання моделей ШІ в більших масштабах потребує гарантування, що інфраструктура зможе ефективно справлятися зі зростаючими робочими навантаженнями.

Контроль версій – ще одна задача, яка має значне значення при розробці машинного навчання, оскільки моделі часто піддаються ітеративним

поліпшенням і оновленням. Використовуючи контейнери, розробники можуть легко керувати різними версіями моделі, створюючи окремі контейнери для кожної версії. Це гарантує доступ до попередніх версій за потреби та забезпечує чітку історію змін моделі. На відміну від системи контролю версій як *Git*, контейнер зберігає не лише код, а і відтворює середовище виконання [64]. Контроль версії контейнеру здійснюється як відображення:

$$v : C \rightarrow V, \quad (2.1)$$

де C — множина контейнерів, P — множина платформ, а V — множина версій. Основна ідея контролю версії полягає в збереженні різних станів середовища в контейнері, тому передбачається використання контейнерів старої (стабільної) версії.

$$P = \{C_1, C_2, \dots, C_n\}, \quad (2.2)$$

де P — множина контейнерів (або пакетів), $C_1, C_2, \dots, C_n$ — окремі компоненти, n — кількість контейнерів у множині P , $n \in \mathbb{N}$. Портативність також є значущою перевагою контейнеризації при розгортанні ШІ. Контейнери можна легко переміщати між різними середовищами, наприклад, з локального виробничого серверу в *Cloud*. Це дозволяє безперешкодно розгортати і тестувати на різних платформах, знижуючи ризик виникнення проблем із сумісністю. Виходячи з характеристик за якими даний метод може бути досліджений, є сенс виконувати заміри часу, використання пам'яті та логування під час розгортання сервісів на вузлах системи.

2.2.1. Застосування планувальника задач для балансування навантаження контейнерів платформи

У сучасному розвитку контейнеризація стала основною технологією створення хмарних програм. Замість великих монолітних додатків розробники тепер можуть використовувати для створення додатків окремі слабозв'язані компоненти (відомі як мікросервіси). Хоча контейнери, як правило, менші,

ефективніші та забезпечують більшу мобільність, вони мають застереження [65]. Чим більше контейнерів використовується в системі, тим важче працювати з ними та керувати ними — одна програма може містити сотні чи навіть тисячі окремих контейнерів, які повинні працювати разом, щоб забезпечити виконання функцій програми.

Оскільки кількість контейнерних додатків продовжує зростати, керування ними в масштабі майже неможливо без використання автоматизації. Тут на допомогу приходить оркестровка контейнерів, яка виконує критичні завдання з управління життєвим циклом за частку часу. Оркестрування — це процес управління та координації декількох контейнерів, що працюють у розподіленій системі. Наприклад, є 50 контейнерів, які потрібно оновити. Оновлення вручну можливе, але скільки часу та зусиль знадобиться команді розробників, щоб виконати роботу. За допомогою оркестровки контейнера можна написати файл конфігурації, а інструмент оркестровки контейнера зробить усе за вас. Декларативний підхід може спростити численні повторювані та передбачувані завдання, необхідні для безперебійної роботи контейнерів, такі як розподіл ресурсів, керування репліками та налаштування мережі.

У контексті ІІІ оркестрування включає такі завдання, як масштабування робочих навантажень ІІІ, забезпечення високої доступності та управління розподілом ресурсів [66]. Зі зростанням складності та масштабу систем ІІІ ручне керування контейнерами стає неефективним і схильним до помилок. Існує кілька платформ оркестрування, які спеціалізуються на управлінні контейнерами систем ІІІ. Ці платформи надають такі функції, як автоматичне масштабування, балансування навантаження та моніторинг стану. Вони також інтегруються з популярними технологіями контейнеризації, такими як *Docker Swarm* і *Kubernetes*, що полегшує управління робочими навантаженнями ІІІ в контейнерному середовищі.

Однією з ключових переваг оркестрування контейнерів для систем ІІІ є підвищення ефективності управління та створення середовищ [66]. Автоматизувавши такі завдання, як масштабування та розподіл ресурсів, організації можуть оптимізувати використання своєї інфраструктури та зменшити витрати. Крім того, платформи оркестрування надають можливості централізованого моніторингу та ведення журналів, що полегшує усунення несправностей і забезпечує надійність систем штучного інтелекту.

Ще однією перевагою оркестрування контейнерів для систем ІІІ є підвищена гнучкість [66]. Завдяки контейнеризації організації можуть легко розгортати й оновлювати моделі ІІІ, не порушуючи роботу наявних сервісів. Це дозволяє їм швидко ітерації і реагувати на мінливі потреби. Крім того, контейнеризація дозволяє організаціям запускати робочі навантаження ІІІ на будь-якій інфраструктурі, як локальній, так і хмарній.

За умов високого навантаження платформи *HTTP*-запитами, створюється черга з метою кешування запитів та обслуговування їх щонайшвидше. При інтеграції з моделлю штучного інтелекту, важливо враховувати, що обробка запитів може бути ресурсоємною, особливо в ситуаціях, коли модель вимагає значних обчислювальних ресурсів. У таких випадках може виникати затримка в обробці запитів, що потребує реалізації механізмів управління потоками. Коли клієнт надсилає запит, платформа ініціює обробку цього запиту, створюючи новий контекст для запиту, що дозволяє доступ до об'єктів, специфічних для даного запиту, таких як дані форми, параметри та заголовки. При реалізації черги в платформі важливо забезпечити, щоб запити були оброблені в порядку їх надходження, оскільки це важливо для прикладних застосувань, де запити потребують строгого дотримання порядку виконання. Найпростішим підходом є черга *FIFO*, першим прийшов – першим обслугований. За допомогою використання черг *FIFO* покращене управління навантаженням, знижуючи ризик перевантаження платформи чи її окремо взятій компоненти під час пікових

навантажень запитами. Оскільки запити обробляються по одному, система має змогу контролювати ресурси, запобігаючи виникненню ситуацій, коли обробка одного запиту блокує інші, або коли відповідь на один запит використовується для іншого запиту.

2.2.2. Опис моделі глибокої нейронної мережі для платформи вбудованих систем для розпізнавання образів

Навчання моделі є досить трудомістким та багатокроковим етапом. Конвеєр *ML*, який будує та розгортає моделі, зазвичай пропонується хмарними постачальниками як основа для хмарних *MLOps*. Така послуга надає обчислювальні ресурси та зберігання даних на вимогу, щоб дозволити навчання моделей машинного навчання. Як правило, ця служба також постачається з репозиторієм моделі *ML* і контейнерним сховищем, щоб забезпечити швидше розгортання [13]. Ми використовуємо конвеєр *ML* для навчання та перенавчання моделей *ML*. Щоб навчити та перевірити конвеєр *ML* для певного набору даних, ми розглянемо ці п'ять кроків. Увесь етап розробки моделі включає оцінку того, наскільки якісну модель можна побудувати, пошук найкращих гіперпараметрів, налаштування компромісу між андерфітінгом і оверфітінгом, а також пошук балансу між вдосконаленням моделі та витратами на обчислення [30]. Цей крок також стосується експериментування, яке відбувається протягом усього процесу розробки моделі: кожне важливе рішення приходить принаймні з деяким експериментом. Коли моделі створені, їх потрібно перевірити, щоб переконатися, що вони працюють належним чином.

В рамках дослідження *MLOps* важливим аспектом є вибір та впровадження моделі нейронної мережі, і, передусім, їх розмір та характеристики мають велике значення. Використання малих моделей, таких як *MobileNet* має низку переваг перед іншими мережами, що робить їх особливо привабливими для експериментів у цій галузі. По-перше, такі моделі забезпечують швидший цикл розробки та тестування, що дозволяє оперативно оцінювати різні аспекти автоматизації,

оптимізації та управління процесами машинного навчання. По-друге, малі моделі є більш зрозумілими та прозорими у порівнянні з їхніми великими аналогами, що, в свою чергу, спрощує процес інтерпретації результатів і дозволяє легше виявляти проблеми, пов'язані із впровадження досліджуваного методу.

Впровадження моделей штучного інтелекту, зокрема згорткових нейронних мереж (*CNN*), в платформи вбудованих систем є складним процесом, що вимагає врахування специфічних вимог до обчислювальних потужностей, енергоспоживання та апаратних ресурсів. Згорткові нейронні мережі, які відзначаються високою ефективністю у задачах обробки зображень, звуку та інших даних, широко використовуються у вбудованих системах, зокрема в системах відеоспостереження, розпізнавання об'єктів, автономних транспортних засобах та інтелектуальних пристроях. Проте для успішного впровадження таких моделей необхідно адаптувати їх до обмежених можливостей вбудованих платформ, шляхом, наприклад прунінгу. Вбудовані системи часто мають обмежену потужність процесорів, пам'ять та пропускну здатність, що вимагає оптимізації нейронних мереж для мінімізації обсягу обчислень і пам'яті. Враховуючи це, необхідно застосовувати саме ті моделі мереж, які є адаптованими до умов вбудованих систем.

Важливим є питання інтеграції програмного забезпечення з апаратним забезпеченням в умовах вбудованих систем. Багато платформ, що використовуються у вбудованих системах, мають обмеження щодо підтримки сучасних бібліотек машинного навчання, через специфіку архітектури таких систем, що може ускладнити розробку і розгортання моделей *CNN*. Для цього розробники часто використовують оптимізовані бібліотеки, такі як *TensorFlow Lite* або *OpenVINO*, що забезпечують крос-платформенну сумісність та можливість розгортання моделей на різних апаратних пристроях з обмеженими ресурсами. Важливим є не тільки питання застосування безпосередньо бібліотек машинного навчання, але й бібліотек для взаємодії з іншими компонентами

платформи, що в свою чергу створює питання ефективності впровадження бібліотек в програмну частину компоненти платформи.

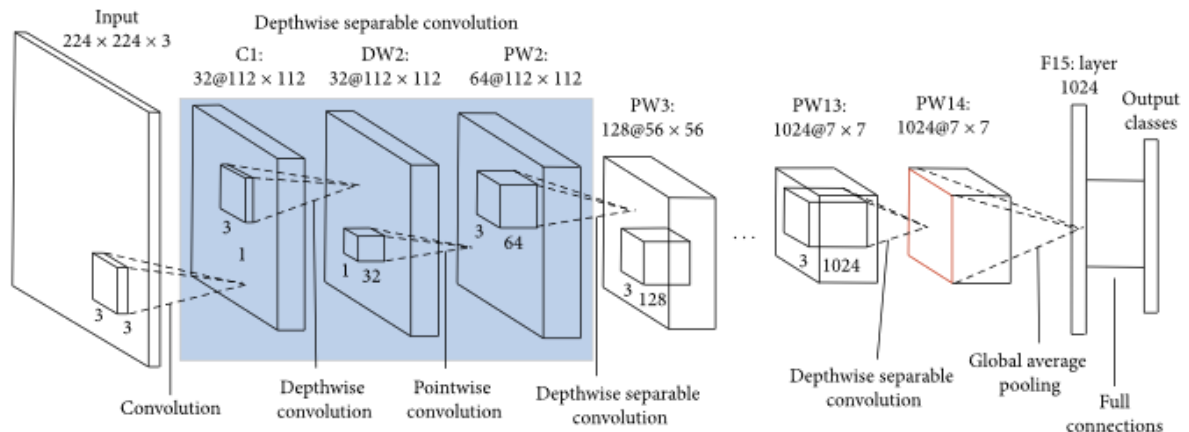


Рис. 2.3. Схематичне зображення *MobileNet* [67]

MobileNet, зображений на рисунку 2.3., побудований на шарах згортки, які можна розділити по глибині. Кожен шар згортки, який можна розділити по глибині, складається з згортки по глибині (*depthwise convolution*) та покрокової згортки (*pointwise convolution*). Враховуючи згортки по глибині та по точках як окремі шари, *MobileNet* має 28 шарів. Стандартний *MobileNet* має 4,2 мільйона параметрів, які можна ще більше зменшити шляхом відповідного налаштування гіперпараметра множника ширини [67].

Таблиця 2.2.

Порівняльна таблиця моделей штучних нейронних мереж

Характеристика	<i>MobileNet</i>	<i>ResNet</i>	<i>InceptionNet</i>
Тип моделі	Легка модель для мобільних пристроїв	Глибока мережа з залишковими зв'язками	Гібридна архітектура з використанням різних фільтрів
Кількість параметрів	~4.2 мільйони (для <i>MobileNetV1</i>)	11 мільйонів (для <i>ResNet-50</i>)	5 мільйонів (для <i>InceptionV3</i>)
Глибина мережі	28 шарів (для <i>MobileNetV1</i>)	50+ шарів (для <i>ResNet-50</i>)	48 шарів (для <i>InceptionV3</i>)
Використання операцій	<i>Depthwise separable convolutions</i>	Залишкові зв'язки (<i>residual connections</i>)	Різні фільтри (1x1, 3x3, 5x5, <i>max pooling</i>)

Продовження таблиці 2.2.

Характеристика	<i>MobileNet</i>	<i>ResNet</i>	<i>InceptionNet</i>
Час навчання	Швидке навчання завдяки оптимізації	Тривалий час навчання через глибину	Тривалий час навчання через складність моделі
Швидкість (Inference Time)	~5.5 мс (для <i>MobileNetV1</i> на мобільному пристрої)	~100 мс (для <i>ResNet-50</i> на сучасному GPU)	~70 мс (для <i>InceptionV3</i> на сучасному GPU)
Точність (Top-1 Accuracy)	~70% (для <i>MobileNetV1</i> на ImageNet)	~76% (для <i>ResNet-50</i> на ImageNet)	~77% (для <i>InceptionV3</i> на ImageNet)
Точність (Top-5 Accuracy)	~89% (для <i>MobileNetV1</i> на ImageNet)	~93% (для <i>ResNet-50</i> на ImageNet)	~93% (для <i>InceptionV3</i> на ImageNet)

В порівнянні з іншими моделями, *MobileNet* має найменшу кількість параметрів і вимагає найменше обчислювальних ресурсів, але зазвичай має меншу точність, особливо на складних наборах даних, тому використання *MLOps* є вигідним способом використання цієї моделі з подальшим її донавчанням на нових даних. *GoogleNet* має глибоку структуру з блоками *Inception*, що дозволяє зменшити кількість параметрів, при цьому забезпечуючи високу точність. *VGG16* зазвичай має велику кількість параметрів, але завдяки своїй простоті може бути легше налаштована та адаптована для конкретних завдань [68].

Останні два кроки це пакування та реєстрація моделі. Пакування моделі: після тестування навченої моделі модель серіалізується та упаковується в контейнер для стандартизованого розгортання. Важливо оцінити модель і порівняти її продуктивність з тим, що існувало раніше; цього можна досягти за допомогою метрик, але універсальних показників не існує [30]. Реєстрація моделі: модель-кандидат, яка була серіалізована, реєструється та зберігається в реєстрі моделей, звідки вона готова для швидкого розгортання на периферійному

пристрої [31]. Ці кроки дозволяють, в подальшому, швидко розгорнути сервіс з використанням раніше навченої моделі.

2.3. Застосування CI/CD конвеєру для розгортання платформи вбудованих систем

Процес *DevOps* пов'язаний як з фазою розгортання моделі, так і з її випуском. Після випуску моделі вона стає програмним компонентом, який процес *DevOps* може далі включити як програми або служби. В рамках даної роботи, виходячи з минулих статей на тематику, [31] ми визначаємо закінчення *MLO* в точці випуску моделі. Таким чином, коли модель передається в репозиторій програми машинного навчання, ми визначаємо точки, в яких *MLOps* і *DevOps* зливаються. Створення конвеєра *CI/CD* для ШІ-платформ має вирішальне значення для забезпечення безперебійної та ефективної розробки, тестування і розгортання моделей машинного навчання. В рамках ШІ-платформи, конвеєр *CI/CD* повинен бути адаптований до унікальних вимог операцій машинного навчання, архітектури кінцевої системи. *MLOps* - це практика застосування принципів і практик *DevOps* до проєктів машинного навчання. У контексті побудови системи штучного інтелекту *MLOps* гарантує, що моделі будуть належним чином навчені, перевірені та розгорнуті у відтворюваний і масштабований спосіб.

Безперервна інтеграція (*CI*) — це практика інтеграції змін коду в існуючу базу коду, щоб будь-які конфлікти між різними змінами коду, зробленими розробниками, швидко виявлялися та вирішувалися [32]. Фаза побудови та тестування — це компоненти *DevOps*, які беруть участь у безперервній інтеграції, оскільки після кожного фіксування коду виконується автоматичне створення коду з подальшими відповідними автоматизованими тестами. Оскільки етапи планування та кодування виконуються перед етапом будівництва, їх можна вважати пов'язаними з постійною інтеграцією. *CI* готує нові зміни коду для розгортання, тому ця практика має вирішальне значення для підвищення ефективності розгортання.

Безперервна доставка (CD) передбачає фазу випуску *DevOps*, оскільки її метою є підтримка готовності програми до розгортання у робочому середовищі. Безперервне розгортання зазвичай виконується пізніше, оскільки це автоматизує розгортання коду. Якщо цей елемент недоступний, розгортання робочої версії буде виконано користувачем вручну. Безперервне розгортання не є звичайним, оскільки воно корисне лише тоді, коли велика кількість розробників працюють разом, щоб створювати численні випуски щодня [32]. Незалежно від того, чи використовується безперервне розгортання, ці елементи конвеєра слідують за безперервною інтеграцією, утворюючи більший конвеєр під назвою *CI/CD*.

Отже, виходячи з вище зазначених даних, є сенс описати метод розгортання компонент ІІІ-платформи. Загальний принцип побудови платформи лежить в *CI/CD* конвеєрі, який можна описати як напрямлений, ациклічний граф $G = (S, d_s)$, де S - етапи *CI/CD*, d_s - залежності між етапами.

1. Перший основний крок – це безперервна інтеграція. Оскільки фази планування та кодування виконуються до фази побудови, їх можна вважати пов'язаними з безперервною інтеграцією. Під час виконання фази *CI*, готуються нові зміни коду до розгортання, які мають безпосереднє значення для підвищення ефективності розгортання з точки зору якості та коректного виконання коду. Цей етап можна розбити на наступні підетапи:

2. На етапі підготовки здійснюється комплексне тестування вихідного коду машинного навчання, включаючи синтаксичний аналіз, перевірку типів даних (семантичний аналіз), оцінку метрик продуктивності моделі. Платформа перевіряє відповідність кодової бази встановленим стандартам якості та вимогам архітектурної сумісності. Фаза збірки та тестування є компонентами *DevOps*, які беруть участь у безперервній інтеграції, оскільки кожна фіксація коду супроводжується автоматизованою генерацією коду з подальшим автоматизованим тестуванням. Даний етап можна розглянути як послідовність функцій, що виконуються одна за одною:

$$\text{Syn}(C) \rightarrow T(C) \rightarrow P(M) \rightarrow Q(C) \rightarrow A(C) \rightarrow G(C) \rightarrow T_{\text{test}}(C), \quad (2.4)$$

де, процес включає в себе наступні етапи, де C – компонента платформи:

- Синтаксичний аналіз: $\text{Syn}(C)$
- Перевірка типів даних (семантичний аналіз): $T(C)$
- Оцінка метрик продуктивності: $P(M)$
- Перевірка відповідності стандартам якості: $Q(C)$
- Перевірка архітектурної сумісності: $A(C)$
- Автоматична генерація нового коду: $G(C)$
- Автоматизоване тестування: $T_{\text{test}}(C)$

3. Наступним етапом конвеєра є контейнеризація та ізоляція обчислювального середовища з використанням технологій як, наприклад, *Docker*. Даний етап забезпечує уніфіковане розгортання ІІІ-компонентів з високим рівнем масштабованості, гнучкості та відмовостійкості. Підхід на основі контейнеризації дозволяє ефективно розгортати нові ітерації моделі, швидко вносити зміни з залежностями в програмній частині та ефективно сегрегувати програмну частину, яка відповідає за ІІІ модель від модулів комунікації з іншими сервісами платформи та користувачами. Виходячи з виразу (2.4), розширюємо її з урахуванням залежностей програмних компонент і версій:

$$C = \{(C_i, (d_i, v_i)) \mid C_i \in P, d_i \subseteq D, v_i \subseteq V\}, \quad (2.5)$$

де C – множина контейнерів платформи, P – множина всіх контейнерів, D – множина всіх можливих залежностей, V — множина всіх можливих версій залежностей.

4. Третій етап – безперервна доставка. Її мета - тримати програмну частину готовою до розгортання на платформі. Безперервне розгортання виконується пізніше, оскільки воно автоматизує розгортання програмної частини платформи. Якщо модель недоступна, виробнича версія буде розгорнута користувачем вручну.

5. Наступним етапом є розгортання на вузлах систем платформи вбудованих систем, яке реалізується через стратегію інкрементальної оновлення, що передбачає поетапне розгортання оновлених контейнерів із забезпеченням неперервності операційних процесів, тобто, під час оновлення програмної частини, платформа має зберігати працездатний стан. На цьому етапі можуть використовуватися механізми паралельного розгортання та інші стратегії направлені на прискорення розгортання та міграції частини програмних компонентів для відмовостійкості системи. Після завершення першого етапу, візьмемо цей етап як функцію U , яка на вході приймає контейнер C , середовище розробки та тестування E та платформу для розгортання P , і на виході дає нову версію контейнеру C_i .

$$C_{i+1} = U(C_i, P, E), \quad (2.6)$$

де C_{i+1} нова версія контейнеру, P – платформа, E – середовище, t_0 та t_1 – це час тестування працездатності платформи. Враховуючи архітектурні особливості платформи, окремо зазначимо паралельне виконання розгортання контейнеру на різних вузлах.

$$D(C_{parallel}, P, E) = \{U(C_i, P, E) \mid C_i \in C_{parallel}\}, \quad (2.7)$$

де D – стратегія розгортання, $C_{parallel} \subseteq C$ — підмножина контейнерів, які оновлюються паралельно, P – платформа, E – середовище. Це дозволяє зберігати працездатність платформи P під час оновлення кількох контейнерів одночасно, при цьому середовище E продовжує забезпечувати виконання на всіх контейнерах.

6. Завершальним етапом є (постійний) моніторинг та логування використання впровадженої платформи. Реалізується безперервне діагностування параметрів продуктивності, акумуляція системних метрик та реалізується автоматизована реакція на потенційні відмови. Під відмовами розуміються відмови як апаратного або програмного рівня. Стратегії відновлення працездатності системи включають або повне, або часткове виконання конвеєру,

для відновлення працездатності. Особливостями відмов в задачах ШІ є те, що на відміну від задач класичного програмування, потрібно встановити межу, коли модель буде вважатись, що модель працює недостатньо коректно. Збір і акумуляція метрик, таких як час відновлення, завантаженість ЦП, частота відмов, у часі можна описати як множину функцій, що агрегують вимірювані параметри:

$$M(t) = \{m_1(t), m_2(t), \dots, m_k(t)\}, t \in [t_0, t_1], \quad (2.8)$$

де $M(t)$ — це вектор значень усіх метрик у момент часу t .

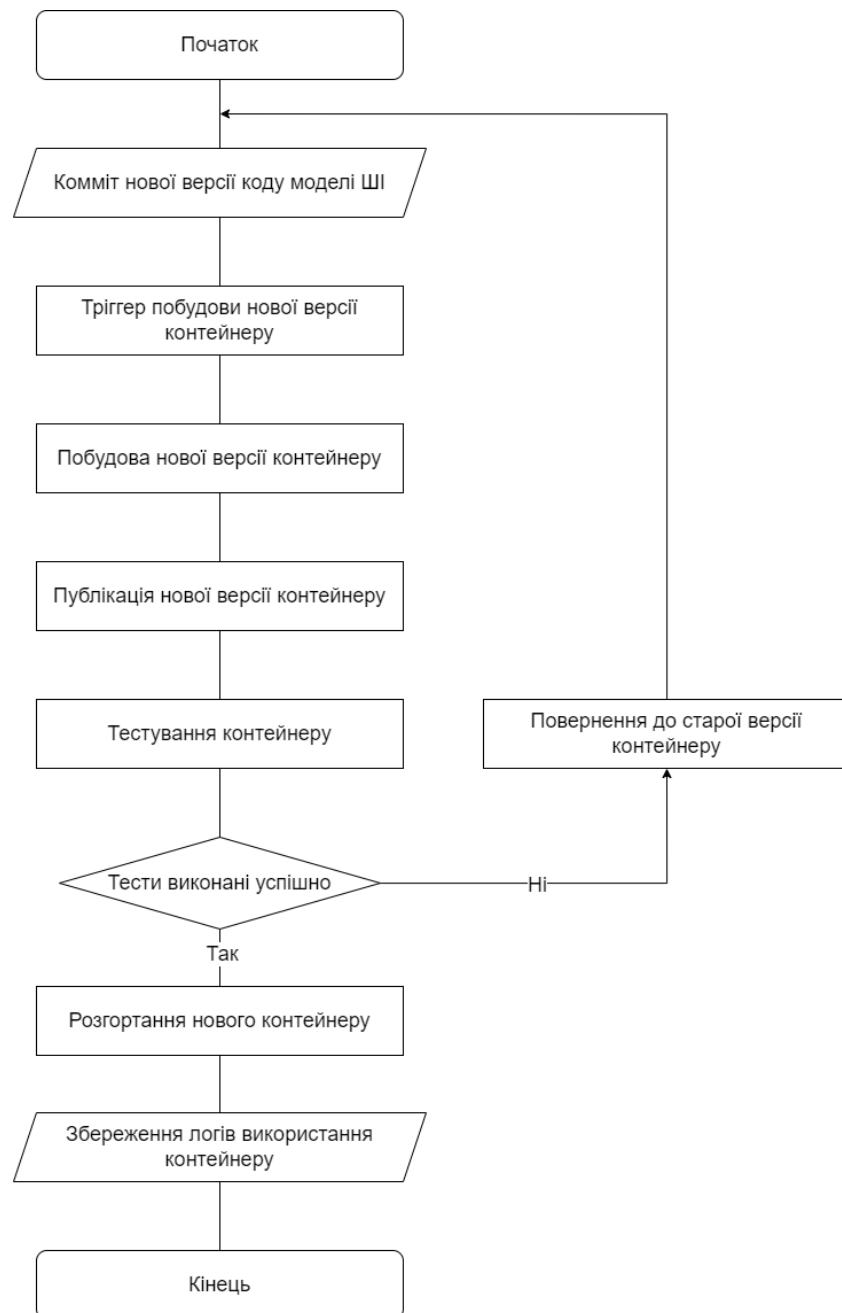


Рис. 2.4. Блок-схема алгоритму розгортання системи шляхом побудови контейнерів

Наявний шлях від оновлення моделі в рамках інструментарію *MLOps* до подальших дій у *DevOps* показаний на рисунку 2.4. Модель, яка пройшла тест на точність у цьому процесі, надсилається диспетчеру моделі разом із відповідним звітом про тестування. Менеджер, який затверджує модель, має бути закріплено в репозиторії кінцевих програм [32]. Ця фаза знаменує початок процесу *CI/CD* кінцевої програми *ML*.

Конвеєр *CI/CD* для системи ШІ зазвичай починається з етапу інтеграції. Це передбачає витягування останніх змін коду з системи контролю версій, наприклад, *Git*, і об'єднання їх у спільну гілку. Потім запускаються автоматизовані тести для перевірки змін у коді. У випадку системи штучного інтелекту ці тести можуть включати модульні тести для окремих компонентів, а також інтеграційні тести, які оцінюють продуктивність моделей машинного навчання за набором попередньо визначених метрик.

Після того, як зміни коду пройшли всі тести, вони можуть бути вбудовані в контейнер. Для системи ШІ цей контейнер може включати не лише код програми, але й навчені моделі машинного навчання та будь-які необхідні залежності. Етап розгортання конвеєра *CI/CD* передбачає розгортання цього контейнеру в тестовому середовищі (або середовищах), де можна проводити подальші випробування. Ці тести можуть включати запуск симуляцій або використання реальних даних для оцінки продуктивності моделі. Це середовище буде доступним для розробників для ручної перевірки працездатності системи.

Нарешті, після проходження всіх тестів у тестовому середовищі, контейнер можна розгорнути у виробництво. Це розгортання може відбуватися автоматично, після успішної збірки в конвеєрі *CI/CD*, або вручну. Важливо забезпечити належний моніторинг і ведення журналів, щоб відстежувати продуктивність системи у виробництві та швидко виявляти будь-які проблеми, які можуть виникнути.

В рамках даної роботи, частину процесу, як ручне тестування та розробку коду з її подальшим відвантаженням на систему контролю версії буде усунено, оскільки фокусом даної роботи є процес саме розгортання компонент автоматизованої платформи вбудованих систем, прийняття рішень з точки зору програмного коду продукту є іншою задачею.

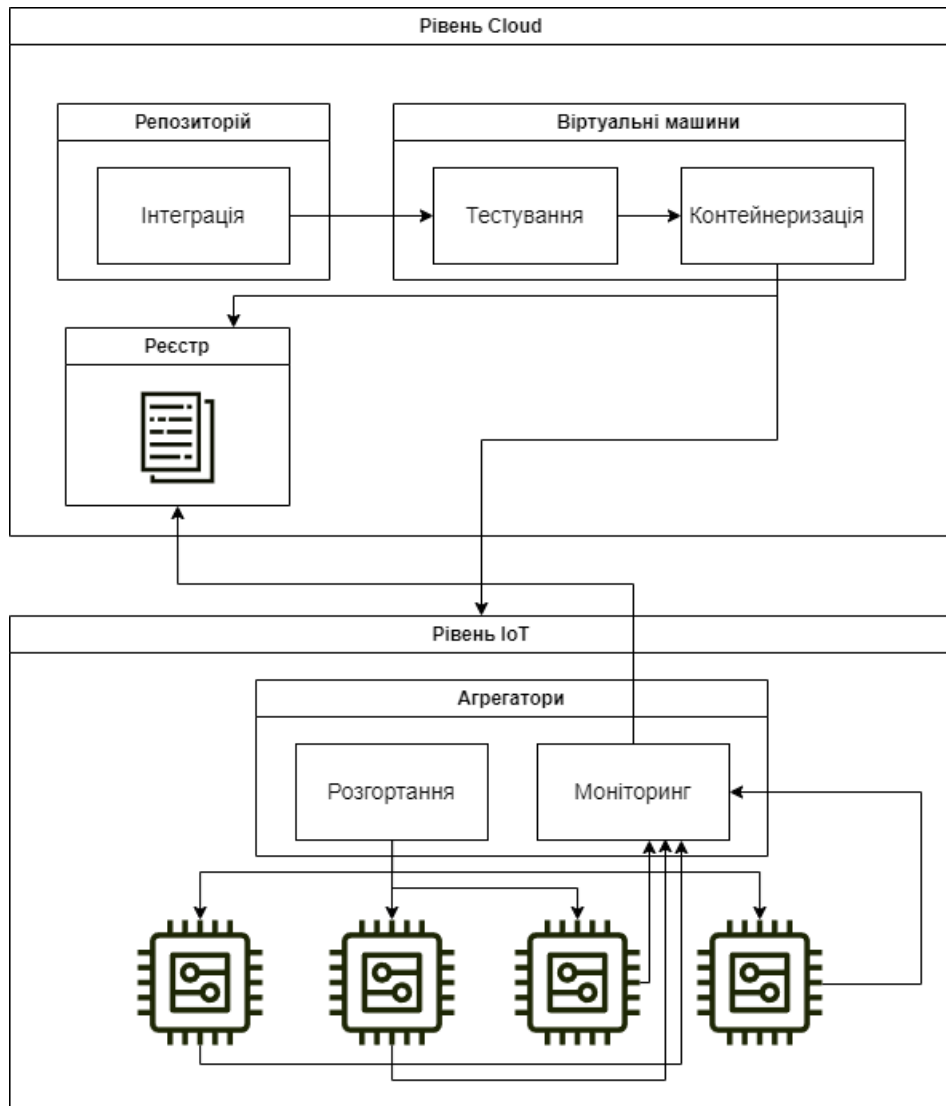


Рис. 2.5. Алгоритм *CI/CD* конвеєру

Отже, створення конвеєра *CI/CD* для системи ШІ має складатися з наступних етапів, показаних на рисунку 2.5.: інтеграція (отримання коду), тестування та контроль якості, збирання контейнеру, розгортання сервісів на вузлах та моніторинг. Впроваджуючи принципи *MLOps* в конвеєр, розробники отримують гарантію, що їхні системи ШІ розробляються, тестуються і розгортаються надійно і з можливістю масштабувати систему. Впровадження

такого конвеєра не тільки підвищує продуктивність, але й покращує загальну якість і продуктивність систем III. Розробник конвеєру може вносити зміни до процесу в залежності від зміни архітектури системи, наприклад, додаткові зміни можуть впровадити автоматичну систему безпеки коду, перевірки на дотримання стандартів написання коду та інше.

2.4. Метод розгортання компонент III-платформи на основі онтології процесів

AI-платформа являє собою інтегровану сукупність компонентів, виражених як ресурсно-інтенсивні архітектурних рішень (PIAP). Вказані PIAP можуть являти собою певні вбудовані пристрої, процеси, інформаційні ресурси та інші мережеві об'єкти. Усі ці системні складові (SC) відображаються певною семантикою (S), яка може бути вербально представлена описами архітектурних та функціональних характеристики об'єктів, сукупність яких визначає функціональні спроможності конкретної PIAP. Кожен об'єкт, що входить до складу певного PIAP, може бути представлений сукупністю семантично зв'язаних між собою концептів [81].

Представлення кожного PIAP на основі концептів семантики S , дозволяє застосувати методологію онтологічного інжинірингу для відображення архітектурних та функціональних характеристик, атрибутів та властивостей компонент III-платформи [82]. Такий підхід до інтеграції PIAP дозволяє відображати кожен SC та її семантику S у форматі онтології O [83].

Методологія *MLOps*, яка поєднує засоби штучного інтелекту і методів машинного навчання з IT-операціями, складає агрегацію складних логістичних ланцюгів процесів. Тому набори практик, що її реалізують, можна представляти у форматі онтології процесів O_{pr} [84], які вона інтегрує у певному часі. Такий підхід щодо формування AI-платформ, являє собою приклад застосування методології онтологічного інжинірингу, якій реалізує активне відображення знань, що характеризують семантику програмного забезпечення, технологій підприємств та бізнес-процесів певних предметних галузей [12]. Фактично усі завдання, що

реалізуються за методологією *MLOps* можуть бути представлені онтологіями процесів, які ці завдання інтерпретують.

Онтологія є системною складовою довільної практики чи процесу формування інформаційного середовища (ІС). Вона забезпечує формальне представлення концептуальних знань про предметну область. Процес побудови такої складової ІС можна представити композицією певних висловлювань, суджень, тверджень, термінів-понять і відношеннями між ними, а його результат – основою для побудови складової частини аксіоматизованої теорії – онтологічної бази знань у заданій предметній області, яка включає декларативний формат опису знань й імперативний формат їхньої актуалізації.

Онтології відображають усі архітектурні та функціональні властивості об'єктів, що складають ІІІ-платформу. Кожна онтологія може бути представлена у форматі графу [85], як одного з доведених ефективних способів відображення структурних властивостей об'єктів довільної природи та їхньої функціональності. Враховуючи той факт, що кожна онтологія включає до себе як мінімум одну таксономію [86], розглянемо подальше цю категорію щодо відображення семантики *S* РІАР, які активно реалізуються у процесі формування ІІІ-платформи.

Розглянемо в якості технологічної основи таксономій – дерева Бема Σ у форматі виду (1) [87]

$$\Sigma = \{\perp\} \cup \{\lambda x_1, \lambda x_2, \dots, \lambda x_n, \lambda a_1, \lambda a_2, \dots, \lambda a_m\} \quad (2.9)$$

Де Σ – множина імен концептів, що складають семантику *S* усіх складових РІАР, λx – терми, які реалізують представлення об'єктів РІАР у безтипової нотації ламбда-числення, $\perp$ – найменший елемент із усіх значень контекстів концептів/вузлов таксономії.

Згідно [87] дерева Бема є індуктивними, вони можуть породжувати новітні терми вузли, кожен з яких є ім'ям певного концепту, що визначає семантику *S* РІАР, й які є їхніми композиціями. Згідно нашого формалізму композиції складаються з λx -термів, які мають контекстну зв'язність. У процесі представлення концептів РІАР у форматі λx – термів, ці терми наслідують ім'я концептів, яких вони заміщують. На основі цього можна стверджувати що

таксономії є поміченими деревами Бема. Такі дерева мають властивість рефлексивності, тобто справедливим є вираз:

$$\varphi(\Sigma) \rightarrow \Sigma \quad (2.10)$$

де φ – рефлексивне відображення Σ самого на себе за умовою, що відповідні контексти λ -термів, що представляють концепти РІАР визначені. Тоді вся сукупність λ -термів, які представляють семантику S РІАР, відображає активність всіх практик та процесів, що функціонально складають мегапроцес *MLOps*. Більш того активність λ -термів реалізує виявлення зв'язності контекстів семантик S , як складових РІАР.

Для більш детального представлення семантики S зробимо наступну інтерпретацію поняття концепт. Згідно [88, 89] всі концепти, на основі яких формується таксономія можуть бути представлені у форматі λx -термів. Тоді відображення знань, що характеризують семантику обчислення смислового характеру можна представити контекстами значень λx -термів-концептів, що визначають умови застосування правил методології *MLOps*, яка реалізує агрегацію складних логістичних ланцюгів процесів AI-платформи.

Наявність контекстів у кожного концепту таксономії реалізує певний процес взаємодії між системними компонентами SC , які формують РІАР та визначають їхню функціональність. Тобто кожну SC , що представлена у форматі простої ієрархії концептів, можна представити у форматі ієрархії класів λ -термів-концептів й як наслідок – у форматі дерев Бема Σ . Таким чином наступний функціональний вираз є коректним:

$$\{X_1, X_2 \dots X_n, a_1, a_2 \dots a_n\} \rightarrow \lambda \rightarrow \check{T} \rightarrow \Sigma = \{X_1, X_2 \dots X_n, a_1, a_2 \dots a_n\}, \quad (2.11)$$

де X – концепт чи клас концептів, n, m – індекси, $\check{T}$ – непуста множина таксономій.

На основі вищевикладеного можна стверджувати, що кожна системна складова SC множини, на основі своєї таксономії чи їх множин, може бути представлена у форматі дерева Бема.

$$\{X_i [] \mid i \in \langle \overrightarrow{1, n} \rangle \} \rightarrow \{X_i [A_j], X_n [A_v, A_p] \mid i \in \langle \overrightarrow{1, n} \rangle \mid j \in \langle \overrightarrow{1, m} \rangle : \quad (2.12)$$

$$\overrightarrow{1, k} \rangle \mid p \in \langle \overrightarrow{1, l} \rangle \} \rightarrow \check{T},$$

де A_j, A_v, A_p – лінгвістичні константні значення контекстних смислів концептів семантик S .

Формування дерева Бема на основі виявлення ієрархій концептів, що визначають семантики S у РІАР, як системоутворюючих *MLOps*-середовища, реалізується наступним чином [87, 90]:

- визначається довільний концепт, що має визначений контекст типу $x[a]$, де a – константа, яка являє собою певне значення: літера, число, лексема, певна фраза, речення та частково упорядкована множина речень тощо;
- всі подібні лексичні структури далі визначаються як λx -терми;
- з множини λx -термів виділяється одна послідовність символів, яка складає певне слово, що є концептом ієрархії семантики S , який неможливо редукувати. Такий терм визначається як найменший/первинний в ієрархії й позначається символом $\perp$;
- визначаються інші λx -терми, які відображають вибрану множину концептів, що визначають усі контексти семантики S , що представими в РІАР;
- над множиною вибраних λx -термів визначається відношення часткового порядку – δr ;
- всі вибрані концепти семантики S , що представлені в РІАР визначаються як відповідні λ -терми;
- на основі визначення над множиною вибраних λ -термів відношення часткового порядку δr формується дерево Бема в форматі виду (2);
- $\perp$ – найменший елемент із усіх значень контекстів концептів дерева Бема.

Як бачимо, у процесі формування дерева Бема кожен концепт визначається множиною контекстів, які фактично визначають правила *RL* реалізації семантик S системних складових *SC* РІАР для агрегації ІІІ-платформи.

$$\{X_i \mid i \in \overrightarrow{1, n} \mid a_j \mid j \in \overrightarrow{1, m} \} \rightarrow \lambda x \rightarrow \check{T} \rightarrow \Sigma = \{X_i \mid i \in \overrightarrow{1, n} \mid a_j \mid j \in \overrightarrow{1, m} \} \quad (2.13)$$

Вирази (3), (5) визначають системологічну стійкість концептів, що складають семантики S практик та процесів $MLOps$, що підтверджується істинністю тверджень, які з них можна сформулювати, тобто λ -терми, що описують ці практики та процеси, є розв'язними. Це означає, що розбивка множини λ -термів, яка складається за властивостями концептів семантик S в конкретному стані, визначає їхню ієрархію. Така розбивка також враховує розподіл концептів по різних класах, за рахунок чого підвищуються їх класифікаційні ознаки в Σ та, як наслідок, системологічна стійкість та точність [91].

Функціональний вираз (1) в себе містить ще одну компоненту, а саме – частковий порядок ∂r , застосування якого визначає послідовність розташування контекстів кожного концепту та кожного правила.

Представлення контекстів у форматі імперативних правил RL , утворює умови агрегації AI-платформи на основі переходу від контекстної зв'язності до визначення порядку виконання правил системних складових SC множини PIAP. При цьому процедура машинного навчання мегапроцесу $MLOps$ може бути застосовано як на етапі формування таксономій у форматі Σ , так й на етапі після встановлення контекстної зв'язності визначених імперативних правил RL . Фактично, визначені при формуванні таксономій процесів $MLOps$, імперативні правила RL , є системними компонентами SC_{rl} процесів практик $DevOps$. Умови формування практик $DevOps$ та їхнього функціонального розширення до мегапроцесу $MLOps$, можна визначити на основі онтологій процесів.

Довільний процес можна представити у наступному форматі: *правило* $\rightarrow$ *результат*. Його можна також представити певною функцією виду: Множина правил F може бути утворена декартовим добутком множин концептів X , що визначають семантики S для PIAP та властивостями концептів R :

$$F = X \cdot R, \quad (2.14)$$

де F , визначається як функція інтерпретації властивостей концептів X . Тоді згідно [14] завжди існує певний набір правил $F_k \subset F$ таких, що завжди існує хоча б одне непусте правило $f^i \in F_k$ таке, що існує також набір концептів X_j для яких правило

$f^i(x_1, x_2, \dots, x_n) \in F_k$. При чому до правила f^i завжди можна додати узгоджений новий концепт (x_{n+1}) з новою для F_k властивістю r' , при наявності якої виконується нове правило $f^i(x_1, x_2, \dots, x_n, x_{n+1}) = F_k$. Узгодженість концептів визначається їхньою контекстною зв'язністю.

На основі набору описаних правил F_k визначимо онтологію процесів O_{pr} . Всі правила, які визначаються контекстами концептів семантик S для РІАР, можуть бути представлені певними програмними модулями, що складають практики *DevOps* й як наслідок процеси *MLOps*.

$$O_{pr} = \langle X_{pr}, R_{pr}, F_k(X_{pr}, \text{ör}, Attr_{pr}), Ax_s \rangle, \quad (2.15)$$

де: O_{pr} – онтологія процесу; $Attr_{pr}$ – атрибути практик; Ax_s – базові аксіоми практик, які визначають функціональність найменших елементів із усіх значень контекстів їхніх концептів.

Об'єкти онтології O_{pr} забезпечують генерацію етапів використання практик *DevOps* й як наслідок процесів *MLOps*. Вони мають складну ієрархічну структуру у форматі композицій дерев Беа Σ . Правила F_k , що визначено онтологією процесу O_{pr} , забезпечують виконання усіх практик, які визначено у *DevOps* й як наслідок у процесах *MLOps*.

Онтологічний формат представлення практик і правил надає ряд переваг для реалізації вказаних практик і процесів, а саме:

- адаптація до задач предметних галузей практик, що визначає гармонізацію їхньої взаємодії;
- масштабованість – включення в контур практик і процесів новітніх контекстно зв'язаних онтологій;
- візуалізація – представлення практик і процесів у форматі графів та дерев, що забезпечує виконання умов реалізації вимог *DevOps* й як наслідок *MLOps*;
- оптимізація практик і процесів, за рахунок їхньої контекстної зв'язності; виявлення слабких місць у взаємодії практик і процесів;
- виявлення впливів практик і процесів між собою та на задачі, що розв'язуються;

- формування новітніх вимог до реалізації практик та процесів.

В методології *MLOps*, онтології процесів мають певну двоїстість. По-перше вони можуть бути згенеровані на першому етапі машинного навчання. Для цього процедури машинного навчання повинні вміти проводити морфологічний, лексикографічний й концептографічний аналіз описів практик та процесів. По-друге – онтології процесів O_{pr} можуть бути використані на наступних етапах машинного навчання для підвищення рівня семантичної взаємодії між практиками та процесами.

Ще однією особливістю представленої онтології процесів O_{pr} є їхня адаптивність під різні формати представлення графових структур. Цьому сприяє їхня топологічна основа у форматі дерев Бема. Фактично довільне дерево Бема можна представити у форматі довільного графу без циклів [81]. Однак при відображенні процесів графової структури з повторами, з дерев Бема можна сформувати складні композиції, які відтворюють вкладеність конкретних семантик S у практики та визначені процеси. У такому випадку дерева Бема ніби розчиняються в середовищі практик, являючи собою первинні топології для *MLOps* середовища. Вони у такому випадку представляють множини концептів з заданим частковим порядком δ_r , який визначає послідовність актуалізації конкретних концептів, що складають семантику практик та процесів.

Однією з системних інтерпретацій такого підходу може служити формування графів де Бруйна, як технологічної основи реалізації практик *MLOps* середовища. Виявлені міжконтекстні зв'язки концептів семантик S , які складають практики та процеси *MLOps* середовища, визначають певні послідовності імен, які функціонально активні при актуалізації конкретної практики та процесу. Такі послідовності типу (3) – (5) можуть бути виділені як на етапі первинного машинного навчання, так й у процесі генерації інформаційного середовища, сервіси якого забезпечують розв'язання визначених задач.

Технологічним наслідком цього є генерація зв'язних фрагментів графа де Бруйна, кожен з яких представлений виявленими при формуванні дерев Бема ієрархічних послідовностей імен концептів. Ці послідовності мають високій

рівень валідності на основі того, що вони виявлені на основі врахування семантичної зв'язності контекстів, що визначають семантику концептів, які є системними складовими *SC* практик та процесів *MLOps* середовища. Ця зв'язність забезпечує наступність при визначенні технологічних ланцюгів активності семантик практик та процесів *MLOps* середовища, що реалізується активністю конкретних програмних модулів, що сформовані з правил типу $F_k \subset F$.

Виходячи з вищенаведеного зробимо ще одне твердження. Онтології процесів O_{pr} забезпечують реалізацію формування ІІІ-платформ на основі валідного відображення всієї зв'язності концептів, які визначають семантики *S* практик та процесів *MLOps* середовища. Також онтології процесів O_{pr} реалізують семантичне покриття *MLOps* середовища, що забезпечує її стійкість та спроможність щодо семантичного розширення при виявленні у процесі машинного навчання новітніх концептів, що визначають семантику *S* практик та процесів *MLOps* середовища.

2.5. Процес розгортання та моніторингу компонент платформи вбудованих систем з можливостями ІІІ

Розгортання є ключовим процесом в рамках *MLOps*. На цьому етапі розробники програмного забезпечення інтегрують перевірені моделі у відповідні додатки та забезпечують стабільність роботи всієї системи додатків. Оскільки аспекти різних рішень і конфігурацій усього конвеєра можна отримати на цьому інтеграційному етапі, постійний моніторинг моделі, загальної програми та споживаних даних повинен здійснюватися інженерами *MLOps* [34]. Іншу роль на цьому етапі розгортання інфраструктури відіграє інженер *DevOps*, який відповідає за проведення, створення та тестування робочої системи.

Існують різні техніки, в яких розгортається навчена модель. Двома найпоширенішими методами є «модель як послуга» та «вбудована модель». У «модель як послуга» модель представлена як кінцеві точки *API* передачі стану представлення (*REST*), тобто розгортання моделі на веб-сервері, щоб вона могла взаємодіяти через *REST API*, і будь-яка програма могла отримувати прогнози,

передаючи вхідні дані через виклик *API* [34]. Веб-сервер може працювати локально або в хмарі. З іншого боку, у випадку «вбудованої моделі» модель упаковується в програму, яка потім публікується [20]. Цей варіант використання практичний, коли модель розгортається на периферійному пристрої. Зауважте, що спосіб розгортання моделі *ML* повністю залежить від взаємодії кінцевого користувача з результатом, створеним моделлю.

Постійний моніторинг усього стеку проекту, наприклад, продуктивності моделі та програми, а також інфраструктурних обставин і (найголовніше) даних, які використовує модель, є основою надійного продукту на основі машинного навчання [21]. Використовуючи метрики, запропоновані в Розділі 2, багато операційних помилок і недоліків можна компенсувати заздалегідь. З фокусом на відстеження даних через конвеєрні операції [34] графові бази даних можуть допомогти в управлінні та підтримці зв'язку між об'єктами даних і відповідними твердженнями.

2.5.1. Шаблони розгортання моделі ШІ для платформ вбудованих систем

Кінцевим результатом процесу *MLOps* – є розгорнуті компоненти платформи вбудованих систем, що включає модель ШІ. Це означає, що модель готова до використання і всі необхідні залежності встановлені, і вона може почати надавати прогнози або виконувати інші завдання, для яких вона була створена. Важливо, що процес розгортання моделі має бути надійним та ефективним, щоб забезпечити безперервну роботу системи та максимальну користь від розробленої моделі, тому існують декілька способів подачі моделі [13].

Коли ми подаємо ШІ модель у виробничому середовищі, слід враховувати три компоненти. Висновок — це процес отримання даних, які поглинає модель для обчислення прогнозів. Цей процес вимагає моделі, інтерпретатора для виконання та вхідних даних [13]. Розгортання ШІ платформ у виробничому середовищі включає два аспекти: спочатку розгортання конвеєра для

автоматизованого перенавчання та розгортання ШІ моделі. По-друге, надання *API* для прогнозування невидимих даних.

Подача моделі (*model serving*) — це спосіб інтеграції моделі *ML* у програмну систему. Ми розрізняємо п'ять шаблонів для впровадження моделі *ML* у виробництво: *Model-as-Service*, *Model-as-Dependency*, *Precompute*, *Model-on-Demand* і *Hybrid-Serving*. Потрібно зауважити, що описані вище формати серіалізації моделі можуть використовуватися для будь-яких шаблонів обслуговування моделі.

Модель як послуга (*Model-as-Service*) — це загальний шаблон для упаковки моделі *ML* як незалежного сервісу. Один із варіантів використання цього підходу, це обернути ШІ модель та інтерпретатор у спеціальну веб-службу, яку програми можуть запитувати через *REST API* або використовувати як службу *gRPC*.

Модель як залежність — це спосіб, в який упакована ШІ модель розглядається як залежність у програмному додатку. Наприклад, програма використовує ШІ модель як звичайну залежність *jar*, викликаючи метод передбачення та передаючи значення. Поверненим значенням виконання такого методу є деяке передбачення, яке виконується попередньо навченою ШІ моделлю. Підхід моделі як залежності в основному використовується для реалізації шаблону прогнозу.

Попередньо обчислений шаблон обслуговування – цей тип обслуговування моделі *ML* тісно пов'язаний із робочим процесом *Forecast ML*. З шаблоном обслуговування *Precompute* ми використовуємо вже навчену модель *ML* і попередньо обчислюємо прогнози для вхідного пакету даних. Отримані прогнози зберігаються в базі даних. Тому для будь-якого вхідного запиту ми надсилаємо запит до бази даних, щоб отримати результат передбачення.

Шаблон *Model-on-Demand* модель по необхідності також розглядає модель *ML* як залежність, яка доступна під час виконання. Ця модель *ML*, на відміну від шаблону *Model-as-Dependency*, має власний цикл випуску та публікується

незалежно. Такі залежності можуть використовуватись в інших додатках або ШІ платформах а також в бібліотеках, які підключаються безпосередньо до іншого ПЗ.

Архітектура посередника повідомлень зазвичай використовується для такої моделі обслуговування на вимогу. Шаблон архітектури топології брокера повідомлень містить два основних типи компонентів архітектури: компонент брокера та компонент процесора подій. Компонент брокера є центральною частиною, яка містить канали подій, які використовуються в рамках потоку подій. Канали подій, які містяться в компоненті брокера, є чергами повідомлень. Процесор подій містить середовище виконання моделі та модель *ML*. Цей процес підключається до брокера, по пакетно зчитує ці запити з черги та надсилає їх моделі для створення прогнозів. Процес обслуговування моделі запускає генерацію прогнозу на вхідних даних і записує отримані прогнози в вихідну чергу.

Гібридне обслуговування є іншим способом надання моделі користувачам. Вона унікальна в тому що існує стільки моделей, скільки існує користувачів, крім тієї, яка зберігається на сервері. Модель на стороні сервера навчається лише один раз із реальними даними. Сервер встановлює початкову модель для кожного користувача. Крім того, це відносно узагальнена модель, тому вона підходить для більшості користувачів [70]. З іншого боку, є моделі на стороні користувача, які є справжніми унікальними моделями. Час від часу пристрої надсилають свої вже навчені дані моделі на сервер. Там модель сервера буде налаштована, таким чином модель охопить актуальні тенденції враховуючи дані з логів. Ця модель стане новою початковою моделлю, яку використовують усі пристрої.

2.5.2. Метрики оцінки відмовостійкості компонентів платформи

Для порівняння різних методів та підходів, при різних конфігураціях системи, необхідно впроваджувати метрики. Метрики є ключовим інструментом для оцінки результатів та визначення ефективності в *MLOps* задачах.

Обґрунтованість та системність цих метрик зумовлює не тільки практичну цінність запропонованого рішення, але й наукову цінність. Також, здатність відтворити метрики, особливо з точки зору ШІ задач, дуже цінне, оскільки, існує необхідність в постійному покращенні стану ШІ-платформи за рахунок впровадження змін до частин системи.

Існують різні метрики і вони характеризують різні частини системи. Метрики, наприклад точність на раніше невідомих даних, надають можливість чисельно визначити, наскільки добре працює модель. Це дозволяє аналізувати її ефективність та порівнювати з іншими альтернативами. Логування дозволяє здійснювати постійний моніторинг функціонування моделі під час її роботи в реальному середовищі. Це важливо для виявлення відхилень та алгоритмічних проблем, що можуть виникати у процесі або розгортання системи або її використання. Аналіз метрик, після повноцінної *end-to-end* роботи конвеєра, дозволяє ідентифікувати слабкі місця в процесі розробки та впроваджувати зміни до процесу розгортання, що дозволяє їх покращувати як за відмовостійкістю так і впроваджувати оптимізацію часу.

Під відмовостійкістю системи, в рамках даної роботи, вважатиметься здатність системи продовжувати стабільну роботу. В практичному полі, така система має автоматично, де і на скільки це можливо без втручання обслуговуючого персоналу, підлаштовуватись під нову апаратну конфігурацію та продовжувати виконувати поставлені задачі. Питання фізичної реструктуризації мережі майже неможливе, тому основний підхід до адаптації враховує можливості системи реплікувати задачі з відмовлених вузлів та делегувати їх на інший вузол, який працює. Окрім апаратного рівня, також рівень програмний не менш важливий, оскільки при некоректній реалізації будь-якого з елементів конвеєру *MLOps* можливий сценарій коли буде блокуватись робота системи за рахунок того, що вузли будуть отримувати некоректні завдання. Самі задачі, які посилаються на вузли, мають бути написані таким чином, що алгоритм їх

виконання не повинен блокувати вузол. Це питання особливо важливе для вузлів-агрегаторів на проміжних рівнях системи. Оцінка відмовостійкості є дуже складним процесом, що зумовлено широкою низкою факторів, які впливають на відмовостійкість системи,. Визначення відмов можна описати через функцію δ , яка сигналізує про наявність відмови на рівні апаратного або програмного забезпечення:

$$\delta(P, E, C_i) = \begin{cases} 1, \text{ якщо відмова (апаратна чи програмна) на рівні } P, E, C_i \\ 0, \text{ якщо система працює коректно} \end{cases}, \quad (2.16)$$

де P – платформа, E – середовище виконання коду та C – компонента платформи. У контексті ІІІ-задачі моделей відмова може означати, що модель працює недостатньо коректно. Позначимо θ як показник працездатності моделі, який визначає межу того, коли модель вважається неспроможною до подальшого використання.

$$\theta(C_i) = A(C_i) - T, \quad (2.17)$$

де $A(C_i)$ — точність роботи моделі, контейнеризованій в C_i , а T — межа точності, при якій модель вважається некоректною.

Різні метрики допомагають визначити стабільність та надійність системи в різних умовах роботи. З точки зору вбудованих пристроїв та створення гібридної архітектури, такі метрики як використання пам'яті та час виконання конвеєру є ключовими. Обмежені ресурси пам'яті можуть обмежувати ефективність моделі та збільшувати час виконання. Метрики використання пам'яті допомагають ідентифікувати ресурсоємні частини моделі та вчасно виявляти проблеми з пам'яттю, що дозволяє вирішувати питання їх оптимізації для ефективної роботи на вбудованих пристроях.

Враховуючи, що конвеєр має багато етапів, має сенс розглядати метрику часу, як суму часу виконання різних кроків виконання, що в свою чергу допоможе

виокремити етапи, які потребують модифікації та оптимізації. Формула для обчислення буде наступною:

$$T = T_t + T_b + T_d, \quad (2.18)$$

де T_t – час для виконання тестів, T_b – час для виконання зборки контейнеру, T_d – час для розгортання системи.

Нажаль, неможливо точно встановити заощаджений час, використовуючи представлений метод, відносно систем, які його не використовують, оскільки, фактично, деякі кроки або не виконуються, або виконуються людиною. Такі кроки як інтеграція і моніторинг не будуть входити в загальну метрику. Тому, замість залучення порівняльної метрики прискорення, відносно класичної ІІІ системи, тобто такої, яка не застосовує практики *MLOps*, пропонується розгляд метрики ефективності системи, відносно найбільш простої конфігурації – виконання всіх кроків на одному вузлі (сервері). Ефективність розраховується за наступною формулою:

$$E = \frac{T_s}{T_n}, \quad (2.19)$$

де T_s – час виконання конвеєру ІІІ-платформи, T_n – час виконання всіх кроків конвеєру на вузлі n .

Останні метрики, які можна запропонувати при впровадженні даного методу в реальних умовах, це метрики *DORA* [71]. Вони складаються з чотирьох контрольних показників:

- Частота розгортання (*Deployment frequency* - *DF*): як часто організація успішно випускає продукт для користувачів або розгортає його у виробництві.
- Час виконання змін (*Lead time to changes* - *LT*): час, потрібний коміту для досягнення продуктивності або випуску.
- Середній час відновлення обслуговування (*Mean time to restore service* - *MTTR*): скільки часу потрібно організації для відновлення після збою у виробництві.

- Зміна частоти відмов (*Change failure rate* - *CFR*): відсоток випусків або розгортань, які викликають збій у виробництві.

В рамках даної роботи можна розглянути метрику *MTTR*, оскільки інші метрики мають більш прикладний характер і виміряють продуктивність з точки зору змін в алгоритмі ПЗ, а не системи в цілому.

Висновки до розділу 2

В даному розділі досліджений метод розгортання компонент платформи з урахуванням архітектурних особливостей вбудованих систем. Окремим викликом поставленим в рамках дисертаційного дослідження є застосування вбудованих (*Edge*) пристроїв. Використання *Edge* архітектури завжди є ризикованим через свою досить низьку надійність в порівнянні з класичними серверами і *Cloud*-системами. Вирішенням цього питання є використання архітектури *Edge-Cloud*. Ця архітектура вирізняється поєднанням обчислювальних ресурсів на *Edge* та у хмарному середовищі, що дозволяє оптимізувати роботу ШІ-платформи, шляхом застосування вбудованих пристроїв, тим самим зменшуючи затримки і підвищувати продуктивність об'єднавши вбудовані пристрої з *Cloud*-серверами в одну мережу.

Процес розгортання ШІ-платформи є важливим, оскільки необхідно залучити кожен елемент мережі до виконання завдань. Ми розглядаємо *MobileNet* в якості базової моделі, оскільки ця модель демонструє високі показники точності при значному зменшенні використання ресурсу пристрою на якому модель застосовується. Для того, щоб розгорнути систему, було обрано метод створення *CI/CD* конвеєру як найбільш доцільний шлях досягнення даної мети. Описано кроки виконання конвеєру та питання постійної підтримки системи в разі відмов вузлів, або падіння точності. В рамках *MLOps*, цей конвеєр виконується автоматично та поетапно, і можна відслідковувати кожен етап, за допомогою чого приймаються рішення щодо наступних кроків.

Для демонстрації правильності обраної методики та порівняння з іншими та майбутніми методами, запропоновано ряд метрик, які кількісно показують працездатність системи. За допомогою логування, кожен етап розгортання, а в подальшому використання цієї системи, має кількісне відображення. Завдяки досягненням в області *DevOps*, існує ряд метрик, за допомогою яких можна встановити швидкість та якість виконання конвеєру. Окрім цього, точність будь-

якої моделі завдає втрат з плином часу та ростом раніше не оброблених даних. Можливість доступу до точності моделі, після її випуску має дуже важливе значення, оскільки велика кількість досліджень в напрямку ШІ, фактично, не може точність розроблених моделей на нових даних, які потребуватимуть або донавчання, або повної переробки моделі.

РОЗДІЛ 3

ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ РОЗГОРТАННЯ КОМПОНЕНТІВ ПЛАТФОРМИ ВБУДОВАНИХ СИСТЕМ

3.1. Застосування технології *Infrastructure as code*

Технологія Інфраструктура як код (*Infrastructure as code – IaC*) все частіше застосовується в системах штучного інтелекту, використовуючи принципи *MLOps*. *IaC* передбачає управління та забезпечення обчислювальної інфраструктури за допомогою коду замість ручних процесів. Основна ідея даного підходу заключається в тому, що дана парадигма інкапсулює дані про апаратне забезпечення в змінні та функції. Використовуючи *IaC* для *MLOps*, організації можуть отримати вигоду від прискореного розгортання, узгодженості та відтворюваності, враховуючи, що архітектура системи буде зберігатись в якості коду, та економії витрат. Такий код має мати властивості як ідемпотентність, структурованості. Використання інфраструктурного коду дозволяє швидше та ефективніше розгортати платформи вбудованих систем та забезпечує різних середовищ розробки. Це зменшує ймовірність помилок і гарантує, що кінцева система працюватиме належним чином у різних середовищах. Крім того, використання *IaC* в рамках *MLOps* дозволяє організаціям легко відтворювати та масштабувати свою інфраструктуру штучного інтелекту за потреби.

Щоб ефективно впровадити *IaC* для *MLOps*, слід дотримуватися кількох практик. Перш за все, використання системи контролю версій має велике значення для відстеження змін у коді інфраструктури та співпраці з іншими членами команди. В минулому розділі було розглянуто метод, за допомогою якого можна інтегрувати такий код в рамках *CI/CD* конвеєру. Це допомагає підтримувати історію змін і дозволяє командам безперешкодно працювати разом. Забезпечення якості коду та тестування також має важливе значення для виявлення та усунення будь-яких проблем перед розгортанням інфраструктури.

Модулізація коду полегшує управління та повторне використання компонентів, що робить розробку та підтримку інфраструктури більш ефективною. Крім того, використання інструментів автоматизації може впорядкувати процес забезпечення та зменшити кількість ручної роботи та помилок, пов'язаних із внесенням ручних змін.

Існує кілька популярних інструментів для впровадження *IaC* в системах штучного інтелекту. *Terraform* - це широко використовуваний інструмент, який дозволяє користувачам визначати та надавати інфраструктуру різних хмарних провайдерів. *AWS CloudFormation* - ще один популярний вибір для автоматизації розгортання ресурсів *AWS*. *Azure Resource Manager* надає аналогічні можливості для *Microsoft Azure*, а *Google Cloud Deployment Manager* призначений для надання ресурсів на *Google Cloud Platform*. Ці інструменти пропонують низку функцій та інтеграцій, які полегшують визначення, розгортання та управління інфраструктурою, необхідною для систем ІІІ.

```
provider "aws" {
  region = "us-east-1"
}

variable "instance_type" {
  description = "Instance type for the VM"
  default     = "t2.micro"
}

variable "ssh_key_name" {
  description = "SSH key pair name"
  default     = "ssh_key_name"
}

variable "external_ip_1" {
  description = "External IP of server 1"
  default     = "X.X.X.X"
}

variable "external_ip_2" {
  description = "External IP of server 2"
  default     = "Y.Y.Y.Y"
}
```

Рис. 3.1. Приклад скрипту на *terraform* для розгортання віртуальної машини на *AWS*

Розгортання віртуальної машини *EC2* за допомогою *Terraform* для створення резервної копії компонента платформи вбудованих систем полягає у використанні інфраструктурного коду для автоматизації процесу створення, налаштування та керування інфраструктурними ресурсами на хмарній платформі *Amazon Web Services (AWS)*. Використовуючи *terraform*, як наведено на рисунку 3.1., можна визначити необхідні ресурси, такі як віртуальні машини, обсяги зберігання та мережеві налаштування, що забезпечують високий рівень доступності та стійкості компонентів системи. Віртуальні машини в платформі вбудованих систем використовуються як холодний резерв, забезпечуючи можливість швидкого відновлення критичних систем і компонентів у разі відмови основної інфраструктури, побудованій на основі вбудованих систем, при цьому зберігаючи низькі витрати на ресурси в періоди неактивності. Такий підхід дозволяє здійснити надійне створення резервних копій вбудованих систем, що є критичними для безперебійної роботи та відновлення після можливих відмов. Завдяки автоматизації процесів за допомогою інфраструктурного коду, можна досягти високої масштабованості, скорочення часу на налаштування та зменшення ймовірності помилок, що є важливим аспектом у контексті забезпечення стабільності та безпеки платформи вбудованих систем. При цьому, можна забезпечити як вертикальне так і горизонтальне масштабування на основі даного підходу.

Також, для виконання схожих задач використовується *Ansible*. *Ansible* часто описують як інструмент керування конфігурацією, і зазвичай згадують разом із *Terraform*. Коли мова йде про керування конфігурацією, зазвичай мається на увазі написання опису стану для серверів, а потім використання інструменту для забезпечення того, що сервери дійсно перебувають у такому стані, точніше, правильні пакети встановлено, файли конфігурації мають очікувані значення і мати очікувані дозволи, потрібні служби запуснені тощо. Як і інші інструменти керування конфігурацією, *Ansible* надає доменно-спеціалізовану мову (*DSL*), яку

ви використовуєте для опису стану своїх серверів. Різниця між *Ansible* та *Terraform* полягає в тому, що *Ansible* використовує імперативний підхід в своїй *DSL*, що наведено на рисунку 3.2., коли *Terraform* – декларативний. Окрім цього, *Ansible* використовує за основу синтаксис *YAML*, подібно до *GitHub Actions*.

```
- name: Install Python and dependencies on Raspberry Pi
  hosts: raspberry_pi
  become: yes # Use become to gain root privileges
  tasks:
    - name: Update apt cache
      ansible.builtin.apt:
        update_cache: yes

    - name: Install Python 3 and pip
      ansible.builtin.apt:
        name:
          - python3
          - python3-pip
        state: present

    - name: Install common Python dependencies
      ansible.builtin.pip:
        name:
          - numpy
          - requests
          - pandas
          - flask
        state: present
```

Рис. 3.2. Приклад скрипту на *Ansible* для інсталяції залежностей на вбудованому пристрої

У розробці *IaC* подання невеликих оновлень системи впливає на стан системи. Діяльність подання невеликих оновлень системи також застосовна для розробки програмного забезпечення в цілому. Наприклад, якщо надсилання та впровадження змін в програмному коді великого розміру пов'язане з дефектами або відмовами, тоді ідентифікований зв'язок підкреслить важливість надсилання малих ітерацій змін в системі. Тому інструменти *IaC* використовують покроковий, ідемпотентний підхід.

Фактично, за допомогою даного методу, можна створити інструмент керування станом системи. Інструменти, такі як *Terraform*, є досить зручним інтерфейсом для спілкування з архітектурою системи. Враховуючи його ідемпотентність, необхідність в додатковому збереженні стану відпадає. Це логічно входить до *MLOps* конвеєру як один із етапів розгортання системи.

Важливим є те, що скрипт може бути модифікований і необхідно, щоби новий стан системи не викликав конфлікту під час впровадження нового алгоритму. Вводимо наступне припущення:

$$\forall (r_i, v_i) \in s_i \in S: v_{i+1} = v_i \quad (3.1)$$

Де r_i – ресурс, v_i – статус ресурсу, s_i – стан i -того розгортання, S – вектор станів ресурсів, або стан системи. Для того щоби статус ресурсу (системи) залишався в тому ж стані, потрібно щоби не виникало конфліктів. Вербально, можна сказати, що як результат виконання скрипту стан ресурсу (системи) має бути незмінним, або таким що не викликає конфлікту. Розширимо припущення:

$$\forall (r_i, v_i) \in s_i \in S: (v_{i+1} = v_i) \vee \neg \text{conflict}(s_i, s_{i+1}) \quad (3.2)$$

Де $\text{conflict}()$ функція, яка визначає чи є конфлікт при переході між двома станами. Типовою ситуацією коли може виникнути конфлікт, це звичайна команда видалення, якщо деякий об'єкт видалений, другий раз його неможливо видалити і тому порушується логічний ланцюг. Отже, ідемпотентність може викликати конфлікти, що треба враховувати завчасно.

Ідемпотентність є однією з найважливіших атрибут *IaC*, тому що, з точки зору оптимізації процесу створення конвеєру, це означатиме збереження часу та ресурсів. Як правило, зміни, які будуть впроваджуватись скриптами *Ansible* або *Terraform*, не будуть повністю змінювати систему, окрім, можливо, першого запуску. На кожній новій ітерації розробки моделі, зміна в інфраструктурі не очікується, в тому випадку, якщо базова модель залишиться тою самою, лише з новими параметрами. Тому, створення та розробка ідемпотентних скриптів, які будуть працювати за принципом найменших змін, будуть приносити користь, окрім того, що вони автоматизують розгорнення інфраструктури системи.

В рамках даного дослідження, використання скриптів, або шаблонів, розроблених для розгортання архітектури є важливим процесом автоматизації. Декларативна парадигма та ідемпотентність таких скриптів дозволяє отримувати результат, який можна перевірити та порівняти за зміни умов використання.

Окрім цього, відмовостійкість рішення, який застосовує даний підхід, підвищується за рахунок того, що, скрипт можна використати для онбордингу нового вузлу, відправивши йому задачу та екземпляр контейнеру.

Поставлена задача буде використовувати *IaC* наступним чином. По-перше, в скриптах описуються основні кроки для розгортання інфраструктури проєкту, в тому числі, принцип обрання зображення та його місце призначення. По-друге, *Cloud* частина системи повністю описана в коді скрипту, починаючи від конкретних сервісів, які вона буде імплементувати та надавати, закінчуючи правилами доступу з зовнішнього світу через блокування певного трафіку *TCP*, *UDP* та похідних від них. Нарешті, постійна підтримка працездатності системи буде виконуватись через надання можливості внесення змін. Процес встановлення конкретного сервісу на кожному системі буде регулюватись окремо.

Недоліками такого підходу є проблеми з синхронізацією файлу стану, які виникають в роботі в великій команді, тому необхідно завчасно розробити систему контролю файлу стану. Іншою проблемою є те, що, як і будь який програмний код, під час інтерпретації можуть виникнути помилки. Помилки на рівні неправильного синтаксису не становлять серйозної загрози, але будь-яка мова програмування, особливо *DSL*, може дозволяти проводити дії, які не є бажаними, що призведе до необхідності до повернення до останнього архівованого стану, а це означатиме збільшення *downtime* і зменшенні загальної ефективності системи. Ідемпотентність, з одного боку гарантує можливість виконання скрипту, не враховуючи стан, але може призводити до конфліктів.

3.2. Оцінка відмовостійкості на основі коефіцієнту посередності

Поняття відмовостійкості та його важливість у контексті стабільної роботи підприємства важко переоцінити, особливо під час активної роботи обчислювальних потужностей для обслуговування клієнтів, з деякою заявленою неперервною роботою. Розробка системи, що дозволяє підтримувати власні операційні процеси у стабільному стані, навіть за умови понаднормової кількості

відмов, є запорукою забезпечення підтримки та подальшого масштабування системи у довгостроковій перспективі. До того ж, відмовостійкість системи є особливо важливою запорукою для коректного функціонування задачі *MLOps*, адже стабільність такої системи, при значному обсягу операцій, означатиме коректну роботу алгоритму агрегації даних та забезпечення логування з можливістю подальшого дослідження на виявлення проблем в системі.

Відмовостійкість враховує можливість відмови того чи іншого вузла, при цьому система загалом залишається працездатною, хоча і матиме дещо гірші показники. На відміну від мереж, розроблених на локальному рівні, наприклад домашня, офісна або університетська, високопродуктивні комп'ютерні системи мають досить складну структуру, та є сенс дослідити аналітично цю структуру перед її реалізацією. Для дослідження топології, необхідно обрахувати її показники: діаметр, топологічний трафік, ступінь, тощо.

Для оцінки відмовостійкості досліджуваної топології, введемо поняття коефіцієнт посередництва. Цей коефіцієнт вказує на кількість найкоротших шляхів між деякими двома вузлами в мережі. Теоретично, якщо відмовляє вузол, або декілька вузлів, з високим коефіцієнтом посередництва, алгоритм пошуку нового шляху має перерахувати новий найкоротший шлях, при чому відстань цього шляху може бути вищою. Вузли з найбільшим показником коефіцієнту посередництва мають більш важливу роль у формування зв'язків з іншими вузлами в системі. Формула обрахунку коефіцієнту посередництва для вузла m наступна [1]:

$$b_m = \sum_{i \neq j} \frac{B(i, m, j)}{B(i, j)}, \quad (3.3)$$

де $B(i, j)$ – загальна кількість найкоротших шляхів між вузлами i та j і $B(i, m, j)$ – кількість найкоротших шляхів між i, j що проходить через вузол m . Враховуючи, що найкоротші маршрути можуть бути невідомими і в мережі використовується пошуковий алгоритм, тоді посередництво вузла може виражатись ймовірністю

його знаходження цим алгоритмом. Для цього вводиться поняття коефіцієнта переважання найбільшого посередника, який обчислюється згідно формули:

$$CPD = \frac{1}{n-1} \sum_i (b_{max} - b_i), \quad (3.4)$$

де b_{max} – найбільше значення коефіцієнта переважання найбільшого посередника [1].

Для пошуку шляху обрано алгоритм Флойда-Воршелла модифікований, для запам'ятовування шляху між вершинами. Алгоритм Флойда-Воршелла порівнює всі можливі шляхи в графі між кожною парою вершин. Для деякого графу $G(V,E)$ конструємо матрицю відстаней, ставимо відстань усіх вершин до самих себе як 0. Шукаємо відстань від однієї вершини до іншої, та в окремій матриці відмічаємо вузол задіяний в пошуку найкоротшого шляху. Таким чином, на виході буде дві матриці: матриця відстаней між вершинами та матриця посередництва. В рамках даної роботи, питання використання алгоритму маршрутизації не розглядається, тому можливо модифікувати інші алгоритми, як A^* .

Дана метрика та підхід до її отримання має декілька прикладних значень в рамках даної роботи. У контексті розподілених обчислень і оркестровки посередництво може бути використане для оцінки масштабованості системи. Аналізуючи, як система покращується при додаванні додаткових ресурсів до існуючого вузла (вертикальне масштабування) або при додаванні більшої кількості вузлів (горизонтальне масштабування), можна визначити масштабованість системи. Потім ця інформація може бути використана для оптимізації розгортання ресурсів і забезпечення того, що система може впоратися зі збільшеним попитом. У контексті реконфігурації в реальному часі взаємодію можна використовувати для оцінки впливу змін на систему. Аналізуючи вплив затримки або втрати пакетів на прикладному рівні, можна визначити вплив змін на систему. Потім отримана інформація може бути використана для оптимізації процесу реконфігурації та забезпечення швидкої та ефективної адаптації системи до змін.

3.2.1. Міграція контейнеру на інший вузол після відмови

Поняття відновлення працездатності є доволі широким і потребує окремого дослідження, тому за основу взято, що система залишається працездатною, коли задача може бути виконана на системі. Як було сказано в другому розділі, можна відмову розділити умовно на два типи: апаратна та програмна. В будь-якому випадку, необхідне стороннє втручання, але процеси *MLOps* допомагають оптимізувати такий процес за рахунок автоматизації. В рамках *MLOps* активно зберігаються та аналізуються логи, проводяться health check. В більшості випадків, проблеми виникають за рахунок програмних помилок розробників, які можна досить швидко виявити і виправити шляхом створення різних середовищ для розробки та тестування. Апаратні відмови є дещо більш складними і можливість легко замінити елементи системи є ключовим. На практиці підтверджено, що, якщо елемент системи замінений на новий це значно підвищує відмовостійкість системи в довгостроковій перспективі.

Автоматизована реакція на відмову може бути задана функцією ρ , яка визначає, чи потрібно застосувати стратегію відновлення після виявлення відмови:

$$\rho(\delta, \theta) = \begin{cases} \text{стратегія відновлення, якщо } \delta = 1 \text{ або } \theta(Ci) < 0 \\ \text{жодних дій, якщо } \delta = 0 \text{ та } \theta(Ci) \geq 0 \end{cases} \quad (3.5)$$

Отже розглянемо сценарій, коли відмовив один з вузлів системи. Принципи *MLOps* є дуже доречними для даного типу задачі, оскільки процес відновлення системи при відмові, фактично, є процесом *MLOps*, але на меншому масштабі. По-перше потрібно розуміти чи взагалі доречна реплікація задачі на інший вузол. При відмові вузла на рівні *Edge*, дані, які кешуються на ньому, як правило невеликі за обсягом а задача, фактично, потрібна лише в цілях «кінцевого користувача». Отже, враховуючи область застосування запропонованої технології, можна сказати, що забезпечення нового вбудованого пристрою зводиться до того, що

потрібно його фізично замінити та отримати на ньому зображення контейнеру та задачу.

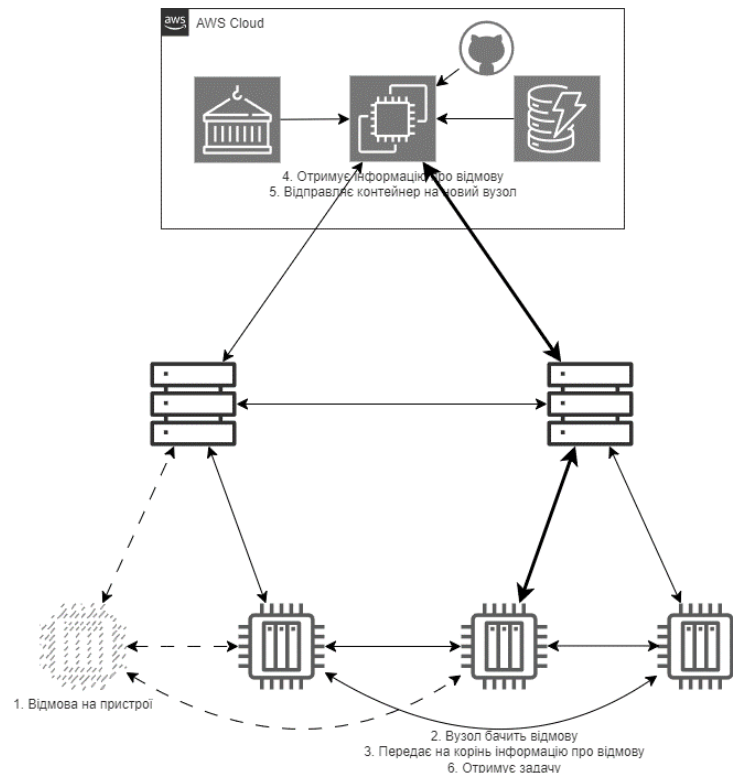


Рис. 3.3. Макет системи з описом алгоритму передачі задачі на інший вузол

Більш складним є ситуація, коли один з вузлів-агрегаторів, тобто тих, що знаходяться на проміжних шарах топологічної організації, відмовляють. В класичній топології дерево, така відмова призводить до того, що одразу декілька вузлів не матимуть доступу до голови дерева. Завдяки додатковим зв'язкам в топології, інший вузол буде отримувати задачі та дані від і до вузлів на нижньому шарі. Для того, щоби поліпшити алгоритм маршрутизації, можна застосовувати комбінацію стандартних алгоритмів, між ярусами можна застосувати алгоритм маршрутизації по дереву, всередині ярусу, можна застосувати алгоритм на зсувах. Таких алгоритм буде дуже простим в реалізації та використовувати значно менше ресурсів, аніж алгоритми, які використовують таблиці та записи. Коли альтернативний маршрут був прокладений, головний вузол буде повідомлений щодо статусу вузлів на нижніх рангах і матиме змогу продовжити взаємодію з ними. Алгоритм дій, представлений на рисунку 3.3., наступний:

1. Вузол відмовляє, будь-які повідомлення, надіслані на цей вузол з нижнього або верхнього ярусу залишаються на вузлах-відправниках.
2. Вузли-відправники рангом нижче прокладають альтернативний маршрут до *Cloud* і відправляють дані.
3. В разі коли повідомлення призначене на вузол, що відмовив, вузли-відправники рангом вище відправляють запит на *Cloud* для отримання адреси вузла-замінника та відправляють дані на цей вузол. Пріоритет надається вузлам, які менш навантажені.
4. В разі коли повідомлення призначене на вузол нижчий за рангом, ніж той що відмовив, прокладається альтернативний маршрут та по ньому відправляється повідомлення.

Даний алгоритм дій є простим і не вимагає додаткових дій з боку обслуговуючого персоналу, як розробників так і системних адміністраторів. Варто зазначити, що постійне логування в рамках *MLOps* означає, що є інформація щодо стану кожного вузла і, як правило, причини, через яку є відмова на вузлі. Подальші дії можуть включати інформування персоналу про відмову, в разі коли немає відповіді вузла на каналному рівні, що означитиме проблеми з апаратним забезпеченням на вузлі.

3.3. Підвищення ефективності розгортання контейнерів для компонентів на основі вбудованих систем

Даний метод дозволяє оцінити, які елементи системи мають найбільший потенціал для впливу на її надійність та точність, та спрямувати увагу на їх подальше вдосконалення або заміну для підвищення загальної відмовостійкості та ефективності системи. В рамках цього методу розглядається вплив окремих елементів та аспектів системи які прямо чи опосередковано впливають на її функціонування. Метод розглянутий у відповідності із технічними вимогами, які виокремлені в контексті *Edge* архітектури, яка потребує ефективне керування обмеженими обчислювальними можливостями, але також може бути використана

на більш потужних пристроях. В рамках даного методу, в якості основного ПЗ для створення та підтримки контейнерів використовується *Docker*.

В якості базового кроку оптимізації, кожен контейнер використовує *bind-mount* для використання системних файлів з хосту. Усі ці файли, крім *.dockerenv*, є критично важливими файлами, які потрібні ядру для належного виконання своєї роботи. Тому, якщо експортувати контейнер в *tar* архіві, можна побачити, що його вміст складається із файлів (та директорій) розміром в 0 байт.

Оптимізація контейнерів в цьому контексті включає в себе ряд заходів, спрямованих на зменшення розміру та використання ресурсів контейнера. Метою оптимізації є прискорення зборки та швидкодії розгортання контейнеризованих додатків на краю мережі, а також зменшення використання ресурсів системи. Існує низка підходів, які можуть оптимізувати контейнери та зображення, зокрема:

Оптимізація конфігурації контейнера шляхом перегляду та внесення зміни, для того щоби зменшити використання ресурсів системи. Це передбачає зменшення кількості процесів, що виконуються в контейнері, оптимізацію розміру зображення контейнера або коригування розподілу пам'яті контейнера.

Одним із найпростіших засобів для зменшення обсягу використаної пам'яті контейнеру. Обмеження доступу контейнера до ресурсів пам'яті забезпечує більш передбачувану продуктивність системи. Крім того, обмеження пам'яті покращують безпеку, запобігаючи атакам на основі ресурсів. Є розділення між жорсткими та програмними обмеженнями пам'яті та надається додаткова інформація про потенційні ризики вичерпання системної пам'яті:

Жорсткі обмеження. Коли контейнер перевищує жорсткий ліміт пам'яті, *Docker* вживає агресивних дій, наприклад припинення роботи контейнера. Жорсткі обмеження зазвичай застосовуються для критичних робочих навантажень, які не можуть дозволити випадкову нестабільність системи.

М'які обмеження. Коли м'яке обмеження досягнуто, *Docker* попереджає користувача, але не вживає негайної дії. Цей тип ліміту дозволяє випадкові стрибки попиту на ресурси, і адміністратори використовують їх для налаштування систем моніторингу та оповіщення.

За умови, коли задача не є надто складною, і не має великої кількості залежностей від ОС, можна використати базове зображення контейнера, оптимізованого для мінімального використання ресурсів. Це може допомогти зменшити обсяг пам'яті контейнера та покращити загальну продуктивність. При цьому, варто враховувати, що таке зображення може не містити повного набору функціоналу, потрібного для забезпечення роботи програми, це особливо важливо в контексті розробки програми, яка використовує можливості штучного інтелекту. В якості альтернативного способу, можна власноруч зробити базове зображення з ОС, використовуючи за основу інше зображення, додавши необхідні програмні пакети та бібліотеки, або можна знайти *open-source* зображення, яке буде мати оптимальний набір пакетів, для успішного виконання задачі.

Контейнеризовані додатки повинні мати належний розмір відповідно до їхніх вимог до ресурсів. Великі контейнери можуть призвести до марної витрати ресурсів, тоді як контейнери малого розміру можуть призвести до зниження продуктивності. Для детального аналізу поточних вимог програмного забезпечення до ресурсів можна використовувати інструменти моніторингу, такі як *Prometheus* і *Grafana*, які можуть допомогти визначити моделі використання ресурсів і прийняти обґрунтовані рішення щодо розмірів контейнерів. Одним із найефективніших способів запобігти завантаженню системних ресурсів контейнерами є встановлення обмежень ресурсів. *Docker* та інші контейнерні платформи надають механізми для визначення обмежень на використання ЦП, пам'яті та введення/виведення.

Використання інструменту оркестровки контейнера, наприклад *Kubernetes*, може бути застосований для того, щоб ефективніше керувати ресурсами

контейнера. Цей і подібні інструменти можуть допомогти оптимізувати використання ресурсів шляхом автоматичного збільшення або зменшення розміру контейнерів залежно від попиту та автоматичного керування розміщенням контейнерів між вузлами для мінімізації використання ресурсів, таким чином вирішуючи питання горизонтального масштабування.

Моніторинг і коригування ресурсів контейнера є важливою частиною *CI/CD* процесу. Для того щоб підтримувати стабільну роботу системи, необхідний збір поточних даних щодо використання ресурсів нею. Постійний моніторинг використання ресурсів контейнера та внесення необхідні коригування для підтримки оптимальної продуктивності за допомогою інструментів, які можуть включати коригування конфігурації контейнера, використання іншого зображення контейнера або збільшення чи зменшення розміру контейнера за потреби, дозволяють підвищити відмовостійкість системи, та допомагають при прийнятті рішень щодо масштабування системи, або зняття зайвих ресурсів. Інструменти, такі як *Grafana*, дозволяють візуалізувати використання системних ресурсів та оповіщати користувачів системи про необхідність підтримки через різні способи відправки повідомлень.

Зменшення розміру *Docker* контейнера за допомогою базових образів може бути досить ефективним способом оптимізації. Запропоновані наступні кроки для реалізації цього:

Використання легких (*lightweight*) базових образів замість тяжких образів, таких як *Ubuntu* або *Debian*, можна використовувати більш легкі альтернативи, наприклад, *Alpine Linux*. Суттєву частину дискового простору займає саме образ ОС і зображення *Alpine Linux* (від якого наслідуються інші зображення, помічені як `—alpine` в назві) відомий своєю мінімалістичною природою та компактними розмірами, що допоможе зменшити загальний розмір контейнера. Окрім цього, існує низка образів, які помічені як *lightweight* образи, які містять базовий набір інструментів для програмного застосунку, який використовує ці бібліотеки. В

контейнері мають використовуватись лише необхідні компоненти та пакети. Як було раніше зазначено, можна використати таке зображення як базове, та розробити інше зображення, таким чином, щоби там був необхідний функціонал для виконання задачі.

Використання офіційних базових образів є більш доцільним, оскільки часто саме офіційні базові зображення, надані розробниками програмного забезпечення, оптимізовані для ефективного використання. Наприклад, *DockerHub* має офіційні образи для різних мов програмування та середовищ виконання, які можуть бути більш ефективними за розміром. Серед них є досить великий набір образів для розробників *python*, з відкритим доступом, для внесення власних модифікацій [77, 78]. Наступна таблиця показує різницю між офіційними базовими образами *python*:

Таблиця 3.1

Порівняння дистрибутивів *Linux* для контейнерів

Дистрибутив	Версія <i>python</i>	Розмір зображення
<i>Alpine Linux</i>	3.12.4	63.33 МБ
<i>Debian 12: bookworm</i>	3.12.4	1.02 ГБ
<i>Debian 12: bookworm slim</i>	3.12.4	155.88 МБ
<i>Debian 11: bullseye</i>	3.12.4	880.11 МБ
<i>Debian 11: bullseye slim</i>	3.12.4	121.01 МБ

Використання мультистадійного збирання *Docker* контейнеру, якщо це можливо, дозволяє використовувати один контейнер для збірки, а інший - для роботи додатку. Наприклад, можна використовувати повний образ з компіляторами та іншими інструментами для збірки, а потім копіювати лише

необхідні файли в кінцевий контейнер. В випадку, розглянутому в рамках даної дисертаційної роботи, контейнер для зборки може бути розгорнутим на найвищому рівні топології, тобто, на рівні *Cloud*. Наведений нижче приклад, показує час збірки нового зображення для *flask API* та *tensorflow AI* разом та окремо.

Таблиця 3.2

Порівняння розміру та часу зборки зображень контейнерів

Дистрибутив	Розмір зображення	Час зборки
<i>Flask API</i>	273.88 МБ	38.3 секунди
<i>Tesnsorflow MobileNetV3</i>	2.6 ГБ	95.5 секунд
<i>Сумарно два контейнери</i>	2.87 ГБ	133.8 секунди (послідовно)
<i>Без декомпозиції</i>	2.63 МБ	142.7 секунд

Окремо можна виділити те, що існує низка способів оптимізації *Dockerfile*. Переміщення найменш змінюваних інструкцій вгору. Завдяки інструменту кешування, перші команди, за допомогою яких будуть побудовані перші шари, будуть виконуватись першими, та враховуючи, що, як правило, вони найбільші за розміром на диску (наприклад, образ ОС), це дозволяє зберегти час в майбутньому. Також є технологія мультистейджинг, яка дозволяє використовувати різні базові образи для різних етапів побудови образу. Уникання копіювання зайвих файлів та папок, а також встановлення зайвих пакунків також дозволяє зберігати ресурс пристрою. При використанні даного методу, необхідно пам'ятати, що якомога більша кількість дій має бути зроблена на початку роботи системи, коли ціна змін найменша, тому кожен оптимізаційний крок вирішує проблеми заздалегідь.

3.3.1. Застосування процесу конвеєрної зборки контейнера для компонентів платформи вбудованих систем

Використання шарів в *Docker* є одним з багатьох процесів оптимізації, які застосовуються для більш швидкого розгортання сервісів. Під час збірки зображення контейнеру, завантажуються шари на пристрій, в залежності від специфікацій в *Dockerfile*, але всі вони, за замовчуванням, завантажуються з серверів *Docker*, що потребує з'єднання до інтернету та щоразу як буде генеруватись нова збірка, вони будуть завантажуватись окремо на кожен вузол. Щоб усунути дану неефективність ми пропонуємо та оцінюємо три оптимізації, які стосуються різних проблем у процесі розгортання. Тому ми можемо об'єднати їх усі разом, що значно покращить продуктивність.

Послідовне завантаження шару зображення. Одночасне завантаження кількох зображень, очевидно, має на меті максимізувати загальну пропускну здатність мережі. Однак це має негативні наслідки, оскільки фази розпакування та вилучення кожного шару зображення мають відбуватися послідовно, щоб зберегти політику копіювання під час запису драйверів сховища *Docker*. Етап розпакування та вилучення може розпочатися лише після завантаження першого шару. Паралельне завантаження кількох шарів зображень призведе до затримки завершення завантаження першого шару, оскільки це завантаження має ділитися мережевими ресурсами з іншими завантаженнями шарів зображень, а отже, також затримає момент, коли перший шар зможе розпочати свою фазу розпакування та вилучення. Тому ми пропонуємо завантажувати шари зображень послідовно, а не одночасно. Цей процес можна описати наступним чином, використовуючи такі позначення:

n — номер шару, який будується (де n приймає значення від 1 до N , де N — загальна кількість шарів).

L_n — стан завантаження шару n , де $L_n = 1$ означає, що шар повністю завантажений, а $L_n = 0$ — що він ще не завантажений.

R_n — стан готовності шару n до видобутку, де $R_n = 1$ означає, що шар готовий до видобутку, а $R_n = 0$ — що шар не готовий до видобутку.

X_n — стан завершення побудови шару n , де $X_n = 1$ означає, що шар побудований і видобутий, а $X_n = 0$ — що побудова ще не завершена.

Перший шар завантажується, що є передумовою для його побудови значення $L_n = 1$. Після завантаження першого шару, він будується і видобувається, разом з тим $C_1 = 1$ якщо $L_1 = 1$, після завершення видобування. Для кожного наступного шару n від 2 до N $L_n = 1$ після завершення $C_{n-1} = 1$. Це означає, що кожен наступний шар починає завантажуватись тільки після того, як попередній шар був побудований та видобутий. Шар n може бути побудований коли він завантажений і готовий до видобутку $C_n = 1$, якщо $L_n = 1$ та $R_n = 1$.

Нехай N — це загальна кількість шарів контейнера. Початково для першого шару $L_1 = 1$, тобто перший шар завантажується відразу. Після того, як він буде завантажений (тобто $L_1 = 1$), цей шар можна побудувати і видобути, тобто $X_1 = 1$ після виконання операції видобутку. При цьому $C_1 = 1$ означає, що перший шар побудовано і видобуто.

Для кожного наступного шару $n \in [2, N]$, значення $L_n = 1$ (шар може почати завантажуватись) стає можливим лише після того, як $C_{n-1} = 1$, тобто після того, як попередній шар побудовано та видобуто. Це забезпечує, що завантаження і побудова шарів відбуваються послідовно, один за одним. Після того, як шар n завантажено (тобто $L_n = 1$), необхідно перевірити, чи готовий він до видобутку. Якщо $R_n = 1$, то шар n може бути побудований і видобутий, тобто $X_n = 1$.

Кінцевий алгоритм включає наступні етапи:

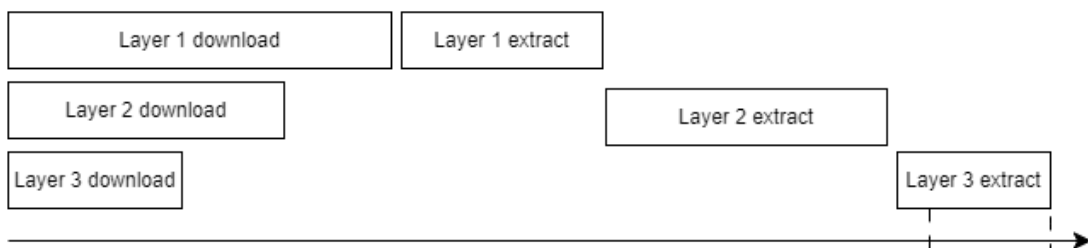
- Відсортувати шари по спаданню часу завантаження.
- Завантажити перший шар, встановивши $L_1 = 1$.

- Побудувати і видобути перший шар, встановивши $X_1 = 1$ і $C_1 = 1=1$.
- Для кожного шару $n \in [2, N]$:
- Перевірити, чи завершено побудову і видобуток попереднього шару, тобто $C_{n-1} = 1$.
- Якщо так, завантажити шар n, встановивши $L_n = 1$.
- Перевірити, чи готовий шар n до видобутку, тобто $R_n = 1$.
- Якщо так, побудувати і видобути шар nnn, встановивши $X_n = 1$ і $C_n = 1$.

Повторювати кроки 3 для всіх шарів від 2 до N.

Алгоритм є жадібним, оскільки на кожному кроці ми прагнемо зробити оптимальний вибір для поточного шару, не зважаючи на майбутні кроки. Кожен наступний шар завантажується і будується лише після того, як вже завершено побудову та видобуток попереднього, більшого шару.

1) Послідовне завантаження



2) Конвеєрне завантаження

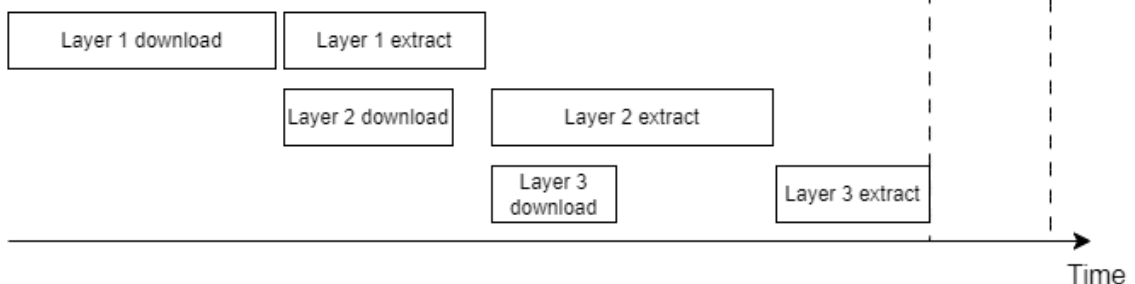


Рис. 3.4. Діаграма Ганта методів паралельної зборки контейнера і конкурентної

Рисунок 2 ілюструє ефект, на діаграмі Ганта, завантаження шарів конкурентно, а не паралельно. В обох випадках створено три потоки для обробки

трьох шарів зображення. Однак у першому варіанті завантаження відбуваються паралельно, тоді як єдина необхідна міжпотоків синхронізація вимагає, щоб декомпресія та витяг шару n могли початися лише після завершення декомпресії та вилучення шару $n-1$. Під час конкурентного завантаження другий рівень починає завантажуватися лише після завершення першого завантаження, що означає, що воно відбувається під час розпакування першого рівня та розпакування на диск. Це дає змогу швидше розпочати видобуток першого рівня, а також збільшує використання ресурсів, оскільки операції завантаження та розпакування й вилучення інтенсивно використовують різні частини апаратного забезпечення машини. Проблема виникає в тому, що пропускна здатність мережі уповільнює швидкість завантаження шарів, тому будь-яка вигода отримана з паралелізації завантаження шарів нівелюється. Конкурентний підхід дозволяє завантажувати шари по мірі доступності. Можливо використовувати предиктивний підхід, з урахуванням потенційного часу завантаження шару, у випадках, коли є блокування ресурсу на завантаження наступного шару, коли минулий вже розпакований. Оскільки операція розпакування здебільшого використовує процесорний ресурс, коли як завантаження *IO*-задача, їх можна виконувати паралельно без значної втрати в часі.

Враховуючи вищесказане, якщо розпакування шару є *CPU-bound* задачею, є сенс використати метод багатопотокової декомпресії шарів. За замовчуванням *Docker* застосовує бібліотеку *gzip* для розпаковування завантажених шарів для зображень, перед збереженням на диску. Однак цей функціонал реалізований однопоточно, що в свою чергу означає, що використання ЦП під час декомпресії ніколи не перевищує 40% із чотирьох доступних ядер на тестовому пристрої *Raspberry Pi*. Тому запропоновано замінити однопотоківу бібліотеку *gunzip.go* на багатопотокову реалізацію, щоб усі доступні ресурси ЦП могли використовуватися для прискорення цієї частини процесу розгортання контейнера. Пропонується використовувати *pgzip*, який є багатопоточною

реалізацією стандартних функцій *gzip*. Його функціонал ідентичний стандартному *gzip*, однак він розподіляє роботу між кількома потоками. У разі застосування до великих файлів розміром принаймні 1 МБ це може значно пришвидшити декомпресію.

Останнім етапом є прунінг. Пропонується ввести наступну формалізацію процесу прунінгу контейнеру, із врахуванням необхідності виконання вимоги щодо функціональної здатності платформи, описаної в (3.6).

$$C_p = \underset{C_i}{\operatorname{argmin}} \{ \operatorname{size}(C_i) \mid \theta(C_i) > 0 \}, \quad (3.6)$$

де C_p – компонента, зображення якого підлягає прунінгу, θ – показник працезданності компоненти, $\operatorname{size}(C_i)$ – розмір зображення контейнеру компоненти. Під час встановлення пакетів, часто використовуються тимчасові файли і є сенс від них позбутись. Для цього, необхідно описати *Dockerfile* таким чином, щоб завантажувався мінімально необхідний набір залежностей та після їх успішного встановлення, тимчасові файли можна видалити. Прикладом завантаження додаткових бібліотек може бути редактор *nano*, який, скоріш за все, використовувати нема сенсу, оскільки можна буде виконувати ручні редагування через редактор на одному з пристроїв по протоколу SSH, або застосувати вбудований редактор *vi*, який значно легший.

Таблиця 3.3

Порівняння розміру зображень контейнеру при використанні прунінгу

Крок оптимізації	Розмір зображення	Час зборки
Базовий розмір зображення	2.6 ГБ	95.5 секунд
Видалення кешованих бібліотек	2.31 ГБ	96.1 секунд
Прунінг	-321.82 МБ (система)	-

В таблиці 3.3. показаний розмір та час зборки зображення контейнеру при використанні різних підходів оптимізації. Також, в рамках оптимізації швидкості зборки, можна застосовувати кешування за рахунок виконання найбільш дорогих

команд вище в *Dockerfile*. Операції, які часто змінюються, мають з'являтися в кінці *Dockerfile*, щоб уникнути перебудови шарів, які не змінилися. Команда *RUN* підтримує спеціалізований кеш, який можна використовувати, коли потрібен більш детальний кеш між зборками. Наприклад, під час встановлення пакетів не потрібно щоразу отримувати всі пакети з Інтернету, достатньо отримувати лише ті, що змінилися. Після зборки зображень, можна провести прунінг за допомогою команди в командному рядку *docker system prune* або *docker buildx prune*.

3.3.2. Архітектурні особливості розгортання контейнерів на Cloud-Edge архітектурі

Значна частина дослідницьких зусиль зосереджена на дослідженні *Edge* архітектури та більшість визнала потенціал одноплатних пристроїв для створення інфраструктури *Edge* або *Fog* обчислень і оцінили їхню придатність для обробки типів робочих навантажень, схожих на хмару. При правильній конфігурації ці пристрої можуть досягти масштабованої продуктивності з мінімальним навантаженням на ресурси системи. Однак дослідження припускає, що зображення контейнера *Docker* вже може кешуватись на локальних вузлах.

Існує низка підходів, що пропонують покращити спосіб взаємодії з реєстрами *Docker* [72]. *CoMICon* — це розподілений реєстр *Docker*, який розподіляє шари зображення між кількома вузлами, щоб підвищити доступність і скоротити час підготовки контейнера [73]. Процес розповсюдження дозволяє отримувати зображення з кількох реєстрів одночасно, що скорочує середній час завантаження шару. Подібні підходи замість цього покладаються на однорангові протоколи [72]. Однак розподілене завантаження ґрунтується на припущенні, що кілька потужних серверів пов'язані між собою високошвидкісною локальною мережею, а отже, головним вузьким місцем продуктивності є міжміська мережа до віддаленого централізованого сховища. У випадку компонентів платформ на основі вбудованих систем сервери будуть географічно розподілені, щоб максимально наблизити їх до кінцевих користувачів, і вони рідко будуть з'єднані

один з одним за допомогою мереж високої пропускної здатності, або мати унікальний доступ до мережі. На противагу цьому ми зосереджуємося на частині розгортання контейнера, пов'язаній із завантаженням і встановленням, і показуємо, що прості модифікації можуть значно підвищити продуктивність цієї операції.

За використанням топологічної організації, в нас є можливість використовувати проміжні вузли та вузли агрегатори для завантаження зображень з них, замість того, щоб їх завантажувати з серверів *Docker*. Завдяки такому підходу, можна зекономити час на завантаження даних з серверів *DockerHub*. Підхід з використанням маршрутизації, використовуючи топологічні особливості дерева та дебруйнівських зв'язків, замість табличних алгоритмів дозволяє додатково зменшити час та використання системних ресурсів під час відправки шарів. Зберігання та кешування шарів та інших об'єктів, пов'язаних з розгортанням контейнеру, на останньому шару, там де будуть використовуватись вбудовані пристрої, є не вигідним, оскільки обмежені ресурси вбудованих пристроїв не дадуть суттєвої переваги, або створять додаткові блокування під час синхронізації шарів на поточному пристрої, тому верхній шар буде застосовуватись для практичного застосування даного методу.

За допомогою простої конфігурації, можна створити власний хостінг *Docker* контейнерів, подібний до *Docker Hub*. Це можна зробити в декілька способів, або можна надсилати вручну кожне зображення на новий сервер, або можна клонувати готовий *Docker registry* та запустити його [79]. Все що потрібно зробити, це згенерувати ключі та сертифікати, та в файлі конфігурації (*config.yaml*) описати конфігурацію серверу. Додатково, можна розробити налаштування для того, щоб інші вузли не могли доступитись власного екземпляру реєстру *Docker*.

На більш пізніх етапах життєвого циклу моделі III, яка була розгорнута, використовуючи даний метод, можна ввести додаткову оптимізацію ресурсів за

допомогою багат шарового методу. При несуттєвих змінах в алгоритмі зборки зображення, достатньо змінити лише один з декількох шарів. Наприклад, при зміні коду програми, який тягне за собою необхідність в копіюванні додаткових даних, ці зміни не будуть нести ніякого впливу на те, як працює базове зображення. Тому, такий підхід є доречним і може призвести до збереження часу. В рамках методу, описаного в минулому пункті, вже було запроваджено використання кешування.

3.4. Обробка залежностей в рамках розгортання платформи

Компонентний підхід до побудови систем є найбільш використаним підходом через зручність завантаження компонентів, та розділення їх в залежності від їх властивостей та функціоналу. Нескладно уявити, що незабаром програмне забезпечення буде являти собою дуже великі колекції компонентів і що повторне використання та спільне використання компонентів стане звичайною практикою. Однак компоненти часто розробляються різними групами, а їхні залежності чітко не визначені. Тому інсталяція (або деінсталяція) компонента часто може бути складним рішенням, оскільки важко знайти всі залежності. Без застосування менеджерів залежностей або завчасного спеціального підбору компонентів, інсталяція може бути неуспішною [15] (встановлений компонент може не працювати), а інсталяція чи деінсталяція можуть бути небезпечними та порушити роботу системи. Щоб активно підтримувати та використовувати еволюцію до систем на основі компонентів, мета полягає в тому, щоб створити підхід із формальними основами, що гарантує успіх і безпеку розгортання.

Основним поняттям є поняття залежності, яке абстрагує зв'язок між компонентом і вимогами до інших програмних пакетів, операційної системи, версії мови програмування або апаратного забезпечення. Залежності використовуються під час встановлення, щоб забезпечити виконання вимог, і під час деінсталяції, щоб гарантувати, що вони все ще виконуються. Як правило, залежності розподіляються на три основні класи:

- обов'язкова залежність (позначена суцільною лінією) є твердою вимогою. Такий тип залежності виникає в разі, коли пакет, який інсталюється, має залежність від іншого пакету, і якщо умова не виконана, установка неможлива. Наприклад, бібліотека *pandas* має пряму залежність від *numpy*, оскільки використовує функції звідти.

- необов'язкова залежність (позначена пунктирною лінією) визначає, що компонент може надавати додаткові послуги. Такі послуги не можуть надаватися (якщо їх вимоги не виконуються) без запобігання інсталяції. Наприклад, пакет *matplotlib* може надавати додаткові можливості рендерінгу графіків, якщо доступні пакети такі як *cairo* або *pdfkit*. В іншому випадку *matplotlib* надає свій базовий вбудований функціонал.

- негативна залежність (виражена запереченням) визначає конфлікт, що забороняє встановлення. Найпростіший приклад це несумісність версії *python* з інсталюваним програмним пакетом. Окрім цього, потрібно враховувати, що деякі програмні пакети можуть перевизначати функції або глобальні атрибути в *runtime*. Також, потрібно знати наперед яка архітектура апаратного забезпечення підтримується пакетом та завчасно знайти або альтернативний пакет від розробника, або використати інший (власний) підхід.

Отже припустимо, що у нас є два пакети, A і B . Між ними є обов'язкова залежність, що в разі, коли ми встановлюємо пакет A , пакет B має також бути інсталюваним та відповідати версії 2.0.3-3.1.2. Це можна описати наступним чином:

$$I_A \Rightarrow \exists v \in [2.0.3, 3.1.2] : I_{B_v}, \quad (3.7)$$

де I_A – можливість інсталювання пакету A , v – необхідна версія, зумовлена залежністю пакета A до B , B_v – множина існуючих (доступних до завантаження) версій пакету, I_{B_v} – показує можливість інсталювання версії v пакету B . Узагальнимо умову можливості інсталювання пакету для n залежностей:

$$\forall d \in [d_1, d_2, \dots, d_n], \exists v \in [v_1, v_2, \dots, v_n], \quad (3.8)$$

де d – це певна залежність, v – версія пакету, яка задовольняє цю залежність.

З наведеної формули вище видно, що проблема залежності можна звести до математичного виду та змодельовати. Як правило, залежності вписані одразу в налаштуваннях пакету, для *python*, це як правило *requirements.txt* або *setup.py*, які швидкодоступні і не потребують попереднього встановлення для перевірки наявності залежностей. Складність обчислення залежностей може серйозно варіюватись, враховуючи їх кількість. В найгіршому випадку, якщо кожен програмний пакет та застосунок має обов'язкову або негативну залежність від іншого, складність обрахунку може бути факторіалом значення кількості залежностей, не враховуючи можливості (пере-)ранжування пакетів та кількість версій, які будуть підходити. Фактично задача зводиться до задачі здійсненності булевих формул (*SAT*), що є *NP*-повною задачею, згідно теоремі Кука-Левіна.

Шляхами оптимізації цього процесу є:

Групування залежностей. Якщо деякі залежності є необов'язковими, або обов'язковими в рамках тестування, їх можна винести окремо і під час зборки нового зображення, вони будуть інстальовані тільки тоді коли необхідно. Такий підхід називається «розділай і володаруй» [76]. Ідея заключається в тому, щоб простір пошуку був розбитий на декілька, але часткові рішення можуть значно відрізнятись в складності і можуть виникати проблеми в балансуванні навантаження, при паралельній обробці з використанням даного методу. Припустимо, що є деякий програмний пакет *A*, при виконанні процесу інсталяції за допомогою менеджерів пакетів, без додаткової інформації, пакет *A* буде встановлений або останньої версії, або стабільної версії. Групування залежностей зменшує простір пошуку версії інших залежностей, які є обов'язковими. Тобто можливість інсталяції зводиться до наступного виразу, де v_1 та v_2 , такі, що $v_2 > v_1$, тобто версія новіша:

$$I_A \Rightarrow \exists v \in B_v: v_1 \leq v \leq v_2 \quad (3.9)$$

де множина версій $B_v = \{v_1, v_2, \dots, v_n\}$, v_i – певна весія пакету, I_a – можливість інсталювання пакету.

Кешування конфліктів. Якщо деякі залежності не можуть бути виконані одна через одну, тоді це може бути закешовано, щоби в другий раз не проходити цей алгоритм. Найбільш «важкі» залежності мають бути першими. Під «важкими» маються на увазі ті залежності, які задовільняють в тій чи іншій мірі наступні умови: функціонально, на скільки сильно вони інтегровані в застосунок, розмір пакету, кількість залежностей. Такий підхід може дозволити розбити на групи залежностей, відносно розміру простору пошуку та відносно статусу залежності. Нехай для кожної залежності v визначено її розмір пам'яті $m(d)$, який може залежати від кількості використовуваних ресурсів, що зберігаються цією залежністю. Якщо залежність v вже була оброблена, то ми можемо скористатися кешованим результатом, щоб уникнути повторних обчислень. Для цього введемо функцію кешу:

$$CacheHit(v) = \begin{cases} 1, \text{ якщо } v \in CD \\ 0, \text{ якщо } v \notin CD \end{cases} \quad (3.10)$$

де CD – це множина залежностей, які зберігаються в кеші. Для кожної залежності d_i , де $i \in \{1, 2, \dots, n\}$, перевіряємо, якщо $CacheHit(v_i) = 1$, то використовуємо кеш. Якщо $CacheHit(v_i) = 0$, то обробляємо цю залежність і додаємо її результат до кешу. Щоб мінімізувати використання пам'яті та зменшити кількість запитів на завантаження та інсталяцію залежностей ми будемо виконувати алгоритм в порядку спадання значення $m(v)$, тобто для $v_i \in \{v_1, v_2, \dots, v_n\}$ так, щоб $m(v_1) \geq m(v_2) \geq \dots \geq m(v_n)$.

Поняття залежності та його важливість на практиці, у контексті обробки декількох залежностей, важко переоцінити, особливо під час активної роботи обчислювальних потужностей для обслуговування, з деякою заявленою неперервною роботою. Розробка ефективної системи потребує можливості

завчасного передбачення можливості виконання умов менеджера пакетів. Для тестування та використання прикладного застосунку використовується прийом ізоляції залежностей через створення віртуального середовища. Найпоширенішим і найпростішим методом ізоляції є *virtualenv* (*venv*). Кожен *venv* має власну копію інтерпретатора *Python* і залежностей у каталозі *site-packages*. Щоб використовувати *venv*, його потрібно спочатку створити за допомогою команди *virtualenv*, а потім активувати. Версії в файлі *requirements.txt* можна зберігати за допомогою простих операторів як `==`, що встановлює обов'язкову залежність з певною версією продукту, оператори нерівності (`>`, `>=`, `<`, `<=`), або використання зірки в версії продукту, що вказує на можливість використання будь-якої версії, в середині цього пакету.

3.5. Розробка схеми розгортання платформи вбудованих систем

В рамках даної роботи, було розкрито питання використання багатоваріантної архітектури в різних аспектах та різних площинах дослідження. Представлення шарів змінювалось в залежності від конкретної досліджуваної сфери, але залишалась необхідність в чіткому семантичному або функціональному розділенні зон. Як відомо, компонентний підхід підвищує гнучкість системи, що, в свою чергу, підвищує здатність системи до масштабування.

За допомогою використання шарів, можна досягнути вищої відмовостійкості, не створюючи додаткових залежностей між шарами, які не поєднані між собою, та зробити downtime якомога меншим, оскільки виокремлення однієї нефункціонуючої частини та її налагодження є простішим процесом і зберігає час.

Компонентний підхід до розробки моделі до даної роботи є важливим, і, тому, пропонується використовувати мікросервісну архітектуру для розробки системи. Це є логічним кроком, оскільки, система, яка розглядається є розподіленою *MPP* системою. На рисунку 3.5. наведені технології, та відповідні їм методи та підходи використані в дослідженні. З точки зору логіки виконання певної задачі, також є сенс розробити декілька сервісів, які будуть розгортатись

за необхідності на певному вузлі системи, таким чином розділяючи обов'язки кожного вузла. Як результат даного підходу, зображення для контейнерів будуть меншими, та створюватимуться швидше. Окрім цього, семантична помилка в описі коду програми на певному вузлі не буде впливати на загальну роботу системи, що підвищить загальну відмовостійкість системи.

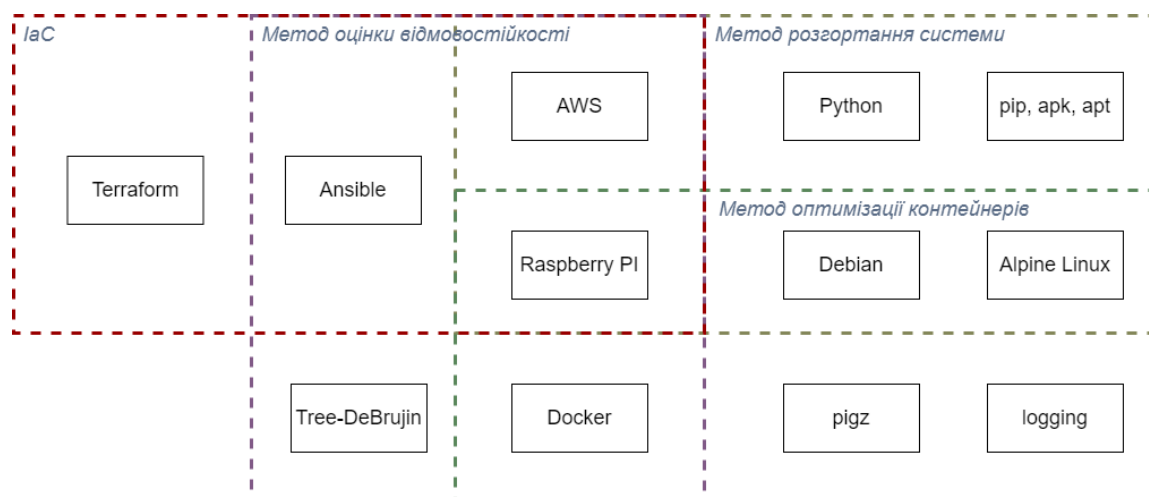


Рис. 3.5. Візуальна класифікація застосування технологій для застосованих методів та підходів

Отже, узагальнюючи раніше запропоновані методи та підходи, запропоновано наступний підхід до оптимізації процесу створення конвеєру для розгортання компонент платформи вбудованих систем із задачами ІІІ з використанням платформ вбудованих систем. Кожен з запропонованих методів буде використаний як шар оптимізації. Це дозволяє використати властивості компонентної розробки в рамках методології, запропонованій в рамках дисертаційного дослідження. Питання розробки ПЗ для моделювання даних процесів буде окремо розглянуто в наступному розділі.

Окремо слід зазначити функціональні вимоги для системи для успішного виконання *SLA* в практичній реалізації цього методу:

- Доступність – дана вимога може бути інтерпретована як відмовостійкість. Потрібно врахувати особливості моделі і гарантувати певний uptime. Для цього є доцільним провести окреме моделювання системи в наступному розділі.

- Масштабованість – вимога, яка пояснюється необхідністю певного сервісу ІІІ компоненти залучати додаткові ресурси для виконання послуг. В рамках вбудованих систем це можна порівняти з онбордингом нових пристроїв. Використання *IaC* значно спрощує цей процес, оскільки будь-які зміни в апаратному забезпеченні системи відображаються в коді та можуть бути відносно легко модифіковані.

- *MTTR* (*mean time to recovery* – середній час для відновлення) – одна з базових метрик, яка пояснює який час потрібен для відновлення працездатності системи. Потрібно врахувати, що система була розроблена на основі топології, яка передбачує певну кількість відмов, що не впливатимуть (в значній мірі) на працезданість системи.

- Моніторинг – для вчасного виявлення проблем та впровадження мір по налагодуванню системи, необхідно мати доступ до логів та поточного стану. Існує програмне забезпечення, яке дозволяє це автоматично виконувати на задньому фоні в необхідній мірі так, щоб це було непомітно для кінцевого користувача.

Висновки до розділу 3

В даному розділі досліджені методи розгортання системи, враховуючи її архітектурні особливості та розглянуто шляхи її доопрацювання. Серед зазначених викликів є використання гібридної *Cloud-Edge* архітектури. Розгортання платформи вбудованих систем, з урахуванням особливостей задач ШІ, є складним процесом, який потребує зусиль для того, щоб ефективно впроваджувати та застосовувати ШІ моделі. Оскільки ці моделі не існують безвідносно, необхідно створити умови, за яких їх розгортання не лише можливе, але й оптимально використовує ресурси системи.

Відповідно до поставленої задачі, було запропоновано підхід з використанням особливостей міжвузлових зв'язків системи. Цей підхід має на меті застосувати наявну архітектуру мережі компонентів платформи для підсилення відмовостійкості системи, та покращення її показників, за рахунок використання вузлів-агрегаторів. Досягнуто збереження часу при завантаженні програмних пакетів (зображень, залежностей) використовуючи вузли-агрегатори. В подальшому, такий підхід може бути застосований для збільшення енергоефективності пристроїв. Відмовостійкість була підсилена можливістю передачі задачі на інший вузол в мережі. Для ефективного виконання алгоритму, використані топологічні особливості мережі.

Розглянуто низку підходів щодо оптимізації контейнерів. Контейнери з моделями ШІ використовують великі обсяги пам'яті для збереження самої навчаної моделі, фреймворку, на якому модель була тренувана, та залежностей які необхідні для запуску фреймворку. Розроблено метод подібний прунінгу для використання в контейнерах з метою прискорення часу зборки та зменшенню розміру зображень. В основу цього методу входить використання найменшої зборки ОС для запуску на девайсі та використання послідовного мультистадійного методу зборки зображення. Додатково, розглянуто оптимізаційні кроки щодо використання залежностей в рамках комплексного

методу, враховуючи необхідність в обробці великої кількості залежностей, та встановленням необов'язкових залежностей.

На основі отриманих підходів, запропоновано метод, який формує онтологічні зв'язки, поєднуючи різні компоненти платформи, її сутності та процеси. Для дослідження комплексного методу, запропоновано підхід з використанням підходу *IaC*. Особливістю даної методології є можливість автоматизувати запровадження розглянутих методів та алгоритмів автоматично. Основною властивістю доменно-спеціалізованих мов *IaC* є ідемпотентність, яка дає можливість робити надбудови до системи, або рекофігурацію системи, без необхідності отримання її загального стану, тобто конфігурації. На практиці, це означатиме можливість швидко робити зміни в системі, що збільшуватиме відмовостійкість системи та зменшуватиме час очікування компонент.

РОЗДІЛ 4

МОДЕЛЮВАННЯ МЕТОДУ РОЗГОРТАННЯ КОМПОНЕНТ ПЛАТФОРМИ ВБУДОВАНИХ СИСТЕМ ТА АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ

4.1. Застосування логування в рамках CI/CD процесу

Вставлення операторів логування в програмне забезпечення є частиною належної практики програмування. Інструкції логування дозволяють нам збирати інформацію про поведінку програми під час її виконання. Ведення журналів, усталена практика традиційної розробки програмного забезпечення [80], відіграє так само вирішальну роль у контексті додатків на основі *ML*. Це дозволяє розробникам і дослідникам даних отримувати інформацію про час виконання, відстежувати поведінку моделі, контролювати прогрес навчання, а також виявляти та вирішувати проблеми. Ця інформація часто використовується для виконання різноманітних завдань під час обслуговування та роботи програмного забезпечення, таких як діагностика збоїв, звітування про помилки, виявлення аномалій і автоматичне генерування коду журналу з використанням даних журналу.

У контексті *DevOps* на *IT*-операторів покладаються нові обов'язки. Зокрема, процес тестування програмного забезпечення повністю змінено в *DevOps* з багатьма тестовими діями, що відбуваються в полі [12]. На жаль, є дуже мало робіт про забезпечення систематичного зворотного зв'язку щодо якості для сторони розробки програмного забезпечення. Крім того, багато з існуючих аналізів журналів зосереджені лише на журналах виконання. Але журнали — це лише один тип телеметричних даних, які існують у полі. Інші типи телеметричних даних, як-от трасування або дані *APM*, можуть надати додаткову інформацію про поведінку системи під час виконання. Проте мало досліджень зроблено для збагачення аналізу журналів шляхом співвіднесення їхньої інформації разом.

Дослідження показує, що логування в програмах на основі ШІ є менш поширеним, ніж у традиційних програмах, із меншою загальною щільністю журналів. Крім того, розподіл рівнів логування в програмах з використанням ШІ відрізняється від розподілу в традиційних програмах. В традиційних програмах використовується рівні логування в залежності від їх критичності, відповідно до працездатності системи. Методи логування в ШІ додатках можуть мати унікальні рівні та заслуговують на подальше дослідження. Конвеєр для платформ з ШІ задачами зазвичай складається з кількох етапів, які розглянуті в першому розділі. Аналіз сучасних рішень показав, що більшість операторів логування відбувалися під час фази навчання моделі. Ці заяви зазвичай містили інформацію про гіперпараметри та продуктивність навчених моделей. Однак додатки на основі *AI* представляють унікальні проблеми та вимоги, що вимагає розробки спеціалізованих рішень для логування. Традиційні бібліотеки журналювання (наприклад: *logging* (Python), *log4j* (Java), *zerolog* (GO), *winston* (Node.js)), які використовуються в розробці традиційного програмного забезпечення, часто не відповідають конкретним вимогам практиків *AI*. Унікальні проблеми, які виникають у машинному навчанні, такі як ефективне керування великими наборами даних, обробка складних архітектур моделей та інтерпретація складних процесів навчання та логічного висновку, спонукали до розробки бібліотек логування, специфічних для ШІ.

Ці спеціалізовані бібліотеки адаптовані до конкретних потреб практиків ШІ. Вони пропонують спрощені підходи до логування показників продуктивності, моделювання гіперпараметрів і надають деяку візуалізацію інформаційної панелі. Зосереджуючись на потребах розробників ШІ, ці бібліотеки допомагають оптимізувати процес журналювання, роблячи його більш доступним і зручним для користувача [16]. Яскраві приклади бібліотек журналювання для ШІ включають *MLflow*, *TensorBoard*, *Neptune* і *Weights & Biases (W&B)*. Ці бібліотеки набули широкого поширення серед спільноти машинного навчання завдяки своїй

ефективності у вирішенні унікальних проблем, пов'язаних із програмами на основі машинного навчання.

4.2. Обґрунтування вибору програмного забезпечення для компонент вбудованих систем

4.2.1. Архітектура тестової системи

Основою системи є топологія, вона диктує, фактично, яким чином взаємодіють вузли системи. Топологія системи побудована наступним чином. Застосовуємо, в якості тестової системи, гібридну топологію *Tree-Debrujin* 3 порядку. Топологія двійкового дерева є однією з фундаментальних структур у теорії алгоритмів, графів, статистиці та інших комп'ютерних науках. Її широке міждисциплінарне застосування зумовлене її простою структурою. Для бінарного дерева, кожен вузол може мати не більше двох дочірніх вузлів, відомих як лівий і правий, які в свою чергу можуть також бути внутрішніми вузлами або листями. Окрім цього, двійкове дерево має ієрархічну структуру, що дозволяє створювати. В рамках цього дослідження буде йти мова про збалансоване двійкове дерево, тобто, в якому у кожного вузла деякого ярусу однакова кількість дочірніх вузлів.

Отже, на кожному наступному етапі масштабування до дерева додається $2^r - 1$ вершина, що також відповідає кількості вершин у графі ДеБруйна на r розрядах. Таким чином, можна використовувати ДеБруйнівські зсуви на кожному рівні дерева, для поліпшення базових характеристик дерева. Рекомендується використовувати подвійну нумерацію, щоб органічно поєднати пошук по дереву і маршрутизацію за ДеБруйном. Рівень визначається за допомогою номера вузла в дереві. Припустимо, що цікавить на якому рівні знаходиться вузол p , для цього знаходимо значення $\lfloor \log_2(p) \rfloor = m$, яке відповідає рівню, на якому знаходиться вершина. Знаючи це значення, можна визначити всі зв'язки між вершинами на рівні вже з нумерацією ДеБруйна. Таким чином, вузол

має подвійний номер, що записується через крапку. Наприклад, 3.7, де 3 - це рівень дерева, а 7 - номер у топології ДеБруйна. Схема архітектури показана на рисунку 4.1.

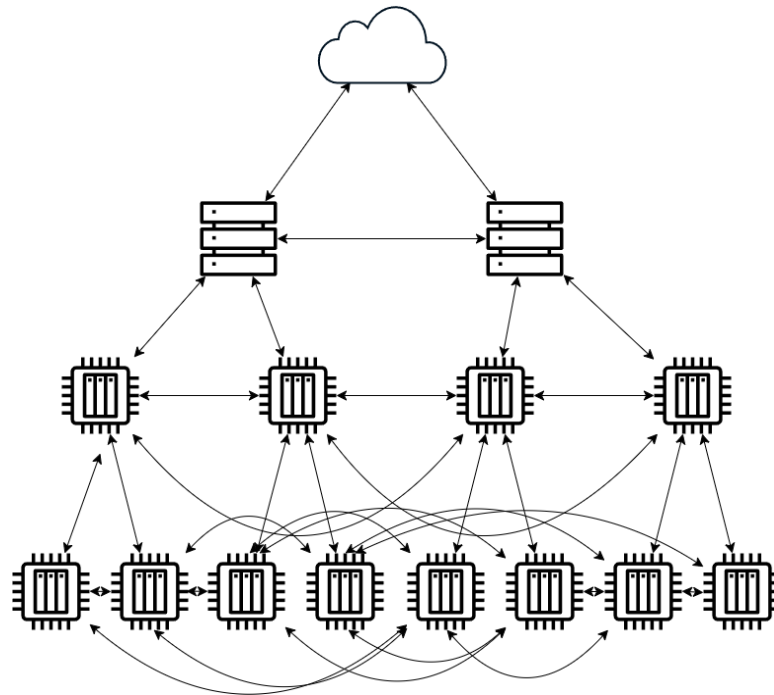


Рис. 4.1. Архітектура системи на основі деревовидної структури з додатковими дебруйнівськими зв'язками

Додаткові зв'язки будуються за наступною формою:

$$D(n, k) = k \cdot D(n - 1, k) + (-1)^k \cdot (k - 1) \cdot D(n - 1, k - 1), \quad (4.1)$$

де $D(n, k)$ - число Дебруйна, n - кількість елементів, k - кількість циклів, $(-1)^k$ — знак, що змінюється залежно від парності k , враховує чергування додавання і віднімання у рекурсії.

Наступним етапом є формування системи. Виходячи з ієрархічної структури системи, пропонується наступний спосіб призначення вузлів. На нижньому рівні використовуються вбудовані пристрої *Raspberry PI 3B+*. Використання *Raspberry PI* в якості вбудованого пристрою є доцільним, наприклад, в літературі є демонстрація, що відносно обмежені пристрої, в плані ресурсів, такі як *Raspberry PI*, можна успішно використовувати для створення хмарних шлюзів [72]. Незважаючи на те, що задачі ІІІ є значно більш ресурсоемними, при

оптимальному використанні ресурсів, за допомогою моделей, навчених для таких пристроїв, подібні задачі можуть бути відносно легко виконані. На середньому рівні використовуються більш потужні сервери-агрегатори, задачею яких є балансування навантаження між вузлами нижче та передача їх стану на вищий рівень. На найвищому рівні використовується хмарні компоненти і різні сервіси, які входять в обслуговування платформи. Рішення, які включають в себе хмарні компоненти будуть розглянуті в пункті 4.1.2.

Зокрема, робота демонструє доцільність використання обмежених вузлів в якості туманних шлюзів: масштабованості та низьких накладних витрат можна досягти за допомогою належного налаштування конфігурації та оптимізації реалізації, також при використанні *Docker* поверх *Raspberry PI* шлюзів. Повідомлені результати продуктивності показують, що контейнеризацію на основі *Docker* можна використовувати в кількох доменах додатків Інтернету речей, де є прийнятною деяка затримка у випадках необхідності повторного розгортання нових функцій, що співпадає з підходом *MLOps*, зі значними перевагами з точки зору гнучкості та легкого розгортання, що сприяє більш швидкому розповсюдженню *Cloud-Edge* рішень також у існуючих середовищах розгортання.

На найвищому рівні топології стоїть *Cloud*. Для розгортання *Cloud* інфраструктури використовується підхід на основі використання *IaC*. *Terraform* підтримує весь необхідний в рамках даного дослідження функціонал для створення віртуальної машини, бази даних та інших допоміжних сервісів. Декларативний підхід в написанні коду дозволяє швидко описувати необхідні ресурси та зробити їх налаштування, тим самим автоматизувавши процес розгортання інфраструктури на *Cloud*. Окремо варто уваги те, що *Cloud* провайдери дозволяють робити резервне копіювання зі збереженням стану системи, що додатково підсилює відмовостійкість системи.

Застосування гібридної топологічної організації дерево з дебруйнівськими зв'язками є доцільним, завдяки досить низьким топологічним характеристикам, що показує що масштабування системи не призведе до неприйнятно високих збитків. Завдяки своїй ієрархічній природі, дана топологія легко може дати основу для створення гібридної *Edge-Cloud* архітектури, якщо ресурси *Cloud* будуть використовуватись на корні дерева і окреструвати задачі на нижніх ярусах. Розглянуто набір факторів, які розглядаються при виборі методу створення гібридної топології та виборі найкращої, і, можна казати, що серед існуючих запропонованих рішень не існує альтернативного варіанта, який би забезпечував при даних характеристиках, необхідні критерії. Окремо варто зазначити, що додаткові зв'язки допомагають підсилити відмовостійкість системи, що є викликом для *Edge* пристроїв та будь-якої мережі.

4.2.2. Технологічний стек системи

Вибір технологічного стеку для тестової системи є важливим етапом розробки програмної системи, в тому числі ІІІ платформи, оскільки в залежності від програмного забезпечення та бібліотек, можуть накладатись додаткові обмеження щодо розробки тестової програми. Як правило, під програмним стеком розуміється комбінація програмних пакетів прикладного та системного рівнів, які будуть використовуватись в рамках програмної системи. В щоденному застосуванні та інженерних виданнях існує велика кількість стеків, таких як *LAMP*, *MERN* та інші, але вони відображають лише прикладний рівень програми та не зможуть показати всі рівні запропонованого пакету.

По-перше, для функціонального виконання більшості вимог до компонент платформи вбудованих систем із моделлю ІІІ, зокрема *API*, завдяки якому можна формалізувати комунікацію з іншими пристроями, управління та логування моделі ІІІ, виконується через мову *Python*. *Python* набув популярності завдяки своєму простому синтаксису, інтерпретованості, що дозволило легко впровадити

компонентний підхід та розробити широку низку бібліотек для виконання різних функцій, в тому числі, розробки III. Завдяки доступу до бібліотек, таких як *numpy* та *scikit-learn*, необхідність в розробці складних структур даних та векторизації власноруч розробником відпадає, при цьому, можливість внесення правок до системного коду залишається відкритим в пакеті *tensorflow*.

Завдяки своїй простоті, з використанням *YAML* файлів інструкцій, *Ansible* дозволяє описати бажаний стан системи, що дозволяє ефективно керувати одразу декількома пристроями на *Edge*. *Ansible* дозволяє зберегти час додатково за рахунок того, що його використання не потребує його встановлення саме на *Edge* пристроях. Достатньо мати встановлений *python* та відкритий *SSH*. Ці кроки описуються для розгортання системи на *Edge* з використанням *Ansible*, спочатку потрібно підготувати інвентаризаційний файл, що описує всі цільові системи та доступ до них. Файл інвентаризації має містити *IP*-адресу системи, де ми розгортаємо, ім'я користувача та пароль системи (можна створити нового користувача в системі керування комп'ютером, який може надати адміністратору доступ до користувача, якого ми створено), також вкажіть номер порту, якщо потрібно, інакше буде використано порт за замовчуванням і підключення сертифіката *SSL*. Після цього можна використовувати модулі (playbook) *Ansible* для встановлення і налаштування *Docker* або розширити можливості на *Edge* пристрої відповідно до заданого конфігураційного файлу, наприклад, оновити завантажений контейнер на новий. *Ansible* також забезпечує можливість моніторингу та управління цими контейнерами, що робить його потужним інструментом для автоматизації розгортання на *Edge*-системах, де досягається доступність і стабільність завдяки наперед описаній конфігурації.

Процес починається зі створення виділеної віртуальної машини *EC2*, який буде служити сервером *CI/CD*. Цей екземпляр має бути налаштований із достатніми обчислювальними ресурсами для обробки процесів збірки та розгортання. Бажано вибрати образ машини *Amazon (AMI)*, який відповідає

бажаній операційній системі команди розробників і містить попередньо встановлені інструменти, які зазвичай використовуються в конвеєрах *CI/CD*.

Коли екземпляр *EC2* запрацює, наступним кроком буде встановлення та налаштування інструменту *CI/CD*, наприклад *Jenkins*, *GitLab CI* або *AWS CodePipeline*. Ці інструменти полегшують автоматизацію процесів збірки, тестування та розгортання. Після встановлення вибраний інструмент має бути інтегрований із системою контролю версій (наприклад, *Git*), де міститься код програми.

4.2.3. Вимоги до тестового середовища

Для проведення експериментів, необхідно розробити тестове середовище, яке здатне наблизити умови використання розробленого методу до реальних. Під середовищем мається на увазі набір програмного забезпечення, серед них, бібліотек, мов програмування, програмних засобів, засобів управління апаратними ресурсами та можливостями. Важливою вимогою до цього етапу є гнучкість середовища, тобто можливість використання схожих версій програмного продукту, та інтегрованість, тобто кожна залежність має щонайменше не створювати перешкод для використання інших.

В основі програмного стеку лежить *Python*, що зумовлює використання програмних пакетів написаних для *Python*. *Flask* та *tensorflow* — це відомі інструменти, що знайшли широке застосування у сучасних дослідженнях і розробках в галузі машинного навчання та штучного інтелекту. *Flask*, як легковаговий веб-фреймворк для *Python*, дозволяє створювати веб-додатки та *API*, що забезпечують зручний інтерфейс для взаємодії з моделями машинного навчання. *Tensorflow*, у свою чергу, є однією з провідних бібліотек для розробки та навчання нейронних мереж і інших моделей глибокого навчання. Можливість застосування в одному полі імен обидвох пакетів потребує додаткового дослідження з урахуванням принципу низької зв'язаності та високої пов'язаності.

За допомогою *flask* можна швидко та масштабовано реалізувати веб-сервіс, який використовує *TensorFlow* для використання або навчання моделей. Це приносить особливу користь для розгортання та взаємодії з моделями, які вимагають взаємодії через мережу, наприклад, у випадках розпізнавання об'єктів, обробки природних мов або передбачення великого обсягу даних, або, в рамках даної роботи, для отримання даних з *Edge*-пристроїв. *Flask* забезпечує простий інтерфейс для інтеграції з *Tensorflow*, що дозволяє як дослідникам так і кінцевим користувачам швидко створювати та розгортати рішення у сфері штучного інтелекту, спрощуючи процес розробки і забезпечуючи масштабованість системи для обробки складних завдань у реальному часі. Варто враховувати, що *API* сервіс може бути замінений на використання більш легких *pub-sub* сервісів як, наприклад, використання протоколу *MQTT* та створення топіків, на які можна публікувати запити, створюючи черги, тим самим вирішивши питання синхронізації.

Пакет *Poetry* для мови програмування *Python* використовується для керування залежностями і віртуальним середовищем проекту. Цей інструмент забезпечує автоматизацію установки, оновлення та видалення пакетів, враховуючи їх взаємозалежності. Використання *poetry* спрощує процес розробки та підтримки проектів, спрощує вирішення проблем, пов'язаних з управлінням залежностями. Крім того, *Poetry* забезпечує ізольоване середовище для кожного проекту, що сприяє уникненню конфліктів версій пакетів і дозволяє легко репродукцію середовища на різних платформах.

За допомогою механізму *lock* файлу *Poetry* фіксує точні версії залежностей, тим самим пом'якшуючи проблеми, пов'язані з конфліктами версій, і забезпечуючи відтворюваність у різних середовищах. Цей підхід не тільки підвищує надійність проектів *Python*, але й спрощує керування складними графами залежностей. Аналізуючи обмеження, зазначені у файлі *pyproject.toml* проекту, *Poetry* систематично визначає оптимальну комбінацію пакетів,

дотримуючись визначених діапазонів версій або точних версій, визначених розробником. Цей ретельний процес підкреслює ефективність *Poetry* у сприянні безперебійному та ефективному управлінню пакетами *Python*, особливо добре задовольняючи деталізовані вимоги наукового обчислення та науково-орієнтованих програмних проектів.

4.3. Проведення експериментів розгортання та використання компоненти платформи вбудованих систем

В даному пункті буде проведено експерименти, згідно вимогам до ПЗ та архітектури системи, описаними в пункті 4.2. Пропонується зробити експерименти з цілю встановлення швидкодії системи та часу реакції на зміни в неї. Перший експеримент направлений на перевірку швидкості розгортання системи в порівнянні швидкості зборки контейнеру для використання на нижньому ярусі. В якості базового значення, пропонується розглядати *Raspberry PI 3B+*, на якому має працювати модель ШНМ. Завдяки своїй декларативній природі *Ansible* дозволяє описати бажаний стан системи, що дозволяє автоматизувати процес відправки та запуску цього контейнеру на вбудованому пристрої без необхідності в його побудові на кінцевому пристрої. Дослідження часу зборки контейнеру показано в таблиці 4.1.

Таблиця 4.1

Час зборки контейнеру без внесення змін

Тип пристрою/системи	Час зборки
<i>Raspberry PI 3B+</i>	732.3 секунди
<i>Ryzen 9 3900X</i>	134.5 секунд
<i>EC2 t4g.nano</i>	524.4 секунди

В таблиці 4.2. відображений час побудови контейнеру з використанням методу конкурентної зборки. В рамках цієї зборки, використовуються ті самі стартові параметри, що відображені для таблиці 4.1.

Таблиця 4.2

Час зборки контейнеру із конкурентною зборкою

Тип пристрою/системи	Параметри	Час зборки
<i>Raspberry PI 3B+</i>	Базова зборка	732.3 секунди
<i>Ryzen 9 3900X</i>	Базова зборка	134.5 секунд
<i>EC2 t4g.nano</i>	Базова зборка	524.4 секунди
<i>Raspberry PI 3B+</i>	Конкурентна зборка	624.6 секунди
<i>Ryzen 9 3900X</i>	Конкурентна зборка	113.2 секунди
<i>EC2 t4g.nano</i>	Конкурентна зборка	442.8 секунди

Наступний тест враховує параметри, використані для тестування з методом конкурентної зборки, та використовується метод прунінгу. В рамках прунінгу, додатково застосовується *poetry* для керування залежностями, що допомагає зекономити час видалення невикористаних залежностей в самому ПЗ. Таким чином такий підхід, потенційно, має вплинути на час зборки контейнеру і системи загалом.

Таблиця 4.3

Час зборки контейнеру із прунінгом

Тип пристрою/системи	Параметри	Час зборки
<i>Raspberry PI 3B+</i>	Базова зборка	745.7 секунди
<i>Ryzen 9 3900X</i>	Базова зборка	138.9 секунд
<i>EC2 t4g.nano</i>	Базова зборка	541.5 секунди
<i>Raspberry PI 3B+</i>	Конкурентна зборка	627.6 секунди
<i>Ryzen 9 3900X</i>	Конкурентна зборка	117.5 секунди
<i>EC2 t4g.nano</i>	Конкурентна зборка	468.1 секунди
<i>Raspberry PI 3B+</i>	Конкурентна зборка з прунінгом	602.4 секунди
<i>Ryzen 9 3900X</i>	Конкурентна зборка з прунінгом	116.1 секунди
<i>EC2 t4g.nano</i>	Конкурентна зборка з прунінгом	471.2 секунди

Як видно з таблиці 4.3. конкурентна зборка з прунінгом дозволяє зберігати час та зменшити навантаження на ресурси системи, в тому числі на системі на якій будуть виконуватись задачі. На рисунках 4.2. та 4.3. наглядно показані результати часу зборки для порівняння комбінації різних методів та апаратного забезпечення.

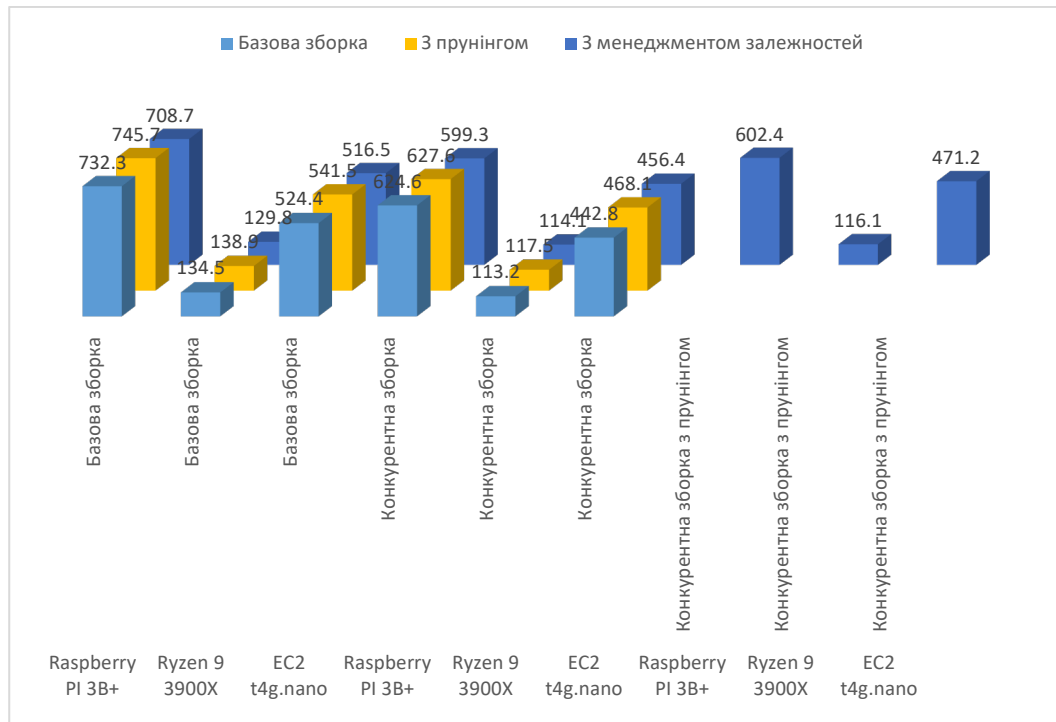


Рис. 4.2. Порівняльний графік представлених методів зборки контейнеру

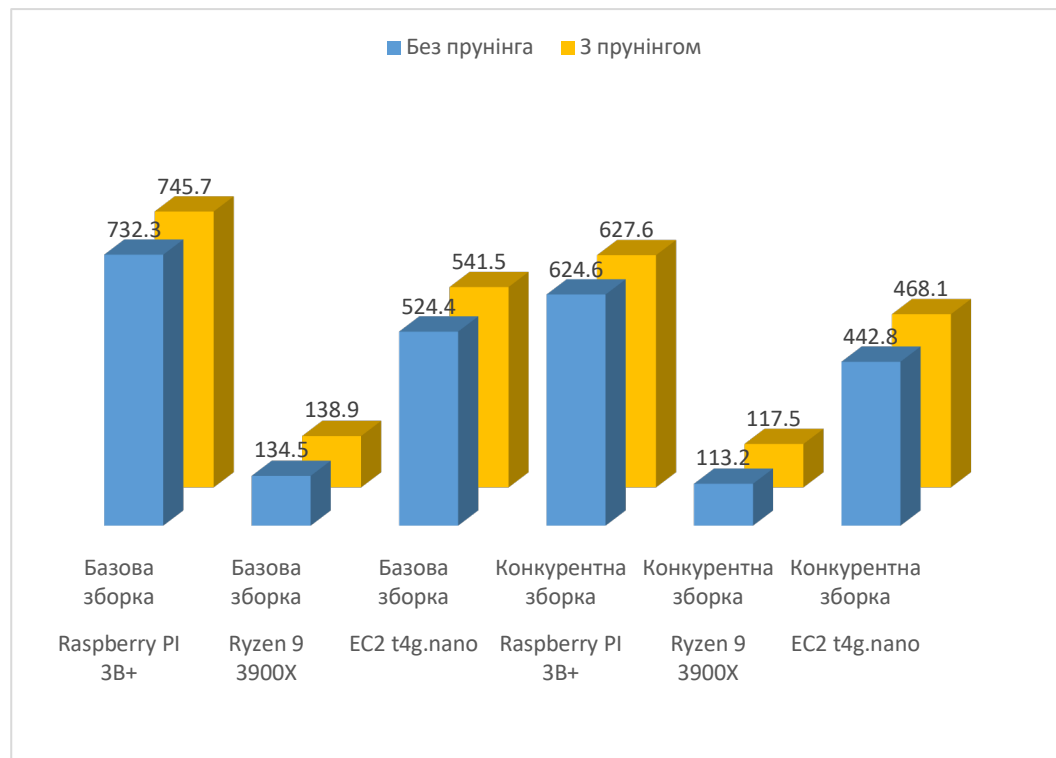


Рис. 4.3. Порівняльний графік методів з прунінгом та без прунінга

Тестування *API* на компонентах із задачею ШІ платформи вбудованих систем є важливим етапом в розробці, який включає в себе оцінку функціональності, надійності та відповідності *API* специфікаціям. Раніше було розглянуто один з найважливіших аспектів тестування, який включає перевірку правильності відповідей системи на вхідні дані, що легко можна розробити за допомогою unit-тестів в *python*. Іншим цікавим аспектом є оцінка часу відповіді та тестування навантаження. Аналізується продуктивність *API* в умовах навантаження та його здатність до масштабування за допомогою масового виклику *API*. Застосування тестування *API* компоненти платформи вбудованих систем дозволяє підтвердити її здатність до ефективної роботи у реальних умовах, коли більше ніж один вузол робить запит, і забезпечує певний рівень масштабованості та відмовостійкості.

Postman та *Newman* – інструменти, які використовуються для автоматизації тестування та управління *API*. *Postman* є інтегрованою розробкою *API*, яка надає графічний інтерфейс для тестування та документації *API*. Він підтримує автоматизацію тестування за допомогою колекцій та наборів тестів, що дозволяє ефективно перевіряти функціональність *API*. *Newman*, у свою чергу, є інструментом командного рядка, який розширює можливості *Postman* шляхом виконання автоматизованих тестів безпосередньо з командного рядка. Він інтегрується з *CI/CD* конвеєрами та дозволяє виконувати тестування *API* в автоматичному режимі під час розробки та впровадження програмного забезпечення. Використання *Newman* в тандемі з *Postman* дозволяє легко задавати параметри для тестування та мігрувати їх в форматі *JSON* для виконання в *newman*.

Для валідації успішного розгортання системи, було проведено тестування навантаження компоненти платформ вбудованих систем, конкретно, вузлу на основі вбудованих систем з метою збору інформації щодо потенційних проблем в обслуговуванні. Тестування включало в себе відправку значної кількості запитів на вузол, на 2 кінцеві точки. Таким чином, було перевірено систему на одразу

декілька можливостей: коректна робота алгоритму, правильність кінцевого результату *CI/CD* конвеєру та здатність виконувати обробку даних за допомогою моделі ІІІ.

Таблиця 4.4

Тестування роботи платформи

Тип пристрою	Тип кінцевої точки	Час обробки
<i>Raspberry PI 3B+</i>	Повний тест (100 ітерацій)	195.2 секунди
<i>Raspberry PI 3B+</i>	Розпізнавання (2 тести, 100 ітерацій)	192.5 секунди
<i>Raspberry PI 3B+</i>	Моніторинг працездатності	2.7 секунд
<i>Ryzen 9 3900X</i>	Повний тест (100 ітерацій)	30.19 секунди
<i>Ryzen 9 3900X</i>	Розпізнавання (2 тести, 100 ітерацій)	29.47 секунди
<i>Ryzen 9 3900X</i>	Моніторинг працездатності	0.72 секунди

Далі, проведено низку тестів з використанням різних моделей ІІІ, таких як *Inception*, *ResNet* та *MobileNet*. Ідеєю проведення даного тесту є перевірка впливу моделі на загальну працездатність системи та перевірка можливостей системи до пристосування до інших моделей, які потребують більше пам'яті та інших обчислювальних ресурсів. Тести були проведені подібно попереднім тестам навантаження за допомогою пакету *newman* та з використанням скриптів *python* для автоматизації отримання результатів. Для перевірки відмовостійкості системи, було виконано 40 пакетів з 3 запитами на 1, 2, 5, 10 та 25 потоків. Даний тест проводиться без урахування відмов та використовуючи лише одну гілку топології, як зазначено на малюнку нижче.

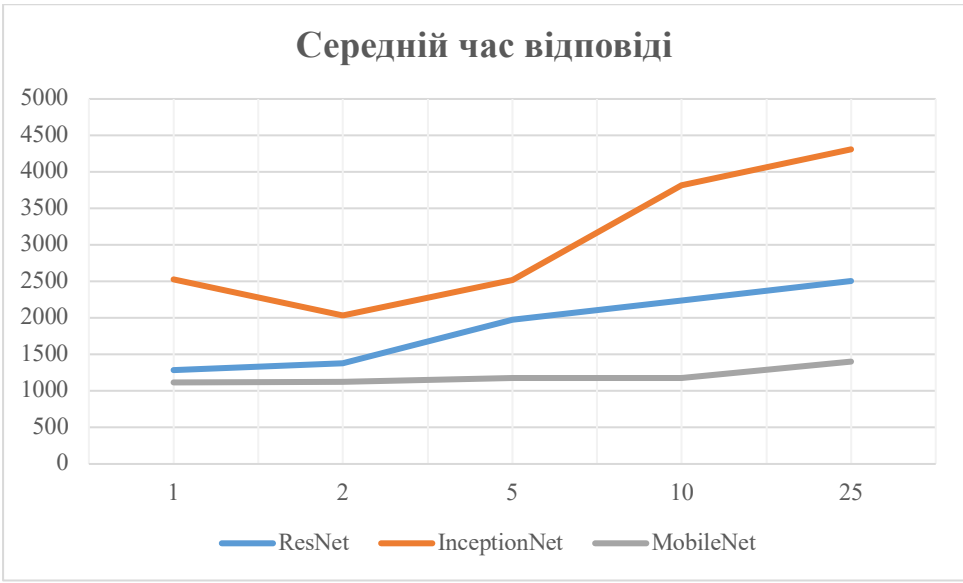


Рис. 4.4. Порівняльний графік середнього часу відповіді при різній кількості потоків запитів для одного вузлу платформи

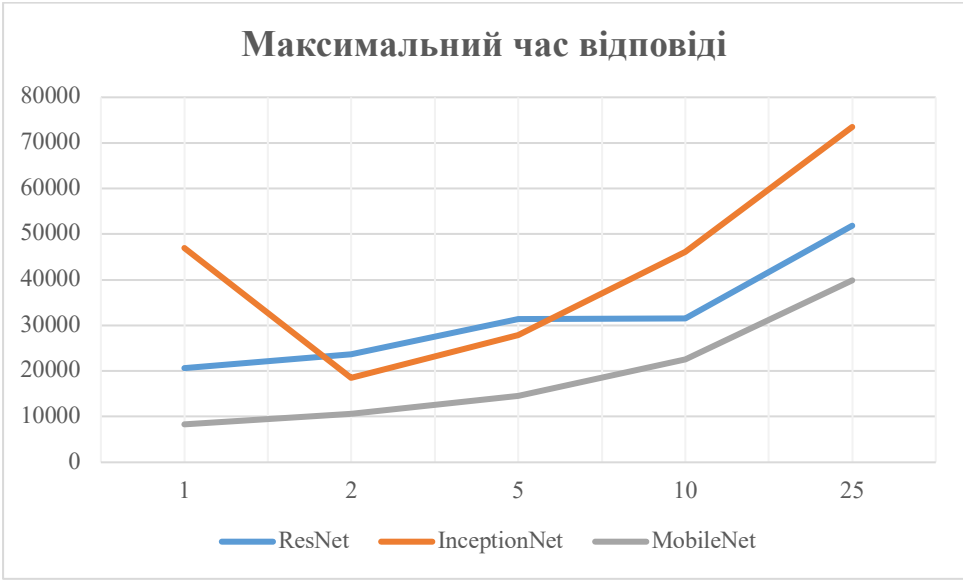


Рис. 4.5. Порівняльний графік максимального часу відповіді при різній кількості потоків запитів для одного вузлу платформи

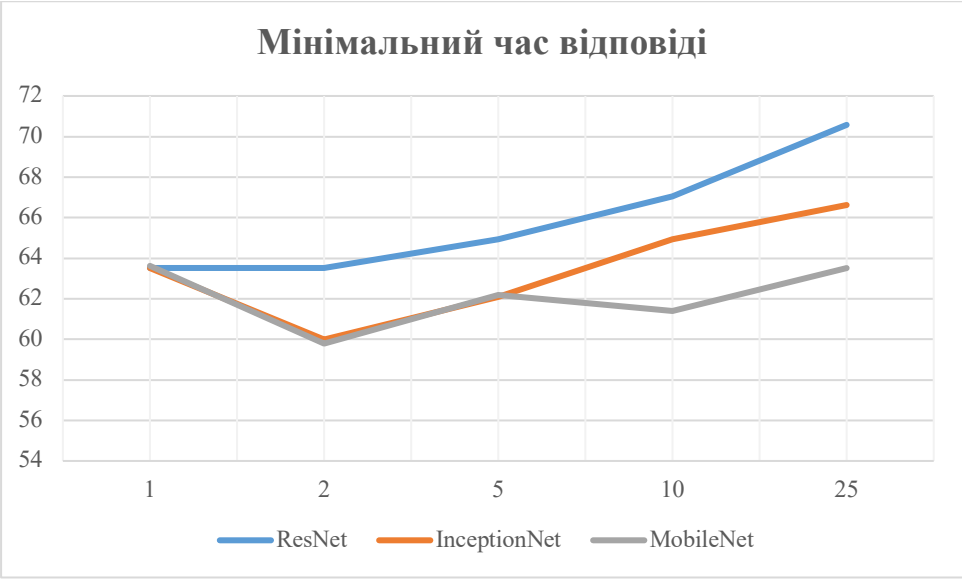


Рис. 4.6. Порівняльний графік мінімального часу відповіді при різних кількості потоків запитів для одного вузлу платформи

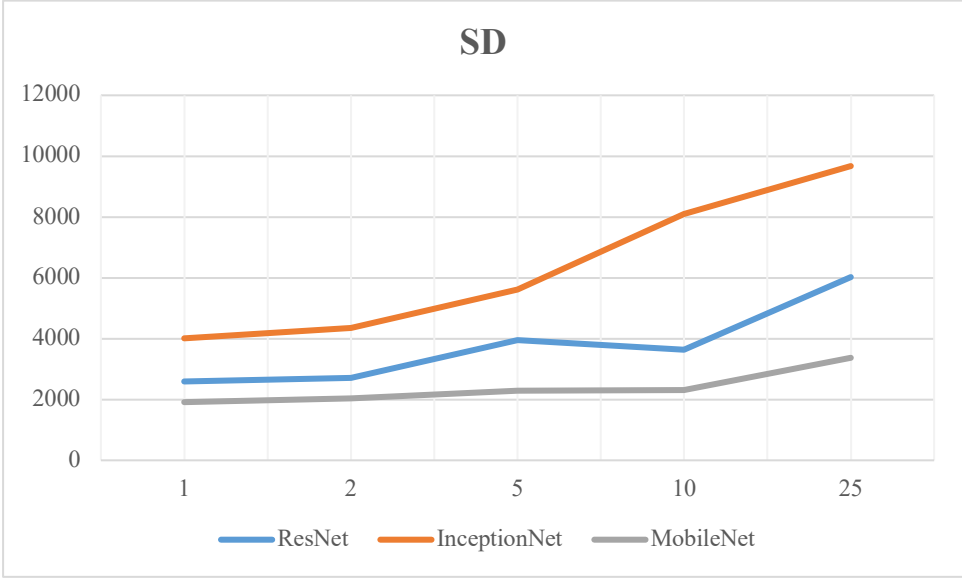


Рис. 4.7. Порівняльний графік стандартного відхилення відповіді при різних кількості потоків запитів для одного вузлу платформи

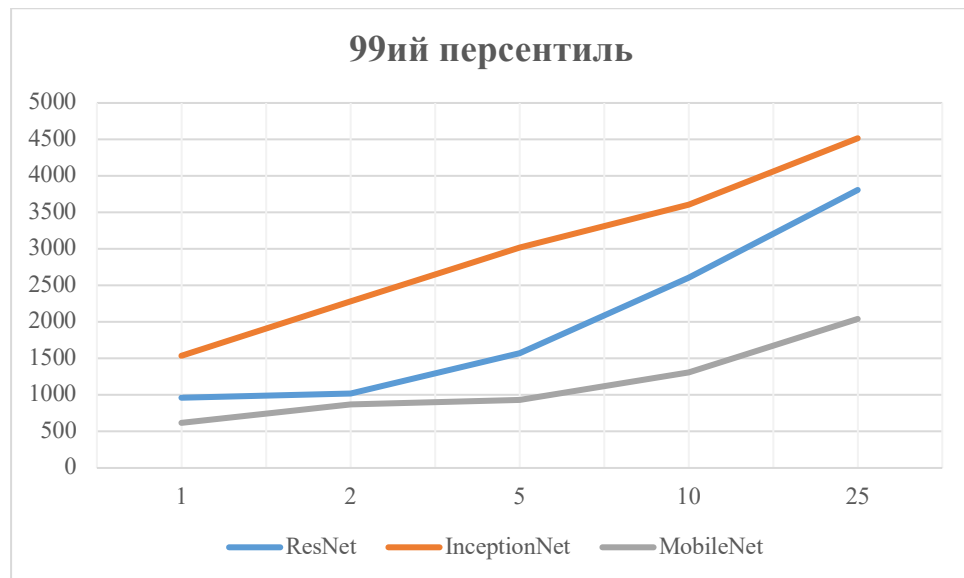


Рис. 4.8. Порівняльний графік часу перцентилу 99 відповіді при різних кількості потоків запитів для одного вузлу платформи

На рисунках 4.4-4.8 наведені графіки зі значеннями часу та стандартного відхилення для тестування навантаження системи, з використанням моделі *ResNet*. Основна ідея *ResNet* полягає в запровадженні залишкових блоків, що використовують механізм прямого пропуску для полегшення навчання глибоких нейронних мереж. Це дозволило оптимізувати модель *CNN* і використовувати її на менш потужних системах. Можна побачити з графіків, що середній час відповіді системи зростає з кількістю запитів, для 25 клієнтів час відповіді становить понад 2500 секунд.

На рисунках 4.4-4.8 наведені графіки для мережі *InceptionNetV3*. Архітектура *InceptionNet* запроваджує концепцію "*Inception* блоків", які включають паралельні шляхи обробки даних через різні типи згортальних шарів: звичайні згортки, згортки з великим ядром, пулінгові шари та їх комбінації. Як видно з графіків, часові значення значно гірші ніж для даної сім'ї мереж.

На рисунках 4.4-4.8 наведені графіки для *MobileNetV3*. Дана модель була обрана як основна для дослідження завдяки впровадженню розділених згорток які значно знижують обчислювальну складність порівняно з традиційними згортками. Цікавим є те, що середній та найменший час відгуку системи незначно

змінюється, і найбільший час залишається значно меншим ніж для моделей *InceptionNet* та *ResNet*.

Таблиця 4.5

Тестування навантаження вузла платформи

<i>Tavg</i>	<i>Tmax</i>	<i>T₉₉</i>	<i>Tmin</i>	<i>SD</i>	<i>Потоки</i>
<i>ResNet</i>					
1285.7	20648	1035.55	63.517	2592.4	1
1379.8	23681	1099.55	63.517	2708.7	2
1977.2	31359	1690.63	64.929	3952.4	5
2234.9	31540	2807.05	67.046	3643.6	10
2506.2	51866	4098.94	70.575	6024.5	25
<i>InceptionNet</i>					
2525	47039	1653.98	63.517	4010.7	1
2034.1	18519	2458.57	59.989	4346.4	2
2518.8	27836	3252.11	62.106	5607.8	5
3817.9	46181	3882.75	64.929	8098.7	10
4310.1	73512	4861.06	66.623	9674.8	25
<i>MobileNet</i>					
1116.7	8323.8	664.9	63.517	1912.6	1
1124.2	10647	937.8	59.989	2041.5	2
1181.9	14531	1002.9	62.106	2297.4	5
1181.7	22500	1412.5	61.4	2318.8	10
1402.3	39887	2197.2	63.517	3371.2	25

Зверху наведена зведена таблиця результатів тестування навантаження системи, за допомогою пакету *newman*. Як видно з результатів тестування, результати для мережі *MobileNet* є найбільш швидкими, і кількість потоків впливає найменше на середній час обробки одного запиту. Можна виділити, що тенденція на збільшення часу відгуку системи залишається для всіх вказаних моделей.

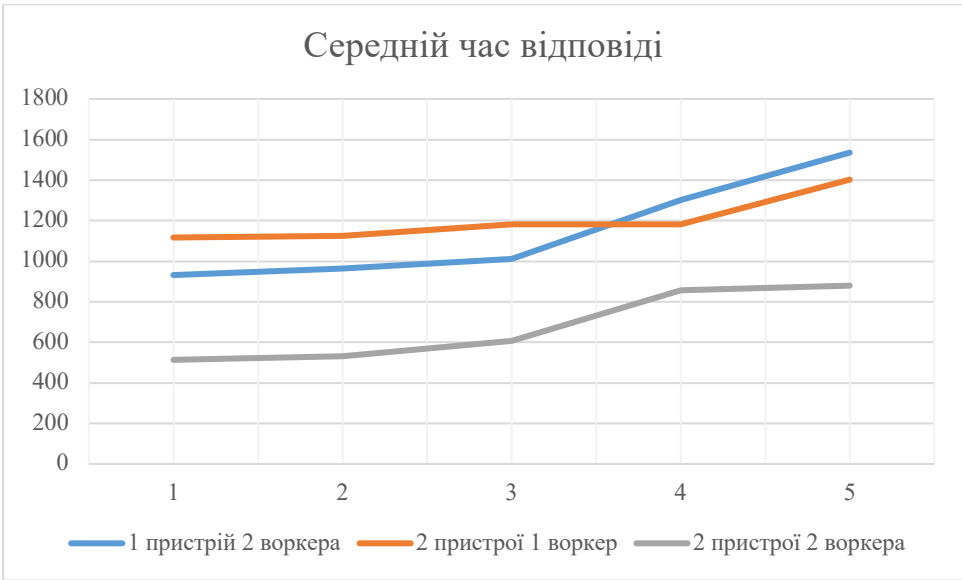


Рис. 4.9. Порівняльний графік середнього часу відповіді при різній кількості потоків запитів для платформи

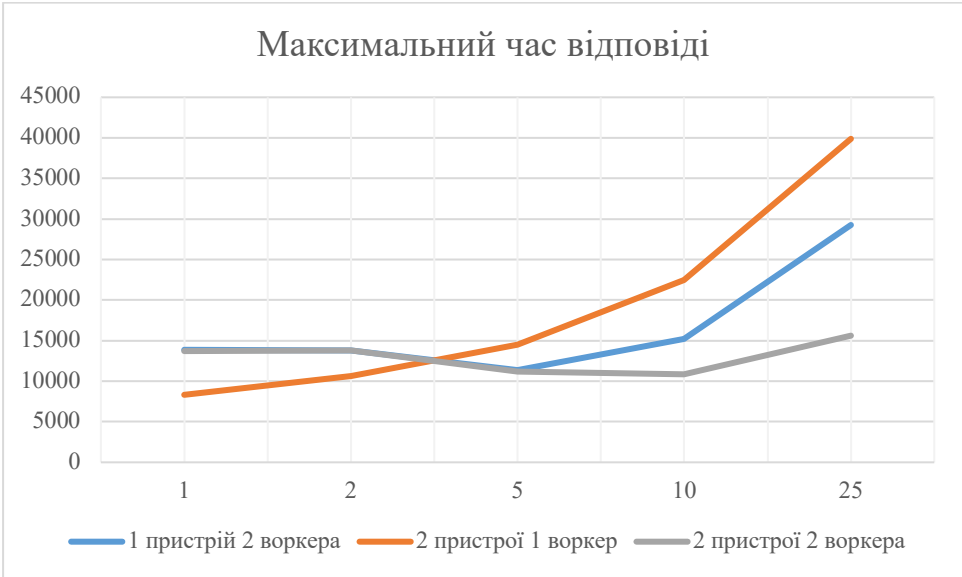


Рис. 4.10. Порівняльний графік максимального часу відповіді при різній кількості потоків запитів для платформи

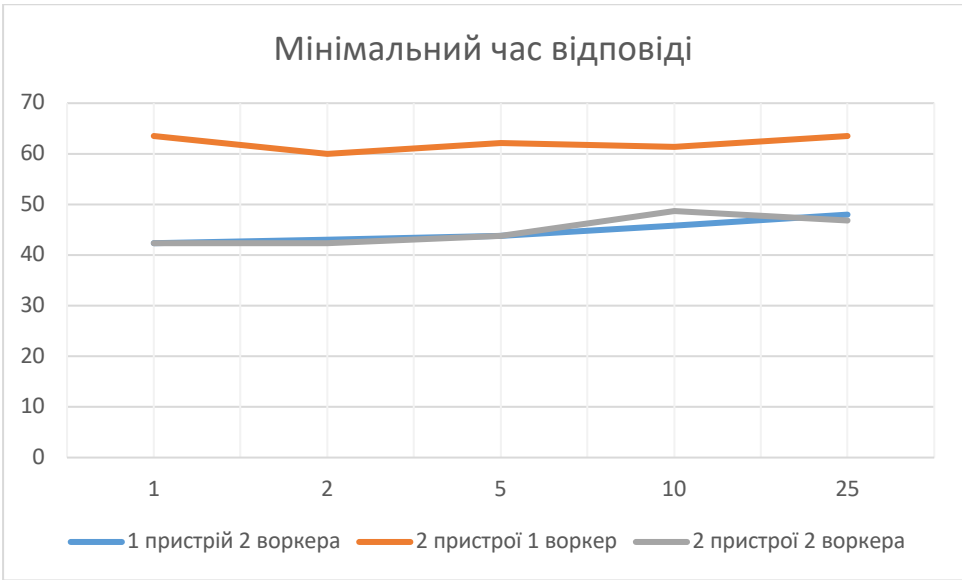


Рис. 4.11. Порівняльний графік мінімального часу відповіді при різних кількості потоків запитів для платформи

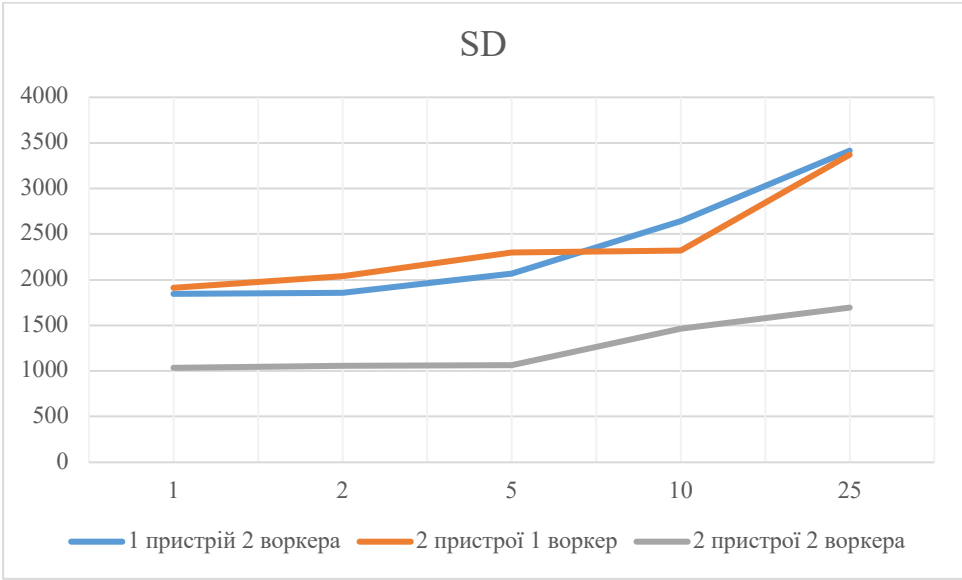


Рис. 4.12. Порівняльний графік стандартного відхилення при різних кількості потоків запитів для платформи

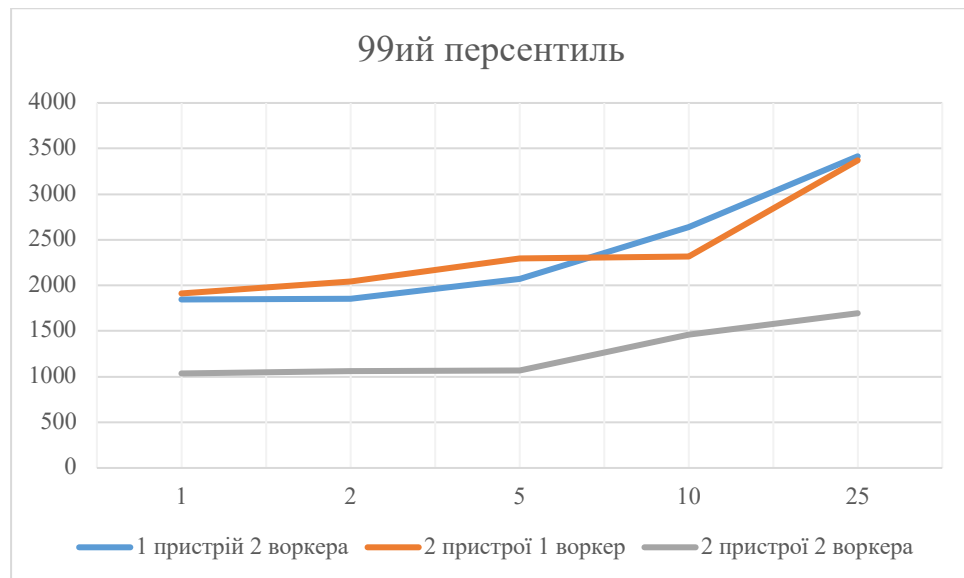


Рис. 4.13. Порівняльний графік персентилу 99 відповіді при різній кількості потоків запитів для платформи

На рисунку 4.9-4.13 наведені графіки для *MobileNetV3* для трьох пристроїв, два з яких містять контейнеризовані моделі ШІ, а також вузол-агрегатор, який забезпечує балансування навантаження між ними, як і в попередньому прикладі. У даному випадку для підвищення ефективності обробки використовуються два воркери, які забезпечують додаткове балансування навантаження на рівні окремого пристрою. Це дозволяє зменшити затримки в обробці запитів та підвищити загальну продуктивність системи, завдяки оптимізації розподілу обчислювальних ресурсів між різними компонентами системи. Графіки відображають динаміку навантаження та продуктивності на кожному з пристроїв, а також взаємодію між воркерами та контейнерами моделей, що демонструє ефективність такої архітектури в умовах реального навантаження.

Таблиця 4.6

Тестування навантаження частини платформи

<i>Tavg</i>	<i>Tmax</i>	<i>T₉₉</i>	<i>Tmin</i>	<i>SD</i>	<i>Потоки</i>
<i>MobileNet</i>					
931.61	13872.96	839.44	42.34	1846.33	1
964.92	13776.17	948.56	43.05	1855.14	2
1011.02	11369.37	1008.52	43.75	2068.31	5

Продовження таблиці 4.6.

<i>Tavg</i>	<i>Tmax</i>	<i>T₉₉</i>	<i>Tmin</i>	<i>SD</i>	<i>Потоки</i>
1302.33	15227.99	1198.00	45.87	2640.26	10
1535.48	29263.55	1660.89	47.99	3415.72	25
<i>MobileNet Raspberry PI Tree-Debrujin (2 degree)</i>					
2525	47039	1653.98	63.517	4010.7	1
2034.1	18519	2458.57	59.989	4346.4	2
2518.8	27836	3252.11	62.106	5607.8	5
3817.9	46181	3882.75	64.929	8098.7	10
4310.1	73512	4861.06	66.623	9674.8	25
<i>MobileNet Raspberry PI Tree-Debrujin (2 degree, 2 workers)</i>					
513.59	13679.38	401.73	42.34	1035.63	1
532.18	13776.17	560.62	42.34	1060.80	2
605.79	11175.8	491.67	43.76	1066.26	5
857.04	10853.17	896.40	48.69	1461.96	10
879.48	15617.72	915.23	46.86	1695.69	25

Частиною відмовостійкості запропонованої системи є можливість відновлення працездатності при відмові одного з компонентів системи. В рамках цього, важливим є проведення експериментів на швидкість відновлення системи в разі відмови. Для цього, проведемо експеримент, при якому відмовляє один з вузлів нижнього ярусу системи.

Таблиця 4.7

Тестування відновлення платформи

Етап конвеєру	Час виконання
<i>Розгортання системи на нижньому ярусі</i>	
<i>Включення вбудованого пристрою до мережі</i>	32.42 секунди
<i>Відправлення контейнеру на нижній ярус</i>	12.84 секунди
<i>Встановлення відповідних програмних пакетів</i>	152.63 секунди
<i>Розгортання системи на верхньому ярусі</i>	
<i>Розгортання Cloud-сервісів</i>	44.98 секунди
<i>Тестування ПЗ</i>	63.11 секунд
<i>Встановлення необхідного ПЗ на віртуальних машинах</i>	56.39 секунд
<i>Перевірка працездатності системи</i>	12.31 секунди

4.3.1. Експерименти з масштабування платформи

Виходячи з описаних раніше вимог, в даному пункті буде описані експерименти з масштабування платформи. Метою даного експерименту є часова оцінка платформи при збільшенні кількості підключених пристроїв. В якості експерименту пропонується провести моделювання компонент платформи вбудованих систем для великої кількості пристроїв. Для виконання цього, пропонується використати раніше описане апаратно-програмне рішення, в тому числі, схема масштабування за використанням гібридної топології *Tree-DeBrujin*. Отримання результатів масштабування дозволить проаналізувати часову характеристику платформи, її здатність обробляти запити при збільшенні кількості пристроїв, і про ефективність масштабування з використанням запропонованого методу. Оцінка результатів з таблиці 4.8. цього експерименту дасть змогу зробити висновки щодо масштабованості платформи і визначити можливі обмеження та шляхи для їх подолання в процесі подальшого розвитку системи.

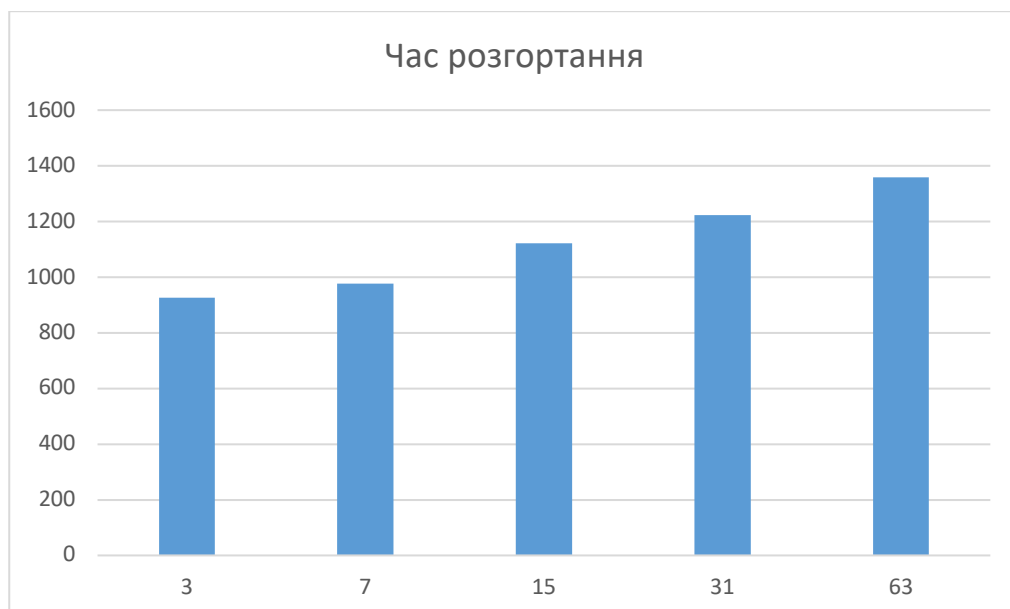


Рис. 4.14. Порівняльний графік часу розгортання платформи в умовах масштабування для топології *Tree-DeBrujin*

Розгортання в умовах масштабування платформи

Кількість пристроїв	Час розгортання
3	925.45 секунди
7	977.08 секунди
15	1122.28 секунди
31	1222.31 секунди
63	1359.14 секунди

В результаті даного експерименту було встановлено збільшення часу розгортання, як наслідок масштабування системи за рахунок додаткових пристроїв. Як видно з рисунку 4.14, зі збільшенням числа пристроїв платформа потребує більшого часу для розгортання, що свідчить про певну залежність часу розгортання від масштабування системи. Даний результат вказує на те, що з ростом кількості підключених пристроїв, зростає навантаження необхідність в додаткових пересилках, в умовах збільшення навантаження на вузли, що в свою чергу вимагає більше часу для налаштування та синхронізації всіх елементів.

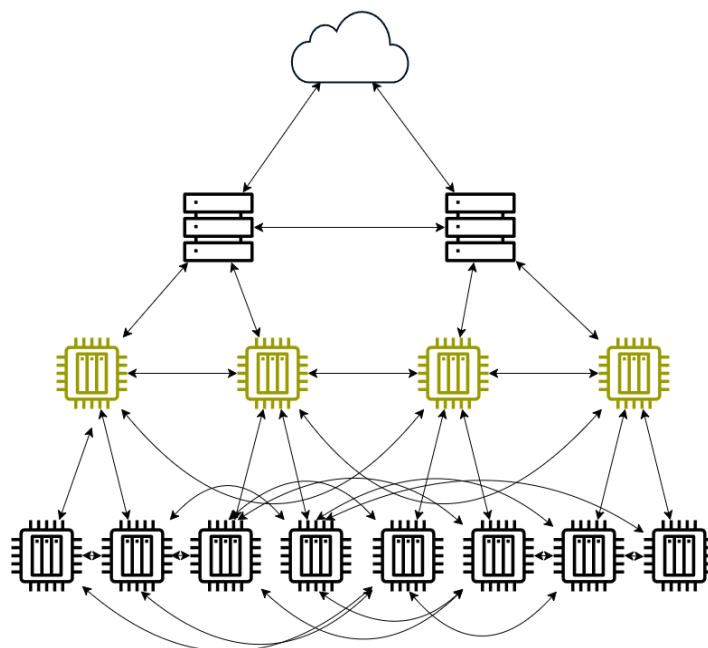


Рис. 4.15. Платформа на основі топології Tree-DeBruijn з виділеними вузлами-посередниками

У наступному експерименті буде досліджено вплив на платформу відмов окремих її пристроїв, що дозволить оцінити стійкість і надійність системи в умовах непередбачених збоїв, притаманних вбудованим пристроям. Планується симулювати відмови різної кількості пристроїв, що складає до 10% платформи, а також дослідити, як ці відмови впливають на загальний час розгортання та ефективність роботи системи в цілому. Відмови, які симулюються на платформі, пов'язані з неможливістю перевірити стан пристрою або перевірку працездатності (*healthcheck*). Моделювання проведене для 31 та 63 пристроїв.

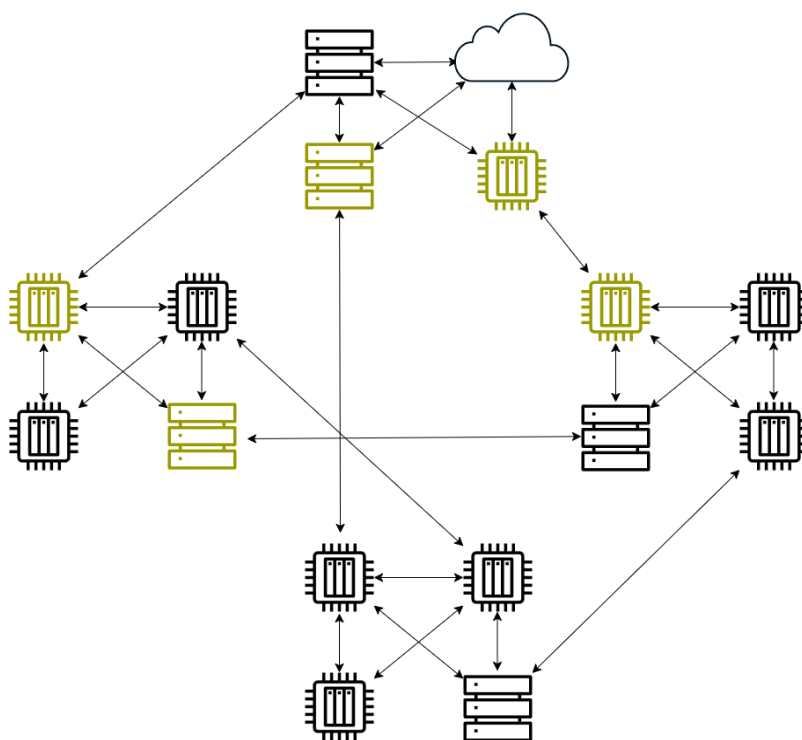


Рис. 4.16. Платформа на основі топології Dragon-DeBrujin з виділеними вузлами-посередниками

В даному експерименті досліджено вплив на платформу відмов її компонент, при підключенні на основі топологічної організації *Dragon-DeBrujin*. Аналогічно, буде симульовано відмови зростаючої кількості пристроїв, що складає до 10% платформи, а також досліджено, їх вплив на загальний час розгортання та ефективність роботи системи в цілому. Моделювання проведене для 30 та 42 пристроїв.



Рис. 4.17. Порівняльний графік часу розгортання платформи в умовах масштабування для топології *Dragon-DeBruijn*

Результати експерименту щодо розгортання системи на платформі, побудованій за топологією *Dragon-DeBruijn*, показують зріст часу розгортання в залежності від кількості пристроїв. Зі збільшенням числа пристроїв від 12 до 42 спостерігається поступове збільшення часу, необхідного для розгортання системи. Зокрема, для 12 пристроїв час розгортання становить 1032.38 секунди, тоді як для 42 пристроїв цей показник досягає 1278.25 секунди. Це свідчить про лінійне зростання часу в процесі масштабування платформи, що може бути пов'язано з додатковими вимогами до ресурсів і складності управління мережею при збільшенні кількості вузлів.

Таблиця 4.9

Час розгортання в умовах масштабування платформи

Кількість пристроїв	Час розгортання
12	1032.38 секунди
20	1106.43 секунди
30	1218.01 секунди
42	1278.25 секунди

Розгортання системи при відмові вузлів посередників для 42 та 30 пристроїв в платформі об'єднаній в топологію Dragon-Debrujin

Кількість пристроїв	Час розгортання
62	1341.09 секунди
61	1351.53 секунди
60	1350.60 секунди
59	1437.05 секунди
58	1388.1 секунди
57	1385.24 секунди
56	1377.48 секунди
30	1142.83 секунди
29	1148.77 секунди
28	1160.43 секунди
27	1219.5 секунди
26	1234.15 секунди
25	1222.74 секунди

Для платформи з 63 пристроями спостерігається незначне (до 7.15%) збільшення часу розгортання при відмовах пристроїв. Наприклад, для випадку на 59 пристроїв, результати свідчать про незначний, але нетиповий вплив відмов на загальний час. Такий результат пов'язаний з тим, що вузол-посередник, який мав відмову, мав значну кількість пов'язаних з ним вузлів, які потребували додаткової маршрутизації та відправки задач на них.

Таблиця 4.11

Розгортання системи при відмові вузлів посередників для 42 та 30 пристроїв в платформі об'єднаній в топологію Dragon-Debrujin

Кількість пристроїв	Час розгортання
41	1209.39 секунди
40	1311.57 секунди
39	1310.77 секунди
38	1213.84 секунди
37	1244.58 секунди

Продовження таблиці 4.11

Кількість пристроїв	Час розгортання
36	1241.87 секунди
29	1124.74 секунди
28	1147.78 секунди
27	1109.32 секунди
26	1131.57 секунди
25	1124.74 секунди

Для платформи з 31 пристроєм збільшення часу розгортання є більш значними (до 7.99%), та спостерігається збільшення часу розгортання в умовах відмов, що зумовлене необхідністю в реконфігурації задач, подібно з експериментом на 63 вузли. Ефект реконфігурації був менший, для вузла-посередника, що відмовив, на 26 пристроях, порівняно з минулим експериментом, що пов'язано з меншою кількістю залежних від нього вузлів, яким постачаються задачі через вузол-посередник.

4.3.2. Порівняння з аналогічними платформами

Щоб отримати ширшу картину роботи платформи, пропонується розглянути використання платформ-аналогів, які були розглянуті в першому розділі. Кожна платформа має свої переваги та недоліки, як в інженерному плані, так і в науковому. Застосування схожого програмного стеку в даному експерименті буде слугувати ключовим моментом для можливості репродукувати результати. Отже, виходячи з вимог до програмної складової системи, в даному пункті будуть продемонстровані результати експериментів з використання аналогічних платформ.

Розгортання платформи на основі запропонованого методу в порівнянні з аналогічними підходами

Тип платформи	Час розгортання
<i>Запропонований метод</i>	925.45 секунди
<i>AWS SageMaker</i>	977.08 секунди
<i>AWS SageMaker + GreenGrass IoT</i>	1122.28 секунди
<i>Мікросервісний Edge MLOps</i>	1222.31 секунди
<i>ERAIA</i>	1359.14 секунди

В даному експерименті, окремо визначене рішення *AWS SageMaker* та *AWS SageMaker + GreenGrass IoT*, оскільки можливості самого *AWS SageMaker* обмежені лише для використання на рівні *Cloud*. Окрім даної тенхології, також для побудови повного конвеєру, аналогічного запропонованому в рамках досліджуваного методу, необхідно використати додаткові допоміжні технології, як зазначені в *whitepaper*[81].

Модель *ERAIA* була модифікована таким чином, що замість використання *Device generator*, запропонованого в статі, був використаний підхід з використанням *IaC* для швидкого розгортання на існуючі пристрої в мережі.

Тестування навантаження вузла платформи на основі запропонованого методу в порівнянні з аналогічними підходами

<i>Tavg</i>	<i>Tmax</i>	<i>T99</i>	<i>Tmin</i>	<i>SD</i>	<i>Потоки</i>
<i>Запропонований метод</i>					
1116.7	8324	664.90	63.52	1912.6	1
1124.2	10647	937.80	59.99	2041.5	2
1181.9	14531	1002.90	62.11	2297.4	5
1181.7	22500	1412.50	61.40	2318.8	10
1402.3	39887	2197.20	63.52	3371.2	25

<i>Tavg</i>	<i>Tmax</i>	<i>T99</i>	<i>Tmin</i>	<i>SD</i>	<i>Потоки</i>
<i>AWS SageMaker</i>					
1407.5	9904	899.08	65.32	2631.9	1
1468.3	12091	1349.44	62.53	2918.3	2
1475.8	16676	1219.92	66.20	2863.1	5
1504.6	29871	1386.28	64.80	3260.4	10
1710.2	40476	1956.49	64.84	3831.5	25
<i>AWS SageMaker + GreenGrass IoT</i>					
1139.8	8686	703.53	64.24	2002.2	1
1180.3	11182	1063.85	61.42	2302.3	2
1250.6	15160	1041.83	64.94	2483.5	5
1227.6	24266	1349.93	63.30	3101.4	10
1519.2	36953	2061.35	64.43	3440.6	25
<i>Мікросервісний Edge MLOps</i>					
1096.5	8156.905	669.57	62.98	1838.2	1
1189.0	10276.08	958.52	60.09	1969.3	2
1244.8	14498.1	951.09	61.94	2259.6	5
1215.1	22736.84	1349.31	60.64	2867.1	10
1554.2	39616.3	2131.72	62.22	3225.2	25
<i>ERAIA (MQTT)</i>					
1172.7	8467.314	689.90	63.83	1927.1	1
1190.0	10909.57	1021.96	60.99	2182.2	2
1242.3	15157.95	1063.35	64.20	2410.3	5
1202.6	24334.81	1572.34	62.59	3026.9	10
1434.8	44139.02	2389.26	63.83	3425.9	25

З результатів, наведених в таблиці 4.14., середній час відповіді платформи, на основі запропонованого методу проявляє себе найкраще в порівнянні зі всіма аналогами. Виключенням є сценарій навантаження одним потоком запитів на платформу на основі мікросервісного підходу, що пояснюється використанням додаткових контейнерів на одному пристрої.

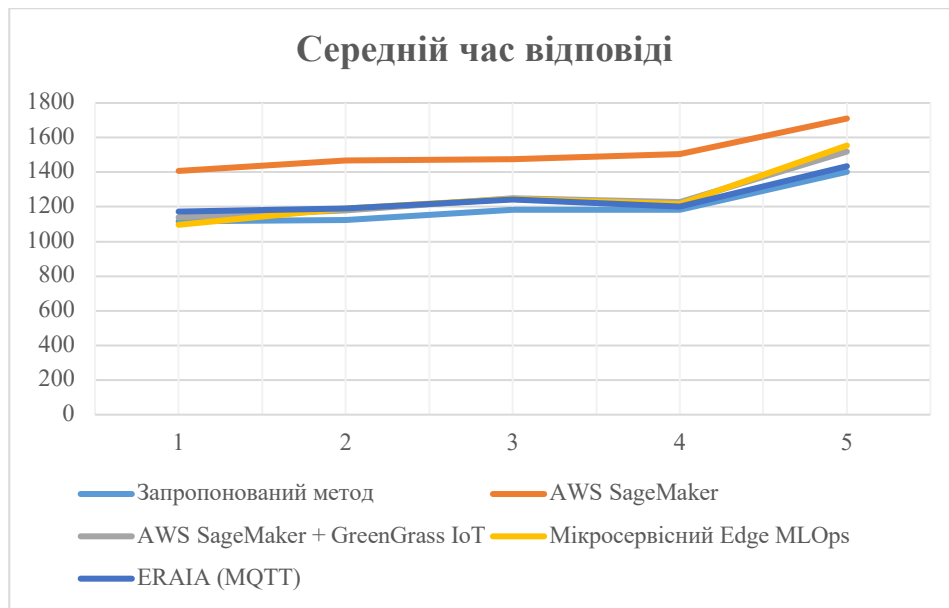


Рис. 4.18. Порівняльний графік середнього часу відповіді платформ на основі розглянутих методів

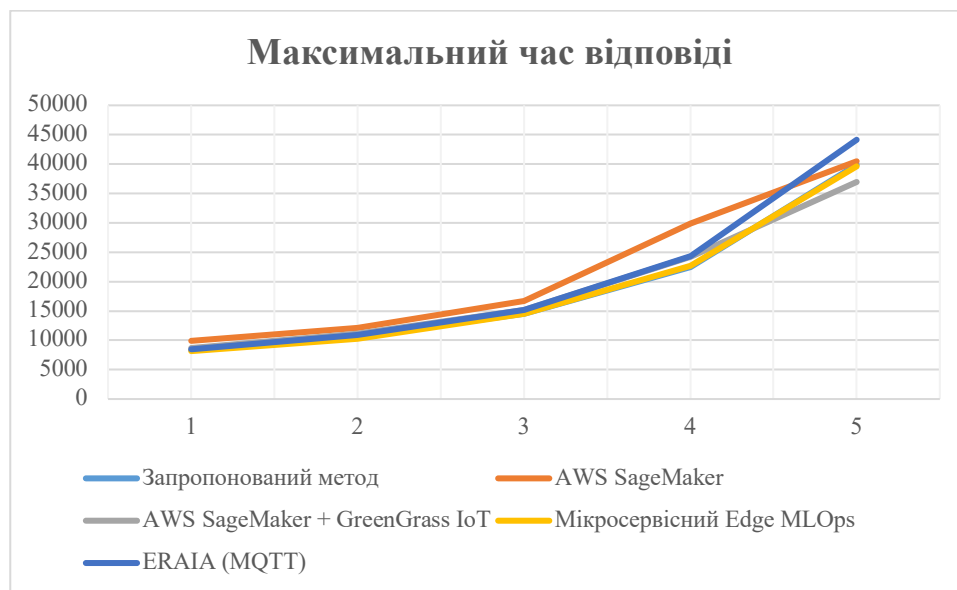


Рис. 4.19. Порівняльний графік максимального часу відповіді платформ на основі розглянутих методів

Середній час усіх платформ, з рисунку 4.18., залишається приблизно однаковим для всіх платформ. З рисунку 4.19 видно, що максимальний час відповіді є найменшим для запропонованого методу та методу на основі мікросервісного підходу, як результат схожості архітектури програмної частини платформи.

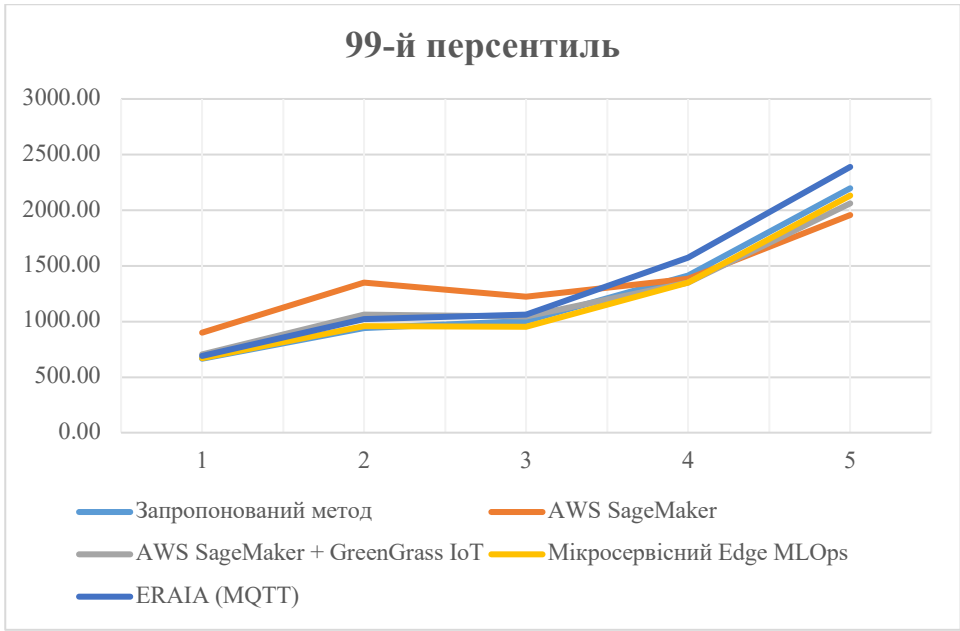


Рис. 4.20. Порівняльний графік персентилію 99 відповіді платформ на основі розглянутих методів

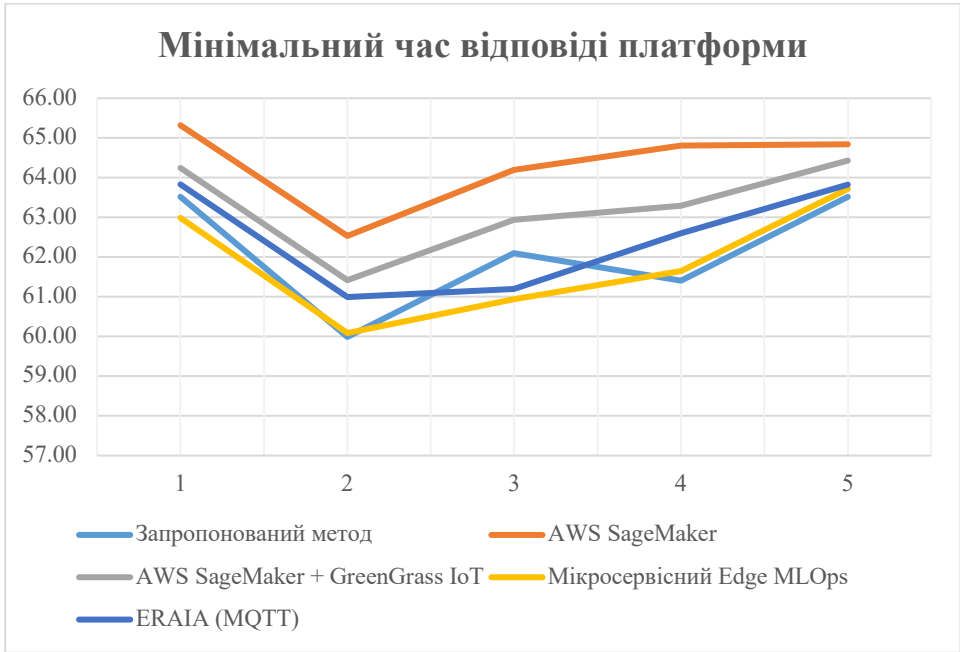


Рис. 4.21. Порівняльний графік мінімального часу відповіді платформ на основі розглянутих методів

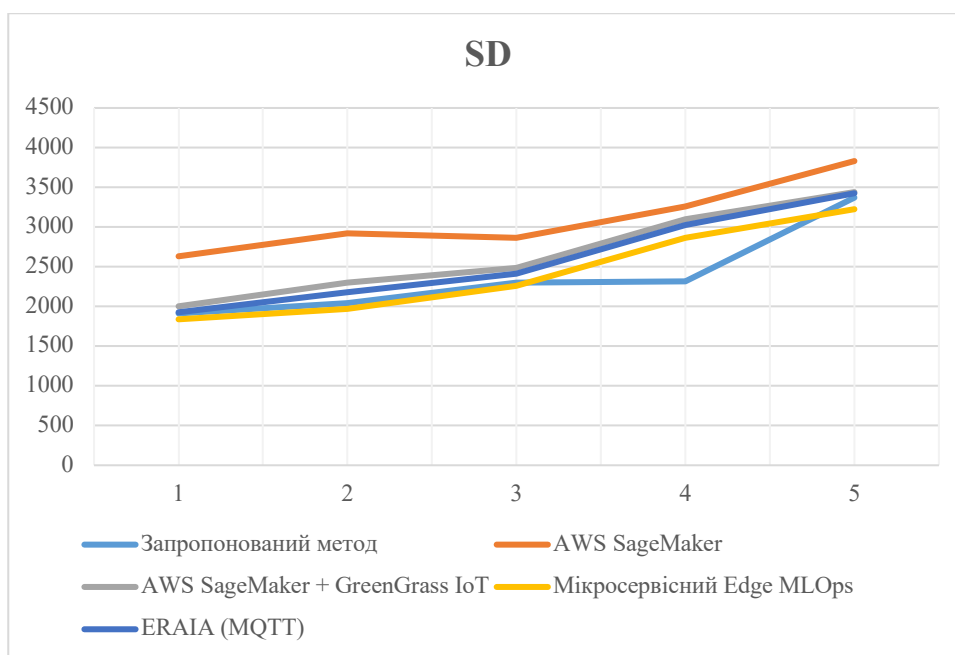


Рис. 4.22. Порівняльний графік стандартного відхилення відповіді платформ на основі розглянутих методів

З результатів, наведених в рисунках 4.20.-4.22., варто виділити, що підхід з використанням мікросервісної архітектури має схожі характеристики, що і запропонований метод, що пояснюється схожим підходом до розгортання, з використанням контейнеризованого застосунку. Очевидною перевагою підходів, з використанням вбудованих пристроїв, перед використанням системи, побудованій на *Cloud* архітектурі є середня швидкість обробки запитів, що до 27.3% швидше оброблюється, через затримку глобальної мережі. В тестах, де порівнюються платформи, які працюють на рівні *Edge*, відрив значно менший (від 2% до 11%) і пов'язаний з використанням додаткових кроків оптимізації, які впливають на швидкодію вбудованого пристрою. Основним недоліком запропонованого методу є відносно високий час 99 персентилю, що зазначено на рисунку 4.20., але варто врахувати, що цей показник залежить від архітектури системи, на якій розгортається додаток, на що вказує результати використання *Cloud* системи, які значно менші за інші показники. Мінімальний та максимальний час відповіді кожної платформи є порівнянними, що видно з рисунків 4.19. та 4.21..

4.4. Аналіз результатів моделювання методу розгортання компонент платформи вбудованих систем

Моделювання системи на основі запропонованого способу побудови дозволило отримати розуміння факторів, які впливають на працездатність системи та виміряти їх вплив з точки зору оптимізації системи, як часової так і в рамках системного ресурсу. Виходячи з отриманих даних, можна зробити висновок, що даний метод дозволяє розгорнути платформу здатну до швидкої реконфігурації в разі необхідності.

Основним способом проведення тестів для забезпечення стабільності та продуктивності систем, що використовують машинне навчання, є їх проведення протягом тривалого часу. В умовах швидко змінюваних даних регулярне тестування дозволяє виявити потенційні проблеми на ранніх етапах, та почати процес навчання нової моделі, за умови коли остання модель втратила точність. Тобто, безперервне тестування не лише оптимізує процеси, пов'язані з впровадженням і підтримкою систем машинного навчання, але також підвищує якість кінцевого продукту. Пропонується розглянути приклад тестування системи, з періодичним трігером на розгортання нової моделі раз у 8-12 годин. В реальних умовах, це може бути спричиненим недостатньою якістю моделі, або необхідністю використання іншої.

Таблиця 4.15

Тестування навантаження платформи протягом значного часу роботи

Показник	10 годин	24 години	Тиждень
Частота розгортання (DF)	1 розгортання	2 розгортання	14 розгортань
Час виконання змін (LT)	121 секунда	123 секунда	124 секунда
Середній час відновлення (MTTR)	305 секунд	306 секунд	311 секунд
Зміна частоти відмов (CFR)	0%	0%	0%
Uptime	99.664%	99.7157%	99.7126%

Проведення тестів протягом тривалого часу також сприяє створенню зворотного зв'язку дозволяє оперативно отримувати інформацію про працездатність системи та вносити зміни у стратегії розгортання моделі. Час виконання змін залишається відносно стабільним: для 10 годин він становить 121 секунду, для 24 годин — 123 секунди, а для тижня — 124 секунди, що може бути зумовлено накопиченням налаштувань чи інших транзитивних параметрів системи. Середній час відновлення демонструє також незначне зростання, починаючи з 305 секунд для 10 годин і до 311 секунд для тижня. Це може вказувати на те, що з часом система потребує більше часу для відновлення через збільшення кількості розгортань або накопичення помилок. Показник *uptime* виявляється дуже стабільним, з незначними змінами: для 10 годин — 99.664%, для 24 годин — 99.7157%, а для тижня — 99.7126%.

Таблиця 4.16

Тестування навантаження системи протягом значного часу роботи при зміні частоти відмов

Показник	Тиждень
Частота розгортання (DF)	14 розгортань
Час виконання змін (LT)	125 секунда
Середній час відновлення (MTTR)	332 секунд
Зміна частоти відмов (CFR)	20%
Uptime	99.7126%
Всього відмов	2

Розглянемо таблицю 4.16, в якій показані основні метрики розгортання та використання системи впродовж деякого об'єму часу. Ключовими висновками, які можна отримати з розглянутих прикладів — це, по-перше, здатність підтримувати систему в працездатному вигляді за умови відмов, по-друге можливість підтримки системи в більше ніж 99.5% часу, за умови постійного

потоків відмов, по-третє середній час відновлення системи є доволі низьким, що дозволяє безперервно проводити зміни, за необхідності.

Одним з важливих факторів аналізу виступає модель ШНМ. Тестування системи з різними моделями може продемонструвати наскільки така система гнучка з точки зору варіювання використанням ресурсів системи для задоволення потреб програмного продукту, в даному випадку, конкретної моделі. Дані моделювання свідчать про те, що ШНМ *MobileNet* є найбільш доцільним вибором для даної платформи, з точки зору швидкості. Для подальшого аналізу доцільності такого вибору, пропонується розглянути навантаження системи, включно з використанням даної моделі, на ЦП.

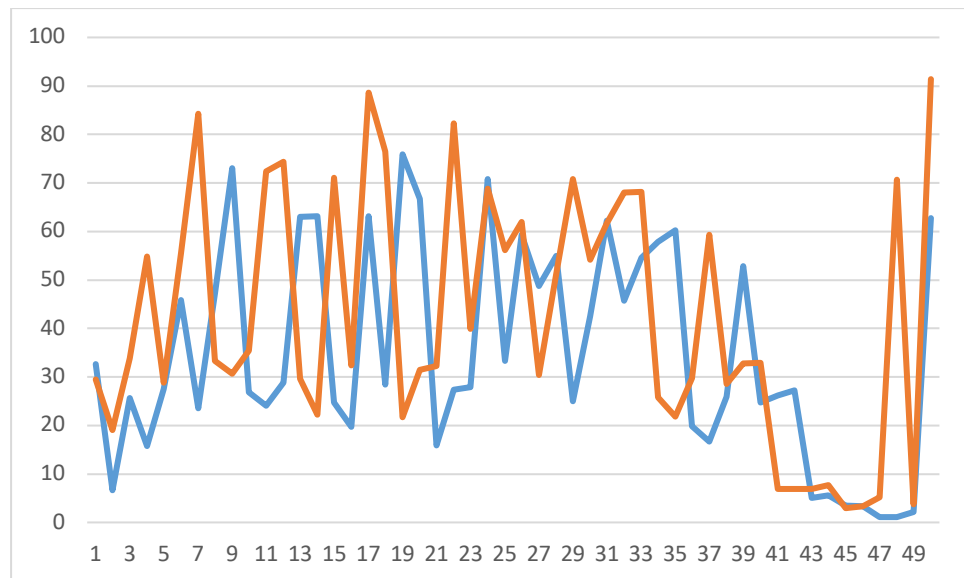


Рис. 4.23. Графік завантаження ЦП при використанні моделі без прунінгу та з прунінгом

Зверху наведений графік завантаження ЦП під час проведення експерименту із відправлення запитів та запуску моделі *MobileNetV2* на *Raspberry PI 3B+*, враховуючи *Flask API* та інші допоміжні функції системи. Середній відсоток навантаження ЦП при використанні моделі з прунінгом – 34.93%, без прунінгу – 42.15%. Окрім навантаження на ЦП, представлений як *SoC*, також використовується оперативна пам'ять, розміром в 847 МБ. Питання використання ресурсів системи є важливим, оскільки платформи вбудованих систем не є

високопродуктивними окремо і потребують кроків оптимізації, з урахуванням цього. Для цього в рамках комплексного метода запропоновано спосіб прунінгу, який дозволяє значно обмежити використання ресурсів системи.

Відмовостійкість системи – важливий її аспект, оскільки в разі відмови системи, з’являється гіпотеза про те, що це відмова викликана саме специфікою способу розгортання системи. Будь-яка система виходить з ладу в той чи інший момент, це може бути викликано атакою або помилкою при написанні коду. Враховуючи незлічену кількість помилок, які можуть привести систему до стану, який не може бути пов’язаним з її коректною роботою, необхідно провести аналіз, виокремити фактори впливу на працездатність системи, дослідити наявні дані щодо використання прототипу. За допомогою цього кроку кілька конвеєрів можна розгорнути як єдину сутність – систему, на відміну від окремо взятої моделі. За допомогою такого підходу та низки інструментів досягаються безперервні та масштабовані конвеєри машинного навчання.

Подальшого розвитку в даній методології вимагає тестування системи протягом великого обсягу часу, зі зміщеннями в ефективності роботи моделі та необхідності базової моделі ШНМ в її перетренуванні. Виокремлені особливості роботи системи в подальшому можуть забезпечити більш стабільну платформу для проведення дослідницького аналізу та використання для розпізнавання образів, ніж нині існуючі аналогічні підходи. Отримані значення *uptime* в 99.9% недостатні для конкурування з існуючими хмарними рішеннями.

Окремо, потрібно сказати, що для найбільш ефективного навчання моделі ШНМ необхідні якісні дані. Серед іншого, потенціальний дисбаланс може виникнути саме через використання неякісних даних, тому потрібно враховувати в процесі перенавчання такі фактори як відповідність даних до відповідних класів, для недетермінованого навчання, перевірка метаданих, аугментація та доповнення даних та оптимізація моделі ШНМ.

Висновки до розділу 4

В даному розділі було виконано моделювання розгортання компонент платформи вбудованих систем на основі запропонованого методу та проведено аналіз на основі отриманих показників. Для проведення експериментів, на основі досліджених методів, було запропоновано власний стек програмних засобів та архітектурного підходу, що дозволило використати дані методи та порівняти їх вплив на систему. За результатами моделювання, можна отримати висновок, що було підвищено ефективність застосування практик *MLOps* для платформи вбудованих систем за рахунок використання даного методу.

Оптимізація процесу зборки контейнера для подальшого розгортання компонент платформи вбудованих систем показало кращі результати, в порівнянні з базовим підходом. Було отримано прискорення процесу зборки на 5.58%. Було досліджено застосування топологій *Dragon-DeBrujin* та *Tree-DeBrujin*, які показали. Тобто, можна зробити висновок про доцільність обраної архітектури, за допомогою якої, при використанні вузлів-агрегаторів, досягнуто додаткову оптимізацію часу.

Використання конвеєру *CI/CD* на основі запропонованого алгоритму розгортання системи дозволило прискорити час розгортання системи після відмов, пов'язаних з програмними помилками, на 14.52%. В практичному полі, це означатиме можливість підтримувати високий *uptime* системи, при тестуванні нових моделей, що викликатиме додатковий ризик відмови. Також, результати моделювання підтвердили можливість підтримки 99% *uptime* в довгостроковому тестуванні, при постійному потоці відмов, з середньою частотою раз на 10 годин.

Тестування навантаження системи показало, що система здатна обробляти високу кількість запитів в реальному часі. В ході тестування навантаження було використано до 25 потоків запитів на систему. Запропонований метод дозволив прискорити відповідь компонент платформи на 2.31% та зменшити використання ЦП на 7.23%.

Узагальнюючи, експерименти підтвердили правильність обраних програмних рішень та показав переваги запропонованого методу розгортання компонент платформи вбудованих систем, зокрема заощадження часу розгортання компонент даної платформи за допомогою адаптованого алгоритму зборки контейнеру та розгортання системи.

ВИСНОВКИ

В дисертаційній роботі було досліджено процес розгортання компонент платформи вбудованих систем із використанням вбудованих пристроїв. Виходячи з аналізу сучасних літературних джерел та інженерних аналогічних рішень для розгортання компонент платформ вбудованих систем, можна зробити висновок про необхідність пошуку підходів розгортання таких систем, які матимуть можливість швидко адаптуватись під зміни в середовищі, як при зміні ефективності моделі ШІ, з урахуванням обмежень, притаманних використанню вбудованих пристроїв. Висновками проведеного літературного аналізу є доречність використання процесу *MLOps*, практики якого широко вводиться в експлуатацію в платформах вбудованих систем і показали високу ефективність у використанні, особливо в умовах плинності середовища, та необхідність в подальшому пошуку ефективних рішень, щодо впровадження подібних практик у вбудованих пристроях.

Досліджено практики *MLOps* та фактори впливу на використання тих чи інших підходів в рамках підзадач розгортання системи. Було запропоновано спосіб розгортання системи із застосуванням гібридного підходу, для *Cloud-Edge* системи, яка враховує сильні та слабкі сторони хмарних рішень та вбудованих пристроїв. В рамках даного способу розгортання, запропоновано алгоритм конвеєру *CI/CD*, який використовує можливості різних компонентів архітектури системи. Запропоновано низку метрик, які мають на меті різнобічний аналіз способу розгортання системи з метою виявлення основних факторів впливу на часові характеристики системи.

Підвищено ефективність розгортання компонент платформ вбудованих систем, який базується на конвеєрній збірці контейнеру та поетапному прунінгу шарів образу контейнера. Метод конвеєрної зборки використовує можливості апаратного забезпечення системи для багатопоточної обробки шарів зображення.

Запропоновано алгоритм прунінгу контейнеру за допомогою очистки невикористаних залежностей контейнеру та математично обґрунтовано спосіб обробки таких залежностей. Запропоновано підхід з використанням інфраструктурного коду для автоматизації розгортання архітектури системи, тим самим заощаджуючи час виконання *CI/CD* конвеєру.

На основі представлених підходів та методів, розроблено програмну реалізацію методу розгортання платформ вбудованих систем. Обґрунтовано вибір архітектурного рішення з використанням додаткових зв'язків між вбудованими пристроями в системі, моделі штучної нейронної мережі *MobileNet* та програмного стеку для реалізації та перевірки запропонованих методів. Тестування способу розгортання платформи вбудованих систем виявило переваги запропонованого підходу в порівнянні із вже існуючими підходами. Виходячи з результатів моделювання, отримані значення метрик показують прискорення розгортання компоненти ІІІ платформи вбудованих систем на 5.79%, в порівнянні з існуючими рішеннями. Прискорено час зборки контейнеру до 16.24%, порівняно з існуючими рішеннями, при зменшенні використання пам'яті на 11.15%. Серед недоліків розглянутого підходу можна виокремити потенційний програш загального часу розгортання системи, за умови коли для певних залежностей виникають збої в працездатності компонентів платформи. Іншим недоліком системи є більш низька відмовостійкість компонентів платформи на основі вбудованих систем, порівняно з компонентами або іншими програмно-апаратними системами. Також, об'єднання пристроїв та компонент в топологію потягне за собою додаткові кошти на реалізацію та підтримку.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. John, M. M., Olsson, H. H., & Bosch, J. (2021). Towards MLOps: A framework and maturity model. In 2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA) (pp. 334–341). IEEE. <https://doi.org/10.1109/SEAA53835.2021.00050>
2. Kazmierczak, J., Salama, K., & Huerta, V. (2023, May 18). MLOps: Continuous delivery and automation pipelines in machine learning. Google Cloud. <https://cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning>
3. Treveil, M., Omont, N., Stenac, C., Lefevre, K., Phan, D., Zentici, J., Lavoillotte, A., Miyazaki, M., & Heidmann, L. (2020). Introducing MLOps: How to scale machine learning in the enterprise. O'Reilly Media.
4. Mäkinen, S., Skogström, H., Laaksonen, E., & Mikkonen, T. (2021). Who needs MLOps: What data scientists seek to accomplish and how can MLOps help? In 2021 IEEE/ACM 1st Workshop on AI Engineering - Software Engineering for AI (WAIN) (pp. 109–112). IEEE. <https://doi.org/10.1109/wain52551.2021.00024>
5. Zhao, Y. (2021). MLOps scaling ML in an industrial setting [Master's thesis, Vrije Universiteit Amsterdam & Universiteit van Amsterdam]. https://staff.fnwi.uva.nl/a.s.z.belloum/MSctheses/MScthesis_Yizhen_Zhao.pdf
6. Amazon Web Services. (n.d.). Amazon Web Services (AWS) - Cloud computing services. <https://aws.amazon.com>
7. Microsoft. (n.d.). Cloud computing services | Microsoft Azure. <https://azure.microsoft.com>
8. Erich, F. M. A., Amrit, C., & Daneva, M. (2017). A qualitative study of DevOps usage in practice. *Journal of Software: Evolution and Process*, 29(6), Article e1885. <https://doi.org/10.1002/smr.1885>
9. Loukides, M. (2012). What is DevOps? O'Reilly Media.

10. Luz, W. P., Pinto, G., & Bonifácio, R. (2019). Adopting DevOps in the real world: A theory, a model, and a case study. *Journal of Systems and Software*, 157, Article 110384. <https://doi.org/10.1016/j.jss.2019.07.083>
11. Lwakatare, L. E., Crnkovic, I., & Bosch, J. (2020). DevOps for AI – Challenges in development of AI-enabled applications. In 2020 International Conference on Software and System Process (ICSSP) (pp. 1–10). IEEE. <https://doi.org/10.23919/SoftCOM50211.2020.9238323>
12. Jha, P., & Khan, R. (2018). A review paper on DevOps: Beginning and more to know. *International Journal of Computer Applications*, 180(48), 16–20. <https://doi.org/10.5120/ijca2018917253>
13. Kreuzberger, D., Kühl, N., & Hirschl, S. (2023). Machine learning operations (MLOps): Overview, definition, and architecture. *IEEE Access*, 11, 31866–31879. <https://doi.org/10.1109/access.2023.3262138>
14. Petrakieva, S., Garasym, O., & Taralova, I. (2014). G-interval method for secure chaotic synchronization and information transmission. In 2014 9th International Conference on Information Technology Security Transactions (ICITST) (pp. 1–6). IEEE. <https://doi.org/10.1109/ICITST.2014.7038771>
15. Cois, C. A., Yankel, J., & Connell, A. (2014). Modern DevOps: Optimizing software development through effective system interactions. In 2014 IEEE International Professional Communication Conference (IPCC) (pp. 1–7). IEEE. <https://doi.org/10.1109/IPCC.2014.7020388>
16. Crawford, K., & Joler, V. (2018). Anatomy of an AI system: The Amazon Echo as an anatomical map of human labor, data and planetary resources. <https://anatomyof.ai>
17. Zhou, Y., Yu, Y., & Ding, B. (2020). Towards MLOps: A case study of ML pipeline platform. In 2020 International Conference on Artificial Intelligence and Computer Engineering (ICAICE) (pp. 494–500). IEEE. <https://doi.org/10.1109/ICAICE51518.2020.00102>

18. Salvucci, E. (2021). MLOps - Standardizing the machine learning workflow [Master's thesis, University of Bologna]. https://amslaurea.unibo.it/id/eprint/23645/1/tesi_enrico_salvucci.pdf
19. Mäkinen, S., Skogström, H., Laaksonen, E., & Mikkonen, T. (2021). Who needs MLOps: What data scientists seek to accomplish and how can MLOps help? In 2021 IEEE/ACM 1st Workshop on AI Engineering - Software Engineering for AI (WAIN) (pp. 109–112). IEEE. <https://doi.org/10.1109/wain52551.2021.00024>
20. Testi, M., Ballabio, M., Frontoni, E., Iannello, G., Moccia, S., Soda, P., & Vessio, G. (2022). MLOps: A taxonomy and a methodology. *IEEE Access*, 10, 63606–63618. <https://doi.org/10.1109/access.2022.3181730>
21. Garg, S., Pundir, P., Rathee, G., Gupta, P. K., Garg, S., & Ahlawat, S. (2021). On continuous integration / continuous delivery for automated deployment of machine learning models using MLOps. In 2021 IEEE 4th International Conference on Artificial Intelligence and Knowledge Engineering (AIKE) (pp. 25–28). IEEE. <https://doi.org/10.1109/AIKE52691.2021.00010>
22. Subramanya, R., Sierla, S., & Vyatkin, V. (2022). From DevOps to MLOps: Overview and application to electricity market forecasting. *Applied Sciences*, 12(19), Article 9851. <https://doi.org/10.3390/app12199851>
23. Ashmore, R., Calinescu, R., & Paterson, C. (2019). Assuring the machine learning lifecycle: Desiderata, methods, and challenges. arXiv. <https://doi.org/10.48550/arXiv.1905.04223>
24. Ruf, P., Madan, M., Reich, C., & Ould-Abdeslam, D. (2021). Demystifying MLOps and presenting a recipe for the selection of open-source tools. *Applied Sciences*, 11(19), Article 8861. <https://doi.org/10.3390/app11198861>
25. Batini, C., Cappiello, C., Francalanci, C., & Maurino, A. (2009). Methodologies for data quality assessment and improvement. *ACM Computing Surveys*, 41(3), Article 16. <https://doi.org/10.1145/1541880.1541883>

26. Peng, G., Lacagnina, C., Ivánová, I., Downs, R. R., Ramapriyan, H., Ganske, A., & Le Roux, J. (2022). Global community guidelines for documenting, sharing, and reusing quality information of individual Earth science datasets. *Data Science Journal*, 21(1), Article 8. <https://doi.org/10.5334/dsj-2022-008>
27. Treveil, M., Omont, N., Stenac, C., Lefevre, K., Phan, D., Zentici, J., Lavoillotte, A., Miyazaki, M., & Heidmann, L. (2020). Introducing MLOps: How to scale machine learning in the enterprise. O'Reilly Media.
28. Verheul, I., Imming, M., Ringersma, J., Mordant, A., van der Hoeven, J., & Pronk, M. (2019). Data stewardship on the map: A study of tasks and roles in Dutch research institutes [Report]. Zenodo. <https://doi.org/10.5281/zenodo.2669150>
29. Mons, B. (2018). *Data stewardship for open science: Implementing FAIR principles*. CRC Press.
30. Ying, X. (2019). An overview of overfitting and its solutions. *Journal of Physics: Conference Series*, 1168(2), Article 022022. <https://doi.org/10.1088/1742-6596/1168/2/022022>
31. Liu, Y., Ling, Z., Huo, B., Wang, B., Chen, T., & Mouine, E. (2020). Building a platform for machine learning operations from open source frameworks. *IFAC-PapersOnLine*, 53(5), 704–709. <https://doi.org/10.1016/j.ifacol.2021.04.161>
32. Parihar, A. S., Gupta, U., Srivastava, U., Yadav, V., & Trivedi, V. K. (2023). Automated machine learning deployment using open-source CI/CD tool. In A. Khanna, Z. Polkowski, & O. Castillo (Eds.), *Proceedings of data analytics and management* (Vol. 572, pp. 209–222). Springer. https://doi.org/10.1007/978-981-19-7615-5_19
33. Pachouly, J., Ahirrao, S., Kotecha, K., Selvachandran, G., & Abraham, A. (2022). A systematic literature review on software defect prediction using artificial intelligence: Datasets, data validation methods, approaches, and tools. *Engineering Applications of Artificial Intelligence*, 111, Article 104773. <https://doi.org/10.1016/j.engappai.2022.104773>

34. BrowserStack. (n.d.). DevOps lifecycle: Different phases in DevOps.
<https://www.browserstack.com/guide/devops-lifecycle>
35. Chu, C., & Chen, G. (2022, November 15). Composite availability: Calculating the overall availability of cloud infrastructure. Google Cloud Blog. Google Cloud.
<https://cloud.google.com/blog/products/devops-sre/composite-cloud-availability>
36. Volokyta, A., Loutskii, H., Rehida, P., Kaplunov, A., Ivanishchev, B., Honcharenko, O., & Korenko, D. (2022). Extended DragonDeBruijn topology synthesis method. *International Journal of Computer Network and Information Security*, 14(6), 23–36. <https://doi.org/10.5815/ijcnis.2022.06.03>
37. Loutskii, H., Volokyta, A., Rehida, P., et al. (2021). Topology synthesis method based on excess De Bruijn and Dragonfly. In Z. Hu, S. Petoukhov, I. Dychka, & M. He (Eds.), *Advances in computer science for engineering and education IV* (pp. 315–325). Springer. https://doi.org/10.1007/978-3-030-80472-5_27
38. Jain, N., Bhatele, A., Howell, L. H., Böhme, D., Karlin, I., León, E. A., Mubarak, M., Wolfe, N., Gamblin, T., & Leininger, M. L. (2017). Predicting the performance impact of different fat-tree configurations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (pp. 1–13). Association for Computing Machinery.
<https://doi.org/10.1145/3126908.3126967>
39. Bossard, A. (2023). Torus-connected toroids: An efficient topology for interconnection networks. *Computers*, 12(9), Article 173.
<https://doi.org/10.3390/computers12090173>
40. Dragon, P. B., Hernandez, O. I., Sawada, J., Williams, A., & Wong, D. (2018). Constructing de Bruijn sequences with co-lexicographic order: The k-ary Grandmama sequence. *European Journal of Combinatorics*, 72, 1–11.
<https://doi.org/10.1016/j.ejc.2018.03.006>

41. Loutskii, H., Volokyta, A., Rehida, P., Kaplunov, A., Ivanishchev, B., Honcharenko, O., & Korenko, D. (2021). Topology synthesis method based on excess de Bruijn and Dragonfly. In Z. Hu, S. Petoukhov, I. Dychka, & M. He (Eds.), *Advances in computer science for engineering and education IV* (pp. 315–325). Springer. https://doi.org/10.1007/978-3-030-80472-5_27
42. Hernández, A., Amat Bertran, M., Bongiovanni, J., & Vila, M. M. (2020). A new edge architecture for AI-IoT services deployment. *Procedia Computer Science*, 175, 10–19. <https://doi.org/10.1016/j.procs.2020.07.006>
43. Raj, E., Buffoni, D., Westerlund, M., & Ahola, K. (2021). Edge MLOps: An automation framework for AIoT applications. In 2021 IEEE International Conference on Cloud Engineering (IC2E) (pp. 195–200). IEEE. <https://doi.org/10.1109/IC2E52221.2021.00034>
44. NVIDIA. (n.d.). Buy the latest Jetson products. NVIDIA Developer. <https://developer.nvidia.com/buy-jetson>
45. Raspberry Pi Foundation. (n.d.). Teach, learn, and make with Raspberry Pi. Raspberry Pi Foundation. <https://www.raspberrypi.org/>
46. Dawson, R. (2020, December). Applying the MLOps lifecycle. Towards Data Science. <https://towardsdatascience.com/applying-the-mlops-lifecycle-3b60033b7cbf>
47. John, M. M., Olsson, H. H., & Bosch, J. (2021). Towards MLOps: A framework and maturity model. In 2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA) (pp. 334–341). IEEE. <https://doi.org/10.1109/SEAA53835.2021.00050>
48. Lima, A., Monteiro, L., & Furtado, A. P. (2022). MLOps: Practices, maturity models, roles, tools, and challenges – A systematic literature review. In *Proceedings of the 24th International Conference on Enterprise Information Systems* (Vol. 1, pp. 308–320). SciTePress. <https://doi.org/10.5220/0010997300003179>

49. Kazmierczak, J., Salama, K., & Huerta, V. (2023, May 18). MLOps: Continuous delivery and automation pipelines in machine learning. Google Cloud. <https://cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning>
50. Cifuentes, I. Á. (2023). Design and development of an MLOps framework [Master's thesis, Universitat Politècnica de Catalunya]. UPCommons. <https://upcommons.upc.edu/handle/2117/395348>
51. Hernández, A., Xiao, B., & Tudor, V. (2020). ERAIA - Enabling intelligence data pipelines for IoT-based application systems. In 2020 IEEE International Conference on Pervasive Computing and Communications (PerCom) (pp. 1–10). IEEE. <https://doi.org/10.1109/percom45495.2020.9127385>
52. Leroux, S., Simoens, P., Lootus, M., Thakore, K., & Sharma, A. (2022). TinyMLOps: Operational challenges for widespread edge AI adoption. In 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) (pp. 1003–1010). IEEE. <https://doi.org/10.1109/IPDPSW55747.2022.00160>
53. Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A., & Bastien, J. (2017). Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (pp. 185–200). Association for Computing Machinery. <https://doi.org/10.1145/3062341.3062363>
54. Yoon, G., Kim, G.-Y., Yoo, H., Kim, S. C., & Kim, R. (2021). Implementing practical DNN-based object detection offloading decision for maximizing detection performance of mobile edge devices. *IEEE Access*, 9, 140199–140211. <https://doi.org/10.1109/access.2021.3118731>
55. NVIDIA. (n.d.). NVIDIA Jetson Xavier NX for embedded & edge systems. <https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-nx/>
56. Kubeflow. (n.d.). Kubeflow. <https://www.kubeflow.org>

57. Gill, S. S. (2022). AI for next generation computing: Emerging trends and future directions. *Internet of Things*, 19, Article 100514. <https://doi.org/10.1016/j.iot.2022.100514>
58. Leff, D., & Lim, K. T. K. (2023). The key to leveraging AI at scale. In B. Vinod (Ed.), *Artificial intelligence and machine learning in the travel industry* (pp. 171–175). Springer. https://doi.org/10.1007/978-3-031-25456-7_14
59. Goncharenko, O., Rehida, P., Volokyta, A., Loutskii, H., & Thinh, V. D. (2020). Routing method based on the excess code for fault tolerant clusters with InfiniBand. In Z. Hu, S. Petoukhov, I. Dychka, & M. He (Eds.), *Advances in computer science for engineering and education II* (pp. 335–345). Springer. https://doi.org/10.1007/978-3-030-16621-2_31
60. Русінов, В., Іваніщев, Б., Антонюк, А., Волокита, А., & Луцький, Г. (2020). Способи синтезу топологічних організацій на основі кодових перетворень Дебруйна. *Технічні науки та технології*, 4(22), 131–143. [https://doi.org/10.25140/2411-5363-2020-4\(22\)-131-143](https://doi.org/10.25140/2411-5363-2020-4(22)-131-143)
61. Yang, C., Lan, S., Wang, L., Shen, W., & Huang, G. Q. (2020). Big data driven edge-cloud collaboration architecture for cloud manufacturing: A software defined perspective. *IEEE Access*, 8, 45938–45950. <https://doi.org/10.1109/access.2020.2977846>
62. A, M., Dinkar, A., Mouli, S. C., B, S., & Deshpande, A. A. (2021). Comparison of containerization and virtualization in cloud architectures. In 2021 6th International Conference on Communication and Electronics Systems (ICCES) (pp. 1–5). IEEE. <https://doi.org/10.1109/CONECCT52877.2021.9622668>
63. Hampau, R. M., Kaptein, M., van Emden, R., Rost, T., & Malavolta, I. (2022). An empirical study on the performance and energy consumption of AI containerization strategies for computer-vision tasks on the edge. In *Proceedings of the 26th International Conference on Evaluation and Assessment in Software Engineering*

- (pp. 50–59). Association for Computing Machinery.
<https://doi.org/10.1145/3530019.3530025>
64. Git. (n.d.). Git. <https://git-scm.com>
 65. Assunção, W. K. G., Krüger, J., Mosser, S., & Selaoui, S. (2023). How do microservices evolve? An empirical analysis of changes in open-source microservice repositories. *Journal of Systems and Software*, 204, Article 111788. <https://doi.org/10.1016/j.jss.2023.111788>
 66. Casalicchio, E. (2019). Container orchestration: A survey. In A. Puliafito & K. Trivedi (Eds.), *Systems modeling: Methodologies and tools* (pp. 221–235). Springer. https://doi.org/10.1007/978-3-319-92378-9_14
 67. Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., & Adam, H. (2017, April 17). MobileNets: Efficient convolutional neural networks for mobile vision applications. arXiv. <https://doi.org/10.48550/arXiv.1704.04861>
 68. Adi, K., Santoso, H., & Indrajit, E. (2021). Comparison of convolutional neural networks model using different optimizers for image classification. *International Journal of Sciences: Basic and Applied Research*, 60(2), 116–126.
 69. ML-Ops.org. (n.d.). Machine learning operations. <https://ml-ops.org>
 70. Li, L., Fan, Y., Tse, M., & Lin, K.-Y. (2020). A review of applications in federated learning. *Computers & Industrial Engineering*, 149, Article 106854. <https://doi.org/10.1016/j.cie.2020.106854>
 71. Forsgren, N., Tremblay, M. C., VanderMeer, D., & Humble, J. (2017). DORA platform: DevOps assessment and benchmarking. In A. Maedche, J. vom Brocke, & A. Hevner (Eds.), *Designing the digital transformation: DESRIST 2017* (pp. 436–440). Springer. https://doi.org/10.1007/978-3-319-59144-5_27
 72. Anwar, A., Mohamed, M., Tarasov, V., Little, M., Rupprecht, L., Cheng, Y., & Butt, A. R. (2018). Improving Docker registry design based on production

workload analysis. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies* (FAST 18) (pp. 265–278). USENIX Association.

73. Nathan, S., Ghosh, R., Mukherjee, T., & Narayanan, K. (2017). CoMICON: A co-operative management system for Docker container images. In 2017 IEEE International Conference on Cloud Engineering (IC2E) (pp. 116–126). IEEE. <https://doi.org/10.1109/ic2e.2017.24>

74. Ochagavía, A. (n.d.). The magic of dependency resolution. ochagavia.nl. <https://ochagavia.nl/blog/the-magic-of-dependency-resolution/>

75. Harter, T., Salmon, B., Liu, R., Arpaci-Dusseau, A. C., & Arpaci-Dusseau, R. H. (2016). Slacker: Fast distribution with lazy Docker containers. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies* (FAST 16) (pp. 181–195). USENIX Association.

76. Heule, M. J. H., Kullmann, O., & Biere, A. (2018). Cube-and-conquer for satisfiability. In Y. Hamadi & L. Sais (Eds.), *Handbook of parallel constraint reasoning* (pp. 31–59). Springer. https://doi.org/10.1007/978-3-319-63516-3_2

77. Docker Library. (n.d.). Python 3.12. GitHub. <https://github.com/docker-library/python/tree/b968d488efc09fd34672fc6238182e1222ae005e/3.12>

78. Docker. (n.d.). Python. Docker Hub. https://hub.docker.com/_/python

79. Kane, S. (n.d.). Basic-registry: A basic, authenticated v2 Docker Registry [Computer software]. GitHub. <https://github.com/spkane/basic-registry>

80. Zeng, Y., Chen, J., Shang, W., & Chen, T.-H. (P.). (2019). Studying the characteristics of logging practices in mobile apps: A case study on F-Droid. *Empirical Software Engineering*, 24(6), 3394–3434. <https://doi.org/10.1007/s10664-019-09687-9>

81. Konar, A. (2005). *Cognitive engineering: A distributed approach to machine intelligence*. Springer.

82. Chen, C.-H., Trappey, A., Peruzzini, M., Stjepandić, J., & Wognum, N. (Eds.). (2017). *Transdisciplinary engineering: A paradigm shift: Proceedings of the*

24th ISPE Inc. International Conference on Transdisciplinary Engineering, July 10-14, 2017. IOS Press.

83. Palagin, A., Kryvyi, S., & Petrenko, N. (2012). *Ontological methods and means of processing subject knowledge*. VNU.
84. Globa, L., Novogrudska, R., Koval, A., & Senchenko, V. (2018). Ontology for application development. In C. Thomas (Ed.), *Ontology in information science*. IntechOpen. <https://doi.org/10.5772/intechopen.74042>
85. Bondy, J. A., & Murty, U. S. R. (1976). *Graph theory with applications*. North-Holland.
86. Lamberts, K., & Shanks, D. R. (Eds.). (1997). *Knowledge, concepts and categories*. Psychology Press.
87. Engeler, E. (1984). [Review of the book *The lambda calculus. Its syntax and semantics*, by H. P. Barendregt]. *Journal of Symbolic Logic*, 49(1), 301–303. <https://doi.org/10.2307/2274112>
88. Kleene, S. C. (1952). *Introduction to metamathematics*. P. Noordhoff N.V.
89. Stjepandić, J., Wognum, N., Peruzzini, M., Chen, C.-H., & Trappey, A. (Eds.). (2017). *Transdisciplinary engineering: A paradigm shift: Proceedings of the 24th ISPE Inc. International Conference on Transdisciplinary Engineering, July 10-14, 2017*. IOS Press.
90. Honchar, A. V., Stryzhak, O. Ye., Berkman, L. N., & Zakharzhevskyi, A. H. (2021). Transdisciplinary consolidation of information environments. *Connectivity*, 149(1). <https://doi.org/10.31673/2412-9070.2021.010310>
91. Dovgyi, S., & Stryzhak, O. (2021). Transdisciplinary fundamentals of information-analytical activity. In M. Ilchenko, L. Uryvsky, & L. Globa (Eds.), *Advances in information and communication technology and systems* (Vol. 152, pp. 99–126). Springer. https://doi.org/10.1007/978-3-030-58359-0_7

ДОДАТОК А

Код реалізації програмного застосунку та інфраструктурні скрипти для розгортання системи

main.py

```
from flask import Flask
from api.endpoints import endpoints_bp

app = Flask(__name__)

if __name__ == '__main__':
    app.register_blueprint(endpoints_bp)
    app.run(debug=True, host="0.0.0.0")
```

test.py

```
from unittest import TestCase

from src.main import classify_image

class TestDNN(TestCase):

    def test_recognize_dog(self):
        predictions = classify_image("./example_image.jpg")
        self.assertEqual(predictions[0][1], "Pembroke")

    def test_recognize_cat(self):
        predictions = classify_image("./example_cat_image.jpg")
        self.assertEqual(predictions[0][1], "Siamese")
```

```

def test_empty_image(self):
    predictions = classify_image("./empty_image.jpg")
    self.assertEqual(predictions, [])

def test_multiple_predictions(self):
    predictions = classify_image("./example_multiple_objects.jpg")
    # Assuming the most likely prediction should be checked
    self.assertGreater(len(predictions), 1)
    self.assertIn(predictions[0][1], ["Dog", "Cat", "Bird", "Horse"])

def test_high_confidence(self):
    predictions = classify_image("./example_high_confidence_image.jpg")
    self.assertGreater(predictions[0][0], 0.9) # confidence > 90%
    self.assertEqual(predictions[0][1], "Dog")

def test_low_confidence(self):
    predictions = classify_image("./example_low_confidence_image.jpg")
    self.assertLess(predictions[0][0], 0.3) # confidence < 30%
    self.assertIn(predictions[0][1], ["Dog", "Cat", "Bird", "Horse"])

```

setup.py

```

from setuptools import setup, find_packages

setup(name="MLOps-test-project",
      version='0.0.1',
      packages=find_packages('src'),
      package_dir={"": 'src'})

```

preprocessing.py

```

import numpy as np
from PIL import Image, ImageDraw, ImageFont
from tensorflow.keras.applications.mobilenet import preprocess_input,
decode_predictions
from tensorflow.keras.preprocessing import image
from tensorflow.keras.models import load_model
import matplotlib.pyplot as plt

```

Load and preprocess the image for classification by path

```

def preprocess_image_path(image_path):
    img = image.load_img(image_path, target_size=(224, 224))
    img_array = image.img_to_array(img)
    img_array = np.expand_dims(img_array, axis=0)
    img_array = preprocess_input(img_array)
    return img_array

```

Load and preprocess the image for classification

```

def preprocess_image(img, target_size=(224, 224)):
    img = img.resize(target_size)
    img_array = image.img_to_array(img)
    img_array = np.expand_dims(img_array, axis=0)
    img_array = preprocess_input(img_array)
    return img_array

```

Perform classification on the input image by path

```

def classify_image_path(image_path, model):
    img_array = preprocess_image_path(image_path)

```

```
preds = model.predict(img_array)
predictions = decode_predictions(preds, top=3)[0]
return predictions
```

Perform classification on the input image

```
def classify_image(img, model):
    img_array = preprocess_image(img)
    preds = model.predict(img_array)
    predictions = decode_predictions(preds, top=3)[0]
    return predictions
```

Load a model from a file

```
def load_model_from_file(model_path):
    model = load_model(model_path)
    return model
```

Save predictions to a file

```
def save_predictions(predictions, output_file):
    with open(output_file, 'w') as file:
        for pred in predictions:
            file.write(f'{pred[1]}: {pred[2]:.2f}\n')
```

Visualize predictions on the image

```
def visualize_predictions(image_path, predictions, output_image_path):
    img = Image.open(image_path)
    draw = ImageDraw.Draw(img)
```

```

font = ImageFont.load_default()

for i, (class_id, class_name, class_confidence) in enumerate(predictions):
    text = f'{class_name}: {class_confidence:.2f}'
    draw.text((10, 10 + i * 20), text, font=font, fill=(255, 0, 0))

img.save(output_image_path)

# Display image with predictions using matplotlib
def display_image_with_predictions(image_path, predictions):
    img = Image.open(image_path)
    plt.imshow(img)
    plt.axis('off')

    plt.title('Predictions:')
    for i, (class_id, class_name, class_confidence) in enumerate(predictions):
        plt.text(0, -20*(i+1), f'{class_name}: {class_confidence:.2f}', color='red')

    plt.show()

# Utility function to load an image from a file
def load_image_from_file(image_path):
    img = Image.open(image_path)
    return img

```

endpoints.py

```

import logging
import werkzeug.exceptions
from flask import request, jsonify, Blueprint
from src.preprocessing import preprocess_image_path, preprocess_image,
classify_image
from src.utils import get_results_from_tf
from models.resnet import model

# Create a Blueprint for endpoints
endpoints_bp = Blueprint('endpoints_bp', __name__)

@endpoints_bp.route('/classify', methods=['POST'])
def classify_image_endpoint():
    try:
        data = request.files["image"]
        data.save("img.png")

        # Choose preprocessing based on model input shape
        if model.input_shape in [(None, None, None, 3), (None, 224, 224, 3)]:
            image_array = preprocess_image_path('img.png')
        elif model.input_shape == (None, 384, 384, 3):
            image_array = preprocess_image_path('img.png', bigger_size=True)
        else:
            raise ValueError(f"Incorrect model input size. Correct: {model.input_shape}")

        predictions = classify_image(image_array, model)
        predictions = get_results_from_tf(predictions)

        return jsonify(predictions)

```

```

except werkzeug.exceptions.BadRequestKeyError:
    logging.error("A bad request has been sent. No image was provided.")
    return jsonify({"error": "No image was provided."}), 400
except Exception as e:
    logging.error(f'An error occurred: {str(e)}')
    return jsonify({"error": str(e)}), 500

```

```
@endpoints_bp.route('/status', methods=['GET'])
```

```
def status():
```

```

    result = {'status': 'Model is up and running'}
    return jsonify(result)

```

```
@endpoints_bp.route('/predict', methods=['POST'])
```

```
def predict():
```

```

    try:
        data = request.files["image"]
        image_path = "uploaded_image.png"
        data.save(image_path)

        # Determine appropriate preprocessing
        if model.input_shape in [(None, None, None, 3), (None, 224, 224, 3)]:
            image_array = preprocess_image_path(image_path)
        elif model.input_shape == (None, 384, 384, 3):
            image_array = preprocess_image_path(image_path, bigger_size=True)
        else:
            raise ValueError(f'Incorrect model input size. Correct: {model.input_shape}')

        predictions = classify_image(image_array, model)

```

```

predictions = get_results_from_tf(predictions)

return jsonify(predictions)
except werkzeug.exceptions.BadRequestKeyError:
    logging.error("A bad request has been sent. No image was provided.")
    return jsonify({"error": "No image was provided."}), 400
except Exception as e:
    logging.error(f'An error occurred: {str(e)}')
    return jsonify({"error": str(e)}), 500

@endpoints_bp.route('/model_info', methods=['GET'])
def model_info():
    try:
        input_shape = model.input_shape
        model_summary = model.summary(print_fn=lambda x: x) # Capture model
summary

        # Return model input shape and summary as a dictionary
        result = {
            'input_shape': str(input_shape),
            'model_summary': model_summary
        }

        return jsonify(result)
    except Exception as e:
        logging.error(f'An error occurred while fetching model info: {str(e)}')
        return jsonify({"error": str(e)}), 500

```

```

@endpoints_bp.route('/health_check', methods=['GET'])
def health_check():
    try:
        # Basic health check endpoint
        result = {'status': 'Healthy'}
        return jsonify(result)
    except Exception as e:
        logging.error(f'An error occurred during health check: {str(e)}')
        return jsonify({"error": str(e)}), 500

@endpoints_bp.route('/resize_image', methods=['POST'])
def resize_image_endpoint():
    try:
        data = request.files["image"]
        target_size = request.args.get('size', '224x224')
        width, height = map(int, target_size.split('x'))

        img = Image.open(data)
        img = img.resize((width, height))
        resized_image_path = "resized_image.png"
        img.save(resized_image_path)

        return jsonify({"message": "Image resized successfully.", "resized_image_path":
resized_image_path})
    except werkzeug.exceptions.BadRequestKeyError:
        logging.error("A bad request has been sent. No image was provided.")
        return jsonify({"error": "No image was provided."}), 400
    except Exception as e:

```

```
logging.error(f'An error occurred: {str(e)}')
return jsonify({"error": str(e)}), 500
```

gh-actions-tests.yml

```
name: ${{ github.actor }}
```

```
on:
```

```
  push:
```

```
    branches:
```

```
      - main
```

```
  pull_request:
```

```
    branches:
```

```
      - main
```

```
jobs:
```

```
  test:
```

```
    name: Run Tests and Check Code Quality
```

```
    runs-on: ubuntu-latest
```

```
    steps:
```

```
      - name: Checkout code
```

```
        uses: actions/checkout@v2
```

```
      - name: Set up Python
```

```
        uses: actions/setup-python@v5
```

```
        with:
```

```
          python-version: '3.10'
```

```
      - name: Update pip
```

run: pip install --upgrade pip

- name: Install dependencies

run: pip install -r requirements.txt

- name: Install additional tools

run: |

pip install pytest

pip install pylint coverage

- name: Run unit tests

run: |

pytest tests/test_dnn.py

- name: Run additional tests

run: |

pytest tests/test_utils.py

pytest tests/test_endpoints.py

- name: Check code quality with pylint

run: |

pylint src/ tests/

- name: Measure test coverage

run: |

coverage run -m pytest

coverage report

coverage html

continue-on-error: true # Optional: to allow the workflow to continue even if
coverage fails

```
- name: Upload test coverage report
  uses: actions/upload-artifact@v3
  with:
    name: coverage-report
    path: htmlcov/
```

Dockerfile

```
# Can possibly be used with other versions
FROM python:3.10.14-bookworm
```

```
WORKDIR /proj
```

```
COPY pyproject.toml poetry.lock ./
```

```
# using poetry
```

```
RUN pip config --global set global.timeout 150 \
```

```
    && pip install poetry \
```

```
    && poetry config virtualenvs.in-project true \
```

```
    && poetry install --no-root --no-dev \
```

```
    && poetry install --no-dev # Install dependencies
```

```
COPY . .
```

```
# Prod env setup
```

```
ENV FLASK_APP=main.py
```

```
ENV FLASK_ENV=production
```

```
# Change according to the role (?)
```

```
EXPOSE 5000
```

```
CMD ["gunicorn", "--bind", "0.0.0.0:5000", "main:app"]
```

```
# Healthcheck to ensure the application is running
```

```
HEALTHCHECK --interval=30s --timeout=10s --retries=3 \
```

```
    CMD /usr/local/bin/healthcheck.sh || exit 1
```

healthcheck.sh

```
#!/bin/bash
```

```
HEALTH_CHECK_URL="http://localhost:5000/health_check"
```

```
FAILURE_REPORT_URL="http://localhost:5000/report_failure"
```

```
# perform the health check
```

```
if curl --silent --fail "$HEALTH_CHECK_URL" > /dev/null; then
```

```
    exit 0
```

```
else
```

```
    # Health check failed - report the failure
```

```
    curl --silent --fail --request POST "$FAILURE_REPORT_URL" \
```

```
        --header "Content-Type: application/json" \
```

```
        --data '{"status":"unhealthy"}' > /dev/null
```

```
    exit 1
```

```
fi
```

pyproject.toml

```
[tool.poetry]
```

```
name = "MLOps dissertation app"
```

```
version = "0.1.0"
```

```
description = ""
```

```
authors = ["Volodymyr Rusinov <email@is.hidden>"]
readme = "README.md"

[tool.poetry.dependencies]
python = "~3.10"
tensorflow = "2.16.1"
numpy = "1.26.4"
absl-py = "2.1.0"
astunparse = "1.6.3"
certifi = "2024.2.2"
charset-normalizer = "3.3.2"
flatbuffers = "24.3.7"
gast = "0.5.4"
google-pasta = "0.2.0"
grpcio = "1.62.1"
h5py = "3.10.0"
idna = "3.6"
keras = "3.1.1"
libclang = "18.1.1"
Markdown = "3.6"
markdown-it-py = "3.0.0"
MarkupSafe = "2.1.5"
mdurl = "0.1.2"
ml-dtypes = "0.3.2"
namex = "0.0.7"
opt-einsum = "3.3.0"
optree = "0.10.0"
packaging = "24.0"
pillow = "10.2.0"
protobuf = "4.25.3"
```

```

Pygments = "2.17.2"
requests = "2.31.0"
rich = "13.7.1"
six = "1.16.0"
tensorboard = "2.16.2"
tensorboard-data-server = "0.7.2"
tensorflow-io-gcs-filesystem = "0.31.0"
termcolor = "2.4.0"
typing_extensions = "4.10.0"
urllib3 = "2.2.1"
Werkzeug = "3.0.1"
wrapt = "1.16.0"

flask = "~3.0.3"
setuptools = "~60.2.0"

```

```

[build-system]
requires = ["poetry-core"]
build-backend = "poetry.core.masonry.api"

```

ec2-create.tf

```

terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "~> 4.16"
    }
  }
}

```

```

    required_version = ">= 1.2.0"
}

provider "aws" {
    region = "eu-north-1"
    profile = "vrusinov"
}

resource "aws_instance" "build_server" {
    availability_zone = "eu-north-1a"
    #ami              = "ami-0914547665e6a707c" # x86
    ami              = "ami-02b7539372433cf6b"
    instance_type    = "c6g.medium"
    key_name         = "ec2-terraform"
    associate_public_ip_address = true

    root_block_device {
        volume_size = 20
        volume_type = "gp3"
    }

    user_data = <<-EOF
    #!/bin/bash -xe
    sudo apt-get update
    sudo apt-get install ca-certificates curl
    sudo install -m 0755 -d /etc/apt/keyrings
    sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o
/etc/apt/keyrings/docker.asc
    sudo chmod a+r /etc/apt/keyrings/docker.asc

```

```

# Add the repository to Apt sources:

echo \
    "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc]
https://download.docker.com/linux/ubuntu \
    $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
    sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update

sudo apt-get install -y docker-ce docker-ce-cli containerd.io docker-buildx-plugin
docker-compose-plugin
EOF

tags = {
    Name = "Build server"
}

vpc_security_group_ids = [aws_security_group.allow_ssh.id]
}

resource "aws_ebs_volume" "build_server_volume" {
    availability_zone = "eu-north-1a"
    size              = 20
    type              = "gp3"
    tags = {
        Name = "Build server EBS"
    }
}

resource "aws_volume_attachment" "attach_ebs" {
    device_name = "/dev/sdh"

```

```

    volume_id = aws_ebs_volume.build_server_volume.id
    instance_id = aws_instance.build_server.id
}

```

```

resource "aws_security_group" "allow_ssh" {
    name_prefix = "my-sg-"
    egress {
        from_port = 0
        to_port   = 0
        protocol  = -1
        cidr_blocks = ["0.0.0.0/0"]
        ipv6_cidr_blocks = ["::/0"]
    }
    ingress {
        from_port = 0
        to_port   = 0
        protocol  = -1
        cidr_blocks = ["0.0.0.0/0"]
        ipv6_cidr_blocks = ["::/0"]
    }
    tags = {
        Name = "SSHSecurityGroup"
    }
}

```

performance_test.ps1

```

# number of parallel executions
$numberOfExecutions = 1

```

```
$command = 'newman run -n 40 "postman_collection.json" --reporters json --reporter-  
json-export "result-test-{0}.json"'
```

```
# Array to hold job references
```

```
$jobs = @()
```

```
# Start the jobs
```

```
for ($i = 1; $i -le $numberOfExecutions; $i++) {  
    $jobCommand = [string]::Format($command, $i)  
    $jobs += Start-Job -ScriptBlock {  
        param($cmd)  
        Invoke-Expression $cmd  
    } -ArgumentList $jobCommand  
}
```

```
# wait for jobs
```

```
$jobs | Wait-Job
```

```
# Retrieve and display job results
```

```
foreach ($job in $jobs) {  
    $result = Receive-Job -Job $job  
    Write-Output "Job ID $($job.Id) completed."  
}
```

```
# clean up jobs
```

```
$jobs | Remove-Job
```

performance_test_collection.json

```
{
  "info": {
    "_postman_id": "dbf852d0-e6a3-4742-ba61-f56416b76b58",
    "name": "CI-CD article test pi",
    "schema": "https://schema.getpostman.com/json/collection/v2.1.0/collection.json",
    "_exporter_id": "29101011"
  },
  "item": [
    {
      "name": "Zebra",
      "request": {
        "method": "POST",
        "header": [],
        "body": {
          "mode": "formdata",
          "formdata": [
            {
              "key": "image",
              "type": "file",
              "src": "Giraffe.PNG"
            }
          ]
        }
      },
      "url": {
        "raw": "http://192.168.0.50:5080/classify",
        "protocol": "http",
        "host": [
          "192",
          "168",
          "0"
        ]
      }
    }
  ]
}
```

```

        "50"
    ],
    "port": "5080",
    "path": [
        "classify"
    ]
}
},
"response": []
},
{
    "name": "Pomeranian",
    "request": {
        "method": "POST",
        "header": [],
        "body": {
            "mode": "formdata",
            "formdata": [
                {
                    "key": "image",
                    "type": "file",
                    "src": "Pomeranian.PNG"
                },
                {
                    "key": "some",
                    "value": "ket",
                    "type": "text"
                }
            ]
        }
    },

```

```
"url": {
  "raw": "http://192.168.0.50:5080/classify",
  "protocol": "http",
  "host": [
    "192",
    "168",
    "0",
    "50"
  ],
  "port": "5080",
  "path": [
    "classify"
  ]
},
"response": []
},
{
  "name": "Heartbeat",
  "protocolProfileBehavior": {
    "disableBodyPruning": true
  },
  "request": {
    "method": "GET",
    "header": [],
    "body": {
      "mode": "formdata",
      "formdata": [
        {
          "key": "image",
```

```

        "type": "file",
        "src": "Pomeranian.png",
        "disabled": true
    }
]
},
"url": {
    "raw": "http://192.168.0.50:5080/",
    "protocol": "http",
    "host": [
        "192",
        "168",
        "0",
        "50"
    ],
    "port": "5080",
    "path": [
        ""
    ]
}
},
"response": []
}
]
}

```

deploy-device.yaml

- name: Distribute task and base dependencies

hosts: raspberry_pi

become: yes # sudo may be needed

tasks:

- name: Synchronize local folder with remote device

synchronize:

src: /home/ubuntu/ansible/MLOps/

dest: /home/pi/Desktop/MLOps-app-landing/

mode: push # push files from local to remote

- name: Update apt package index

apt:

update_cache: yes

cache_valid_time: 3600 # Cache valid for 1 hour

- name: Install required packages

apt:

name:

- apt-transport-https

- ca-certificates

- curl

- software-properties-common

state: present

- name: Add Docker's official GPG key

apt_key:

url: <https://download.docker.com/linux/ubuntu/gpg>

state: present

- name: Add Docker APT repository

apt_repository:

```
repo: deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release
-cs) stable
```

```
state: present
```

```
- name: Update apt package index again
```

```
apt:
```

```
update_cache: yes
```

```
- name: Install Docker CE (Community Edition)
```

```
apt:
```

```
name: docker-ce
```

```
state: present
```

```
- name: Start Docker service
```

```
service:
```

```
name: docker
```

```
state: started
```

```
enabled: yes
```

```
- name: Verify Docker installation
```

```
command: docker --version
```

```
register: docker_version
```

```
changed_when: no
```

```
- name: Display Docker version
```

```
debug:
```

```
msg: "Docker version: {{ docker_version.stdout }}"
```

```
- name: Ensure dependencies are installed
```

```
apt:
```

```

name:
  - curl
  - python3
  - python3-pip
state: present
when: ansible_os_family == "Debian"

- name: Ensure dependencies are installed (RedHat)
yum:
  name:
    - curl
    - python3
  state: present
when: ansible_os_family == "RedHat"

- name: Install Poetry using the official installation script
shell: |
  curl -sSL https://install.python-poetry.org | python3 -
args:
  creates: "{{ ansible_env.HOME }}/.local/bin/poetry"

- name: Add Poetry to the PATH
lineinfile:
  path: "{{ ansible_env.HOME }}/.profile"
  line: 'export PATH="$HOME/.local/bin:$PATH"'
  create: yes
  state: present

- name: Ensure Poetry is installed
command: "{{ ansible_env.HOME }}/.local/bin/poetry --version"

```

register: poetry_version_output

- name: Display Poetry version

debug:

msg: "Poetry version installed: {{ poetry_version_output.stdout }}"

ДОДАТОК Б

Список публікацій здобувача

Наукові праці, в яких опубліковано основні наукові результати дисертації:

1. Волокита А., Русінов В., Мугуєв К. Дослідження відмовостійкості для топології Де Бруйна на основі коефіцієнта посередництва // Технічні науки та технології. – 2021. – Вип. 1, № 23. – С. 69–80. – doi: 10.25140/2411-5363-2021-1(23)-69-80.
2. Korenko D., Cherevatenko O., Rusinov V., Kulakov Y. Creation of the method of multipath routing using known paths in software-defined networks // Technology Audit and Production Reserves. – 2022. – Vol. 4, No. 2(66). – P. 19–24. – doi: 10.15587/2706-5448.2022.262787.
3. Rusinov V., Honcharenko O., Volokyta A., Loutskii H., Pustovit O., Kyrianov A. Methods of topological organization synthesis based on tree and dragonfly combinations // Lecture Notes on Data Engineering and Communications Technologies. – 2023. – P. 472–485. – doi: 10.1007/978-3-031-36118-0_43.
4. Volokyta A., Loutskii H., Honcharenko O., Cherevatenko O., Rusinov V., Kulakov Y., Tsybulia S. Fault Tolerance Exploration and SDN Implementation for de Bruijn Topology based on betweenness Coefficient // International Journal of Computer Network and Information Security. – 2024. – Vol. 16, No. 1. – P. 97–112. – doi: 10.5815/ijcnis.2024.01.08.
5. Русінов, В. (2025). Спосіб розгортання AI платформи з використанням методології MLOps. // Смарт технології: промислова та цивільна інженерія, 3(16), 19-28. – doi: 10.32347/st.2025.3.1202.
6. Rusinov, V., & Basenko, N. (2025, April). Exploration of the Efficiency of SLM-Enabled Platforms for Everyday Tasks. In 13th International Conference on Applied Innovations in IT (p. 133). doi: 10.25673/119225

Анотація наукових результатів дисертації:

7. Rusinov V., Cherevatenko O. Method of neural network training for Edge architecture // The International Conference on Security, Fault Tolerance, Intelligence. – Kyiv, Ukraine, June 30, 2022. – 2022. – [Electronic resource]. – Available: <http://icsfti.kpi.ua/proc/article/view/280999/291232>.
8. Rusinov V., Muhiyev K. Development of a scalable AI platform based on integration of Edge computing with Cloud technologies // The International Conference on Security, Fault Tolerance, Intelligence. – Kyiv, Ukraine, June 28, 2024. – 2024. – [Electronic resource]. – Available: <https://icsfti-proc.kpi.ua/article/view/298011>.
9. Rusinov V. Increasing the efficiency of AI task deployment for Cloud-Edge environments // The International Conference on Security, Fault Tolerance, Intelligence. – Kyiv, Ukraine, June 28, 2024. – 2024. – [Electronic resource]. – Available: <https://icsfti-proc.kpi.ua/article/view/307047>.
10. Rusinov V. Development of MLOps pipeline to support AI systems // The International Conference on Security, Fault Tolerance, Intelligence. – Kyiv, Ukraine, June 28, 2024. – 2024. – [Electronic resource]. – Available: <https://icsfti-proc.kpi.ua/article/view/307046>.

Праці, які додатково відображають результати дисертації:

11. Oleksandr Pustovit, Rusinov Volodymyr, Oleksii Cherevatenko, Leonid Pustovit, Artem Volokyta. Isoefficient calculation method for discrete Fourier transform. Information, Computing and Intelligent systems : International Conference ICSFTI2022, м. Київ, 30 черв. 2022 р. Київ, 2022.
12. Голованенко М.В., Русінов В.В. (2025). Цифровізація бізнес-процесів компанії із впровадженням AI-платформи. Теоретичні та прикладні питання економіки. Збірник наукових праць, 1(50).

ДОДАТОК В

Акт впровадження результатів дисертаційного дослідження

ТОВ "ТЕЛЕКАРТ-ПРИЛАД"

Україна, 65104, м. Одеса, пр-т. Небесної Сотні, 105
 тел. +38 048-705-15-15
 e-mail: office@telecard.com.ua;
<http://www.telecard.com.ua>

UA-65104, 105, Nebesnoi Sotni ave, Odesa, Ukraine
 phone +38 048 705-15-15
 e-mail: office@telecard.com.ua;
<http://www.telecard.com.ua>

АКТ

впровадження результатів дисертаційної роботи

Русінова Володимира Володимировича,

поданої на здобуття наукового ступеня доктора філософії, на тему

«Метод підвищення ефективності розгортання компонентів платформи
 вбудованих систем»

ТОВ «ТЕЛЕКАРТ-ПРИЛАД» є національним виробником сучасних цифрових засобів зв'язку та комплексів управління на різноманітних базових шасі. Основні сфери діяльності - виробництво продукції для силових структур та виробництво продукції для енергетичного сектора, розробка системних проєктів для Замовників приватної та державної форм власності.

Підприємство «ТЕЛЕКАРТ-ПРИЛАД» підтверджує, що результати дисертаційної роботи Русінова Володимира Володимировича були впроваджені в межах прикладних розробок і виробничих процесів, зокрема в рамках реалізації інженерних рішень для автоматизованих систем контролю і керування технологічними об'єктами. Запропоновані у дисертації методи були використані для удосконалення механізмів розгортання компонентів програмного забезпечення на розподілених вбудованих платформах, що функціонують в умовах обмежених ресурсів.

Завдяки впровадженню запропонованого методу вдалося істотно підвищити ефективність використання обчислювальних ресурсів систем, скоротити загальний час налаштування й оновлення компонентів на цільових пристроях, а також зменшити ймовірність виникнення помилок у процесі розгортання. У процесі дослідного впровадження було виявлено, що система з оновленими компонентами демонструє підвищену стійкість до збоїв. Це досягалося за рахунок вбудованих механізмів самодіагностики та здатності до адаптивного перенесення обчислювальних задач між вузлами системи. Такий підхід дозволив мінімізувати час простою при часткових відмовах, а також забезпечити безперервність обробки критично важливих даних.

Розподіл навантаження між функціональними сервісами відбувався з урахуванням поточного стану апаратної інфраструктури та змін у зовнішньому середовищі. Система змогла підтримувати стабільну роботу навіть в умовах непередбачуваних змін трафіку чи доступності окремих вузлів. Оптимізоване управління навантаженням дозволило знизити кількість збоїв, підвищити продуктивність та покращити якість сервісу.

У процесі інтеграції результатів дослідження були проведені додаткові експериментальні випробування в умовах, максимально наближених до реального виробництва. Це дозволило перевірити стійкість програмного забезпечення до

неочікуваних обставин, а також перевірити адаптивність розроблених механізмів до нестандартних сценаріїв використання.

Підприємство позитивно оцінює рівень дисертаційної роботи Русінова Володимира Володимировича, відзначаючи її інноваційний характер, чітке формулювання завдань, а також націленість на вирішення практичних задач. Запропоновані методи оптимізації функціонування телекомунікаційних систем довели свою ефективність в умовах реальної експлуатації, що свідчить про високу технологічну зрілість та актуальність дослідження.

Генеральний директор



Олексій КОЗЛОВ



AQAP 2110

ДОДАТОК Г

Акт впровадження результатів дисертаційного дослідження

ДОВІДКА

про впровадження результатів дисертаційної роботи
Русінова Володимира Володимировича, поданої на здобуття наукового
ступеня доктора філософії, на тему «Метод підвищення ефективності
розгортання компонентів платформи вбудованих систем»

Інститут прикладних систем управління Національної академії наук України (ІПСУ НАН України) - провідна наукова установа у сфері інформаційних та адитивних технологій, штучного інтелекту та робототехніки. Останні роки в Інституті проводяться дослідження у сфері розробки елементів штучного інтелекту в задачах математичного прогнозування глобальних процесів, робототехніки та адитивних технологій з метою впровадження високих технологій в економіку України, зокрема в медичну, авіаційну, та оборонну галузі.

ІПСУ НАН України підтверджує, що результати дисертаційної роботи Русінова Володимира Володимировича були використані при виконанні науково-дослідної роботи: «Розроблення систем керування з елементами штучного інтелекту для роботизованих систем та адитивних технологій», номер державної реєстрації №0124U003352, при описі проєктних та виробничих процесів, пов'язаних із розробкою, тестуванням і розгортанням платформ вбудованих систем, зокрема — у рамках реалізації проєктів зі створення систем автоматизованого контролю та керування технічними об'єктами.

Розроблені в дисертаційній роботі методи були використані при опробовуванні спеціалізованого програмного забезпечення для управління життєвим циклом компонентів у вбудованих системах, що дало змогу:

- Скоротити час розгортання компонентів на цільових пристроях шляхом оптимізації завантаження та конфігурації.
- Забезпечити автоматичну перевірку сумісності програмних модулів у середовищах із обмеженими ресурсами.
- Підвищити надійність процесу оновлення за рахунок включення механізмів зворотного відкату.
- Спростити масштабування систем на нові апаратні платформи без втрати цілісності конфігурацій.

- Забезпечити підтримку «гарячого» оновлення критичних компонентів без зупинки основних процесів.

ІНСТИТУТ НАУКИ І СПОРТУ НАН УКРАЇНИ
Відзначає достатню якість дисертаційної роботи
Русакова Володимира Володимировича, що характеризується науковою
новизною, глибоким аналізом проблеми та практичною спрямованістю. У
роботі запропоновано ефективний метод розгортання компонентів платформи
вбудованих систем, який враховує обмеження цільового середовища та
забезпечує підвищену надійність, масштабованість і швидкість програмного
забезпечення.

Директор



Олег КОПЫКА

ДОДАТОК Д

Подяка

Ця дисертаційна робота і всі розробки пов'язані з нею — довготривалий процес, в якому мені допомогло багато людей своїми ідеями, баченням та просто своїм прикладом. Це ті люди, які не ховались за своїми рангами і просто не знали слова «ні». Формально, я не можу їх внести до списку на перших сторінках роботи, але вважаю що їх вклад має бути віддячений.

Дякую Антонюку Андрію Івановичу за велику допомогу та довіру до мене. Ви допомогли прокласти мені шлях до науки, коли ще навчався на бакалавраті, і дали мені великий кредит довіри, коли передали свій предмет мені. Можливість спілкуватись зі студентами дозволила мені бачити різні сторони проблематики, та шукати нові, інновативні підходи до вирішення задач.

Дякую Єханурову Юрію Івановичу за відкриття нових горизонтів, за курс системного мислення, за можливість бути в колах, в яких я навіть не міг очікувати себе знайти. Ваше бачення розвитку країни не тільки імponує мені, але й показує як можна застосувати нові розробки не лише в цілях заповнення звітів і наблизити те чудо на Дніпрі, яке ми не чекаємо, а над яким активно працюємо.

Дякую Копійці Олегу Валентиновичу за змістовну роботу над розумінням наукового методу та дирекції тих напрацювань, що були, в наукове русло. В наших зустрічах я отримав розуміння того що зараз є актуальним, вартим свого часу і яким чином організувати роботу над науковими проєктами.

Дякую друзям, колегам та близьким, які були поруч при написанні дисертації. Окрема подяка і велика шана Збройним Силам України, які забезпечили можливість написання цієї роботи, дякую кожному, хто долучився до захисту держави.