



**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

ДМИТРЕНКО ОЛЕКСАНДРА АНАТОЛІЇВНА

УДК 004.75

ДИСЕРТАЦІЯ

**МЕТОД ДОПОВНЮВАЛЬНИХ НАВАНТАЖЕНЬ ДЛЯ РОЗПОДІЛУ
ЗАДАЧ В ХМАРНИХ СИСТЕМАХ**

Спеціальність 172 – Телекомунікації та радіотехніка

Галузь знань 17 – Електроніка та телекомунікації

Подається на здобуття наукового ступеня доктора філософії

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

О. Дмитр Дмитренко О.А.

Науковий керівник: Скулиш М.А. доктор технічних наук, професор

Київ – 2025

АНОТАЦІЯ

Дмитренко О.А. Метод доповнювальних навантажень для розподілу задач в хмарних системах. — Кваліфікаційна наукова праця на правах рукопису.

Дисертація на здобуття наукового ступеня доктора філософії за спеціальністю 172 — Телекомунікації та радіотехніка. — Навчально-науковий інститут телекомунікаційних систем КПІ ім. Ігоря Сікорського, Київ, 2025.

Завдання оптимізації використання ресурсів хмарних систем має високий пріоритет, оскільки ця технологія стала широко розповсюдженою та легкодоступною. Поширеною практикою є створення ефективних сервісів, які легко адаптуються до різних типів продуктів, наприклад, безсерверні технології, платні підлаштовані під потреби бізнесу бази даних та програмне забезпечення, оптимізоване для хмари.

Тестування продуктивності хмарних застосунків стало значно простішим із появою засобів розгортання, т.я. Docker та засобів оркестрації, т.я. Kubernetes, що додатково збільшило попит на хмарні ресурси. Оскільки значна кількість ресурсів витрачається на розробку та обслуговування хмарної інфраструктури, важливо розуміти, як ці ресурси можна оптимізувати.

Для організації роботи застосунків на хмарі використовується планувальник. В його функції входить аналіз вимог до роботи контейнерів з обчислювальними задачами та пошук кластера та вузла розміщення для них. Також виконується збір метрик роботи контейнера.

В ході аналізу наявних підходів до розподілення навантаження задач виявлено певні їх недоліки. Для оцінки потреб роботи контейнеризованої задачі використовуються наступні параметри: бажані та максимальні значення requests та limits по RAM та CPU (можна додавати додаткові показники). Ці значення є основоположними при створенні подів на

Kubernetes чи їх аналогів на хмарах. Якщо ці значення задані, то у задачі більший пріоритет та менша ймовірність бути вилученою з вузла в разі його перевантаження. Водночас задача може бути вилучена, якщо потребує більше ресурсів, ніж вказано у граничному значенні. Інша проблема, що виникає при роботі ПЗ, є вивільнення RAM. Якщо обчислювальна задача нею заволоділа, то при зменшенні необхідності в оперативній пам'яті її вивільнення може не відбуватись без особливих налаштувань програми.

Ідея роботи — трансформувати роль бажаних та максимальних значень ресурсів що виділяються на роботу пода з вирішального чинника на координаційний параметр, прибираючи права критичного впливу на вилучення справних задач з вузла. Це можливо замінивши підхід до виділення вузлового ресурсу.

Проаналізувавши історичні дані та в разі їх однорідності, можна оцінити потреби задач в кожен окреслений період часу. Відповідно до потреб однієї задачі, буде знайдено доповнення її «прогалин» у навантаженні, та заповнене в результаті комбінації цієї задачі з іншими.

Робота присвячена обґрунтуванню цієї ідеї за допомогою **розробки математичної моделі** з використанням нечіткої теорії ґраток, та доведенню її життєздатності через розробку методу та алгоритму пошуку доповнювальних навантажень та проведенню експериментальних та аналітичних розрахунків її якості.

З метою забезпечити стале повне використання виділеного ресурсу була побудована математична модель. Вона базується на принципі доповнюваності (комплементарності) та теорії нечітких ґраток (об'єднання теорії ґраток та нечіткої логіки). Параметрами моделі є RAM, CPU, мережевий ресурс, а також вона враховує дисковий простір.

Під комплементарністю мається на увазі що поди в межах доповнювальних груп сумарно формують повне навантаження на доступні вузли, розподілене відповідно до часових періодів. Задача оптимізації, яку ставить матмодель є максимізувати ефективність використання ресурсів, мінімізуючи їх простій. Як обмеження, розглядається прив'язаність

контейнера до вузла або іншої частини ПЗ та подільність обчислювальної задачі на менші.

Вперше запропоновано метод доповнювальних навантажень для формування груп сталої навантаженості. Алгоритм методу спрямований на оптимізацію використаних ресурсів під час роботи багатокомпонентної системи, зокрема, при масштабуванні її вузлів для забезпечення покриття навантаження всіх користувачьких запитів.

Суть методу доповнювальних навантажень. Під час пошуку комплементарних навантажень обчислювальних задач алгоритм виконує кластеризацію, в результаті чого екземпляри мікросервісів групуються за схожістю робочих шаблонів у класи еквівалентності.

В межах кожної групи екземпляри сортуються за амплітудою використання ресурсів. Далі групи, що визначені як доповнювальні, перетинаються з метою знайти пари комплементарних мікросервісів. Екземпляри з протилежними шаблонами навантаження, але схожими амплітудами, комбінуються для ефективнішого використання ресурсів. Також можуть розглядатися пари з низькою амплітудою для заповнення «пробілів».

Далі відбувається пошук екземплярів мікропослуг, що відповідають критеріям доповнення. Коли доповнення в межах першого підходячого кластера, чи наступних, не знаходиться, виконується збереження статистики комбінацій з усіма екземплярами, і пошук триває до знаходження доповнень або вичерпання доповнювальних груп.

Процес повторюється до знаходження всіх можливих пар без повторних включень; залишкові екземпляри мікропослуг є очікуваними. Для мікросервісів, що не знайшли доповнень, застосовується додатковий аналіз екстремумів з використанням середнього та стандартного відхилення, щоб підвищити ймовірність знайти доповнення. Кластеризація і пошук продовжуються до останньої успішної спроби. Якщо залишається невелика кількість мікросервісів без пари, вони комбінуються в більшій кількості, ніж по два для досягнення мети.

При реалізації алгоритму на основі методу застосовувалось Z-масштабування для нормалізації та виділення шаблону роботи мікропослуги, KMeans кластеризація для групування за схожим шаблоном роботи нормалізованих рядів. Додатково для візуалізації використовувались алгоритми FastDTW для накладання графіків шаблонів, та PCA — для зображення двовимірного простору для часових рядів замість n-вимірного.

При пошуку доповнень використовувались сортування, бінарний пошук та жадібний алгоритм. Поділ на основі середнього значення та стандартного відхилення застосовувався для написання алгоритму пошуку екстремумів та розширених екстремумів для виділення їх в окремі контейнери. Групування малозатратних мікропослуг здійснювалось за допомогою розв'язання задачі про множинний рюкзак.

Доповнювальний розподіл навантажень сприятиме суттєвому зменшенню необхідності розв'язання проблеми перегрупування навантаження в режимі реального часу, коштом розміщення на серверах заздалегідь сформованих груп навантажень, що утворюють доповнені ґратки.

У роботі розглядаються ключові показники комп'ютерних систем, які можна виміряти та на які можна впливати. Ці характеристики включають пропускну здатність каналів передачі, оцінку продуктивності на основі затримки, обсяг оперативної та постійної пам'яті, обчислювальну потужність та кількість ядер.

Запропоноване удосконалення підходу до автомасштабування у хмарній системі, спрямоване на зниження частоти необхідності масштабування шляхом ефективного планування навантаження на вузли на етапі розподілу за~~С~~дач, яке апріорі не вимагатиме масштабування. У традиційних підходах до автомасштабування хмарних ресурсів, масштабування відбувається у відповідь на зміну поточного навантаження, що призводить до частих операцій створення та видалення екземплярів обчислювальних вузлів, споживаючи додаткові ресурси для управління цими процесами. Якщо ж навантаження не змінюватиметься, то і

необхідності в масштабуванні не буде. Все ж, воно відбуватиметься в екстремальних випадках, коли ПЗ переживатиме незвичайні навантаження та відбуватиметься не для одної задачі, а для всієї групи.

У дисертаційній роботі **запропоновано новий підхід до планування навантаження в хмарних системах**, що забезпечує оптимальне формування та розподіл пакетів задач у багатосерверному середовищі. Основою запропонованого підходу є моніторинг метрик використання ресурсів та застосування методу доповнювальних навантажень.

Такий підхід оптимізує використання ресурсів хмарної системи шляхом ефективного розподілу навантаження доповнених мікропослуг між різними серверними групами. Оптимізація полягає в максимізації використання серверних ресурсів шляхом заповнення його вільного часу іншими мікросервісами й, таким чином, зменшення простою серверів. Це, своєю чергою, сприятиме покращенню загальної продуктивності РС та зменшенню витрат на обслуговування хмарних інфраструктур.

Наведений підхід застосовується як до мікросервісів, так і до монолітів та безсерверні технологій, з описаними мінімальними змінами.

Апробація підходу виконувалась методом імітаційного моделювання, проведенням наукового експерименту та із застосуванням методу інтелектуального аналізу даних.

Для виконання апробації було розроблено алгоритм та програму для визначення комплементарних екземплярів мікросервісів, які могли б ефективно використовувати серверні ресурси. Запропонований алгоритм може бути корисним для постачальників хмарних послуг та організацій, які використовують хмарні середовища для розгортання своїх застосунків.

Апробація показала, що залежно від обраного ключового критерію: CPU, RAM чи мережевий ресурс, можна зменшити кількість технічного забезпечення на 8%, 10% та 17% відповідно, що сумарно становить понад 9% з врахуванням частки присутності кожного критерію. До технічних покращень також відноситься підвищення енергоефективності послуг хостингу на 10–15% та подовження строку служби на 8–12% на вузлах

повністю з доповнювальним навантаженням. Причиною цього є сталість навантаження та відсутність стрибків електроенергії.

Також, запропонований підхід сприяє зменшенню кількості перепланувань розміщення обчислювальних задач на 25–35%, що становить 9–15% від задач вузлів-планувальників. Таких вузлів на кожному кластері залежно від розміру від 3 до 7. Це дозволяє зменшити технічні вимоги до вузлів, або ж підвищити на них навантаження шляхом додавання нових вузлів у кластер.

У зв'язку з тим, що кількість доповнювальних груп складатиме 20–30%, відповідно спостерігатиметься зменшення потреб в автомасштабуванні до 20%. На вузлах, де частково чи повністю будуть розміщені доповнювальні групи, можна зменшити кількість запасного ресурсу headroom відповідно до пропорції присутності доповнювальних груп на обчислювальних вузлах. Пропорція буде ефективною щодо кількості запасного ресурсу, якщо доповнювальні групи складатимуть 20–24% залежно від ключового ресурсу, а отже запасний ресурс можна зменшити на максимально 20%.

В разі виділення вузлів, повністю заповнених групами доповнювальних навантажень, можна забезпечити збільшення пропускної можливості багатосерверної системи в години пік, яке становитиме 5–15% на 10–15% відсотках серверів.

Основні ідеї методу доповнювальних навантажень впроваджені у курсі «Big Data», що викладається аспірантам НН ІТС на кафедрі ІТТ.

Ключові слова: хмарна інфраструктура, горизонтальне масштабування, Kubernetes, оптимізація витрат, розподіл завдань, телекомунікаційні сервіси, конфігурація ресурсів, мікросервісна архітектура, оркестрація контейнерів, часові ряди, енергоефективність, комп'ютерні системи, доповнювальні навантаження, багатосерверні системи, класифікація трафіку, мікропослуги, безсерверні технології.

ABSTRACT

O.A. Dmytrenko. Method of complementary loads for task distribution in cloud systems. — Qualifying scientific work on manuscript rights.

Thesis for graduation scientific degree of Philosophy Doctor by speciality 172 — Telecommunications and radio engineering. — Educational and Scientific Institute of Telecommunication Systems of KPI named after Igor Sikorsky, Kyiv, 2025.

The task of optimizing cloud system resource usage has high priority, as this technology has become widespread and easily accessible. Creating efficient services that easily adapt to different types of products is a common practice, for example, serverless technologies, paid databases customized for business needs, and software optimized for the cloud.

Testing cloud application performance has become significantly easier with the emergence of Docker and Kubernetes, which has further increased demand for cloud resources. Since a significant amount of resources is spent on developing and maintaining cloud infrastructure, it's important to understand how these resources can be optimized.

A scheduler is used to organize applications on the cloud. Its functions include analyzing the requirements for running containers with computational tasks and finding a cluster and placement node for them. The scheduler also collects container performance metrics.

When analyzing existing approaches to task load distribution, certain drawbacks were identified. The following parameters are used to assess the needs of a containerized task: desired and maximum values of requests and limits for RAM and CPU (additional indicators can be added). These values are fundamental when creating pods on Kubernetes or their analogs on clouds. If these values are set, the task has higher priority and less likelihood of being evicted from a node in case of overload. At the same time, a task can be evicted if it requires more resources than specified in the limit value. Another problem that occurs during software operation is the release of RAM. If a computational

task has seized it, then when the need for RAM decreases, its release may not occur without special program settings.

The idea of this work is to transform the role of desired and maximum resource values allocated for pod operation from a decisive factor to a coordination parameter, removing the right of critical impact on the eviction of functioning tasks from a node. This is possible by changing the approach to node resource allocation.

By analyzing historical data and in case of their homogeneity, it's possible to estimate the needs of tasks in each outlined time period. According to the needs of one task, a complement to its «gaps» in the load will be found and filled as a result of combining this task with others.

The work is dedicated to substantiating this idea through the development of a mathematical model using fuzzy lattice theory, proving its viability through the development of a method and algorithm for finding complementary loads, and conducting experimental and analytical calculations of its quality.

To ensure consistent full use of allocated resources, a mathematical model was built. It is based on the principle of complementarity and the theory of fuzzy lattices (combining lattice theory and fuzzy logic). The model parameters are RAM, CPU, network resource, and it also takes into account disk space.

By complementarity, it means that pods within complementary groups collectively form a complete load on available nodes, distributed according to time periods. The optimization task set by the mathematical model is to maximize resource utilization efficiency while minimizing idle time. As constraints, the attachment of a container to a node or other part of the software and the divisibility of a computational task into smaller ones are considered.

For the first time, a method of complementary loads for forming groups of stable loading is proposed. The algorithm of the method is aimed at optimizing the resources used during the operation of a multi-component system, in particular, when scaling its nodes to ensure coverage of the load of all user requests.

The essence of the complementary loads method: During the search for complementary loads of computational tasks, the algorithm performs clustering,

as a result of which microservice instances are grouped according to the similarity of working patterns into equivalence classes.

Within each group, instances are sorted by the amplitude of resource use. Then, groups defined as complementary intersect to find pairs of complementary microservices. Instances with opposite load patterns but similar amplitudes are combined for more efficient resource use. Pairs with low amplitude may also be considered to fill «gaps.»

Next, a search is conducted for microservice instances that meet the complementary criteria. When a complement within the first suitable cluster or subsequent ones is not found, statistics of combinations with all instances are saved, and the search continues until complements are found or complementary groups are exhausted.

The process is repeated until all possible pairs are found without repeated inclusions; residual microservice instances are expected. For microservices that didn't find complements, additional analysis of extremes is applied using mean and standard deviation to increase the probability of finding a complement. Clustering and search continue until the last successful attempt. If a few microservices remain without pairs, they are combined in larger quantities than two to achieve the goal.

When implementing the algorithm based on the method, Z-scaling was used for normalization and highlighting the microservice operation pattern, KMeans clustering for grouping by similar normalized series operation pattern. Additionally, for visualization, FastDTW algorithms were used to overlay pattern graphs, and PCA to represent a two-dimensional space for time series instead of n-dimensional.

When searching for compliments, sorting, a binary search, and a greedy algorithm were used. Division based on mean value and standard deviation was applied to write an algorithm for finding extremes and extended extremes to separate them into individual containers. Grouping of low-cost microservices was carried out by solving the multiple knapsack problem.

Complementary load distribution will significantly reduce the need to solve the problem of load regrouping in real-time, at the cost of placing pre-formed load groups on servers that form complemented lattices.

The paper considers key indicators of computer systems that can be measured and influenced. These characteristics include channel bandwidth, latency-based performance assessment, RAM and permanent memory capacity, computing power, and number of cores.

An improvement to the approach to auto-scaling in the cloud system is proposed, aimed at reducing the frequency of scaling needs through effective planning of node load at the task distribution stage, which a priori will not require scaling. In traditional approaches to auto-scaling cloud resources, scaling occurs in response to changes in current load, leading to frequent operations of creating and deleting computing node instances, consuming additional resources to manage these processes. If the load doesn't change, there will be no need for scaling. Still, it will occur in extreme cases when the software experiences unusual loads and will occur not for a single task but for the entire group.

The dissertation proposes a new approach to load planning in cloud systems, providing optimal formation and distribution of task packages in a multi-server environment. The basis of the proposed approach is monitoring resource usage metrics and applying the complementary loads method.

This approach optimizes the use of cloud system resources through efficient distribution of the load of complemented microservices between different server groups. Optimization consists in maximizing the use of server resources by filling its free time with other microservices and thus reducing server downtime. This, in turn, will contribute to improving the overall cloud performance and reducing the cost of maintaining cloud infrastructures.

The approach applies to microservices as well as monoliths and serverless technologies, with described minimal changes.

The approach was tested using simulation modelling, conducting a scientific experiment, and applying the data mining method.

For testing, an algorithm and program were developed to determine complementary instances of microservices that could efficiently use server resources. The proposed algorithm can be useful for cloud service providers and organizations that use cloud environments to deploy their applications.

Testing has shown that depending on the selected key criterion: CPU, RAM, or network resources, it's possible to reduce the amount of technical

equipment by 8%, 10%, and 17% respectively, which in total represents over 9% when accounting for the proportion of each criterion. Technical improvements also include increasing the energy efficiency of hosting services by 10–15% and extending the service life by 8–12% on nodes fully loaded with complementary workloads. This is due to the stability of the load and the absence of power surges.

Also, the proposed approach helps reduce the number of computational task placement rescheduling by 25–35%, which represents 9–15% of scheduler node tasks. Depending on the size, there are 3 to 7 such nodes in each cluster. This allows for reducing technical requirements for nodes or increasing their load by adding new nodes to the cluster.

Since the number of complementary groups will comprise 20–30%, there will be a corresponding reduction in auto-scaling needs of up to 20%. On nodes where complementary groups are partially or fully placed, the amount of headroom reserve resource can be reduced in proportion to the presence of complementary groups on computational nodes. The proportion will be effective regarding the amount of reserve resource if complementary groups make up 20–24% depending on the key resource, thus headroom can be reduced by a maximum of 20%.

In the case of allocating nodes fully loaded with complementary workload groups, it is possible to increase the throughput capacity of a multi-server system during peak hours, which will amount to 5-15% on 10-15% of servers.

The main ideas of the method of complementary loads are implemented in the course «Big Data», which is taught to graduate students of the National Institute of Information Technology at the Department of Information Technology.

Keywords: cloud infrastructure, horizontal scaling, Kubernetes, cost optimization, traffic classification, task distribution, telecommunications services, resource allocation, microservice architecture, container orchestration, time series, energy efficiency, computer systems, complementary workloads, multi-server systems, serverless technologies.

СПИСОК ПУБЛІКАЦІЙ ЗА ТЕМОЮ ДИСЕРТАЦІЇ

Наукові праці, в яких опубліковані основні наукові результати дисертації

1. О. А. Дмитренко і М. А. Скулиш, «Методи збору інформації та реалізації алгоритму доповнювальних навантажень», Системи управління, навігації та зв'язку. Збірник наукових праць, вип. 4(78), с. 56–59, Лис 2024, doi: 10.26906/SUNZ.2024.4.056
2. М. А. Скулиш, О. А. Дмитренко, «Метод організації мікросервісів на серверних групах Kubernetes», Вчені записки Таврійського Національного Університету Імені В.І. Вернадського. Серія: Технічні науки, вип. 1(5), с. 291–297, Груд 2024, doi: 10.32782/2663-5941/2024.5.1/41
3. О.А. Дмитренко, М.А. Скулиш, «Методи відмовостійкості резервування пристроїв IoT». Науковий журнал «Інфокомунікаційні та комп'ютерні технології» №2(04). Київ: Університет «Україна». УДК 004.75. грудень 2022. 59-65 с., doi: 10.36994/2788-5518-2022-02-04-06
4. О. Дмитренко, М. Скулиш, «Групування мікропослуг для використання неповно залучених ресурсів», Сучасні інформаційні технології, номер 2024, 1(3), с. 78-85, Київський національний університет імені Тараса Шевченка, Видавничо-поліграфічний центр «Київський університет», DOI: <https://doi.org/10.17721/AIT.2024.1.08>
5. O. Dmytrenko, M. Skulysh, L. Globa, «Microservice Complimentary Groups Determination Algorithm for the Effective Resource Usage», in Proceedings of the 14th International Scientific and Practical Programming Conference (UkrPROG 2024), I. Sinitsyn and P. Andon, Eds., in CEUR Workshop Proceedings, vol. 3806. Kyiv, Ukraine: CEUR, May 2024, pp. 180–201. Accessed: Dec. 28, 2024. [Online]. Available: https://ceur-ws.org/Vol-3806/#S_55_Dmytrenko_Skulysh_Globa
6. O. Dmytrenko and M. Skulysh, «Method of Grouping Complementary Microservices Using Fuzzy Lattice Theory», presented at the International Conference on Applied Innovations in IT (ICAIIIT), E. Siemens, Ed., in 1, vol.

12. Koethen, Germany: Anhalt University of Applied Sciences Bernburg / Koethen / Dessau, Mar. 2024, pp. 11–18. doi: 10.25673/115636

Наукові праці, які засвідчують апробацію матеріалів дисертації

7. О.А. Дмитренко, «Мікросервісна архітектура та її задачі на прикладах з реального життя», Зібрання тез до конференції «XVII Міжнародна науково-технічна конференція «Перспективи телекомунікацій 2023», 18-21 квітня, 2023. Київ. УДК 004.75. с. 239-242
8. О.А. Dmytrenko, M.A. Skulysh, «Handling with a Microservice Failure and Adopting Retries», Collection of abstracts for the conference «All-Ukrainian science-practical conference of students, Ph. D. students and young scientists», June 15 2022, pp 194-196
9. Дмитренко О.А., Скулиш М.А., «Визначення мікропроцесорних груп для ефективного використання процесорних потужностей», журнал «Проблеми програмування» №2–3, грудень, 2024, 8 стр, УДК 004.75, DOI: 10.15407/pp2024.02–03.215

ЗМІСТ

АНОТАЦІЯ.....	1
ABSTRACT.....	7
ЗМІСТ.....	14
СПИСОК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАЧЕНЬ.....	22
ВСТУП.....	23
РОЗДІЛ 1	29
ПРОБЛЕМА КІЛЬКОСТІ РЕСУРСІВ ТА ЕНЕРГОЕФЕКТИВНОСТІ ОБСЛУГОВУВАННЯ НАВАНТАЖЕННЯ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНОЇ СИСТЕМИ.....	29
1.1. Особливості побудови та обслуговування телекомунікаційної мережі....	29
1.1.1. Архітектура сучасних телекомунікаційних мереж.....	30
1.1.2. Мережа доставляння контенту в телекомунікаційних мережах.....	31
1.1.3. Проблема нерівномірного навантаження на телекомунікаційну інфраструктуру.....	33
1.1.4. Стратегії оптимізації використання телекомунікаційної інфраструктури.....	35
1.1.5. Конвергенція послуг та розподілена платформа обслуговування....	37
1.1.6. Дата-центри оператора телекомунікаційних послуг.....	38
1.1.7. Технічні проблеми в телекомунікаційній інфраструктурі.....	40
1.1.8. Програмне забезпечення як послуга для телекомунікаційних сервісів.....	41
1.1.9. Особливості побудови та обслуговування телекомунікаційної мережі.....	41
1.1.10. Інтеграція з іншими операторами та провайдерами.....	42
1.1.11. Майбутнє телекомунікаційних мереж.....	43
1.2. Хмарна інфраструктура та класичні послуги хостингу.....	45
1.2.1. Класичні або традиційні послуги хостингу.....	47
1.2.2. Хмарні або клауд послуги хостингу.....	47
1.2.3. Види серверів у дата-центрах.....	49
1.2.4. Форма ПЗ для хмарних технологій.....	53

1.3. Особливості розподілу навантаження та балансування в сучасних інформаційно-комунікаційних мережах.....	54
1.3.1. Критерії розподілу навантаження в Kubernetes.....	55
1.3.2. Балансування навантаження в Amazon Web Service.....	55
1.3.3. Балансування на Google Cloud.....	59
1.3.4. Порівняння балансувальників GCP.....	61
1.3.5. Порівняння балансувальників за типом їх роботи в різних хмарних системах.....	62
1.4. Планування розподілу задач у дата-центрі та на хмарі.....	64
1.4.1. Задачі планувальника, його обмеження та дотримання політик.....	64
1.4.2. Будова розподіленого планувальника задач.....	65
1.4.3. Типи планувальників.....	67
1.4.4. Планувальник Kubernetes.....	68
1.4.5. Ефективність планувальника Kubernetes.....	70
1.4.6. Енергоощадження на Kubernetes.....	70
1.4.7. Визначення кількості потрібних ресурсів на Kubernetes.....	71
1.4.8. Інструменти створення автоматичної конфігурації поду.....	74
1.4.9. Врахування історичних даних та пікових навантажень.....	75
1.4.10. Зменшення кількості подів.....	76
1.4.11. Проблеми планувальників на базі Kubernetes.....	76
1.4.12. Планувальники на AWS.....	78
1.4.13. Планувальники GCP.....	82
1.4.14. Порівняння планувальників.....	85
1.4.15. Розміщення подів одного програмного застосунку на різних кластерах.....	86
1.4.16. Розміщення контейнерів різних програмних застосунків на одному поді.....	87
1.5. Додаткові можливості для підвищення надійності роботи системи та її швидкого відгуку.....	90
1.5.1. Класифікація несправностей.....	90
1.5.2. Додавання резервування в систему.....	91

1.5.3. Холодний режим очікування.....	92
1.5.4. Теплий режим очікування.....	92
1.5.5. Гарячий режим очікування.....	93
1.6. Особливості мікросервісної архітектури.....	95
1.6.1. Мікросервісна архітектура в порівнянні з монолітною.....	97
1.6.2. Особливості комунікація між мікросервісами.....	98
1.6.3. Недоступність частин мікросервісної системи.....	102
1.7. Види групування ресурсів.....	103
1.8. Особливості масштабування.....	104
1.8.1. Вертикальне масштабування.....	106
1.8.2. Горизонтальне масштабування.....	108
1.8.3. Горизонтальне масштабування мікросервісів, монолітів та баз даних	110
1.9. Проблеми планування задач у багатосерверній інфраструктурі.....	112
1.9.1. Проблема вивільнення невживаної оперативної пам'яті.....	113
1.9.2. Проблема різниці між заданою та необхідною кількістю ресурсів	114
ВИСНОВКИ.....	116
РОЗДІЛ 2	121
МАТЕМАТИЧНА МОДЕЛЬ РОЗПОДІЛУ НАВАНТАЖЕННЯ В ХМАРНОМУ	
ПЛАНУВАЛЬНИКУ РЕСУРСІВ.....	121
2.1. Теорія нечітких ґраток у застосуванні до навантажень.....	121
2.1.1. Базові поняття теорії ґраток.....	121
2.1.2. Моделювання задач на хмарі через графи.....	123
2.1.3. Доповнювальні ґратки, або ґратки з доповненнями.....	124
2.1.4. Впровадження нечіткості для оцінки вжитку серверного ресурсу.	125
2.1.5. Доповнювальні ґратки у нечіткій логіці.....	126
2.1.6. Обґрунтування ідеї.....	128
2.1.7. Порівняння прикладу з теорією нечітких ґраток.....	130
2.1.8. Моделювання доповнювальних часових рядів задач у хмарній	
інфраструктурі.....	130
2.1.9. Багатозначна логіка Поста.....	132

2.1.10. Обґрунтування вибору теорії нечітких ґраток для опису задач в хмарній інфраструктурі.....	134
2.2. Математична модель формування груп доповнювальних навантажень	135
2.2.1. Параметри математичної моделі.....	135
2.2.2. Цільова функція.....	136
2.2.3. Обмеження матмоделі.....	136
ВИСНОВКИ.....	137
РОЗДІЛ 3	139
КОМПЛЕКСНИЙ МЕТОД ДОПОВНЮВАЛЬНОГО РОЗПОДІЛЕННЯ ОБЧИСЛЮВАЛЬНОГО НАВАНТАЖЕННЯ В БАГАТОСЕРВЕРНІЙ СИСТЕМІ.	139
3.1. Стратегії групування.....	139
3.1.1. Безпека та множинна оренда (multi-tenency) ресурсу.....	139
3.1.2. Спільні ресурси.....	140
3.1.3. Пропускна здатність каналу.....	142
3.1.4. Час і стан завантаження процесора.....	142
3.1.5. Географічні міркування та міркування про час активності.....	143
3.1.6. Моделювання способів групування задач.....	144
3.2. Загальний метод доповнювального розподілу навантаження.....	147
3.2.1. Для якого ПЗ застосовується метод.....	147
3.2.2. Прогнозування навантаження.....	148
3.2.3. Групування серверів.....	148
3.2.4. Оркестрування мікросервісів.....	149
3.2.5. Балансування навантаження.....	149
3.2.6. Перегрупування серверів.....	150
3.3. Етапи групування мікропослуг.....	150
3.3.1. Профілювання мікропослуг.....	150
3.3.2. Аналіз подібності.....	152
3.3.3. Аналіз взаємодоповнюваності.....	152
3.3.4. Динамічне групування.....	152
3.3.5. Постійний моніторинг та коригування.....	153

3.4. Основні характеристики та особливості хмарного середовища.....	153
3.4.1. Використання оперативної пам'яті.....	154
3.4.2. Навантаження на канал.....	156
3.4.3. Завантаження процесора.....	157
3.4.3.1. Приклад розрахунку завантаження процесора.....	159
3.4.3.2. Трансформація розрахунку відсотка завантаження на нове середовище.....	160
3.5. Алгоритм пошуку доповнювальних навантажень.....	162
3.5.1. Етап 1: Аналіз історичних даних, їх первинна нормалізація та систематизація.....	163
3.5.1.1. Крок 1: Збір історичних даних роботи програмних застосунків..	164
3.5.1.2. Крок 2: Первинна нормалізація вхідних даних.....	164
3.5.1.3. Крок 3: Поділ велико-затратних елементів ПЗ.....	165
3.5.1.4. Крок 4: Систематизація всіх мікрозадач, що потребують серверного розміщення.....	166
3.5.2. Етап 2: Нормалізація шаблонів роботи мікрозадач.....	171
3.5.2.1. Крок 5: Нормалізація шаблону роботи мікросервісу.....	171
3.5.2.2. Крок 6: Візуалізація шаблонів вжитку ресурсу задачами.....	173
3.5.3. Етап 3: Кластеризація шаблонів роботи мікропослуг за ключовим критерієм. Вибір між KMeans і DBSCAN алгоритмами.....	175
3.5.3.1. Вибір між KMeans, DBSCAN алгоритмами та косинусоїдальною подібністю.....	176
3.5.3.2. Крок 7: Пошук подібних шаблонів роботи за ключовим критерієм.....	178
3.5.3.3. Крок 8: Двовимірна візуалізація кластерів часових рядів.....	178
3.5.4. Етап 4: Пошук доповнювальних груп та пар процесорів.....	181
3.5.4.1. Крок 9: Пошук доповнювальних (комплементарних) груп....	181
3.5.4.2. Крок 10: Пошук доповнювальних пар мікросервісів.....	182
3.5.4.3. Крок 11. Повторення ітерацій пошуку.....	186
3.5.4.4. Вибір кількості кластерів.....	187

3.5.4.5. Крок 12: Доповнення неможливі.....	188
3.5.5. Етап 5: Розділення мікропослуг на частини при передбачуваних різких змінах в навантаженні.....	188
3.5.5.1. Крок 13: Розділення базового навантаження та екстремумів для не згрупованих особин.....	188
3.5.5.2. Крок 14: Пошук доповнень для розділених мікропроцесорів	191
3.5.6. Етап 6: Комбінування малогабаритних мікропослуг у потенційно великі набори.....	191
3.5.6.1. Крок 15: Комбінація малогабаритних мікропослуг.....	191
3.5.7. Етап 7: Виділення серверних потужностей під розміри групи та моніторинг.....	192
3.5.7.1. Крок 16: Налаштування під час роботи ПЗ на стороні провайдера.....	192
ВИСНОВКИ.....	193
РОЗДІЛ 4	196
АПРОБАЦІЯ ТА АНАЛІЗ ЕФЕКТИВНОСТІ РОЗРОБЛЕНОГО КОМПЛЕКСНОГО МЕТОДУ ПОШУКУ ДОПОВНЮВАЛЬНИХ НАВАНТАЖЕНЬ.....	196
4.1. Апробація методу пошуку доповнювальних навантажень.....	196
4.1.1. Kubernetes як технологія для реалізації методу доповнювальних навантажень.....	197
4.1.2. Sidecars як помічники в балансуванні навантаження.....	199
4.1.3. Додаткові інструменти для реалізації методу.....	200
4.2. Методи збору вхідних даних.....	201
4.2.1. Моніторинг, збір метрик та накопичення історичних даних за допомогою Prometheus.....	201
4.2.2. Схема застосування інструментів аналізу та моніторингу в поєднанні з Prometheus.....	203
4.3. Мови програмування для написання скриптів.....	206
4.4. Налаштування Kubernetes для виконання алгоритму.....	206
4.5. Вибір вузлів для виконання завдань залежно від ключового критерію.	208

4.5.1. CPU-інтенсивні сервери.....	208
4.5.2. RAM-інтенсивні сервери.....	209
4.5.3. I/O-інтенсивні сервери.....	210
4.5.4. Інші варіації навантажень.....	211
4.5.5. Завантаженість серверів різного типу.....	211
4.6. Реалізація методу пошуку доповнювальних навантажень.....	212
4.6.1. Симуляція вхідних даних.....	213
4.6.2. Підготовка даних та кластеризація, супутня візуалізація результатів..	214
4.6.3. Пошук доповнювальних навантажень.....	214
4.6.4. Розбиття початкових навантажень на мілкіші.....	214
4.6.5. Подальші кроки.....	215
4.7. Оцінка ефективності планувальника доповнювальних навантажень....	215
4.7.1. Опис імітаційного моделювання та експериментального дослідження.....	215
4.7.2. Тестування алгоритму.....	217
4.7.3. Аналіз результатів імітаційного моделювання та експериментального дослідження в розрізі ключового ресурсу.....	219
4.7.3.1. Підвищення ефективності у мережі.....	219
4.7.3.2. Підвищення ефективності у CPU.....	219
4.7.3.3. Підвищення ефективності у RAM.....	219
4.7.3.4. Відмовостійкість системи залежно від навантаження.....	220
4.7.3.5. Удосконалення методу автомасштабування через баланс між максимізацією використання ресурсів та відмовостійкістю системи залежно від навантаження.....	222
4.7.3.6. Удосконалення методу автомасштабування через виділення серверів з виключно доповнювальним навантаженням.....	224
4.7.4. Аналіз результатів імітаційного моделювання та експериментального дослідження в розрізі архітектури задач.....	226
4.7.5. Аналіз зменшення кількості перепланувань.....	228
4.7.6. Аналіз витрат часу при плануваннях та переплануваннях.....	229

4.7.7. Аналіз економії на серверах-планувальниках.....	231
4.7.8. Аналіз витрат та економії матеріального забезпечення й електроенергії в робочих вузлах.....	234
4.8. Вдосконалення сучасного підходу до масштабування.....	236
4.9. Впровадження підходу до планування навантаження на серверах.....	237
4.10. Впровадження та апробація наукових досліджень.....	238
ВИСНОВКИ.....	239
ЗАГАЛЬНІ ВИСНОВКИ.....	241
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	244
Додаток 1. Приклад симуляції даних. У доповнювальні вектори вводиться додаткова похибка.....	255
Додаток 2. Згенеровані доповнення до часових рядів як повноцінне доповнення.....	258
Додаток 3. Кластеризація повноцінних доповнень до часових рядів та звичайних даних.....	259
Додаток 4. Код підготовки даних та їх кластеризації.....	260
Додаток 5. Приклад виводу статистики по кластерах після кластеризації...	262
Додаток 6. Вивід результатів суміщення елементів з доповнювальних кластерів.....	263
Додаток 7. Результати пошуку пар задач з доповнювальними навантаженнями.....	264
Додаток 8. Код розділення навантаження на задачах, які не можуть бути доповнені.....	265
Додаток 9. Результати розділення пікових та середніх навантажень у задачах з навантаженнями, які не можуть бути доповненими.....	269
Додаток 10. Акт впровадження результатів наукового дослідження.....	271

СПИСОК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАЧЕНЬ

АЗ	Апаратне забезпечення
БД	База даних
ВМ	Віртуальна машина
ВПХ	Віртуальною приватною хмарою
ЕОМ	Електронно-обчислювальна машина
ІТ	Інформаційні технології
МДК	Мережа доставляння контенту
ОВ	Обчислювальний вузол
ОЗ	Обчислювальна задача
ОП	Оперативна пам'ять
ОР	Обчислювальний ресурс
ПЗ	Програмне забезпечення
РС	Розподілене середовище
ЦП	Центральний процесор
AWS	Amazon Web Services
CI/CD	Continuous integration and continuous deploy
CPU	Central Processing Unit
GCP	Google Cloud Platform
GKE	Google Kubernetes Engine
MIG	Managed Instance Groups
OSI	Open Systems Interconnection
RAM	Random-Access Memory
VPN	Virtual Private Network
QoS	Quality of Service (якість обслуговування)

ВСТУП

Актуальність теми. Сучасний світ інформаційних технологій характеризується тенденцією до забезпечення безперервного доступу до сервісів для довільної кількості користувачів незалежно від їх географічного розташування. Програмні продукти глобального призначення, такі як системи бронювання авіаквитків, функціонують в умовах непередбачуваних коливань навантаження, що зумовлює необхідність імплементації механізмів масштабування та рівномірного розподілу навантаження.

З метою вирішення зазначених викликів було розроблено хмарні системи (початок 2000-х років), а згодом створено платформу оркестрації контейнерів Kubernetes [1], що забезпечує відповідний функціонал. Дані технології сформували оптимальне середовище для еволюції мікросервісної архітектури, надавши інструментарій для ефективного моделювання та динамічного масштабування мікропослуг у хмарному середовищі.

Мікросервісна архітектура демонструє підвищену ефективність у масштабних програмних комплексах [2], де функціональні компоненти характеризуються варіативним ступенем взаємозалежності. Істотною особливістю користувацької поведінки є нерівномірність використання різних функціональних модулів застосунку, що призводить до диспропорції у розподілі обчислювальних ресурсів.

Мікросервісний підхід уможливорює селективне масштабування, що передбачає розгортання додаткових екземплярів виключно тих компонентів, які зазнають підвищеного навантаження. Паралельно з цим, монолітні застосунки також можуть функціонувати у мультиекземплярному режимі, що сприяє оптимізації розподілу навантаження та підвищує ефективність адміністрування системи. Даний підхід забезпечує економічну доцільність, оскільки точкове масштабування окремих компонентів потребує меншого обсягу ресурсів порівняно з розгортанням додаткових копій цілісного застосунку при використанні балансування навантаження [3]. В останній

період спостерігається інтенсифікація впровадження безсерверних технологій, які характеризуються відсутністю стану та компактністю, що забезпечує їх ефективне масштабування. Усі зазначені технологічні рішення класифікуються як мікропослуги, що є об'єктом дослідження даної роботи.

Для забезпечення надійності хмарної інфраструктури, оновлення серверного обладнання планується відповідно до термінів завершення гарантійного обслуговування. Це створює передумови для максимізації ефективності використання наявних обчислювальних ресурсів протягом повного життєвого циклу обладнання, що є критичним фактором для досягнення економічної рентабельності.

В роботі пропонується метод пошуку доповнювальних навантажень ПЗ з метою формування груп спільної роботи незалежних задач, спрямований на оптимізацію розподілу навантаження між мікросервісами та ефективне використання серверних і мережевих ресурсів хмарної інфраструктури. Цей алгоритм вбудується у стандартні добре відтестовані процеси хмарної системи та допоможе ресурсоефективно розподілити навантаження додатків, щоб використовувалось менше серверів для забезпечення тої самої продуктивності шляхом розташування доповнювальних навантажень разом.

Характерною особливістю серверних застосунків є нерівномірне навантаження, що залежить від специфіки застосунку та часових патернів використання. Передбачуваність цих патернів створює можливість для оптимізації – формування серверних груп або кластерів, де комбінація різних мікросервісів забезпечує сумарне навантаження близьке до повного та сталого. Як результат — підвищення енергетичної та обчислювальної ефективності хмарних систем шляхом впровадження алгоритму пошуку доповнювальних навантажень для роботи на серверних групах.

Зв'язок роботи з науковими програмами, планами, темами.
Робота виконувалась відповідно до планів наукових досліджень кафедри

інформаційних технологій в телекомунікаціях Навчально-наукового інституту телекомунікаційних систем:

В рамках держбюджетної теми «Інтелектуальне керування динамічною енергоефективною реконфігурацією обчислювальних ресурсів для підтримки технології NaaS» (Номер державної реєстрації роботи 0123U101635).

Мета дослідження. підвищення ефективності використання серверної інфраструктури шляхом розподілу навантаження на обчислювальні ресурси з урахуванням активності та властивостей задач.

Завдання, які були вирішені для досягнення поставленої мети:

1. Проаналізувати сучасні підходи до планування, розподілу та балансування навантаження в серверних інфраструктурах.

2. Розробити математичну модель багатосерверної системи для підвищення її ефективності з урахуванням особливостей навантаження задач.

3. Запропонувати метод доповнювальних навантажень для оптимального планування використання ресурсів багатосерверної системи та зниження потреби в масштабуванні.

4. Запропонувати покращення методу горизонтального автомасштабування з метою зменшення його об'ємів.

5. Виконати апробацію методу та оцінити його ефективність у порівнянні з існуючими підходами.

Об'єкт дослідження: процес планування навантаження на вузли у хмарній інфраструктурі.

Предмет дослідження: методи та моделі роботи планувальників у хмарній інфраструктурі.

Методи дослідження:

1. Методи інтелектуального аналізу даних – для аналізу об'єкта дослідження, його формалізації, виділення сутностей та структурних взаємозв'язків у досліджуваній системі.

2. Теорія нечітких ґраток – для опису процесу роботи мікропослуг в багатосерверному середовищі через використання алгебраїчних властивостей нечітких ґраток та їх графового представлення.

3. Методи імітаційного моделювання та лабораторного експерименту – для перевірки ефективності застосування запропонованої математичної моделі та методу.

Наукова новизна отриманих результатів:

1. Запропоновано математичну модель розподілу навантажень на обчислювальні ресурси, яка відрізняється тим, що базується на теорії нечітких ґраток та використовує групи процесів з постійними навантаженнями.

2. Розроблено метод доповнювальних навантажень, який відрізняється тим, що визначає задачі-доповнення, які разом формують групу завдань сталого ресурсоспоживання задля мінімізації сумарної кількості необхідних ресурсів в багатосерверній інфраструктурі.

3. Удосконалено метод автомасштабування виконуваних завдань, який відрізняється від існуючих частковим завантаженням серверів кластера групами задач повного навантаження, що дозволяє зменшити кількість зарезервованого ресурсу цих серверів та загальну кількість серверів у кластері.

4. Удосконалено метод автомасштабування виконуваних завдань, який відрізняється від існуючих повним завантаженням частини серверів кластера групами задач повного навантаження, що дозволяє мінімізувати кількість зарезервованого ресурсу цих серверів, зменшити їх енерговитрати, а також збільшити їх термін служби.

Практичне значення отриманих результатів:

1. В роботі досліджено різноманітні типи обчислювальних задач з погляду придатності до групування заради забезпечення сумарного однакового довготривалого використання ресурсу.

2. На основі методу доповнювальних навантажень побудовано алгоритм планувальника навантажень. Його використання дозволяє:

- зменшити витрати на апаратне забезпечення до 9%;
- зменшити кількість перепланувань розміщення обчислювальних задач на 25–30% в разі всіх доповняльних задач або 7,5–10,5% середньозважений показник, що становить 9–15% та 2,1–4,5% відповідно від задач вузлів-планувальників;
- зменшити потреби в автомасштабуванні на 20–24% та відповідно до відсотку присутності доповнювальних груп на сервері, зменшити кількість запасного ресурсу серверів headroom до досягнення його мінімуму;
- збільшення пропускної можливості багатосерверної системи в години пік, яке складатиме 5–15% економії місця на 10–15% відсотках серверів при виділенні відповідної кількості серверів під доповнювальні групи;
- підвищити енергоефективність послуг хостингу на 10–15% та продовжити строк служби на 8–12% на вузлах повністю з доповнювальним навантаженням.

3. Основні ідеї методу доповнювальних навантажень впроваджені у курс «Big Data», що викладається аспірантам НН ІТС на кафедрі ІТТ.

Особистий внесок здобувача. Дисертаційна робота узагальнює результати теоретичних та експериментальних досліджень, проведених автором самостійно. Основні результати, отримані автором особисто: аналіз підходів до виявлення проблем на різних рівнях роботи мікросервісних застосунків [4–6], обґрунтування вибору нечіткої теорії ґраток для формалізації моделі об'єкта дослідження [7]; метод доповнювальних навантажень та алгоритм його реалізації [7–10], спосіб реалізації методу на серверних групах Kubernetes [11] спосіб роботи мікропослуг, необхідних для роботи алгоритму [12].

Апробація результатів дисертації. Основні положення та результати дисертаційної роботи були представлені й одержали схвалення на:

- Всеукраїнській науково-практичній конференції «Теоретичні і прикладні проблеми фізики, математики та інформатики» (м. Київ, Україна, 2022 р.);
- Міжнародній науково-технічній конференції «Перспективи телекомунікацій» (Київ, Україна, 2023 р.).
- Міжнародній конференції прикладних винаходів та інновацій в IT «ІСАІТ» (Кетен, Німеччина, 2024 р.);
- Міжнародній науково-практичній конференції з програмування «14th International Scientific and Practical Programming Conference (UkrPROG 2024)» (Київ, Україна, 2024 р.);

За результатами досліджень опубліковано 9 наукових публікацій, у тому числі чотири статті у наукових фахових виданнях України за спеціальністю 172 Телекомунікації та радіотехніка в кількості двох співавторів, а також одну статтю у журналі спорідненої спеціальності, дві статті у періодичних наукових фахових виданнях проіндексованих у базах Scopus та CEUR, двоє тез виступів на наукових конференціях.

Структура та обсяг дисертації. Дисертаційна робота складається зі вступу, чотирьох розділів, висновків, списку використаних джерел із 119 найменувань, 10 додатків. Загальний обсяг роботи 272 сторінки, з яких 222 сторінок основного тексту, 10 сторінок використаних джерел та 16 сторінок додатків. Робота містить 40 рисунків, 17 таблиць.

РОЗДІЛ 1

ПРОБЛЕМА КІЛЬКОСТІ РЕСУРСІВ ТА

ЕНЕРГОЕФЕКТИВНОСТІ ОБСЛУГОВУВАННЯ

НАВАНТАЖЕННЯ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНОЇ

СИСТЕМИ

В розділі описується загальна модель роботи хмарної системи та пояснюються принципи її роботи, а саме як відбувається розподіл навантаження та його балансування, масштабування, планування розподілу задач, особливості групування задач, а також аналізується мікросервісна архітектура, як найбільш зручна для масштабування. Найбільша увага приділена планувальникам навантажень на кластери та вузли багатосерверної системи, адже в наступних розділах мова йтиме саме про новий підхід до планування навантаження, який розв'язуватиме проблеми планувальників, описані у розділі.

1.1. Особливості побудови та обслуговування телекомунікаційної мережі

Сучасні телекомунікаційні системи та мережі становлять фундамент інформаційного суспільства, забезпечуючи передачу, обробку та зберігання даних у глобальному масштабі. Розвиток телекомунікаційної інфраструктури відбувається надзвичайно швидкими темпами, що зумовлено постійно зростаючими потребами користувачів у високошвидкісному доступі до інформаційних ресурсів та різноманітних послуг. Особливої актуальності набуває питання ефективного використання наявних ресурсів телекомунікаційних мереж, зокрема серверного обладнання та хмарних технологій, що забезпечують функціонування різноманітних сервісів.

Дана робота присвячена дослідженню особливостей побудови та обслуговування телекомунікаційних мереж з акцентом на оптимізацію використання серверної інфраструктури та хмарних технологій. Особлива

увага приділяється проблемі нерівномірного навантаження на телекомунікаційну інфраструктуру протягом доби та пошуку шляхів її вирішення.

1.1.1. Архітектура сучасних телекомунікаційних мереж

Сучасна телекомунікаційна мережа являє собою складну багаторівневу систему, що складається з різноманітних компонентів: комутаційного обладнання, серверів, систем зберігання даних, програмного забезпечення та каналів зв'язку. Архітектура телекомунікаційної мережі визначає принципи взаємодії цих компонентів та забезпечує надійну та ефективну передачу інформації.

Серверна інфраструктура є ключовим елементом телекомунікаційної мережі, що забезпечує функціонування різноманітних сервісів та обробку даних. Сервери в телекомунікаційних мережах виконують різноманітні функції, як-от обробка та маршрутизація даних, зберігання та архівування інформації, забезпечення роботи інформаційних сервісів, управління мережевими ресурсами, забезпечення безпеки та захисту даних.

Візьмемо для прикладу оператора мобільного зв'язку Київстар, який використовує розгалужену серверну інфраструктуру для надання послуг мобільного зв'язку, інтернету та телебачення. Компанія має кілька потужних обчислювальних центрів, розташованих у різних регіонах України, що забезпечують обробку величезних обсягів даних та надання різноманітних послуг мільйонам абонентів. Ці центри обробки даних об'єднані високошвидкісними оптоволоконними лініями зв'язку, що забезпечує надійну та швидку передачу інформації між ними [13].

Хмарні технології стали невіддільною частиною сучасних телекомунікаційних мереж. Хмара в контексті телекомунікацій являє собою набір серверів для завантаження послуг та розподілу їх між користувачами за запитом, а також забезпечення функціонування додаткових сервісів.

Технічна реалізація хмари в телекомунікаціях включає множину підсистем, що працюють як єдиний комплекс.

Наприклад, оператор телекомунікаційних послуг може використовувати хмарну інфраструктуру для надання своїм абонентам послуг зберігання даних. Абонент може зберігати свої фотографії, відео, документи та інші файли в хмарному сховищі, доступ до якого можливий з будь-якого пристрою, підключеного до мережі інтернет. При цьому фізично ці дані зберігаються на серверах оператора, розташованих у різних дата-центрах, що забезпечує високу надійність зберігання та швидкий доступ до інформації.

Хмарні технології дозволяють телекомунікаційним операторам оптимізувати використання ресурсів, підвищити гнучкість та масштабованість інфраструктури, а також знизити експлуатаційні витрати. Наприклад, за допомогою хмарних технологій оператор може динамічно розподіляти обчислювальні ресурси між різними сервісами залежно від поточного навантаження, що дозволяє ефективно використовувати наявне обладнання та забезпечувати високу якість послуг.

1.1.2. Мережа доставляння контенту в телекомунікаційних мережах

Мережа доставляння контенту (МДК) є важливим компонентом сучасних телекомунікаційних мереж, що забезпечує ефективне доставлення контенту кінцевим користувачам. МДК являє собою географічно розподілену мережу серверів, що розміщуються в різних точках світу з метою забезпечення швидкої та надійного доставляння контенту користувачам. Рисунок 1.0 показує приклад телекомунікаційної мережі, через яку можна доставляти дані.

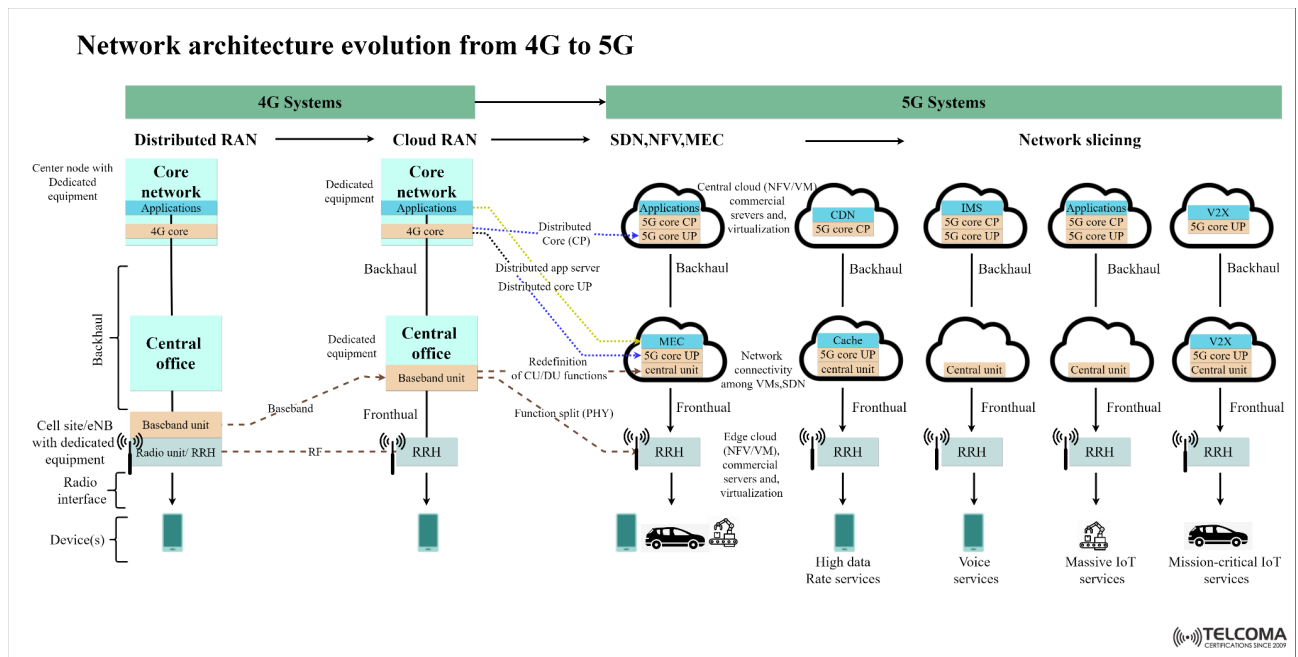


Рисунок 1.0 — Приклад телекомунікаційної системи, організації 4G та 5G [14]

Основний принцип роботи МДК полягає в кешуванні контенту на серверах, розташованих ближче до кінцевих користувачів, що дозволяє зменшити затримки при доступі до контенту та знизити навантаження на основні сервери. Розглянемо, як це працює на прикладі популярного відеосервісу.

Коли користувач із Києва хоче переглянути відеоролик, розміщений на сервері в США, запит спочатку надходить до найближчого сервера МДК, розташованого, наприклад, у Києві або іншому місті України. Якщо відео вже є в кеші цього сервера (тобто, його раніше переглядав інший користувач із цього регіону), воно відразу надається користувачу без необхідності завантаження з віддаленого сервера в США. Це значно прискорює доступ до контенту та покращує досвід користувача.

Якщо ж відео відсутнє в кеші локального сервера, МДК-сервер завантажує його з оригінального джерела (центрального сервера в США), зберігає в своєму кеші та надає користувачу. При наступних запитах на це відео від інших користувачів у тому ж регіоні, воно надається безпосередньо з кешу локального МДК-сервера.

Такий підхід має кілька важливих переваг. По-перше, зменшуються затримки при доступі до контенту, оскільки дані передаються з найближчого сервера, а не з віддаленого джерела. По-друге, знижується навантаження на центральні сервери, оскільки один і той же контент не потрібно завантажувати багато разів. По-третє, економиться трафік між операторами, оскільки оператор телекомунікаційних послуг не платить початковому провайдеру за багаторазове завантаження одного й того ж контенту.

Сучасні МДК активно використовують технології крайових обчислень, що передбачають розміщення обчислювальних ресурсів ближче до кінцевих користувачів. Це дозволяє не лише кешувати контент, але й виконувати певну обробку даних безпосередньо на крайових серверах, що особливо важливо для додатків, чутливих до затримок.

Наприклад, при перегляді потокового відео в реальному часі, крайовий сервер може адаптувати якість відео залежно від швидкості з'єднання користувача та навантаження на мережу. Якщо швидкість з'єднання користувача знижується, сервер може автоматично зменшити якість відео, щоб забезпечити безперервний перегляд без буферизації. Це покращує досвід користувача та знижує навантаження на мережу.

1.1.3. Проблема нерівномірного навантаження на телекомунікаційну інфраструктуру

Однією з ключових проблем сучасних телекомунікаційних мереж є нерівномірність навантаження протягом доби. Типовий графік навантаження має виражені піки в денний час та значне зниження активності вночі, що призводить до неефективного використання ресурсів.

Основними причинами нерівномірного навантаження на телекомунікаційну інфраструктуру є людський фактор, бізнес-процеси, освітні процеси та споживання розважального контенту. Більшість користувачів активно використовують телекомунікаційні послуги в денний

час та значно менше вночі. Корпоративні користувачі генерують значний обсяг трафіку в робочий час. Використання онлайн-платформ для навчання створює значне навантаження в денний час. Пікове споживання розважального контенту припадає на вечірні години.

Наприклад, у типовому українському місті пік навантаження на телекомунікаційну мережу припадає на період з 19 до 22 години, коли більшість людей повертаються з роботи та починають активно використовувати інтернет для розваг, спілкування, перегляду відео тощо. У цей час навантаження на мережу може перевищувати середньодобове в 2–3 рази. Натомість, у період з другої до шостої години ранку активність користувачів мінімальна, і значна частина мережевої інфраструктури простоює.

Нерівномірне навантаження на телекомунікаційну інфраструктуру призводить до ряду негативних наслідків. Серверне обладнання, розраховане на пікові навантаження, простоює в нічний час, що призводить до неефективного використання ресурсів. Виникає необхідність придбання додаткового обладнання для забезпечення пікових навантажень, що збільшує капітальні витрати. Зростають експлуатаційні витрати на електроенергію та обслуговування обладнання, що використовується неефективно. У періоди пікових навантажень можливі перебої в роботі сервісів, що знижує якість послуг.

Наприклад, оператор телекомунікаційних послуг змушений мати серверну інфраструктуру, розраховану на обслуговування пікового навантаження у вечірні години, хоча більшу частину доби ці сервери завантажені лише на 30–40% від своєї потужності. Це призводить до значних витрат на придбання та обслуговування обладнання, яке більшу частину часу використовується неефективно [13].

1.1.4. Стратегії оптимізації використання телекомунікаційної інфраструктури

Для вирішення проблеми нерівномірного навантаження на телекомунікаційну інфраструктуру можуть бути застосовані різні стратегії, спрямовані на стимулювання нічного споживання та оптимізацію використання ресурсів.

Стимулювання нічного споживання телекомунікаційних послуг може здійснюватися різними способами. Одним із найефективніших методів є впровадження диференційованих тарифів, коли вартість послуг у нічний час значно нижча, ніж у денний. Наприклад, оператор мобільного зв'язку може пропонувати своїм абонентам безлімітний доступ до інтернету в період з першої до шостої години ранку за символічну плату або взагалі безкоштовно. Це стимулюватиме користувачів планувати ресурсомісткі операції (завантаження великих файлів, оновлення програмного забезпечення, резервне копіювання даних) на нічний час, що дозволить розвантажити мережу в пікові години.

Іншим ефективним підходом є автоматизоване планування завантажень. Оператор може розробити спеціальні застосунки або функції в існуючих сервісах, які автоматично плануватимуть завантаження великих обсягів даних на нічний час. Наприклад, користувач може вдень вибрати фільм для перегляду, а система автоматично завантажить його вночі, коли навантаження на мережу мінімальне. Вранці користувач матиме доступ до фільму без необхідності його завантаження в пікові години.

Створення спеціальних нічних пропозицій також може стимулювати нічне споживання. Оператор може пропонувати додатковий контент, бонуси або підвищену швидкість доступу для користувачів, які активні вночі. Наприклад, абоненти, які користуються інтернетом у нічний час, можуть отримувати додаткові бали в програмі лояльності, які потім можна обміняти на знижки або додаткові послуги.

Розвиток та популяризація контенту, орієнтованого на нічне споживання, також може сприяти більш рівномірному розподілу навантаження на мережу. Це може включати контент для дорослих, нічні трансляції спортивних подій з інших часових поясів, онлайн-ігри з нічними бонусами тощо. Наприклад, оператор може організувати нічні кіберспортивні турніри з привабливими призами, що заохочуватиме гравців бути активними саме в нічний час.

Для оптимізації використання серверної інфраструктури можуть бути застосовані різні підходи. Один із них – надання обчислювальних ресурсів як послуги. В нічний час, коли навантаження на телекомунікаційну інфраструктуру знижується, вивільнені обчислювальні ресурси можуть бути надані компаніям, зацікавленим у виконанні об'ємних обчислень та обробці даних без прив'язки до конкретного часу.

Наприклад, фармацевтична компанія, що займається розробкою нових ліків, потребує значних обчислювальних ресурсів для моделювання взаємодії різних молекул. Такі обчислення можуть тривати кілька годин або навіть днів, але не вимагають негайного виконання. Компанія може орендувати обчислювальні ресурси у телекомунікаційного оператора в нічний час за зниженою ціною, що вигідно обом сторонам: фармацевтична компанія отримує доступ до потужних обчислювальних ресурсів за меншу ціну, а телекомунікаційний оператор ефективно використовує свою інфраструктуру, яка інакше простоювала б.

Планування фонових процесів на нічний час також є ефективним способом оптимізації використання серверної інфраструктури. Ресурсомісткі фонові процеси, такі як індексація даних, аналітика, резервне копіювання, можуть бути перенесені на нічний час, коли навантаження на мережу мінімальне.

Наприклад, великий інтернет-магазин може проводити аналіз поведінки користувачів та формувати персоналізовані рекомендації в нічний час, коли більшість користувачів неактивні. Це дозволить не

перевантажувати сервери в денний час, коли вони обслуговують запити активних користувачів, і при цьому забезпечити наявність актуальних рекомендацій для користувачів, які відвідують сайт вдень.

Оптимізація трафіку Інтернету речей також може сприяти більш рівномірному розподілу навантаження на мережу. Пристрої Інтернету речей можуть бути налаштовані на синхронізацію з центральними серверами в нічний час, коли навантаження на мережу мінімальне.

Прикладом слугують розумні лічильники електроенергії, води або газу можуть накопичувати дані протягом дня і передавати їх на центральні сервери вночі. Це дозволить розвантажити мережу в денні години та забезпечити більш стабільну роботу критично важливих сервісів.

1.1.5. Конвергенція послуг та розподілена платформа обслуговування

Конвергенція послуг є важливим трендом у розвитку сучасних телекомунікаційних мереж. Вона передбачає об'єднання різних типів послуг (голосовий зв'язок, передача даних, відео) в рамках єдиної мережевої інфраструктури. Одним із проявів конвергенції є концепція «Triple Play», що передбачає надання трьох типів послуг (інтернет, телефонія, телебачення) через єдину мережеву інфраструктуру.

У традиційному підході для кожного типу послуг використовувалася окрема мережева інфраструктура: телефонні лінії для голосового зв'язку, кабельне телебачення для передачі відеоконтенту, окремі лінії для доступу до інтернету. Це призводило до дублювання інфраструктури та неефективного використання ресурсів.

Конвергенція дозволяє об'єднати всі ці послуги в рамках єдиної мережевої інфраструктури, що значно підвищує ефективність використання ресурсів та знижує витрати на розгортання та обслуговування мережі. Наприклад, оператор може надавати послуги інтернету, телефонії та

телебачення через єдину оптоволоконну лінію, що підключається до будинку абонента.

Розглянемо конкретний приклад. Абонент компанії Київстар може отримувати послуги мобільного зв'язку, домашнього інтернету та телебачення від одного оператора. При цьому всі ці послуги інтегровані між собою: абонент може керувати своїм телевізійним контентом через мобільний додаток, переглядати телепрограми на смартфоні, планшеті або комп'ютері, здійснювати дзвінки з домашнього телефону на мобільний і навпаки за пільговими тарифами тощо.

Розподілена платформа обслуговування є ключовим елементом сучасних телекомунікаційних мереж, що забезпечує ефективне надання послуг великій кількості користувачів одночасно. Вона передбачає розподіл навантаження між різними серверами та дата-центрами, що дозволяє забезпечити високу доступність та надійність послуг.

Наприклад, під час трансляції популярної спортивної події, такої як фінал Ліги чемпіонів УЄФА, мільйони користувачів можуть одночасно дивитися онлайн-трансляцію. Щоб забезпечити якісний перегляд для всіх користувачів, оператор використовує розподілену платформу, що складається з багатьох серверів, розташованих у різних дата-центрах. Кожен сервер обслуговує певну групу користувачів, що дозволяє розподілити навантаження та забезпечити стабільну роботу сервісу навіть при пікових навантаженнях.

1.1.6. Дата-центри оператора телекомунікаційних послуг

Дата-центри є ключовим елементом інфраструктури сучасного оператора телекомунікаційних послуг. Вони представляють собою спеціалізовані приміщення, обладнані для розміщення серверів, систем зберігання даних та мережевого обладнання. Дата-центри забезпечують надійне функціонування телекомунікаційних сервісів та обробку великих обсягів даних.

Сучасний дата-центр оператора телекомунікаційних послуг – це складний технічний комплекс, що включає не лише серверне та мережеве обладнання, але й системи електроживлення, охолодження, пожежогасіння, фізичної безпеки тощо. Все це необхідно для забезпечення безперебійної роботи обладнання та захисту даних.

Наприклад, дата-центр компанії Київстар у Києві займає площу кілька тисяч квадратних метрів і містить тисячі серверів, що забезпечують роботу різноманітних сервісів для мільйонів абонентів. Дата-центр обладнаний системами безперебійного електроживлення, що включають дизель-генератори та джерела безперебійного живлення, які забезпечують роботу обладнання навіть у разі тривалого відключення електроенергії. Системи охолодження підтримують оптимальну температуру для роботи обладнання, а системи пожежогасіння забезпечують захист від пожеж.

Хмарні технології тісно пов'язані з дата-центрами і представляють собою модель надання обчислювальних ресурсів на вимогу. Хмара в контексті телекомунікацій – це набір серверів для завантаження послуг та розподілу їх між користувачами за запитом, а також забезпечення функціонування додаткових сервісів [13].

Варіантом реалізації хмарних послуг може бути як спеціалізований сервер у дата-центрі оператора, так і розподілена мережа комп'ютерів користувачів. Наприклад, деякі сервіси розподілених обчислень використовують обчислювальні ресурси комп'ютерів користувачів, коли вони не використовуються активно (наприклад, вночі або коли комп'ютер перебуває в режимі очікування).

Хмара як центр обробки даних забезпечує функціонування різноманітних сервісів, таких як системи управління навчанням (LMS), сервіси зберігання та обробки даних, електронна пошта, онлайн-офіси тощо. Наприклад, популярна система управління навчанням Moodle може бути розгорнута в хмарі оператора телекомунікаційних послуг, що

забезпечить її доступність для студентів та викладачів з будь-якого пристрою, підключеного до інтернету.

1.1.7. Технічні проблеми в телекомунікаційній інфраструктурі

Сучасна телекомунікаційна інфраструктура стикається з рядом технічних проблем, що потребують вирішення для забезпечення ефективної роботи мережі. Однією з таких проблем є масштабування інфраструктури відповідно до зростаючих потреб користувачів.

Масштабування телекомунікаційної інфраструктури може здійснюватися як горизонтально (додавання нових серверів), так і вертикально (збільшення потужності існуючих серверів). Для управління розподіленою інфраструктурою використовуються спеціалізовані системи оркестрації, такі як Kubernetes.

Kubernetes – це система оркестрації контейнерів, що дозволяє автоматизувати розгортання, масштабування та управління додатками в контейнерах. Вона добре працює при горизонтальному масштабуванні, автоматично розподіляючи навантаження між додатковими серверами. Однак при зворотному масштабуванні (зменшенні кількості серверів) можуть виникати проблеми, пов'язані з перерозподілом навантаження та міграцією даних.

Наприклад, якщо оператор телекомунікаційних послуг використовує Kubernetes для управління своєю інфраструктурою, він може легко додати нові сервери для обслуговування зростаючої кількості користувачів. Однак, якщо кількість користувачів зменшується (наприклад, після закінчення популярної онлайн-події), і оператор хоче зменшити кількість активних серверів для економії ресурсів, можуть виникнути проблеми з перенесенням даних та сервісів з серверів, що виводяться з експлуатації.

Іншою проблемою є забезпечення безпеки телекомунікаційної інфраструктури. З розвитком хмарних технологій та розподілених систем зростає кількість потенційних вразливостей, що можуть бути використані

зловмисниками. Забезпечення безпеки вимагає комплексного підходу, що включає захист від несанкціонованого доступу, шифрування даних, моніторинг безпеки тощо.

Наприклад, оператор телекомунікаційних послуг повинен забезпечити захист персональних даних своїх абонентів, що зберігаються в його дата-центрах. Це включає як технічні заходи (шифрування даних, контроль доступу, моніторинг безпеки), так і організаційні (навчання персоналу, розробка політик безпеки, аудит безпеки).

1.1.8. Програмне забезпечення як послуга для телекомунікаційних сервісів

Програмне забезпечення як послуга (Software as a Service, SaaS) є моделлю надання програмного забезпечення, при якій постачальник розробляє веб-додаток, розміщує його на своїх серверах і надає доступ до нього через інтернет. Ця модель дозволяє користувачам отримувати доступ до програмного забезпечення без необхідності його встановлення та обслуговування на власних комп'ютерах.

У контексті телекомунікаційних сервісів SaaS може бути використаний для надання різноманітних послуг, таких як системи управління взаємовідносинами з клієнтами (CRM), системи білінгу, аналітичні платформи тощо. Одним із перспективних напрямків розвитку SaaS у телекомунікаціях є розробка систем для збалансованого розподілення навантаження в телекомунікаційній системі.

1.1.9. Особливості побудови та обслуговування телекомунікаційної мережі

Ідея такої системи полягає в автоматизації процесів розподілу навантаження між різними компонентами телекомунікаційної мережі для забезпечення оптимального використання ресурсів. Система може аналізувати поточне навантаження на різні компоненти мережі, прогнозувати майбутнє навантаження на основі історичних даних та

автоматично перерозподіляти ресурси для забезпечення стабільної роботи мережі.

Наприклад, система може виявити, що певний сервер перевантажений, і автоматично перенаправити частину трафіку на інші, менш завантажені сервери. Або, навпаки, система може виявити, що кілька серверів працюють з низьким навантаженням, і об'єднати їх навантаження на одному сервері, а інші перевести в режим очікування для економії електроенергії.

Така система може бути особливо корисною для оптимізації використання ресурсів у нічний час, коли загальне навантаження на мережу знижується. Вона може автоматично виявляти періоди низького навантаження та перерозподіляти ресурси для виконання фонових завдань, таких як резервне копіювання даних, оновлення програмного забезпечення, аналітика тощо.

1.1.10. Інтеграція з іншими операторами та провайдерами

Сучасні телекомунікаційні мережі не існують ізольовано, вони взаємодіють з мережами інших операторів та провайдерів для забезпечення глобальної зв'язності. Інтеграція з іншими операторами та провайдерами є важливим аспектом побудови та обслуговування телекомунікаційної мережі.

Інтеграція може здійснюватися на різних рівнях: фізичному (з'єднання мереж), логічному (маршрутизація трафіку), сервісному (надання послуг). На фізичному рівні інтеграція здійснюється через точки обміну трафіком (Internet Exchange Points, IXP), де мережі різних операторів фізично з'єднуються для обміну трафіком.

В Україні існує кілька великих точок обміну трафіком, таких як UA-IX у Києві, що забезпечують обмін трафіком між українськими операторами та провайдерами. Це дозволяє забезпечити швидку та ефективну передачу даних між різними мережами всередині країни, без необхідності використання міжнародних каналів зв'язку.

На логічному рівні інтеграція здійснюється через протоколи маршрутизації, такі як BGP (Border Gateway Protocol), що дозволяють обмінюватися інформацією про маршрути та забезпечувати оптимальну маршрутизацію трафіку між різними мережами.

Наприклад, коли користувач одного оператора хоче отримати доступ до ресурсу, розміщеного в мережі іншого оператора, протокол BGP забезпечує визначення оптимального маршруту для передачі даних між цими мережами. Це дозволяє забезпечити швидкий та надійний доступ до ресурсів, розміщених у різних мережах.

На сервісному рівні інтеграція може включати угоди про роумінг, що дозволяють користувачам одного оператора отримувати послуги в мережі іншого оператора. Наприклад, абонент українського оператора мобільного зв'язку, перебуваючи за кордоном, може користуватися послугами місцевого оператора завдяки угоді про роумінг між цими операторами.

1.1.11. Майбутнє телекомунікаційних мереж

Майбутнє телекомунікаційних мереж пов'язане з розвитком нових технологій та підходів до побудови та обслуговування мереж. Одним із ключових трендів є віртуалізація мережевих функцій (Network Functions Virtualization, NFV), що передбачає заміну апаратних мережевих пристроїв програмними реалізаціями, що працюють на стандартних серверах.

Віртуалізація мережевих функцій дозволяє значно підвищити гнучкість та ефективність використання ресурсів телекомунікаційної мережі. Замість використання спеціалізованого апаратного забезпечення для кожної мережевої функції (маршрутизатори, комутатори, брандмауери тощо), оператор може використовувати стандартні сервери, на яких працюють віртуалізовані мережеві функції.

Наприклад, замість встановлення фізичного маршрутизатора для кожного нового клієнта або сервісу, оператор може створити віртуальний маршрутизатор на існуючому сервері. Це дозволяє значно знизити витрати

на обладнання, прискорити розгортання нових сервісів та підвищити ефективність використання ресурсів.

Іншим важливим трендом є програмно-конфігуровані мережі (Software-Defined Networking, SDN), що передбачають відокремлення площини управління від площини даних у мережевих пристроях. Це дозволяє централізувати управління мережею та автоматизувати багато процесів, що раніше вимагали ручного втручання.

Наприклад, у традиційній мережі для зміни маршрутизації трафіку необхідно налаштовувати кожен маршрутизатор окремо. У програмно-конфігурованій мережі всі зміни можуть бути здійснені централізовано через контролер SDN, що значно спрощує управління мережею та підвищує її гнучкість.

Розвиток технологій 5G та Інтернету речей також матиме значний вплив на майбутнє телекомунікаційних мереж. Технологія 5G забезпечує значно вищу швидкість передачі даних, меншу затримку та можливість підключення значно більшої кількості пристроїв порівняно з попередніми поколіннями мобільного зв'язку (Рисунок 1.0).

Це відкриває нові можливості для розвитку різноманітних сервісів, таких як автономні транспортні засоби, розумні міста, телемедицина тощо. Наприклад, завдяки низькій затримці та високій надійності 5G, хірург може проводити операцію, керуючи роботом-хірургом, що знаходиться в іншому місті або навіть країні [14].

Інтернет речей передбачає підключення до інтернету різноманітних пристроїв, від побутової техніки до промислового обладнання, що дозволяє збирати дані та керувати цими пристроями віддалено. Це створює нові виклики для телекомунікаційних мереж, пов'язані з необхідністю обробки та передачі величезних обсягів даних від мільярдів пристроїв.

Для вирішення цих викликів розробляються нові підходи до побудови та обслуговування телекомунікаційних мереж, такі як крайові обчислення (Edge Computing), що передбачають розміщення обчислювальних ресурсів

ближче до джерел даних для зменшення затримок та навантаження на центральні сервери.

Сучасні телекомунікаційні мережі є складними системами, що забезпечують передачу, обробку та зберігання даних у глобальному масштабі. Ефективна побудова та обслуговування таких мереж вимагає врахування багатьох факторів, від технічних аспектів до економічних та соціальних.

Одним із ключових викликів є нерівномірність навантаження на телекомунікаційну інфраструктуру протягом доби, що призводить до неефективного використання ресурсів. Для розв'язання цієї проблеми можуть бути застосовані різні стратегії, спрямовані на стимулювання нічного споживання та оптимізацію використання ресурсів.

Розвиток нових технологій, таких як віртуалізація мережевих функцій, програмно-конфігуровані мережі, 5G та Інтернет речей, відкриває нові можливості для підвищення ефективності та гнучкості телекомунікаційних мереж. Однак, це також створює нові виклики, пов'язані з необхідністю обробки та передачі величезних обсягів даних, забезпечення безпеки та приватності, а також інтеграції з іншими системами та мережами [14].

Успішне вирішення цих викликів вимагає комплексного підходу, що враховує як технічні, так і економічні, соціальні та екологічні аспекти. Тільки такий підхід дозволить забезпечити ефективну побудову та обслуговування телекомунікаційних мереж, що відповідають потребам сучасного інформаційного суспільства.

1.2. Хмарна інфраструктура та класичні послуги хостингу

Розподілене середовище (РС) – віртуальний обчислювальний простір, який може обмежуватися однією розподіленою системою або містити кілька розподілених систем, які взаємодіють між собою. Такий віртуальний обчислювальний простір, який також може бути хмарию, надається

користувачеві у вигляді систематизованого сховища інформаційних та програмних ресурсів, має певну структуру, зрозумілу систему адресації ресурсів та певні моделі обчислювальних процесів або бізнес-процесів цього користувача, які є проблемно-орієнтованими, відповідають певним видам робіт. Користувач звертається до бізнес-процесів, які, своєю чергою, отримують необхідні ресурси для своєї роботи [15].

Хмарна інфраструктура – це розподілене віртуальне середовище, яке забезпечує обчислювальні ресурси, зберігання даних, мережеві можливості та інші IT-послуги через Інтернет. Вона складається з обчислювальних ресурсів (сервери, віртуальні машини, контейнеризація), систем зберігання даних (S3, Google Cloud Storage тощо), мережевої інфраструктури (віртуальні мережі, балансувальники навантаження), а також інструментів моніторингу, автоматизації та управління.

Хмари дозволяють масштабувати ресурси під потреби бізнесу (горизонтально або вертикально), що підтримує існування високонавантажених застосунків та ПЗ зі змінним попитом, мінімізуючи затримки в час пікових навантажень. Окрім того, оплата за оренду АЗ здійснюється по факту його використання. Внаслідок широкої поширеності, хмарні технології даються можливість розподіляти дані між кількома датацентрами, підтримуючи резервне копіювання та аварійне відновлення (Disaster Recovery), а також запускати застосунки в кількох географічних регіонах одночасно для зменшення затримок і підвищення продуктивності [16].

Програмне забезпечення, що розробляється ключовими хмарними провайдерами, на високому рівні, оптимізоване бути швидким та ефективним, дозволяє інтеграцію з інструментами неперервної інтеграції та впровадження CI/CD для автоматизації розгортання та тестування. Через множинні розподілені обчислення, хмари підходять для аналізу великих обсягів даних завдяки сервісам типу BigQuery, Redshift, Hadoop.

Отже, хостинг на хмарі дуже зручний в багатьох випадках окрім коли застосунок постійно використовує значні обсяги ресурсів; тоді власна інфраструктура може бути дешевшою.

В цьому пункті надано інформацію про сучасний стан послуг класичного та хмарного хостингу, які типи серверів використовуються та який формат програмного забезпечення найбільш зручний для роботи на хмарній інфраструктурі.

1.2.1. Класичні або традиційні послуги хостингу

Існують різні рішення для обслуговування застосунків, що потребують хостингу. Найпростіші — це виділений сервер, який орендується у компанії. Також можна орендувати віртуальний приватний сервер або ж частину дата-центру, де розмістити своє обладнання. Ці рішення підходять для додатків, що вимагають високої продуктивності й надійного доступу до апаратних ресурсів.

Обмеженнями класичного хостингу є або відсутність або ручний режим використання зручних та дуже потрібних для багатокористувацьких застосунків функцій масштабованості, розподілу навантаження та відновлення після збоїв, які притаманні хмарним провайдерам. Невеликий хостинг якщо навіть займеться організацією додаткового програмного забезпечення для забезпечення високої доступності (High Availability) та можливості масштабування зокрема, але це рішення навряд буде універсальним. Однак, не слід відкидати цю опцію і пропонуючи легке та ефективне рішення, можна розраховувати на його імплементацію.

1.2.2. Хмарні або клауд послуги хостингу

Рішенням для забезпечення високої доступності (High Availability) є хмарна технологія. Вона являє собою модель обчислень, яка дозволяє надавати ресурси (сервери, сховища, бази даних, мережі, програмне забезпечення) як сервіси через Інтернет. У хмарних обчисленнях ресурси можуть автоматично масштабуватися й розподілятися на вимогу,

забезпечуючи більшу гнучкість, доступність та ефективність порівняно зі звичайним хостингом.

Окрім вищезгаданого, багато користувачів переходять на хмарну технологію через її швидкодію у регіонах, віддалених від серверної кімнати, адже у клауд-провайдера можуть бути свої дата-центри та кращий інтернет-зв'язок у потрібному регіоні, через які відбуватиметься комунікація. Глобально, це дозволяє розподіляти дані й обчислення між різними географічними регіонами, покращуючи продуктивність і скорочуючи затримки для користувачів у різних частинах світу.

Щобільше, якщо один компонент виходить з ладу, інші автоматично підхоплюють навантаження, мінімізуючи простої. Якщо ж єдиний сервер класичного хостингу виходить з ладу, додаток або сайт перестав працювати до усунення проблеми. Додатково можна легко налаштувати резервне копіювання та відновлення після збоїв

Постачальники хмарних послуг постійно розширюють набір пропонуваних функцій, яких нема на звичайних хостингах, та дають можливість скористатись все більше популярним машинним навчанням (для автоматичного кастомізованого пропонування товарів), аналітикою великих даних (наприклад, для аналізу якості реклами залежно від категорії глядачів), базами даних, системою безпеки тощо.

Таким чином, застосунки зі змінним навантаженням, що вимагають широкої доступності надійніше, зручніше та дешевше хостити на хмарах, аніж в одному чи декількох звичайних дата-центрах.

Також існує гетерогенна хмара — це хмарне середовище, яке використовує ресурси або сервіси з різних типів хмарних платформ і постачальників, об'єднуючи їх в одну інтегровану інфраструктуру [15]. Це може бути комбінація публічних і приватних хмар, або навіть кількох публічних хмар від різних постачальників (наприклад, Amazon Web Service (AWS), Google Cloud, Microsoft Azure). Плюсом є можливість обирати найкращі сервіси від різних постачальників для кожної задачі, та не залежати від певного провайдера. В той самий час слід зважати на

необхідність контролювати ресурси з різних платформ, що може ускладнити управління.

1.2.3. Види серверів у дата-центрах

У сучасних хмарних дата-центрах значна частка серверних рішень, а саме ~80–90%, припадає на високощільні, модульні, та оптимізовані для масштабованості технології. Така статистика пов'язана з високою щільністю розміщення обладнання, а отже меншим необхідним простором. Також, такі сервери вже оптимізовані з погляду енерговитрат. Вони мають вбудовану можливість горизонтального масштабування, адже в процесі роботи легко додавати або замінювати елементи при горизонтальному масштабуванні [17]. Тому вони підходять для сучасних технологій, таких як контейнеризація (Kubernetes, Docker) та мікросервіси. Додатково, на щільних серверах достатньо просто організувати інтеграцію з DevOps інструментами для автоматизації управління інфраструктурою, такими як Terraform та Ansible.

Різновидом таких серверів є блейд-сервери – тип високощільних серверів, де окремі модулі (блейди) вставляються у спільний корпус.

Іншим варіантом є гіперконвергована інфраструктура (HCI) – поняття, коли сервери об'єднують обчислювальні, мережеві функції та сховище в одному компактному рішенні [18]. Приклад такої структури наведено на Рисунку 1.1.

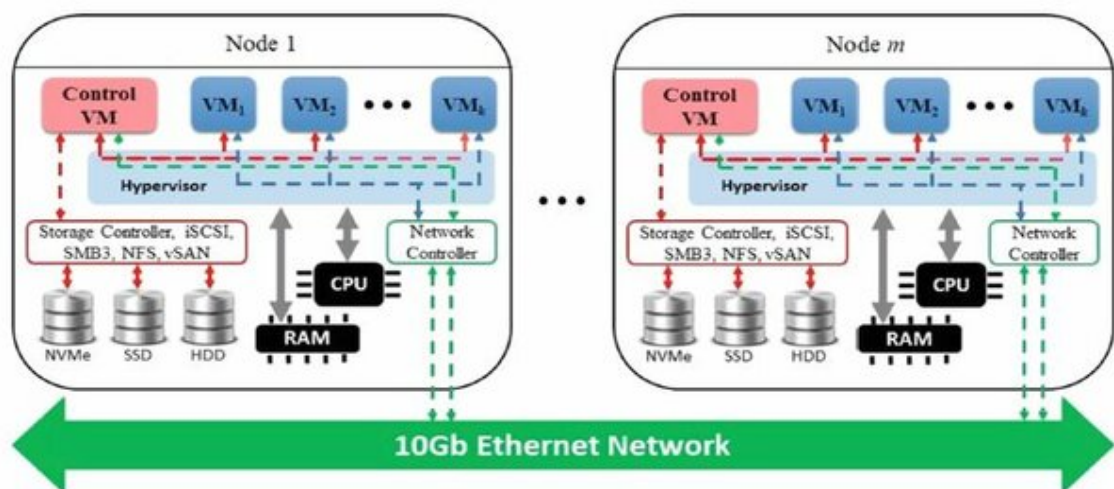


Рисунок 1.1 — Структура вузлів гіперконвергованої інфраструктури (HCI) [18]

За заздалегідь зрозумілою необхідністю масштабуватись, можна використати Scale-Out Servers, тобто сервери, які мають здатність масштабуватися горизонтально, зберігаючи високу щільність. На рисунку 1.2 показана різниця між масштабуванням за моделлю суперкомп'ютера (Scale Up) та за моделлю Scale Out сервера. В першому випадку збільшується об'єм кожного ресурсу, і відповідно, здорожується кожен ресурс, а в другому — кожен ресурс не є дуже дорогим, але таких багато. При цьому, використання електроенергії буде меншим або «набірним» у другому випадку [19].

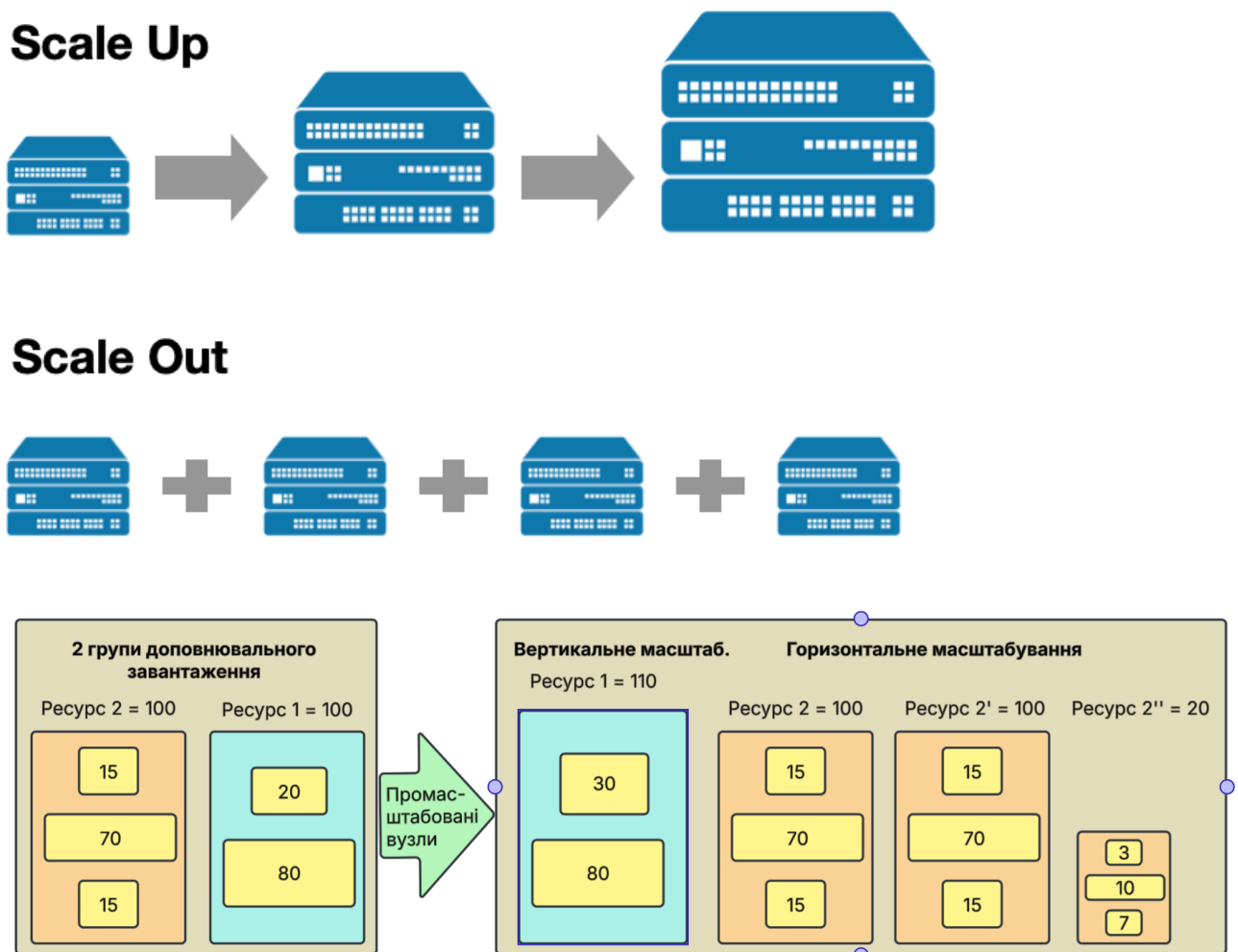


Рисунок 1.2 — Зверху: порівняння розширення ресурсів в сторону збільшення кожного (Scale Up) та в сторону збільшення модулів ресурсів (Scale Out) [19]. Знизу: приклад вертикального та горизонтального масштабування груп мікросервісів

Стандартні ЕОМ практично не використовуються у хмарних дата-центрах через їх низьку ефективність для масштабованих і розподілених навантажень. Їх відсоткова частка складає ~10–20%, та використовують вони здебільшого в приватних дата-центрах або компаніях, які ще не перейшли на хмарну архітектуру, де спостерігаються обмежені потреби в масштабуванні. Також, вони можуть бути застосовані як спеціалізовані рішення, наприклад для наукових досліджень або промислових систем. В підрозділах представлено основні варіації для хостингу як монолітів, так і мікросервісів.

Нижче розглянуто два варіанти: одиничний ресурс та віртуальна приватна хмара.

Одиничним ресурсом є один сервер, призначений для хостингу веб-додатків, обробки великих обсягів даних, запуску серверів баз даних, виконання аналітичних задач, тестування та розробки додатків або ж використання у високопродуктивних обчисленнях (HPC). У кожного провайдера є певні особливості своїх ресурсів.

Для прикладу, візьмемо EC2 інстанс (Amazon Elastic Compute Cloud Instance). Це віртуальний сервер у хмарному середовищі AWS, який надає можливість запускати різноманітні застосунки. Інстанси EC2 забезпечують масштабовані обчислювальні потужності та дозволяють користувачам швидко налаштовувати, запускати й керувати віртуальними машинами для хостингу додатків, обробки даних або виконання інших завдань.

На EC2 можна встановлювати обрану операційну систему та ПЗ, запускати в різних (географічних) зонах доступності, масштабувати та автоматично додавати або видаляти відповідно до навантаження завдяки інтеграції з іншими сервісами AWS, такими як Auto Scaling і Elastic Load Balancing.

Існує декілька типів EC2 інстансів залежно від оптимізаційних завдань:

- загального призначення – для балансування процесорних та ресурсів пам'яті;
- інстанси з оптимізацією для обчислень (підходять для високопродуктивних обчислень);
- інстанси для зберігання даних;
- інстанси з оптимізацією для машинного навчання тощо.

Віртуальні сервери на Google Cloud (Google Compute Engine) і Microsoft Azure (Azure Virtual Machines) виконують подібні функції до Amazon EC2, але мають свої особливості та відмінності залежно від платформи.

Google Cloud надає інстанси різних типів, включаючи інстанси загального призначення, оптимізовані для пам'яті або обчислень, та інстанси для графічних обчислень (GPU). Google дозволяє користувачам створювати налаштовані конфігурації, вибираючи бажану кількість віртуальних процесорів (vCPU) і обсяг оперативної пам'яті, що може бути гнучкішим у порівнянні з фіксованими конфігураціями.

Microsoft Azure пропонує широкий вибір типів інстансів, включаючи для загального використання, обчислювальні, з високою кількістю пам'яті, GPU та для завдань штучного інтелекту (AI). Azure підтримує масштабовані конфігурації та надає спеціальні типи інстансів для Windows Server і SAP.

Віртуальною приватною хмарою (ВПХ) є налаштований пул спільних обчислювальних ресурсів, доступний у публічному хмарному середовищі, що забезпечує певний рівень ізоляції між різними організаціями, тобто, користувачами цих ресурсів. Ізоляція для конкретного користувача ВПХ від інших користувачів тієї ж хмари досягається за допомогою виділення приватної IP-підмережі та створення віртуальних каналів зв'язку, таких як, VLAN або зашифрованих з'єднань для кожного користувача. У ВПХ цей ізоляційний механізм доповнюється функцією VPN, яка також виділена окремо для кожного користувача та забезпечує безпечний віддалений доступ організації до її ресурсів у хмарі завдяки

аутентифікації та шифруванню. Завдяки цим рівням ізоляції організація, яка використовує таку послугу, отримує віртуально приватне хмарне середовище, що і визначає назву «віртуальна приватна хмара».

ВПХ може охоплювати декілька зон доступності в одному регіоні. Це є стандартною практикою для забезпечення високої доступності та стійкості до збоїв. Додаткове розділення можна зробити створюючи підмережі для кожної зони в межах одного ВПХ. Це дозволяє налаштовувати ізольовані середовища та керувати мережевими політиками на рівні зони. З метою додаткового рівня захисту можна налаштувати мережеві ACL та групи безпеки для підмереж у різних зонах.

Бонусом є те, що трафік між зонами в межах одного ВПХ є внутрішнім і зазвичай має низькі затримки, що дозволяє швидко передавати дані між інстансами у різних зонах без необхідності виходу у відкритий інтернет.

1.2.4. Форма ПЗ для хмарних технологій

На хмарному провайдері можна запускати довільне програмне забезпечення, але у кожному випадку є свої особливості, зокрема, зважаючи на потребу в масштабуванні. Порівняння трьох основних форм представлення програмного забезпечення з позиції хмарного провайдера наведено в таблиці 1.1.

Таблиця 1.1 Порівняння безсерверного ПЗ з ПЗ без та з внутрішнім станом

Критерій	Serverless (безсерверне ПЗ)	Stateless (ПЗ без внутрішнього стану)	Stateful (ПЗ з внутрішнім станом)
Концепція	Модель виконання, де немає необхідності управляти інфраструктурою. Розробники пишуть	Застосунки, що не зберігають стан між запитами. Кожен запит	Застосунки, які зберігають стан між запитами або сесіями.

	код, а хмара запускає та масштабує його за запитом.	обробляється незалежно.	
Інфраструктура	Управляється хмарним провайдером (AWS Lambda, Azure Functions).	Може працювати де завгодно: сервери, контейнери, хмара чи локально.	Зазвичай потребує серверів, баз даних або іншого сховища для підтримки стану.
Підтримка стану	Може бути як стейтлес (без збереження стану), так і стейтфул (використання зовнішніх сховищ).	Завжди стейтлес: немає локального збереження між викликами.	Локально або за допомогою зовнішніх механізмів (сховища, бази даних).
Складність управління	Низька: провайдер автоматизує більшість операцій.	Середня: потребує налаштування інфраструктури.	Висока: вимагає комплексного управління станом і збереженням.
Масштабування	Автоматичне, провайдер керує підвищенням чи зниженням потужності.	Легко масштабується за рахунок незалежності від стану.	Важке: зберігання стану ускладнює масштабування.
Приклади	AWS Lambda, Google Cloud Functions, Azure Functions.	Модульні REST API застосунки, мікросервіси без стану.	Банківські системи, онлайн-магазини та ін.

1.3. Особливості розподілу навантаження та балансування в сучасних інформаційно-комунікаційних мережах

Щоб зрозуміти чинні дослідження в цій галузі, проведено базовий огляд найбільш популярних проєктів оптимізації використання ресурсів

мікросервісів. Головними аспектами, які розглядаються в цьому підрозділі є планування розподілу та балансування навантаження як для окремого елемента ПЗ, так і для всієї системи. Порівняння балансувальників основних постачальників хмарних послуг AWS, GCP та Azure наведено в таблиці 1.3.

1.3.1. Критерії розподілу навантаження в Kubernetes

У документації Kubernetes зазначено, що існує багато факторів для визначення найбільш доцільного сервера для запуску нового екземпляра. Основними з них є індивідуальні та колективні вимоги до ресурсів, апаратні/програмні/політичні обмеження, специфікації спорідненості або прив'язаності (affinity) та анти-спорідненості (anti-affinity), локальність даних та взаємні перешкоди робочих навантажень [20]. Ці підходи базуються на фізичних можливостях та найкращій можливій швидкості взаємодії. Це не включає оптимальне рішення з економічного чи екологічного погляду, які сьогодні мають велике значення.

1.3.2. Балансування навантаження в Amazon Web Service

Amazon Web Service (AWS) має еластичний балансувальник навантаження, який включає три взаємосуперечливі елементи, розглянуті у підпунктах.

Взагалі, на AWS підходи розподілу навантаження сформовані для забезпечення справедливих розподілу ресурсів, тим самим продовжуючи довговічність роботи елементів системи [12]. Ресурси вимикаються, як тільки вони більше не потрібні, що запобігає марному споживанню енергії. Проте, в документації з управління хмарою немає інформації про те, як зменшити кількість необхідних серверів для оптимізації активності хмари. Методи, запропоновані в дисертації, могли б допомогти досягти цього.

Google Cloud [11] пропонує подібну функціональність.

Розглянемо три види балансувальників:

- балансування навантаження додатків;

- юалансування мережевого навантаження;
- класичне балансування на мережевому та протокольному рівнях.

Перший вид балансувальника в ELB – це *балансувальник навантаження додатків* (ALB) працює на 7-мому рівні OSI (рівень прикладних протоколів). Він підтримує маршрутизацію на основі хоста та шляху, що означає, що трафік спрямовується на основі його вмісту або його заголовків, доменного імені.

Групування запитів здійснюється за близькими локаціями. Можна налаштовувати правила маршрутизації, щоб направляти трафік до різних груп цілей (Target Groups), зокрема до різних мікросервісів або інстансів. Внаслідок підтримки ALB WebSocket і HTTP/2 протоколів, можна задовольнити потребу постійного з'єднання для передачі даних у реальному часі, яка є актуальною для деяких сучасних додатків, наприклад, стрімінгу відео чи аудіо. Підтримка липучої сесії (Sticky sessions) виконується за допомогою кукісів, що дозволяє прив'язувати користувачів до певного цільового ресурсу на час сесії.

Для коректної роботи, а також балансування навантаження використовуються інтелектуальні перевірки стану (Health Checks). ALB може проводити розширені перевірки стану для кожного Target Group, дозволяючи виключати з балансування непридатні інстанси. Також, є підтримка інтеграції з AWS CloudWatch для моніторингу та ведення журналів Access Logs, що дозволяє відстежувати та аналізувати статистику трафіку, продуктивність роботи та безпечність.

ALB характеризується автоматичною масштабованістю відповідно до навантаження, забезпечуючи високу доступність і стійкість до великих обсягів трафіку шляхом підтримки інтеграції з Auto Scaling, що дозволяє автоматично збільшувати або зменшувати кількість інстансів.

Такий підхід до балансування навантаження, який також суміщений з розширенням підходить для:

- веб-додатків з динамічним контентом і високими вимогами до маршрутизації запитів;
- для мікросервісних архітектур і розподілених систем, де важливо маршрутизувати трафік залежно від змісту запитів;
- для додатків у реальному часі, які потребують WebSocket підтримки;
- для систем із високими вимогами до безпеки, які можуть використовувати WAF і SSL.

Другий вид балансувальника в ELB – це *балансувальник мережевого навантаження* (NLB), яке відбувається на 4-тому рівні OSI, тобто на рівні мережі, а отже забезпечує низьку затримку і високу пропускну здатність. Його метою є запобігання виснаженню з'єднання перед тим, як ресурс вважатиметься нездоровим, і рівномірний розподіл трафіку в режимі крос-зони (Cross-Zone Load Balancing), що сприяє оптимальному використанню ресурсів.

Режим Cross-Zone Load Balancing дозволяє балансувати навантаження між інстансами або іншими ресурсами в усіх зонах доступності (Availability Zones), незалежно від того, звідки надходить трафік. Без cross-zone mode NLB розподіляє вхідний трафік лише на ресурси в тій самій зоні доступності, де було отримано запит. Це може створити ситуацію, коли в одній зоні ресурси можуть бути перевантажені, а в іншій — недоавантажені. З cross-zone mode NLB рівномірно розподіляє трафік між усіма доступними ресурсами в усіх зонах, незалежно від того, в якій зоні був отриманий запит. Це допомагає досягти більш рівномірного розподілу навантаження. Таким чином, запити з однієї зони можуть бути обслужені в іншій, в разі перевантаження спорідненої зони.

Такий спосіб балансування навантаження допомагає уникнути перевантаження ресурсів в одній зоні, коли інші зони недовантажені, а також забезпечує високу доступність (high availability), дозволяючи підтримувати продуктивність додатків навіть при нерівномірному трафіку по зонах.

Іншою особливістю NLB є алгоритм розподілу на основі потоку (flow-based distribution) використовується для того, щоб зробити балансування мережевого навантаження особливо придатним для додатків, які працюють з передбачуваною та послідовною обробкою з'єднань, та мають вимоги до збереження стану (stateful). Для stateful додатків потрібно, щоб весь потік даних оброблявся одним цільовим ресурсом (наприклад, для сесійних з'єднань). Таким чином, NLB дозволяє підтримку поняття липучої сесії (Sticky Session), що дозволяє прив'язувати певних користувачів до одних і тих самих інстансів на час сесії.

Консистентність розподілу, тобто весь трафік одного потоку спрямовується на один і той самий цільовий ресурс, поки сеанс активний, забезпечується через спрямування кожного TCP або UDP потоку на один конкретний цільовий ресурс протягом усього сеансу. TCP або UDP потоку визначається парою «IP-адреса джерела:порт» і «IP-адреса призначення:порт», якій своєю чергою ставиться у відповідність значення хешу для швидкого пошуку цільового ресурсу, тому однакові потоки завжди направляються до того ж цільового ресурсу. При виході з ладу одного цільового ресурсу нові потоки автоматично перенаправляються на інші активні ресурси.

Третій вид балансувальника в ELB — це часткова комбінація двох вищезгаданих балансувальників, яка називається *класичним балансувальником навантаження* [10], [11]. Він працює на 4-му (TCP) рівні прикладного трафіку та 7-му (HTTP/HTTPS) рівні OSI моделі — рівні мережевих протоколів. Цей вид балансувальника був першим, та добре справляється з рівномірним розподілом навантаження. Однак, він з часом показав, що більш підходить до невеликих систем, тому вважається не універсальним та застарілим зокрема через відсутність функцій інтелектуальної маршрутизації, яка працює на основі параметрів запиту або детальної інспекції HTTP-заголовків, як працює ALB.

1.3.3. Балансування на Google Cloud

У Google Cloud (GC) є кілька видів балансування навантаження, що підходять для різних типів додатків та архітектур. Google Cloud Load Balancing надає глобальні та регіональні балансувальники, які працюють на різних рівнях мережевої моделі. На додачу, всі балансувальники автоматично масштабуються відповідно до навантаження, забезпечуючи високу доступність.

Ці балансувальники виконують по суті ті самі функції, що і балансувальники на AWS, але вони більш точкові. Водночас їх можна комбінувати між собою для забезпечення більш витонченого балансування з врахуванням особливостей продукту.

Деякі балансувальники на GC працюють глобально, тобто в кількох регіонах одночасно та можуть розподіляти трафік між серверами або інстансами в різних географічних локаціях (регіонах) на основі певних критеріїв. На відміну від регіональних балансувальників, які обмежені одним конкретним регіоном, глобальні балансувальники забезпечують єдину точку доступу для клієнтів по всьому світу, перенаправляючи їх запити до найближчого або найменш завантаженого регіону. Це дозволяє оптимізувати роботу додатків і зменшити затримку. Порівняння балансувальників наведено в підрозділах, а також у таблиці 1.2.

Розгляньмо глобальні балансувальники та регіональні балансувальники.

Глобальні балансувальники наступні:

1. *HTTP(S) Load Balancing* – розподіляє HTTP та HTTPS трафік та дозволяє маршрутизувати його до найближчих доступних ресурсів у різних регіонах (за географічним принципом). Дана функція схожа з NLB з AWS.

В той самий час, даний балансувальник підтримує SSL термінацію (розшифровку даних на балансувальниках, а не основних серверах для зменшення навантаження на останні), HTTP/2, WebSocket, а також інтегрується з Google Cloud Armor для захисту від DDoS-атак. Він здійснює

підтримку інтелектуальної маршрутизації, включаючи маршрутизацію на основі контенту (залежно від шляху, домену чи заголовків). Таким чином, він найбільше підходить для веб-додатків і API, які потребують глобального охоплення і високої доступності, що схоже на ALB з AWS.

2. *SSL Proxy Load Balancing* – дозволяє розподіляти зашифрований трафік через SSL/TLS на рівні балансувальника, надсилаючи трафік до ресурсів у незашифрованому вигляді. Він підходить для додатків, які потребують захищеного з'єднання, але можуть передавати трафік у відкритому вигляді між балансувальником і бекендом та не потребують HTTP(S) функціоналу.

3. *TCP Proxy Load Balancing* – дозволяє обробляти TCP-з'єднання і передавати їх найближчому доступному бекенду, не використовуючи HTTP або HTTPS. Застосовується для балансування ігрових серверів або систем з високими вимогами до пропускної здатності.

Регіональні балансувальники наступні:

1. *Internal HTTP(S) Load Balancing* – призначений для внутрішнього трафіку в рамках одного регіону. Підтримує маршрутизацію HTTP/HTTPS на основі контенту. Застосовується для мікросервісних архітектур і внутрішніх API, де балансування відбувається всередині ВПХ мережі (Virtual private cloud, тобто відмежованої частини хмари, див. [1.3.2. Віртуальна приватна хмара](#)).

2. *Internal TCP/UDP Load Balancing* дозволяє розподіляти TCP та UDP трафік між ресурсами в межах одного регіону. Підходить для stateful-додатків та програм, баз даних або систем обробки великих обсягів даних.

3. *UDP Load Balancing* – підтримує балансування UDP трафіку для зовнішніх з'єднань, що робить його зручним для програм з потребою у високопродуктивному розподілі пакетів. Підходить для програм, які використовують UDP трафік, наприклад, ігрові сервери, VoIP або потокове передавання мультимедіа.

4. *Network Load Balancing* – для розподілу TCP/UDP трафіку з високою пропускнуою здатністю та мінімальною затримкою. Працює на 4-му рівні моделі OSI та має високу продуктивність; підходить для програм, що потребують мінімальної затримки, наприклад для фінансових додатків, ігрових платформ або служб обміну повідомленнями.

5. *External TCP/UDP Network Load Balancing* – використовується для балансування зовнішнього TCP/UDP трафіку з можливістю його перенаправлення до найближчих доступних ресурсів. Підходить для публічних додатків, що працюють з TCP та UDP протоколами.

Нижче наведено Таблиці 1.2 та 1.3 з порівнянням балансувальників.

1.3.4. Порівняння балансувальників GCP

Таблиця 1.2 Основні види та характеристики балансувальників в GCP

Тип балансувальника	Рівень OSI	Масштаб	Протоколи	Основні можливості	Приклади використання
HTTP(S) Load Balancer	Рівень 7	Глобальний	HTTP, HTTPS	– Балансування трафіку на основі HTTP(S) запитів – URL-маршрутизація – SSL-термінація	Веб-застосунки з глобальною аудиторією, розподіл трафіку між регіонами.
TCP/UDP Load Balancer	Рівень 4	Регіональний або глобальний	TCP, UDP	– Балансування трафіку на основі протоколів – Регіональна доступність – Висока продуктивність	Високонавантажені обчислення, реальні застосунки (ігри, мультимедіа).
Internal HTTP(S) LB	Рівень 7	Регіональний	HTTP, HTTPS	– Балансування внутрішнього HTTP(S) трафіку – Підтримка приватних IP-адрес	Внутрішні мікросервісні архітектури або корпоративні API.

Internal TCP/UDP LB	Рівень 4	Регіональний	TCP, UDP	– Внутрішнє балансування для приватних мереж – Підтримка тільки внутрішніх IP-адрес	Розподіл навантаження між сервісами в одній ВПХ.
SSL Proxy Load Balancer	Рівень 7	Глобальний	SSL (TCP)	– SSL-термінація – Балансування зашифрованого TCP-трафіку – Інтелектуальне розподілення трафіку	Безпечні веб-застосунки, що потребують глобального доступу.
TCP Proxy Load Balancer	Рівень 4	Глобальний	TCP	– Балансування TCP-трафіку без шифрування – Оптимізація з'єднань	Низьколатентні застосунки, які не використовують шифрування.
Network Load Balancer (NLB)	Рівень 4	Глобальний або регіональний	TCP, UDP, ICMP	– Надзвичайно низька затримка – Балансування необроблених мережових запитів	IoT-сервіси, фінансові застосунки з вимогою мінімальної затримки.

1.3.5. Порівняння балансувальників за типом їх роботи в різних хмарних системах

Таблиця 1.3 Основні види за принципом дії балансувальників в GCP, AWS та Asure

Тип балансувальника	Рівень OSI	Масштаб	Технології	Приклади (GCP, AWS, Azure)	Додаткові характеристики
HTTP(S) Load Balancer	Рівень 7	Глобальний	HTTP/HTTPS протоколи, CDN, Anycast	– Google HTTP(S) Load Balancer – AWS Application Load Balancer (ALB) – Azure Front Door	Для веб-застосунків із розширеними функціями SSL-термінації, маршрутизації URL.

TCP/UDP Load Balancer	Рівень 4	Регіональний або глобальний	TCP/UDP протоколи, IP Hash	<ul style="list-style-type: none"> – Google Cloud TCP/UDP Load Balancer – AWS Network Load Balancer – Azure Load Balancer 	Балансування на основі IP або портів.
Internal HTTP(S) LB	Рівень 7	Регіональний	HTTP/HTTPS, Приватні IP	<ul style="list-style-type: none"> – Google Internal HTTP(S) Load Balancer – AWS Private ALB – Azure Application Gateway (внутрішній) 	Для внутрішніх сервісів та API в межах одного регіону.
Internal TCP/UDP LB	Рівень 4	Регіональний	TCP/UDP, IP Hash	<ul style="list-style-type: none"> – Google Internal TCP/UDP Load Balancer – AWS NLB (внутрішній) – Azure Internal Load Balancer 	Для приватних мереж і мікросервісів.
SSL Proxy Load Balancer	Рівень 7	Глобальний	SSL, Anycast	<ul style="list-style-type: none"> – Google SSL Proxy Load Balancer – AWS Global Accelerator – Azure Front Door 	Для зашифрованого трафіку (SSL/TLS).
TCP Proxy Load Balancer	Рівень 4	Глобальний	TCP, Anycast	<ul style="list-style-type: none"> – Google TCP Proxy Load Balancer – AWS NLB – Azure Load Balancer 	Підтримка балансування TCP-з'єднань із низькою затримкою.
Network Load Balancer (NLB)	Рівень 4	Регіональний або глобальний	TCP/UDP, IP Direct Routing	<ul style="list-style-type: none"> – Google Network Load Balancer – AWS NLB – Azure Load Balancer 	Для необробленого трафіку.

1.4. Планування розподілу задач у дата-центрі та на хмарі

Планувальник або Scheduler — це система або компонент, що відповідає за прийняття рішень щодо того, які ресурси мають бути виділені для виконання завдань у певний момент часу. У контексті хмарних обчислень та оркестрації контейнерів, таких як Kubernetes, це механізм для розподілу робочих навантажень між доступними ресурсами (сервери, вузли, контейнери тощо).

1.4.1. Задачі планувальника, його обмеження та дотримання політик

Планувальник виконує наступні задачі:

1. *Розподіл ресурсів* – визначення, який обчислювальний вузол (ОВ або сервер) підходить для запуску завдання (контейнера, поду, віртуальної машини) враховуючи доступність ресурсів, таких як CPU, пам'ять, диск, мережеві обмеження тощо.

2. *Оптимізація навантаження* – забезпечення рівномірного розподілу навантаження між серверами та зменшення ризику перевантаження окремих вузлів.

3. *Автоскейлінг* чи *автомасштабування* – планування розширення або зменшення кількості вузлів залежно від навантаження.

4. *Моніторинг та повторне планування* – оцінка виконання завдань і, за потреби, повторне планування, наприклад, у разі збою.

Обмеження та дотримання політик планувальником мають включати наступне:

1. Врахування обмежень, заданих в налаштуваннях (taints, tolerations, affinity, anti-affinity).
2. Дотримання правил локальності даних, зон доступності, мережевих або безпекових вимог.
3. Енергозбереження – за будь-яких дій, зокрема при автомасштабуванні слід зважати на економну витрату ресурсів та концентрувати робоче

навантаження на мінімальній кількості вузлів для відключення невикористовуваних серверів.

1.4.2. Будова розподіленого планувальника задач

Архітектура розподіленого планувальника задач, наведена на Рисунку 1.3, ілюструє базову модель розподіленої системи для виконання завдань у хмарній інфраструктурі. Додавання механізмів моніторингу, автоскейлінгу, безпеки та розподілу по зонах доступності покращить продуктивність, стійкість до збоїв та масштабованість системи.

Ключові компоненти планувальника описані в [21]:

1. Клієнти – це завдання від клієнтських програм (веб-додатків чи мобільних пристроїв), які потрібно обробити. Клієнти взаємодіють із Load Balancer – балансувальником навантаження для розподілення виправлених запитів.

2. Балансувальник навантаження рівномірно розподіляє вхідні запити клієнтів між екземплярами сервісів. Він відповідає за зменшення перевантаження окремих серверів та підтримку високої доступності застосунку через масштабування.

3. Сервіс виправлення завдань приймає запити від балансувальника навантаження і зберігає дані про завдання у Job Store. Мимохідь, він також здійснює валідацію запитів, агрегацію та форматування завдань.

4. Сховище завдань – це база даних для зберігання завдань, метаінформації та статусу їх виконання, яка може бути реалізована на основі SQL або NoSQL. Сховище завдань гарантує доступність даних для Scheduling Service. Можливо виконувати регулярні резервні копії для Job Store для запобігання втраті інформації.

5. Сервіс планування своєю чергою планує розподіл завдань між робочими вузлами, беручи перші з розподіленої черги завдань.

6. Розподілена черга завдань зберігає чергу завдань, які очікують обробки. У завдань наявні критерії для їх вибору такі як пріоритет, час виконання та доступність ресурсів.

7. Сервіс виконання виконує завдання, передані з черги. Його керівним модулем є координатор.

8. Координатор відповідає за розподіл завдань між робочими вузлами. В разі невдачі з запуском задачі, вступає механізм повторних запусків, описаний в [4].

9. Робочі вузли є безпосередніми виконавцями завдань, які використовують паралельні потоки для обробки завдань, а також відправляють звіти про результати назад до Job Store. Для організації автоматичного масштабування робочих вузлів залежно від навантаження, яке має бути однією з основних функцій системи, слід використати масштабувальники, зокрема для горизонтального масштабування, наприклад Kubernetes Horizontal Pod Autoscaler. Це сприятиме забезпеченню високої доступності задач шляхом розподілу сервісів між зонами доступності у хмарній інфраструктурі.

10. Додатково, через інтеграцію з інструментами моніторингу, такими як Prometheus чи Elasticsearch, можна слідкувати за продуктивністю системи, статусом завдань та вузлів.

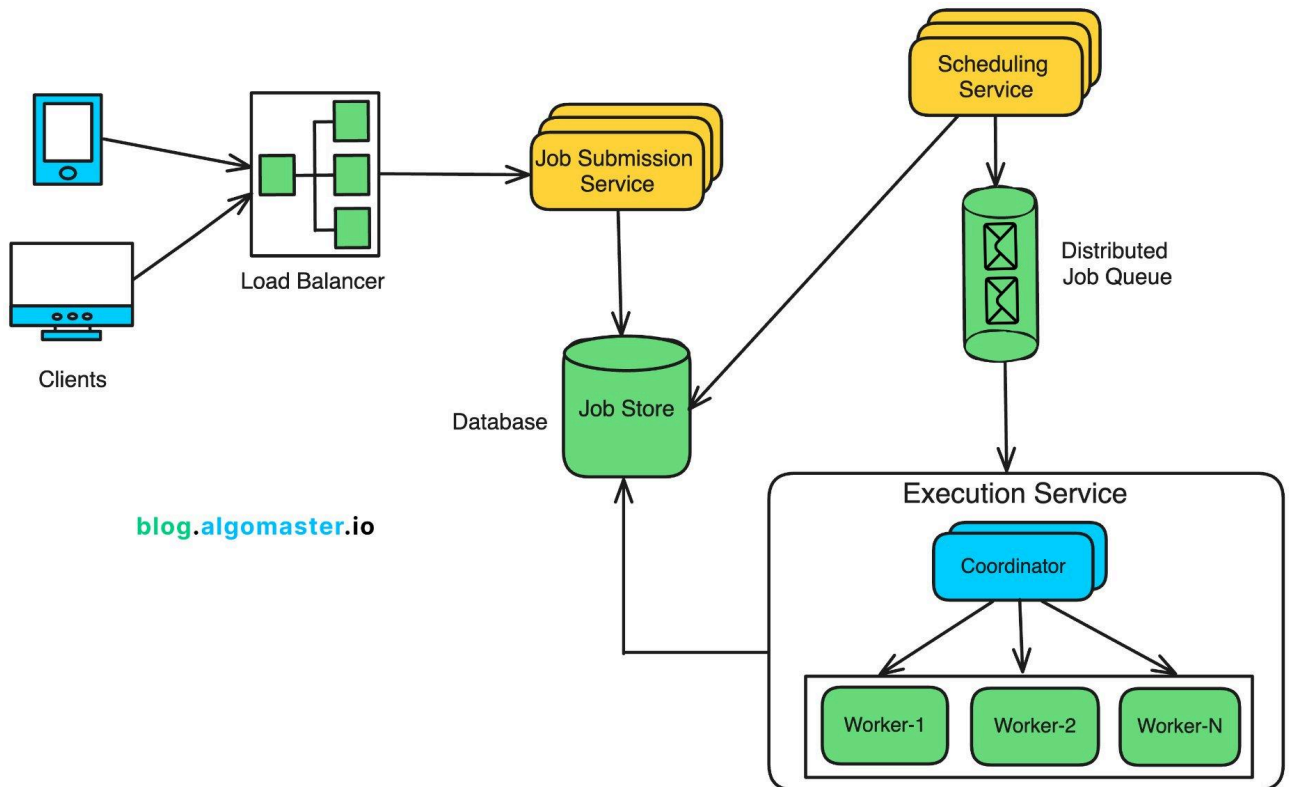


Рисунок 1.3 — Високорівневий дизайн розподіленого планувальника задач (job scheduler)

1.4.3. Типи планувальників

Планувальник поділяють за зоною дії планування на глобальні та локальні. При **глобальному плануванні** Scheduler оцінює всі ресурси в межах кластера або хмарного середовища. Наприклад, Kubernetes Scheduler працює з усіма вузлами кластера, тоді як хмарні провайдери (AWS, GCP) можуть враховувати навіть різні регіони чи зони доступності. **Локальне планування** – використовується для управління ресурсами в межах одного вузла, наприклад, розподіл контейнерів Docker на фізичній машині.

За необхідністю втручання планувальники поділяють на ручні, автоматичні та гібридні. При **ручному плануванні** завдання призначаються адміністраторами вручну, напр., запуск контейнерів на конкретному сервері. Така модель поведінки звична для класичних дата-центрів. **Автоматичні планувальники**, які використовуються у хмарних технологіях та Kubernetes, мають певний алгоритм для автоматичного розподілу задач, про

який написано в нище. **Гібридні планувальники** поєднують автоматичне та ручне управління.

Планувальники в Kubernetes, AWS, GCP, Alibaba, Azure мають спільні основи, але відрізняються залежно від призначення та середовища. Зазвичай, в кожній системі розроблений власний планувальник, але є і планувальник на основі Kubernetes для забезпечення сумісності з ручним тестуванням поза хмарною системою. Kubernetes Scheduler є більш універсальним і модульним, тоді як хмарні провайдери додають специфічні функції для інтеграції з їхньою інфраструктурою.

1.4.4. Планувальник Kubernetes

Kubernetes Scheduler відповідає за розподіл подів (Pods), які ще не призначені до жодного вузла (Node), на найбільш підхожий ОВ у кластері. Його робота базується на аналізі ресурсів вузлів, потреб подів та конфігурації кластера.

Стосовно термінології, под – найменша керована одиниця в Kubernetes, яка може містити один або кілька контейнерів, які працюють разом. Контейнери в поді поділяють одне мережеве середовище (IP-адресу) та можуть обмінюватися файлами через спільні томи. Поди використовуються для забезпечення логічного об'єднання кількох контейнерів, які мають тісну взаємодію. *Контейнер* – це ізольована середа, яка запускає додаток і всі залежності.

Алгоритм розподілу подів на сервери складається з наступних частин [22]:

1. **Вхідними даними** у Scheduler є список незапланованих подів і інформація про всі доступні вузли (Node).
2. **Фільтрація вузлів** (Predicate Filtering), яка відбувається на основі запитів поду, в результаті яких відсіюються вузли, які не відповідають вимогам. До **причини невідповідності** відносяться наступні:
 - недостатньо CPU, RAM або іншого ресурсу;

- вузол позначений як неактивний або має обмеження taints (заборони) [23];
- невідповідна конфігурація вузла, наприклад, відсутність необхідного AZ або тегів – інформації про наявні характеристики.

Знаки відповідності вузла визначаються, якщо под має відповідні nodeSelector та affinity. Якщо под потребує підключення до дисків – PersistentVolume, перевіряється доступність диска на кожній ноді.

3. **Оцінка вузлів (Scoring)** – обирається найбільш відповідний вузол, розраховуючи оцінки на основі критеріїв:

- рівномірний розподіл є ціллю розподілення, щоб уникнути перевантаження окремих вузлів. Може бути обраний вузол, на якому залишиться найбільше вільних ресурсів після запуску поду;
- локальність даних є іншою частиною функції оптимізації планувальника для підвищення продуктивності системи шляхом зменшення латентності доступу до даних, тобто якщо дані, потрібні для пода, вже знаходяться на певній ноді, вона отримує вищий пріоритет;
- антиафініті – налаштування, яке вказує обмеження на розміщення подібних подів на одному вузлі для підвищення надійності системи.
- додаткові користувацькі правила Custom Rules теж будуть враховані, які можуть виражатись власними алгоритмами оцінювання через Scheduler Extenders.

4. **Призначення (Binding)** – створення відповідного Binding об'єкту, який закріплює под за вузлом після вибору найбільш нагідного вузла, тобто того, який набрав найвищу оцінку.

В разі недостаточності нодів у кластері, може залучитись Cluster Autoscaler, який відповідає за автоматичне масштабування кластера. Його функція — додавати нові ноди до групи вузлів (Node Group або кластеру) у клауді (AWS, GCP, Azure). Він вмикається на етапі перевірки наявності нерозміщених под через брак ресурсів. Тоді до групи вузлів додається новий екземпляр (віртуальна машина).

Для цього Kubernetes API повідомляє про проблему, а Cloud Provider API запускає новий вузол. Нова нода автоматично додається до кластера Kubernetes. Також користувач може вручну додати нові ноди, якщо автоскейлінг вимкнено або налаштований неправильно. Коли навантаження зменшується, Cluster Autoscaler може видалити вузли, якщо вони не використовуються.

1.4.5. Ефективність планувальника Kubernetes

Ефективність планувальника забезпечується через:

1. *розумне використання ресурсів*, тобто врахування обсягу ресурсів, які потрібні поду (requests та limits) та забезпечення балансування навантаження між вузлами. У разі, коли ресурсів залишається мало, Kubernetes починає «ущільнювати» поди, щоб вивільнити інші ноди для видалення або зменшення споживання;
2. *підтримку Node Pools* – набору вузлів з різними конфігураціями, що дозволяє краще підбирати підходящий до різних типів подів;
3. *забезпечення антиколізії* – використання політик Pod Affinity/Anti-affinity, Taints і Tolerations [23], що зменшує конкуренцію подів за ресурси.

1.4.6. Енергоощадження на Kubernetes

Енергоощадження – це не головна ціль Kubernetes. В першу чергу система дбає про якісне забезпечення послуг підтримки програмного забезпечення в робочому стані залежно від налаштувань. Тому першочергова задача виконання оптимізаційних процесів – звільнення ресурсів для можливої їх зайнятості іншим ПЗ. Але другорядна ціль – це зменшення розтрат на АЗ та супутніх витрат, таких як енергетика. Оскільки ці дві цілі мають в даному контексті однакові методи реалізації, можна говорити, що енергоефективність забезпечується через:

1. *автоскейлінг вузлів (Cluster Autoscaler)* – автоматичне додавання або видалення вузлів залежно від потреб кластера. Якщо ресурсів достатньо, кластер може відключити невикористовувані вузли.

2. *оптимізацію розподілу вузлів*, яка реалізується через концентрацію подів на меншій кількості вузлів більшої місткості та вимикання «порожніх» вузлів для економії енергії;

3. *використання менш енергомістких вузлів*, тобто спеціальних вузлів з підтримкою енергоефективних процесорів, які доцільно використовувати для менш інтенсивних завдань.

Отже, Kubernetes Scheduler намагається знайти баланс між продуктивністю, економією ресурсів та швидкістю виконання. Він динамічно адаптується до змін у кластері, забезпечуючи рівномірний розподіл навантаження і мінімізацію енергоспоживання.

1.4.7. Визначення кількості потрібних ресурсів на Kubernetes

Планувальник використовує інформацію про ресурси, зазначену в маніфесті (конфігурації) поду або контейнера, щоб визначити кількість потрібних CPU, RAM, та інших ресурсів. Це відбувається завдяки запитам (requests) і лімітам (limits), які задають розробники або адміністратори.

Запити (Requests) або базове навантаження – це мінімальна кількість ресурсів, необхідна для роботи поду або контейнера та використовується для перевірки, чи може ОВ забезпечити достатньо ресурсів для виконання завдання. *Ліміти (Limits) або максимальне навантаження* – це максимальна кількість ресурсів, яку може використовувати контейнер. Ліміти не впливають на планування, але визначають межі, які контейнер не може перевищити під час роботи. Чітке визначення requests дозволяє Scheduler ефективно використовувати ресурси вузлів, адже відсутність requests або limits може призвести до перевантаження системи.

Kubernetes також дає можливість налаштувати дані стосовно графічної кари (GPU), об'єму та швидкості жорсткого диска, але поки не має прямої підтримки для requests та limits на мережеву пропускну здатність. Однак її можна забезпечити через використання Network Policies для обмеження мережевого трафіку або спеціальних плагінів контейнеризованого мережевого інтерфейсу (Container Network Interface, CNI), які підтримують QoS (Quality of Service).

Kubernetes дозволяє «перевищення» (Overcommitment) ресурсів коштом оптимізації використання, яку бере на себе. В параметрі requests заведено резервувати мінімум необхідних ресурсів, тоді як limits задають максимум. Scheduler планує роботу на основі requests, що дозволяє зменшити незалучені ресурси. В разі необхідності за даними моніторингу, Kubernetes використовує Horizontal Pod Autoscaler (HPA) або Vertical Pod Autoscaler (VPA) для динамічного коригування ресурсів.

Kubernetes надає рекомендації не встановлювати ліміти, значно вищі за запити, без необхідності, щоб уникнути перевантаження вузлів, адже при плануванні місця на ноді береться тільки одна характеристика – запити. Краще виконати горизонтальне масштабування, ніж щоб контейнер постійно збільшував свою місткість.

Прикладом конфігураційного файлу yaml, в якому зазначені параметри ресурсів та лімітів може служити наступний лістинг, в якому зарезервовано 0.25 CPU і 64 MiB RAM на вузлі, але може використовуватись до 0.5 CPU і 128 MiB RAM під час роботи:

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
spec:
  containers:
    - name: example-container
```

```

image: nginx
resources:
  requests:
    cpu: "250m" # 0.25 CPU
    memory: "64Mi"
  limits:
    cpu: "500m" # 0.5 CPU
    memory: "128Mi"

```

Говорячи про заповнення вузла подами, система залишає нерозмічений ресурс, придатний для масштабування подів. Загальна кількість ресурсів вузла, які доступні для розподілу між подами називається *Allocatable Resources* та складається з наступних частин:

- *Capacity* – це загальні ресурси вузла (CPU, RAM, диск).
- *KubeReserved* – ресурси для компонентів Kubernetes.
- *SystemReserved* – ресурси для операційної системи.
- *EvictionThreshold* – ресурси, які зберігаються для уникнення критичного перевантаження вузла.

Ці логічні ресурси можна визначити скориставшись наступною формулою:

$$Allocatable = Capacity - (KubeReserved + SystemReserved + EvictionThreshold)$$

Зазвичай близько 10–20% загальної місткості вузла зарезервована під системні потреби, але точне значення залежить від розміру вузла, робочого навантаження, і конфігурації kubelet.

Типові значення зарезервованих ресурсів наступні:

CPU: для системних служб зазвичай резервується 0.1–0.5 CPU. Конкретна кількість залежить від розміру вузла та навантаження.

RAM: для системних служб може резервуватися 200–500 МБ пам'яті. У випадку великих вузлів це значення може бути більшим.

Диск: Kubernetes зазвичай резервує частину дискового простору для логів та тимчасових файлів.

1.4.8. Інструменти створення автоматичної конфігурації поду

Оскільки функція створення такої конфігурації не є складною, та для неї достатньо аналітики використання контейнеру, існують інструменти, які дозволяються її згенерувати на основі історичних даних. Це можуть бути Helm – інструмент, головна функція якого відображення статистики використання ресурсів та Kubernetes Dashboard з командним рядком `kubectl`. Kustomize – інструмент для генерації та зміни YAML-файлів без створення нових файлів із нуля, який підтримується `kubectl`. Системи GitOps ArgoCD та FluxCD можуть автоматично синхронізувати маніфести з репозиторіїв і генерувати їх на основі шаблонів.

VPA здатен радити оновлення ресурсів та лімітів на основі історичних даних або їх оновлювати автоматично. KubeCost – інструмент для аналізу вартості та ефективності роботи кластерів. Він може допомогти оцінити перевитрати або недовикористання ресурсів і запропонувати зміни. Деякі комерційні платформи, як-от StormForge або Cast AI, використовують історичні дані та машинне навчання для створення оптимальних маніфестів користуючись історичними та кастомними даними. Інструменти є платним, але є можливість безплатної пробної версії. Приклад роботи такого застосунку наведений на Рисунку 1.4

Terraform та інші IaC (Infrastructure as a Code) інструменти дозволяють створювати конфігурації кластерів, включно з маніфестами, через код.

У більшості випадків, історичні дані не використовуються для первинного створення маніфестів, але у випадках автоматичного скейлінгу (HPA, VPA) або за допомогою спеціалізованих інструментів, як-от Prometheus чи StormForge, історичні дані можуть бути ключовими для оптимізації.

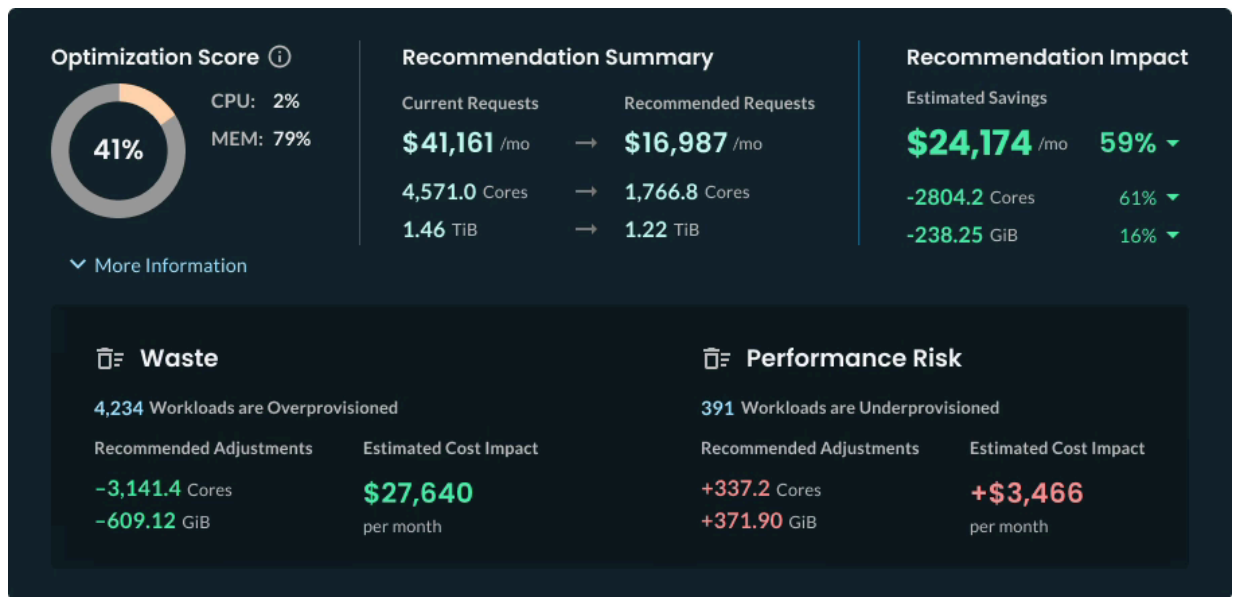


Рисунок 1.4 — Аналіз метрик за допомогою StormForge [24]

1.4.9. Врахування історичних даних та пікових навантажень

Історичні дані відіграють важливу роль у точному налаштуванні автоскейлінгу, особливо в складних сценаріях, коли активність подів змінюється протягом дня, тижня чи залежно від сезонних факторів. Можна використовувати горизонтальне масштабування та калібрування порогових значень на основі метрик знятих через Prometheus, Metrics Server тощо про CPU, RAM, кількість запитів або інших налаштованих історичних даних. Додатково можна залучити інструменти прогнозування: Grafana, або спеціалізовані сервіси StormForge чи AWS Forecast. Доступна інтеграція з календарями, адже через спеціалізовані API можна врахувати святкові дні та адаптувати стратегію скейлінгу.

Для прикладу, ці інструменти допоможуть визначити, що активність CPU часто перевищує 80% о 10:00 ранку, і завчасно збільшити кількість подів, або ж передбачати пік навантаження та збільшити кількість реплік перед початком акцій чи в святкові дні, коли навантаження зростає.

VPA теж використовує історичні дані про використання ресурсів для динамічного налаштування requests і limits подів. Нехай якщо протягом останніх тижнів середнє використання RAM становило 256 MiB, але на піках доходило до 512 MiB, VPA може автоматично збільшити ліміт до 512

MiB, щоб уникнути виснаження оперативної пам'яті (Out Of Memory Exception). В той самий час, на практиці не нехтують і ручним коригуванням: адміністратори можуть вручну масштабувати кластер перед святковими днями, коли навантаження прогнозовано збільшується.

На основі історичних даних Cluster Autoscaler вивчає, чи достатньо вузлів у кластері для розміщення всіх подів, а також, чи певні вузли не залучені в роботу, і вимкнути їх для енергоощадження.

Kubernetes не підтримує автоматизацію скейлінгу за розкладом, але можна використовувати

- KEDA (Kubernetes Event-driven Autoscaler), який дозволяє масштабувати поди на основі подій, наприклад, часу доби чи зовнішніх сигналів.
- CronJobs, які використовуються для запуску завдань та змінюють конфігурацію ресурсу за розкладом.

1.4.10. Зменшення кількості подів

Kubernetes не відстежує «низьку задіяність» подів автоматично, але може використовувати **моніторинг**. За допомогою Custom Metrics, Prometheus або інших систем для аналізу використання ресурсів поду (CPU, RAM, мережа) можна налаштувати вимикання мало задіяних вузлів. При використанні HPA якщо навантаження на под падає нижче визначеного порогу (наприклад, CPU < 20%), кількість реплік може бути зменшена. Под, який більше не потрібен, закривається через Graceful Shutdown з попередженням і завершенням поточних процесів. В разі відсутності подів на вузлах, Cluster Autoscaler може їх вимикати або «скейлити до 0».

1.4.11. Проблеми планувальників на базі Kubernetes

Планувальник Kubernetes продуманий та добре налаштований, але має декілька проблем, зазначених нижче.

1. **Недостатня кількість ресурсів вузла апіорі та висхідна необхідність масштабування.** Оскільки запити (requests) визначають

мінімум ресурсу, який необхідний поду для запуску, а планувальник Kubernetes враховує тільки ці значення при розміщенні подів на вузлах, та не перевіряє, чи вистачить ресурсу для покриття максимуму (лімітів), може скластись ситуація, коли обчислювальний ресурс (ОР або вузол, сервер) буде перевантажений, якщо багато подів збільшують використання ресурсів до лімітних значень. Запас міцності вузла може бути не дуже великим, якщо він розміщує багато подів. В крайньому разі він рівний `EvictionThreshold` (див. [Визначення кількості потрібних ресурсів на Kubernetes](#)).

2. Більший запит ніж потрібний для уникнення частого масштабування. Зчитуючи історичні дані в автоматичному режимі, VPA змінює запити та ліміти для поду залежно від попиту використання ресурсів та може рекомендувати більші запити, якщо поточні значення часто перевищуються. Негативом ситуації є неможливість автоматичного зменшення використання ресурсу, коли він непотрібний повною мірою. Таким чином, кількість ресурсу може бути невиправдано високою в деяких ситуаціях.

3. Обмеження ресурсів та зниження продуктивності при перевищенні лімітів. Якщо контейнер намагається перевищити CPU limits, ядро Linux обмежує доступ до CPU, використовуючи cgroups (control groups) або CFS Quota (Completely Fair Scheduler) ядра Linux, який обмежує кількість доступних обчислювальних циклів.

Це може проявлятися як зниження продуктивності. Якщо контейнер намагається перевищити limits пам'яті, Kubernetes примусово завершить контейнер користуючись OOMKill – out of memory kill. Це захищає ОР від перевантаження, але не грає на користь запущеному застосунку, адже він буде перезапущений на новій ноді тільки в разі відповідних налаштувань та наявності вільного вузла. Якщо нема вузла з достатньою кількістю необхідних рекомендованих ресурсів, то под буде у стані очікування до появи такого.

4. Зарезервована оперативна пам'ять під под, який може нею не користуватись, а також не відпускати. Якщо контейнер запросив більше оперативної пам'яті, ніж значення requests, але менше, ніж limits, ця пам'ять залишиться зарезервованою для поду, навіть якщо вона більше не використовується. Операційні системи та середовища виконання зазвичай не повертають зарезервовану пам'ять автоматично. Це означає, що фізична пам'ять вузла залишатиметься зайнятою, доки контейнер не буде перезапущений або програма не звільнить пам'ять вручну. З однієї сторони, це гарантує, що програма зможе швидко отримати доступ до пам'яті, яку вона вже запитала, але при цьому, на вузлі може бути фактично не зайнята оперативна пам'ять, але нею не можна скористатись іншим подом в разі потреби.

Такої проблеми з CPU та каналним ресурсом нема, адже фактично використовувана потужність і буде актуальною, додаткова не буде резервуватись або не відпускатись подом. CPU є спільним і перерозподіляється між контейнерами на вузлі в режимі реального часу.

1.4.12. Планувальники на AWS

Алгоритми планувальників у AWS, GCP, Azure та Kubernetes мають як спільні риси, так і відмінності, які зумовлені їх призначенням, архітектурою та рівнем інтеграції з іншими сервісами. Розглянемо деталі.

Планувальник в AWS, як і в інших хмарних платформах, забезпечує автоматичний розподіл подів або контейнерів по вузлах у кластері, зокрема на EC2 (Elastic Compute Cloud) інстанси. Він враховує конфігурації Kubernetes (EKS – Elastic Kubernetes Service) або власні підходи для специфічних служб AWS, як ECS (Elastic Container Service) чи Fargate.

На Рисунку 1.5 зображено користувача, який подає завдання на основі шаблону визначення завдання до черги завдань, яка потім повідомляє обчислювальному середовищу про необхідність ресурсів. Ресурс обчислювального середовища масштабує обчислювальні ресурси на Amazon EC2 або AWS Fargate та реєструє їх у сервісі оркестрації

контейнерів — Amazon ECS або Amazon Elastic Kubernetes Service (Amazon EKS) [25].

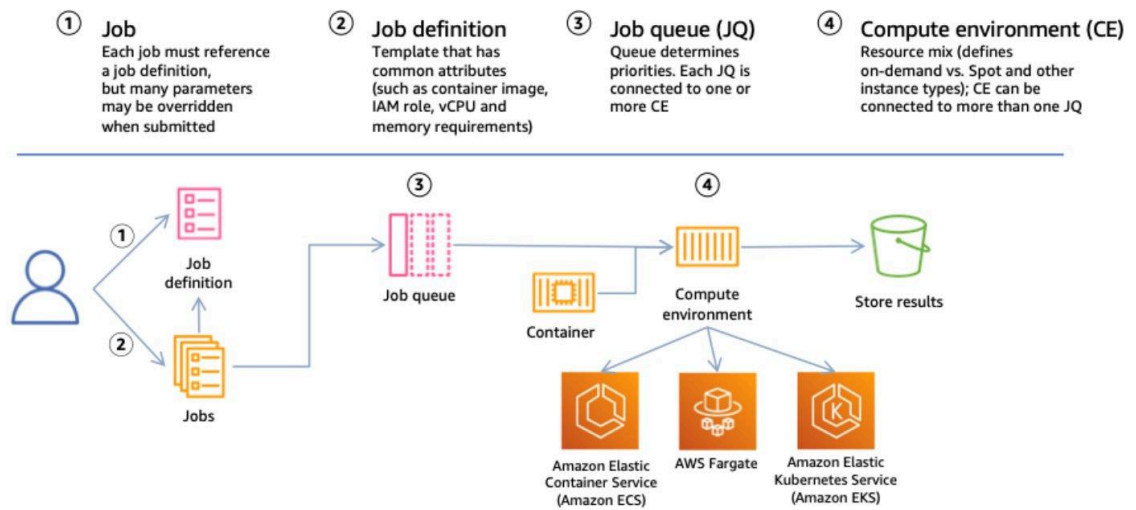


Рисунок 1.5 — Високорівнева структура ресурсів AWS Batch та їх взаємодій [25].

Алгоритми роботи EKS та ECS наступний:

Планувальник Kubernetes в AWS EKS працює за стандартними алгоритмами Kubernetes. При розподілі подів на вузли виконуються наступні кроки:

- 1) перевірка, чи вузол відповідає вимогам до ресурсів CPU та RAM та додатково налаштованих на основі requests;
- 2) врахування політики афінності/антиафінності;
- 3) оцінка доступності вузла з урахуванням taints і tolerations;
- 4) використання стратегії «найкраще підходить» (best fit) для оптимізації розміщення.

Додатково AWS пропонує покращення у розподілі Node Groups. В EKS вузли організовані в Node Groups, які можуть налаштовуватись для обробки різного типу подів (наприклад, поди з GPU, великі поди тощо). Залежно від запитів поди, їй можуть пропонуватись вузли з різних груп.

AWS звісно надає можливості Cluster Autoscaler для додавання вузлів, коли поточна інфраструктура перевантажена. Горизонтальне та вертикальне масштабування можуть активувати розподіл нових подів за потреби.

У випадку перевантаження ноди Kubernetes перезапускає поди, намагаючись перемістити їх на інші ноди. AWS Elastic Load Balancer (ELB) або Application Load Balancer (ALB) часто використовується для рівномірного розподілу трафіку між подами, про які більше написано в розділі [Балансування навантаження в Amazon Web Service](#).

Еластичний планувальник контейнерів ECS має свої особливості. Нижче наведено його стратегія.

1. При **розміщенні завдань** враховуються типи розгортання:

- *REPLICA*: рівномірно розподіляє завдання по всіх вузлах.
- *DAEMON*: створює одне завдання на кожному доступному вузлі.

Аналогічно до Kubernetes планувальника, враховуються вимоги користувача до ресурсів для кожного Task (аналог Pod): CPU, RAM, мережева конфігурація, сховища. Політики розміщення має наступні аспекти:

- *placement constraints*: наприклад, запуск завдання тільки на певних вузлах.
- *placement strategies*: наприклад, розподіл за принципом мінімального (spread), рівномірного або максимального використання вузлів. На відміну від Kubernetes Scheduler, ECS може дотримуватись принципу «один под на ноду» в разі налаштування **spread** з метою оптимізувати розподіл завдань для рівномірного покриття вузлів. Інша опція – **bin packing** – ECS може обрати розміщення на найбільш завантаженому вузлі для уникнення запуску зайвих EC2 та мінімізації простою. Також ECS може виконувати оптимізацію під специфічні ресурси такі як GPU або вузли з низькою затримкою.

2. **Фільтрування** – вузли, які не відповідають вимогам (ресурсів, мережевих конфігурацій), виключаються з розгляду.

3. **Використання Fargate**, планувальника, що працює всередині Amazon ВПХ, та який забезпечує автоматичний розподіл навантаження на інфраструктурі AWS. Завдання ECS запускаються без прив'язки до конкретного вузла та автоматично масштабуються. Користувач задає список підмереж (subnets) у конфігурації. AWS автоматично розподіляє Task/Pod по підмережах з урахуванням доступності та балансу між AZ. Для забезпечення безпеки Fargate використовує ізольовані підмережі (ВПХ) для кожного Task/Pod, а також виконує інтеграцію з IAM (Identity and Access Management), що дозволяє забезпечити доступ тільки до необхідних ресурсів [25].

Стосовно мережевих вимоги, обидва планувальники враховують конфігурацію ВПХ/Subnet і використовують конфігурацію підмереж для розподілу Task/Pod по різних Availability Zones (AZ). Якщо кластер мультизональний, планувальник намагається зберегти баланс між зонами. Пріоритет надається підмережам із кращою доступністю ресурсів. Якщо Task/Pod пов'язаний із Load Balancer, розподіл забезпечується за допомогою балансувальника, який автоматично обирає найоптимальнішу AZ.

Кластери в AWS можуть бути ізольовані за

- географією (регіони),
- призначенням (тестові/продакшн), або
- типами завдань (наприклад, кластери з GPU).

При досягненні обмежень ресурсів кластер розширюється через додавання нових вузлів за допомогою Cluster Autoscaler, збільшенням кількості підписаних під кластер EC2-інстансів або використанням Fargate. Для міжкластерної взаємодії використовуються AWS PrivateLink, Transit Gateway або peering між ВПХ для зв'язку між кластерами [26] або не треба.

Енергозаощадження в AWS: AWS активно впроваджує механізми для зменшення енергоспоживання та оптимізації використання інфраструктури. Для цього в EKS вузли додаються або видаляються автоматично, якщо поди не можуть бути розміщені на наявних вузлах. Також

використовуються Spot Instances, через які AWS дозволяє використовувати дешеві, незалучені ресурси для тимчасових завдань [27].

Додатково, AWS впроваджує власні чіпи Graviton на базі ARM, які споживають менше енергії та забезпечують високу продуктивність. Відбувається оптимізація дата-центрів шляхом залучення сучасних систем охолодження, екологічних джерел енергії, наприклад, сонячної та вітрової.

Використання певних алгоритмів теж сприяє енергозаощадженню. Fargate завжди обирає AZ з оптимальною доступністю, щоб мінімізувати затримки та уникнути надлишкового використання ресурсів у конкретній зоні. AWS Compute Optimizer аналізує завантаження і пропонує оптимізації, наприклад, зменшення розміру EC2 або перехід на Spot Instances. CloudWatch надає історичні дані про навантаження, що допомагає оптимізувати масштабування.

AWS, використовуючи інструменти як Fargate, Auto Scaling і Load Balancing, досягає оптимального балансу між ефективністю, економією енергії та високою доступністю, дозволяючи користувачам масштабуватися без зайвих витрат.

1.4.13. Планувальники GCP

На Google Cloud Platform (GCP) **Managed Instance Groups (MIG)** є основним рішенням для автоматичної оркестрації груп віртуальних машин (VM). Однак, є й альтернативні способи.

Основним конкурентом є **Kubernetes Engine (GKE)**, який застосовується для контейнеризованої інфраструктури. Він автоматично управляє контейнерами на основі робочого навантаження, забезпечуючи більшу автоматизацію, ніж MIG для VM, а також підтримує **Cluster Autoscaler** для динамічного масштабування вузлів. Алгоритм його роботи аналогічний алгоритму роботи з Kubernetes, який розглянутий у підрозділі [Планувальник Kubernetes](#).

Додатково Гугл хмара пропонує альтернативні рішення залежно від задач. App Engine – кероване середовище для розробки та розгортання додатків з функціями автомасштабування та управління інфраструктурою. Підходить для безсерверних додатків, але не для завдань, що потребують контроль над VM.

Альтернативною безсервеною платформою для розгортання контейнерів, яка автоматично масштабується залежно від навантаження, але призначена для користувачів, які хочуть уникнути оркестрації інстансами або кластерами є Cloud Run.

Додатково існує ще Cloud Functions – структура для виконання коротких задач за викликом чи подій без потреби в управлінні VM. Хмарні функції можуть автоматично масштабуючись, але не забезпечують постійного виконання задач, як MIG чи GKE.

Якщо не потрібна група керованих інстансів, можна вручну налаштувати Auto Scaling для окремих інстансів за допомогою Compute Engine Autoscaler. Ця платформа менш зручна, ніж MIG, але дає більше контролю для нестандартних сценаріїв.

Також є опція вручну керувати розгортанням, оновленням і масштабуванням інстансів для особливих випадків за допомогою Unmanaged Instance Groups, що дає більше контролю.

Managed Instance Groups (MIGs) — це сервіс у хмарних платформах, зокрема Google Cloud, але також присутній у AWS (у вигляді Auto Scaling Groups), Azure та інших, який дозволяє створювати й управляти групами віртуальних машин (VM). MIG забезпечує автоматичне масштабування, самовідновлення та балансування навантаження для інстансів, які працюють за однаковими налаштуваннями.

MIG базується на шаблоні інстансу, який містить конфігурацію віртуальної машини, наприклад, тип машини (vCPU, RAM), операційну систему, мережеві налаштування, стартові скрипти та метадані. Усі інстанси в групі мають однакові характеристики, визначені цим шаблоном.

Для роботи цього планувальника необхідно визначити кількість інстансів, які потрібно створити, або діапазон кількості машин (мінімальне та максимальне значення для автоскейлінгу). Створені на основі шаблону інстанси будуть додані до групи.

Інстанси в MIG інтегруються з балансувальниками навантаження (наприклад, Google Cloud Load Balancer або Azure Load Balancer). Трафік рівномірно розподіляється між інстансами на основі політик балансування.

MIG може автоматично масштабуватися залежно від метрик. Наприклад, якщо завантаження CPU всіх інстансів перевищує 70%, група додає нові інстанси. Метрики наступні:

- Завантаженість CPU.
- Споживання пам'яті.
- Запити до балансувальника навантаження.
- Користувацькі метрики, наприклад, метрики з Google Cloud Monitoring.

MIG виконує автоматичний моніторинг стану кожного інстансу за допомогою health checks. Якщо інстанс виявлено «нездоровим» (наприклад, він не відповідає або перевантажений), MIG автоматично перезавантажує або замінює його новим.

MIG дозволяє розгортати нові версії програмного забезпечення або змінювати конфігурацію інстансів за допомогою оновлення групи. Можна виконувати оновлення поступово, щоб зменшити ризики простоїв.

Робота відбувається як в межах однієї зони, так і між зонами. У зонального MIG усі інстанси розташовані в одній зоні. Таке розташування простіше у налаштуванні, але не захищає від відмови зони. Напротивагу першому, регіональний MIG забезпечує високу доступність у випадку недоступності окремої зони шляхом розташування нодів в кількох зонах одного регіону.

Алгоритм роботи при масштабуванні:

1. Моніторинг метрик – MIG використовує Cloud Monitoring для збору даних про завантаження CPU, пам'яті, трафіку та інших параметрів.

2. Прийняття рішення – якщо поточне завантаження виходить за межі визначеного діапазону, алгоритм додає або видаляє інстанси.

3. Додавання інстансів – нові інстанси створюються на основі шаблону та автоматично додаються до балансувальника навантаження.

4. Видалення інстансів – інстанси з найменшим навантаженням або найстарішими ідентифікаторами видаляються першими.

У Google Cloud Platform є підтримка регіональних та зональних MIG, виконується інтеграція з Google Cloud Load Balancer та масштабування за спеціальними метриками.

У AWS Auto Scaling Groups інтеграція аналогу MIG може вимагати додаткової конфігурації для роботи балансувальників, а також пропонуються розширені політики масштабування (Predictive Scaling).

У Azure Virtual Machine Scale Sets є можливість гнучкого налаштування, включаючи змішані типи інстансів. Можна створювати гібридні групи з Windows та Linux.

1.4.14. Порівняння планувальників

Планувальники AWS, GCP та Azure працюють на глобальному рівні, забезпечуючи як оркестрацію кластерами, так і автоматизацію ресурсів. Відмінності полягають у способі інтеграції з іншими сервісами провайдера, пріоритетах оптимізації (витрати, продуктивність) та рівнях конфігурації. Водночас важливо, що Kubernetes Scheduler присутній на всіх широковідомих провайдерах, та оптимізований для роботи з кожною хмарою (див. Таблицю 1.4).

Таблиця 1.4. Ключові відмінності між планувальниками

Критерій	Kubernetes	AWS	GCP	Azure
Рівень роботи	Локальний (кластер)	Глобальний і локальний	Глобальний і локальний	Глобальний і локальний

Фільтрація вузлів	Ручна конфігурація або політики	Автоматичні фільтри (зони, інстанси)	Автоматичні фільтри (зони, Managed Groups)	Автоматичні фільтри (VM Scale Sets)
Оптимізація витрат	Потрібне зовнішнє налаштування (наприклад, Spot)	Вбудована підтримка Spot-інстансів	Використання Preemptible VM	Використання Azure Low Priority VM
Автоскейлінг	Horizontal Pod Autoscaler, Cluster Autoscaler	Auto Scaling Groups	Managed Instance Groups	VM Scale Sets
Пріоритети	Афініті, антиафініті, топологія	Placement Groups	Пріоритет інстансів за конфігурацією	Пріоритет інстансів за конфігурацією
Інтеграція	Широкі можливості налаштування	Тісна інтеграція з AWS сервісами	Тісна інтеграція з Google сервісами	Тісна інтеграція з Azure сервісами

1.4.15. Розміщення подів одного програмного застосунку на різних кластерах

Кластер Kubernetes — це набір вузлів (нод), що працюють разом і управляються центральним контролером. Він забезпечує ізоляцію середовища так, що ресурси та робочі навантаження кластера є незалежними від інших кластерів, а також виокремлює мережеве оточення, політики доступу, конфігурації та дані кластеру від інших. Це дає можливість керувати навантаженням кластера, автоматично масштабувати ресурси та підтримувати високу доступність за допомогою планувальника, який відповідає за розподіл подів. Кожен кластер має свій API-сервер для управління роботою додатків.

Програмні застосунки або мікросервіси з однієї програми можуть бути запущеними в різних кластерах. Для взаємодії між такими

мікросервісами потрібна добре налаштована мережа, напр., VPN або глобальний балансувальник, та підключення через API.

Це може бути корисним у наступних випадках:

1. **географічний розподіл** відбувається, коли частина сервісів потребує бути ближчою до користувачів у певному регіоні. Наприклад, фронтенд може бути запуснений у кластері в США та Європі, а бекенд — у кластері в Європі;
2. **ізоляція робочих навантажень** в першу чергу з причин безпеки, тоді сервіси з підвищеними вимогами до безпеки можуть розташовуватись в окремому кластері. Також мікросервіси різних команд чи клієнтів можуть працювати в окремих кластерах для зручності управління.
3. **відокремлення середовищ**, котрі працюють або з різною інтенсивністю, або з різною потребою в ресурсах. Наприклад, тестові мікросервіси можуть запускатися в окремому кластері, щоб не заважати продуктивному середовищу.

Для масштабування на інший кластер можуть знадобитись додаткові інструменти. Для Kubernetes це *Kubernetes Federation*, який дозволяє керувати кількома кластерами як єдиною системою, *Istio* [28], [29] або *Linkerd* для маршрутизації трафіку між кластерами, а також *Crossplane* для управління ресурсами в різних кластерах. В разі багатокластерних застосунків на клауді, слід користуватись глобальними балансувальниками для автоматичного розподілення навантаження.

В разі класичного хостінгу, можна вручну додати нові кластери та перенести частину робочих навантажень, налаштувавши API-з'єднання.

1.4.16. Розміщення контейнерів різних програмних застосунків на одному поді

Зазвичай прийнято розміщувати в одному поді один контейнер, що містить усі необхідні елементи, такі як сам мікросервіс та йому потрібні бібліотеки, БД тощо. В той самий час, є випадки, коли це доречно, а саме

при архітектурі «Sidecar», коли один контейнер забезпечує допоміжну функцію для іншого, наприклад, логування, кешування чи проксі. Інший випадок – це коли компоненти мають потребу в спільному ресурсі: дисковому сховищі або спільному мережевому інтерфейсі, працюють з однією базою даних, конфігурацією або кешем, та потенційно мають низький рівень взаємодії із зовнішнім середовищем.

Технічними причинами розміщення декількох контейнерів на поді можуть бути затримки комунікації, яких слід уникнути, або необхідність зменшити залученість інструменти моніторингу або безпеки, котрі теж можуть давати затримку.

Особливості розміщення декількох контейнерів на одному поді:

1. спільний життєвий цикл – усі контейнери в одному поді розгортаються, рестартуються та видаляються разом, що може бути зручним, але тоді якщо один контейнер виходить з ладу, весь под перезапускається, що може порушити роботу інших контейнерів;
2. менші затримки та ефективніше використання ресурсів шляхом спільного доступу до сховищ даних та мережевих портів, але при цьому зменшується ізоляція, що може призвести до конфліктів за спільні ресурси;
3. зручніша конфігурація, адже для організації взаємодії контейнерів менше складності у зв'язуванні та налаштуванні;
4. у разі необхідності масштабування потрібно масштабувати всі контейнери разом, навіть якщо навантаження зросло лише на одному з них [30];
5. ускладнення моніторингу та налагодження, бо утруднюється визначення першопричини проблем через спільний моніторинг.

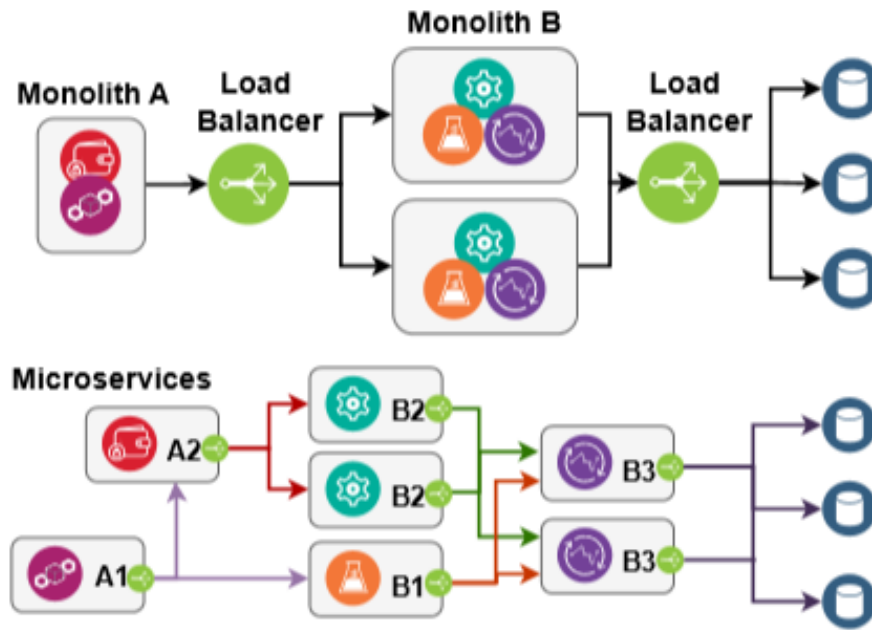


Рисунок 1.6 — Приклад мікросервісної архітектури з двома шинами та балансувальником навантаження перед групою однакових мікросервісів (зверху), та в якості *sidecar* поруч з кожним контейнером (знизу) [31].

На Рисунку 1.6 зображено багаторівневий додаток, побудований на основі монолітних сервісів (зверху), може бути розкладений на компоненти мікросервісів (знизу), що потенційно поліпшує практики розробки, але ускладнює топологію додатку. Балансувальники навантаження «*sidecar*» (зелені круги) розгортані поруч із кожним компонентом мікросервісу для маршрутизації запитів до вузлів обслуговування (нижні вузли) [11]. Таким чином, один з варіантів використання *sidecar* – це в якості балансувальника навантаження. В даній роботі більше цікавить використання «*sidecar*» в якості сусіднього повноцінного контейнера, щоб на одному поді було запущено їх два або більше.

Зважаючи на всі за та проти, можна зробити висновок, що це хороша практика коли використовується архітектура за шаблонами *sidecar*, *adapter*, *ambassador* або *init-containers*, коли додаткові контейнери допомагають основному контейнеру виконувати логування, проксі, оновлення кешу, функції доступу до ресурсів або налаштування, але так не слід робити,

якщо контейнери мають незалежні життєві цикли, різні вимоги до масштабування або не тісно пов'язані один з одним [32].

1.5. Додаткові можливості для підвищення надійності роботи системи та її швидкого відгуку

1.5.1. Класифікація несправностей

Щоб зрозуміти причину помилки, важливо зрозуміти походження проблеми. Якщо проблема помічена на сервері, вона могла виникнути не обов'язково там, але, можливо, через будь-яку залежність сервера. Дивлячись на Рисунок 1.7, можна сказати, що якщо системний елемент не виконує свою роботу належним чином, причиною цього можуть бути помилкові дані в хмарі, які можуть бути викликані проблемами зі шлюзом або проблемами з будь-якими пристроями edge або IoT. Так що в принципі, помилка може викликати все, що завгодно. Нижче описана модель цибулевого розлому. У ньому містяться основні класи несправностей.

Модель цибулевих розломів представляє поділ основних несправностей на п'ять основних класів (Рисунок 1.7). Структура описує, що будь-які причини верхніх несправностей включають будь-яку причину з нижнього шару. Наприклад, причиною довільної несправності може бути збій сервера, який належати до найглибшого рівня зламу в процесі роботи [33].

- *Довільні несправності* — це несправності, які трапляються час від часу, але крім цього, сервер працює правильно.
- *Час* — сервер не може відповісти за заданий проміжок часу, але пізня відповідь буде правильною.
- *Помилка упуцнення* – це коли сервер не відповідає або не приймає запити.
- *Тиха зламання (fail-silent)* — модель несправності — це коли сервер раптово перестає працювати, причому це зрозуміло не всім

робочим компонентам. У синхронних системах такі несправності можуть виявлятися за допомогою тривалих тайм-аутів.

- *Злам в процесі виконання роботи (fail-stop)* — виникають, коли сервер зупиняється в процесі виконання завдання, і про це дізнаються відносні робочі служби.

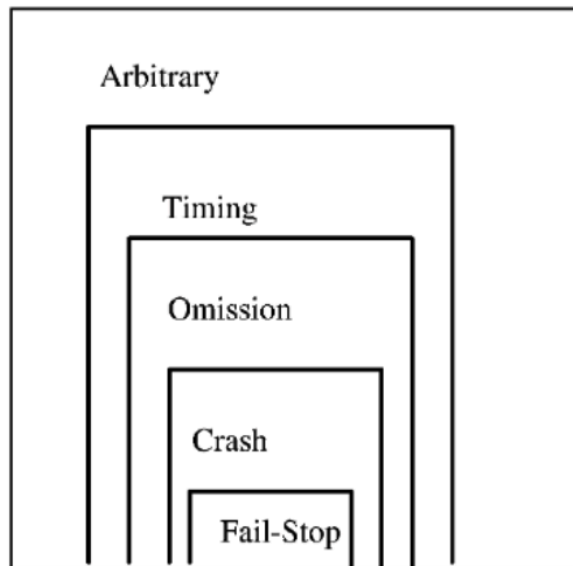


Рисунок 1.7 — Модель несправності Onion [33]

Підводячи підсумок, важливо зазначити, що на кожному етапі має бути певна гарантія того, що збої не відбудуться через «дурні» причини. Якщо є можливість довести до досконалості важливу систему з додатковими гарантіями, що простий випадок не знищить всі алгоритми і аналіз, це потрібно робити. Нижче розглянуті методи, які можуть бути використані для запобігання базового збою, і застосовую їх до різних пристроїв IoT, наводячи приклади.

1.5.2. Додавання резервування в систему

Збій може статися в будь-якій системі в будь-який момент. Йдеться про те, щоб бути готовим до цього. Впровадження системи резервного режиму може бути дуже корисним. Це означає, що додаткові елементи, такі як база даних, сервери, живлення або будь-який інший компонент, будуть у

стані готовності. На випадок, якщо щось не вийде, є інші деталі, готові підхопити роботу.

При виборі місця для резервних елементів слід враховувати багато факторів. Якщо це сервер або база даних, може розглядатися ймовірність землетрусу, війни або їжі. Це означає, що, з одного боку, політично безпечніше не розголошувати дані користувачів через базування баз даних в інших країнах, але з іншого боку, ймовірність фізичного пошкодження має бути достатньо малою, щоб розмістити сервер у сусідньому місці.

1.5.3. Холодний режим очікування

Холодний режим очікування або cold standby, це метод підготовки до запуску мікропослуг або резервних копій, коли додаткова мікропослуга, яка дублює запуснену, встановлюється і запускається один раз для того, щоб бути активованою. Після того, як всі дії по налаштуванню були виконані, система відключається і запускається тільки в разі збою поточної робочої системи [34].

Ця техніка може бути використана в системах, які або не призначені для надання миттєвих відповідей користувачам, або в системах, які не включають збережені дані. Для таких систем краще підходять інші резерви.

Прикладом холодного резерву, може служити автономний генератор електроенергії, який не працює без будь-якої потреби. Якщо раптово станеться проблема з електростанцією, і вона більше не може забезпечувати електроенергією, генератор електроенергії увімкнеться автоматично та продовжить роботу системи.

1.5.4. Теплий режим очікування

Теплий режим очікування або warm standby відрізняється від холодного резерву тим, що працююча система дублювання виконується паралельно, яка час від часу синхронізує дані з основною системою, так що в разі її резервного варіанту вона зможе підхопити роботу і продовжити її виконання, але тільки з останнього часу резервного копіювання. Якщо

дзеркальне відображення системи виконується часто, це добре для надійності резервного копіювання, але також слід враховувати вартість дзеркального відображення. Вона може виконуватися тільки в той час, коли сама система не перевантажена [35].

Прикладом можуть служити розподілені або NoSQL бази даних. Зазвичай застосунки пишуться таким чином, що сервери не мають стану і вся інформація зчитується з конфіг-файлів або з бази даних. До тих пір, поки репліки бази даних містять найновішу інформацію, не виникає питання про продовження повноцінної роботи з моменту і коли проблема з основною системою буде вирішена, синхронізація всіх даних на основних сховищах.

1.5.5. Гарячий режим очікування

Особливість гарячого режиму очікування або hot standby полягає в тому, що система дублювання працює постійно, одночасно з основною системою. Всі дані віддзеркалюються в режимі реального часу. Через це його ще називають «гарячим запасним».

Це корисно для систем з високою ймовірністю виходу з ладу або з необхідністю періодичного перезапуску. Така система забезпечує не тільки поступову деградацію, але і включення додаткової системи може бути не відчутно багатьма користувачами. Проте часто такі системи реалізуються таким чином, що коли працює вторинна система, вона дає відповіді лише на дії, пов'язані лише з читанням. Запис налаштовується тільки на основній системі, яка ніколи не вимикається протягом тривалого часу [36]. Може бути розміщено повідомлення про тимчасову недоступність запису або правильні запити можуть бути зібрані у форматі que та виконані, як тільки початкова система буде включена і запущена.

Приклади такої системи можна легко знайти в корпоративних мережах, головне, щоб другий включений принтер міг служити реплікацією первинного і виконувати всі його дії в разі перенаправлення запиту

користувачем. Подібним чином будь-який пристрій IoT може копіювати базовий і бути готовим до роботи, коли цього вимагатиме керуюча система.

Гарячий режим очікування дуже нагадує балансування навантаження. Але чи так це насправді? Балансування навантаження – це техніка наявності двох або більше серверів, які можуть обробляти однакові запити. Ідеально, коли вся кількість запитів ділиться порівну між усіма серверами, щоб на кожен сервер надходило однакове число запитів і жоден з них не був перевантажений [37]. Проте ця техніка не потрібна для пристроїв IoT, перш за все, тому, що вони є джерелом інформації та приймають дані лише у вигляді своїх конфігурацій, не витрачаючи достатньо часу на їх обробку [5].

Ідея протоколу резервування загальних адрес полягає в спільному використанні загальної IP-адреси та спільного віртуального ідентифікатора між кількома хостами в одному сегменті мережі. Таке скупчення господарів називається групою резервування. Один хост є головним хостом і йому належить IP-адреса. Інші господарі готові забрати, якщо майстер не відповідає [38] [5].

Протокол резервування віртуального маршрутизатора (VRRP) корисний у випадках, коли фізичне з'єднання з пристроєм може перерватися і цей пристрій буде (тимчасово) замінено на інший, що має свою адресу. Для того, щоб усунути проблеми з ними, можна використовувати віртуальну адресу, яка з'єднає всіх адресатів з робочим елементом через віртуальну адресу, тоді як реальна буде відома лише майстер-маршрутизатору, який керує всіма з'єднаннями. Слід зазначити, що головний маршрутизатор також резервується групою інших маршрутизаторів, готових взятися за його роботу в разі виходу з ладу [37].

Дистанційне вимірювання рівня води може служити прикладом групи приладів, де в залежності від свого стану виконує дію тільки один. Кілька пристроїв IoT можна налаштувати на різній висоті в каналі. Якщо один з датчиків змінить свій стан, наприклад, його затопило, але тепер його немає, рівень води впав нижче необхідного і буде відправлена команда на

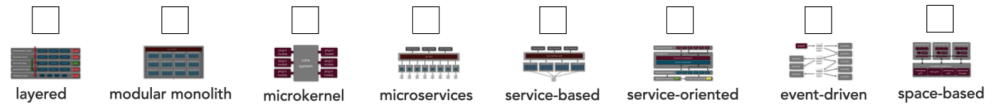
відкриття воріт дамби. Коли вода доходить до іншого датчика, використовуючи ту саму IP-адресу, він надсилає іншу команду, яка полягає в тому, щоб закрити ворота. Цим пристроєм не потрібно мати різні ідентифікатори, оскільки вони виконують одну й ту саму функцію [5].

Для того, щоб будь-яка техніка резервування відбулася, на серверах і в світі IoT зазвичай використовується техніка серцебиття. Справне обладнання посилає своє серцебиття з певною регулярністю на репліку або на сервер управління. У разі будь-якої зміни регулярності серцевих скорочень має діяти заздалегідь підготовлене резервне обладнання. Його можна активувати вручну. У цьому випадку це називається «автоматизована з конфігурацією ручного затвердження» [5].

1.6. Особливості мікросервісної архітектури

Архітектура мікросервісів стала панівною в інформаційному середовищі (ІТ). Вона спрощує розробку додатків, розбиваючи монолітні застосунки на мікросервіси, які можна розробляти та розгортати навіть в декількох екземплярах незалежно від усього цілого. Це хоч і сприяє подовженню часу розробки та впровадження через необхідність забезпечення інтеграції різних модулів, порівняно з монолітними застосунками, але забезпечує можливість використання різних технологічних стеків для окремих компонентів додатка, наприклад, виконувати поступове оновлення системи, а також спрощує заміну окремих частин, дозволяючи безперервно інтегрувати зміни без впливу на роботу всієї системи [39]. Щобільше, така структура програмних застосунків (ПЗ) є найкращим вибором у випадку необхідності масштабування та відповіді на різку зміну кількості запитів до системи в тому числі в режимі реального часу. Детальніше розписано на Рисунку 1.8.

Selected Architecture(s):



partitioning	technical	domain	domain	domain	domain	technical	technical	technical
cost	\$	\$	\$	\$\$\$\$	\$\$	\$\$\$\$	\$\$\$	\$\$\$\$
maintainability	★	★★	★★★	★★★★★	★★★★★	★	★★★	★★★★
testability	★★	★★	★★★	★★★★★	★★★★★	★	★★	★
deployability	★	★★	★★★	★★★★★	★★★★★	★	★★★	★★★★
simplicity	★★★★★	★★★★★	★★★★★	★	★★★★	★	★★	★
scalability	★	★	★	★★★★★	★★★★	★★★★	★★★★★	★★★★★
elasticity	★	★	★	★★★★★	★★	★★★★	★★★★	★★★★★
responsiveness	★★★	★★★	★★★	★★	★★★	★★	★★★★★	★★★★★
fault-tolerance	★	★	★	★★★★★	★★★★	★★★	★★★★★	★★★
evolvability	★	★	★★★	★★★★★	★★★★	★	★★★★★	★★★
abstraction	★	★	★★★	★	★	★★★★★	★★★★	★
interoperability	★	★	★★★	★★★	★★	★★★★★	★★★	★★

Рисунок 1.8 — Особливості розробки та підтримки ПЗ різних видів архітектури [39]

Однак перехід від монолітної або простої багаторівневої архітектури до розподіленої мікросервісної призводить до нових викликів через складнішу топологію додатків в ІТ. Особливою проблемою при автоматичному керуванні продуктивністю мікросервісів є проблеми нерівномірного, перевищеного або неповного навантаження на послуги хостингу через можливість кожного компоненту сервісу масштабуватись за потреби. У більшості сучасних сценаріїв мікросервіси розгортаються як контейнери в кластерах, керованих оркестратором, таким як Kubernetes.

Людина може дійти у своїх роздумах та уявленнях до того, що ніколи не існувало. Розум нам допомагає створювати нове та небувале, але в той самий час речі, які люди намагаються створити, творяться за образом та подобою того, що вже існує. Наприклад, винаходячи літак, людина дивилась на птаха, роботи створюються, щоб повторювати тіло людини та бути на нас схожими, машинний інтелект та нейронні мережі – щоб бути пародією на мозок людини та скомбінувати надможливості мозку у вигляді комп'ютерів із емоційною часткою, яка братиметься з даних про прийняття рішень реальних людей.

В цьому розділі наводиться загальна інформація про мікросервіси, мікросервісні системи порівнюються з елементами справжнього життя, а

також розповідається про проблеми, спричинені мікросервісною архітектурою, такі як ключові задачі консенсусу. Це важливо знати аби розуміти та відповідно планувати розміщення мікропослуг, щоб попередити або не спричинити проблем. Для загального розуміння, паралелі проводяться з усім відомими об'єктами та життєвими ситуаціями, щоб полегшити розуміння, на перший погляд, складних речей.

1.6.1. Мікросервісна архітектура в порівнянні з монолітною

Мікросервісна архітектура – це багатоетапне виробництво. Зараз поширення набрала мікросервісна технологія, яка має під собою розподілену систему, або ж систему, що поділена на функціональні частини, які, зазвичай, можуть реплікуватись у випадку великого навантаження на них з метою ефективного реагування на кількість запитів, що необхідно обробляти. Раніше натомість застосунки розроблялись як моноліти, які обробляти стільки запитів, скільки обробляло їх найвужче місце. Також існує проміжний варіант — множення одного монолітного сервісу на декілька додаткових серверів. Іншими словами, повноцінне програмне забезпечення існуватиме в декількох репліках. При організації рівномірного розподілення навантаження, це дозволить підвищити відмовостійкість застосунку, і, таким чином, пропускна можливість підвищиться. Мінусом даного підходу є менш ефективне використання ресурсів ніж у випадку мікросервісної архітектури, де, за потреби, збільшується тільки кількість необхідних елементів системи, а не створюється додаткова повноцінна система [6].

В сучасному світі, незабаром після того, як людство освоїло певну технологію, з'являються покращені версії цієї технології, або відбувається адаптація ідеї в інші сфери вжитку. Цікавим є те, що все нове насправді є старим і далеко не завжди забутим, те саме вже десь існувало, але в інших іпостасях, інші, і не обов'язково вчені, ці ж проблеми вже вирішував, тільки не завжди у тому самому середовищі.

Ідею мікросервісної архітектури можна порівняти зі звичайним виробництвом, наприклад, сирним. Залежно від етапу виробництва, потрібна різна кількість місткостей та місця для пастеризації молока, відстоювання його після скисання та витримки сиру. Моноліт можна проасоціювати, з виробництвом, що наявне в одному приміщенні, де підтримується різна температура залежно від етапу. Якщо етап витримки сиру займає багато часу, то виробництво чекатиме, доки даний етап не закінчиться.

Мікросервісною архітектурою було б створення різних кімнат, можливо, навіть, приміщень, для відповідних процесів з необхідною для них температурою та кількістю місткостей. Тоді б можна було в режимі реального часу приймати нове молоко та починати його обробляти. У випадку збільшення поголів'я корів, можна було б додатково розмістити більшу кількість місткостей в одні кімнати, а якщо розміри не дозволяються, побудувати нові зали, призначені для певного процесу. Відзначу, що зважаючи на об'єми різного роду сировини, додаткові приміщення можуть знадобитись для скисання молока, адже пастеризація проходить швидше, ніж скисання, та наявних потужностей може вистачити. Сир не займає багато місця, а отже і для його зціджування можливо вийде обійтись лише додатковими стелажми. Якщо кількість вхідного молока зменшуватиметься, додаткові кімнати та місткості можна не заповнювати сировиною, а отже і не витратити додаткові кошти на підтримку робочого стану. Словами мікросервісної архітектури, збільшується чи зменшується кількість робочих серверів, виділених на певний мікросервіс, або ж залучаємо додаткові потоки для обробки вхідних запитів [6].

1.6.2. Особливості комунікація між мікросервісами

Розгляньмо задачу гарантованої комунікації або двох генералів. Добре, коли новий процес працює, полегшує життя та робить його якіснішим. Але ж зазвичай, з ускладненням самого виробництва, необхідно

створювати нові та додаткові процеси. Наприклад, розташувавши різні частини виробництва в різних приміщеннях, необхідно організувати транспорт між приміщеннями, що вимагатиме не тільки додаткових вкладень, але і ставитиме під загрозу саме виробництво, у випадку неможливості транспортування через відсутність бензину, електрики, поломки транспортних засобів, сторонніх подій, що заважатимуть руху тощо. Проводячи аналогію з мікросервісною архітектурою, потрібно організувати відправлення повідомлень між частинами застосування. Це і собі створює питання доставляння сповіщень, а також їх правдивість [15].

Проблема транспортування сповіщень між елементами розподіленої системи опублікована ще у 1975 р. у [40] та отримала назву «Задача двох генералів» у 1978 р. Суть полягає в тому, що два генерали атакують спільного ворога, армія якого розташована між арміями спільників. Успіх буде тільки у випадку одночасної атаки, про яку слід домовитись. Головний генерал має надіслати іншому сповіщення про атаку, а також отримати від нього підтвердження. Далі можливі дві негативні ситуації: до другого генерала не доїде гінєць, або ж сповіщення буде отримано, але перший генерал не отримає підтвердження через перехоплення гінця на зворотному шляху. Таким чином, неможливо гарантувати, що обидві армії підуть в атаку на ворога в один і той самий час.

Вище наведено Рисунок 1.9 з [40], що описує наведену проблему мовою мікросервісної архітектури. P1 (partition 1) та P2 (partition 2) – це окремі мікросервіси, між якими має відбутись комунікація.

IPCM – inter process communication mechanism, тобто механізм кожної сторони, через який відбувається комунікація, наприклад, REST протокол. Network – мережа, яка з одного боку служить шляхом комунікування, а з іншого, – може порватись в будь-який момент, тому асоціюється з ворожою армією.

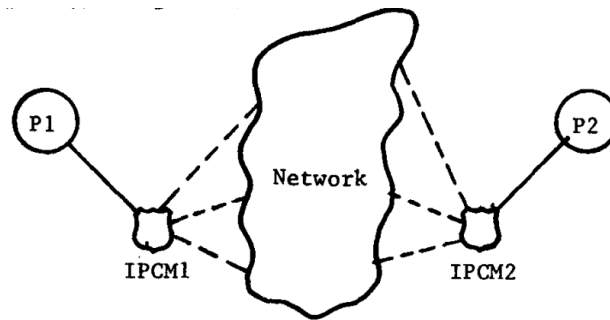


Рисунок 1.9 — Задача двох генералів у розподіленій мережі [40]

Задача гарантування отримання відповіді відправником не має розв'язку. Проте, ситуація достатньо поширена тому нижче наводяться способи суттєво зменшити вірогідність неотримання відповіді.

1) Побудувати безпечний канал зв'язку між двома арміями, але це додаткові ресурси.

2) Замість надсилання одного гінця надіслати багато, наприклад, 100. Інформацію вважати прийнятою у випадку отримання однієї чи декількох відповідей. Декілька відповідей може знадобитись коли сповіщення може бути підроблене по дорозі.

3) Заради оптимізації ресурсів, можна відправляти кожного наступного гінця після того, як пройшов час, за який мав повернутись перший.

Додатково, щоб зрозуміти якість каналу, можна нумерувати сповіщення та робити висновки з отриманих результатів.

Задача неправдивих сповіщень або візантійських генералів. Окрім питання доставлення сповіщень, постає питання правдивості отриманих сповіщень. Якщо зв'язок погано захищений, то по дорозі можна підробити сповіщення. Тим не менш, така проблема може статись навіть при захищеному з'єднанні, коли в одного мікросервіса чи розподіленої бази даних з різних причин застаріла інформація, яка вже неактуальна. Тобто, необхідно гарантувати, що або ж інформація буде останньою, або даний вузол системи не братиме учать в консенсусному прийнятті рішення.

Проводячи паралель з реальним життям – можемо уявити армію, що складається з багатьох легіонів. Головний генерал, який може бути зрадником, має дати наказ генералам легіонів або нападати або відступати. Зважаючи на те, що субординати можуть теж бути зрадниками, і їх може бути до третини, успіх буде тільки у випадку нападу всіма чесними генералами. Таким чином, накази головнокомандуючого слід перевіряти, а без комунікації війна буде програною. Дана задача називається «Задачею візантійських генералів» і також не має розв’язку. Така проблематика була описана у 1985 році у [41] та ставила під питання невалідність тільки одного вузла системи. Тим не менш, схожу задачу розглядали ще з 1950х років у NASA та у авіаційній галузі з метою створення бортових комп’ютерів, які б мали

- декілька реплікацій даних та окремих обчислювальних машин, а також,
- паралельні процеси однакових обчислень, які б гарантували правильність висновків та максимально унеможливлювали помилку,
- шукали та знаходили помилку методом порівняння результатів та голосуванням (більшість однакових результатів вказує, що це правильний результат)
- виправляли помилку,
- реконфігурувались, щоб позбавитись зламаних вузлів та не брати їх «думку» до уваги [42].

У наведеній статті [42] використовувалось шість реплікацій для забезпечення коректної роботи тогочасних літаків, які обмінюються станом між собою кожні 50мс, але в загальному випадку реплікацій може й не потрібно стільки. Наприклад, в Європейській організації з ядерних досліджень, що знаходиться в Женеві, використовується три репліки даних, які розташовані в трьох різних країнах на випадок війни та бомбардувань. Варто зазначити, що кількість даних величезна, адже йдуть записи щомоментного розташування частинок в адронному колайдері.

1.6.3. Недоступність частин мікросервісної системи

Паксос протокол та модель частково парламенту, що працює. Існує не один алгоритм, що допомагає розв'язанню проблеми візантійських генералів. Нижче наводиться алгоритм, який розв'язує проблему нерегулярності роботи окремих вузлів та оповіщення новими законами депутатів, які час від часу виходили з зали засідань. Стаття, що описує даний алгоритм була надрукована у 1989 Леслі Лампортом, але до неї поставились серйозно пізніше, коли була надрукована стаття без порівняння процесу з грецьким парламентом на острові Паксос.

Ідея алгоритму полягає у тому, що за допомогою порівняння поточного стану системи та метаінформації про дані на етапі їх зчитування з різних реплік, шляхом голосування приймається рішення, які саме дані валідні, якщо вони не співпадають.

Нижче описується реалізація запису та читання інформації за Паксос алгоритмом.

Існує три ролі:

- Пропонувач – пропонує закони або ж висловлює бажання запису певної інформації, нумеруючи її у зростальній послідовності.
- Приймачі – депутати, які розглядають нові закони, або ж вузли системи, які отримують сповіщення. Розгляд відбувається тільки у випадку, якщо є кворум (більшість справних вузлів системи).
- Секретар (учень) – який, бере-запиує виключно прийняті закони, та віддає їх кінцевому користувачу [43].

Метод, через який відбувається голосування заснований на порівнянні максимального номера вже чинного запису в пам'яті вузла. Тобто, якщо новий номер більший за вже наявний, приймач відповідає обіцянкою не розглядати сповіщення з меншим номером, а якщо ні – приймач відхиляє запис та у відповідь надсилає максимальний наявний в нього номер. Якщо більшість приймачів відповіла обіцянками прийняти сповіщення, воно буде розіслане з запитом «прийняти» усім наявним вузлам, і вони мають його

прийняти, якщо попередньо не відповіли, що мають вищий номер. Сповіднення відсилається клієнтам тільки після того, як воно було прийняте кворумом [44].

1.7. Види групування ресурсів

Групування ресурсів відбувається в хмарних технологіях у різних провайдерів, а також на Kubernetes. Метою можуть бути різні ідеї. В пунктах цього підрозділу будуть розглянуті різні види групування та зазначена їх мета.

1. Групування заради безпеки у мережеві групи (Security Groups) використовується для управління вхідним та вихідним трафіком, наприклад для заборони доступу до бази даних із зовнішнього інтернету (доступ лише для додатків із конкретного підмережевого сегмента). Виконується створення віртуальної приватної мережі та обмеження підключень лише з певних IP-адрес [45].

2. Групування за зручністю розміщення (Colocation) – виконується для розміщення компонентів, які часто взаємодіють, у тому самому регіоні або зоні доступності для зниження затримок. Може виконуватись розділення підмереж на публічні (для доступу з інтернету) та приватні (для внутрішніх операцій), наприклад, фронтенд знаходиться у публічній підмережі, а бекенд і база даних – у приватній. Це також може бути об'єднання в кластер (Clustering) для серверів або контейнерів, які працюють разом, наприклад, Kubernetes або ECS кластери [46].

3. Групування через життєвий цикл ресурсів на основі міток на цих ресурсах (Tagging) для зручності пошуку, фільтрації та організації, або групування ресурсів за середовищами (розробка, тестування, продакшн) для ізоляції та мінімізації впливу на основні робочі сервіси.

4. Групування для масштабованості (Scaling Grouping) тих ресурсів, які мають разом масштабуватись, додаватись або видалятись. В Load

Balancing виконується розподіл трафіку між інстансами чи контейнерами, які працюють у групі, для забезпечення високої доступності [47].

5. Групування за типом даних чи задач виконується як розділення ресурсів на рівні мікросервісів (наприклад, окремі групи для API Gateway, обробки запитів чи баз даних). Інша варіація – групування за зонами зберігання. Воно містить організацію даних за рівнем доступу чи швидкістю (гаряче, тепле, холодне зберігання, як у S3).

6. Групування для спрощення обслуговування або оркестрації (Maintenance Grouping), наприклад для деплою (впровадження) чи оновлення програмного забезпечення (наприклад, у CodeDeploy чи Kubernetes).

7. Групування через фінансовий контроль з формуванням Billing Groups, що є розподілом ресурсів за командами, проектами чи замовниками для моніторингу витрат.

Це всі основні поширені види групування, і в їх перерахунку нема групування за повним вжитком ресурсу, про що йтиметься далі в роботі.

1.8. Особливості масштабування

При збільшенні кількості користувачів, або ж даних, які треба обробляти, може виникнути вузьке місце у застосунку, результати роботи якого всі чекатимуть. Щоб запобігти такій ситуації, або ж її вирішити в режимі реального часу, слід продумати механізм масштабування.

Одним зі способів дати відповідь на висхідну кількість запитів та зменшити навантаження на основний сервер є застосування **кешування** на рівні сервера або зовнішнього кеш-сервера, наприклад, Redis або Memcached. Кешування дозволяє прискорити роботу додатків, зберігаючи часто використовувані дані в пам'яті та зменшуючи кількість звернень до основної бази даних або серверного застосунку.

Іншим способом є масштабування через додаткову інфраструктуру – *Content Delivery Network (CDN)* та проксі-сервери забезпечує розподіл

статичного контенту на сервери ближче до кінцевих користувачів, знижуючи навантаження на основний сервер. На Рисунку 1.10 зображено основний сервер червоним та локальні сервери зберігання статичних даних синім. Користувачі зображені як комп'ютери.

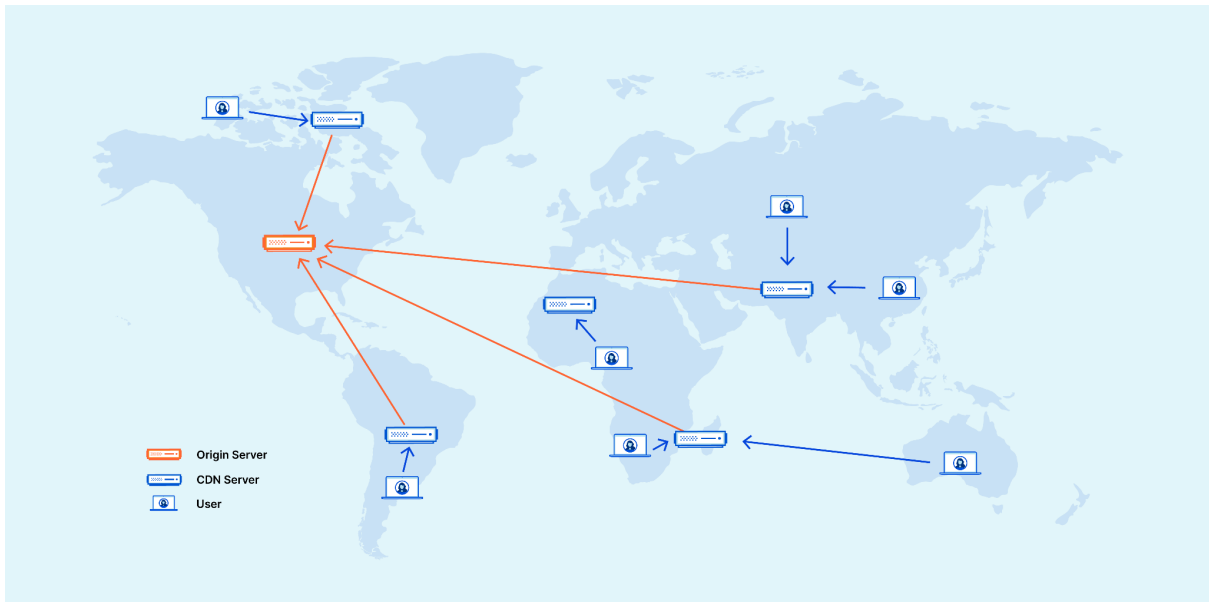


Рисунок 1.10 — Приклад роботи Content Delivery Network [48]

Використання зворотних проксі-серверів для обробки запитів може допомогти збалансувати навантаження, обробляючи запити та передаючи їх до різних серверів.

Говорячи про динамічне масштабування, тобто забезпечення оперативної відповіді на основі актуальних даних, або ж обробки інформації та відповідь, слід зважати на необхідність розширення технічного устаткування, відданого під програмне забезпечення. Про це йтиметься в цьому підрозділі.

Масштабування на основі черг може бути ще одним способом абсорбції стрибків кількості запитів, якщо їх обробка може виконуватись у фоновому режимі. Якщо система не справляється, вона динамічно збільшує та потім зменшує кількість обробників запитів у черзі залежно від обсягу повідомлень. Даний підхід використовується для обробки запитів

асинхронно за допомогою черг повідомлень таких як Amazon SQS, RabbitMQ та ін.

1.8.1. Вертикальне масштабування

Вертикальне масштабування – це збільшення ресурсів сервера, таких як процесор, пам'ять (RAM) або обсяг дискового простору. Для прикладу, можна перейти з менш потужного VPS на потужніший або замінити виділений сервер на більш потужний.

У цього підходу є фізичне обмеження. Якщо додаток досягає максимальних можливостей одного сервера, потрібно переходити до горизонтального масштабування. Інша особливість — необхідність постійно використовувати більший ресурс, навіть, якщо в поточний момент нема потреби, адже вставивши раз більшу пам'ять в сервер не можна буде її вийняти не вимикаючи ЕОМ та дати більше пам'яті сусідньому серверу на час його пікового навантаження.

Говорячи про хмарних провайдерів, деякі, наприклад, AWS, Google Cloud, Azure) дозволяють вертикальне автоматичне масштабування на основі метрик, тобто змінювати розмір віртуальної машини в режимі реального часу або з мінімальним простоєм. Для цього необхідно зупинити інстанс на кілька секунд, вибрати інший тип інстансу з більшою кількістю ресурсів і перезапустити. Можуть бути затримки в обслуговуванні в процес перезапуску. Слід зважати, що це робиться автоматично доходячи тільки до певного розміру оперативної пам'яті, та не завжди можна повернутись до попередніх показників [49]. Також можна збільшити розмір диска і відразу під'єднати його до сервера без перезавантаження.

Деякі системи підтримують функцію гарячого додавання (Hot Add) оперативної пам'яті та гарячого включення (Hot Plug) процесорного ресурсу CPU, які дозволяють додати ресурси без зупинки сервера або віртуальної машини на базі VMware або Hyper-V [50], [51], [52], але для деяких ОС перезавантаження буде необхідним. При виборі цього режиму,

не буде виконуватись оптимізація оперативної пам'яті vNUMA в VMWare. Важливою особливістю даної технології є відсутність можливості прибрати зайві ресурси після додавання в автоматичному режимі.

У випадку, коли інфраструктура побудована за допомогою контейнерів, наприклад, Kubernetes, можна динамічно виділяти більше або менше ресурсів CPU та RAM контейнерам через інструмент Vertical Pod Autoscaler (VPA) на основі їх метрик. Тоді кластер автоматично розподіляє додаткові ресурси. Графічне представлення процесу можна побачити на Рисунку 1.11. Виділення нових ресурсів потребує перезапуску подів з новими параметрами request і limit – бажаний та максимальний обсяг використання ресурсу, заданий у відсотках процесорних одиниць чи кількості [49], [53], [53].

Процес виглядає наступним чином. Нехай є под, який обробляє потоки даних. На початку він може використовувати мінімальні ресурси, але під час пікових навантажень потребує значно більше CPU. З VPA поди автоматично отримують більше CPU та пам'яті під час зростання навантаження, а коли навантаження спадає — знижуються до економних обсягів ресурсів.

VPA зручний для додатків, котрим масштабування треба рідко або непередбачувано, та котрі не передбачають додаткового горизонтального масштабування. Даний підхід забезпечить стабільність роботи додатків. Вертикальне та горизонтальне масштабування важко налаштувати разом, бо вони можуть конфліктувати.

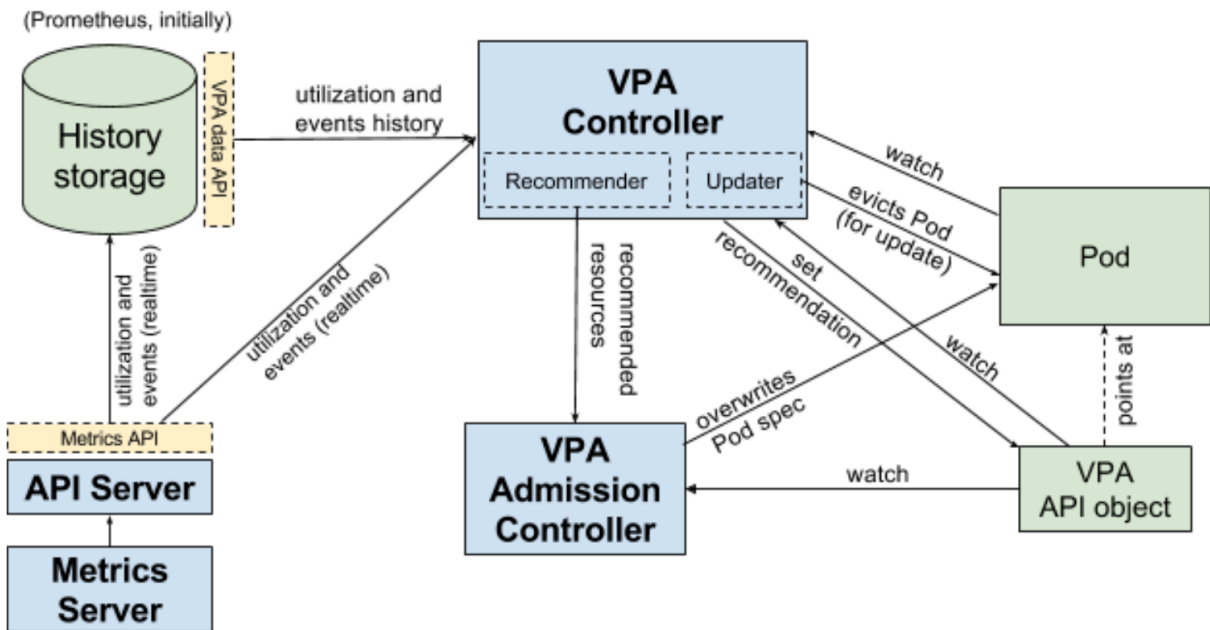


Рисунок 1.11 — Модель вертикального масштабування у Kubernetes [49]

1.8.2. Горизонтальне масштабування

Горизонтальне масштабування полягає у додаванні нових серверів для розподілу навантаження між кількома машинами. Зазвичай для цього використовується *балансувальник навантаження*, який рівномірно спрямовує трафік на декілька серверів, щоб запобігти перевантаженню одного з них.

У традиційному хостингу горизонтальне масштабування потребує ручного налаштування. Необхідно налаштувати балансування, резервне копіювання та синхронізацію даних між серверами. Додаються нові сервери чи інстанси адміністратором хостингу, коли це потрібно. Такий варіант підходить для невеликих проєктів або систем з передбачуваним навантаженням [54].

Говорячи про горизонтальне масштабування для баз даних, які здебільшого є невідійманою частиною багатьох додатків, можна реплікувати базу даних на кілька серверів або замість одного великого сервера базу даних розподілити на кілька менших серверів, що зменшить навантаження та підвищить продуктивність. Цей процес називається *шардінгом*.

Автоматизовані хмарні рішення підходить для додатків з непередбачуваним або змінним навантаженням. Система автоматично додає чи видаляє екземпляри на основі налаштованих метрик, таких як завантаження CPU, пам'яті або кількість запитів. Використовується в хмарних платформах, таких як AWS, Google Cloud та Azure, де є вбудовані механізми для автоматичного масштабування (наприклад, Auto Scaling Group в AWS).

Як аналог клауд-системи використовуються середовища контейнеризації, такі як Kubernetes або Docker Swarm, де нові екземпляри додатків запускаються як контейнери. Kubernetes має вбудовану функцію Horizontal Pod Autoscaler (HPA), яка автоматично додає поди на основі метрик завантаження [55], [56]. Даний варіант масштабування підходить для мікросервісної архітектури, оскільки дозволяє масштабувати кожен компонент незалежно. Схема такого масштабування зображена на Рисунку 1.12.

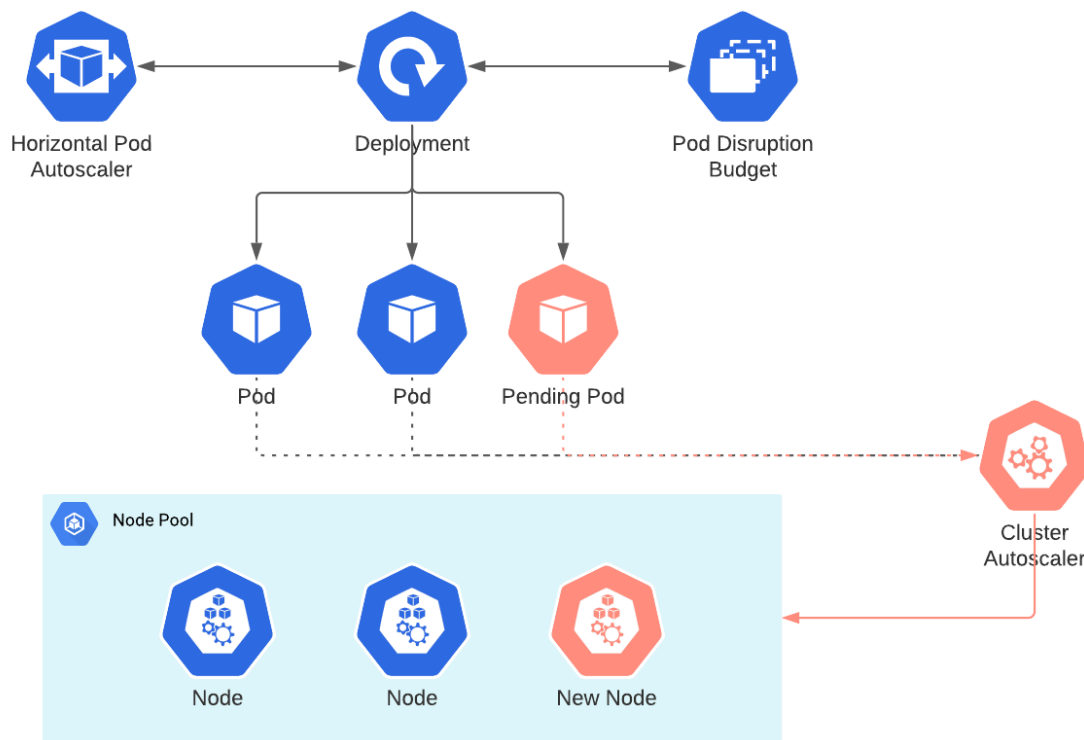


Рисунок 1.12 — Схема горизонтального масштабування в Kubernetes [56]

Горизонтальне масштабування в Kubernetes автоматично активується у ситуації, коли под намагається перевищити свій ліміт. Тоді контейнер отримує обмеження або може бути завершений через механізми контролю ресурсу, наприклад, cgroup в Linux. В такому разі НРА додає нові поди, щоб зменшити навантаження та повернути перевищені ліміти у норму.

Окрім масштабування подів існує ще Cluster Autoscaler – масштабувальник, який працює на рівні вузлів. Якщо кластер не може розмістити нові поди через брак ресурсів, він додає нові вузли. При зменшенні навантаження, він видаляє вузли, якщо вони не використовуються.

Окрім додавання серверів у тому самому хмарному дата-центрі, є опція *географічного масштабування*, при якому нові сервери додаються в різних регіонах для забезпечення більш швидкого доступу до даних користувачам у різних частинах світу. Відповідно, використовуються *глобальні балансувальники навантаження* та розподілена архітектура CDN (Content Delivery Network). В результаті, зменшуються затримки та підвищується доступність, особливо для додатків з глобальною аудиторією.

1.8.3. Горизонтальне масштабування мікросервісів, монолітів та баз даних

Найзручніше горизонтальному масштабуванню піддається мікросервісна архітектура. Контейнерне горизонтальне масштабування дозволяє незалежно масштабувати кожен мікросервіс залежно від навантаження, забезпечуючи ефективніше використання ресурсів. В разі необхідності географічне масштабування допомагає підтримувати низькі затримки й високу доступність, обслуговуючи користувачів у різних країнах із найближчих дата-центрів.

Монолітні застосунки складніше горизонтально масштабувати, оскільки вони зазвичай являють собою цілісні, нероздільні системи, в яких

усі компоненти (UI, бізнес-логіка, база даних) тісно пов'язані та розгортаються разом. Проте це можливо.

Найпоширеніший спосіб горизонтального масштабування монолітних застосунків — це додавання декількох серверів із балансуванням навантаження. Балансувальник розподіляє вхідні запити між кількома екземплярами моноліту, щоб зменшити навантаження на кожен сервер. Такий спосіб підходить, якщо застосунок може працювати незалежно на кількох серверах, але важливо забезпечити коректне управління станом (state management).

Багато монолітних застосунків зберігають стан у пам'яті, наприклад, сесії користувачів, що ускладнює горизонтальне масштабування. Щоб розв'язувати цю проблему, стан користувача можна зберігати поза самим застосунком, у централізованому сховищі (наприклад, Redis, Memcached) або в базі даних. Це дозволяє будь-якому екземпляру моноліту обробляти запити від будь-якого користувача, оскільки стан доступний усім екземплярам.

Централізована БД монолітного застосунку теж може стати вузьким місцем під час горизонтального масштабування. Щоб обійти це обмеження, можна скористатись *реплікацією бази даних*. Це дозволить створити копії даних для читання на кількох серверах, розподіляючи навантаження між ними (наприклад, основна база для запису та кілька реплік для читання). Вище згаданий шардінг теж може бути доречним.

Монолітний застосунок можна запакувати в контейнер (Docker), що дозволить масштабувати його в контейнеризованому середовищі, як-от Kubernetes і звести задачу до вже описаної вище за умови відсутності внутрішнього стану застосунку.

У випадках, коли тільки певні частини моноліту перевантажені, може мати сенс провести розбиття моноліту на незалежні сервіси (Microservices Transition) та розв'язати питання горизонтального масштабування таким чином, або ж задовольнитись вертикальним масштабуванням.

Отже, схожому масштабуванню, наприклад, через контейнери, піддаються як мікрсервіси, так і моноліти в разі розв'язаного питання з внутрішнім станом та централізованою базою даних. Слід також зважати на можливість справитись з надвисокими навантаженнями на монолітний застосунок більш традиційними методами такими як кешування та CDN. Проте в довгостроковій перспективі перехід до мікросервісної архітектури є ефективнішим рішенням для масштабування.

1.9. Проблеми планування задач у багатосерверній інфраструктурі

Планувальник Kubernetes є дуже потужним та продуманим інструментом, здатним справлятися з будь-якими навантаженнями. Це забезпечується шляхом вдалого налаштування рекомендованих та граничних навантажень, горизонтального та вертикального масштабування як подів, так і кластерів в режимі реального часу та балансування навантаження між запущеними дублювальними подами (див рис 1.13 та 1.14).

Одночасно, є певні незручності пов'язані з використанням ресурсів, які виділяються на под в Kubernetes.

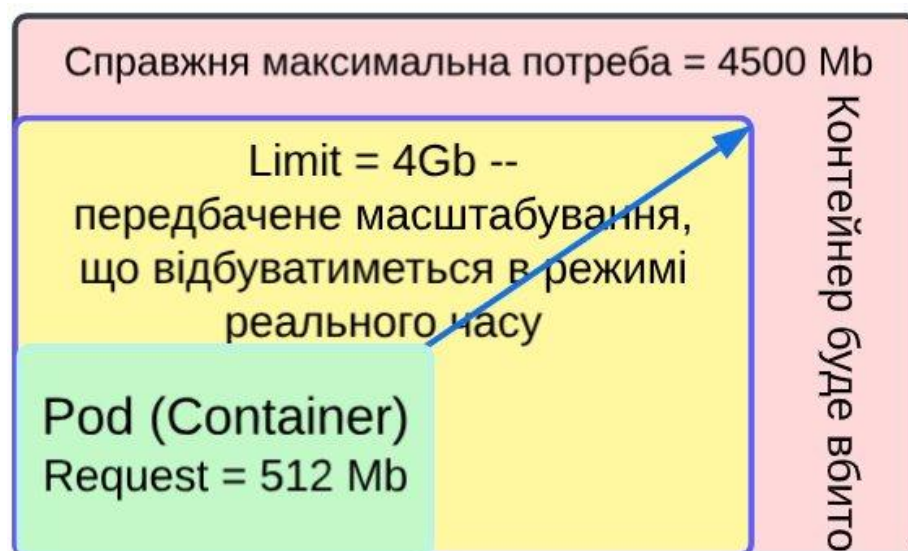


Рисунок 1.13 — Проблема запиту на використання більшої кількості ОП, ніж було написано в лімітах, яка призводить до виокремлення проблемного інстансу

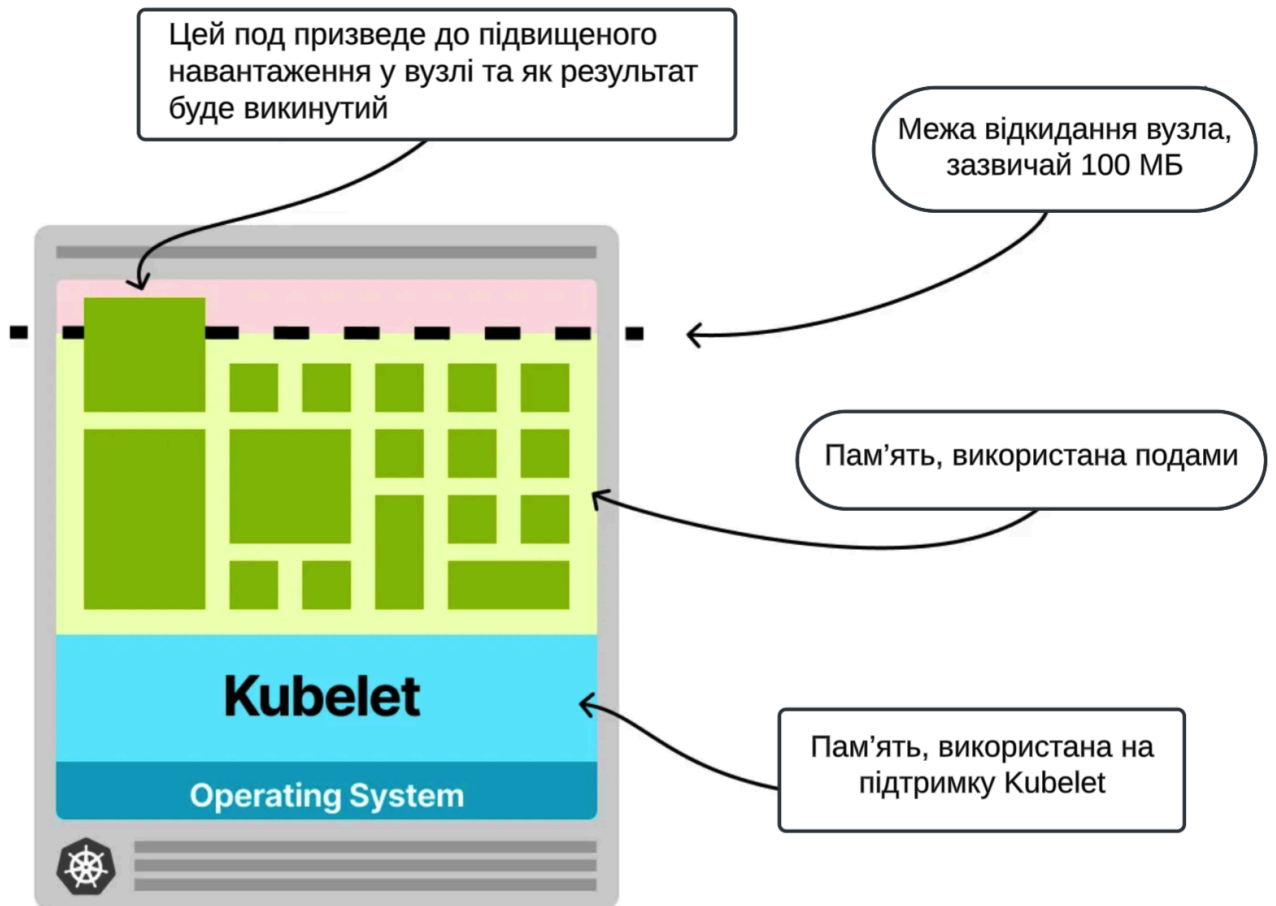


Рисунок 1.14 — Вилучення пода через недостатню кількість ресурсу

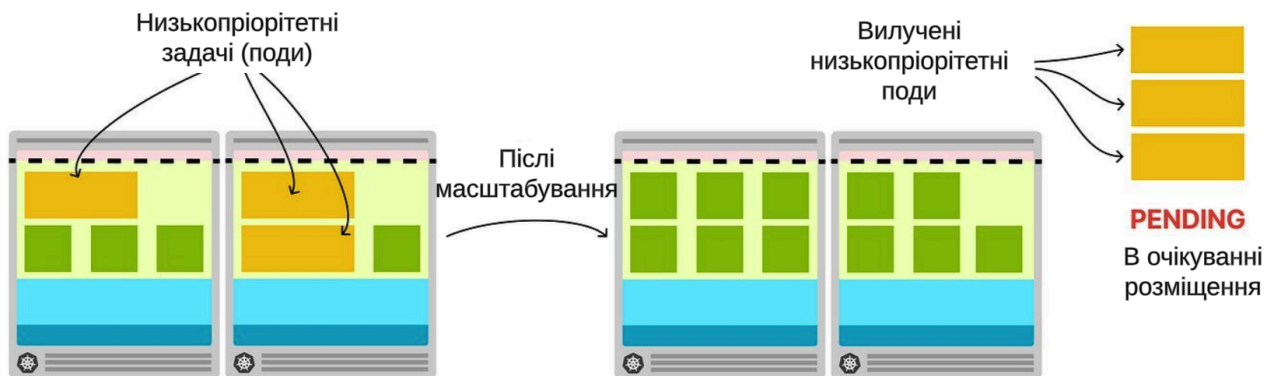


Рисунок 1.15 — Ріст подів, яким не було місця та очікування викинутих програмних застосунків в черзі задач на розміщення

1.9.1. Проблема вивільнення неживаної оперативної пам'яті

До першої проблеми можна віднести складнощі адаптації використовуваної ОП до зменшення її вжитку. Якщо процесорний та каналний ресурс можуть адаптуватись під кількість навантаження, незалежно, чи вона спадна, чи висхідна, то оперативна пам'ять не буде відпущена подом без явної команди

або його перезапуску. Таким чином, після першого пікового навантаження, ресурс оперативної пам'яті буде зарезервований на максимум, хоча використання може стрибнути до мінімального. Це тягне за собою перевищені грошові витрати замовника хостингу коли він на хмарі.

1.9.2. Проблема різниці між заданою та необхідною кількістю ресурсів

При створенні пода є потреба задавати бажане (request) та максимальне значення (limit) використання ресурсу для його роботи, які в загальному випадку можуть сильно відрізнятися. Існує три стратегії визначення запитів та лімітів на Kubernetes [57], [58]:

1. *Guaranteed* (гарантоване розміщення) – бажана кількість CPU та RAM дорівнює максимальній. Така модель підходить для баз даних. Под не буде викинутий при перевантаженні вузла, доки його ресурси задовольняються. Він має найвищий пріоритет серед усіх класів. Недоліком є відсутність гнучкості – контейнер не може споживати більше ресурсів, ніж задано в обмеженнях.

2. *Burstable* (еластичний рівень) – коли заданий запит, а лімітів може й не бути, тобто застосунок розширюватиметься допоки не закінчиться ресурс вузла, але якщо вузол перевантажений, Kubernetes може завершити цей под на користь *Guaranteed*.

3. *BestEffort* (найнижчий рівень або найкраща спроба) – не має ні бажаної кількості ресурсу, ні максимальної. Под отримує ресурси тільки якщо вони доступні. Якщо вузол перевантажений, такі поди будуть першими на видалення (Eviction) [59].

Розуміючи нестачу ресурсів слід налаштувати автомасштабування:

- а) задавати недостатню кількість ресурсів вузла (Requests) априорі та налаштовувати горизонтальне масштабування для задоволення Limits;

b) задавати більший запит ніж середньо-мінімальна вживаність для уникнення частого масштабування, особливо якщо є ймовірність частих збільшених потреб;

Результати такої поведінки призведуть до наступного:

a) Необхідність попередити перевантаженість сервера, при якій задача ще не вичерпала свій ліміт, але не може отримати більше ресурсу, через що знижується продуктивність та з'являється можливість її краху.

b) Задачу буде викинуто з вузла при перевантаженні сервера, та якщо у неї низький пріоритет.

c) Задача не знайде нового місця та буде в черзі на очікування розміщення.

Кожен варіант неприємний, тому якщо можна попередити поведінку викидання робочих задач з вузла, при цьому надавши їм необхідну кількість ресурсу, краще так зробити. Отже, стратегія роботи з запитами та лімітами має бути модифікована або замінена, щоб попередити проблеми з ОП та не мати необхідність постійного масштабування або пошуку місця викинутим з серверів задачам. Це буде продемонстровано в наступних розділах.

ВИСНОВКИ

З кожним днем кількість розподілених систем стає все більше й більше, адже власники бізнесів хочуть бути впевненими, що їх продукт витримає усяке навантаження та залишиться працювати навіть у випадку втрати частини серверів. Питання вирішення конфліктних ситуацій у розподілених системах залишається актуальним вже 70 років, та з розвитком інформаційних технологій з'являються нові нюанси, ідеї та потреби. Мікросервісна архітектура та безсерверна технологія покликані розв'язати питання масштабування та відмовостійкості систем обслуговування. Одночасно її розробка лишається високовартісною. Тому рішення масштабування та оптимального розподілу ПЗ треба розробляти універсальними, щоб і монолітні застосунки, які є одними з найскладніших для скейлінгу, теж могли брати в ньому участь.

1. Класичний варіант хостингу стає все менш популярним через необхідність втручання оператора у випадках необхідності масштабування або заміни ПЗ. Все більше компаній віддає перевагу хостингу на хмарі, де процеси балансування, масштабування та планування задач автоматизовані та іноді існують власні програмні рішення, оптимізовані під роботу на цій хмарі.

2. Масштабуванню найкраще піддаються безсерверні програмні застосунки та ті, що без стану, тобто так звані лямбда-функції та мікросервіси. Монолітні застосунки теж можуть бути масштабовані, але в разі наявності стану цей процес може бути складним через необхідність перенесення цього стану на масштабовану копію. Є технології адаптації монолітів до кращого масштабування.

3. Існує багато варіацій балансувальника в різних хмарах, але за принципом дії виділяють два основні:

- який працює на 4-тому рівні OSI, тобто на рівні мережі, а отже забезпечує низьку затримку і високу пропускну здатність. Його метою є

запобігання виснаженню з'єднання перед тим, як ресурс вважатиметься нездоровим, і рівномірний розподіл трафіку між іншими нодами.

- який працює на 7-мому рівні OSI (рівень прикладних протоколів) та підтримує маршрутизацію на основі хоста та шляху, тобто трафік спрямовується на основі його вмісту або його заголовків, доменного імені. Групування запитів здійснюється за близькими локаціями. Можна налаштовувати правила маршрутизації, щоб направляти трафік до різних груп цілей. Підходить для передачі даних в режимі реального часу.

4. Балансувальники розділяються за зоною дії на глобальні та локальні, та працюють використовуючи протоколи HTTP, UDP, TCP, SSL.

5. При масштабуванні є варіант завчасної підготовки сервісів до ефективної роботи: гаряче, тепле та холодне очікування (hot, warm, cold standby). Тоді стан та дані з робочої задачі дублюються на таку саму обчислювальну задачу, що в очікуванні. Такий підхід мінімізує чи попереджає затримки в обслуговуванні в разі старту дублювального інстансу при великому навантаженні на перший інстанс.

6. Задачами планувальника є рівномірний розподіл задач по ресурсах враховуючи їх доступність та обмеження вузлів і задач (taints, tolerations, affinity, anti-affinity), планування кількості вузлів залежно від навантаження та моніторинг запущених завдань з їх повторним плануванням за потреби.

7. Аналіз планувальників найпоширеніших хмар GCP, AWS, Azure дозволяє виділити наступні основні види планувальників:

- Kubernetes загальнопоширений на хмарах та у класичних дата-центрах, доступний для локального встановлення та тестування в домашніх умовах. Існують хмарні аналоги, наприклад, ECS;
- Managed Instance Groups (MIGs) — планувальник віртуальних машин від Google, яким також користуються такі хмарні лідери як Amazon та Azure. Він відповідає за масштабування та балансування за стандартними налаштуваннями;

- планувальники для безсерверних програм;
- власні планувальники хмар, більш адаптовані під конкретну хмарну архітектуру, і які можуть суміщати всі ці види планувальників.

8. Загальний алгоритм роботи планувальника, який є спільним для основних хмар:

- оцінити список вхідних подів (завдань) з їх запитами (Requests & Limits) CPU та RAM, й обмеженнями щодо характеристик вузлів, кластерів та сусідніх подів (affinity, antiaffinity, traints, tolerations)
- підібрати необхідний кластер та перевірити наявність місця на вузлах кластер, пріорітезувати вузли;
- розмістити под на вузлі, який набере найбільший бал.

9. Основні види стратегій розміщення вузлів наступні:

- spread — розподіл за принципом мінімального завантаження вузлів з гаслом «один под на ноду». Мета: рівномірне покриття вузлів (ECS від Амазон);
- рівномірного (Kubernetes Scheduler);
- bin packing — максимального використання вузлів. ECS може обрати розміщення на найбільш завантаженому вузлі для уникнення запуску зайвих вузлів та мінімізації простою;
- також ECS може виконувати оптимізацію під специфічні ресурси такі як GPU або вузли з низькою затримкою.

Отже, стратегії групування за змінною в часі кількістю ресурсу, що вживається, нема.

10. Основні способи енергоощадження на різних хмарах:

- вимкнення зайвих інстансів, користування менш витратними серверами (стратегія scale out, а не scale up);
- вибір найближчої зони доступності хмарного сервісу для розміщення нодів;

- використання Spot Instances — дешеві, незалучені ресурси для тимчасових завдань або вузлів з меншою продуктивністю для нетермінових задач;
- використання чіпів меншого енергоспоживання (Graviton на базі ARM);
- використання екологічних систем охолодження на основі відновлювальних ресурсів.

Отже, змінити підхід до планування для досягнення енергоощадження це нова ідея.

11. Варіант розміщення декількох задач на одному поді не дуже поширений, хоча існує опція «Sidecar». Її застосовувати можна в наступних випадках:

- коли один контейнер забезпечує допоміжну функцію для іншого (логування), що також називається спільним життєвим циклом;
- коли компоненти мають потребу в спільному ресурсі (БД);
- заради зменшення затримки комунікації між сервісами.

Отже, розміщувати декілька задач на одному поді буде коректним, якщо вони чимось пов'язані. Про пошук задач, які пов'язані не спільною ідеєю, а споживанням ресурсу, яке доповнює одне-одне, йтиметься у наступних розділах.

12. Способи групування інстансів можуть бути за наступних причин:

- спільний життєвий цикл;
- норми безпеки;
- одночасна потреба в масштабуванні;
- використання спільного ресурсу;
- за типом задач, швидкістю задач;
- за зонами вжитку;
- для спрощення оркестрації;
- з причини фінансового контролю.

Отже, групування за доповнювальним використанням ресурсу нема.

13. Масштабування буває горизонтальним — створення реплік обчислювальних задач, та вертикальним, тобто збільшення ресурсів в межах одного сервера для тої самої задачі. Перше є дешевим та поширеним, друге рідше зустрічається, та може вимагати дорогих серверів, що не завжди виправдано.

14. Проблеми, які спостерігають у різних хмарах та планувальниках:

- вивільнення RAM, якщо вона вже була зайнята процесом, але не використовується;
- можливість, що задача недоотримає ресурс, навіть заздалегідь вказаний в описі, бо інші мікропослуги могли його вже використати;
- можливість, що контейнеру доведеться працювати на обмежених потужностях при рості його потреб, коли вичерпався ліміт ресурсів, що він запитував у сервера, якщо не відбувається вилучення контейнера з вузла;
- вилучення задачі з вузла, якщо їй треба більше місця, що може призвести до затримок в обслуговуванні;
- ймовірність, що вилучені задачі не будуть розміщені на вузлах кластера через брак місця.

15. Способом розв'язання деяких з цих проблем може бути зміна алгоритму планувальника та підхід до аналізу запитаного задачею ресурсу:

- до проблеми з повільним поверненням задачею оперативної пам'яті у спільний вжиток після зменшення в ній потреби можна підійти зі сторони формування задач в групі таким чином, що сумарно вони не будуть потребувати більше за надану кількість ресурсу;
- для зменшення наслідків проблем, пов'язаних з обмеженням чи вилученням задачі з вузла через лімітування в її базовому запитаному та максимальному ресурсах можна зменшивши значення цих понять для кожної задачі, та замість цього накладаючи їх на групу задач, які сумарно не мають вимагати збільшення запитаних ОП, ЦП та мережевого ресурсу.

Елементи розділу опубліковані в наукових публікаціях [\[4\]](#), [\[5\]](#), [\[6\]](#).

РОЗДІЛ 2

МАТЕМАТИЧНА МОДЕЛЬ РОЗПОДІЛУ НАВАНТАЖЕННЯ В ХМАРНОМУ ПЛАНУВАЛЬНИКУ РЕСУРСІВ

Даний розділ присвячений теоретичному підґрунтю роботи. Тут описується ідея пошуку доповнювальних навантажень користуючись теорією ґраток та нечіткою логікою. З використанням груп доповненого навантаження будується математична модель, вводиться функція оптимізації та зазначаються обмеження, які слід враховувати.

2.1. Теорія нечітких ґраток у застосуванні до навантажень

У цьому підрозділі наведена інформація про теорію ґраток та нечітких ґраток, а також багатозначну логіку Поста, проаналізовано поняття доповнення у цих теоріях. Пропонується розглянути проблему оптимізації розподілення навантажень через пошук доповнювальних навантажень користуючись цією теорією, адже узгоджене або ж компактно згруповане навантаження сприятиме ефективнішому використанню засобів апаратного та логічного забезпечення.

2.1.1. Базові поняття теорії ґраток

Ґратка — це частково впорядкована множина L , в якій будь-які два елементи x і y мають точну нижню грань, що називають перетином (позначається $x \wedge y$), і точну верхню грань, що називають об'єднанням (позначається $x \vee y$); між елементами визначений частковий порядок [60]. Ґратки зображаються графами.

Вершини ґратки — це елементи, що є граничними значеннями в контексті часткового порядку. **Висхідна точка** — елемент, який пов'язаний із верхньою межею (супремумом) певної множини. **Стік** — елемент, що відповідає найменшій нижній межі (інфімуму) для певної підмножини.

Теорію ґраток можна застосувати до часових рядів, адже часові ряди є ланцюгами (Твердження 2.2.1 [60]): Будь-який ланцюг є ґраткою, в якій

перетин $x \wedge y$ співпадає з меншим, а об'єднання $x \vee y$ – з більшим з елементів x, y . Твердження очевидне, оскільки в будь-якому ланцюзі або

$$\begin{aligned} & x \leq y, \text{ або } y \leq x, \text{ тому} \\ & \text{або } x \wedge y = x, \text{ або } x \wedge y = y. \end{aligned} \quad (2.1)$$

Двоїсто для об'єднання:

$$\begin{aligned} & \text{або } x \vee y = x, \text{ або } x \vee y = y. \end{aligned} \quad (2.2)$$

Розглянемо визначення доповнювальних (комплементарних) елементів з теорії ґраток. Ґратка b є доповненням до ґратки a — це обмежена ґратка, яка задовольняє наступним умовам

$$a \vee b = 1 \text{ та } a \wedge b = 0 \quad (2.3)$$

З лінгвістичного погляду, початкова формула може бути інтерпретована як відсутність перекриття використання ресурсів, тобто одночасне використання одного ресурсу обома мікросервісами не відбувається. Водночас обидві мікропослуги спільно використовують всі доступні ресурси під час роботи. Комплементарна ґратка не обов'язково повинна бути унікальною. Обмежена ґратка означає, що мінімальний елемент дорівнює 0, а максимальний — 1 [61] Теорія ґраток містить визначення відносно доповненої ґратки та ортодоповнення, які також можуть бути корисними для демонстрації паралелізму з ідеєю організації мікросервісів у групі.

Розширимо цю ідею на мікропослуги, що працюють на одній групі ресурсів, або, спрощено, на одному сервері. Це означає, що мікропослуги є взаємодоповнювальними, коли, працюючи разом, вони повністю використовують потужність сервера, а якщо вони вимкнені, то сервер простоює.

Для узагальнення, беручи до уваги, що може бути важко знайти ідеальний взаємодоповнювальний тип мікросервісу, припустимо, що b можна рекурсивно замінити набором з двох інших мікросервісів, c і d , які слідує тому ж правилу, що і в (2.3), розподіляючи загальне навантаження в

пропорціях, які ніколи не перевищують 1, і які не будуть повністю взаємодіяти з a :

$$a \vee (c \vee d) = 1, c \wedge d = 0, a \wedge d = 0, a \wedge c = 0$$

(2.4)

Цей тип рекурсії може відбуватися багато разів, тобто немає обмежень на те, скільки мікросервісів може містити група ресурсів. Основним принципом є досягнення ефективного використання групи [7].

2.1.2. Моделювання задач на хмарі через графи

Початково задача виглядає як зважений граф часового ряду, де вагами виступають використання всіх ресурсів, актуальні для даної роботи. Для ілюстрації процесу, на Рисунку 2.1 наведено такий граф з тільки одним ресурсним навантаженням. Початково задача надходить в формі графа-дерева A , де через μ позначено навантаження певного ресурсу, наприклад процесорної потужності. Розхід змінюється з часом, деякий розхід може перевищувати можливості одного сервера. Він позначений числами більшими за 1. В такому випадку, слід запустити задачу в декількох екземплярах. Такі процеси як нормування — виділення цілого навантаження та розділення, тобто запуск ймовірно що на різних серверах, групах, а може й кластерах, показане у деревах B та B' .

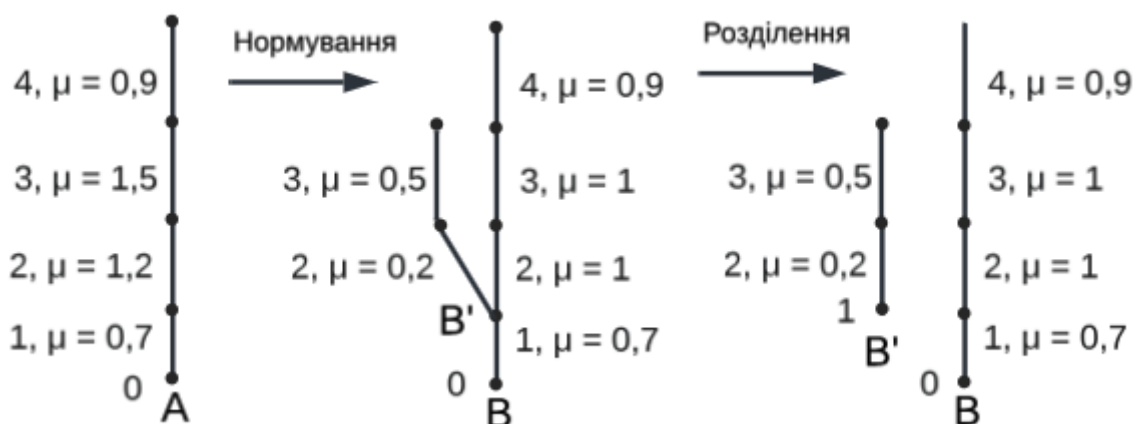


Рисунок 2.1 А — Початкове завантаження, представлене у вигляді графа-дерева, В та В' — дерево та листок, що відповідають дереву А після нормування та розділення навантаження

Процес обслуговування задач у хмарній системі, можна змоделювати як ліси. **Ліс** — це набір дерев, а **дерево** — це зв'язний граф без циклів. Граф часового ряду є частинним випадком дерева. За бажання, можна зобразити процес відгалуження процесів, спрямованих на підвищення відмовостійкості задачі як дерево з **листяками** (термінальні вершини без дуг, що входять, або без циклів).

Приклад п'яти таких задач показано на Рисунку 2.2. Перше дерево Д1 описує задачу, якій треба три паралельні запуски в період часу 2–4, два запуски в період часу 1–2, та 1 на початку. Дерево Д4 — процес, який повністю може обслужити потреби користувачів будучи запущеним в одному екземплярі. Д5 показує, що не завжди додатковий процес буде запущено до кінця життєвого циклу. Він може бути обірваний за відсутності збільшеної кількості споживачів роботи процесу. На мові Kubernetes, те саме буде звучати як под (завдання), що стосується одного мікросервісу, але запущене в декількох екземплярах в заданий час.



Рисунок 2.2 — Ліс дерев, що моделює групу обчислювальних задач, які розпаралелюються в певний момент, тим самим формуючи дерево з листками

2.1.3. Доповнювальні ґратки, або ґратки з доповненнями

Доповнювальна (комплементарна) ґратка — це частково впорядкована множина, яка є ґраткою та має властивість, що кожен її елемент має доповнення. Формально це записується наступним чином:

1. Для кожного елемента a існує елемент b , що доповнює a , такий що:
 - $a \wedge b = 0$, тобто інфімум дорівнює мінімальному елементу ґратки;
 - $a \vee b = 1$ – супремум дорівнює максимальному елементу ґратки.
2. Ґратка є дистрибутивною, тобто операції \wedge (перетин) та \vee (об'єднання) задовольняють закони дистрибутивності.

Дані визначення та приклади наводяться в Розділі «2.2.2 Властивості ґраток» [60]. Прикладом комплементарної множини, яку можна описати ґраткою є Булева алгебра, адже Булева алгебра оперує множиною значень $\{0, 1\}$ або будь-якими підмножинами цієї універсальної множини U . Операції, що можливі над множиною: \wedge , \vee , \neg (доповнення). Для множини $A \subseteq U$, доповненням є $\neg A = U \setminus A$.

Прикладом пошуку доповнень у множині підмножин $U = \{1, 2, 3\}$ будуть наступні твердження:
якщо $A = \{1, 2\}$, доповнення $\neg A = \{3\}$, оскільки $A \wedge \neg A = \emptyset$, $A \vee \neg A = \{1, 2, 3\}$.

2.1.4. Впровадження нечіткості для оцінки вжитку серверного ресурсу

Коли мова йде про навантаження на ресурси, включення нечіткості може бути більш доречним. З цією метою пропонується модифікація поняття кількості використаного серверного ресурсу мікропослугами в сторону нечіткості, або ж середнього показника за певний термін, щоб оперувати поняттям часткового завантаження ресурсів з варіацією приналежності ступеня використання ресурсу до нечіткої групи. При суміщенні навантажень, загальне навантаження в будь-який момент часу не має перевищувати повного можливого навантаження.

Нижче наведений опис характеристик:

С: певна характеристика процесора, наприклад, завантаження процесора,

пропускна здатність каналу, кількість користувачів або використання пам'яті;
 T_j : певний часовий інтервал, наприклад, година або «ранок», протягом якого вимірюється C . Цикли повторюються кожні 24 години. Певні відмінності можуть бути зроблені для святкових та вихідних днів;

j — одиниця часу. Поділятимемо часовий ряд використання ресурсу на розумні одиниці. Наприклад, оперуватимемо погодинними вимірюваннями протягом дня або введемо визначення «ранку», «дня», «вечора» і «ночі».

i : індекс мікропроцесора, який обчислюється від 1 до деякого натурального числа N .

$\mu_T(C_i)$: ступінь певної характеристики C мікропроцесора i протягом часу T .

Якщо мікропослуги 1 і 2 є взаємодоповнювальними, то $\mu_T(C_1)$ та $\mu_T(C_2)$ — це ступені присутності заданої характеристики C протягом обраного часу T . Разом вони повинні сформувати завантаження ресурсу сервера, що наближається до повного можливого. Виходячи з наведених вище визначень, можна переписати формулу нечіткого доповнення, представлену в [62] наступним чином:

$$\mu_{Tj}(C_2) = 1 - \mu_{Tj}(C_1) \quad (2.5)$$

Кожен мікросервіс має звичайне, середнє, мінімальне та максимальне навантаження. Посилаючись на (2.5), це означає, що, не беручи до уваги точне навантаження одного мікропроцесора, слід приймати навантаження другого. Доповнення повинно базуватися на визначенні: максимальне пересічне навантаження в будь-який момент часу не повинно перевищувати повного навантаження. Говорячи словами формул, це означає:

$$\max(\mu_{Tj}(C_1), \mu_{Tj}(C_2)) < 1 \quad (2.6)$$

Досі мова йшла про дві взаємодоповнювальні мікропослуги. Аналогічно можна екстраполювати формули на більшу кількість елементів. Щоб обмежити N — максимальну кількість екземплярів мікросервісів, які можуть працювати на одній групі серверів, візьмемо до уваги кількість потоків K на груповому процесорі (процесорах). Тоді $i \in [1, K]$. Узагальнюючи, формули (2.5) та (2.6) матимуть наступний вигляд [7]:

$$\mu(C_N) = 1 - \mu_{T_j}(C_1) - \mu_{T_j}(C_2) - \dots - \mu_{T_j}(C_{N-1}) \quad (2.7)$$

$$\max(\mu_{T_j}(C_1), \mu_{T_j}(C_2), \dots, \mu_{T_j}(C_{N-1}), \mu_{T_j}(C_N)) < 1 \quad (2.8)$$

2.1.5. Доповнювальні ґратки у нечіткій логіці

У нечіткій логіці (fuzzy logic) теорія доповнювальних ґраток також має аналоги, хоча вони дещо відрізняються від класичних, адже замість конкретних значень операції йдуть над інтервальними значеннями, котрі детерміновані, але чітко не визначені. Характеристики нечітких ґраток наступні:

- елементи представляють ступені належності в інтервалі $[0, 1]$;
- операції \wedge і \vee часто визначаються через мінімум і максимум:
 $a \wedge b = \min(a, b), \quad a \vee b = \max(a, b).$

Доповнення в нечіткій логіці визначається наступними виразами:

- Доповненням елемента $a \in [0, 1]$ є $1 - a$.
- Умови для доповнення:
 - $a \wedge (1 - a) = 0,$
 - $a \vee (1 - a) = 1.$

(2.9)

Описати пошук доповнення може наступний приклад з нечіткою множиною, що виражає значення належності $A = \{0.3, 0.7, 0.5\}$, доповненням буде $\neg A = \{0.7, 0.3, 0.5\}$, обчислене як $1 - A$.

У нечіткій логіці ґратки стають більш гнучкими через використання різних функцій для \wedge, \vee , і \neg . Це дозволяє моделювати реальні системи, де класичні закони не завжди виконуються. Таблиця 2.1 групує основні характеристики класичних та нечітких ґраток, показуючи основні відмінності та принцип пошуку доповнення.

Таблиця 2.1 Порівняння класичних і нечітких ґраток

Характеристик а	Класичні ґратки	Нечіткі ґратки
Елементи	Чіткі множини або значення	Ступені належності в межах $[0, 1]$

Доповнення	Єдине і чітко визначене. Визначається виразами: $a \wedge b = 0$, $a \vee b = 1$	Може змінюватися залежно від функцій. Визначається виразами: $a \wedge (1 - a) = 0$, $a \vee (1 - a) = 1$.
Операції \wedge, \vee	Фіксовані мінімум/максимум або аналогічні $a \wedge b = 0, \quad a \vee b = 1$	Гнучкі мінімум, максимум, логічні функції $a \wedge b = \min(a, b), \quad a \vee b = \max(a, b)$
Приклад	Для множини $U = \{1, 2, 3\}$ якщо $A = \{1, 2\}$, то доповнення $\neg A = \{3\}$, оскільки $A \wedge \neg A = \emptyset, A \vee \neg A = \{1, 2, 3\}$	Якщо $A = \{0.3, 0.7, 0.5\}$, доповненням буде $\neg A = \{0.7, 0.3, 0.5\}$, обчислене як $1 - A$.

2.1.6. Обґрунтування ідеї

Щоб проілюструвати ідею створення збалансованих ресурсних груп, розглянемо простий приклад. Припустимо, що є дві компанії з приватними хостингами: квиткова компанія, яка відчуває високе навантаження вдень, і онлайн-казино, основна діяльність якого відбувається вночі. Щоб оптимізувати кількість апаратних ресурсів, що витрачаються на хостинг цих двох підприємств, можна розглянути можливість створення спільного хостингу. Одні й ті ж сервери повторно використовувалися б додатком, який має обробляти більше навантаження в певний момент. Загальне навантаження залишається незмінним, але використовується менше ресурсів для роботи обох програм. Перехід у використанні ресурсів конкретною програмою буде відносно поступовим.

Потенційні сплески навантаження для кожного додатку в нестандартний час можуть відбуватися через розпродажі та спеціальні пропозиції, а також непередбачувані раптові події, такі як складні погодні умови та катастрофи. Якщо застосунки надають послуги в одному регіоні, очевидно, що в другому

випадку люди з більшою ймовірністю віддадуть перевагу купівлі квитків і порятунку свого життя, а не грі в казино, як вдень, так і вночі.

З іншого боку, перший випадок є складнішим і вимагає або локального, або комплексного вирішення. Потенційне локальне вирішення передбачає переговори між компаніями щодо рознесених у часі спеціальних пропозицій. Така можливість є малоімовірною, оскільки вимагатиме, щоб компаніями керували друзі або одна й та сама особа. Цей підхід також передбачає врахування ризику того, що одна або обидві компанії можуть втратити свій дохід.

Загальне рішення передбачає надання додаткових ресурсів, які рідко використовуються, але які мають вирішальне значення для безпечних і стабільних умов праці. Такі ресурси також стануть у пригоді, коли компоненти обладнання вийдуть з ладу. Можна очікувати більшої довговічності обладнання, коли воно працює нижче максимальної потужності. Додаткові ресурси також допоможуть забезпечити довший термін служби серверів.

Збільшення розподілу ресурсів і одночасне збільшення навантаження на обидва застосунки може викликати занепокоєння щодо рентабельності спільного використання АЗ. Саме тут концепція поділу додатку на мікросервіси може стати оптимізатором ресурсів. Кожен мікросервіс має різні завдання і різне загальне навантаження в один і той самий період часу порівняно з іншими мікросервісами того самого додатка. В інтернет-магазинах люди витрачають значно більше часу на вибір товару, ніж на процедуру замовлення та оплати. Як наслідок, можна очікувати набагато менше дій, пов'язаних із входом на сайт, ніж з фільтруванням товарів. Отже, поєднання низьконавантажених мікросервісів одного додатка з високонавантаженими мікросервісами іншого додатка є важливим для досягнення збалансованої групи.

В результаті, при високій активності в обох додатках в один і той же момент, збалансована група мікросервісів не потребуватиме значної кількості ресурсів. Як наслідок, загальний запас міцності для спільного простору між декількома мікросервісними додатками зменшиться порівняно з

використанням приватних просторів. Водночас загальна відмовостійкість буде вищою [5], [7].

2.1.7. Порівняння прикладу з теорією нечітких ґраток

В контексті теорії ґратки, слід визначити додатковий мікросервіс, який досягає того ж максимального стану використання процесора, гарантуючи, що тільки один мікросервіс буде активним в будь-який момент часу.

Позначимо через a мікросервіс квиткової компанії, який працює вдень, а через b — мікросервіс казино, який працює вночі. Позначимо також через 1 потужність сервера/кластера/ресурсної групи, яка забезпечує роботу обох мікросервісів і водночас є потенційною максимальною безпечною потужністю одного з мікросервісів. Аналогічно, нехай 0 позначає стан, коли все простоє або якого не повинно бути.

Виходячи з (2.3) та її нечіткого аналога (2.5), можна сказати, що метою є досягти стану повного завантаження, рівному цілому, або 1, коли працює *або* a , або b . Стан часткового завантаження також прийнятний: якщо на 80% завантажено a , то b має бути завантажено не більше 20%, і аналогічно для (2.7) і (2.8). Стани, коли один завантажений на 60%, а інший на 10%, також є прийнятними, оскільки вони відповідають формулам для нечітких множин. У цьому випадку сервер може виконувати додаткові обчислювальні задачі в цей час.

Максимальний обсяг ресурсів розрахований лише на підтримку часткового навантаження. У випадку, коли обидва мікросервіси потребують більше ресурсів, слід активувати реплікацію і не допускати стану 0.

Беручи до уваги, що шляхів, які доповнюють один одного, може бути кілька, варто зазначити, що мікросервіс, який би міг підійти як пара до першого, не є унікальним. Не тільки квиткові компанії працюють вночі. Банківський мікросервіс міг би так само підійти до мікросервісу казино в цьому плані [7].

2.1.8. Моделювання доповнювальних часових рядів задач у хмарній інфраструктурі

Як було сказано у підпункті [Моделювання задач на хмарі через графи](#), задачі, які є на хмарі, можна представити як часові ряди. Уточненням цього підпункту буде додавання зваженості у часові ряди. Вагами або значеннями ребер буде вжиток певного ресурсу, наприклад, ОП(RAM), процесорна частота або каналний ресурс. Не звужуючи загальної картини, для спрощення будемо розглядати добовий період кожної задачі. В реальному житті, пропонується виділити окремі шаблони роботи кожної задачі по буднях, вихідними та святковими днями. У випадку часових рядів вигляду шаблону дерев з листям (див. Рисунок 2.2), листя слід розглядати як окремі часові ряди, які зазвичай беруть початок не з нульової точки, тобто в кожному разі оперуємо тільки стовбуром дерева.

Тоді основні процеси двох задач А та В можна зобразити часовими рядами з вагами, що відповідають нормованій кількості вживаного ресурсу (нехай, оперативної пам'яті), див. Рисунок 2.3. При нормуванні, максимальне значення використаного ресурсу взятє за 1. Ваги ребер — це нечіткі величини. Першою причиною нечіткості є неоднакове, а точніше, усереднене значення використання цього ресурсу. Другою причиною нечіткості є усереднення цього значення за всі обрані часові проміжки спостереження за поведінкою певної задачі, тобто це можуть бути всі робочі понеділки за три останні місяці.

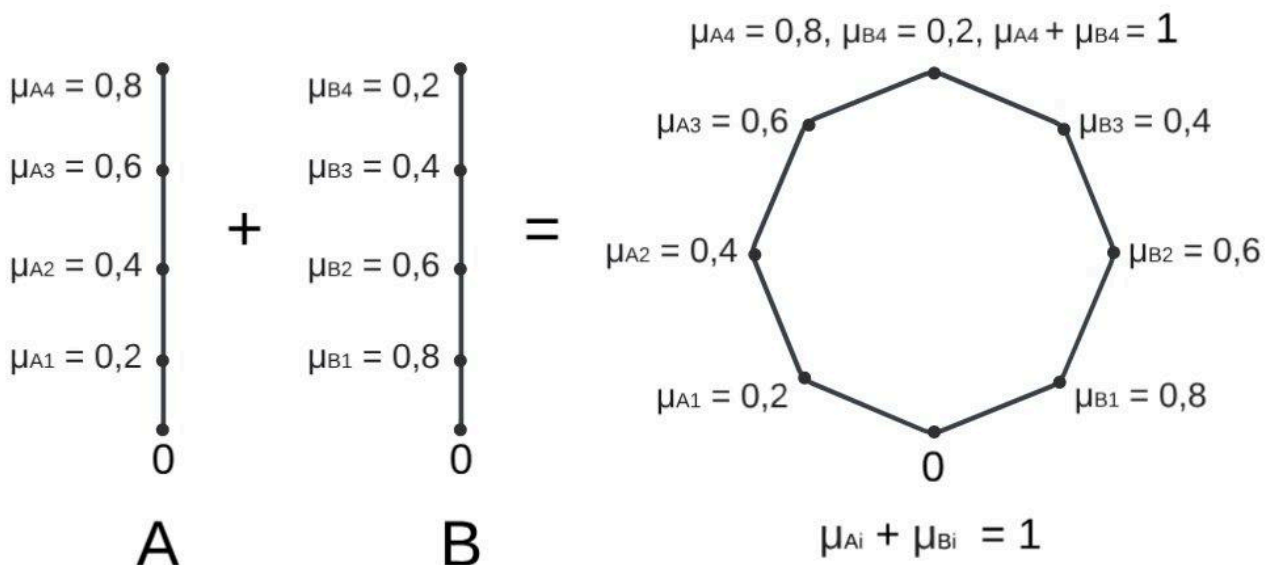


Рисунок 2.3 — Доповнення (справа) часових рядів А та В в теорії нечітких ґраток

Зважаючи на значення недовикористаного ресурсу у задачі А у відповідні моменти часу, комбінація цієї задачі з задачею В, забезпечить повне використання ресурсу одного вузла системи. Це показано в правій частині Рисунка 2.3. Таким чином, **ідеєю цієї роботи** є пошук доповнювальних часових рядів на добових проміжках з метою забезпечення повного використання ресурсу.

2.1.9. Багатозначна логіка Поста

Логіка Поста (або k-значна логіка Поста) — це багатозначна логіка, в якій істинність висловлювань може набувати значень зі скінченної множини k рівнів. Наприклад, якщо $k=3$, то можливі значення можуть бути:

- 0 (хибність),
- 1 (нейтральне/невизначене значення),
- 2 (істинність).

Головна особливість логіки Поста — її дискретна природа та те, що вона підпорядковується лінійному впорядкуванню значень. Перехід між значеннями відбувається за чітко визначеними правилами.

Формально, логіка Поста використовує алгебру, де функції можуть бути монотонними (тобто зростаючими) і будуються за принципом:

$$f(a_1, \dots, a_n) = \max(\min(a_i + c_i, k - 1)) \quad (2.10)$$

де c_i — деякі константи.

Логіка Поста є узагальненням двозначної логіки, зберігаючи при цьому класичні логічні операції та додаючи проміжні рівні істинності [60].

Багатозначну логіку слід розглянути як альтернативу застосуванню нечіткої логіки для моделювання часових рядів розподілу ресурсу. Для цього важливими є поняття заперечення та доповнення.

Заперечення (доповнення до «істини») в логіці Поста визначається за такою формулою:

$$\neg x = (k-1) - x, \text{ де:}$$

x — значення істинності, яке належить множині $\{0, 1, \dots, k-1\}$

k — кількість рівнів у логіці [63].

Приклад для тризначної логіки ($k=3$)

Якщо істинність виразу:

$x=2$ (істина), то $\neg x=0$ (хибність).

$x=1$ (проміжне значення), то $\neg x=1$ (не змінюється).

$x=0$ (хибність), то $\neg x=2$ (істина).

Тобто, заперечення є симетричним відносно середнього значення.

В класичній логіці доповнення визначається як «усе, що не входить у множину». У логіці Поста поняття доповнення визначається через її лінійне впорядкування, що означає можливість задавати доповнення через проміжні значення.

Одним із можливих підходів є введення монотонного доповнення:

$$C(x) = (k-1) - x \quad (2.11)$$

Ця формула збігається з негачією в логіці Поста. Вона гарантує, що найбільше значення переходить у найменше, а середні значення можуть мати свої особливості.

Інший варіант (асиметричне доповнення):

$$C'(x) = \frac{k-1}{x+1} - 1, \quad (2.12)$$

який використовується в окремих варіаціях багатозначної логіки.

Отже поняття монотонного доповнення, яке збігається з запереченням підходить для опису повноти використання ресурсу, яке б дозволило комбінувати часові ряди споживання ресурсів. Все ж, в роботі обрана нечітка логіка через забезпечення більшої гнучкості, відсутність потреби в монотонності функції, а також відсутність чітких меж для визначення групи, див. Табл. 2.1.

Табл. 2.1 — Відмінність логіки Поста від нечіткої логіки

Характеристика	Логіка Поста	Нечітка логіка
----------------	--------------	----------------

Кількість значень істинності	Скінченне k (дискретні рівні)	Нескінченна множина (континуальна шкала $[0,1]$)
Природа значень	Чітко визначені проміжні рівні	Градiєнтна невизначеність між істинністю та хибністю
Операції	Лінійне впорядкування значень, дискретні логічні операції	Нечіткі множини, операції мінімуму, максимуму, доповнення
Використання	Моделювання логічних схем, узагальнення булевої алгебри	Розмиті поняття, керування, експертні системи
Приклад логічного висловлювання	«Дзвінкість звуку може бути слабка, середня або сильна» (чіткі рівні)	«Дзвінкість може бути 0.73 із 1» (нечітке значення)

2.1.10. Обґрунтування вибору теорії нечітких ґраток для опису задач в хмарній інфраструктурі

Комплементарні ґратки в класичному вигляді мають чіткі правила пошуку доповнень, які можна застосувати до булевої алгебри, логіки та теорії множин. У нечіткій логіці ця теорія адаптована для роботи зі ступенями належності, що робить її зручною для моделювання невизначеностей у реальних системах.

В роботі саме взята за основу теорія нечітких ґраток для опису характеру роботи задач у кластерній інфраструктурі аби позбавитись необхідності поділу певних величин на групи.

Комплементарні ґратки в класичному вигляді мають чіткі правила пошуку доповнень, які можна застосувати до булевої алгебри, логіки та теорії множин. У нечіткій логіці ця теорія адаптована для роботи зі ступенями належності, що робить її зручною для моделювання невизначеностей у реальних системах.

2.2. Математична модель формування груп доповнювальних навантажень

В цьому розділі наведена математична модель процесу пошуку доповнювальних навантажень, заснована на *принципі доповнюваності (комплементарності), тобто формування доповнювальних груп подів, які разом забезпечують ефективне використання ресурсів (сумарне навантаження = 100% або 1).*

1. Головний критерій оптимізації — це не RAM як така, а інтегральне використання ресурсів (RAM, CPU, мережеві ресурси тощо), з акцентом на RAM через складність її звільнення.
2. Часовий фактор в обмеженнях використання ресурсів.

2.2.1. Параметри математичної моделі

Матмодель оперує наступними **множинами**:

- $P = \{p_1, p_2, \dots, p_n\}$: множина подів, (2.13)
- $N = \{n_1, n_2, \dots, n_m\}$: множина вузлів,
- $T = \{t_1, t_2, \dots, t_h\}$: часові періоди.

Параметри матмоделі:

$$\text{Demand}(p_i, t_k) = \alpha \cdot \text{RAM}(p_i) + \beta \cdot \text{CPU}(p_i) + \gamma \cdot \text{NET}(p_i) + \delta \cdot \text{DISK}(p_i), \quad (2.14)$$

це загальне навантаження поду p_i в час t_k , де $\alpha, \beta, \gamma, \delta$ — вагові коефіцієнти.

$$\text{Supply}(n_j, t_k) = \alpha \cdot \text{RAM}(n_j, t_k) + \beta \cdot \text{CPU}(n_j, t_k) + \gamma \cdot \text{NET}(n_j, t_k) + \delta \cdot \text{DISK}(n_j, t_k), \quad (2.15)$$

це доступні ресурси вузла n_j в час t_k .

Групи нечіткості:

$$G_k = \{P_1, P_2, \dots, P_g\} \quad (2.16)$$

це множина груп, що формують доповнювальні навантаження для періоду t_k . Усі групи повинні відповідати **принципу комплементарності**:

$$\sum_{P_g \in G_k} \text{Demand}(P_g, t_k) = \text{Supply}(N, t_k) \quad (2.17)$$

Змінні:

$x_{ij} \in \{0, 1\}$: 1, якщо под p_i розміщено на вузлі n_j , інакше 0.

$y_{ik} \in [0, 1]$: частка ресурсу для поділеного поду p_i у вузлі n_j , активна, якщо $\text{canChunk}(p_i) = 1$.

2.2.2. Цільова функція

Цільова функція ефективності використання ресурсів $F(P, N, T)$:

$$F(P, N, T) = \sum_{n_j \in N} \sum_{t_k \in T} \left(\frac{\sum_{p_i \in P} x_{ij} \cdot \text{Demand}(p_i, t_k)}{\text{Supply}(n_j, t_k)} \right), \quad (2.18)$$

де $\text{Demand}(p_i, t_k)$ та $\text{Supply}(n_j, t_k)$ визначаються для інтегрального використання ресурсів.

Задачею оптимізації є максимізувати ефективність використання ресурсів $F(P, N, T)$, мінімізуючи їх простій. Останнє позначимо через функцію обмеження (constraints) $C(N, T)$. Тоді можна сформулювати наступну систему:

$$\begin{cases} F(P, N, T) \rightarrow \max \\ C(N, T) \rightarrow \min \end{cases} \quad (2.19)$$

2.2.3. Обмеження матмоделі

1) Ресурси вузлів (із часовим фактором):

Для кожного вузла n_j у кожен момент часу t_k :

$$\sum_{p_i \in P} x_{ij} \cdot \text{Demand}(p_i, t_k) + \sum_{p_i \in P} y_{ij} \cdot \text{Demand}(p_i, t_k) \leq \text{Supply}(n_j, t_k) \quad (2.20)$$

2) Принцип комплементарності:

Для кожного періоду t_k :

$$\sum_{P_g \in G_k} \text{Demand}(P_g, t_k) = \sum_{n_j \in N} \text{Supply}(n_j, t_k) \quad (2.21)$$

Групи G_k формуються так, щоб сума ресурсів у час t_k дорівнювала доступній пропускній здатності вузлів.

3) Афінність/антиафінність ((не) прив'язаність задач):

Дані обмеження повторюють чинні обмеження поточної матмоделі.

- $\text{Affinity}(p_i, n_j) = 0 \implies x_{ij} = 0$. (2.22)
- $\text{AntiAffinity}(p_i, n_j) = 1 \implies x_{ij} = 0$

4) Taints і tolerations (виключення та толерації щодо розміщення на вузлі):

$$\text{Якщо } \text{Taint}(n_j) \notin \text{Tolerance}(p_i), \text{ то } x_{ij} = 0. \quad (2.23)$$

Додаткові обмеження, присутні виключно в цій матмоделі:

5) Подільність подів:

$$\text{Якщо } \text{canChunk}(p_i) = 0, \text{ то } y_{ij} = 0. \quad (2.24)$$

6) Групи нечіткості:

Поди, що належать до однієї групи нечіткості G_g , повинні разом утворювати повне навантаження (доповнювати інші групи):

$$\sum_{p_i \in P_g} x_{ij} \cdot \mu_{\text{group}}(p_i) \leq \text{Supply}(n_j, t_k) \quad (2.25)$$

ВИСНОВКИ

1. Розглянуто основні поняття теорії ґраток. Для опису використання ресурсів задачами у розподіленій системі хостингу пропонується накласти часові ряди на теорію ґраток. Лінійна впорядкованість часового ряду дозволяє взяти за основу зважений за принципом нечіткої логіки граф-дерево для представлення навантаження на сервер в різний час, спричинене кожною обчислювальною задачею.
2. Проаналізовано принципи нечіткої логіки та логіки Поста, зокрема поняття доповнень. Логіка Поста зручна для моделей, що мають чіткі проміжні рівні, а нечітка логіка — для роботи з нечіткими, розмитими поняттями, під які підходять не завжди стабільні зведені навантаження на ресурс.
3. Доведено несуперечність принципам та придатність використання як нечіткої логіки, так і логіки Поста для роботи з навантаженнями на сервери та пошуку повного доповненого навантаження.
4. Обґрунтовано вибір нечіткої теорії ґраток для моделювання пошуку доповнених навантажень, що формуватимуть повне використання ресурсу в довготривалій перспективі. Вибір пав на нечітку логіку порівняно з багатозначною логікою Поста оскільки остання оперує дискретним набором значень, тоді як нечітка логіка дозволяє будь-які значення з інтервалу $[0,1]$.
5. Запропоновано нову математичну модель розподілу навантажень, яка оперує групами процесів з постійними навантаженнями, базується на теорії нечітких ґраток, та дозволяє планувати використання фізичних ресурсів серверів.

6. Математична модель містить часовий фактор для врахування всіх змін протягом заданого періоду часу та пропонує розв'язувати задачу максимізації ефективності використання ресурсів, мінімізуючи їх простій.
7. Ключовим параметром матмоделі визначено інтегральне значення ресурсу Demand по RAM, адже процес вивільнення оперативної пам'яті набагато складніший, ніж CPU або мережевого ресурсу, а також зазвичай вимагає додаткових налаштувань на стороні програмного забезпечення. Канальні ресурси, процесорне завантаження, а також дисковий простір також беруться в обрахунок.
8. Обмеження матмоделі враховують вимоги до безпеки та розміщення обчислювальних задач окремо на вузлах (taints), толерантність до заборони розміщення — tolerations, прив'язаність до певних типів вузлів або задач або подів (affinity), або ж відхилення запусків разом з певними задачами (antiaffinity). Також береться до уваги подільність програмного застосунку canChunk.

Елементи розділу опубліковані в науковій публікації [7].

РОЗДІЛ 3

КОМПЛЕКСНИЙ МЕТОД ДОПОВНЮВАЛЬНОГО РОЗПОДІЛЕННЯ ОБЧИСЛЮВАЛЬНОГО НАВАНТАЖЕННЯ В БАГАТОСЕРВЕРНІЙ СИСТЕМІ

Матмодель та приклади, наведені для опису системи в минулому розділі пояснюють механізм роботи системи, однак наявність стандартного набору правил або міркувань для групування мікросервісів дає структурне розуміння випадків застосування принципу доповнювальних навантажень, пояснюючи місця оптимізації.

Метою цього розділу є продемонструвати можливість повного ефективного завантаження серверів, що сприятиме зменшенню їх кількості, а також збалансованій завантаженості всіх компонентів системи.

Розділ складається з опису стратегій групування, які пояснюють причини існування доповнювальних навантажень, опису методу доповнювального розподілу навантажень та етапів групування мікропослуг з метою спільного однакового довготривалого використання ресурсу. Розділ також включає методику нормування отриманих характеристик вимірів на локальних серверах на сервери хмарної інфраструктури з іншими технічними характеристиками для забезпечення коректної оцінки вхідних даних. В завершальному пункті розділу детально описано алгоритм пошуку доповнювальних навантажень та загального адміністрування черги задач.

3.1. Стратегії групування

Групування передбачає, що фізичні сервери розташовані якомога ближче і з'єднані в локальну групу за допомогою дротів, або це один суперкомп'ютер з великою кількістю ресурсів або ж підмодулів, які можуть підключатись в разі потреби. Нижче перераховані критерії, які можуть вплинути на прийняття рішення про групування задач.

3.1.1. Безпека та множинна оренда (multi-tenency) ресурсу

Мікросервіси можна згрупувати з міркувань безпеки або за правами

доступу користувачів. Це найбезпечніший спосіб групування. Однак проблеми, пов'язані з багатокористувацькою орендою, можна вирішити кількома способами, залежно від конкретних потреб і ризиків.

1. Віртуалізація дозволяє чітко розділити спільний простір між кількома запущеними програмами.
2. Контейнеризацію можна використовувати на додаток до вже запущеної операційної системи.
3. Можна запускати застосунки від різних користувачів в рамках однієї операційної системи, де кожен користувач має свій заздалегідь визначений простір.
4. У публічному хмарному середовищі можна розділити один і той самий фізичний сервер на безпечні логічні простори для кожного користувача або орендаря, тим самим забезпечуючи високий рівень безпеки.
5. Окрема оренда, яка вимагає окремих мікросервісів для окремих груп користувачів або користувачів, також може бути надана в рамках однієї групи.

Водночас надзвичайно суворі вимоги до безпеки зустрічаються відносно рідко. Щоб забезпечити належний рівень безпеки, не обов'язково запускати мікросервіси, які вимагають однакових прав доступу користувачів, в одній групі. Якщо це не є необхідним, з погляду ефективності використання ресурсів, буде ефективніше базувати поділ на альтернативних принципах.

У Kubernetes багатокористувацька оренда часто передбачає, що багато команд використовують один і той самий кластер. Також існує так звана оренда SaaS, коли одній команді користувачів надається кілька кластерів з різними додатками [64]

3.1.2. Спільні ресурси

Більшість додатків не є автономними, а мають окремі ресурси, такі як бази даних. В мікросервісній архітектурі зазвичай є спільний реєстр схем, класи DTO, домени, бібліотеки та протоколи, які зберігаються окремо, виключно для використання двома або більше мікросервісами. Групування

також може ґрунтуватися на використанні одного і того ж спільного ресурсу.

Якщо кілька мікросервісів часто взаємодіють один з одним, корисно мати дизайн системи, орієнтований на максимальне запобігання збоєм, особливо під час передачі даних. Неузгодженість даних, спричинена частково надісланими та обробленими даними, може бути складною для відкату. Усунення таких невідповідностей може вимагати дій від кожної мікрослужби, яка обробляла інформацію. Це описано в патерні SAGA. Альтернативним варіантом може бути створення спеціалізованих процедур очищення. Ці процедури будуть знати всі потенційні сценарії збоїв і зможуть виконувати ручні відкати, отримувати інформацію з файлів резервних копій або перераховувати оновлені значення і записувати попередні узгоджені значення.

Процедури відкату в мікросервісній архітектурі є ризикованими. Немає гарантії, що якась частина програми не прочитає тимчасово неправильні значення. Обидві стратегії займають багато часу і містять кілька операцій, замість того, щоб виконати все за один крок. Людський фактор також присутній, оскільки при оновленні коду розробник може забути внести відповідні зміни в процедуру відкату. Весь процес ускладнюється проблемами комунікації. Якщо проблема з комунікацією виникає на прямому шляху, вона може виникнути та в зворотному напрямку. Тому усунення проблеми є більш корисним, ніж її вирішення.

В мікросервісній архітектурі можливі такі проблеми: затримки, перебої в роботі однієї частини системи, неможливість встановлення з'єднання через оновлення обладнання, що підключається, або його налаштувань, обмеження черг та особливості їх роботи, а також інші фізичні або логічні проблеми. Завжди краще усунути якомога більше потенційних збоїв і підвищити відмовостійкість, якщо це не впливає на якість роботи і не збільшує мінуси програмного продукту. Групування за спільними ресурсами зменшить багато можливих проблем, пов'язаних з проблемами підключення. У випадку перебоїв у роботі ціла група вийде з ладу, що з більшою ймовірністю збереже інформацію в базі даних. Одночасно, недоліком такого групування є

неможливість врахувати навантаження мікросервісів та забезпечити ефективне використання ресурсів.

3.1.3. Пропускна здатність каналу

Бувають випадки, коли найбільш завантаженим ресурсом є ресурс каналу, коли мікросервіс повинен обробляти масивні та/або дуже часті фрагменти даних. Це може не призводити до високого навантаження на процесор, але вимагає багато потоків для обробки вхідної інформації. Якщо активність такого мікросервісу зростає, виникає потреба в масштабуванні.

Підтримка такого мікросервісу потребує значної кількості ресурсів. Найкращим варіантом було б об'єднати його з іншими мікросервісами, які б або приймали рідкісні запити й, можливо, мали високе навантаження на процесор, або функціонували аналогічно, але в інші години. Зрозуміло, що такі мікросервіси є ризикованими, і якщо можливо, бажано уникати інтенсивного трафіку між компонентами програми шляхом перепроєктування системи та призначення мікросервісів. Існують методи запобігання цьому. Введення додаткового етапу попередньої обробки даних або зберігання проміжних результатів у базі даних може бути корисним. Дані могли б проходити первинну фільтрацію і направлятися за кількома напрямками.

3.1.4. Час і стан завантаження процесора

Існують мікросервіси, розроблені спеціально для обробки даних, наприклад, програми машинного навчання, які вимагають багато обчислювальних ресурсів [65]. Такі завдання можуть займати багато часу і вимагати високого навантаження на процесор. Можливо, можна розділити мікросервіси з високим навантаженням на процесор на декілька, щоб зменшити кількість операцій, які вони мають виконувати. Відповідно, це зменшить час обробки одного запиту і, відповідно, збільшить пропускну здатність одного екземпляра такого мікросервісу. Однак у багатьох випадках дослідники та інженери вважають за краще мати всю логіку в одному місці і повторно використовувати її компоненти в різних місцях, ніж розпорошувати

її по декількох мікросервісах, таким чином дублюючи код або збільшуючи трафік між компонентами додатку [66].

Цей вид мікросервісу зазвичай є ядром додатку і отримує менше запитів порівняно з іншими компонентами. Це пов'язано з попередніми процедурами фільтрації помилкових даних та агрегуванням даних, які зменшують кількість початкових запитів. Для створення групи, що оптимізує ресурси, такі мікросервіси можуть бути об'єднані з іншими, які обробляють свої завдання швидко і не відчують великого трафіку одночасно з активною фазою першого набору. Доповнювальні мікропослуги не повинні зв'язувати потоки на тривалий час, дозволяючи їм швидко ставати доступними для нових завдань. Іншими словами, об'єднувати мікросервіси варто тоді, коли одному з них потрібно дві секунди на обробку одного запиту, тоді як його аналог обробляє 10 запитів за одну секунду.

3.1.5. Географічні міркування та міркування про час активності

Швидкість реакції користувача зменшується зі збільшенням відстані до оброблювального сервера. Тому глобально розподілені сервіси воліють запускати окремі екземпляри своїх додатків на регіональних серверах або хмарних центрах, щоб забезпечити найкращий користувацький досвід [67]. Наприклад, сервери Amazon наразі розташовані в 32 географічних регіонах [68]. Розглянемо, з іншого боку, компанію, яка присутня на одному континенті, наприклад, у Північній Америці, а також має представництво на іншому континенті, де робочий час суттєво не перетинається, наприклад, у Європі. Залежно від цілей компанії та найбільш ймовірних джерел її прибутку, може не бути вагомих причин для створення дорогого хостингу на декількох континентах. Люди, які користуються послугами в непопулярних місцях, погодяться чекати на відповідь від віддаленого сервера і залишаться лояльними до постачальника послуг, яким вони користуються.

Наприклад, розглянемо операторів мобільного зв'язку, які працюють у

певному окрузі. Коли їхні клієнти подорожують, послуги зв'язку надаються місцевими провайдерами, але мобільний додаток повинен працювати незалежно від поточного місцезнаходження користувача. Перебуваючи в туристичній поїздки в іншій країні та очікуючи на оновлення інформації в мобільному додатку, користувач може бути більш готовим отримати відповідь із затримкою, ніж зазвичай. Цей час відповіді не вплине на його вибір оператора мобільного зв'язку, коли він повернеться. Усвідомлення існування компаній і додатків, які працюють таким чином, дає підстави планувати оптимальніше використання спільного простору.

З погляду групування, це означає, що можна об'єднати мікропослуги для надання послуг не тільки в різний час для одного регіону, але й для різних географічних регіонів, чий часові пояси можна вважати протилежними через відсутність перекриття в часі активного використання. У наведеному вище прикладі було б зручно створити одну логічну і, можливо, фізичну групу серверів як набір екземплярів, на яких розміщуються однакові мікропослуги, кожен з яких надає послуги протилежним географічним регіонам з низькою інтенсивністю користувачів.

У Таблиці 3.1 представлені всі потенційно ефективні способи поєднання мікросервісів, що дозволяють їм формувати взаємодоповнювальні групи та розміщуватися в межах однієї серверної групи. Рисунок 3.1 зображає один з варіантів структури системи, де взаємодіють всі наведені види груп.

3.1.6. Моделювання способів групування задач

В таблиці зведені вищеописані причини групування мікросервісів. Частина з них є необхідністю, наприклад групування заради безпеки вище за інші вигоди. Групування заради спільного ресурсу може бути високого і середнього пріоритету, вигоду слід враховувати.

Таблиця 3.1 Можливості групування для оптимізації використання ресурсів

Назва групи	Первинна (проблемна) мікропослуга	Додаткова мікропослуга	Коли використовувати	Проблеми

1.Безпека	Мікросервіс з певними стандартами безпеки.	Інші мікропослуги з тими ж стандартами безпеки.	Керування доступом декількох користувачів до спільного ресурсу.	Може не враховувати навантаження на мікропроцесори, і група не буде оптимально використовувати ресурси.
2.Спільний ресурс	Мікросервіс, який використовує певний зовнішній ресурс, що запускається на тому ж підкластері.	Інші мікросервіси, які використовують той самий спільний ресурс.	Коли потрібно мати максимальну швидкість зв'язку між зовнішніми ресурсами та мікросервісом.	Може не дотримуватися балансу навантаження на групу мікросервісів і не забезпечувати ефективне використання ресурсів.
3.Велика пропускна здатність каналу	Через мікросервіс повинні проходити масивні та/або дуже часті фрагменти даних.	Мікросервіси, які б або приймали рідкісні запити і, можливо, мали високе навантаження на процесор, або працювали за схожим принципом, але в інший час доби.	Коли для обробки вхідної інформації потрібно багато потоків.	Мікросервіси з великим трафіком є ризикованими через можливу нестачу місця в чергах, частіші випадки недоотримання інформації та затримки в обробці.
4.Високе навантаження на процесор/ ОП	Завдання, які виконуються протягом тривалого часу та/або сильно завантажують процесор чи ОП.	Доповнювальні мікросервіси не повинні довго утримувати потоки, тому вони будуть швидко готові до нових завдань.	Коли не потрібно багато потоків, але потрібна висока продуктивність процесора чи ОП.	Час повертання відповіді може бути проблемою в додатках, що працюють в режимі реального часу.
5.Географічно відокремлені	Застосунки, які обробляють велику кількість одночасних з'єднань, що призводить до високого навантаження на пропускну здатність каналу та/або навантаження на процесор.	Мікросервіс, який має високе навантаження подібного типу, але години роботи якого не перетинаються.	Коли час великого навантаження стабільний і обмежений певною частиною дня.	Проблеми можуть виникнути, якщо взаємодоповнювальні мікросервіси мають раптові несподівані піки активності в часі, що перетинаються.

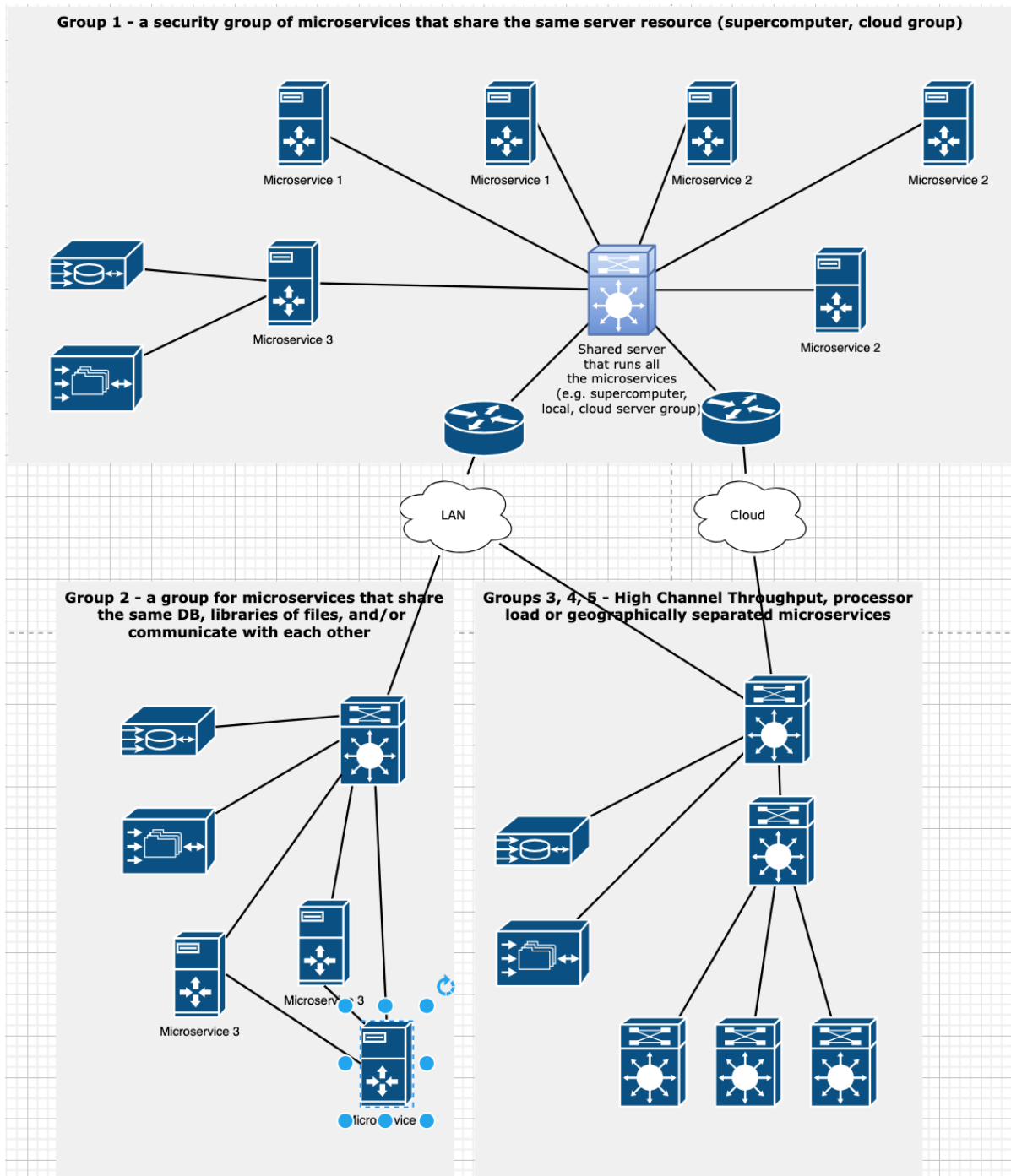


Рисунок 3.1 — Можливі стратегії організації серверів у групи

3.2. Загальний метод доповнювального розподілу навантаження

В роботі пропонується метод розподілу навантаження, метою якого є оптимізувати використання ресурсів у хмарних обчисленнях шляхом ефективного управління та розподілу робочих навантажень. Цей метод призначений для прогнозування структури навантаження, динамічного оркестрування мікросервісів та балансування навантаження між серверами для запобігання вузьких місць та недовикористання.

Метод складається з етапів, представлених у підрозділах.

3.2.1. Для якого ПЗ застосовується метод

Для представленого методу, основною якістю, якою має бути наділений елемент програмного забезпечення, є можливість горизонтального масштабування. Всі мікросервіси мають таку характеристику, тому для спрощення опису, поняття «мікросервіс» використовується в значенні елементу ПЗ, який може контейнеризуватись (be containerised) з потенційною метою реплікуватись задля збільшення пропускної можливості. Контейнеризація дозволяє упакувати застосунок разом із усіма його залежностями для незалежного запуску в різних середовищах. До групи підхожих елементів ПЗ можуть також належати усі безсерверні технології, наприклад, лямдба-функції (AWS lambda).

Слід звернути увагу, що є такі моноліти, які теж можна реплікувати, зокрема якщо у них нема прив'язки до внутрішнього стану, тобто їх поведінка не залежить від минулої роботи, але тільки від поточних запитів. При цьому, стан застосунку може бути наявним та зберігатись на клієнтській стороні, або приходити з БД або інших зовнішніх джерел. Також не має виникати конфліктів запису-читання з БД, коли до такої є прив'язка у випадку наявності декількох монолітних контейнеризованих запусчених застосунків. Додатково можна прочитати у [69] стосовно ПЗ, яке можливо контейнеризувати. Також, у розділі про [Горизонтальне масштабування](#) описано методику модифікації монолітного застосунку на такий, що є

придатний для масштабування та контейнеризації.

3.2.2. Прогнозування навантаження

Перший крок включає зняття та аналіз історичних даних для прогнозування майбутніх моделей навантаження. Визначаючи тенденції та час пікового використання, метод може передбачити періоди високого та низького попиту. Функція прогнозування дозволить системі підготуватися до різних навантажень, забезпечуючи ефективний розподіл ресурсів.

3.2.3. Групування серверів

Метою групування серверів є максимізація використання їх пропускну здатності каналів, пам'яті та процесорів. Поєднання декількох мікропроцесорів на одному сервері дозволяє досягти збалансованого розподілу навантаження.

Метод спрямований на оптимізацію використаних ресурсів під час роботи багатокомпонентної системи, зокрема, при масштабуванні її вузлів для забезпечення покриття навантаження всіх користувацьких запитів.

Метод виглядає наступним чином. Під час пошуку доповнювальних (комплементарних) навантажень мікропослуг метод виконує кластеризацію, в результаті чого екземпляри мікросервісів групуються за схожістю робочих шаблонів у класи еквівалентності. В межах кожної групи екземпляри сортуються за амплітудою використання ресурсів. Далі групи, що визначені як доповнювальні, перетинаються з метою знайти пари комплементарних мікропослуг. Екземпляри з протилежними шаблонами навантаження, але схожими амплітудами, комбінуються для ефективнішого використання ресурсів. Також можуть розглядатися пари з низькою амплітудою для заповнення «пробілів». Метод використовує жадібний підхід, щоб знайти перший екземпляр мікросервісу, що відповідає критеріям доповнення. Якщо доповнення в межах першої групи не знайдено, зберігається статистика комбінацій з усіма екземплярами, і пошук триває до знаходження доповнень або вичерпання доповнювальних груп.

Процес повторюється до знаходження всіх можливих пар, хоча залишкові екземпляри мікросервісів є очікуваними. Для мікросервісів, що не знайшли доповнень, застосовується додатковий аналіз екстремумів з використанням середнього та стандартного відхилення, щоб підвищити ймовірність знайти комплементи. Кластеризація і пошук продовжуються до останньої успішної спроби. Якщо залишається невелика кількість мікросервісів без пари, вони комбінуються за допомогою рішення задачі про множинний рюкзак [8], [9], [70].

Отже, кластеризуючи сервери відповідно до очікуваного попиту, система може створювати конфігурації з груповим навантаженням, які запобігають перевантаженню та недовикористанню потужностей, забезпечуючи збалансований розподіл ресурсів.

3.2.4. Оркестрування мікросервісів

Динамічне управління мікросервісами має вирішальне значення для реагування на мінливі навантаження. Організовується розгортання та закриття мікросервісів на основі попиту в режимі реального часу. У пікові періоди розгортаються додаткові мікросервіси, щоб впоратися зі збільшеним навантаженням. Нові екземпляри можуть мати стан «холодного» або «гарячого» навантаження, що означає, що вони підготовлені. Це можливо, якщо спрогнозувати час збільшення навантаження [5]. І навпаки, в непіковий час непотрібні мікросервіси деактивуються для економії ресурсів. Такий динамічний підхід гарантує, що система залишається швидкою та ефективною [71], [20].

3.2.5. Балансування навантаження

Щоб жоден ОР не став вузьким місцем, використовуються методи балансування навантаження, такі ж, як у хмарних системах або Kubernetes, але для груп. Запити рівномірно розподіляються між серверами, забезпечуючи безперебійну та ефективну роботу. Балансування навантаження допомагає підтримувати стабільність і продуктивність

системи, запобігаючи перевантаженню окремих серверів і забезпечуючи ефективне використання всіх ресурсів [3], [72], [73].

3.2.6. Перегрупування серверів

Під час роботи мають безперервно відстежуватись показники продуктивності для оцінки ефективності використання ресурсів. Якщо виявляється неефективність, слід ініціювати процес перегрупування серверів. Це передбачає перерозподіл ресурсів і реорганізацію груп серверів, щоб краще відповідати поточним умовам навантаження. Динамічно змінюючи конфігурацію груп серверів, система може покращити використання ресурсів і підтримувати оптимальну продуктивність.

3.3. Етапи групування мікропослуг

Зазвичай, коли програми працюють на сервері, ресурси не використовуються на повну потужність. Це може бути пов'язано з різними факторами, такими як специфікації програми, час доби або день тижня. Наприклад, у робочий час сервери можуть бути більш завантажені, ніж у нічний час або на вихідних. Важливою характеристикою навантаження є його передбачуваність, що дозволяє оптимізувати розподіл ресурсів.

Ключовим елементом ефективного управління та розподілу робочих навантажень є доповнювальний розподіл навантаження, який містить механізми виявлення схожих або взаємодоповнювальних мікросервісів. Ця можливість підвищує ефективність системи, гарантуючи, що мікросервіси згруповані та організовані таким чином, щоб максимізувати їхній синергетичний потенціал. Нижче зображені ключові етапи цього процесу.

3.3.1. Профілювання мікропослуг

Першим кроком у визначенні подібних або взаємодоповнювальних мікросервісів є профілювання кожного мікросервісу в системі. Профілювання передбачає збір детальних метаданих про кожен мікросервіс, включаючи його обчислювальні вимоги, типові моделі навантаження, споживання ресурсів і функціональні залежності. Ці метадані дають повне

уявлення про те, як працює кожен мікросервіс. На основі цієї інформації буде зроблено висновок про те, як організувати взаємодію цього мікросервісу з іншими.

Для того, щоб ефективно керувати та розподіляти робочі навантаження в хмарних обчисленнях, важливо точно вимірювати навантаження на процесор. Навантаження на процесор можна оцінити кількісно, враховуючи

1. кількість ядер,
2. основний потенціал, і
3. поточне навантаження на кожне ядро.

Більш детально про те, як це робиться, описано в розділі [Завантаження процесора](#). В Таблиці 3.2 наведено приклад характеристик та їх зазначень. Також слід вказати основні якості кожного ресурсу, оскільки подібне слід порівнювати з подібним.

Таблиця 3.2 Приклад загального профілювання мікропослуг

Назва характеристики	Особливості	Значення
Пропускна здатність каналу	10 Мбіт/с (Ethernet)	8 Мб/с
δ_{Channel}	Добові коливання навантаження на канал	$\pm 2\%$
Навантаження на процесор (CPU)	2,5 ГГц чотирьохядерн ий Intel Core i7	75%
Ядра	Чотирьохядерн ий	4
δ_{CPU}	Щоденні коливання навантаження на процесор	$\pm 8\%$
Оперативна пам'ять	8 ГБ 1600 МГц DDR3	4 Gb

δ_{RAM}	Щоденні коливання оперативної пам'яті	$\pm 7\%$
Жорсткий диск (HD)	Жорсткий диск Seagate BarraCuda	60 Gb
Можна розбивати на частини (canChunk)	Необхідний, якщо вимоги занадто високі	Так.

3.3.2. Аналіз подібності

Використовуючи зібрані профілі, виконується аналіз подібності, щоб виявити мікропослуги зі схожими характеристиками. Такі методи, як кластеризація, кореляційний аналіз і розпізнавання образів, використовуються для групування мікропослуг зі схожими моделями навантаження, використанням ресурсів і робочою поведінкою. Таке групування допомагає формувати кластери серверів, які можуть ефективніше обробляти однорідні робочі навантаження.

3.3.3. Аналіз взаємодоповнюваності

На додаток до пошуку схожих мікропослуг, можна також визначати взаємодоповнювальні мікропослуги — ті, які, будучи розгорнутими разом, підвищують загальну продуктивність системи. Аналіз взаємодоповнюваності передбачає оцінку функціональних залежностей і моделей використання ресурсів, щоб визначити, які мікропослуги добре працюють разом. Наприклад, мікросервіс з високим навантаженням на процесор, але низьким споживанням пам'яті може доповнювати інший мікросервіс з протилежною схемою використання ресурсів.

3.3.4. Динамічне групування

На основі результатів аналізу подібності та взаємодоповнюваності, динамічно формуються групи мікропослуг, які є схожими або взаємодоповнювальними. Потім ці групи розгортаються на одному сервері

або кластері серверів для оптимізації використання ресурсів. Забезпечуючи спільне використання ресурсів схожими мікросервісами, а взаємодоповнювальні мікропослуги врівноважують навантаження один одного, система може досягти більшої ефективності та стабільності.

3.3.5. Постійний моніторинг та коригування

Система постійно відстежує продуктивність згрупованих мікропослуг, щоб гарантувати, що початкове групування залишається оптимальним. Якщо показники продуктивності вказують на неоптимальне використання ресурсів або збільшену затримку, відбувається переоцінка групування і вносить необхідні корективи. Цей динамічний процес перегрупування гарантує, що система адаптується до мінливих умов і підтримує високу продуктивність.

3.4. Основні характеристики та особливості хмарного середовища

Основні характеристики мікросервісів, такі як навантаження на процесор, використання пам'яті та мережевого трафіку, спочатку визначаються на основі середовища, в якому вони розробляються та тестуються. Однак ці середовища часто відрізняються від реальної хмарної інфраструктури, де мікропослуги з часом будуть розгорнуті. Ця невідповідність може призвести до відмінностей у показниках продуктивності, оскільки хмарні провайдери часто оптимізують свої сервіси для ефективної роботи у власних екосистемах.

Розглянемо Amazon DynamoDB як приклад. DynamoDB — це повністю керована служба баз даних NoSQL, що пропонується Amazon Web Services (AWS). Він розроблений для забезпечення швидкої та передбачуваної продуктивності з легкою масштабованістю. Архітектура DynamoDB оптимізована для інфраструктури AWS, пропонуючи покращені характеристики продуктивності порівняно з іншими NoSQL базами даних, які можуть бути використані на етапі розробки. У джерелах є інформація про порівняння DynamoDB з деякими іншими БД [74], [74], [75].

Розробники можуть спочатку профілювати свої мікропослуги, використовуючи різні бази даних NoSQL, такі як MongoDB або Cassandra, на локальній або не-AWS інфраструктурі. Коли ці мікропослуги мігрують до AWS і починають використовувати DynamoDB, вони можуть демонструвати інші характеристики продуктивності. Наприклад, DynamoDB може запропонувати меншу затримку та вищу пропускну здатність завдяки оптимізації, специфічній для AWS, що призводить до більш ефективного використання ресурсів [76], [77].

Попри ці відмінності, початкова статистика, надана командою розробників, все одно повинна слугувати розумною основою для розгортання хмари. Ці статистичні дані включають

1. завантаження процесора: відсоток потужності процесора, що використовується мікросервісом;
2. використання пам'яті: обсяг оперативної пам'яті, необхідний для оптимальної роботи;
3. попит на мережу: вимоги до пропускну здатності та затримки для зв'язку.

Хоча ці початкові показники можуть не повністю відповідати продуктивності хмарного середовища, вони є відправною точкою. Для плавного переходу дуже важливо встановити межі на основі цих початкових статистичних даних. Ці межі діють як огорожа, гарантуючи, що мікропослуги працюють у прийнятних межах після розгортання в хмарі.

3.4.1. Використання оперативної пам'яті

Профілювання на основі використання оперативної пам'яті передбачає моніторинг моделей споживання пам'яті кожним мікросервісом. Деякі мікропослуги можуть потребувати великих обсягів пам'яті для кешування, обробки даних або інших операцій, тоді як інші можуть мати мінімальні потреби в пам'яті. Коли на комп'ютері менше оперативної пам'яті, ніж потрібно програмі, це може призвести до частого підкачування

або свопінгу. Цей процес передбачає переміщення даних між оперативною та дисковою пам'яттю, що значно уповільнює роботу програми [78]. Цьому слід запобігати, наскільки це можливо.

Достатня або надлишкова оперативна пам'ять гарантує, що дані та процеси програми зберігаються в пам'яті, що призводить до швидшого доступу до них і кращої загальної продуктивності. Більший обсяг оперативної пам'яті дозволяє краще підтримувати паралельні процеси та потоки, забезпечуючи вищий рівень паралелізму та ефективну багатозадачність. Застосунки, які значною мірою покладаються на обробку даних у пам'яті (наприклад, бази даних, задачі аналізу великих обсягів даних), отримують більше користі від збільшення обсягу оперативної пам'яті, що особливо важливо у хмарному середовищі, де є великий спільний простір пам'яті.

Використовуйте хмарні сервіси, які підтримують динамічний розподіл ресурсів [79], що дозволяє програмам запитувати додаткову пам'ять за потреби. Такі сервіси, як AWS Auto Scaling та Azure Virtual Machine Scale Sets можуть допомогти ефективно керувати ресурсами. Проте, використовуючи їх, користувач повинен розраховувати на горизонтальне масштабування, тобто відкриття нових екземплярів [80]. Відомі хмарні провайдери не пропонують автоматичне вертикальне масштабування, тобто додавання лише оперативної пам'яті без інших додаткових ресурсів чи перероблювання VM [81].

AWS пропонує використовувати різні типи екземплярів, які мають більше оперативної пам'яті, в межах попередньо визначених у конфігурації груп автоматичного масштабування, визначивши їх у шаблоні запуску. Azure Automation пропонує створювати сценарії, які відстежують показники продуктивності та змінюють розмір VM при досягненні певних порогових значень. Такий підхід вимагає складної стратегії налаштування та управління. Google Cloud Platform працює аналогічно, зупиняючи VM, перерозподіляючи ресурс пам'яті та запускаючи її знову.

Класифікуючи мікропослуги відповідно до використання ними оперативної пам'яті визначається, які мікропослуги можуть бути розміщені на одному сервері, не викликаючи конфлікту пам'яті. Розробники повинні проводити навантажувальне тестування на різних конфігураціях машин, щоб зрозуміти, як різний обсяг оперативної пам'яті впливає на продуктивність, щоб правильно сформулювати вимоги до програмного забезпечення. Це допомагає приймати обґрунтовані рішення щодо розподілу пам'яті та запобігати проблемам, пов'язаним з надмірним або недостатнім використанням пам'яті.

3.4.2. Навантаження на канал

Профілювання навантаження на канал передбачає оцінку вимог мікросервісів до мережі та вводу/виводу. Це включає моніторинг використання пропускної здатності, чутливості до затримок і швидкості передачі даних, необхідних для кожного мікросервісу. Як і будь-яка інша характеристика, деякі мікросервіси потребують високої пропускної здатності мережі для операцій з великими обсягами даних, наприклад, передачі відео або інших великих файлів, тоді як інші можуть бути більш чутливими до затримок і потребувати каналів зв'язку з низькою затримкою, наприклад, для пакетної передачі.

Фізична відстань між джерелом та отримувачем даних впливає на затримку та використання пропускної здатності. Розподілені системи можуть відчувати більше навантаження на канал через передачу даних на великі відстані [72] Кількість одночасних користувачів або пристроїв, що отримують доступ до мережі, впливає на навантаження на канал. Висока паралельність може призвести до перевантажень і зниження продуктивності. Спільні мережеві ресурси можуть стати вузькими місцями, коли кілька додатків або сервісів конкурують за одну і ту ж смугу пропускання [3], [73]. Трафік з раптовими сплесками передачі даних може перевантажувати мережеві канали, викликаючи тимчасові затори і втрату

пакетів [3], [73]. Це проблеми, які можуть спричинити серйозні неприємності, і хмарне середовище допомагає вирішити багато з них, зокрема, завдяки балансуванню навантаження та стабільному з'єднанню між глобальною мережею центрів обробки даних. У випадку з IoT на перший план виходять периферійні обчислення. Граничні обчислення наближають обчислення і зберігання даних до місця, де вони потрібні, щоб поліпшити час відгуку і заощадити пропускну здатність [72], [82], [83]. Хмарні провайдери пропонують рішення для периферійних обчислень, які дозволяють обробляти дані ближче до кінцевих користувачів або пристроїв IoT.

Зі стабільним, передбачуваним потоком даних легше керувати та оптимізувати роботу додатків. Розуміючи метрики завантаження каналу, його пропускну здатність, затримки, можливі втрати пакетів і джиттер, проводячи навантажувальні тести на каналах з різною пропускну здатністю, можна зрозуміти вимоги до каналу і вести статистику використання. Використовуючи ці вимоги в кожен момент часу, можна згрупувати мікропослуги таким чином, щоб оптимізувати використання мережевих ресурсів. Кілька мікросервісів, згрупованих разом, можуть зробити одну більш передбачувану систему і водночас отримати більше ресурсів для каналу. Це сприятиме зменшенню перевантаження мережі, підвищенню ефективності передачі даних та забезпечить безперебійну роботу мікросервісів, чутливих до затримок.

3.4.3. Завантаження процесора

Профілювання навантаження на процесор фокусується на моделях використання процесора мікросервісами. Деякі мікропослуги можуть виконувати ресурсомісткі завдання, такі як шифрування даних, складні обчислення або обробка в реальному часі, що призводить до високого навантаження на процесор. Інші можуть мати менші вимоги до процесора. Аналізуючи характеристики навантаження на процесор, можна згрупувати

мікросервіси, які або мають схожі моделі використання процесора, або можуть доповнювати один одного, балансуючи високе і низьке навантаження на процесор на одному сервері. Це забезпечує ефективне використання процесора і дозволяє уникнути сценаріїв, коли деякі мікросервіси голодують за процесорний час, тоді як інші залишають процесор недозавантаженим.

Сучасні процесори мають кілька ядер, кожне з яких здатне одночасно виконувати окремі завдання. Загальна обчислювальна потужність сервера безпосередньо пов'язана з кількістю ядер, які він має. Залежно від програми, може бути важливо надати мікросервісу певну кількість ядер. Ця інформація зрозуміла лише розробникам, оскільки саме вони можуть використовувати спеціальні алгоритми розпаралелювання, які ефективні для певної кількості ядер, наприклад, до 2-х, 3-х, 4-х або будь-якого множника цих чисел відповідно.

Кожне ядро має певну пропускну здатність, яка зазвичай вимірюється в ГГц. Пропускна здатність ядра визначає, скільки інструкцій в секунду може обробляти ядро. Поточне навантаження на кожне ядро зазвичай виражається у відсотках від його загальної потужності. Це навантаження відображає співвідношення часу, протягом якого ядро активно виконує завдання, і часу, протягом якого воно простоює. Час, за який береться ця інформація, також важливий. Деякі мікропроцесори можуть мати випадкове навантаження, оскільки завдання мають різну якість і надходять у випадковий час. Проте, ідея мікропроцесорів полягає у вирішенні певного типу завдань, тому зазвичай дотримується регулярність у часі завантаження процесора, якщо не з дня на день, то з тижня на тиждень.

Перетворення цих трьох факторів в одну репрезентативну цифру може спростити процес розподілу та оптимізації ресурсів. Проте комп'ютери, на яких тестується програмне забезпечення, можуть бути іншої якості, ніж ті, що знаходяться в серверних кімнатах хмарного провайдера. Брендом можна знехтувати, але потужність — це те, що слід витягти і

нормалізувати до можливостей хмарного середовища. Пізніше, на ітерації [Перегрупування серверів](#) мікросервіс може бути замінений, якщо його початкові характеристики були оцінені дуже неправильно. Кроки наведено нижче.

Щоб розрахувати *загальне навантаження на процесор* і перетворити його в однозначне число, необхідно виконати наступні кроки [84]. Для кожного ядра навантаження можна розрахувати за формулою:

$$\text{Load per Core} = \left(\frac{\text{Current Load (\%)}}{100} \right) \times \text{Core Capacity (GHz)} \quad (3.1)$$

Загальне навантаження на процесор - це сума навантажень на всі ядра:

$$\text{Total Load} = \sum_{i=1}^N \left(\left(\frac{\text{Load}_i (\%)}{100} \right) \times \text{Capacity}_i (\text{GHz}) \right) \quad (3.2)$$

де N — кількість ядер, Load_i навантаження i -го ядра, i — смієність i -го ядра.

Щоб перетворити загальне навантаження в однозначне число, необхідно нормалізувати його відносно максимально можливого навантаження. Максимально можливе навантаження виникає, коли всі ядра працюють на 100% потужності. Тому:

$$\text{Max Possible Load} = \sum_{i=1}^N \text{Capacity}_i (\text{GHz}) \quad (3.3)$$

Нормоване навантаження буде:

$$\text{Normalized Load} = \left(\frac{\text{Total Load}}{\text{Max Possible Load}} \right) \times 10 \quad (3.4)$$

3.4.3.1. Приклад розрахунку завантаження процесора

Припустимо, що є процесор з наступними характеристиками:

- 4 ядра, кожне потужністю
- 2,5 ГГц, і
- поточні навантаження 60%, 75%, 50% та 90% відповідно.

Розрахуйте навантаження на кожне ядро, використовуючи (3.1):

$$\begin{aligned} Core1 &: \left(\frac{60}{100}\right) \times 2.5 = 1.5GHz \\ Core2 &: \left(\frac{75}{100}\right) \times 2.5 = 1.875GHz \\ Core3 &: \left(\frac{50}{100}\right) \times 2.5 = 1.25GHz \\ Core4 &: \left(\frac{90}{100}\right) \times 2.5 = 2.25GHz \end{aligned} \quad (3.1')$$

Підсумовування навантажень усіх ядер за формулою (3.2):

$$Total Load = 1.5 + 1.875 + 1.25 + 2.25 = 6.875 GHz \quad (3.2')$$

Розрахуймо максимально можливе навантаження за формулою (3.3):

$$Max Possible Load = 4 \times 2.5 = 10 GHz. \quad (3.3')$$

Нормалізуймо загальне навантаження за допомогою (3.4):

$$Normalized Load = \left(\frac{6.875}{10}\right) \times 10 = 6.875 \quad (3.4')$$

Нормалізоване навантаження, переведене в число, становить приблизно 6,88. Це значення можна округлити до одного знаку після коми, тобто до 6,9.

3.4.3.2. Трансформація розрахунку відсотка завантаження на нове середовище

При перенесенні значень, розрахованих на середовище розробника, в хмарне середовище слід враховувати кілька складнощів. Ось фактори, які слід враховувати.

1. **Накладні витрати на міждерний зв'язок** – більша кількість ядер може призвести до збільшення накладних витрат на управління зв'язком між ядрами, особливо якщо мікросервіси потребують частоті синхронізації або обміну даними.

2. **Неефективність планування завдань** – розподіл завдань між більшою кількістю ядер може призвести до неефективності планування завдань, що потенційно може призвести до недовикористання деяких ядер.

Через згадані вище причини, зі збільшенням кількості ядер приріст

продуктивності від додавання кожного додаткового ядра може зменшуватися. Щоб врахувати ці фактори, можна ввести коригувальний коефіцієнт α який враховує накладні витрати, пов'язані з розподілом завдань між більшою кількістю ядер. Цей коефіцієнт можна отримати емпіричним шляхом або оцінити на основі типової поведінки системи. Це слід робити на основі статистики хмар з їхнім специфічним середовищем. Пристосування до потреб хмари саме по собі є зміною бази процесора.

Формула для скоригованого відсотка навантаження на ядро стає такою:

$$\text{Adjusted Load Percentage per Core} = \frac{\text{Total Required Load}}{\text{Max Possible Cloud Load}} \times (100 + \alpha) \quad (3.5)$$

Використання коефіцієнта коригування множить нескоригований відсоток навантаження на 110%, щоб врахувати неефективність.

Застосуємо коригувальний коефіцієнт, згаданий у (5), до наведеного прикладу, описаного в розділі [Приклад розрахунку завантаження процесора](#) з перенесенням обчислень у хмарне середовище для обчислення скоригованого відсотка завантаження.

Задано групу серверів з наступними характеристиками:

- Загальне необхідне навантаження: 6,875 ГГц
- Кількість ядер: 8
- Місткість ядра: 3,0 ГГц кожне
- Коефіцієнт коригування: 0,1 або 10% (разом зі 100% дає 110%)

Розрахуємо загальну місткість хмарного середовища, яка називається *Max Possible Cloud Load*, за формулою (3.3):

$$\text{Max Possible Cloud Load} = 8 \times 3.0 = 24 \text{ GHz}. \quad (3.3'')$$

Скоригований відсоток навантаження на ядро розраховується наступним чином:

$$\text{Adjusted Load Percentage per Core} = \left(\frac{6.875}{24} \right) \times 110 = 28.65 \times 1.1 = 31.515\% \quad (3.4'')$$

Підсумовуючи, початкове значення 68,7% від загального

навантаження на хмарний ОР слід замінити на 31,5% від загального навантаження. Невеликий відсоток пристосування до нового середовища дозволяє врахувати додаткові накладні витрати та неефективність при розподілі завдань між більшою кількістю ядер. Цей скоригований відсоток навантаження забезпечує більш точне відображення фактичного завантаження процесора в хмарному середовищі. 10% у прикладі – це досить великий відсоток, і він взятий лише для зручності підрахунку. На практиці він має бути меншим.

3.5. Алгоритм пошуку доповнювальних навантажень

В цьому розділі представлений алгоритм для пошуку комплементарних або доповнюваних навантажень, що описує як декілька елементів програмного забезпечення (ПЗ) або задач можуть сформувати повне навантаження на серверну групу, таким чином використовуючи максимально повно її ресурс. Високорівнева блок-схема алгоритму наведена на Рисунку 3.2. Її опис знаходиться в підпунктах.

Алгоритм, як і метод доповнювальних навантажень, оснований на ідеї доповнень в нечіткій теорії ґраток [7]. Елементи ПЗ, мікросервіси чи моноліти, для спрощення будуть надалі називатись мікросервісами, адже і до монолітів можна ставитись як до мікросервісів при використанні скейлінгу (запуску додаткових екземплярів та користуванні балансувальника навантаження). У випадках, коли поведінка мікросервісів та монолітів відрізнятиметься, буде слідувати додаткове пояснення.

Алгоритм наводиться по пунктах, у деяких пунктах є посилання на інші, тому номери допоможуть зорієнтуватись.

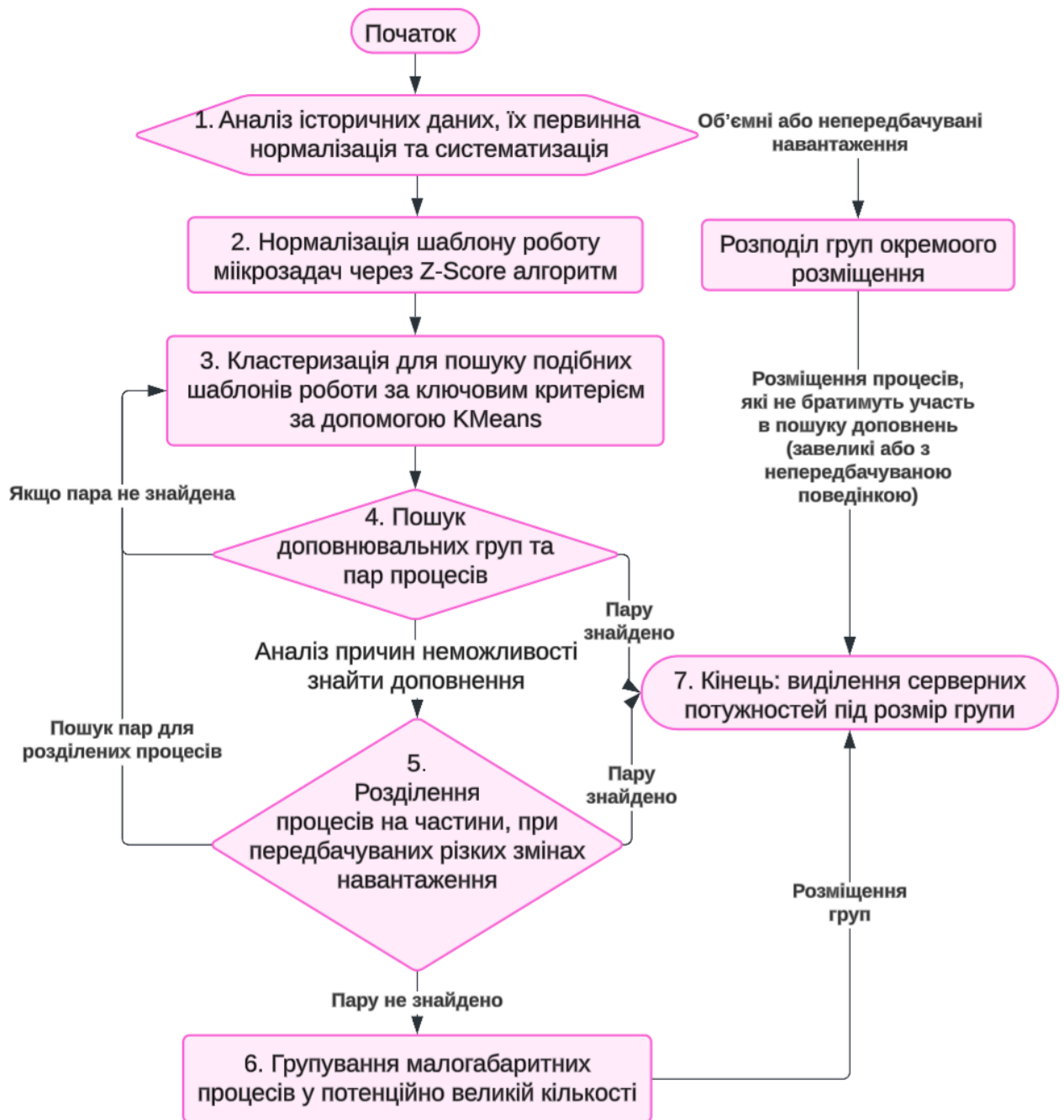


Рисунок 3.2 — Блок-схема алгоритму пошуку доповнювальних навантажень

3.5.1. Етап 1: Аналіз історичних даних, їх первинна нормалізація та систематизація

Ефективне управління ресурсами в хмарних обчисленнях передбачає кілька етапів підготовки мікросервісів, групування на основі моделей використання ресурсів, а також пошук взаємодоповнювальних мікросервісів у цих групах. Коли група сформована, її можна розгортати на

серверах. Такий підхід підвищує продуктивність і ефективність використання ресурсів. Крім того, мікропослуги з високими вимогами слід розділяти на менші частини, якщо це можливо. У цьому підрозділі наведено детальне пояснення цього процесу.

3.5.1.1. Крок 1: Збір історичних даних роботи програмних застосунків

Наступні кроки є специфічними для хмарних технологій і повинні базуватися на апаратному забезпеченні хмарного провайдера. Ідея полягає в тому, щоб перетворити метрики, зібрані командою розробників, на метрики, що відповідають ресурсам, присутнім у серверних кімнатах. Зазвичай більшість серверів схожі або однакові, тому це полегшує завдання.

Вхідні дані: статистика стосовно роботи кожного мікросервіса або моноліта, який сприйматиметься як мікросервіс. Статистика включає наступні дані:

- CPU (очікуване завантаження в інтервал часу, кількість ядер та їх частота);
- RAM (очікуване завантаження та загальна необхідна кількість);
- каналний ресурс — (інтервальний очікуваний об'єм та загальна пропускна здатність каналу);
- базові характеристики техніки, на якій виконувались випробування, такі як кількість ядер, вимоги до жорсткого диска;
- прапорець можливості поділу чи реплікації вхідного елементу ПЗ `canBeChunked`;
- географічна зона для роботи мікросервісу. Вона може задаватись на етапі вхідних даних замовником, або ж вираховуватись в процесі роботи ПЗ на клауд-системі та оновлюватись на наступних ітераціях алгоритму.

3.5.1.2. Крок 2: Первинна нормалізація вхідних даних

Перед початком роботи з ПЗ, яке прийшло з іншої системи, слід перевести метрики з іншого середовища на підхожі серверні потужності клауд-провайдера.

Щоб переконатися, що вхідні дані є коректними й відповідають хмарному середовищу, або ж їх підлаштувати під специфіку підхожих серверів, слід скористатись формулою (3.5), тобто всі зібрані метрики необхідно розділити на відповідну характеристику навантаження стандартного сервера хмарного середовища, враховуючи можливу похибку або коефіцієнт коригування. Етапи наступні:

1. Зібрати дані про використання зазначених ресурсів задачами.
2. Змінити базу для профілювання даних (нормалізувати) за характеристиками хмарних серверів, використовуючи формулу (3.5).

3.5.1.3. Крок 3: Поділ велико-затратних елементів ПЗ

Ділення повного необхідного ресурсу на менші можливо для мікропроцесорів, а також здебільшого і для монолітів. Для цього в початковій умові задається параметр «*canChunk*», тобто можливість поділу.

На практиці «поділ» називається масштабування, тобто при надмірному навантаженні створення додаткових екземплярів мікросервісів, або ж дублювання монолітів. Для розв'язання задачі ефективного використання ресурсів, зручніше оперувати поняттям повного та часткового навантаження, тому замість «розширення» використовується слово «поділ». На Рисунках 2.1 та 2.2 зображені дерева з листками. Листки являють собою додаткові репліки запущеної задачі у відповідний час. Це і є розділення задачі, на шматки, якими зручно оперувати.

Поділ здійснюється на основі найбільшого, тобто **ключового показника** (CPU, RAM чи каналний ресурс). Жоден з показників не має перевищувати дозволеної норми. Після поділу ключовим показником стає новий найбільший показник. Навантаження мікросервісу в час його найбільшого піка має бути меншим за граничне значення.

$$P_{t_{max}} \leq P_{boundary} - P_{boundary} \times \alpha_{boundary}, \quad (3.6)$$

де $\alpha_{boundary}$ — прийнятний коефіцієнт недоповненості навантаження, яке може бути рівним D_{cmax} (див. Крок 9), тобто нехай понад 70–80%, то до нього буде важко знайти доповнювальний. Такий мікросервіс слід теж помітити як габаритний.

3.5.1.4. Крок 4: Систематизація всіх мікрозадач, що потребують серверного розміщення

Перш ніж групувати та шукати схожі та доповнювальні мікропослуги, необхідно відфільтрувати мікропослуги, які можуть не підійти. Це мікропослуги з надзвичайно високими вимогами до ресурсів. Позначимо P_{tmin} момент часу, коли навантаження досягає максимального значення. Тоді критерій для поділу виглядає наступним чином:

$$P_{tmax} + \delta \leq P_{boundary}, \quad (3.7)$$

де δ — невеликий відсоток для врахування добових коливань, яке можна асоціювати з $\alpha_{reserve}$ — безпечна резервна частка ресурсу, яку слід залишити пустою при плануванні навантаження. Ці дві величини можна сприймати як взаємозамінні, але їх введено дві через контекст, яким зручніше оперувати.

Ці мікропослуги повинні бути розділені на менші частини, якщо їх можна розбити на частини. Можливо, це буде монолітний додаток, оскільки ідея мікросервісів пов'язана з зазвичай невеликими завданнями, які відокремлені від інших [85], тому вони не повинні займати багато місця. Більше інформації про поділ мікросервісів та використання меж можна знайти в [8]. Такий поділ може допомогти передбачити час запуску мікросервісів і підготувати додаткові сервери, встановивши на них передстартові дані та записавши останню актуальну інформацію, наприклад, про холодний, теплий або гарячий стан очікування [5].

При декомпозиції монолітного додатку на менші компоненти доцільно базувати сегментацію на звичайному або середньому навантаженні, з яким система справляється. Для періодів, коли навантаження перевищує середнє, ефективно передбачити додаткові

екземпляри моноліту, які налаштовані на обробку підвищеного попиту. Ця стратегія гарантує, що система залишатиметься швидкою та стабільною під час пікових навантажень, водночас оптимізуючи використання ресурсів під час нормальної роботи. Рисунок 3.3 наведено для візуалізації процесу, він впроваджує в життя ідею, зображену на Рисунку 2.1.

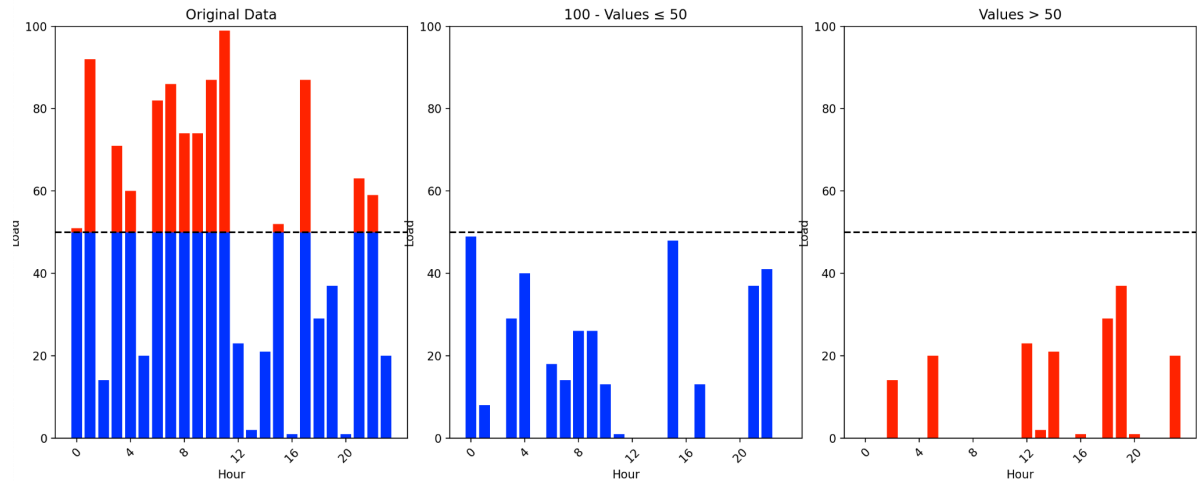


Рисунок 3.3 — Поділ високонавантаженого мікросервісу на два екземпляри

Поділ може відбуватись на основі активності роботи в межах часових проміжків. Створення додаткового екземпляра відбуватиметься тільки на певний період часу, коли буде перевищено максимальне допустиме значення за якоюсь характеристикою з кроку 1:

$$P_{boundary} \leq P_{t_{max}} + \alpha_{reserve} \cdot P_{t_{max}} . \quad (3.7)$$

Поділ відбуватиметься на стільки частин m , скільки повних максимальних завантажень з урахуванням безпечного резерву $\alpha_{reserve}$ може вміститись в заданий елемент, плюс те навантаження, яке не буде повним, але залишковим від ділення:

$$m = \left\lceil \frac{P_{max} + \alpha_{reserve}}{P_{boundary}} \right\rceil . \quad (3.8)$$

У випадках, коли той самий мікросервіс необхідно запустити на кількох географічних зонах, він не буде сприйматись як той самий мікросервіс із погляду алгоритму. Якщо однаковий мікросервіс деплоїться (інсталюється) на кілька географічних зон, щоб швидше забезпечувати

відповідь клієнтам, розташованим у цих зонах, попередньо необхідно зробити заміри використання потужностей, що будуть відповідати цим зонам. Такі заміри можуть бути наведені у вхідних даних або обчислені на наступних ітераціях клауд-провайдером.

Існують ситуації, коли створення додаткових екземплярів програми може бути недоцільним або невигідним. Наприклад, якщо додаток починає послідовно отримувати незвично великі повідомлення — замість невеликих, частих. Це може призвести до перевантаження ресурсів каналу. Такі сценарії, які можуть бути очевидними лише для бізнес-аналітиків і розробників, знайомих з контекстом задачі, повинні спонукати цих зацікавлених осіб встановити атрибут `canChunk` у таблиці 3.2 у значення `false`, що вказує на недоцільність поділу задачі. Це слід розглядати як винятковий випадок. Крім того, можна перепроєктувати програму для більш ефективного управління передачею даних, таким чином зменшивши потребу у додаткових екземплярах.

Для цієї частини алгоритму будуть наступні вхідні значення:

- «canChunk» — ця характеристика вказує на те, чи можна розділити мікросервіс на менші, більш керовані частини,
- і максимально можливі значення, на основі яких відбудеться відсікання надмірних навантажень:
 - $P_{\text{boundary, RAM}}$
 - $P_{\text{boundary, CPU}}$
 - $P_{\text{boundary, Channel}}$

Алгоритм поділу задачі на частини наступний:

1. Визначити мікропослуги з потребами в ресурсах, що перевищують $P_{\text{boundary, RAM}}$, $P_{\text{boundary, CPU}}$ або $P_{\text{boundary, Channel}}$.
2. Перевірити характеристику «canChunk».
3. Якщо (3.7) виконується, розділити мікропослуги з високими вимогами (опис і приклад наведено нижче).
4. Сформулювати окрему групу «Мікропослуги з високими вимогами» для тих, які не можуть бути розбиті на частини, але перевищують

межі. Елементи з цієї групи займатимуть окремі великі ресурси і не братимуть участі в наступному алгоритмі.

Поділ, згаданий у 3-му кроці, відбувається, якщо мікропослугу або моноліт можна і потрібно подрібнити, тобто розділити на менші частини з потребами в ресурсах, нижчими за максимальні порогові значення. Такий поділ здійснюється лише тоді, коли конкретна характеристика перевищує своє мінімальне значення на певний відсоток, відомий як « Δ » (дельта), а їх сума є нижчою за поріг. Δ являє собою додатковий відсоток, який слід додати до мінімального значення характеристики, що призводить до поділу.

Нехай P_{tmin} мінімальний поріг для характеристики, Δ відсоток приросту від 0% до 100%, і P_{boundary} максимальний поріг. Ця формула повинна також включати невеликий відсоток варіацій δ , які можуть відбуватися щодня. Цей параметр надається разом з початковими метриками. Алгоритм працює для всіх критеріїв однаково, тому формули і приклади написані без уточнення, для якої саме характеристики вони взяті. Поділ має відбуватися, якщо характеристика перевищує P_{boundary} :

$$P_{\text{chunk}} = P_{\text{tmin}} + \Delta P_{\text{tmin}} + \delta < P_{\text{boundary}} \quad (3.9)$$

де:

P_{chunk} – поріг для ініціювання поділу,

P_{tmin} – момент з мінімальним значенням певної характеристики,

Δ – відсоток збільшення, необхідний для поділу,

P_{boundary} – максимальне порогове значення.

Наприклад, якщо $\Delta = 30\%$, $\delta = 5\%$, $P_{\text{tmin}} = 10$, і $P_{\text{boundary}} = 20$.

Підставивши значення у формулу, отримаємо наступне:

$$P_{\text{chunk}} = 10 + 0.30 \times 10 + 0.05 \times 10 = 10 + 3 + 0.5 = 13.5. \quad (3.9')$$

Оскільки 13,5 менше граничного значення 20, мікросервіс повинен бути розділений, якщо характеристика перевищує 13,5 одиниць. Це гарантує, що мікропослуги ефективно розділяються тільки тоді, коли це необхідно, оптимізуючи використання ресурсів у хмарному середовищі.

Поділ все одно буде відбуватися за граничним значенням, а не за відсотком Δ .

Для визначення кількості частин m на які слід розбити мікросервіс, використовуємо формулу (3.8). Ця формула враховує суму максимального навантаження P_{\max} і невеликий відсоток для врахування добових коливань δ ділить цю суму на граничне значення P_{boundary} та округлення до найближчого цілого числа, яке позначається функцією «підлога-стеля $\lfloor \cdot \rfloor$ ».

В результаті останнє навантаження на мікропроцесор може виявитися занадто малим і ним можна знехтувати. Критерії повинні бути специфічними для хмарних та інших обчислень. Пропонується не створювати останній блок, якщо залишок не перевищує δ . Цей параметр буде використовуватися в інших формулах і повинен враховуватися при підрахунку загального навантаження групи. У формулах це виглядає наступним чином:

$$\text{якщо } \left(\left\lfloor \frac{P_{\max} + \delta}{P_{\text{boundary}}} \right\rfloor - \frac{P_{\max} + \delta}{P_{\text{boundary}}} \right) \leq \delta \quad \text{то } m = \left\lfloor \frac{P_{\max} + \delta}{P_{\text{boundary}}} \right\rfloor \quad \text{інакше } m = \left\lceil \frac{P_{\max} + \delta}{P_{\text{boundary}}} \right\rceil, \quad (3.10)$$

Якщо P_{\max} дорівнює 50, δ дорівнює 8% або 0,08, що становить $50 * 0,08 = 4$, а P_{boundary} дорівнює 20, то результати застосування формул (8) та (9) розраховуються наступним чином:

$$m = \left\lfloor \frac{50 + 4}{20} \right\rfloor = \left\lfloor \frac{54}{20} \right\rfloor = \lfloor 2.7 \rfloor = 3. \quad (3.10')$$

Оскільки $3 - 2.7 = 0.7$ що < 0.08 , то округлення робиться в більшу сторону

На виході: поділені мікропослуги з новими ідентифікаторами, розподілені за географічними зонами. Кожна географічна зона містить наступний список груп:

1) список мікросервісів вільного розподілу, тобто мікропослуги, які можна комбінувати з будь-якими іншими;

2) список списків мікросервісних груп, які мають знаходитись разом на підставі використання спільного ресурсу або питань безпеки;

3) список мікросервісів, які вимагають окремого серверного ресурсу, адже їх потреби завеликі, навантаження непередбачуване, або через на питання безпеки. Виявити великогабаритних користувачів можна за наступним принципом: якщо максимальне значення мікросервісу велике, близьке до максимального, а середнє навантаження перевищує поріг, скажімо, 70%, то цей мікросервіс потребує окремого екземпляру.

Дивлячись на блок-схему на Рисунку 3.2, з першого блоку відходить блок окремого розміщення. Це і є виділення задач, які не будуть брати участь у подальшому алгоритмі та заносяться в групу три вищезгаданого списку.

3.5.2. Етап 2: Нормалізація шаблонів роботи мікрозадач

Подібність слід шукати лише після нормалізації векторів. Ця нормалізація відрізняється від тієї, що була виконана на попередніх кроках, описаних у підрозділі [Крок 2: Первинна нормалізація вхідних даних](#). Тут ідея полягає в тому, щоб привести всі характеристики мікросервісів до однієї логічної шкали, а не адаптувати їх до фізичних серверів хмари. Ця нормалізація буде використовуватися лише для пошуку схожої поведінки та кластеризації, яка буде наступним кроком, а не для пошуку результатів [10], [70].

3.5.2.1. Крок 5: Нормалізація шаблона роботи мікросервісу

Основним критерієм для ідентифікації подібних мікросервісів слугує шаблон вжитку ресурсу програмою. Є три неперервні види навантаження, які слід враховувати під час обчислення подібності, описані в підрозділі [Основні характеристики та особливості хмарного середовища](#). Їх статистичні дані можна порівнювати окремо або об'єднати в один масив, який буде однакового розміру для кожного мікросервісу. В роботі пропонується збирати статистику навантаження стандартних мікрозадач щогодини, бо у випадку малих змін щогодини, такий розріз буде не занадто дрібним, а у випадку різких змін навантаження протягом години не буде

кардинально важко системі пережити посилене навантаження без додаткової допомоги. Таким чином, запис трьох неперервних видів навантаження разом складе 3 помножити на 24, тобто 72 місця в результативному масиві.

Для більш детальної кластеризації, яка може мати сенс за наявності великого набору мікропослуг, зробити об'єднання тижневих даних. Як меншу і швидшу альтернативу можна розглянути поєднання робочого і вихідного днів. При наявності декількох мікросервісів, подача високодеталізованого набору вхідних даних в алгоритм провокує ризик занадто високого рівня кластеризації. Це може призвести до того, що буде занадто багато кластерів для перевірки на збіги, що знизить якість взаємодоповнюваності, оскільки додаткові характеристики можуть розглядатися як шум. Отже, поріг для визначення взаємодоповнюваності потрібно знизити.

Нормалізація за **Z-Score** алгоритмом розв'язує проблему різних амплітуд, зберігаючи при цьому притаманні патерни продуктивності. Цей підхід є більш придатним, ніж мінімаксна нормалізація, яка забезпечує однаковий масштаб для всіх ознак, але погано справляється з викидами – сильно віддаленими точками [86]. У нашому сценарії можуть існувати викиди – дуже віддалені значення, тому слід обрати алгоритм для виявлення основних закономірностей шаблону ряду, навіть ігноруючи масштаб, до якого можна повернутись на більш пізній стадії. Отже, нормалізація за критерієм min-max не підходить у цьому контексті.

Нормалізація за допомогою Z-Score алгоритму дозволяє масштабувати дані до загальної шкали, що особливо корисно для порівняння векторів і візуалізації відмінностей у поведінці. Це робиться за допомогою середнього значення та стандартного відхилення елементів вектора. Формула для нормалізації вектора має вигляд:

$$Z_i = \frac{X_i - \mu}{\sigma}, \quad (3.11)$$

де:

Z_i – нормалізоване значення для i -го елемента.

X_i – це i -й елемент вихідного вектора.

μ – середнє арифметичне елементів вектора.

σ – середньоквадратичне відхилення елементів вектора. Це міра кількості варіації або дисперсії в наборі значень.

Середнє значення μ та стандартне відхилення σ для вектора X обчислюються як:

$$\mu = \frac{1}{N} \sum_{i=1}^N X_i \quad \sigma = \sqrt{\frac{1}{N} \sum_{i=1}^N (X_i - \mu)^2}, \quad (3.12), (3.13)$$

де N – кількість елементів у векторі [10], [70].

Підсумовуючи, нормалізація виконується через Z-Score алгоритм [87] з метою пошуку класів еквівалентності. Ідея полягає в записі статистики використання мікропослугою кожного ресурсу у проміжку, який приблизно рівний $[-2; 2]$. Середні значення показників будуть лежати близько нуля, менші за середні – у від’ємній площині, а більші за середні — у додатній. Якщо показник дуже виділяється від інших, може бути сенс розділити цей мікросервіс на два чи більше, щоб створювати виділений екземпляр на екстремальний час для обробки підвищеної кількості запитів. В разі проблем підбору доповнювальної мікропослуги, це буде зроблено на пізньому етапі алгоритму [60].

3.5.2.2. Крок 6: Візуалізація шаблонів вжитку ресурсу задачами

Як опційний крок можна розглядати візуалізацію результату порівняння схожості зразків роботи мікросервісів прораховану за допомогою **FastDTW**, який дає можливість побачити різницю між векторами наклавши їх [88] в оптимальному місці [8], [70].

Для підрахунку сили подібності між векторами та її візуалізації можна використовувати алгоритм **Dynamic Time Warping (DTW)**. Він призначений для пошуку оптимального вирівнювання між двома часовими

рядами, що робить його корисним для визначення схожості часових рядів, класифікації та виявлення відповідних областей між рядами. Однак DTW має квадратичну просторово-часову складність, що обмежує його застосування до невеликих наборів даних. FastDTW, наближення до DTW, усуває це обмеження, забезпечуючи лінійну просторово-часову складність. FastDTW використовує багаторівневий підхід, рекурсивно проєктуючи рішення з грубої роздільної здатності та уточнюючи його, що дозволяє ефективно обробляти великі набори даних часових рядів [88].

Мета полягає в тому, щоб знайти шлях, який мінімізує загальну кумулятивну відстань. Ось ключові кроки алгоритму FastDTW:

1. Огрублення — зменшення роздільної здатності послідовностей шляхом зменшення вибірки. Це передбачає створення версії послідовностей з нижчою роздільною здатністю. Наприклад, береться кожне друге число у векторі.
2. Рекурсивний розрахунок шляху з грубою роздільною здатністю. Цей крок знаходить приблизний шлях з меншою роздільною здатністю.
3. Проекція шляху від огрубленої роздільної здатності назад до початкової роздільної здатності.
4. Уточнення контуру з вищою роздільною здатністю за допомогою локального пошуку навколо спроектованого контуру. Це передбачає коригування шляху для подальшої мінімізації відстані.

Відстань DTW D між двома послідовностями X та Y обчислюється за допомогою:

$$D(X, Y) = \min \left\{ \sum_{k=1}^K d(x_{i_k}, y_{j_k}) \right\}, \quad (3.14)$$

де

- $d(x_{i_k}, y_{j_k})$ відстань між точками x_{i_k} і y_{j_k} . Вона може бути виміряна за евклідовим алгоритмом або іншим обраним способом.

- K довжина шляху викривлення.

Щоб продемонструвати, як можна порахувати схожість між

статистичними даними з двох мікропроцесорів з часовими рядами, і що використання Z-Score нормування дозволить вирівняти значення, навіть якщо вони сильно відрізняються, але знайти закономірність, нижче наведено графік, отриманий після обробки наборів даних. Різниця в їхніх амплітудах становить два рази. Але час перебування в стані більшої та меншої активності схожий. Зображення на Рисунку 3.3 ілюструє приклад.

Приклад:

1: [10, 8, 15, 21, 25, 17, 13, 15, 11, 7, 12, 8, 7, 6, 5, 10, 12, 13, 22, 25, 15, 12, 14, 11]

2: [20, 24, 30, 40, 50, 36, 26, 24, 22, 20, 18, 16, 14, 12, 10, 20, 24, 30, 40, 50, 36, 26, 24, 22]

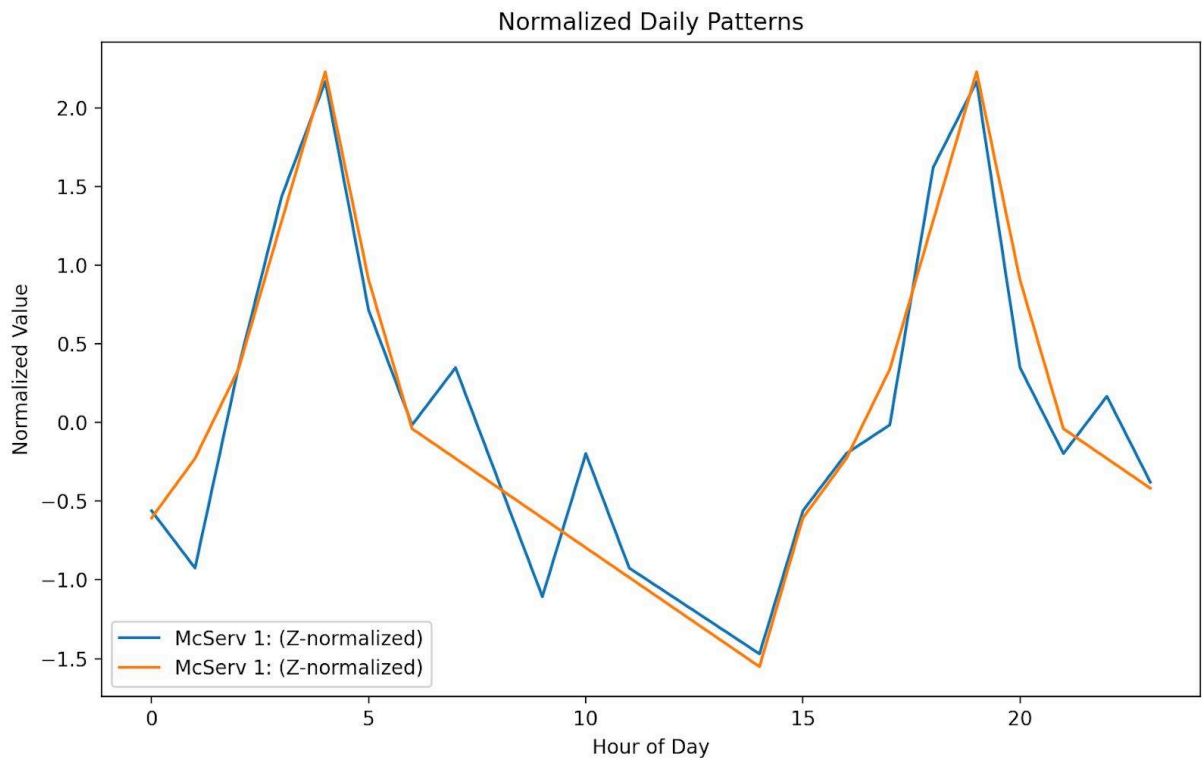


Рисунок 3.3 — Нормалізація + FastDTW для часових рядів, початкові амплітуди яких відрізняються в два рази

3.5.3. Етап 3: Кластеризація шаблонів роботи мікропослуг за ключовим критерієм. Вибір між KMeans і DBSCAN алгоритмами

Для групування мікросервісів за схожою поведінкою можна

використовувати алгоритми кластеризації, такі як **KMeans** [89] та **DBSCAN** (Density-Based Spatial Clustering of Applications with Noise) [90], [91]. Об'єкти в одній групі, яка називається кластером, більш схожі один на одного, ніж на об'єкти в інших групах. **KMeans** і **DBSCAN** — це два широко використовувані алгоритми кластеризації, кожен з яких має свої особливості. У цьому розділі наведено детальне порівняння цих двох методів, зосереджено увагу на їхніх основних принципах, перевагах, обмеженнях та варіантах використання.

Алгоритм **DBSCAN** ідентифікує кластери на основі щільності точок. Він може знаходити кластери довільної форми та працювати з шумом. У цьому завданні очікується, що точки даних утворюють переважно круглі кластери, тому кластеризація **K-Means** є більш відповідним вибором, ніж **DBSCAN**.

3.5.3.1. Вибір між **KMeans**, **DBSCAN** алгоритмами та косинусоїдальною подібністю

Далі описано алгоритми та показано, чому **K-Means** [89] краще підходить для цього сценарію:

1. Очікується, що точки даних знаходяться в радіусі кола. **K-Means** добре виявляє сферичні або кругові кластери, що добре узгоджується з очікуваним розподілом даних. **DBSCAN** [90], [91], хоча і здатний виявляти кластери довільної форми, може надмірно ускладнити аналіз для круглих кластерів.
2. Для подальшого пошуку компліментів потрібні центроїдні точки як база, для якої буде шукатися комплімент. **K-Means** видає явні центроїди кластерів, що є цінним.
3. Обробка контурів не передбачається, оскільки вони мають бути відфільтровані на попередньому етапі. Здатність **DBSCAN** ідентифікувати точки шуму може бути корисною в деяких сценаріях, однак, завдання вимагає включення точок контурів до кластерів як вони є. **K-Means**

природним чином об'єднує всі точки в кластери, що є корисним, оскільки контури не слід розглядати як шум. Цей крок попередньої обробки підвищує ефективність K-Means для даної задачі.

4. При ініціалізації нової комплементарної структури, що відповідає ідеї, описаній у статті, може виникнути потреба в обробці великої кількості мікросервісів. Ось чому обчислювальна ефективність є важливим моментом, який слід враховувати. K-Means зазвичай пропонує кращу масштабованість для великих наборів даних. У сценаріях, де швидкість обробки має вирішальне значення, K-Means може забезпечити швидші результати, особливо якщо кількість кластерів відома або може бути оцінена.

KMeans об'єднує дані в групи, мінімізуючи дисперсію всередині кожного кластера. Цільова функція полягає в тому, щоб знайти аргумент — центроїд, для якого функція відстані до точки матиме мінімальне значення. Іншими словами, найближчий центроїд повинен притягувати точку до свого кластера:

$$\arg \min_S \sum_{i=1}^k \sum_{x \in S_i} \|x - \mu_i\|^2, \quad (3.15)$$

де:

- S множина кластерів.
- μ_i центроїд кластера i .

Алгоритм ітераційно оновлює центроїди та перепризначає точки до найближчих центроїдів до збіжності.

Існує також інший метод, який можна розглянути — **косинусоїдальна подібність**. Він вимірює косинус кута $\cos(\theta)$ між двома векторами, показуючи, наскільки вони схожі, незалежно від величини. Цей метод працював би добре, якби вектори були багатовимірними в реальності. Вектори в задачі насправді є часовими рядами. Ангели в різних частинах одного часового ряду є суперечливими. Метод косинусної подібності

працює скоріше з лінією, а не з кривою, тому не підходить у цьому випадку.

3.5.3.2. Крок 7: Пошук подібних шаблонів роботи за ключовим критерієм.

Наступним кроком є пошук схожих шаблонів роботи ПЗ. Основною задачею пошуку схожості шаблонів є виділення класів подібних за стилем роботи мікросервісів користуючись ключовим (найбільш витратим) критерієм. Здебільшого це CPU, а каналний ресурс та RAM перевіряються додатково як ті, що не накладатимуться. Якщо ж для частини мікросервісів ключовий критерій інший, то їх слід обробляти в окремо за тим же самим принципом.

Для пошуку схожості шаблонів використовується алгоритм кластеризації KMeans. За цим алгоритмом формуються центроїди – точки, які представляють центри груп мікросервісів схожої поведінки. Оскільки статистика використання мікросервісу включає значення за певною характеристикою за цілий день, то і точка-центроїд теж являє собою часовий ряд значень обраної характеристики.

3.5.3.3. Крок 8: Двовимірна візуалізація кластерів часових рядів

Щоб зробити приклад наближеним до реального, для кожного типу п'яти поведінок було згенеровано 24 мікропослуги, які виконують один одного. Поведінки з кількістю згенерованих мікропроцесорів представлені нижче:

1. 2 мікросервіси зі стабільним навантаженням для.
2. 4 мікросервіси з піковим навантаженням у робочі години (з 9 ранку до 5 вечора).
3. 2 мікросервіси з піками після опівночі та до 6 ранку.
4. 2 мікросервіси з піками з 17:00 до ночі.
5. 2 мікросервіси з випадковими піками.
6. Доповнення для кожного з цих шаблонів (завантаження в протилежний час).

Вищезгадані мікросервіси пройшли наступні етапи:

1. Підготовка даних шляхом нормалізації даних мікросервісів за допомогою методу Z-Score. Ці дані мають 24 виміри (по одному для кожної години доби). Приклад можна побачити на Рисунок 3.5.

2. Кластеризація за допомогою алгоритму KMeans. Для векторів та центроїдів було використано аналіз головних компонент (PCA) для отримання чіткого та зрозумілого зображення [92]. Алгоритм використовується лише для графічного представлення для кращого розуміння якості роботи алгоритму KMeans, тому в роботі він не описується. Більш детальну інформацію про нього можна знайти тут [93], [94]. Він зводить 24-вимірні дані до 2-вимірних, що дозволяє візуалізувати багатовимірні дані у вигляді двовимірного графіка. Проте, слід зазначити, що частина інформації неминуче втрачається в процесі зменшення розмірності.

На діаграмі розсіювання на Рисунок 3.4 кожна точка представляє мікропослугу. Центроїди позначені великими маркерами «X». Точки одного кольору утворюють кластер. Координати x та y кожної точки є першою та другою головними компонентами відповідно. Ця візуалізація дозволяє спостерігати, наскільки добре розділені кластери та чи є в даних чіткі угруповання.

На Рисунок 3.5 показано 24-D вектори, розділені на кластери, позначені певними кольорами. Пунктирні лінії — це центроїдні вектори. Симетрія зображення пояснюється методом комплементарної генерації векторів. Додавання шуму змінює картину. Для більш наочного і зрозумілого прикладу показано «ідеальний» випадок.

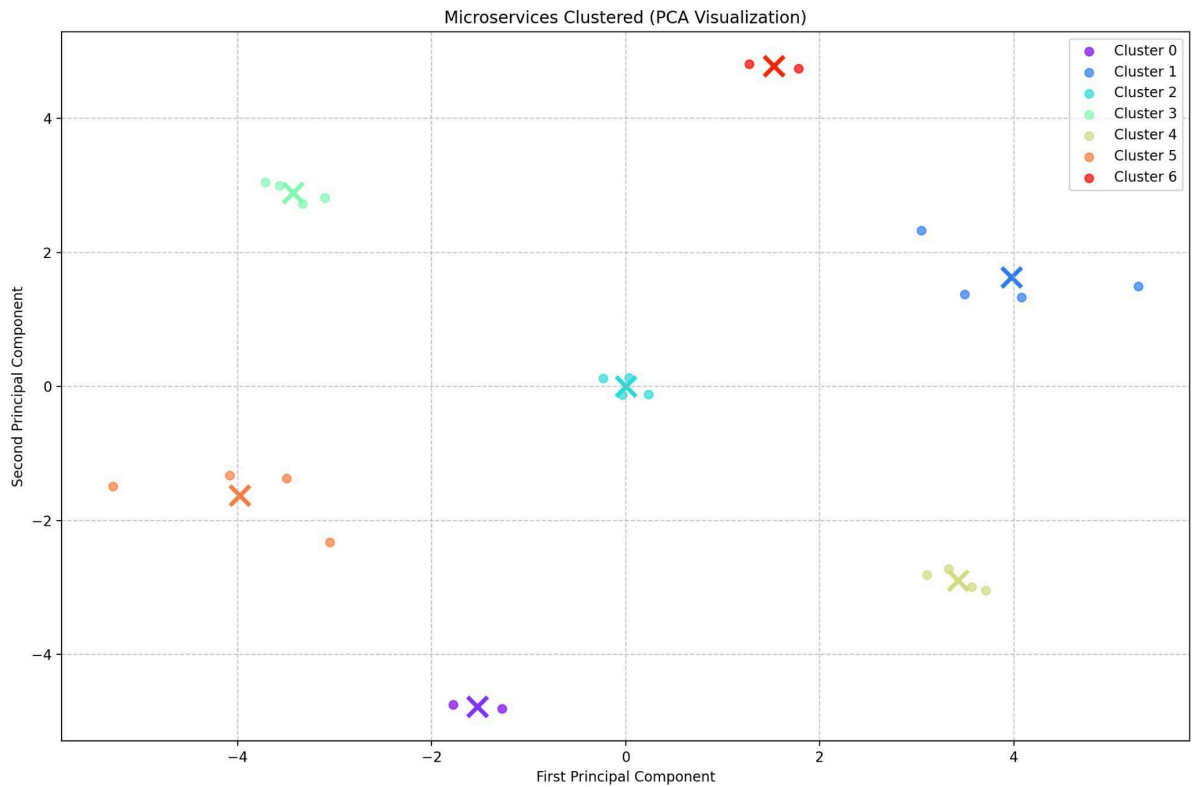


Рисунок 3.4 — Візуалізація 2D векторів та центроїдів за алгоритмом KMeans після аналізу головних компонент. X — центроїди, точка — 24-D вектори, перетворені у 2-D вектори за допомогою PCA. Колір вказує на приналежність до певних центроїдів

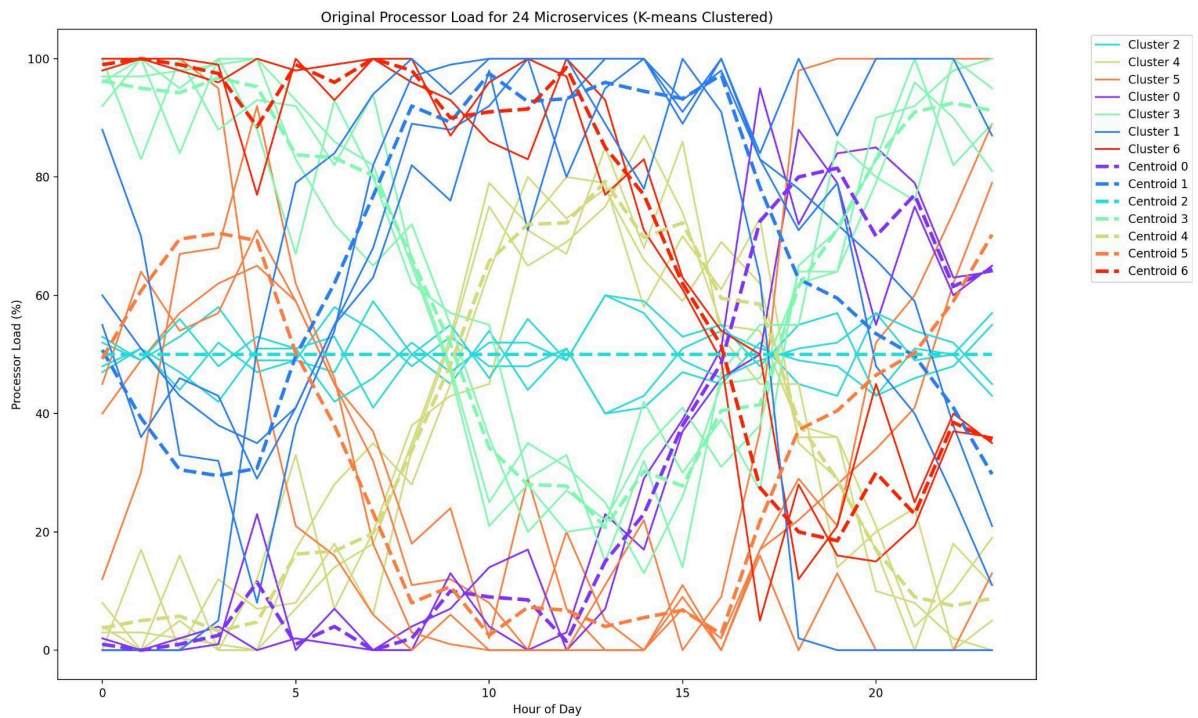


Рисунок 3.5 — 24-D вектори розділені на кластери, які можна побачити за кольором. Пунктирні лінії – це центроїдні вектори. Симетрія зображення пояснюється методом комплементарної генерації векторів

3.5.4. Етап 4: Пошук доповнювальних груп та пар процесорів

Ідея пошуку компліментарних груп походить з теорії ґраток. Більше інформації на цю тему можна прочитати в [7]. Знайшовши кластери подібності, буде швидше знайти доповнення, що відповідають основним характеристикам кластерів, замість повного перебору всіх векторів мікросервісу. Для цього можна використати кілька методів, які описані в наступному підрозділі.

3.5.4.1. Крок 9: Пошук доповнювальних (комплементарних) груп

Кульмінаційним моментом алгоритму є пошук доповнювальних груп мікросервісів та подальший пошук доповнювальних мікросервісів, описаний в кроці 10. В цьому кроці пропонуються два способи пошуку доповнювальних груп.

1. Спочатку необхідно знайти кластери, які містять потенційно комплементарні вектори. Для кожної пари кластерів, включаючи себе, обчислюється розбіжність між їхніми центроїдами, щоб визначити, чи є вони комплементарними. Поєднання з самим собою необхідне для випадку рівномірного навантаження протягом усього періоду часу. При пошуку доповнювальних груп вираховується показник розбіжності центроїдів $D_{c\ i,j}$ або $D_{centroid}$ за ключовим параметром, як нормована різниця між максимальним завантаженням релевантної серверної групи $sum_{maxi,j}$ та сумою навантажень, яке зроблять обидва центроїди sum_{max} . Формула для обчислення розбіжності центроїдів $D_{c\ i,j}$ має вигляд

$$D_{c\ i,j} = \frac{1}{n} \sum_{i=1}^n \left(\frac{sum_{maxi,j} - (vector_i + vector_j)}{sum_{maxi,j}} \right), \quad (3.16)$$

- де n кількість параметрів у векторах $vector1_i$ та $vector2_i$,
- для суми беруться всі i -ті параметри двох векторів,

Якщо $D_{centroid} \leq 30\%$, вектори, а отже і групи вважаються комплементарними, якщо прийняти 30% як поріг розбіжності. Це змінний

поріг, який можна налаштовувати.

$$D_{c_{i,j}} = \frac{1}{n} \sum_{i=1}^n \left(\frac{\text{sum}_{\max i,j} - (\text{vector}_i + \text{vector}_j)}{\text{sum}_{\max i,j}} \right).$$

Якщо результівне значення знаходиться в межах заданої похибки $D_{c_{i,j}} \leq D_{c_{\max}}$, то групи вважаються доповнювальними. Результатом цього кроку є список проранжованих доповнень до кожного кластеру.

2. Іншим або додатковим способом визначення комплементарності кластерів є застосування алгоритму KMeans для визначення того, чи може комплементарний центроїд, розрахований як максимальне значення мінус вектор центроїда, бути частиною кожного іншого кластера. Якщо може – групи доповнювальні. Оскільки перший спосіб дає цифрове значення розбіжності, що зручніше, він був обраний для практичних розрахунків.

3. Результати перших двох перевірок зберігаються в масиві даних під назвою *complementation_stats*. Цей масив буде використано для пошуку точних збігів мікросервісів в кроці 10.

3.5.4.2. Крок 10: Пошук доповнювальних пар мікросервісів

При здійсненні пошуку доповнених мікросервісів використовуються початкові метрики кожного екземпляру, а не нормовані.

1. З двох найбільш доповнювальних груп береться по одному мікросервісу i та j з найближчою різницею в амплітуді. Для того, щоб не перевіряти заздалегідь неробочі варіанти, можна порівняти різницю хвилі потужності, вираховану як різниця потужностей у час максимуму t_{\max} та час мінімуму t_{\min} , та переконатись, що вона знаходиться в певній умовою заданій межі $\Delta_{\text{similarity}}$:

$$|(P_{i,t_{\max}} - P_{i,t_{\min}}) - (P_{j,t_{\max}} - P_{j,t_{\min}})| \leq \Delta_{\text{similarity}}. \quad (3.17)$$

Мікросервіси накладаються як на Рисунку 3.6 — ідеальний варіант, тільки безпечний надлишковий діапазон додаватиметься вже перед

виділенням заданого місця на хостингу. На практиці хороша картина виглядає як 3.7.

2. Потім робиться оцінка сумарної невідповідності (Discrepancy) $D_{i,j}$ за кожним показником, тобто кількості недовикористаних ресурсів. Якщо цей показник менший за граничний, заданий у вхідній умові, за ключовим показником, напр., D_{CPU} — невідповідність, дозволена для CPU, а також нема перетинів за іншими показниками (канальний ресурс та RAM) то вибрані мікросервіси формують групу та записуються в масив з назвою `'complimentary_microservices'`. Вони видаляються з груп пошуку доповнюваних мікросервісів, адже їм вже не треба шукати іншу пару.

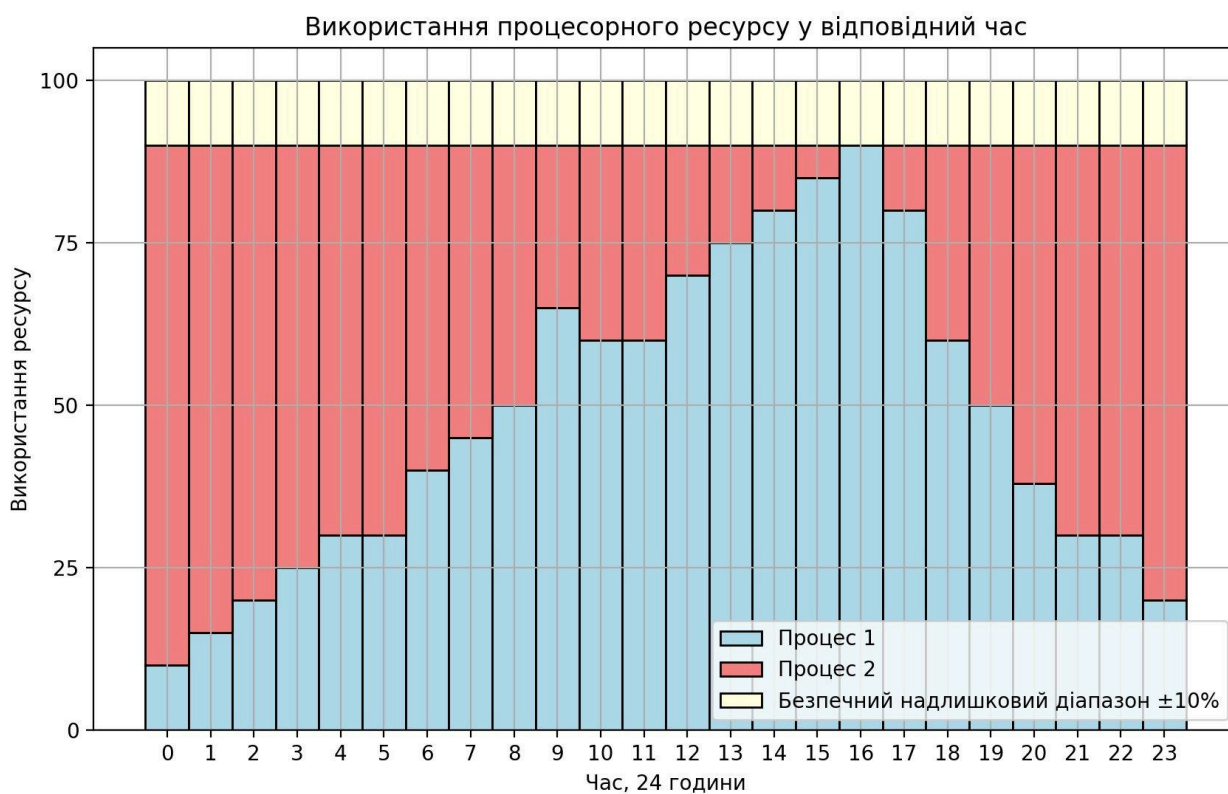


Рисунок 3.5 — Доповнювальна група з двох мікропослуг (знизу) та доданий безпечний інтервал (зверху), ідеалістична модель

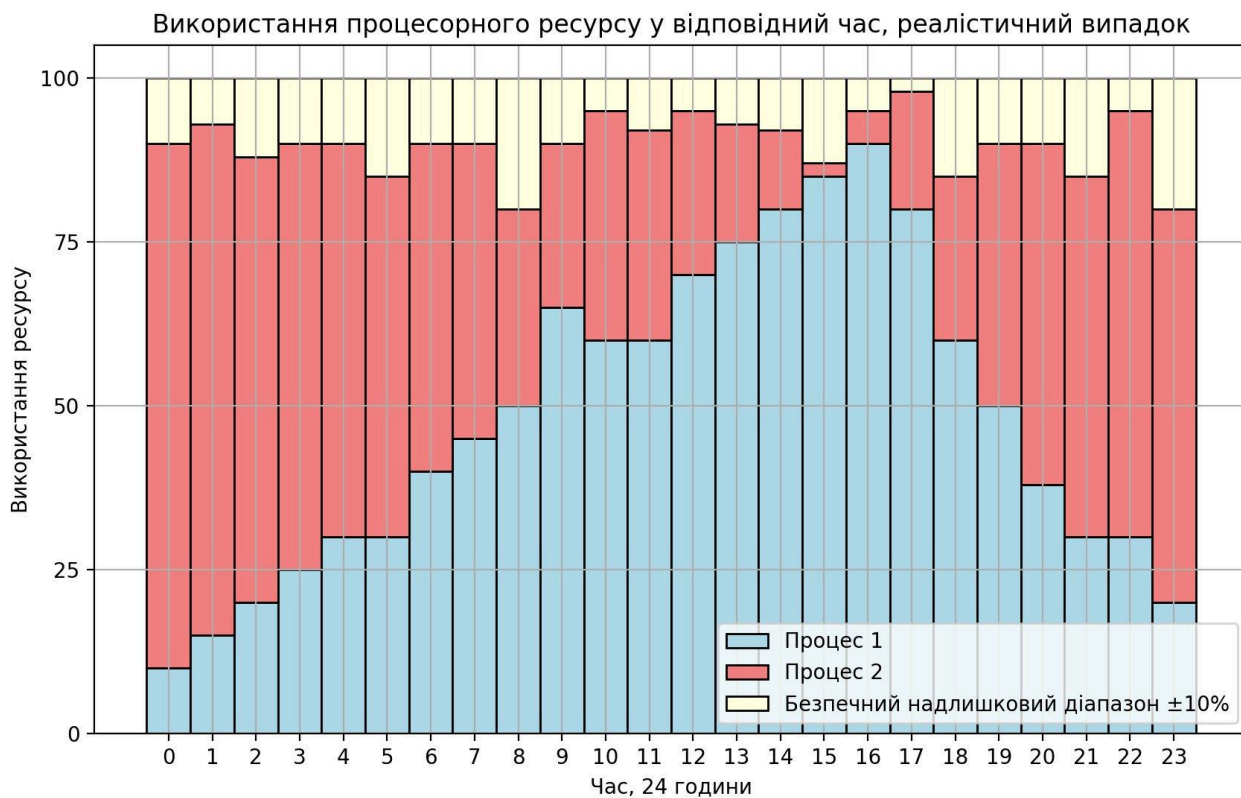


Рисунок 3.7 — Доповнювальна група з двох мікропослуг (знизу) та доданий безпечний інтервал (зверху), реалістична модель

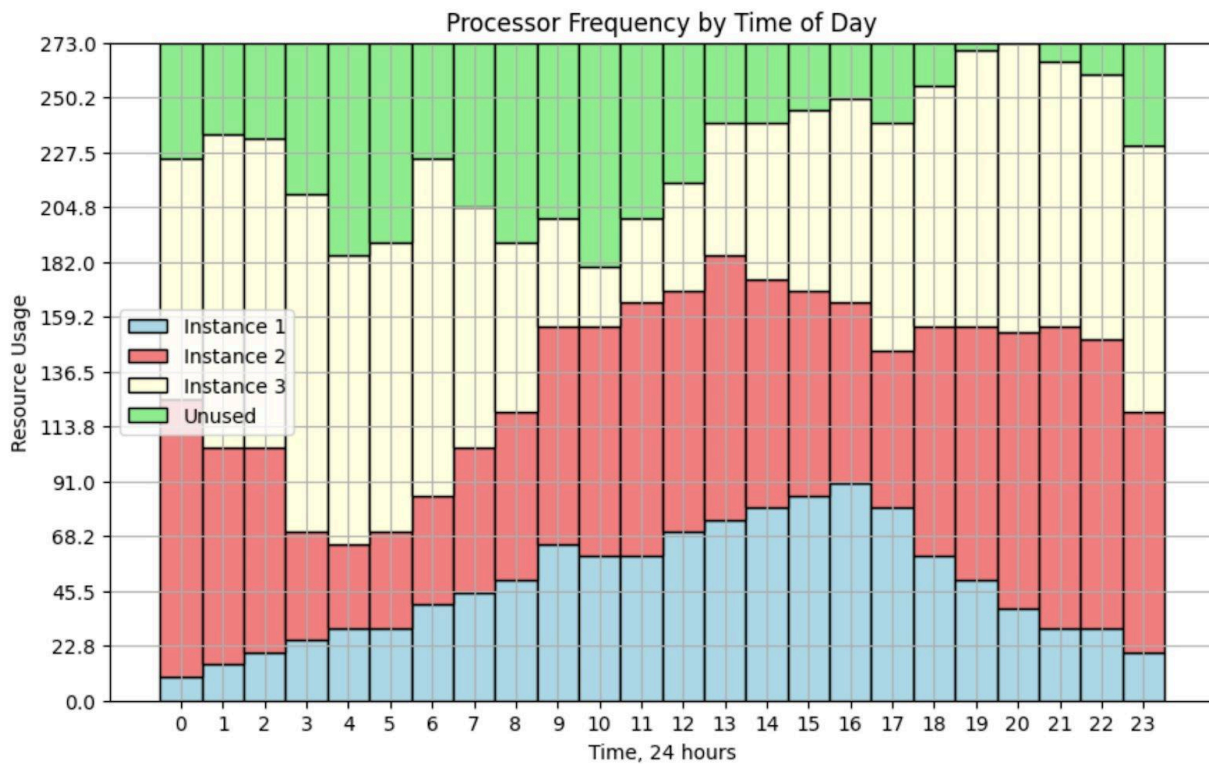


Рисунок 3.8 — Скомбіновано три мікросервіси (знизу). Зверху — ресурс, котрий пустий

3. Під їх комбінацію буде виділено серверну групу, загальний ресурс якої $P_{\text{group } g}$ рівний їх максимальному ресурсу $P_{\text{max } g}$ з урахуванням заданої початковою умовою похибки α_{reserve} (alpha) — відсоток запасу ресурсу. Числове значення обчислюється як відсоток від сумарного найбільшого завантаження в одиницю часу t . У наступній формулі i та j — індекси мікропроцесорів, які входять в групу. Ця формула валідна для довільної кількості елементів в групі g .

$$P_{\text{max } g} = \max \left(\sum_{g \in i,j} P_g(t) \right), \quad (3.18)$$

$$P_{\text{group } g} = P_{\text{max } g} + \alpha_{\text{reserve}} \cdot P_{\text{max } g}. \quad (3.19)$$

Якщо ж невідповідність $D_{i,j}$ більша максимальної дозволеної $D_{g \text{ max}}$, обидва мікросервіси продовжують шукати собі пару помічаючи пройдені мікросервіси як непідхожі.

Пункти 1–3 повторюються для всіх активних пар кластерів у *complementation_stats*, доки не буде знайдено всі можливі збіги.

Хоча сортування на цьому етапі не є обов'язковим, може бути корисно відсортувати пари за значенням розбіжностей та за справжніми амплітудами. Це дозволить швидше знаходити більш підхожу пару центроїдів та провести менше ітерацій пошуку у випадку наявності пар в масивах.

Водночас сортування може виявитись неоптимальним рішенням на цьому етапі, адже під час роботи алгоритму розмір масиву зменшуватиметься через видалення або позначення порожніми деяких елементів, які ставатимуть частинами інших пар. Тобто час та складність сортування може бути більшою, ніж користь в разі дуже великих груп, зокрема, якщо збіги з цією групою будуть малоймовірні. Тому перед вибором, чи треба сортування слід звертати увагу на співвідношення кількості груп та елементів, а також на номер ітерації алгоритму. В останні

ітерації при дуже малих групах та маленькій ймовірності знаходження пар сортування буде зайвим.

Якщо використовується сортування початкової групи мікропослуг, пошук найкраще здійснювати користуючись алгоритмом бінарного пошуку.

Якщо в межах найкращої доповнювальної групи не було знайдено пару, слід взяти наступну за рейтингом підхожу групу та шукати пару там. Процес повторюється, поки група, для мікросервісів якої шукали пари, стане пустою, або ж поки не будуть проаналізовані всі доповнювальні групи.

Мікропроцесори, котрі не знайшли пари на цій ітерації, записуються в окремий масив тих, які підуть на наступну ітерацію.

Продовжуючи приклад, наведений в [Крок 8: Двовимірна візуалізація кластерів часових рядів](#), результати зіставлення кластерів будуть наступні. Позначка «Активний: Ні» означає, що один або обидва кластери містять нульові елементи після зіставлення:

Кластери 3 і 4:

Розбіжність: 2,53

Центроїдний збіг: Так

Активний: Ні

Кластери 1 і 3:

Розбіжність: 16,44

Центроїдний збіг: Так.

Активний: Ні

Детальніше про практичну реалізацію можна прочитати у наступному розділі.

3.5.4.3. Крок 11. Повторення ітерацій пошуку

Якщо не всі мікросервіси знайшли свої відповідності з першої ітерації, слід повторити кроки 7–10, тобто починаючи з кластеризації за допомогою методу KMeans. Ітерації повторюються, поки утворюються

пари. З кожною ітерацією кількість груп у пункті 7, на які діляться мікросервіси, зменшується пропорційно кількості екземплярів.

3.5.4.4. Вибір кількості кластерів

Стосовно вибору кількості кластерів, існують тільки емпіричні методи, такі як метод ліктя, силуетний аналіз, Гар-статистика та правило \sqrt{n} . Останній метод доцільно використовувати при вибірці меншій ніж 8^2 , а при більшій вибірці брати завжди 8 або 7 за умови регіонального розподілу мікросервісів. Останнє важливо, оскільки в інших країнах при зсуві часу робочі години можуть бути інші, тому аби коректніше знаходити доповнення, можна виділити більше груп кластеризації. Якщо у групі лишатиметься менше ніж 2–3 елементи, нема сенсу створювати групу для такої малої кількості.

Кількість кластерів може бути обрана як кількість можливих груп мікросервісів, описаних в прикладі в [Крок 8: Двовимірна візуалізація кластерів часових рядів](#) можна помножити на 2, щоб краще розділити набори та точніше підібрати доповнення на великій вибірці. Загалом вийде 10, але 7–8 кластерів теж дають хороший підбір. Основні групи наступні:

1. робота в денний час з 9–17,
2. робота в нічний час 23–6,
3. робота у вечірні години: 17–24,
4. робота у ранкові години: 6–11,
5. стабільне завантаження протягом доби.

Якщо мікропослуг не так багато, більше 27, то номер кластера можна вибрати як кількість мікропослуг, поділену на 3 плюс 1, як у (3.20).

$$\text{Cluster Number} = \left\lceil \frac{\text{Number of Microservices}}{4} \right\rceil + 1 \leq 10, \quad (3.20)$$

де $\lceil x \rceil$ позначає функцію стелі, яка округлюється до найближчого цілого.

3.5.4.5. Крок 12: Доповнення неможливі

Можуть бути випадки, що не всі мікросервіси знайшли доповнювальні навіть після 11 кроку. Це може статись в наступних випадках:

1) характер роботи мікропослуги випадковий. В такому разі йому слід виділити окремий ОР та не намагатись його сумістити з іншими (див. етап 7);

2) екстремуми роботи мікросервіса дуже різкі, або ж в нестандартний час. В такому випадку пропонується розділити цей мікросервіс на декілька, якщо виражених екстремумів не більше двох, що описано в кроці 13. Інакше — виділити окремий простір під таку мікропослугу (див. етап 7).

3) мікропослуги замалі, тому сумарне доповнення з іншими не дає повної завантаженості. Рішення — об'єднувати більш як два мікросервіси. Виконавши об'єднання з метою досягнення рівномірності, можна прогнати алгоритм починаючи з кластеризації знову.

Підсумовуючи всі розв'язки вищезгаданих випадків виходить, що для групи мікросервісів будуть виділені сервери, а інші випадки слід звести до стандартних мікросервісів, котрі можуть знайти собі доповнення. Також, зважаючи на малий розмір шматочків, слід користуватись іншим алгоритмом їх збору (кроку 15).

3.5.5. Етап 5: Розділення мікропослуг на частини при передбачуваних різких змінах в навантаженні

3.5.5.1. Крок 13: Розділення базового навантаження та екстремумів для не згрупованих особин

Розглянемо випадок 12.2 з минулого кроку, коли існують мікросервіси з не дуже великим навантаженням, меншим за $P_{boundary} - P_{boundary} \times \alpha_{boundary}$, але все ж які залишились без пари. Вірогідно,

екстремуми трапляються в нестандартний час, або ж кількість денних завантажень переважає нічні завантаження, тому пари не існує. Все ж є вірогідність знайти пару, якщо «згладити» піки роботи такого мікросервісу, якщо їх небагато.

Для визначення стандартного навантаження та піків часового ряду можна проаналізувати значення середнього та стандартного відхилення [95], що допоможе виявити екстремуми. Визначення середнього значення часового ряду відбувається за формулою (3.12) та стандартного відхилення — за формулою (3.13).

Екстремуми визначаються як значення, що перевищують певну кількість стандартних відхилень від середнього. В дані задачі одного стандартного відхилення вже достатньо для такої оцінки, тобто, якщо значення виходять за межі $\mu \pm \sigma$, їх можна вважати екстремумами (аномаліями). Нас цікавлять тільки пікові значення, бо їх можна «зрізати», виділивши під них окремий мікросервіс. Для монолітів дана практика схожа: доведеться створювати дублювальний моноліт у вибраний час та виконувати розподіл навантаження (load balancing) між двома екземплярами.

Зрізати має сенс всі значення, котрі ідуть підряд і перевищують середнє. Для наочності наведено приклад. На Рисунку 3.9 показаний часовий ряд з вираженим піком, через який важко знайти доповнювальний мікросервіс. Цей пік слід зрізати, тоді графік буде рівніший, знайти доповнення буде імовірніше.



Рисунок 3.9 — Виділення навантаження мікропроцесора, яке слід запустити на іншому сервері. Стосовно моноліта — виділення часу його додаткового запуску

За вищезгаданим алгоритмом знайдуться пікові точки, що лежать вище верхньої жовтої лінії (з крапочок). Ці пікові значення позначені зеленими вертикальними лініями з крапочок. Екстремуми нижчі нижньої жовтої лінії не розглядаються. Купол графіку спостерігається довший час, ніж у верхньому квадранті оточеному зеленими та жовтою лініями з крапочок. Є сенс розширити цей проміжок в тому випадку, якщо він лежить вище середнього навантаження позначеного голубою лінією «крапка-тире». Для цього алгоритм проходить по всіх ближніх точках зліва та справа піка, та підбирає ті, що більші середнього значення.

Після проводиться пошук максимальної точки у часовому ряді, який не включає пік та близькі точки. Ця точка і буде навантаженням поділу мікросервісу. Наступним кроком відкидаються всі потенційно-пікові точки, які лежать між жовтою та голубою лініями, які є менші за обраний новий максимум. Останній позначений червоною горизонтальною лінією «тире». Розширений екстремум позначений синіми вертикальними лініями «тире».

На Рисунку 3.10 показано поділ мікросервісу на дві частини: до нової максимальної лінії та вище за неї.

Говорячи про дії клауд-провайдера, має сенс заздалегідь запустити пікове навантаження на окремому сервері чи серверній групі, ввівши його в стан гарячого очікування (hot stand-by) [96].

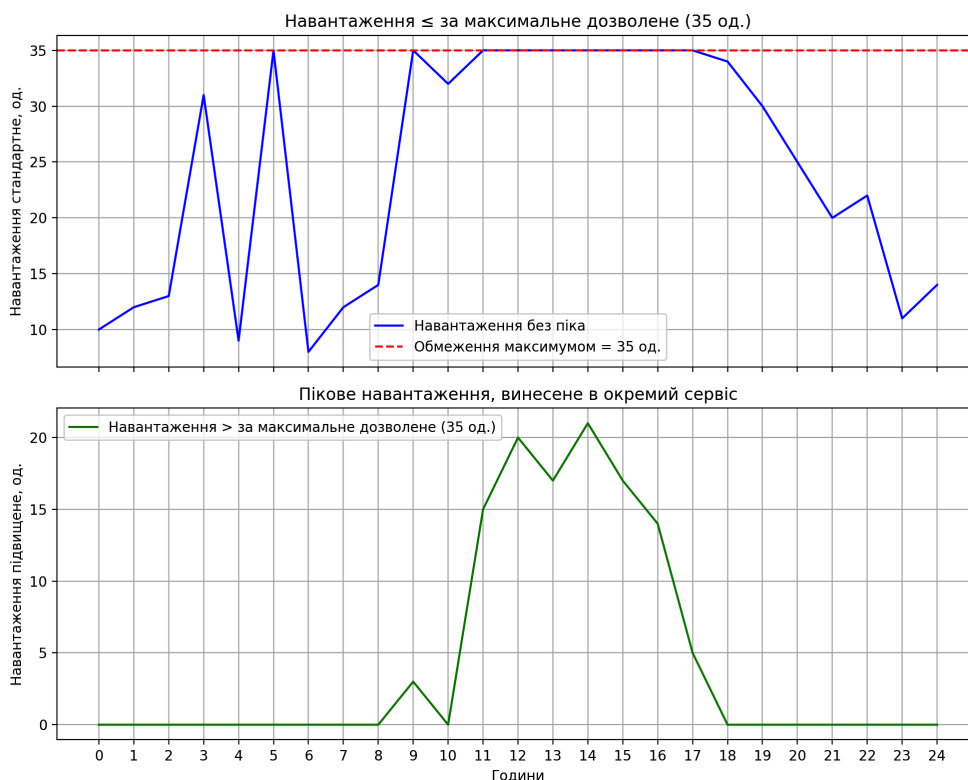


Рисунок 3.10 — Розділення мікросервісу на дві частини, виділяючи екстремум в окремий екземпляр.

3.5.5.2. Крок 14: Пошук доповнень для розділених мікропроцесорів

Оскільки мікропослуги змінились з часу останнього пошуку доповнень, слід повторити крок 11.

3.5.6. Етап 6: Комбінування малогабаритних мікропослуг у потенційно великі набори

3.5.6.1. Крок 15: Комбінація малогабаритних мікропослуг

Для решти мікросервісів є два наступні кроки, перший можна опустити:

1. оскільки розмір мікросервісів малий, можна кластеризувати всі мікропослуги через KMeans, об'єднати по 2–3 мікропослуги з груп

неперетину, у котрих різні доповнювальні групи, після чого прогнати нові групи по кроку 11;

2. решту мікросервісів, які залишаться, об'єднати користуючись алгоритмом упаковки множинних рюкзаків із гнучким обмеженням цільової ваги [97].

Таким чином, всі мікропроцесори будуть згруповані, щоб обчислювальні потужності максимально не простоювали, а також щоб можна було підготувати потрібну машину для запуску обчислень на ній аби зменшити час розігріву, ввівши її в стан гарячого очікування [96].

3.5.7. Етап 7: Виділення серверних потужностей під розміри групи та моніторинг

Виділення місця під групу описане у кроці 10.3. Додатково є потреба в моніторингу за станом системи та за коректністю розподілу задач. В разі, якщо певна група часто перевищує або недобирає своєї норми, слід розглянути включені мікропослуги знову з метою знайти інші доповнення, користуючись оновленими історичними даними.

3.5.7.1. Крок 16: Налаштування під час роботи ПЗ на стороні провайдера.

Після запуску система потребує продовження моніторингу. На цій стадії потрібно слідкувати за оновленням метрик та за якістю розподілу навантажень. Якщо виявляється, що протягом певного часу x серверна група використовує замалу кількість ресурсів, чи навпаки, потребує більше, то такі групи слід виокремити, скомбінувати з новими мікросервісами за нагоди та виконати пошук нових доповнень.

ВИСНОВКИ

В розділі розглянуто різні випадки, коли пошук доповнювальних навантажень буде виправданим, описано метод та алгоритм доповнювальних навантажень.

1. Ідея доповнювальних навантажень можлива у зв'язку з різноманіттям обчислювальних задач, які вимагають обслуговування в різний час доби.
2. Глобалізація також відіграє суттєву роль, оскільки програмне забезпечення, розташоване на серверах в одній часовій зоні може обслуговувати користувачів у різних частинах світу, що сприяє більш стабільному, а отже ефективному завантаженню серверів протягом доби. Таке завантаження сприяє зменшенню їх кількості, а також збалансованій завантаженості всіх компонентів системи.
3. Обмеженнями при пошуку доповнювальних навантажень можуть виступати норми безпеки та використання спільних ресурсів групами мікропослуг, розміщення яких слід вибирати за цими принципами на противагу вищій ефективності завантаження CPU, RAM та мережі серверів.
4. Запропоновано метод доповнювального розподілу навантажень для визначення оптимального доповнення до базового навантаження, що забезпечує мінімізацію сумарної кількості необхідних ресурсів в багатосерверній інфраструктурі.
5. Описано етапи групування обчислювальних задач, які включають нормування шаблонів роботи мікропослуг та подальшу кластеризацію, на основі якої відбувається пошук доповнювальних кластерів, а потім в їх межах і пошук доповнювальних шаблонів задач з урахуванням справжніх ненормованих амплітуд. В результаті цих дій буде сформовано групи доповнювального навантаження для розподілення на сервери, якими

оперує матмодель, та які забезпечують сумарне однакове довготривале використання ресурсу.

6. Наведена методика нормування отриманих метрик на локальних серверах на сервери хмарної інфраструктури, що забезпечує можливість переходу з локальних серверів, або серверів одного багатосерверного постачальника послуг на сервери іншого.

7. Детально описано алгоритм планувальника навантаження, оснований на пошуку доповнювальних навантажень, який

- аналізує навантаження задач в черзі;
- відділяє задачі, непридатні до групування;
- знаходить схожі шаблони роботи мікропослуг,
- виділяє місце на серверах під визначену пару задач;
- розбиває навантаження, що не знайшли доповнювального, на частини;
- повторює пошук доповнювального навантаження за потреби групуючи маленькі навантаження та комбінуючи більше задач, ніж дві.

Суть алгоритму пошуку доповнювальних навантажень

Щоб скористатись методом пошуку доповнювальних мікропослуг, виконується кластеризація та екземпляри мікросервісів групуються у класи еквівалентності на основі подібності шаблону роботи. В межах груп екземпляри сортуються за амплітудою використання ресурсів, після чого групи, які визначаються як доповнювальні, перетинаються з метою пошуку конкретної пари комплементарних мікросервісів.

В результаті, екземпляри зі значними відмінностями в шаблонах (протилежні шаблони навантаження) комбінуються з іншими, що мають подібні амплітуди, але протилежні фази, для максимального використання ресурсів. Комбінації з екземплярами, що мають низьку амплітуду, також можуть бути доречними для заповнення «дірок».

У доповнювальних групах, алгоритм шукає перший екземпляр мікросервісу, що відповідає умовам доповнення (жадібний алгоритм). На випадок труднощів знайти доповнення в першій найліпшій групі, зберігається статистика комбінацій з усіма екземплярами. Пошук продовжується до знаходження доповнень всім мікросервісам групи або поки не будуть перебрані всі доповнювальні групи.

Таких ітерацій проводиться стільки, скільки будуть знаходитись пари, але «залишки» після пошуку комбінацій є очікуваними. Мікропослуги, що не знайшли доповнень додатково поділяються з використанням розширеного пошуку екстремумів за допомогою пошуку середнього та стандартного відхилення для збільшення їх шансів знайти компліменти. Кластеризація та пошук проганяються до останньої успішної знахідки. Коли лишається невелика кількість мікросервісів, які не знайшли пари, вони комбінуються за допомогою задачі про множинний рюкзак.

Елементи розділу опубліковані в наукових публікаціях [8], [10], [11], [70].

РОЗДІЛ 4

АПРОБАЦІЯ ТА АНАЛІЗ ЕФЕКТИВНОСТІ РОЗРОБЛЕНОГО КОМПЛЕКСНОГО МЕТОДУ ПОШУКУ ДОПОВНЮВАЛЬНИХ НАВАНТАЖЕНЬ

В розділі проводиться аналіз вживаних ресурсів та економії, що може спостерігатись після застосування методу доповнювальних навантажень. Алгоритм запропонованого методу був проєкспериментований, а дані до нього зімітовані. Способи запровадити в системі такий алгоритм наведені в цьому розділі також.

Вибір позиціювання обчислювальної задачі (ОЗ) зазвичай залежить від

- заданої стратегії заповнення вузлів (часткова, рівномірна, максимальна);
- локації користувачів – чим ближче сервер, тим швидше оброблятимуться запити;
- потреб самої задачі щодо апаратного забезпечення та її розміщення відносно інших задач, таких як БД та споріднені інші сервіси.

В роботі беруться до уваги всі вищезгадані параметри, але поняття геолокації (або роботи в денний та нічний час) та стратегії заповнення вузлів розглядаються з погляду забезпечення максимальної економії для хмари, при цьому без погіршення якості послуги.

4.1. Апробація методу пошуку доповнювальних навантажень

Розробка багатокористувацьких або високонавантажених програми, як групи мікросервісів, які можна реплікувати на кількох екземплярах, стає все більш поширеною, адже багато підприємств намагаються зробити свої програмні продукти масштабованими для ефективного управління високими навантаженнями. Для підтримки такої структури потрібна платформа управління мікропослугами. Інструменти контейнеризації та розгортання (Docker та Kubernetes) спільно з різними хмарними платформами пропонують допомогу в управлінні навантаженням та

масштабованістю шляхом розгортання нових екземплярів мікросервісів, які можуть допомогти подолати обмеження одного екземпляра.

У більшості випадків навантаження програм є передбачуваним. Вплив раптових сплесків навантаження має значний ефект на стабільність системи. Основна ідея полягає в розподілі мікросервісів між групами серверів, спочатку враховуючи пріоритети балансування загального навантаження групи, потім передбачуваність та забезпечення того, щоб не було перевищено певний допустимий максимум, який може забезпечити група серверів. Ідею та модель групування навантажень з метою максимізації навантаження кожної групи, а отже зменшення сумарної кількості необхідних ресурсів в ІКС буде описано в цьому розділі.

Задача даного розділу — запропонувати способи реалізації методу пошуку доповнювальних мікросервісів локально з метою реалізації та тестування алгоритму доповнювальних додатків (мікросервісів) та почати його використання для свого проєкту, розгорнутого на хмарній системі чи іншому сервер провайдері.

4.1.1. Kubernetes як технологія для реалізації методу доповнювальних навантажень

Коли мова йде про алгоритм, котрий має виконуватись на хмарному середовищі, та ще й пов'язаний з планувальником, який автоматично налаштований оператором хмари, слід мати ресурси для імітації процесу на локальних серверах, аби його протестувати та налаштувати параметри. На сьогоднішній день по суті єдиним інструментом оркестрації, який може зімітувати хмарну систему є Kubernetes. Це платформа з відкритим вихідним кодом для оркестрації контейнеризованими (через Docker [98]) робочими навантаженнями та супутніми службами (наприклад, трекінг метрик). Вона характеризується кросплатформенністю та розширюваністю, тобто в разі посиленого навантаження, вона може адаптуватись та вирости, аби якісно надати послугу користувачам.

Kubernetes часто асоціюється з хмарними технологіями (cloud) через те, що обидві концепції мають спільну мету — забезпечити масштабованість, автоматизацію та ефективне управління ресурсами для додатків. Kubernetes, як і хмарна система, включає балансування навантаження, розподіл ресурсів і автоматичне відновлення контейнерів після збоїв [1]. Хмарні платформи (AWS, Azure, Google Cloud) також надають ресурси автоматично і вимагають мінімального втручання користувача.

Як і в хмарних середовищах, Kubernetes дозволяє масштабувати застосунки за необхідності, автоматично створюючи нові інстанси (екземпляри) контейнерів. Це схоже на масштабування ресурсів в хмарних інфраструктурах, де можна динамічно додавати або видаляти обчислювальні ресурси на основі попиту.

Хмарні сервіси надають можливість керування різними ресурсами, такими як віртуальні машини (VM), мережеві підключення, сховища і бази даних. Kubernetes забезпечує оркестрацію контейнеризованих додатків, що є аналогом такого управління на рівні додатків.

І хмарні технології, і Kubernetes добре підходять для роботи з мікросервісною архітектурою. Вони дозволяють легко розгортати, масштабувати та підтримувати незалежні сервіси, з яких складається додаток, що є основним підходом у хмарних системах.

Таким чином, імітація роботи алгоритму на Kubernetes є способом протестувати його, а після, і повноцінно користуватись, якщо мова йде про послуги сервер-провайдера, котрий не обслуговує клієнтів на рівні хмарної технології, а лиш надає локальні послуги хостінгу. Щобільше, то багато хмарних платформ використовують Kubernetes як основний механізм для управління контейнерами. Вони надають сервіси, що базуються на Kubernetes для спрощення розгортання та оркестрації контейнеризованими додатками в хмарі. Сам Kubernetes був розроблений у Google, який відкрив доступ до вихідного коду проекту Kubernetes у 2014. Google Kubernetes

Engine (GKE) — сервіс від Google Cloud, побудований на основі Kubernetes. Так само існує Amazon Elastic Kubernetes Service (EKS) — сервіс від AWS для управління Kubernetes-кластерами (імітується за допомогою інструментів kOps або EKS-D) та Azure Kubernetes Service (AKS) — сервіс від Microsoft Azure для розгортання та оркестрації Kubernetes.

Зважаючи на це, головною задачею тестування алгоритму є розгортання Kubernetes на локальних серверах. Можна навіть створити приватну хмару. Minikube — один з найпоширеніших інструментів для локальної інсталяції Kubernetes-кластера на одному вузлі (single-node). Він підходить для тестування та розробки. Minikube дозволяє працювати з Kubernetes на локальному комп'ютері, імітуючи хмарне середовище.

4.1.2. Sidecars як помічники в балансуванні навантаження

Мікросервіси зазвичай запускаються в контейнерах з використанням оркестраційних фреймворків. Якщо мікросервіси є частинами розподіленого монолітного сервісу, то системи контейнерної оркестрації йдуть далі, дозволяючи кожному мікросервісу розміщуватися в кількох контейнерах. Наприклад, один контейнер може містити бізнес-логіку додатку, а інші — виконувати функції моніторингу або балансування навантаження. Такі додаткові контейнери називають «sidecars», оскільки вони розгортаються поруч із основним контейнером і обробляють вхідні або вихідні запити. Група контейнерів, що включає як основні, так і допоміжні контейнери, які разом утворюють логічний сервіс, об'єднується в єдиний простір імен, відомий як «pod» у Kubernetes.

Оскільки кожен «pod» може бути реплікований для масштабування мікросервісів, для маршрутизації запитів до відповідних вузлів потрібні балансувальники навантаження. Можливість легко інтегрувати функціональні компоненти дозволила замінити централізовані балансувальники навантаження розподіленими балансувальниками «sidecar», які розгортаються разом із кожним «pod». Кожен «проксі-sidecar»

відповідає за балансування вихідних запитів мікросервісу через кілька реплік у ланцюжку. Це робить систему більш масштабованою, хоча і без глобального огляду, як у централізованих рішеннях [11].

Для забезпечення функціонування доповненої групи береться схожа ідея, як і sidecars, адже дві чи більше мікропослуги мають розміщуватись в особистих контейнерах, але на одному поді. В цьому випадку зважаючи на стаке завантаження повністю підходить ідея спільного масштабування в разі потреби. Доречно користуватись вертикальним масштабуванням, оскільки передбачається, що воно буде мінімальним.

4.1.3. Додаткові інструменти для реалізації методу

Іншим інструментом є Kind (Kubernetes in Docker). Він дозволяє розгортати Kubernetes кластери безпосередньо у Docker-контейнерах. Це також хороший варіант для розробників, яким потрібно тестувати застосунки в контейнерах.

Kubeadm — це інструмент для створення повноцінних кластерів Kubernetes на локальних машинах або ВМ. Він забезпечує більш повну конфігурацію, що підходить для тестування або навіть продакшн-оточень.

Також існує MicroK8s, але оскільки це легка реалізація Kubernetes від Canonical (розробники Ubuntu), він не дуже підходить для тестування габаритного алгоритму, адже створений для розгортання Kubernetes-кластер на малопотужних пристроях, таких як Raspberry Pi, але теж і на локальних машинах.

Отже, хмарні платформи використовують Kubernetes як один з основних інструментів для оркестрації контейнерів. Локальне розгортання Kubernetes дозволяє імітувати хмарну інфраструктуру для тестування та розробки. З допомогою Kubernetes можна імітувати хмарні системи AWS, Azure, Google Cloud [79], [99], [100] та інші локально, що надає можливості для тестування додатків та оркестрації додатків у хмарному контексті без необхідності використовувати реальні хмарні ресурси. Таким чином, розгорнувши Kubernetes за допомогою Minicube можна зімітувати потрібні мікросервіси та протестувати алгоритм пошуку доповнювальних особин.

Метою підрозділу є показати можливість використання алгоритму доповнювальних навантажень як на локальному проєкті, запущеному на серверному провайдері чи хмарній системі, так і глобально для оркестрації всіх проєктів, мікросервісів та монолітів на рівні провайдера серверів чи хмари.

4.2. Методи збору вхідних даних.

Алгоритм базується на статистичних даних, запропонованих в першу чергу розробниками програмного забезпечення (ПЗ). Ці дані важливі для початкової класифікації мікропроцесора, вибору йому доповнювальних елементів. При неправильному виборі напарників, результати алгоритму не будуть кращими, ніж стандартні. Вартість помилки збору початкових даних не дуже велика, адже клауд-провайдер зніматиме і свої метрики, котрі будуть застосовуватись на наступних ітераціях. Аби коректно провести збір даних, слід користуватись додатковими інструментами.

Шаблони інструментів для збору та обробки метрик мікросервісів наступні:

- система моніторингу та збору метрик, яка добре інтегрується з Kubernetes, напр., Prometheus;
- інструмент для візуалізації метрик, напр., Grafana, Kibana;
- інструменти для збору та обробки логів. В логах можуть бути додаткові метрики, які можуть бути необхідні для формування повної картини у розробників ПЗ (напр., Fluentd або Logstash, Grafana Loki, ElasticSearch).

Оскільки для хмарного провайдера головне — самі метрики, а не їх візуалізація, нижче описано деталі стосовно роботи Prometheus.

4.2.1. Моніторинг, збір метрик та накопичення історичних даних за допомогою Prometheus

Prometheus — це потужна система для моніторингу та збирання метрик, яка працює шляхом надсилання HTTP-запитів до інтерфейсу взаємодії (API, endpoints), що повертає необхідні метрики. Такий інтерфейс

з локальними метриками має бути написаний в застосунках та сервісах, які потребують моніторингу. Зібрані дані зберігаються в спеціалізованій базі даних часових рядів Prometheus, що забезпечує високу ефективність і швидкість доступу до інформації, надає можливість оперативно аналізувати зміни у метриках із плином часу. Ця система є чудовим вибором для моніторингу використання ресурсів, таких як CPU, RAM і мережеві інтерфейси, оскільки містить в собі Node Exporter для експорту цих даних [101].

Prometheus підтримує кілька форматів даних, серед яких простий текстовий формат і OpenMetrics. Він здатен збирати метрики з різноманітних джерел, таких як:

- Веб-сервери
- Бази даних
- Апаратні пристрої
- Користувацькі застосунки

Для розробників при аналізі проблем продуктивності системи та статистики буде корисними PromQL (Prometheus Query Language) — потужна мова запитів, яка дозволяє гнучко запитувати та аналізувати збережені дані.

Для моніторингу системи в реальному часі в Prometheus існує механізм для створення сповіщення на основі заздалегідь визначених умов. Ці сповіщення обробляються через Alertmanager, який може інтегруватися з різними системами сповіщень, такими як електронна пошта, Slack або PagerDuty, забезпечуючи своєчасну реакцію на критичні події.

Прикладом збору Custom Metrics про кількість запитів у певні години дня може слугувати наступна конфігурація у yaml файлі [102]:

metrics:

- type: External
- external:
 - metricName: http_requests_per_second
 - targetValue: 100

4.2.2. Схема застосування інструментів аналізу та моніторингу в поєднанні з Prometheus

Приклад злагодженої роботи Kubernetes, Prometheus та Grafana представлено на Рисунку 4.1. Це налаштування можна зробити вручну, або ж скористатись помічниками з автоматичного налаштування, наприклад Deckhouse [103]. Модулі на рисунку описані нижче.

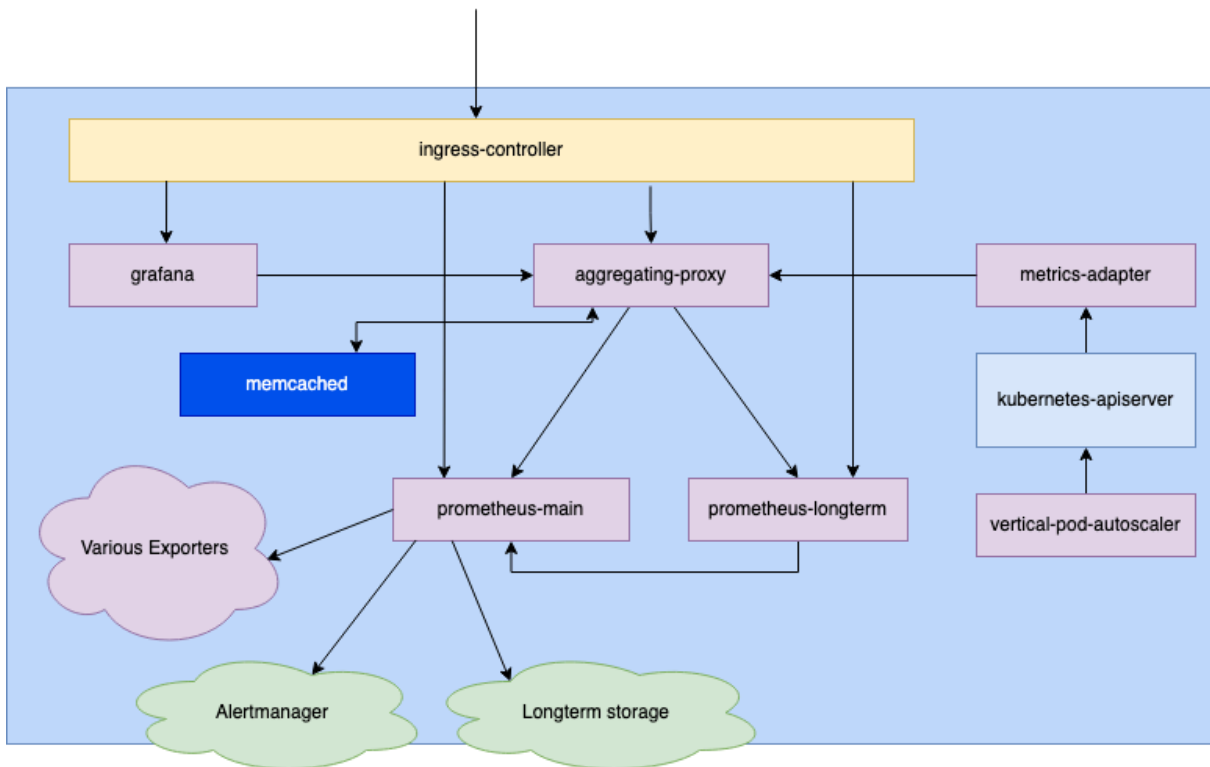


Рисунок 4.1 — Налаштування моніторингу за допомогою Prometheus та Graphana у Kubernetes [103]

Модуль prometheus-main — основний екземпляр Prometheus, який збирає метрики кожні 30 секунд. Цю величину можна змінити за допомогою параметра ``scrapeInterval``. Він обробляє всі правила, надсилає сповіщення та виступає як основне джерело даних.

Модуль prometheus-longterm — вторинний екземпляр Prometheus, який збирає дані з основного екземпляра Prometheus (``prometheus-main``) кожні 5 хвилин. Це значення можна змінити за допомогою параметра ``longtermScrapeInterval``. Він використовується для довгострокового зберігання історії та відображення даних за великі періоди часу.

Модуль `trickster` це кешуючий проксі, який зменшує навантаження на Prometheus.

Модуль `aggregating-proxy` це агрегуючий і кешуючий проксі, який зменшує навантаження на Prometheus та агрегує як основні, так і довгострокові дані в єдине джерело даних.

Модуль `memcached` — розподілена система кешування в оперативній пам'яті.

Модуль `grafana` — керована платформа спостереження з готовими панелями приладів для всіх модулів Deckhouse та популярних додатків. Екземпляри Grafana є високодоступними та налаштовуються за допомогою Custom Resource Definition (CRD).

Модуль `metrics-adapter` це компонент, що з'єднує Prometheus і API метрик Kubernetes. Він дозволяє підтримку НРА в кластері Kubernetes.

Модуль `vertical-pod-autoscaler` це інструмент автомасштабування, який допомагає оптимізувати ресурси CPU та пам'яті, необхідні для подів.

Various Exporters це попередньо підготовані експортери, підключені до Prometheus. Список включає експортерів для всіх необхідних метрик: `'kube-state-metrics'`, `'node-exporter'`, `'oomkill-exporter'`, `'image-availability-exporter'` та багато інших [103].

Підсумовуючи дану структуру слід зазначити, що збір історичних даних та їх аналіз перерахованими інструментами дають можливість забезпечити:

- прогнозування піків та спадів;
- врахування циклічності чи специфічних подій;
- виконати оптимізацію використання ресурсів;
- автоматизацію адаптивного масштабування.

Альтернативні інструменти для аналізу метрик

Альтернативу Prometheus в розрізі збору та збереження метрик по ресурсах для часових рядів складають InfluxDB та Telegraf. Telegraf — агент, який збирає дані про використання CPU, RAM, дискові ресурси та

мережеві інтерфейси з різних джерел і передає їх в InfluxDB. InfluxDB — одна з найпотужніших баз даних часових рядів, оптимізована для високошвидкісного зберігання і запитів з InfluxQL — SQL-подібна мова запитів, що дозволяє виконувати складні запити для аналізу продуктивності систем у реальному часі.

Також можна користуватись Zabbix, безплатним інструментом для збору різного роду метрик та логів, що працює через власні агенти або SNMP. Він має розвинену систему сповіщень і підтримує візуалізацію часових рядів.

Для здійснення професійного моніторингу на хмарному провайдері корисним інструментом буде комерційний хмарний сервіс Datadog. Він надає прості в налаштуванні агенти для збору метрик з серверів, контейнерів і хмарних платформ, таких як AWS, Azure чи GCP. Ним зручно збирати метрики, відстежувати ресурси та аналізувати продуктивність систем у реальному часі. Додатково, для старту та налаштування роботи системи, корисними будуть автоматичні дашборди – візуалізації з зібраних даних.

Cloud-платформи AWS, GCP, Azure та більшість інших мають вбудовані сервіси для аналізу історичних даних і прогнозування. Це AWS Predictive Scaling, GCP Recommendations та Azure Advisor відповідно.

Іншими доступними аналогами є New Relic, VictoriaMetrics та Netdata, які працюють з часовими рядами, великими даними та знімають потрібні метрики [101], [104].

Найуніверсальнішим, з відкритим кодом (opensource) та безплатним є Prometheus. З ним інтегруються багато додаткових інструментів, а головне — він співпрацює з Kubernetes за допомогою додатків Thanos або Cortex. Вони дозволяють масштабувати Prometheus для довготривалого зберігання метрик і роботи з розподіленими кластерами Kubernetes. Вони додають реплікацію, що забезпечує резервне зберігання, захист від втрат даних і централізацію метрик із декількох кластерів.

4.3. Мови програмування для написання скриптів

Обрана мова програмування має добре співпрацювати як з Kubernetes, так і з системою збору метрик. Як останню, оберемо Prometheus. До нього існують бібліотеки від виробника на шість мов програмування, а також додаткові бібліотеки для всіх поширених мов [105]. Kubernetes написаний на **Go**. Вона забезпечує високу продуктивність та інтеграцію з системою. Іншою поширеною та зручною мовою для написання скриптів, які можуть обробляти та аналізувати метрики є **Python**. Бібліотеки до цих мов написані авторами Prometheus, а отже є гарантія якості та підтримка в разі виникнення проблем. Для Python використовується бібліотека `kubernetes-client`, а для Go — `client-go`. Тому пропонується обрати одну з цих мов для виконання алгоритму пошуку доповнень для Kubernetes та хмарних систем.

4.4. Налаштування Kubernetes для виконання алгоритму

У Kubernetes можна налаштувати інтелектуальний розподіл мікросервісів між подами та вузлами, використовуючи різні механізми, такі як планувальники (schedulers) [71], політики аффініті та антиаффініті — приналежність до певних вузлів, чи навпаки, про які розказано нижче, а також метрики для масштабування і «розігріву» ресурсів. Для застосування алгоритму розподілу навантаження для вирішення, на які поди направляти застосунки, базуючись на ряді факторів, таких як наявність ресурсів (CPU, RAM) та визначення кількості подів на вузлі, слід інтегрувати зовнішні інструменти та метрики з бази даних для детального планування та передбачуваного «гарячого розігріву» (hot standby) [5], [96] вузлів.

Node Affinity i Pod Affinity/Anti-affinity — це механізми, що дозволяють задавати правила для планувальника, щоб він розміщував поди на конкретних вузлах або на основі близькості до інших подів. Наприклад, можна сказати Kubernetes розміщувати певні мікросервіси ближче один до одного або навпаки — розподіляти їх на різні вузли для зменшення

навантаження. Приклад наведено у наступному пункті [Типи обчислювальних ресурсів](#).

Механізм *Custom Schedulers* дає можливість розробити власний планувальник (scheduler) [20], [71], [72], який буде враховувати зазначені в алгоритмі фактори або для розподілу подів між вузлами. На цьому етапі, можна інтегрувати метрики з БД, щоб впливати на рішення планувальника.

За допомогою *Taints and Tolerations* можна «позначати» вузли, щоб певні поди могли бути розгорнуті тільки на них або уникати цих вузлів [23]. Це корисно для оркестрації подів, на яких навантаження може сильно змінюватись, аби запобігти ситуації призначення додаткового навантаження на вузол в той час, як скоро вільний простір на вузлі зникне. За умови додаткового навантаження довелось би робити додатковий небажаний скейлінг [81], [106], [107]. Приклад наведено у наступному пункті [Типи обчислювальних ресурсів](#).

Щоб реалізувати «гарячий розігрів» [5], [96] вузлів на основі метрик, можна інтегрувати рішення на базі Horizontal Pod Autoscaler (HPA) — додавання додаткових вузлів або цю задачу теж покласти на алгоритм на основі статистики метрик. Останнє допоможе запобігати піковому навантаженню і дасть більше часу для розігріву нової ноди. Горизонтальне автомасштабування використовує метрики, такі як CPU і пам'ять, для автоматичного збільшення або зменшення кількості подів на основі навантаження [107]. Витягнути метрики з бази даних і використовувати їх у HPA можна використовуючи *Custom Metrics API* або *Prometheus Adapter*.

Іншим механізмом масштабування є Vertical Pod Autoscaler (VPA) дозволяє автоматично змінювати кількість ресурсів (CPU, пам'ять) для окремих подів, але у цього підходу є обмеження. При зміні кількості ресурсу под необхідно перезавантажувати, таким чином це не може бути єдиний под для мікросервісу, щоб не припиняти обслуговування додатка [56], а також це бажаніше робити при зменшенні навантаження, а не збільшенні, щоб не переставати обслуговування пікового навантаження

користувачів та щоб не спричиняти потенційних проблем обслуговування користувачів пов'язаних з перевантаженням додатка. Таким чином, алгоритм пошуку доповнень орієнтований на горизонтальне розширення, котре гарантовано даватиме необхідний результат та буде більш передбачуване, не ставлячи під загрозу забезпечення сервісом користувачів.

Додатково слід відмітити Kubernetes Event-Driven Autoscaling (KEDA), що дозволяє масштабувати поди на основі подій або зовнішніх метрик, таких як повідомлення з черг, HTTP-запитів, або аналітики з баз даних. KEDA підтримує інтеграцію з Prometheus, Azure Monitor, AWS CloudWatch та іншими системами збору метрик. В разі раптової зміни обстановки можна скористатись цим механізмом, але на цей час алгоритм доповнювальних навантажень це не передбачає [12].

4.5. Вибір вузлів для виконання завдань залежно від ключового критерію

Як було вказано в [Кроці 7](#), у кожної задачі виділяється ключовий критерій, яким є ЦП, ОП чи мережевий ресурс. Залежно від цього відбуватиметься первинне групування задач на основі цього критерію та алгоритм проганятиметься окремо для кожної групи. Відповідно до ключового критерію задачам виділятиметься ОР. В підрозділах розглянуто типи серверів залежно від їх обчислювальних ресурсів [108].

4.5.1. CPU-інтенсивні сервери

CPU-інтенсивні сервери мають високі вимоги до процесора та часто потребують багатоядерних ресурсів, тобто віддається перевага вузлам з потужними CPU, наприклад, C-type інстанси в AWS. Вони використовуються для обробки даних, розрахунків або компіляції коду. Вони найкраще підходять для горизонтального масштабування та зазвичай мають більш передбачуваний профіль навантаження; мають потребу часто застосовувати «headroom» (запасний ресурс) в 25–30% для обробки неочікуваних піків.

Приклад маркування нод для CPU-інтенсивних задач на Kubernetes:

```
apiVersion: v1
kind: Node
metadata:
  labels:
    workload-type: cpu-intensive
    instance-type: c5.2xlarge
```

4.5.2. RAM-інтенсивні сервери

RAM-інтенсивні сервери, які потребують великої кількості RAM, але можуть помірно використовувати CPU. Для них використовуються R-type інстанси в AWS або memory-optimized VM у GCP. Застосовуються для баз даних, кешування, та in-memory аналітики, задач, розраховані на високе постійне навантаження. Ці сервери зазвичай підтримують найвищий рівень сталої утилізації серед усіх типів. Їм важче швидко масштабуються через тривалий час ініціалізації.

Щоб задати характеристики обмеження, які вказуватимуть, що для задачі необхідні тільки вузли з великою ОП, можна за допомогою tolerations. Своєю чергою слід схарактеризувати вузол як той, що має велику пам'ять за допомогою taint. В Kubernetes можна це описати наступним чином:

```
Taint на ноді:
spec:
  taints:
    - key: workload-type
      value: memory-intensive
      effect: NoSchedule
```

```
Toleration у поді:
spec:
  tolerations:
```

- key: workload-type
- operator: Equal
- value: memory-intensive
- effect: NoSchedule

4.5.3. I/O-інтенсивні сервери

I/O-інтенсивні сервери характеризуються високою активністю дискових операцій або мережевого трафіку. Для них використовуються інстанси з оптимізованим IOPS (Input/Output Operations Per Second), локальними SSD (solid-state drive — твердотілий накопичувач) або розширеною мережею. Найнижча стабільна утилізація серед усіх типів. Мережевий тип навантаження характеризується короткими, але інтенсивними піками активності. Такі сервери зазвичай розраховані на найгірший сценарій навантаження, що призводить до значної надлишковості у звичайних умовах. Це пов'язано з раптовими сплесками трафіку, які можуть виникати раптово, а також з потенційним вірусним контентом. Згадки в новинах або маркетингові кампанії можуть спричинити миттєве зростання у 2–10 разів.

Прикладами ОЗ, які розміщуватимуться на цих серверах є сховища даних, повідомлень або проксі-сервери. Приклад Node Affinity на Kubernetes для бази даних, який необхідний вузол з SSD наведено нижче:

spec:

affinity:

nodeAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

nodeSelectorTerms:

- matchExpressions:

- key: storage-type

- operator: In

values:

- ssd

4.5.4. Інші варіації навантажень

Додатково можна схарактеризувати ще *Batch-навантаження* (періодичні задачі) для короткострокових, але ресурсомістких операцій, під які виділяють пули нод з можливістю масштабування до нуля. Прикладами задач можуть бути ETL-процеси, аналітичні звіти, бекапи.

Також існують *Stateful-навантаження*, які вимагають постійного сховища та стабільного розташування, наприклад для баз даних, або розподілених файлових систем.

4.5.5. Завантаженість серверів різного типу

Для даної роботи цікаві в першу чергу перші три варіанти серверів, адже розглядається поділ саме за цими трьома характеристиками. Періодичні задачі часто можуть виконуватись у нічний час, тому вони підходять як варіанти доповнень до стандартних та можуть розміщатись на серверах першого чи другого типу, можливо третього. Сервери для *Stateful* навантажень зазвичай не будуть виконувати додаткових задач.

Для дослідження важливо проаналізувати завантаженість серверів різного типу, щоб могли оцінити ступінь можливої економії ресурсів в разі запровадження методу доповнювальних навантажень на серверах-планувальниках. Дані стосовно їх використання наведені в Таблиці 4.1 [108].

Таблиця 4.1 Використання ресурсів різних типів серверів

Тип серверів та їх навантаженість	CPU-intensive	Memory-intensive	I/O-intensive
Середнє використання	40–60%	60–75%	30–55%
Пікове використання	70–85%	80–90%	60–80%
Цільовий діапазон	65–75%	70–80%	50–65%

Безпечний резерв	15–25%	10–20%	20–35%
------------------	--------	--------	--------

Цільовий діапазон використання мережевих серверів найнижчий, що пов'язано з найменшою передбачуваністю мережевих піків, адже вони можуть спричинюватись сторонніми факторами та подіями у світі. Використання RAM-вузлів найбільш ефективно, що спричинено найменшою залежністю від сторонніх подій та передбачуваністю навантаження. Таким чином, чим більше передбачуваності, тим менша потреба в «headroom» (запасний ресурс) [108].

4.6. Реалізація методу пошуку доповнювальних навантажень

Алгоритм пошуку доповнювальних навантажень реалізований мною на мові Python. Реалізація складається із декількох логічних блоків:

- 1) симуляція вхідних даних;
- 2) підготовка даних та кластеризація, супутня візуалізація результатів;
- 3) пошук доповнювальних навантажень;
- 4) для безпарних задач, розбиття початкових навантажень на мілкіші з виконанням розумного обрізання піків, після чого рекурсивно виконується пункт 2 на нових навантаженнях.

У підрозділах детальніше описано особливості реалізації кожного етапу та в додатках наведено код базової реалізації алгоритму, котрий можна сприймати як псевдокод або алгоритм дій виражений програмно.

На Рисунку 4.2 зображено чергу задач, на кожній написано кількість необхідного ключового (основного) ресурсу, яким може бути ОП, потужність ЦП або мережевий ресурс. Одиниці виміру будуть різними залежно від ресурсу. В загальному випадку хоч ключовий ресурс один, але при роботі алгоритму до уваги беруться й інші характеристики.

Говорячи про навантаження, мається на увазі вектор навантажень, де кожній годині відповідає середній рівень завантаженості задачі. Водночас на малюнку як навантаження наведено одне значення, а не вектор. Це

зроблено для спрощення розуміння схеми роботи алгоритму методу та для пришвидшення підрахунків. Мається на увазі, що в кожен момент часу доповнення буде виконуватись так само, хоч і з іншими значеннями. Під одиничним числовим значенням можна розуміти також нормовану суму всіх величин кожного вектора, яка сумарно не буде перевищувати ресурс вузла в кожен момент часу, а буде наближатись до нього.

Рисунок відповідає всьому процесу, описаному включно до [Кроку 15: Комбінація малогабаритних мікропослуг](#), тобто первинному, проміжному та фінальному етапу розподілу задач по серверах з урахуванням цільової функції (2.18) та обмежень (2.19) матмоделі.

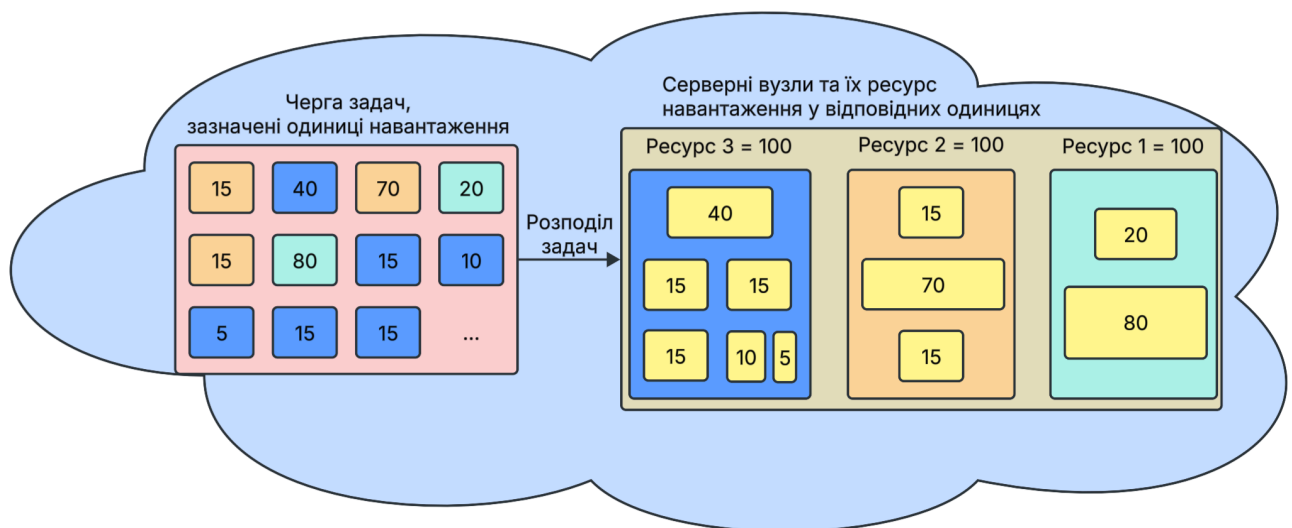


Рисунок 4.2 — Розподіл обчислювальних задач із черги по серверних вузлах за методом пошуку доповнювальних навантажень

4.6.1. Симуляція вхідних даних

Симуляцію даних виконано на основі виділених кластерних груп, описаних у підрозділі [Вибір кількості кластерів](#) на основі реальної статистики, а також використовувались дослідження наведені у [109]. З метою множення даних для досліджень вибрана тактика генерування довільної кількості даних за обраними шаблонами. Для симуляції пропорцій задач відповідно до реальних кількостей у РС, використовувались дані Таблиць 4.4 та 4.9. Код прикладів генерування даних наведено у [Додатку 1](#). Візуалізацію згенерованих ідеалізованих (близьких до симетричних) даних

можна побачити у [Додатку 2](#). [Додаток 3](#) показує кластеризацію як ідеалізованих, так і реальних часових рядів.

4.6.2. Підготовка даних та кластеризація, супутня візуалізація результатів

Реалізація коду цього пункту містить виклик *Z-score* нормалізації, *K-Means* алгоритму для кластеризації, а також візуалізацію за *PCA* алгоритмом. Зображення результатів наведених етапів знаходяться на Рисунках 3.3, 3.4, 3.5, а код у [Додатку 4](#). Для пошуку доповнень оцінюється статистика по кожному кластеру. Приклад наведено у [Додатку 5](#).

4.6.3. Пошук доповнювальних навантажень

Доповнення шукається тільки за ключовим ресурсом (див. [Крок 3](#)), додаткові характеристики мають не суперечити одна одній, тобто не має бути перетину чи накладення, неможливе результівне перевищення максимального ресурсу, навпаки, очікується недовикористання повного ресурсу за другорядними характеристиками. Якщо через другорядні показники неможливо знайти доповнювальне навантаження, слід зробити другорядний показник ключовим та перезапустити процедуру на основі нового показника.

Початково проводиться пошук доповнювальних кластерів, результати якого виведені у [Додатку 6](#). При пошуку безпосередніх пар слід порівнювати справжні амплітуди, тому у [Додатку 7](#) максимальне, мінімальне значення та проміжок, які виведені для кожної задачі, відповідають нормованому під заданий ресурс навантаженню, що використовується кожною задачею. Групи сформовані з урахуванням відсутності накладань навантаження в однакові моменти часу та аби використати максимальну можливу амплітуду.

4.6.4. Розбиття початкових навантажень на мілкіші

Знайти екстремуми в часовому ряді неважка задача, але знайти початок надмірного зростання, який зазвичай не збігається зі звичайним

початком зростання вже важче. Для розв'язання цієї задачі відбувається пошук екстремальних періодів, після чого суміжні значення порівнюються з середнім та максимальним значеннями часового ряду, які є поза екстремумом. Після цього виділяється розширений проміжок екстремуму, який може включати й локальні мінімуми в тому числі.

Такий підхід допоможе запобігати підвищеному навантаженню на серверному вузлі, адже цей проміжок є підозрілим на предмет потенційного перевантаження, і якщо не в один день, то іншого дня воно може відбутись. Детальніше про визначення області відсікання можна прочитати в [Кроці 13](#) та подивитись код та графіки відсічень у [Додатку 8](#) та [Додатку 9](#).

4.6.5. Подальші кроки

Після запуску система потребує продовження моніторингу. На цій стадії потрібно слідкувати за оновленням метрик та за якістю розподілу навантажень. Якщо виявляється, що протягом певного часу серверна група використовує замалу кількість ресурсів, чи навпаки, потребує більше, то такі групи слід виокремити, скомбінувати з новими мікросервісами за нагоди та виконати пошук нових доповнень.

4.7. Оцінка ефективності планувальника доповнювальних навантажень

Кількісні значення підвищення ефективності роботи планувальника на основі методу доповнювальних навантажень залежить від типу архітектури ОЗ, та від пропорції кількості задач в денний та нічний час. Детальна інформація, на основі якої робились висновки, наведена у підпунктах. Табличні дані були взяті здебільшого з опитувань про хмарні технології, здійснених вповноваженими організаціями [71], [110], [111], [112], а також взяті з документації Kubernetes, AWS, GCL, Azure.

4.7.1. Опис імітаційного моделювання та експериментального дослідження

Для виконання імітаційного моделювання були взяті дані, згенеровані по принципу, описаному у пункті [Симуляція вхідних даних](#).

1) Дані являли собою часові ряди, в кожному по 24 елементи, що відповідало середньому навантаженню за кожну годину доби.

2) Вибірка репрезентативно представляла дані щодо мікросервісів, монолітів та безсерверні застосунків в пропорціях, взятих з Таблиці 4.4.

3) Ключовим критерієм було взято CPU, RAM та мережевий ресурс в пропорціях, вказаних у Таблиці 4.2 в колонці «Ключовий критерій». Цей розподіл відповідає реальному, представленому на хмарах.

4) Кількість доповнень була згенерована відповідно до колонки «Кількість доповнень» у Таблиці 4.2. Ці дані відповідають середньому нічному завантаженню системи, яке є найменшим за добу. Відповідно, більше доповнень, ніж активних процесів в період найнижчої активності, бути не може. Все ж не всі нічні процеси є доповненнями до денних, тому для імітації слід взяти **коефіцієнт доповнюваності k** , який має значення у межах $[0; 1]$.

Наприклад, вирахуємо кількість доповнень, якщо $k = 0.7$ для задач з ключовим критерієм CPU. Візьмемо значення з таблиці, що знаходиться на перетині рядка «CPU» та колонки «Кількість доповнень», яке становить 30%. Порахуємо відсоток рядків, який відповідає кількості доповнень, що слід згенерувати для CPU задач: $30\% \cdot 0.7 = 21\%$.

Отже, задаючи у вихідних параметрах до алгоритму генерації даних $k = 0.7$ очікуватимемо що близько 21% буде повних доповнень. Інші процеси можуть доповнюватись частково, або ділитись формуючи доповнювальні групи можливо більшого розміру, ніж дві задачі, тому 21% доповнень не означає, що тільки стільки їх буде. очікується, що їх буде близько значення, вказаного у таблиці, але решта доповнень буде у вигляді груп більших ніж по дві задачі. Для наведеного прикладу це $30\% - 21\% = 9\%$

Під час реального експерименту, навантаження на мережу, а через неї й на ОП та ЦП будь-якого рівня можна емулювати за допомогою Gatling або JMeter, як доповнювальне, так і надмірне з замалим.

В лабораторних умовах в разі імітації натуральних даних навантаження, експеримент слід проводити добу, щоб побачити якість роботи алгоритму. Водночас навантаження в інші дні може змінюватись та більше не бути доповнювальним. Тоді слід буде виконувати перепланування формуючи коректні групи на кожен день, або ж окремо на будні та вихідні зі святковими днями.

4.7.2. Тестування алгоритму

Для тестування алгоритму пошуку доповнень, генеруються вектори та доповнення до заданих векторів, повні (симетричні), які точно мають бути знайденими, а також з похибками, аби перевірити коректність роботи алгоритму в ситуації, що імітує реальну.

Кількість тестових даних відповідала даним з Таблиць 4.6 та 4.7, що дозволило максимально близько до реальності зімітувати процес завантаження вузлів маленького, середнього та великого кластерів. А саме, були взяті наступні варіації:

- маленький кластер з 30 маленькими та 15 середніми вузлами, 1350 задач (навантажень) для розподілу;
- середній кластер з 30 маленькими, 80 середніми та 10 великими вузлами, 7250 задач для розподілу
- великий кластер з 600 маленькими, 400 середніми та 200 великими вузлами, 69000 задач для розподілу.

Розмір вузла відігравав обмежувальну роль на сумарний об'єм розміщення задач на кожному елементі.

Результати наведені у Таблиці 4.2 є оцінювальними. Дані про ефективність говорять, на стільки відсотків більше задач можна виконувати на обчислювальному вузлі продовжуючи надавати ту саму якість та враховуючи безпечний запас ресурсів. Іншими словами, **ефективність дорівнює** кількості простою ресурсу протягом доби з урахуванням незайнятого безпечного запасу ресурсу (CPU, RAM, каналний ресурс),

який буде зайнятий роботою. Значення ефективності вираховується як різниця між піковим та середнім навантаженнями на сервери (див. Таблицю 4.1), призначені для активної роботи з відповідним ресурсом з урахуванням особливостей архітектури застосунків, які виконуються. Порахувавши різницю між цими значеннями для всіх типів ключового ресурсу відсоткові значення ефективності будуть знаходитись в межах інтервалу [10; 20]. Детальніше розписано в Таблиці 4.2. Дані наведені за день роботи. Якщо брати тиждень, показники можуть бути менші, адже доведеться виконувати перепланування через зміну характеру навантаження та задачі.

Пропорції кількості доповнень були взяті з даних про час доби – Таблиця 4.9: не може бути доповнень більше, ніж активних задач у нічний час, коли їх кількість мінімальна. Ефективність, зазначена не для повної вибірки, а для її доповненої частини.

Таблиця 4.2 Ефективність застосування алгоритму доповнювальних навантажень. Дані ефективності відносяться тільки до доповнювальних груп, а не повної вибірки

Ефективність методу при виборі критерія ключовим	Ключовий критерій	Кількість доповнень (Додатково існує коеф. k)	Мікросервіси	Моноліти	Безсерверні застосунки	Усереднена ефективність
CPU	30–40%	30%	13–17%	12–15%	6–9%	10–14%
RAM	25–35%	50%	18–21%	15–17%	4–7%	12–15%
Мережевий ресурс	15–25%	25%	19–22%	17–20%	10–13%	15–18%
Усереднена ефективність			16–19%	13–16%	6–8%	

4.7.3. Аналіз результатів імітаційного моделювання та експериментального дослідження в розрізі ключового ресурсу

4.7.3.1. Підвищення ефективності у мережі

Показники мережі найкращі, що пов'язано з моментальною зміною стану від завантаженого до пустого в момент закінчення передачі одного запиту. Ефективність на мережевому рівні дуже важлива, адже це саме те, де спостерігається найменший рівень передбачуваності, через що закладається високий додатковий ресурс headroom.

Зважаючи на те, що можна працювати не з усім навантаженням одночасно, але можна виділити групу доповнюваних наборів задач, розмістивши їх на окремих серверах, то там можна буде підвищити рівень середньої завантаженості сервера з 50–65% максимальних до 75–80% (додаючи значення усередненої ефективності до цільового діапазону, зазначеного в Таблиці 4.1).

4.7.3.2. Підвищення ефективності у CPU

Аналогічна ситуація про навантаження на ЦП. Цільовий діапазон складає 65–75% (Таблиця 4.1). Додаючи відсоток ефективності з Таблиці 4.2, але пам'ятаючи, що це буде тільки для не більше ніж для 30% задач, отримаємо 75–85% — результативний цільовий ресурс на 30% серверів (збільшений на 10%). Максимальне використання таких серверів становить 85%. В разі його підвищення до 95%, можна спостерігати уповільнення роботи, адже висока утилізація CPU (>90%) призводить до збільшення часу очікування у черзі процесів. Тому слід дивитись, щоб не було використання більше ніж на 90% процесору у всякий час.

4.7.3.3. Підвищення ефективності у RAM

По ОП спостерігається високий рівень доповнюваності задач, ефективність теж на хорошому рівні, але сама ситуація з передбачуваністю рівня використання ресурсу задачами вже є достатньо на хорошому рівні. Щобільше, сервери, заточені для операцій на пам'яті вимагають безпечний

інтервал запасу, що в першу чергу пов'язано з обчислювальними операціями з великим об'ємом даних. «Запасну пам'ять» обов'язково слід залишити для збільшеної обчислювальної раптової потреби.

Все ж, покращення спостерігається через повільність відпускання ОП операційною системою. У випадку доповнювальних груп через достатньо довгі затримки у часі між повною зміною потреб в пам'яті між задачами, впродовж яких «більша» задача все ж встигає відпустити пам'ять, щоб зростальна задача нею скористалась. Це і призводить до вищої ефективності розміщення таких задач разом. В разі великих груп задач з піковими навантаженнями, показники якості доповнення по RAM будуть менші через бажання кожної задачі зарезервувати пам'ять на майбутнє. Підсумовуючи, оскільки початкові характеристики брались з урахуванням терміну відпускання RAM, доповнення за цим показником теж існують та показують хороші результати.

Цільовий діапазон використання Memory-intensive сервера складає 70–80%, але в пікові навантаження це значення зростає і до 90%. Завантаження більше ніж на 90% може бути небезпечним для роботи системи. Все ж, можна зробити планове навантаження більшим, наскільки це дозволяє надійність виконання операцій (десь до 95%).

Таке покращення можливе не на усіх 50% доповнень, а скоріш на кількості, що відповідатиме коефіцієнту повних доповнень k , адже якість доповнень до RAM в разі раптових сплесків, а не планомірних задач може бути контрпродуктивною. Зависоке використання ОП може призвести до свопінгу і катастрофічного падіння продуктивності.

4.7.3.4. Відмовостійкість системи залежно від навантаження

При виході з ладу одного вузла в кластері з кількох вузлів інші повинні мати запас для прийняття додаткового навантаження. Для прикладу, у кластері з чотирьох вузлів з навантаженням 75%, вихід одного вузла призведе до 100% утилізації інших, рівномірно розподіливши навантаження

по 25% між кожним. Тому завантажуючи сервери до максимуму слід залишати місце на випадок падіння одного чи двох вузлів одночасно, про що каже **правило N+1 або N+2** [113], [114]. Загалом це не має бути проблемою навіть для маленького кластера на хмарі (див. Таблицю 4.6).

Headroom (запас ресурсів) – це навмисно зарезервований обсяг невикористаних обчислювальних ресурсів, який залишають доступним для обробки непередбачуваних сплесків навантаження, збоїв та інших непередбачених обставин. Для його розрахунку слід брати декілька видів показників, описаних нижче [115].

На основі історичних даних виконавши аналіз піків навантаження за останні 6–12 місяців можна порахувати:

$\text{Headroom} = (\text{Історичний максимум} - \text{Середнє навантаження}) * 1.2-1.5$
(коефіцієнт запасу).

За типом навантаження залежно від типу системи є теж загальні показники:

- Передбачуване навантаження (корпоративні системи): 20–30% headroom
- Публічні сервіси з помірною волатильністю: 30–40% headroom
- Високоволатильні сервіси (соціальні мережі, e-commerce): 40–50% headroom

Статистичні методи дають загальний показник додаткового запасу ресурсів. Наприклад, можна скористатись стандартним відхиленням:

$\text{Headroom} = \text{Середнє навантаження} + (2-3 * \text{Стандартне відхилення})$.

Стосовно галузевих практик, дані щодо запасних ресурсів наступні:

- Netflix: використовує «буфер 70%» (тобто 30% headroom) для своїх систем автоматичного масштабування;
- Google Cloud: рекомендує підтримувати утилізацію CPU нижче 75% для оптимальної продуктивності;
- Amazon: для критичних систем дотримується правила «подвійного запасу» (2x від розрахункової потреби).

У сучасних хмарних середовищах мистецтво полягає в балансуванні між економічною ефективністю (висока утилізація) та надійністю системи (достатній headroom), і цей баланс значною мірою залежить від конкретних бізнес-вимог.

4.7.3.5. Удосконалення методу автомасштабування через баланс між максимізацією використання ресурсів та відмовостійкістю системи залежно від навантаження

Формування вузлів повністю з задачами, сформованими за принципом доповнювальних навантажень призведе до створення штучного запасного ресурсу, який на практиці майже ніколи не використовуватиметься. Все ж, заміна частини звичайних завдань ОР на доповнені дозволить зменшити headroom залишивши його використовуваним здебільшого іншими задачами, або ж щоб він служив на випадок відмови вузлів та перебалансування навантаження.

Беручи до уваги пікове навантаження на вузли (Таблиця 4.1), можна говорити про підвищення навантаження на вузли CPU та RAM на 5%, і I/O на 10% за умови сталості навантаження, яким і є доповнювальні групи.

Щоб порахувати частку присутності доповнювальних груп на ОВ та кількість зменшення вузлів в результаті такої оптимізації, скористаємось наступними поняттями:

Позначимо:

- N – початкова кількість задач.
- L – початкове навантаження на кожен сервер (у відсотках).
- M – початкова кількість серверів.
- a – частка задач, що стали ефективнішими (у відсотках).
- e – приріст ефективності цих задач (у відсотках).
- L_{\max} – максимально допустиме навантаження на сервер (у відсотках).
- M_{new} – нова кількість серверів після оптимізації.
- α – частка нових, ефективніших задач на кожному сервері.

1. Формула для частки нових задач на сервері

Кожна нова задача тепер використовує на $e\%$ менше ресурсу. Враховуючи, що частка таких задач становить $a\%$ від загальної кількості, частка завантаження, яку вони займають на сервері, змінюється на:

$$\alpha = a \cdot \frac{L}{L_{\max}} \cdot \left(1 - \frac{e}{100}\right). \quad (4.1)$$

2. Формула для нової кількості серверів

Загальне навантаження після оптимізації буде таким:

$$M_{\text{new}} = M \cdot \frac{L}{L_{\max} - \alpha \cdot L}. \quad (4.2)$$

Рахуючи зміни для RAM, підставимо наступні значення:

$$a = 30\% = 0.3, e = 15\% = 0.15, L = 85\% = 0.85, M = 100, L_{\max} = 90\% = 0.90.$$

Обчислюємо α :

$$\alpha = 0.3 \cdot \frac{0.85}{0.90} \cdot (1 - 0.15) = 0.3 \cdot 0.9444 \cdot 0.85 = 0.2402. \quad (4.3)$$

Обчислюємо M_{new} :

$$M_{\text{new}} = 100 \cdot \frac{0.85}{0.90 - 0.2402 \cdot 0.85} \approx 90.3. \quad (4.4)$$

Отже, кількість серверів після оптимізації зменшиться приблизно до 90, тобто економія становить 10%.

Провівши розрахунки для всіх, отримаємо наступні значення (Таблиця 4.3):

Таблиця 4.3 Сумарний результат збільшення ефективності на серверах різного типу

Тип серверів та їх навантаженість	CPU-intensive	Memory-intensive	I/O-intensive
Дозволений проміжок підвищення навантаження	5%	5%	10%
Кількість	30%	$50\% \cdot k = \pm 30\%$	25%

доповнювальних груп			
Підвищення продуктивності доповн. груп	10–14%	12–15%	15–18%
Максимальний дозволений відсоток доповн. груп на вузлі	20%	24%	22%
Результівне зниження кількості необхідних вузлів	8%	10%	17%

Таким чином, найбільша ефективність теоретично припадає на мережеві навантаження. Сумарне зменшення кількості вузлів рахується як Результівне зниження кількості необхідних вузлів (Таблиця 4.3) помножене на відсоток присутності технології на хмарі – Таблиця 4.2, колонка Ключовий критерій.

Загальне зменшення ресурсу складає 9.2% серверів:

$$8 * 0.35 + 10 * 0.30 + 17 * 0.20 = 2.8 + 3 + 3.4 = 9.2\% \quad (4.5)$$

Окрім загального зменшення серверів, пропонується **знижити кількість запасного ресурсу headroom** на серверах, де є групи доповнювальних навантажень на відсоток, скільки є присутності групи, тобто за Таблицею 4.3, рядок «Максимальний дозволений відсоток доповн. груп на вузлі» – **20–24%**, залишаючи мінімально декілька відсотків (5%) на непередбачувані ситуації.

4.7.3.6. Удосконалення методу автомасштабування через виділення серверів з виключно доповнювальним навантаженням

Удосконалення методу автомасштабування, що пов'язане з частковим максимально корисним з погляду запасного ресурсу введенням груп доповнювальних навантажень на сервери кластера можна представити

іншим чином. Альтернативою є формування кластера з більшістю серверів заповненими групами доповнювального навантаження. Такий стиль розміщення дасть можливість з високою ймовірністю передбачати енерговитрати на його підтримку, а також зменшити кількість серверів-планувальників.

Водночас не рекомендується повністю виключити з кластера сервери зі звичайно розпланованими задачами, адже в разі поломки одного з вузлів, його задачі слід буде перерозподілити між іншими. Тому кількість звичайних серверів, а саме їх резервного ресурсу має відповідати розміру одного чи двох вузлів.



Рисунок 4.3 — Кількість безпечного ресурсу на звичайному вузлі залежно від його ключового критерію та пріоритетності запущених задач за звичайних умов лежить в межах 15–50%.

На Рисунку 4.3 зображено запасний ресурс, який можна зменшити на кожному сервері пропорційно до кількості доповнювальних груп, виконуваних на ньому, але залишаючи мінімально необхідне запасне значення, яке може становити 5% для серверів, повністю заповнених доповнювальними групами. Зважаючи, що значення запасного ресурсу коливається в межах 10–50% (див. Таблицю 4.1), залежно від типу виконуваних задач та їх пріоритету, на серверах з виключно доповнювальними групами навантаження можна буде розміщувати більше задач, що суттєво зменшить кількість необхідного обладнання, при цьому

підвищуючи відмовостійкість системи. Таку тактику можна застосовувати в години пік для збільшення пропускної здатності інфраструктури, що дасть можливість зекономити на покупці нового обладнання та місці його розміщення.

Оскільки доповнювальних груп може бути 20–30%. Беручи до уваги, що запасного ресурсу в екстремальний період лишається менше, ніж зазвичай, тобто 10–20%. Для переформатування навантаження не всі завдання може бути доречно виводити з виконання, тому обмежимо кількість доповнювальних груп 10–15%, отримаємо такий самий відсоток серверів заповнених з безпечним інтервалом не 10–20%, а 5%. Отже, **виграш складатиме 5–15% місця на 10–15% відсотках серверів.** Загальний приріст задач складає 20–30% (див. Таблицю 4.1). Виграш не покриє повністю всю необхідність, але все ж дозволяє зекономити.

4.7.4. Аналіз результатів імітаційного моделювання та експериментального дослідження в розрізі архітектури задач

Таблиця 4.4 Присутність кожного типу задач на хмарах

Тип задачі	Мікросервіси	Моноліти	Безсерверні застосунки	Інші архітектури
Кількість	60–65%	15–20%	15–20%	5%
Кількість перепланувань на 100 подів	100–200	20–30	300–500	50–70
Зменшення об'єму на вихідні	30–70% (50%)	10–30% (20%)	40–90% (65%)	20–50%

Основну частину задач складають **мікросервіси** (60–65% навантажень), що показано в Таблиці 4.4. Вони легко переміщуються між вузлами, комбінуються з іншими задачами, адже не є дуже габаритними, мають різні профілі споживання ресурсів та передбачувані патерни

навантаження. Це все робить внесок у підвищену ефективність методу доповнювальних навантажень для застосунків мікросервісної архітектури (див. Таблицю 4.2), яка становить 16–19% для вузлів, що знайшли пари.

Монолітні застосунки хоч і виходять з моди, але складають 15–20% ПЗ на хмарі. Вони створюють базове навантаження, навколо якого можна оптимізувати розміщення інших сервісів. Вони мають стабільне споживання ресурсів, що дає змогу теж планувати та розподіляти навантаження і з ними. Через меншу гнучкість та можливість масштабуватись, використання методу доповнювальних навантажень буде середньоефективним, адже деякі моноліти непридатні до реплікацій та співпраці з балансувальником навантажень. Тому ефективність методу для них буде близько 13–16% для вузлів, що знайшли пари.

Безсерверні технології складають близько 15–20% ПЗ, що використовується на хмарі. Це короткострокові задачі добре підходять для заповнення «прогалин». Вони можуть динамічно розміщуватись на вільних ресурсах. Як і мікросервіси, мають чіткі метрики споживання ресурсів. Через свою малогабаритність та низькі потреби, при використанні методу доповнювальних навантажень підвищення ефективності роботи на хмарі не буде великим, що пов'язано з вже достатньо високою опитмізованістю технології, запуском її часто вже як доповнення до поточних задач на сервері, а також з меншою передбачуваністю кількості необхідних безсерверні технологій. Тому може не бути потреби в запуску декількох таких програм, на які буде розрахунок. Відносно малі показники по RAM пов'зані з важкістю ОП адаптуватись до раптових навантажень, які спричиняє цей тип архітектури.

Детальний аналіз витрат та ефективності буде наведений в підпунктах. Слід зауважити, що числові дані в таблицях є приблизними, і базуються на типових корпоративних розгортаннях. Розрахунки ефективності базуються на пропорційно усереднених даних. Перед початком впровадження методу доповнювальних навантажень у серверній

інфраструктурі слід проаналізувати статистичні дані окремої інфраструктури та зробити перерахунки базуючись на наведених прикладах. Фактичні значення можуть суттєво відрізнятися залежно від:

- конфігурації кластера;
- характеристик навантаження;
- хмарного провайдера;
- стабільності інфраструктури;
- шаблонів розгортання.

4.7.5. Аналіз зменшення кількості перепланувань

Основною перевагою методу є зведення до мінімуму кількості перепланувань. Говорячи про кількість перепланувань, важливо розглянути їх причини. Залежно від задачі, вони можуть бути різні та траплятись в різних пропорціях, але взагалом вони наступні:

- оновлення версії ПЗ та планове обслуговування (30%);
- автомасштабування(15–25%);
- масштабування на вимогу (більш схоже на планове з погляду ПЗ, але все ж для хмарної системи воно непередбачуване) (10–15%);
- холодний/гарячий старт (тобто планове масштабування) (15–20%);
- перебалансування навантаження(10%);
- оптимізація ресурсів хмари(10–15%);
- відмови вузлів(10–15%).

Позбутись перепланування повністю неможливо, адже планове обслуговування чи то ПЗ, чи сервера є необхідним. Холодний та гарячий старт — це той вид масштабування, котрий плановий, і до якого слід звести максимум всіх масштабувань, які відбуваються, звівши до мінімуму автомасштабування, бо воно є причиною піків напруження та непередбачуваного навантаження на вузли планування. Також, для його передбачення витрачається немало супутніх ресурсів інфраструктури.

Слід очікувати, що перед вихідними буде велика кількість перепланувань, адже навантаження на вихідних зовсім інакше розподіляється, ніж буднями.

Оптимізація ресурсів хмари є необхідною процедурою, але в ній буде мінімальна потреба при плануванні за допомогою методу доповнювальних навантажень.

Підсумовуючи, залежно від архітектури задач, позбутись можна було б 35–55% перепланувань, спричинених автомасштабуванням, масштабуванням на вимогу та перебалансування навантаження. Однак, слід розуміти, що масштабування все одно може статись хоч і меншою мірою. Щобільше, воно буде регулярно траплятись для більшості задач перед вихідними, що складатиме більше звичайного навантаження. Оцінимо кількість таких перепланувань як таку, що зменшить виграш на 30%, отже $35\text{--}55\% \cdot 0.7$ складатиме **кількість перепланувань, якої можна позбутись стає 25–35% для груп доповненого навантаження. Вцілому, якщо таких груп до 30%, це значення слід помножити на 0.3, що становить 7,5 – 10,5%**

4.7.6. Аналіз витрат часу при плануваннях та переплануваннях

Ефективність може виражатись у відсутності витрати часу, зменшенні використання CPU та мережевого ресурсу, оперативної пам'яті.

Таблиця 4.5 Витрати на планування та перепланування задач

Планування	CPU на операцію	RAM на оп.	Мережевий ресурс	Запуск контейнера	Початкове планування	Перепланування	Сворення поду, інстансу
Витрати	0.1–0.5 ядра	100–300 МБ	10–50 МБ	2–5 сек	100–500 мс	200–800 мс	10–30 сек

Причиною витрат мережевого ресурсу на планування є передача даних для міграції стану.

Беручи до уваги дані з Таблиці 4.5, середня найменша кількість часу, витрачена на планування чи перепланування складає 15 секунд.

При відсутності 100 перепланувань, буде зекономлено $15 \text{ сек} * 100 = 1500 \text{ сек} = 25 \text{ хвилин}$ процесорного часу, що є мінімальним показником для 100 завдань (див. Таблицю 4.4). Операцій паралельно на одному процесорі відбувається багато, тому говорячи про економію часу для великого кластера (див. табл. 4.6), де показник економії часу складатиме в 10 разів більше, тобто 250 хвилин на ту саму кількість завдань. Слід зазначити, що ця економія виражатиметься у меншій завантаженості процесора та мережі, таким чином на планування можна буде виділити менш потужний сервер, наприклад, не на 16 ядер, а на 14.

Якщо взяти, що кількість перепланувань пропорційна кількості вузлів в кластері, то малі (50–200 вузлів) та середні (200–1000 вузлів) кластери при відсутності необхідності перепланувань могли б зекономити в середньому від 0.5 до 3 годин в день, а це є дуже суттєво. Від усіх перепланувань позбутись не вдалось би, адже вони трапляються не тільки через відмову вузлів, обмеження ресурсів та масштабування, а ще і через оновлення додатків (~20%) та оптимізацію продуктивності (~15%), що зазначено в Таблиці 4.8. Тому 70% зекономленого часу через відсутність перепланування слід прибрати (див. минулий пункт, відсоток перепланувань, які можна прибрати алгоритмом становить 25–35%). Віднімемо ще 70%. Результівний зекономлений час буде становити від декількох хвилин до однієї години на 100 задач на великому кластері.

Сумарно якщо брати кількість планувань та перепланувань (див. Таблицю 4.6, рядок «Співвідношення кількості планувань до перепланувань»), виходить 2.3–3 частини. Тоді кількість перепланувань залежно від розміру кластера становить 30–45%. Тобто **зекономити можна 25–35% від цієї кількості в разі коли всі задачі доповнювальні, що становить 9–15% від всіх операцій планування та перепланування.** Якщо ж взяти, що кількість доповнювальних задач не перевищує 30%, то перемноживши ці значення отримаємо **середньозважене значення зменшення перепланувань в 2.1–4.5%.** Користуючись цими

розрахунками, можна відповідно знижувати потужність серверів на планування або збільшувати кількість задач, які на них надходять.

4.7.7. Аналіз економії на серверах-планувальниках

Таблиця 4.6 Параметри кластерів

Параметри кластерів	Малі кластери	Середні кластери	Великі кластери
Кількість вузлів	50–100 вузлів	100–500 вузлів	500–1000+ вузлів
Кількість кластерів	60–65%	25–30%	5–10%
Кількість операцій (пере)планування	100–300 операцій/год	300–1000 оп/год	1000–3000+ оп/год
Співвідношення кількості планувань до перепланувань	2:1	1,7:1	1,3:1
Кількість master вузлів планування	3 вузлів	3–5 вузлів	5–7 вузлів
Характеристики вузлів планування	4 cores, 8 GB RAM	8 cores, 16 GB RAM	16 cores, 32 GB RAM
Нове відношення кількості пере			

Стосовно кількості планувальників слід зазначити, що активним завжди є тільки один. Інші знаходяться в режимі leader-election. При відмові активного (мастера) – один з резервних стає головним. Типовий час перемикання становить 10–30 секунд.

Розглядаючи витрати ресурсів, залучених під час планування та розподілу задач, можна побачити, що у кластері залежно від розміру знаходиться 3–5–7 вузлів-планувальників, але активний тільки один в кожен момент часу. Тому беручи до уваги розрахунки, наведені в минулому пункті, можна зменшити потужність кожного планувальника, що дозволить зекономити на якості серверів, але не на кількості, адже зазначена кількість

необхідна для розподіленої системи планування та високої надійності системи. Детальніші розрахунки наведено нижче.

Таблиця 4.7 Кількість завдань (подів) на вузлі залежно від його типу

	Малі ноди	Середні ноди	Великі ноди
Кількість подів (завдань)	10–30	30–100	100–250+
Тип серверів	t3.medium, t3.large	m5.xlarge, c5.2xlarge	m5.4xlarge і вище

Кількість виконуваних завдань на нодах, що представлені в Таблиці 4.7 може сильно варіюватись не тільки від розміру ноди, а і від розміру самих завдань, тобто їх власного лімітування ресурсів представлених як requests та limits в описі подів. Також впливають налаштування Kubernetes – зокрема, розмір kube-reserved і system-reserved.

Поди з високою ресурсомісткістю обмежують загальну кількість завдань. Наприклад, AWS має встановлені обмеження: за замовчуванням максимум 110 подів на ноду в EKS. У GCP в GKE типові значення аналогічні, а в Azure AKS ці обмеження трохи нижчі. Загалом, оптимальна щільність подів визначається при балансуванні між ефективністю використання ресурсів та ризиками відмови вузла [27], [47], [116].

Таблиця 4.8 Причини перепланування

Причина	Відмова вузлів	Обмеження ресурсів	Масштабування	Оновлення додатків	Оптимізація продуктивності
Кількість	~10%	~30%	~25%	~20%	~10%

Що стосується кількості операцій (пере)планування на кожному класері, то вона нелінійно залежить від кількості вузлів (Таблиця 4.8). Нелінійність пов'язана з більшою можливою кількістю точок відмов та складнішою топологією мережі. При збільшенні кластера в два рази

кількість операцій зростає в 2.5–3 рази, складність планування – в 3–4 рази, а час на прийняття рішень збільшується на 40–60%. Через це зменшення кількості необхідних планувань суттєво покращить параметри роботи саме кластерів з великою кількістю вузлів.

За даними наведеними в Таблиці 4.6 про кількість операцій планування та перепланування та їх пропорції залежно від розміру кластера, зменшення необхідних ресурсів може становити 15% в результаті мінімізації кількості перепланувань до 60–70% наявного, тобто на 30–40% менше. Такі кластери типові для cloud-native провайдерів та великих data-центрів, використовуються великими підприємствами, підходять для глобальних сервісів.

Таблиця 4.9 Використання ресурсів залежно від часу

Час доби	CPU	RAM	Мережевий ресурс
Денний (9:00–18:00 = 9 годин = 0.375 доби)	65–85%	70–90%	70–90%
Нічний (22:00–6:00 = 8 год = 0.33 доби)	20–40%	40–60%	15–35%
Вечірній та ранковий час (6:00–9:00 та 18:00–22:00 = 7 годин = 0.29 доби)	40–65%	50–70%	35–70%

Дані в Таблиці 4.9 наведено середні, кожна клауд система підлаштована під свій набір користувачів та її параметри можуть відрізнятись. Причини такого порівняно низького завантаження мережі в нічний час при немалій завантаженості інших ресурсів пов'язана з виконанням інших типів задач, які менше пов'язані з користувачами, таких як бекапи, обробка накопичених пакетів даних (batch), виконання індексації та аналітики. Найбільша активність в онлайн магазинах спостерігається ввечері. Слід також взяти до уваги, що значення має тип серверів, і різниця між денним та нічним споживанням залежить від географічного розподілу в наступному відношенні:

- Для локальних сервісів: 40–60%

- Для глобальних сервісів: 20–30%
- Для гібридних систем: 30–40%

4.7.8. Аналіз витрат та економії матеріального забезпечення й електроенергії в робочих вузлах

Таблиця 4.10 Витрати worker-вузла на діяльність мікропослуги

	CPU	RAM	Мережа	Диск I/O
При інсталяції обч. задачі	пік 0.1–0.2 cores на 2–3 секунди	тимчасове збільшення на 50–100 MB	10–50 MB для завантаження образу	залежить від розміру образу
При видаленні обч. задачі	пік 0.05–0.1 cores на 1–2 секунди	тимчасове збільшення на 20–50 MB	10–50 MB для вивантаження образу	мінімальне навантаження
Постійні витрати на контроль (kubelet)	0.1–0.2 cores	100–200 MB	~1 MB/s для метрик та логів	1–2 GB
Додаткові витрати на моніторинг	0.1–0.2 cores	50–100 MB	залежить від частоти збору метрик	залежить від кількості збору метрик

Дані, наведені в таблиці 4.10 показують, що при інсталяції та видаленні обчислювальної задачі відбувається зміна навантаження на сервері, а також потенційно це навантаження пікове. Серверу слід мати в запасі додаткові ресурси для можливості додаткового екстремального навантаження, щоб з постійною якістю продовжувати обслуговувати вже наявні задачі.

Зміна навантаження відіграє немалу роль в зношенні обладнання, зокрема, порівняно з постійним високим навантаженням. При постійному піковому навантаженні на сервері стала висока температура компонентів, стабільне споживання енергії, передбачуване навантаження на системи охолодження, компоненти працюють в стабільному тепловому режимі.

Вплив частих пікових навантажень сприяє швидшому зношенню компонентів обладнання через різкі температурні коливання, через що відбуваються термічні цикли розширення/стиснення компонентів, відповідно, спостерігається нестабільне енергоспоживання через часті зміни швидкості вентиляторів.

Роблячи висновок, сервер швидше зношується при частих пікових навантаженнях, ніж при постійній роботі на високому рівні навантаження. Термічні цикли викликають мікротріщини в паяних з'єднаннях, призводять до деградації термопасти, спричиняють механічний стрес компонентів. Додатково, часті зміни швидкості вентиляторів зменшують їх ресурс, нестабільне енергоспоживання впливає на блоки живлення, різкі зміни навантаження створюють додаткове механічне напруження.

Стосовно електричного навантаження, через пікові струми при різких змінах навантаження та нестабільну напругу в компонентах спостерігається підвищене споживання енергії при частих змінах стану [117], [118].

За оцінками виробників та досліджень:

- постійне високе навантаження зменшує термін служби на 15–20%;
- часті пікові навантаження можуть зменшити термін служби на 25–35%.

Таким чином, оптимально це запобігати піковим навантаженням та завантажувати сервери на 80%, що сприятиме подовженню терміну їх служби та попередить можливі робочі пікові навантаження, не пов'язані з інсталяцією нових задач.

Підсумовуючи вищесказане, загальний потенціал підвищення технічної ефективності на вузлах з виключно доповнювальними групами:

- продовжити строк служби обладнання шляхом зменшення пікових навантажень на 8–12%;
- зниження енергоспоживання на 10–15%.

Енергоефективність, яка буде наслідком економії серверних ресурсів, та електроенергії є однією з цілей ООН [119], що сприятиме зменшенню перевикористання наявних ресурсів та як результат – сталому розвитку.

4.8. Вдосконалення сучасного підходу до масштабування

Використання груп стійкого (доповнюваного) навантаження та передбачуваність роботи системи внаслідок аналітики минулих даних приводить до удосконалення підходу до масштабування у хмарній системі, спрямоване на зниження частоти необхідності масштабування шляхом ефективного планування навантаження на вузли на етапі розподілу задач, яке апріорі не вимагатиме масштабування, або потребує його у мінімальній кількості, що можна забезпечити методом горизонтального масштабування, не вдаючись до вертикального.

У традиційних підходах до автомасштабування хмарних ресурсів, масштабування відбувається у відповідь на зміну поточного навантаження, що призводить до частих операцій створення та видалення екземплярів обчислювальних вузлів, споживаючи додаткові ресурси для оркестрації цих процесів. Якщо ж навантаження не змінюватиметься, то і необхідності в масштабуванні не буде. Все ж, воно відбуватиметься в екстремальних випадках, коли ПЗ переживатиме незвичайні навантаження та відбуватиметься не для одної задачі, а для всієї групи.

При виділенні доповнювальних груп в години пік, спостерігатиметься зменшення навантаження на систему за рахунок зменшення кількості безпечного ресурсу на відповідних серверах (див. [Удосконалення методу автомасштабування](#)).

Числовою характеристикою зменшення кількості автомасштабувань виступає кількість доповнювальних груп. Це значення коливається від 20 до 30%. Оскільки всього автомасштабування позбутись не можна, то зменшення кількості автомасштабувань очікується до 20%.

4.9. Впровадження підходу до планування навантаження на серверах

У дисертаційній роботі запропоновано новий підхід до планування навантаження в хмарних системах, що забезпечує оптимальне формування та розподіл пакетів задач у багатосерверному середовищі. Основою запропонованого підходу є моніторинг метрик використання ресурсів та застосування методу доповнювальних навантажень. Головні особливості підходу:

1. Аналіз та моніторинг метрик використання ресурсів (CPU, RAM, пропускна здатність мережі, дискові операції) у режимі реального часу з метою визначення завантаженості окремих вузлів та прогнозування їх стану.
2. Застосування методу доповнювальних навантажень для збалансованого використання всіх видів ресурсів на вузлі шляхом підбору таких задач, які оптимально використовують незалучені ресурси, мінімізуючи їх простої.
3. Динамічне формування пакетів задач на основі аналізу їхніх ресурсних характеристик та визначення найбільш ефективної комбінації для розміщення на доступних вузлах.
4. Зниження потреби в автомасштабуванні через оптимізацію розподілу навантаження, що дозволяє більш ефективно використовувати вже доступні ресурси без необхідності залучення нових обчислювальних вузлів.
5. Формування завантаження вузлів, що не вимагає змін, тобто таких конфігурацій навантаження, які забезпечують стабільне використання ресурсів без потреби в частому перерозподілі задач чи міграції контейнерів, що додатково знижує витрати на перепланування.

Очікувані переваги:

- Підвищення ефективності використання ресурсів коштом зменшення простоїв та максимального завантаження доступних вузлів.

- Скорочення енергоспоживання завдяки зменшенню кількості активних вузлів, необхідних для виконання задач.
- Мінімізація затримок у виконанні задач завдяки зниженню необхідності їхньої міграції (перепланування) або запуску нових вузлів.
- Зменшення експлуатаційних витрат на масштабування, балансування навантаження та резервування ресурсів.

Запропонований підхід дозволяє змінити традиційну модель управління ресурсами, у якій автоматичне масштабування є основним інструментом балансування навантаження, та перейти до розпланованої оркестрації ресурсами, що ґрунтується на прогнозуванні та оптимальному формуванні завантаження вузлів.

4.10. Впровадження та апробація наукових досліджень

Результати дослідження апробовано на всеукраїнських наукових конференціях [4], [6], опубліковано у фахових українських виданнях [8], [11], [12], та у виданнях рівня SCOPUS та CEUR [7], [70]. Прийнято до друку у фахове українське видання статтю [10].

Основні ідеї методу доповнювальних навантажень впроваджені у курс «Big Data», що викладається аспірантам на кафедрі ІТТ факультету НН ІТС Київського політехнічного інституту імені Ігоря Сікорського.

ВИСНОВКИ

1. У розділі розглянуто засоби реалізації алгоритму пошуку доповнювальних навантажень [8] локально з метою перевірити якість його роботи та за потреби скористатись ним при деплої власного програмного забезпечення на хмарну систему. Все ж, планується, що даний алгоритм застосовуватиметься більше на всій хмарі, адже його ефективність тоді буде вищою.

2. Основною технологією для реалізації алгоритму пропонується взяти Kubernetes. Як локальний помічник може виступати Minicube. Kubelet контролюватиме стан на кластерах.

3. Prometheus – пропонована система для зняття метрик. Аналіз метрик слід покласти частково на Python або Go, а частково на Kubernetes. При написанні власного планувальника (Custom Scheduler) слід взяти до уваги обмеження, такі як афініті, антіафініті та толерації — характеристики прив’язаності до вузлів та подів для формування правил розгортання додаткових вузлів на етапі горизонтального масштабування, яке передбачене алгоритмом.

4. Впровадження алгоритму доповнювальних навантажень сприятиме зменшенню кількості необхідних серверів для хостингу в середньому на 9% через впровадження доповнювальних навантажень, які ефективніше використовують ресурс зменшуючи його час простою. Також на серверах, повністю присвячених доповнювальним навантаженням має відбуватись повільніший знос техніки на 8–12%. Це пов’язаною зі зведенням до мінімуму пікових навантажень та забезпеченням рівного довготривалого навантаження на вузол.

5. При запровадженні методу доповнювальних навантажень для розподілу задач на хмарі відбуватиметься зменшення витрат електроенергії на 10–15%, пов’язане також з відсутністю стрибків електроенергії через відсутність пікових навантажень та загальну сталу роботу інфраструктури. Це сприяє зменшенню негативного впливу на екологію через антропогенний фактор.

6. Метод доповнювальних навантажень дозволяє позбутись 25-35% перепланувань, що становить 9–15% всіх планувань, зводячи до мінімуму

перепланування, пов'язані з відмовою вузлів та автомасштабуванням, залишаючи основною мірою планування, пов'язані з оновленням програмного забезпечення та технічним обслуговуванням вузлі. Частина перепланувань стане плануваннями, на які витрачається трохи менше ресурсу і які на багато менше викликають пікові навантаження у мережі та на обладнанні.

7. Планування через алгоритм доповнювальних навантажень дає можливість зменшити потужність серверів-планувальників (CPU, RAM) до 15% через суттєве зменшення кількості операцій перепланування. Серверів-планувальників на кластер виділяється 3–7 штук. Найбільшу економію апаратного забезпечення та електроенергії можна спостерігати при застосуванні методу на великих кластерах, адже кількість перепланувань при збільшенні кількості вузлів у кластері зростає швидше в 1.5 рази через складнішу топологію, а час на прийняття рішень збільшується на 40–60%.

8. Запропонований алгоритм дає можливість сформувати очікувану поведінку та ввести у стан розігріву додаткові мікропослуги, аби користувачі ПЗ не відчували затримок у відповідях при різкому зростанні кількості запитів.

9. Відсоток економії серверів може варіюватись залежно від локації хмарної технології, та, зокрема, з якими задачами доводиться справлятися, чи збалансоване навантаження у денний та нічний час, чи ні. Порівняно з поточним станом справ на хмарних ресурсах та Kubernetes, в разі потреби починає розгортатись новий екземпляр мікросервісу в момент, коли поточні метрики повідомлять про це. Економія буде проявлятися у час, коли ресурс не повністю заповнений, тобто після 18 та до 9 ранку.

Енергоефективність, яка буде наслідком економії серверних ресурсів, є однією з цілей ООН [119]. Природними методами перероблювання електроенергії у зручний формат для використання, важко забезпечити потреби людства. Зменшуючи потреби в електроенергії людство наближає кращу екологічну ситуацію у світі та сприяє сталому та передбачуванішому майбутньому.

Елементи розділу опубліковані в статтях [11] [12].

ЗАГАЛЬНІ ВИСНОВКИ

У дисертаційній роботі розв'язано актуальну науково-практичну задачу підвищення ресурсо- та енергоефективності та продуктивності обслуговування навантаження багатосерверної системи при збереженні вимог щодо доступності системи шляхом застосування методу доповнювального навантаження.

В дисертаційній роботі отримані такі теоретичні результати:

1. Виділено основні критерії, які впливають на розміщення задач у багатосерверній системі, а також сформульовано загальний алгоритм розміщення в результаті аналізу наявних рішень планування розташування обчислювальних задач на вузлах кластерів, що використовуються у лідерів поміж провайдерів логістики у хмарних технологіях, а саме AWS, Azure, GCP, та Kubernetes.

2. Виявлено декілька недоліків у плануванні розподілу задач у вищезгаданих провайдерів: віддається центральна роль параметрам базового (request) та максимального (limit) завантаження CPU та RAM (а також опційно додаткових параметрів) сервера, що спричиняється мікропослугою. Ці параметри впливають на розміщення задачі, а також на її автомасштабування та вилучення з вузла. Останнє можливе в разі перевищення її максимального завантаження, або перевантаження сервера. Після вилучення, нема гарантій, що задача буде швидко заново розміщена, зокрема, якщо спостерігається високе завантаження хмари.

3. Запропоновано нову математичну модель розподілу навантажень між вузлами багатосерверної системи, яка оперує групами процесів з постійними навантаженнями. Вона базується на теорії нечітких ґраток, що дозволяє гнучко підійти до понять неповного навантаження на сервер та графічно представити мікропослуги та їх масштабування у вигляді графів.

4. Запропоновано метод доповнювальних навантажень для визначення оптимальних задач-доповнень, які разом сформують групу завдань сталого ресурсоспоживання, що забезпечує мінімізацію сумарної кількості необхідних ресурсів в багатосерверній інфраструктурі. Основна ідея методу полягає в реорганізації хмарної інфраструктури таким чином, щоб програмне забезпечення розподілялось групами стабільного

навантаження, а не окремими задачами з різноманітним навантаженням. Це сприятиме передбачуваному стабільному навантаженню на все апаратне забезпечення (АЗ) хмари.

5. Запропоновано два удосконалені методи автомасштабування виконуваних завдань, що дозволяють зменшити необхідність їх масштабування, внаслідок часткового або повного завантаження вузлів групами обчислювальних задач сталого ресурсоспоживання. Вони працюють на основі методу доповнювальних навантажень з відповідними налаштуваннями при плануванні розподілу задач у багатосерверній системі. Беручи до уваги історичні дані, отримуємо можливість передбачити кількість та якість навантаження на серверну систему та, підбрати оптимальні комбінації груп задач, щоб розподілити навантаження, мінімізуючи потребу в непередбачуваному масштабуванні.

6. Запропоновано підхід до планування навантаження, який на основі моніторингу метрик та методу доповнювальних навантажень оптимально формує та розподіляє пакети задач у багатосерверній інфраструктурі з мінімальною потребою в автомасштабуванні, що сприяє зменшенню витрат на функціонування системи.

Практичні результати отримані в дисертаційній роботі:

1. Досліджено різноманітні типи обчислювальних задач та пропорції, в яких вони представлені у середовищі найпоширеніших хмарних провайдерів. Це дало змогу оцінити їх придатність до групування заради забезпечення сумарного сталого довготривалого використання обчислювального ресурсу.

2. Представлено алгоритм пошуку доповнювальних навантажень у вигляді блок-схеми та коду, що демонструє використання запропонованих методів та підходу загалом. Його застосування дає можливість зменшити технічні вимоги до CPU та RAM серверів-планувальників до 15% в разі заповнення вузлів кластера групами задач доповнювального завантаження. Це можливо через зменшення кількості задач для планування, що пов'язане з мінімізацією перепланувань, спричинених вилученням справних завдань із вузлів. До таких вилучень призводить перевищення запиту дозволених ресурсів на завдання чи перевантаження обчислювальних вузлів. Сумарно

це складає 25–35% всіх причин вилучення завдань, що становить 9–15% від усіх задач планувальника в разі планування тільки доповнювальних груп задач, та 2,1– 4,5% в разі середньостатистичних задач, 30% яких належать доповнювальним групам.

3. Спостерігається можливість зменшення витрат на апаратне забезпечення в багатосерверній системі на 8–12% завдяки сприянню меншому зносу техніки завдяки плануванню через методу доповнювальних навантажень, а також підвищення енергоефективності послуг хостингу на 10–15% завдяки мінімізації перепадів струму в мережі через сталий рівень завантаженості серверів на вузлах з повним використанням доповнювальних завдань.

4. Спостерігається зменшення потреб в автомасштабуванні до 20% оскільки кількість доповнювальних груп складає 20–30%. На вузлах, де частково чи повністю будуть розміщені доповнювальні групи, можна зменшити кількість запасного ресурсу headroom відповідно до пропорції присутності доповнювальних груп на обчислювальних вузлах. Пропорція буде ефективною щодо кількості запасного ресурсу, якщо доповнювальні групи складатимуть 20–24% залежно від ключового ресурсу, а отже headroom можна зменшити на той самий відсоток, але все ж слід врахувати мінімальний безпечний інтервал.

5. Можливість зменшити кількість технічного забезпечення на 8%, 10% та 17% відповідно, що сумарно становить понад 9% з врахуванням частки присутності залежно від обраного ключового критерію: CPU, RAM чи мережевий ресурс. Це можливо на серверах, повністю завантажених доповнювальними групами.

6. Збільшення пропускної можливості багатосерверної системи в години пік, яке складатиме 5–15% економії місця на 10–15% відсотках серверів при виділенні відповідної кількості серверів під доповнювальні групи.

7. Основні ідеї методу доповнювальних навантажень впроваджені у курс «Big Data», що викладається аспірантам НН ІТС на кафедрі ІТТ.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

- [1] «Що таке Kubernetes?», Kubernetes. Дата звернення: 22, Жовтень 2024. [Online]. Доступний у: <https://kubernetes.io/uk/docs/concepts/overview/what-is-kubernetes/>
- [2] Aspire Systems, «Microservices Architecture: The Foundation of Cloud-Native Applications», Software Engineering. Дата звернення: 14, Липень 2024. [Online]. Доступний у: <https://blog.aspiresys.com/software-product-engineering/microservices-architecture-the-foundation-of-cloud-native-applications/>
- [3] Y. Sharma, «Key Strategies for Implementing AWS Network Load Balancer», DEV Community. Дата звернення: 29, Грудень 2023. [Online]. Доступний у: <https://dev.to/aws-builders/key-strategies-for-implementing-aws-network-load-balancer-35fc>
- [4] О. Dmytrenko і М. Skulysh, «Handling with a Microservice Failure and Adopting Retries», представлена на Pan-Ukrainian science-practical conference of students, postgraduates and young scientists «Theoretical and Applied Problems of Physics, Mathematics and Informatics», Kyiv, Ukraine: ФІЗТЕХ, Чер 2022, с. 194–196. [Online]. Доступний у: <https://drive.google.com/file/d/1MbM9YqnSitNNdCp0CZ1yguDHweWth3TK/view>
- [5] О. Dmytrenko і М. Skulysh, «Fault Tolerance Redundancy Methods for IoT Devices», *Infocommunication Comput. Technol.*, вип. 2(04), вип. University «Ukraine», с. 59–65, Груд 2022.
- [6] О. Дмитренко, «Мікросервісна архітектура та її задачі на прикладах з реального життя», в *Збірник матеріалів Міжнародної науково-технічної конференції «Перспективи телекомунікацій»*, Київ: КПІ ім. Ігоря Сікорського, Квіт 2023, с. 385. Дата звернення: 30, Квітень 2024. [Online]. Доступний у: <http://conferenc.its.kpi.ua/proc/article/view/281911>
- [7] О. Dmytrenko і М. Skulysh, «Method of Grouping Complementary Microservices Using Fuzzy Lattice Theory», представлена на International Conference on Applied Innovations in IT (ICAИТ), E. Siemens, Ред., в 1, vol. 12. Koethen, Germany: Anhalt University of Applied Sciences Bernburg / Koethen / Dessau, Бер 2024, с. 11–18. doi: 10.25673/115636.
- [8] О. А. Дмитренко і М. А. Скулиш, «Визначення мікропроцесорних груп для ефективного використання процесорних потужностей», *Проблеми Програмування*, вип. 2–3, с. 215–222, Груд 2024, doi: 10.15407/pp2024.02-03.215.

- [9] O. Dmytrenko, M. Skulysh, i L. Globa, «Microservice Complimentary Groups Determination Algorithm for the Effective Resource Usage», в *Proceedings of the 14th International Scientific and Practical Programming Conference (UkrPROG 2024)*, I. Sinitsyn i P. Andon, Ред., в CEUR Workshop Proceedings, vol. 3806. Kyiv, Ukraine: CEUR, Трав 2024, с. 180–201. Дата звернення: 28, Грудень 2024. [Online]. Доступний у:
https://ceur-ws.org/Vol-3806/#S_55_Dmytrenko_Skulysh_Globa
- [10] О. Дмитренко і М. Скулиш, «Групування мікропослуг для використання неповно залучених ресурсів», *Сучасні Інформаційні Технології*, вип. 2024, вип. 1(3), с. 78–85, Бер 2025, doi: <https://doi.org/10.17721/AIT.2024.1.08>.
- [11] М. А. Скулиш і О. А. Дмитренко, «Метод організації мікросервісів на серверних групах Kubernetes», *Sci. Notes Taurida Natl. VI Vernadsky Univ. Ser. Tech. Sci.*, вип. 1, вип. 5, с. 291–297, Груд 2024, doi: [10.32782/2663-5941/2024.5.1/41](https://doi.org/10.32782/2663-5941/2024.5.1/41).
- [12] О. А. Дмитренко і М. А. Скулиш, «Методи збору інформації та реалізації алгоритму доповнювальних навантажень», *Системи Управління Навігації Та Зв'язку Збірник Наукових Праць*, вип. 4, вип. 78, с. 56–59, Лис 2024, doi: [10.26906/SUNZ.2024.4.056](https://doi.org/10.26906/SUNZ.2024.4.056).
- [13] А. І. Блозва, Ю. В. Матус, і В. В. Смолій, *Комп'ютерні мережі*. Київ: Компрінт, 2017.
- [14] «Difference between 4G and 5G network architecture». Дата звернення: 07, Березень 2025. [Online]. Доступний у:
<https://telcomaglobal.com/p/difference-between-4g-and-5g-network-architecture>
- [15] Л. С. Глоба, *Розробка інформаційних ресурсів та систем*, вип. 1, 2 вип. Київ: Політехніка, 2013. Дата звернення: 01, Березень 2025. [Online]. Доступний у: https://duikt.edu.ua/uploads/l_1690_29298415.pdf
- [16] S. Robinson, J. Montgomery, i K. Marko, «What is Cloud Infrastructure? | Definition from TechTarget», Search Cloud Computing. Дата звернення: 06, Лютий 2025. [Online]. Доступний у:
<https://www.techtarget.com/searchcloudcomputing/definition/cloud-infrastructure>
- [17] L. Shen, S. Qian, T. Zhai, L. Li, i Z. Li, «Research on cloud computing high-density data center infrastructure and environment matching technology», *MATEC Web Conf.*, вип. 336, с. 02028, Січ 2021, doi: [10.1051/matecconf/202133602028](https://doi.org/10.1051/matecconf/202133602028).
- [18] B. Gurumurthy, «A Paradigm Shift towards On-Premise Modern Data Center Infrastructure for Agility and Scalability in Resource Provisioning», *Int. J. Adv. Trends Comput. Sci. Eng.*, вип. 9, с. 4964–4971, Сер 2020, doi: [10.30605/ijatce.v9i9.4964-4971](https://doi.org/10.30605/ijatce.v9i9.4964-4971)

- 10.30534/ijatcse/2020/111942020.
- [19] A. Goller, «Scaling up vs scaling out your security segmentation». Дата звернення: 16, Листопад 2024. [Online]. Доступний у: <https://www.linkedin.com/pulse/scaling-up-vs-out-your-security-segmentation-a-lexander-goller/>
 - [20] Kubernetes Team, «Kubernetes Scheduler». Kubernetes Documentation, 14, Грудень 2023. [Online]. Доступний у: <https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/>
 - [21] A. P. Singh, «Design a Distributed Job Scheduler - System Design», AlgoMaster Newsletter. Дата звернення: 06, Лютий 2025. [Online]. Доступний у: <https://blog.algomaster.io/p/design-a-distributed-job-scheduler>
 - [22] Kubernetes Team, «Assigning Pods to Nodes», Kubernetes. Дата звернення: 23, Листопад 2024. [Online]. Доступний у: <https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/>
 - [23] Kubernetes Team, «Taints and Tolerations», Kubernetes. Дата звернення: 23, Листопад 2024. [Online]. Доступний у: <https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/>
 - [24] «Optimize Live | Autonomous Kubernetes Rightsizing», stormforge.io. Дата звернення: 22, Листопад 2024. [Online]. Доступний у: <https://stormforge.io/optimize-live/>
 - [25] Amazon Team, «Optimizing your AWS Batch architecture for scale with observability dashboards | AWS HPC Blog», Amazon Documentation. Дата звернення: 06, Лютий 2025. [Online]. Доступний у: <https://aws.amazon.com/blogs/hpc/optimizing-aws-batch-with-observability-dashboards/>
 - [26] «Secure Cross-Cluster Communication in EKS with VPC Lattice and Pod Identity IAM Session Tags | Containers». Дата звернення: 07, Лютий 2025. [Online]. Доступний у: <https://aws.amazon.com/blogs/containers/secure-cross-cluster-communication-in-eks-with-vpc-lattice-and-pod-identity-iam-session-tags/>
 - [27] Amazon Team, «When to Use Spot Instances - Overview of Amazon EC2 Spot Instances», Amazon Documentation. Дата звернення: 06, Лютий 2025. [Online]. Доступний у: <https://docs.aws.amazon.com/whitepapers/latest/cost-optimization-leveraging-ec2-spot-instances/when-to-use-spot-instances.html>
 - [28] Istio Team, «Що таке Istio?», Istio. Дата звернення: 24, Листопад 2024. [Online]. Доступний у: <https://istio.io/latest/uk/docs/overview/what-is-istio/>
 - [29] L. B. Nagy, «Istio ingress controller as an API gateway». [Online]. Доступний у: <https://banzaicloud.com/blog/backyards-api-gateway/>

- [30] Kubernetes Team, «Sidecar Containers», Kubernetes. Дата звернення: 05, Лютий 2025. [Online]. Доступний у:
<https://kubernetes.io/docs/concepts/workloads/pods/sidecar-containers/>
- [31] R. Bhattacharya, Y. Gao, і T. Wood, «Dynamically Balancing Load with Overload Control for Microservices», *ACM Trans. Auton. Adapt. Syst.*, вип. 19, вип. 4, с. 22:1–22:23, Лип 2024, doi: 10.1145/3676167.
- [32] gregoriopalama, «Two feet in a shoe: more than one container in a single Pod», Codemotion Magazine. Дата звернення: 24, Листопад 2024. [Online].
Доступний у:
<https://www.codemotion.com/magazine/devops/cloud/two-feet-in-a-shoe-more-than-one-container-in-a-single-pod/>
- [33] C. J. Walter і N. Suri, «The customizable fault/error model for dependable distributed systems», *Theor. Comput. Sci.*, вип. 290, вип. 2, с. 1223–1251, Січ 2003, doi: 10.1016/S0304-3975(01)00203-1.
- [34] V. Haranadh і D. Ch, «Enhancing Reliability and Fault Tolerance in IoT», в *2020 International Conference on Artificial Intelligence and Signal Processing (AISP)*, Січ 2020, с. 6. doi: 10.1109/AISP48273.2020.9073174.
- [35] R. Peng, Q. Zhai, і J. Yang, «Reliability Modelling and Optimization of Warm Standby Systems», Січ 2021, doi: 10.1007/978-981-16–1792-8.
- [36] Y. Cao, L. Ma, і S. Du, «Design and Analysis of Double One Out of Two with a Hot Standby Safety Redundant Structure», *Chin. J. Electron.*, вип. 29, с. 586–594, Трав 2020, doi: 10.1049/cje.2020.03.015.
- [37] J.-H. Kuo *et al.*, *An Evaluation of the Virtual Router Redundancy Protocol Extension with Load Balancing*. 2006. doi: 10.1109/PRDC.2005.16.
- [38] R. Nur, Z. Saharuna, I. Irmawati, I. Widi Widayat, і R. Wahyuni, «Gateway Redundancy Using Common Address Redundancy Protocol (CARP)», *IJITEE Int. J. Inf. Technol. Electr. Eng.*, вип. 2, с. 71, Лют 2019, doi: 10.22146/ijitee.43701.
- [39] M. Richards і N. Ford, *Fundamentals of Software Architecture: An Engineering Approach*, 1st edition. Beijing Boston Farnham Sebastopol Tokyo: O'Reilly Media, 2020. [Online]. Доступний у:
<https://www.developertoarchitect.com/downloads/architecture-styles-worksheet.pdf>
- [40] E. A. Akkoyunlu, K. Ekanadham, і R. V. Huber, «Some constraints and tradeoffs in the design of network communications», в *Proceedings of the fifth ACM symposium on Operating systems principles*, в SOSP '75. New York, NY, USA: Association for Computing Machinery, 1975, с. 67–74. doi: 10.1145/800213.806523.
- [41] M. J. Fischer, N. A. Lynch, і M. S. Paterson, «Impossibility of distributed

- consensus with one faulty process», *J ACM*, вип. 32, вип. 2, с. 374–382, 1985, doi: 10.1145/3149.214121.
- [42] J. Goldberg, «The SIFT computer and its development», в *Digital Avionics Systems Conference*, St. Louis, MO, U.S.A.: American Institute of Aeronautics and Astronautics, Лис 1981. doi: 10.2514/6.1981-2278.
- [43] К. Babitz, «The Strange Story of the Paxos Algorithm», Medium. Дата звернення: 05, Листопад 2024. [Online]. Доступний у: <https://towardsdatascience.com/the-strange-story-of-the-paxos-algorithm-52a9f3f53ae0>
- [44] L. Lamport, «The part-time parliament», *ACM Trans. Comput. Syst.*, вип. 16, вип. 2, с. 133–169, Трав 1998, doi: 10.1145/279227.279229.
- [45] Amazon Team, «Control traffic to your AWS resources using security groups - Amazon Virtual Private Cloud», Amazon Documentation. Дата звернення: 24, Лютий 2025. [Online]. Доступний у: <https://docs.aws.amazon.com/vpc/latest/userguide/vpc-security-groups.html>
- [46] К. Ruban, «Cloud vs Colocation: What is the difference?», Macquarie Data Centres. Дата звернення: 24, Лютий 2025. [Online]. Доступний у: <https://macquariedatacentres.com/blog/cloud-vs-colocation/>
- [47] Amazon Team, «Auto Scaling groups - Amazon EC2 Auto Scaling», Amazon Documentation. Дата звернення: 24, Лютий 2025. [Online]. Доступний у: <https://docs.aws.amazon.com/autoscaling/ec2/userguide/auto-scaling-groups.html>
- [48] С. Cloudflare, «What is a content delivery network (CDN)? | How do CDNs work?» Дата звернення: 13, Листопад 2024. [Online]. Доступний у: <https://www.cloudflare.com/learning/cdn/what-is-a-cdn/>
- [49] «The vertical-pod-autoscaler module | Deckhouse». Дата звернення: 15, Листопад 2024. [Online]. Доступний у: <https://deckhouse.io/products/kubernetes-platform/documentation/v1/modules/vertical-pod-autoscaler/>
- [50] L. Patrão, «VMware Hot Add and Hot Plug», 2024, с. 493–509. doi: 10.1007/979-8-8688-0208-9_25.
- [51] «Enabling Hot-Add by default? /cc @gabvirtualworld», Yellow Bricks. Дата звернення: 12, Листопад 2024. [Online]. Доступний у: <https://www.yellow-bricks.com/2012/01/16/enabling-hot-add-by-default-cc-gab-virtualworld/>
- [52] «How to hot-add RAM and hot-plug vCPUs to your vSphere VMs in different environments», StarWind Blog. Дата звернення: 12, Листопад 2024. [Online]. Доступний у: <https://www.starwindsoftware.com/blog/hot-add-ram-hot-plug-vcpus-vsphere-v>

- ms-different-environments/
- [53] «Vertical Autoscaling | Cloud Dataflow», Google Cloud. Дата звернення: 12, Листопад 2024. [Online]. Доступний у:
<https://cloud.google.com/dataflow/docs/vertical-autoscaling>
 - [54] O. Rolik i V. Volkov, «Method of horizontal pod scaling in Kubernetes to omit overregulation», *Inf. Comput. Intell. Syst.*, вип. 5, с. 55–67, 2024.
 - [55] A. Muppada, «✳ A Hands-On Guide to Kubernetes Horizontal & Vertical Pod Autoscalers 🛠», Medium. Дата звернення: 12, Листопад 2024. [Online]. Доступний у:
<https://medium.com/@muppadaanvesh/a-hands-on-guide-to-kubernetes-horizontal-vertical-pod-autoscalers-%EF%B8%8F-58903382ef71>
 - [56] «Kubernetes Autoscaling: Horizontal and Vertical explained». Дата звернення: 23, Жовтень 2024. [Online]. Доступний у:
<https://www.fullstack.com/knowledge-hub/blogs/autoscaling-in-kubernetes>
 - [57] Kubernetes Team, «Pod Quality of Service Classes», Kubernetes Documentation. Дата звернення: 23, Лютий 2025. [Online]. Доступний у:
<https://kubernetes.io/docs/concepts/workloads/pods/pod-qos/>
 - [58] D. Polencic, «Memory requests and limits in Kubernetes», Medium. Дата звернення: 13, Січень 2025. [Online]. Доступний у:
<https://itnext.io/memory-requests-and-limits-in-kubernetes-1c9cd573b3ab>
 - [59] Kubernetes Team, «Node-pressure Eviction», Kubernetes Documentation. Дата звернення: 23, Лютий 2025. [Online]. Доступний у:
<https://kubernetes.io/docs/concepts/scheduling-eviction/node-pressure-eviction/>
 - [60] О. Темнікова, *Дискретна математика. Конспект лекцій (Частина 1)*, вип. 1. Київ: КПІ ім. Ігоря Сікорського, 2021. [Online]. Доступний у:
<https://ela.kpi.ua/server/api/core/bitstreams/990893b6-f853-408a-8476-d3dd7c89d2a1/content>
 - [61] G. A. Gratzner, *General lattice theory*. в Pure and applied mathematics : a series of monographs and textbooks, no. 75. New York: Academic Press, 1978.
 - [62] N. Ajmal i K. V. Thomas, «Fuzzy lattices», *Inf. Sci.*, вип. 79, вип. 3–4, с. 271–291, Лип 1994, doi: 10.1016/0020-0255(94)90124-4.
 - [63] О. Темнікова, *МАТЕМАТИЧНА ЛОГІКА ТА ТЕОРІЯ АЛГОРИТМІВ*. Київ: КПІ ім. Ігоря Сікорського, 2021. [Online]. Доступний у:
<https://ela.kpi.ua/server/api/core/bitstreams/90cfd1e3-1a98-4370-95f3-bd1d677d0e7c/content>
 - [64] Kubernetes Team, «Multi-tenancy in Kubernetes», Kubernetes. Дата звернення: 23, Грудень 2023. [Online]. Доступний у:
<https://kubernetes.io/docs/concepts/security/multi-tenancy/>
 - [65] «A review of in-memory computing for machine learning: architectures,

- options», *Int. J. Web Inf. Syst.*, Груд 2023, doi: 10.1108/IJWIS-08-2023-0131.
- [66] F. Wilhelmi, D. Salami, G. Fontanesi, L. Galati Giordano, і M. Kasslin, *AI/ML-based Load Prediction in IEEE 802.11 Enterprise Networks*. 2023.
- [67] A. Meir, «Does Location Matter In Cloud Computing?», Ridge Cloud. Дата звернення: 23, Грудень 2023. [Online]. Доступний у:
<https://www.ridge.co/blog/location-in-cloud-computing/>
- [68] Amazon Team, «AWS Global Infrastructure». 22, Грудень 2023. [Online]. Доступний у:
https://aws.amazon.com/about-aws/global-infrastructure/?nc1=h_ls
- [69] RenovaCloud, «Stateful vs. Stateless Architecture Overview», RenovaCloud. Дата звернення: 17, Листопад 2024. [Online]. Доступний у:
<https://renovacloud.com/en/stateful-vs-stateless-architecture-overview/>
- [70] O. Dmytrenko, M. Skulysh, і L. Globa, «Microservice Complimentary Groups Determination Algorithm for the Effective Resource Usage», *CEUR Workshop Proc. Forthcom.*, вип. Scientific and Practical Programming-UkrPROG 2024, с. 20.
- [71] P. Salot, «A Survey of Various Scheduling Algorithm in Cloud Computing Environment», *Int. J. Res. Eng. Technol.*, вип. 02, вип. 02, с. 131–135, Лют 2013, doi: 10.15623/ijret.2013.0202008.
- [72] H. Liu, Y. Li, і S. Wang, «Request Scheduling Combined with Load Balancing in Mobile Edge Computing», *IEEE Internet Things J.*, с. 1–1, 2022, doi: 10.1109/IJOT.2022.3176631.
- [73] R. L. Collins і L. P. Carloni, «Flexible filters: load balancing through backpressure for stream programs», в *Proceedings of the seventh ACM international conference on Embedded software - EMSOFT '09*, Grenoble, France: ACM Press, 2009, с. 205. doi: 10.1145/1629335.1629363.
- [74] S. Dutta, «MySQL vs DynamoDB: Comparing Relational and NoSQL Database Solutions», Sprinkle. Дата звернення: 14, Липень 2024. [Online]. Доступний у:
<https://www.sprinkledata.com/blogs/mysql-vs-dynamodb-a-comprehensive-comparison>
- [75] G. Weintraub, *Dynamo and BigTable — Review and comparison*. 2014, с. 5. doi: 10.1109/EEEI.2014.7005771.
- [76] «Amazon DynamoDB Customers | AWS», Amazon Web Services, Inc. Дата звернення: 14, Липень 2024. [Online]. Доступний у:
<https://aws.amazon.com/dynamodb/customers/>
- [77] «Database Migration - AWS Database Migration Service - AWS», Amazon Web Services, Inc. Дата звернення: 14, Липень 2024. [Online]. Доступний у:
<https://aws.amazon.com/dms/>

- [78] R. E. Bryant i D. R. O'Hallaron, *Computer systems: a programmer's perspective*, 2. ed., вип. Chapter 9: VM as a Tool for Caching. Boston, Mass.: Prentice Hall, 2011. [Online]. Доступний у:
<http://54.186.36.238/Computer%20Systems%20-%20A%20Programmer%27s%20Persp.%202nd%20ed.%20-%20R.%20Bryant%2C%20D.%20O%27Hallaron%20%28Pearson%2C%202010%29%20BBS.pdf>
- [79] A. Belgacem, «Dynamic resource allocation in cloud computing: analysis and taxonomies», *Computing*, вип. 104, вип. 3, с. 681–710, Бер 2022, doi: 10.1007/s00607-021-01045-2.
- [80] «AWS vs Azure vs GCP cloud comparison: Databases». Дата звернення: 14, Липень 2024. [Online]. Доступний у:
<https://www.pluralsight.com/resources/blog/cloud/aws-vs-azure-vs-gcp-cloud-comparison-databases>
- [81] «Vertical Pod autoscaling | Google Kubernetes Engine (GKE)», Google Cloud. Дата звернення: 14, Липень 2024. [Online]. Доступний у:
<https://cloud.google.com/kubernetes-engine/docs/concepts/verticalpodautoscaler>
- [82] S. J. Bigelow, «What is edge computing? Everything you need to know», Search Data Accelerator. Дата звернення: 07, Червень 2024. [Online]. Доступний у:
<https://www.techtarget.com/searchdatacenter/definition/edge-computing>
- [83] X. Cao, F. Wang, J. Xu, R. Zhang, i S. Cui, «Joint Computation and Communication Cooperation for Energy-Efficient Mobile Edge Computing», *IEEE Internet Things J.*, вип. 6, вип. 3, с. 4188–4200, Чер 2019, doi: 10.1109/IIOT.2018.2875246.
- [84] B. Gregg, *Systems Performance*, 2nd edition. Pearson, 2020.
- [85] O. Al-Debagy i P. Martinek, «A Comparative Review of Microservices and Monolithic Architectures», в *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, Лис 2018, с. 000149–000154. doi: 10.1109/CINTI.2018.8928192.
- [86] Codecademy Team, «Normalization», Codecademy. Дата звернення: 18, Липень 2024. [Online]. Доступний у:
<https://www.codecademy.com/article/normalization>
- [87] «Z-score standardization - Feature Engineering Made Easy [Book]». Дата звернення: 19, Липень 2024. [Online]. Доступний у:
<https://www.oreilly.com/library/view/feature-engineering-made/9781787287600/23a14672-9d83-4d43-a85c-b3117d7e1c9e.xhtml>
- [88] S. Salvador i P. Chan, *Toward Accurate Dynamic Time Warping in Linear Time and Space*, вип. 11. 2004, с. 80. [Online]. Доступний у:

- <https://cs.fit.edu/~pkc/papers/tm04.pdf>
- [89] Y. Li i H. Wu, «A Clustering Method Based on K-Means Algorithm», *Phys. Procedia*, вип. 25, с. 1104–1109, Груд 2012, doi: 10.1016/j.phpro.2012.03.206.
 - [90] S. Chakraborty, «Analysis and Study of Incremental DBSCAN Clustering Algorithm», *Int. J. Enterp. Comput. Bus. Syst.*, вип. 1, Лип 2011.
 - [91] M. Ester, H.-P. Kriegel, J. Sander, i X. Xu, «A density-based algorithm for discovering clusters in large spatial databases with noise», в *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, в KDD'96. Portland, Oregon: AAAI Press, 1996, с. 226–231.
 - [92] I. T. Jolliffe, Ред., «Graphical Representation of Data Using Principal Components», в *Principal Component Analysis*, New York, NY: Springer, 2002, с. 78–110. doi: 10.1007/0-387-22440-8_5.
 - [93] I. T. Jolliffe, Ред., «Principal Components as a Small Number of Interpretable Variables: Some Examples», в *Principal Component Analysis*, New York, NY: Springer, 2002, с. 63–77. doi: 10.1007/0-387-22440-8_4.
 - [94] I. T. Jolliffe, Ред., «Choosing a Subset of Principal Components or Variables», в *Principal Component Analysis*, New York, NY: Springer, 2002, с. 111–149. doi: 10.1007/0-387-22440-8_6.
 - [95] A. Zieffler i Catalysts for Change, *Statistical Thinking: A Simulation Approach to Modeling Uncertainty (UM STAT 216 edition)*, 4.2. Minneapolis: MN: Catalyst Press, 2019. Дата звернення: 21, Жовтень 2024. [Online].
Доступний у:
https://bookdown.org/frederick_peck/textbook/the-mean-and-standard-deviation.html
 - [96] G. Levitin, L. Xing, i Y. Dai, «Cold vs. hot standby mission operation cost minimization for 1-out-of-N systems», *Eur. J. Oper. Res.*, вип. 234, вип. 1, с. 155–162, Квіт 2014, doi: 10.1016/j.ejor.2013.10.051.
 - [97] V. Cacchiani, M. Iori, A. Locatelli, i S. Martello, «Knapsack problems — An overview of recent advances. Part II: Multiple, multidimensional, and quadratic knapsack problems», *Comput. Oper. Res.*, вип. 143, с. 105693, Лип 2022, doi: 10.1016/j.cor.2021.105693.
 - [98] *Docker*. [Online]. Доступний у:
<https://www.docker.com/resources/what-container/>
 - [99] S. Al-Raheym, S. C. Açan, i Ö. T. Pusatli, «Investigation Of Amazon And Google For Fault Tolerance Strategies In Cloud Computing Services», *AJIT-E Online Acad. J. Inf. Technol.*, вип. 7, вип. 23, с. 7–22, Лис 2016, doi: 10.5824/1309-1581.2016.4.001.x.
 - [100] E. R. Jamzuri, R. Analia, H. Mandala, i S. Susanto, «Cloud-Based Architecture for YOLOv3 Object Detector using gRPC and Protobuf», *J. Tek. Elektro*, вип.

Vol 14, No 1, с. 18–23, 2022.

- [101] «Що таке Prometheus? | Адміністрування серверів та cloud рішень.» Дата звернення: 22, Жовтень 2024. [Online]. Доступний у:
<https://itfb.com.ua/uk/shho-take-prometheus/>
- [102] J. Martínez, «Monitoring with Custom Metrics», Sysdig. Дата звернення: 22, Листопад 2024. [Online]. Доступний у:
<https://sysdig.com/blog/monitoring-custom-metrics/>
- [103] «The Prometheus monitoring module | Deckhouse». Дата звернення: 15, Листопад 2024. [Online]. Доступний у:
<https://deckhouse.io/products/kubernetes-platform/documentation/v1/modules/prometheus/>
- [104] Prometheus, «Overview | Prometheus». Дата звернення: 23, Жовтень 2024. [Online]. Доступний у: <https://prometheus.io/docs/introduction/overview/>
- [105] Prometheus, «Client libraries | Prometheus». Дата звернення: 22, Жовтень 2024. [Online]. Доступний у:
<https://prometheus.io/docs/instrumenting/clientlibs/>
- [106] S. N. A. Jawaddi, A. Ismail, i V. Cardellini, «Modeling and Verifying Microservice Autoscaling Using Probabilistic Model Checking», In Review, preprint, Трав 2022. doi: 10.21203/rs.3.rs-1682990/v1.
- [107] «Vista do State of the Art on Microservices Autoscaling: An Overview». Дата звернення: 14, Липень 2024. [Online]. Доступний у:
<https://sol.sbc.org.br/index.php/semish/article/view/15804/15645>
- [108] K. Zhang, D. Ou, C. Jiang, Y. Qiu, i L. Yan, «Power and Performance Evaluation of Memory-Intensive Applications», *Energies*, вип. 14, с. 4089, Лип 2021, doi: 10.3390/en14144089.
- [109] Pooja i A. Pandey, *Impact of memory intensive applications on performance of cloud virtual machine*. 2014, с. 6. doi: 10.1109/RAECS.2014.6799629.
- [110] Cloud Native Computing Foundation i The Linux Foundation, «CNCF Annual Survey 2023 | CNCF», Cloud Native Computing Foundation. Дата звернення: 21, Лютий 2025. [Online]. Доступний у:
<https://www.cncf.io/reports/cncf-annual-survey-2023/>
- [111] Storm Forge Community, «Cloud Waste Survey 2021 Findings», stormforge.io. Дата звернення: 21, Лютий 2025. [Online]. Доступний у:
<https://stormforge.io/survey-report/cloud-waste-survey-findings-thank-you/>
- [112] Q. Zhang, S. Geng, i X. Cai, «Survey on Task Scheduling Optimization Strategy under Multi-Cloud Environment», *Comput. Model. Eng. Sci.*, вип. 135, вип. 3, с. 1863–1900, 2022, doi: 10.32604/cmesci.2023.022287.
- [113] A. Hatzenbuehler, «What is Data Center Redundancy? N, N+1, 2N, 2N+1», CoreSide. Дата звернення: 02, Березень 2025. [Online]. Доступний у:

- <https://www.coresite.com/blog/data-center-redundancy-n-1-vs-2n-1>
- [114] C. Bassek, S. Pierre, і A. Quintero, «Redundancy Schemes for High Availability Computer Clusters», *J. Comput. Sci.*, вип. 2, Січ 2006, doi: 10.3844/jcssp.2006.33.47.
- [115] Z. Zhuang *et al.*, «Capacity Planning and Headroom Analysis for Taming Database Replication Latency: Experiences with LinkedIn Internet Traffic», *ICPE 2015 - Proc. 6th ACM SPEC Int. Conf. Perform. Eng.*, с. 39–50, Січ 2015, doi: 10.1145/2668930.2688054.
- [116] Amazon Team, «Cloud Compute Instances – Amazon EC2 Instance Types», Amazon Web Services, Inc. Дата звернення: 01, Березень 2025. [Online]. Доступний у: <https://aws.amazon.com/ec2/instance-types/>
- [117] J. Cho, T. Lim, і B. S. Kim, «Viability of datacenter cooling systems for energy efficiency in temperate or subtropical regions: Case study», *Energy Build.*, вип. 55, с. 189–197, Груд 2012, doi: 10.1016/j.enbuild.2012.08.012.
- [118] J. von Kistowski, J. Beckett, K.-D. Lange, H. Block, J. Arnold, і S. Kounev, *Energy Efficiency of Hierarchical Server Load Distribution Strategies*. 2015. doi: 10.1109/MASCOTS.2015.11.
- [119] United Nations Development Programme, «Цілі сталого розвитку / Goals of sustainable development», UNDP. Дата звернення: 21, Жовтень 2024. [Online]. Доступний у: <https://www.undp.org/uk/ukraine/tsili-staloho-rozvytku>

Додаток 1. Приклад симуляції даних. У доповнювальні вектори вводиться додаткова похибка

```
import numpy as np
import matplotlib.pyplot as plt

class TimeSeriesGenerator:
    def __init__(self, microservices=None):
        if microservices is None:
            self.microservices = []
        else:
            self.microservices = microservices

    # Set random seed for reproducibility
    np.random.seed(42)

    # Helper function to add noise to the data and round to integers
    def add_noise_and_round(data, noise_level=0.1):
        noisy_data = data + np.random.normal(0, noise_level, data.shape)
        return np.clip(np.round(noisy_data).astype(int), 0, 100)

    # Helper function to create gradual peaks
    def gradual_peak(length, peak_height, peak_center, peak_width):
        x = np.arange(length)
        return peak_height * np.exp(-((x - peak_center) ** 2) / (2 *
peak_width ** 2))

    # 1. Stably loaded (2 microservices)
    stable_load = np.full((2, 24), 50)
    stable_load = add_noise_and_round(stable_load, 5)

    # 2. Big peak in working hours (4 microservices)
    working_hours_peak = np.zeros((4, 24))
    for i in range(4):
        working_hours_peak[i] = gradual_peak(24, 80, 13, 4)
    working_hours_peak = add_noise_and_round(working_hours_peak, 10)
```

```

# 3. Peaks after 12 am and before 6 am (2 microservices)
night_peak = np.zeros((2, 24))
for i in range(2):
    night_peak[i] = gradual_peak(24, 70, 3, 3) + gradual_peak(24, 70,
23, 3)
night_peak = add_noise_and_round(night_peak, 10)

# 4. Peaks from 5 pm till night (2 microservices)
evening_peak = np.zeros((2, 24))
for i in range(2):
    evening_peak[i] = gradual_peak(24, 75, 20, 4)
evening_peak = add_noise_and_round(evening_peak, 10)

# 5. Random peaks (2 microservices)
random_peaks = np.zeros((2, 24))
for i in range(2):
    num_peaks = np.random.randint(3, 6)
    for _ in range(num_peaks):
        peak_center = np.random.randint(0, 24)
        peak_height = np.random.randint(50, 80)
        random_peaks[i] += gradual_peak(24, peak_height, peak_center, 2)
random_peaks = add_noise_and_round(random_peaks, 10)

# Combine all patterns
all_patterns = np.vstack((stable_load, working_hours_peak, night_peak,
evening_peak, random_peaks))

# Generate complements (opposite time load)
complements = 100 - all_patterns

# Combine original patterns and complements
microservices = np.vstack((all_patterns, complements))

# Ensure all values are integers between 0 and 100
microservices = np.clip(microservices, 0, 100).astype(int)

# Plot the data with colors for each type of microservice
plt.figure(figsize=(15, 10))
colors = ['blue', 'green', 'red', 'purple', 'orange']
labels = ['Stable', 'Working Hours Peak', 'Night Peak', 'Evening Peak',
'Random Peaks']

```

```

hours = np.arange(24)  # Generate 24 hours of data

for i, (color, label) in enumerate(zip(colors, labels)):
    if i == 0:
        plt.plot(hours, microservices[i], color=color, label=f'{label}
(Original)')
        plt.plot(hours, microservices[i + 12], color=color,
linestyle='--', label=f'{label} (Complement)')
    elif i == 1:
        for j in range(4):
            plt.plot(hours, microservices[i + j], color=color,
label=f'{label} (Original)' if j == 0 else '')
            plt.plot(hours, microservices[i + j + 12], color=color,
linestyle='--',
label=f'{label} (Complement)' if j == 0 else '')
    else:
        for j in range(2):
            plt.plot(hours, microservices[i * 2 + j + 2], color=color,
label=f'{label} (Original)' if j == 0 else '')
            plt.plot(hours, microservices[i * 2 + j + 14], color=color,
linestyle='--',
label=f'{label} (Complement)' if j == 0 else '')

plt.title('Processor Load for 24 Microservices')
plt.xlabel('Hour of Day')
plt.ylabel('Processor Load (%)')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.tight_layout()
plt.show()

# Save the data to a file
np.savetxt('microservices_load.csv', microservices, delimiter=',',
fmt='%d')
print("Data saved to 'microservices_load.csv'")

```

Додаток 2. Згенеровані доповнення до часових рядів як повноцінне доповнення

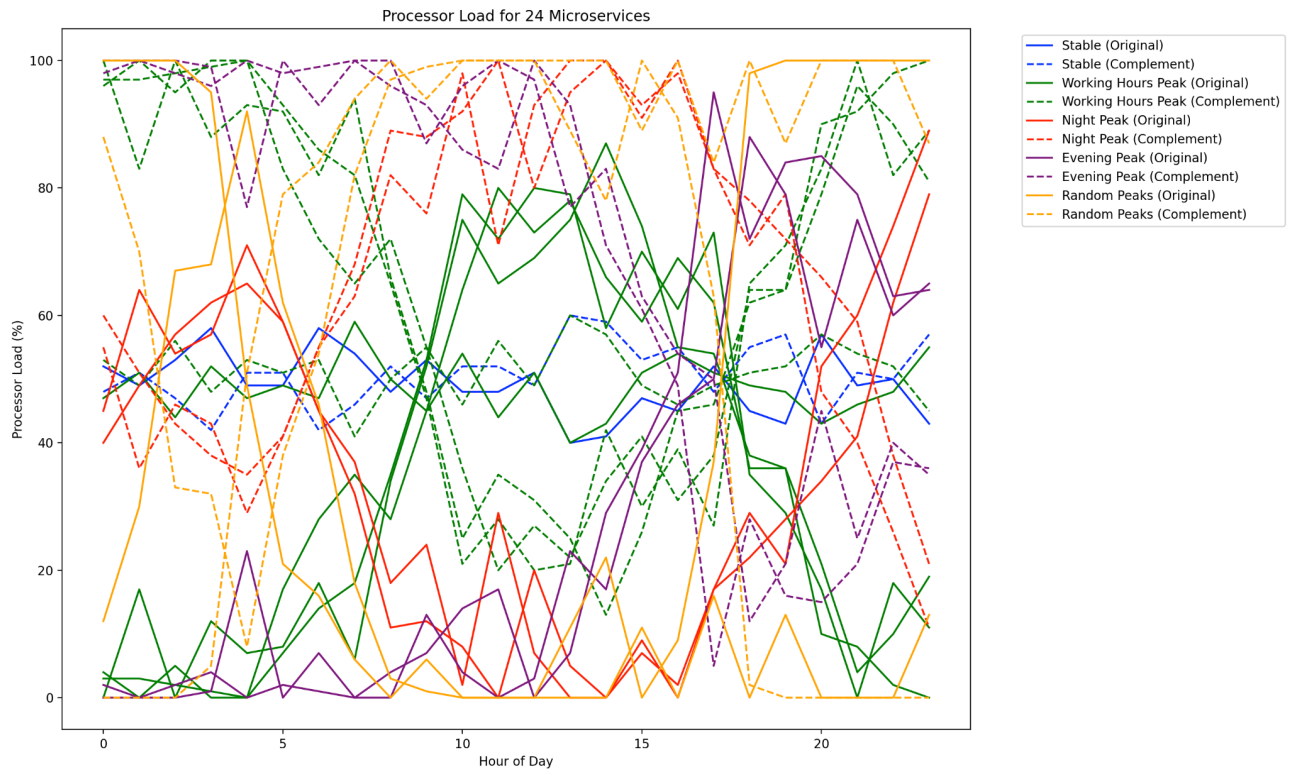


Рисунок Д2.1 — Повні доповнення до часових рядів як повноцінне доповнення

Додаток 3. Кластеризація повноцінних доповнень до часових рядів та звичайних даних

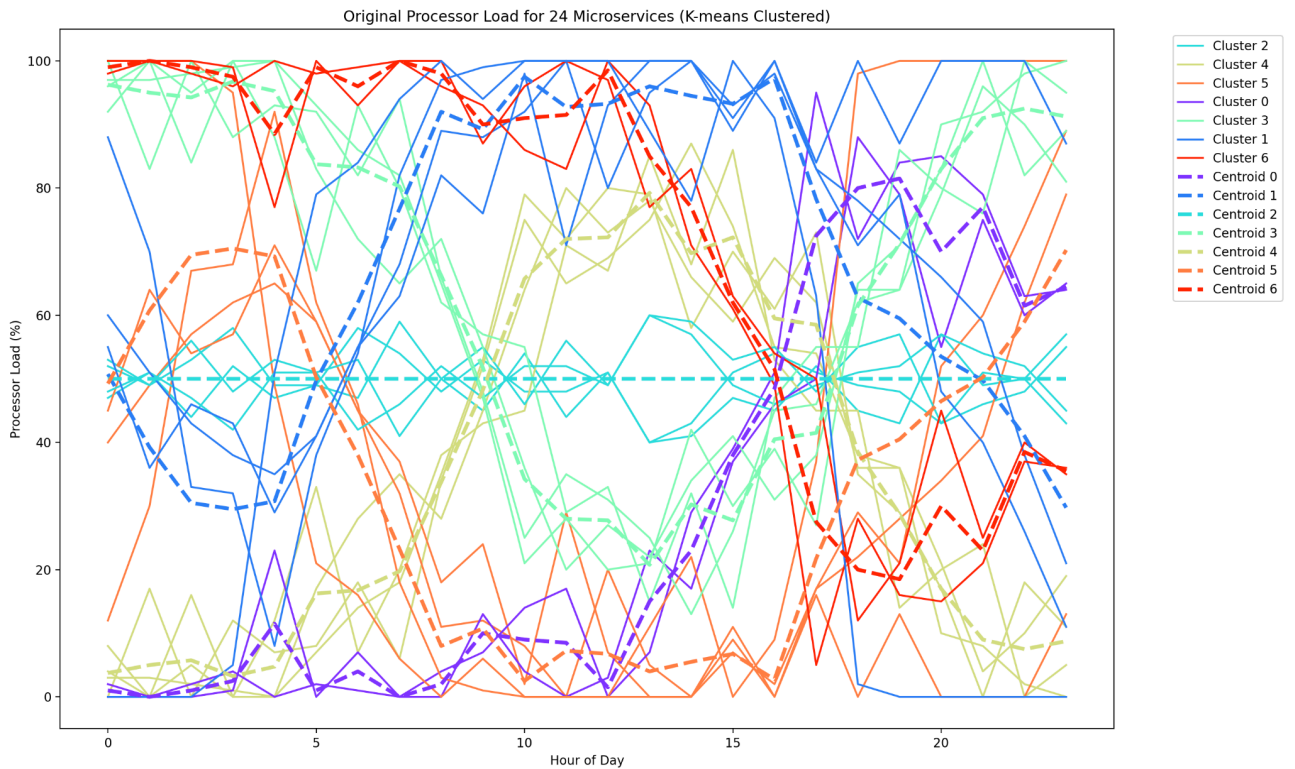


Рисунок Д3.1 — Кластеризовані повні доповнення до часових рядів

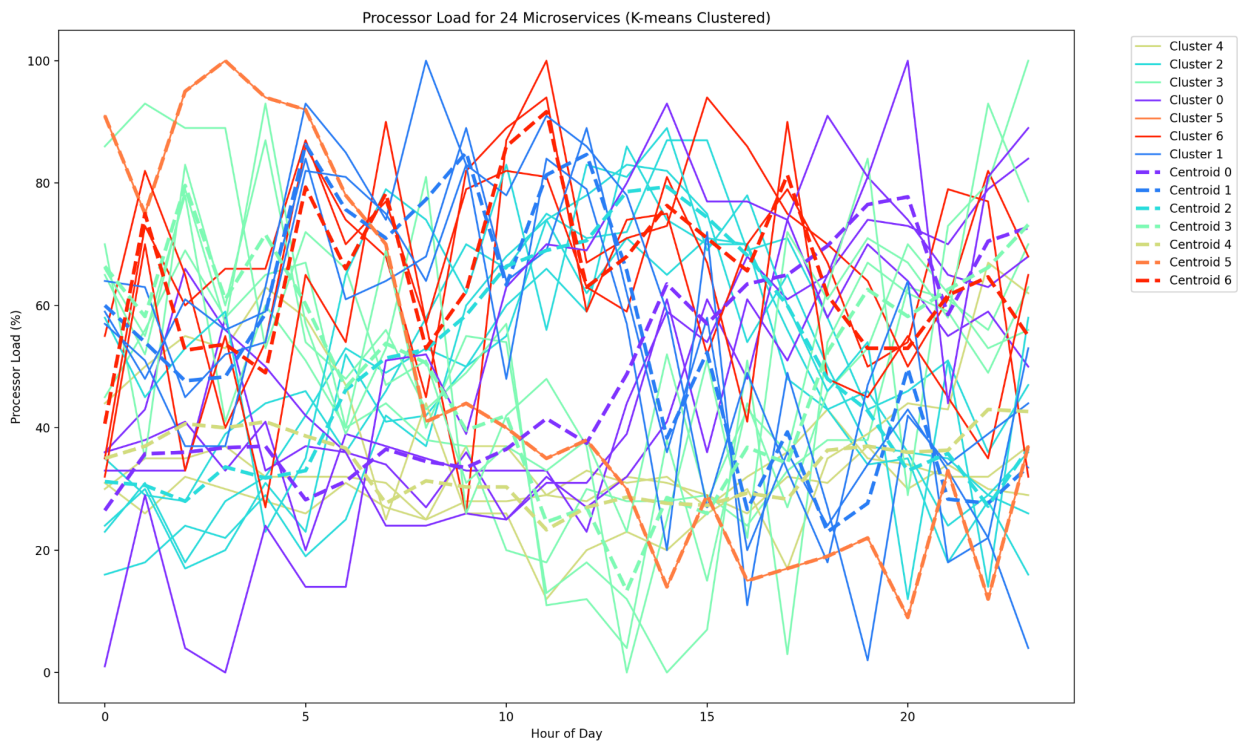


Рисунок Д2.3 — Кластеризовані дані про навантаження задач

Додаток 4. Код підготовки даних та їх кластеризації

```
#-----ALGORITHM-----#

#---Reshape the data to be 2D: (n_samples, n_features)---#
microservices_2d = microservices.reshape(microservices.shape[0], -1)

#---Apply Z-normalization---#
scaler = StandardScaler()
microservices_normalized = scaler.fit_transform(microservices_2d)

#---Perform K-means clustering---#
n_clusters = 7 # This number can be taken as a square root of n or as a dif.
valid cluster numb.
kmeans = KMeans(n_clusters=n_clusters, random_state=42)
cluster_labels = kmeans.fit_predict(microservices_normalized)

#---Apply PCA for visualization---#
pca = PCA(n_components=2)
microservices_pca = pca.fit_transform(microservices_normalized)
centroids_pca = pca.transform(kmeans.cluster_centers_)

#---Plot the clustered data points and centroids---#
plt.figure(figsize=(12, 8))
colors = plt.cm.rainbow(np.linspace(0, 1, n_clusters))

for i, color in enumerate(colors):
    cluster_points = microservices_pca[cluster_labels == i]
    plt.scatter(cluster_points[:, 0], cluster_points[:, 1], color=color,
label=f'Cluster {i}', alpha=0.7)
    plt.scatter(centroids_pca[i, 0], centroids_pca[i, 1], color=color,
marker='x', s=200, linewidths=3)

plt.title('Microservices Clustered (PCA Visualization)')
plt.xlabel('First Principal Component')
plt.ylabel('Second Principal Component')
plt.legend()
plt.grid(True, linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()

#---Plot original time series data with cluster colors---#
```

```

plt.figure(figsize=(15, 10))

for i in range(microservices.shape[0]):
    plt.plot(hours, microservices[i], color=colors[cluster_labels[i]],
             label=f'Cluster {cluster_labels[i]} ' if cluster_labels[i] not in
cluster_labels[:i] else "")

#---Plot cluster centroids---#
centroids_original = scaler.inverse_transform(kmeans.cluster_centers_)
for i, centroid in enumerate(centroids_original):
    plt.plot(hours, centroid, color=colors[i], linewidth=3, linestyle='--',
             label=f'Centroid {i}')

plt.title('Original Processor Load for 24 Microservices (K-means Clustered)')
plt.xlabel('Hour of Day')
plt.ylabel('Processor Load (%)')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.tight_layout()
plt.show()

#---Print cluster information---#
for i in range(n_clusters):
    cluster_members = np.where(cluster_labels == i)[0]
    print(f"\nCluster {i}:")
    print(f"  Members: {cluster_members}")
    print(f"  Centroid: {centroids_original[i]}")

#---Save the normalized data and cluster labels to files---#
np.savetxt('microservices_load_normalized.csv', microservices_normalized,
           delimiter=',')
np.savetxt('cluster_labels.csv', cluster_labels, delimiter=',', fmt='%d')
print("\nNormalized data saved to 'microservices_load_normalized.csv'")
print("Cluster labels saved to 'cluster_labels.csv'")

```

Додаток 5. Приклад виводу статистики по кластерах після кластеризації

Cluster 0:

Members: [8 9]

Centroid: [1. 0. 1. 2.5 11.5 1. 4. 0. 2. 10. 9. 8.5 1.5 15.
23. 38. 48.5 72.5 80. 81.5 70. 77. 61.5 64.5]

Cluster 1:

Members: [18 19 22 23]

Centroid: [50.75 39.25 30.5 29.5 30.75 49.75 62. 76.75 92. 89.25 97.5 92.75
93.25 96. 94.5 93.25 97.25 78.25 62.75 59.5 53.5 49.75 41. 29.75]

Cluster 2:

Members: [0 1 12 13]

Centroid: [50. 50. 50. 50. 50. 50. 50. 50. 50. 50. 50. 50. 50. 50. 50. 50. 50.
50. 50. 50. 50. 50. 50.]

Cluster 3:

Members: [14 15 16 17]

Centroid: [96.25 95. 94.25 96.75 95.25 83.75 83.25 80.25 66.25 51.75 34.25 28.
27.75 20.75 30.25 27.75 40.5 41.5 61.5 71.25 83. 91. 92.5 91.25]

Cluster 4:

Members: [2 3 4 5]

Centroid: [3.75 5. 5.75 3.25 4.75 16.25 16.75 19.75 33.75 48.25 65.75 72.
72.25 79.25 69.75 72.25 59.5 58.5 38.5 28.75 17. 9. 7.5 8.75]

Cluster 5:

Members: [6 7 10 11]

Centroid: [49.25 60.75 69.5 70.5 69.25 50.25 38. 23.25 8. 10.75 2.5 7.25
6.75 4. 5.5 6.75 2.75 21.75 37.25 40.5 46.5 50.25 59. 70.25]

Cluster 6:

Members: [20 21]

Centroid: [99. 100. 99. 97.5 88.5 99. 96. 100. 98. 90. 91. 91.5
98.5 85. 77. 62. 51.5 27.5 20. 18.5 30. 23. 38.5 35.5]

Додаток 6. Вивід результатів суміщення елементів з доповнювальних кластерів

Complementation Stats:

Clusters 0 and 6:

Discrepancy: 0.00%
Centroid Match: Yes
Active: No

Clusters 1 and 5:

Discrepancy: 0.00%
Centroid Match: Yes
Active: No

Clusters 2 and 2:

Discrepancy: 0.00%
Centroid Match: Yes
Active: No

Clusters 3 and 4:

Discrepancy: 0.00%
Centroid Match: Yes
Active: No

Clusters 1 and 3:

Discrepancy: 16.44%
Centroid Match: Yes
Active: No

Clusters 4 and 5:

Discrepancy: 18.59%
Centroid Match: Yes
Active: No

Clusters 2 and 6:

Discrepancy: 18.99%
Centroid Match: Yes
Active: No

Clusters 2 and 3:

Discrepancy: 20.95%
Centroid Match: Yes
Active: No

Clusters 1 and 2:

Discrepancy: 21.20%
Centroid Match: Yes
Active: No

Clusters 1 and 6:

Discrepancy: 28.16%
Centroid Match: Yes
Active: No

Clusters 6 and 6:

Discrepancy: 28.48%
Centroid Match: Yes
Active: No

Clusters 3 and 6:

Discrepancy: 29.57%
Centroid Match: Yes
Active: No

Додаток 7. Результати пошуку пар задач з доповнювальними навантаженнями

Complementary Microservices:

Pair 1:

Microservice 1: Max=95, Min=0, Range=95

Microservice 2: Max=100, Min=5, Range=95

Pair 2:

Microservice 1: Max=88, Min=0, Range=88

Microservice 2: Max=100, Min=12, Range=88

Pair 3:

Microservice 1: Max=100, Min=11, Range=89

Microservice 2: Max=89, Min=0, Range=89

Pair 4:

Microservice 1: Max=100, Min=21, Range=79

Microservice 2: Max=79, Min=0, Range=79

Pair 5:

Microservice 1: Max=100, Min=0, Range=100

Microservice 2: Max=100, Min=0, Range=100

Pair 6:

Microservice 1: Max=100, Min=0, Range=100

Microservice 2: Max=100, Min=0, Range=100

Pair 7:

Microservice 1: Max=58, Min=40, Range=18

Microservice 2: Max=58, Min=40, Range=18

Pair 8:

Microservice 1: Max=60, Min=42, Range=18

Microservice 2: Max=60, Min=42, Range=18

Pair 9:

Microservice 1: Max=100, Min=20, Range=80

Microservice 2: Max=80, Min=0, Range=80

Pair 10:

Microservice 1: Max=100, Min=13, Range=87

Microservice 2: Max=87, Min=0, Range=87

Pair 11:

Microservice 1: Max=100, Min=20, Range=80

Microservice 2: Max=80, Min=0, Range=80

Pair 12:

Microservice 1: Max=100, Min=14, Range=86

Microservice 2: Max=86, Min=0, Range=86

Додаток 8. Код розділення навантаження на задачах, які не можуть бути доповнені

```

# Часовий ряд
time_series = np.array([10, 12, 13, 31, 9, 35, 8, 12, 14, 38, 32, 50, 55, 52,
56, 52, 49, 40, 34, 30, 25, 20, 22, 11, 14])

# time_series = np.array([10, 12, 13, 11, 9, 35, 8, 12, 14, 11])

# Середнє та стандартне відхилення
mean = np.mean(time_series)
std_dev = np.std(time_series)

# Визначення екстремуми (аномалії)
threshold = 1 # поріг у стандартному відхиленні
#| (time_series < mean - threshold * std_dev)]
extremes = time_series[(time_series > mean + threshold * std_dev)]

# Індeksi першого та останнього елементів в підмножині extremes
first_idx = np.where(time_series == extremes[0])[0][0]
last_idx = np.where(time_series == extremes[-1])[0][0]

# Масив для розширення extremes
extremes_extention_left = []

# Перевірка елементів ліворуч від першого елементу extremes
for i in range(first_idx - 1, -1, -1):
    if time_series[i] > mean:
        extremes_extention_left.append(time_series[i])
    else:
        break

# Перевірка елементів праворуч від останнього елементу extremes
extremes_extention_right = []
for i in range(last_idx + 1, len(time_series)):
    if time_series[i] > mean:
        extremes_extention_right.append(time_series[i])
    else:
        break

# Максимальне значення за виключенням extremes та extremes_extention

```

```

mask = ~np.isin(time_series, np.concatenate([extremes_extention_left, extremes,
extremes_extention_right]))
new_max_value = np.max(time_series[mask])

print("extremes_extention_left:", extremes_extention_left)
print("extremes_extention_right:", extremes_extention_right)
print("new_max_value:", new_max_value)

def extract_smaller_values_then(extremes_extention, new_max_value):
    if extremes_extention:
        min_extention = np.min(extremes_extention)
        if new_max_value > min_extention:
            extremes_extention = [x for x in extremes_extention if x >=
new_max_value]
    return extremes_extention

# Якщо нове максимальне значення більше за мінімальне значення у
extremes_extention, видаляємо всі менші значення
extremes_extention_left = extract_smaller_values_then(extremes_extention_left,
new_max_value)
extremes_extention_right =
extract_smaller_values_then(extremes_extention_right, new_max_value)

print("extremes_extention:", extremes_extention_left)
print("extremes_extention:", extremes_extention_right)

# Об'єднуємо extremes та extremes_extention
final_extremes = np.concatenate([extremes_extention_left, extremes,
extremes_extention_right])

print("Final extremes:", final_extremes)
print("Середнє:", mean)
print("Стандартне відхилення:", std_dev)
print("Екстремуми:", extremes)
print("Екстремуми розширені:", final_extremes)

plt.plot(time_series, label='Часовий ряд')
plt.axvline(np.where(time_series == extremes[0])[0][0], color='g',
linestyle='dotted', label='Початок екстремуму')

```

```

plt.axvline(np.where(time_series == final_extremes[0])[0][0], color='b',
linestyle='dashed', label='Початок розширеного екстремуму')
plt.axvline(np.where(time_series == extremes[-1])[0][-1], color='g',
linestyle='dotted', label='Кінець екстремуму')
plt.axvline(np.where(time_series == final_extremes[-1])[0][-1], color='b',
linestyle='dashed', label='Кінець розширеного екстремуму')
plt.axhline(y=new_max_value, color='r', linestyle='--', label=f'Лінія поділу =
{new_max_value} од.')
plt.axhline(y=mean, color='c', linestyle='dashdot', label=f'Середнє =
{mean:.2f} од.')
plt.axhline(y=mean + std_dev, color='y', linestyle='dotted', label=f'Середнє +
стандартне відхилення = {mean + std_dev:.2f} од.')
plt.axhline(y=mean - std_dev, color='y', linestyle='dotted', label=f'Середнє -
стандартне відхилення = {mean - std_dev:.2f} од.')
plt.title('Виділення часової рамки та навантаження для створення окремого
екземпляра мікросервісу', loc='center')
plt.xlabel('Години')
plt.ylabel('Навантаження, од.')
plt.legend()
plt.show()

#-----Розділення графіків-----#
# 1. Графік зі значеннями меншими та рівними за new_max_value
modified_series = np.where(time_series > new_max_value, new_max_value,
time_series)

# 2. Графік зі значеннями, більшими за new_max_value, з відніманням
new_max_value
above_new_max = np.where(time_series > new_max_value, time_series -
new_max_value, 0)

fig, axs = plt.subplots(2, 1, figsize=(10, 8), sharex=True)

# Графік початкового часового ряду
axs[0].plot(modified_series, label='Навантаження без піка', color='b')
axs[0].set_title('Навантаження ≤ за максимальне дозволене (35 од.)')
axs[0].set_ylabel('Навантаження стандартне, од.')
axs[0].axhline(y=new_max_value, color='r', linestyle='--', label='Обмеження
максимумом = 35 од.')
axs[0].legend()
axs[0].grid()

```


Розділений графік, екстремум

```
axs[1].plot(above_new_max, label='Навантаження > за максимальне дозволене (35  
од.)', color='g')
```

```
axs[1].set_title('Пікове навантаження, винесене в окремий сервіс')
```

```
axs[1].set_ylabel('Навантаження підвищене, од.')
```

```
axs[1].legend()
```

```
axs[1].grid()
```

```
axs[1].set_xticks(np.arange(0, len(time_series), step=1))
```

```
axs[1].set_xticklabels(np.arange(0, 25, step=1))
```

```
plt.xlabel('Години')
```

```
plt.tight_layout()
```

```
plt.show()
```

Додаток 9. Результати розділення пікових та середніх навантажень у задачах з навантаженнями, які не можуть бути доповненими

```

extremes_extention_left: [np.int64(32), np.int64(38)]
extremes_extention_right: [np.int64(40), np.int64(34), np.int64(30)]
new_max_value: 35
extremes_extention: [np.int64(38)]
extremes_extention: [np.int64(40)]
Final extremes: [38 50 55 52 56 52 49 40]
Середнє: 28.96
Стандартне відхилення: 16.17029375119698
Екстремуми: [50 55 52 56 52 49]
Екстремуми розширені: [38 50 55 52 56 52 49 40]

```

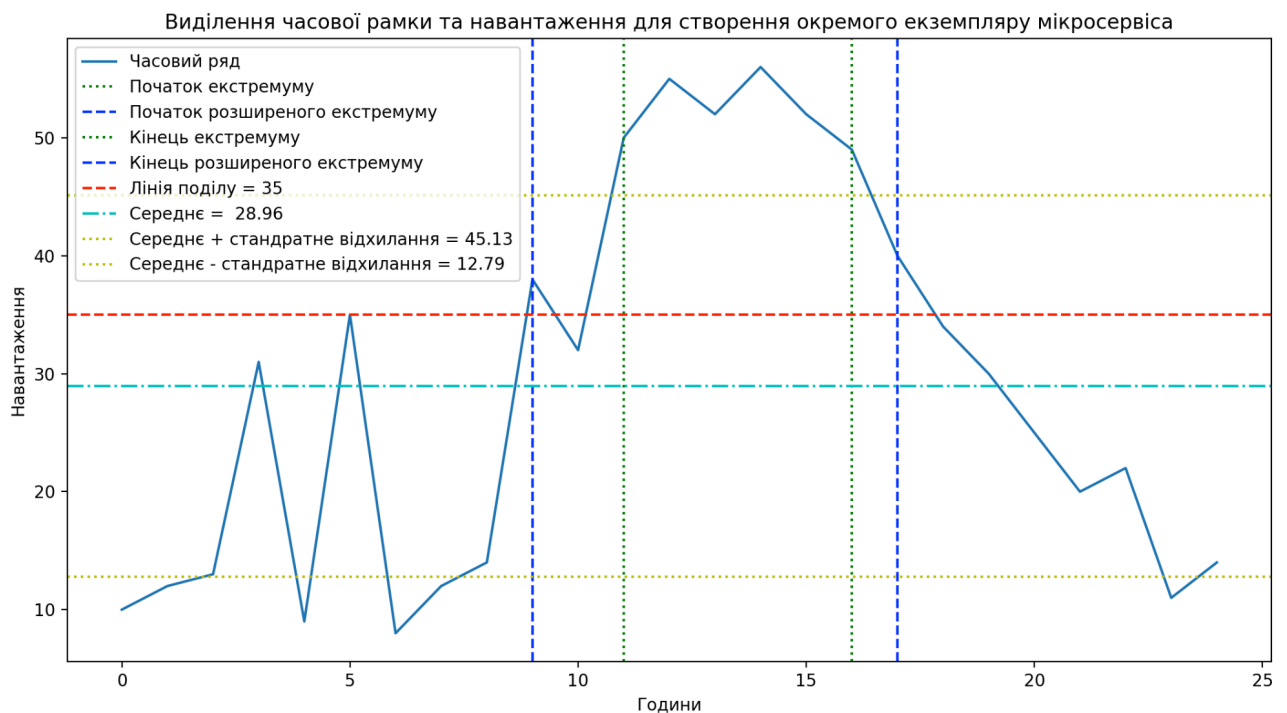


Рисунок Д9.1 — Виділення навантаження мікропроцесора, яке слід запустити на іншому сервері. Стосовно моноліта - виділення часу його додаткового запуску.

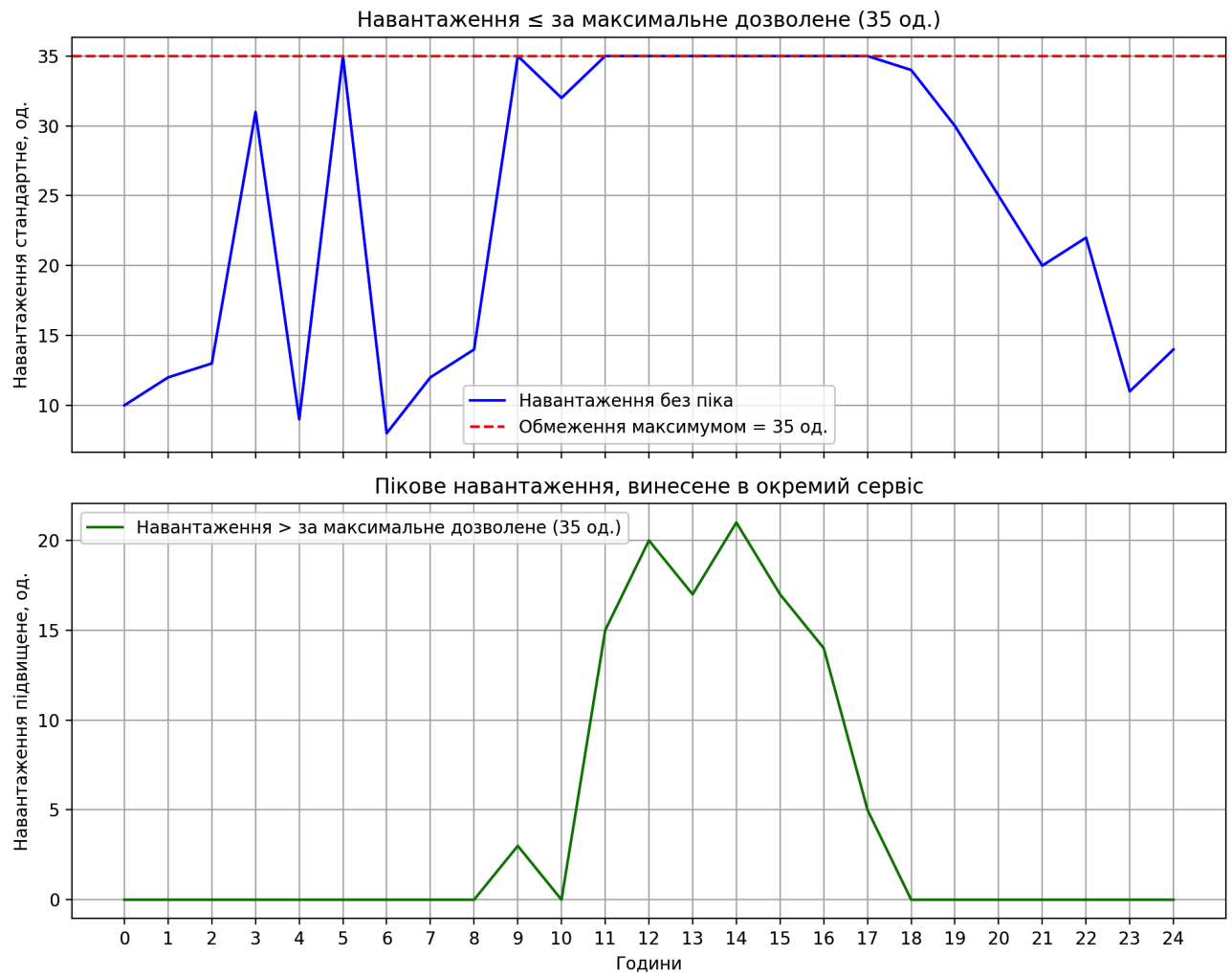


Рисунок Д9.2. — Розділення мікросервісу на дві частини, виділяючи екстремум в окремий екземпляр.

Додаток 10. Акт впровадження результатів наукового дослідження

«ЗАТВЕРДЖУЮ»

В.о. директора

Навчально-наукового Інституту телекомунікаційних систем

Національного технічного університету України

“Київський політехнічний інститут імені Ігоря Сікорського”

к.т.н., доц. Валерій ПРАВИЛО

21 січня 2025 р.

Акт

впровадження результатів наукових досліджень,

викладених у дисертації на здобуття наукового ступеня доктора філософії,

Дмитренко Олександри Анатоліївни, виконаної на тему

«Метод доповнювальних навантажень для розподілу задач в хмарних системах»

Теоретичні та практичні напрацювання, отримані Дмитренко О.А. при розробці основних наукових результатів дисертації, які викладено у робочих матеріалах, проміжних та остаточних звітах ініціативних НДР «“Інтелектуальне керування динамічною енергоефективною реконфігурацією обчислювальних ресурсів для підтримки технології NaaS”», Д/р – № 0123U101635 (початок 15.03.2023 — закінчення 15.03.2026) спец. 172 – Електронні комунікації та радіотехніка. Голова комісії: д.т.н., проф. Лариса Глоба, члени комісії: д.т.н., проф. Марія Скулиш, к.т.н., доц. Олег Цуканов, к.т.н., доц. Світлана Суліма.

Комісія у складі: Голова комісії – д.т.н., проф., професор кафедри інформаційних технологій в телекомунікаціях НН ІТС Глоба Лариса Сергіївна. Члени комісії – д.т.н., проф., завідувач кафедри інформаційних технологій в телекомунікаціях НН ІТС Скулиш Марія Анатоліївна, цим актом засвідчує, що наукові та науково-практичні результати дисертаційного дослідження Дмитренко О.А. за темою «Метод доповнювальних навантажень для розподілу задач в хмарних системах», впроваджені у 2024 році у навчальному процесі на кафедрі інформаційних технологій в телекомунікаціях Навчально-наукового Інституту телекомунікаційних систем та використані при підготовці та викладанні курсу «BigData та методи їх обробки», а також при розробці практичних робіт що викладається Глобою Л.С. для підготовки студентів за спеціальністю 172 – Телекомунікації та радіотехніка.

В навчальний процес було впроваджено:

- 1) підхід до побудови матмоделі для оцінки потреб в розподілі навантажень обчислювальних задач;
- 2) метод доповнювальних навантажень для пошуку комплементарних груп навантажень при розподілі задач на серверах;
- 3) алгоритм побудови планувальника розподілу задач в хмарному середовищі з розрахунком доповнення навантаження для ефективного його використання.

Впровадження результатів дослідження, отриманих протягом дисертаційної роботи Дмитренко О.А., підвищує якість підготовки студентів за спеціальністю 172 – Телекомунікації та радіотехніка, оскільки відображає стан та перспективи розвитку наукових досліджень в Україні та світі.

Голова комісії:

д.т.н., проф. _____ Глоба Л.С.

В.о. директора НН ІТС:

к.т.н доцент _____ Правило В.В.

Члени комісії:

к.т.н доцент _____ Цуканов О.Ф.

д.т.н., проф. _____ Скулиш М.А.

к.т.н доцент _____ Суліма С.В.