М. І. Бешлей, М. М. Климаш,
О. М. Шпур, Г. В. Бешлей

МОДЕЛІ ТА МЕТОДИ ПОБУДОВИ РОЗПОДІЛЕНИХ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ СИСТЕМ



М. І. Бешлей, М. М. Климаш, О. М. Шпур, Г. В. Бешлей

МОДЕЛІ ТА МЕТОДИ ПОБУДОВИ РОЗПОДІЛЕНИХ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ СИСТЕМ

Монографія

Полтава
ЛітЕра
2026

УДК 621.391
М 74

Колектив авторів:
Бешлей М. І., Климаш М. М., Шпур О. М., Бешлей Г. В.

Рецензенти:

Русин Б. П. – доктор технічних наук, професор, завідувач відділу інформаційних технологій дистанційного зондування, Фізико-механічний інститут ім. Г. В. Карпенка НАН України;

Кочан О. В. – доктор технічних наук, професор, професор кафедри інформаційно-вимірювальних технологій, Інститут комп'ютерних технологій, автоматики та метрології Національного університету «Львівська політехніка»;

Бондарєв А. П. – доктор технічних наук, професор, професор кафедри програмно-апаратних систем інфокомунікацій, Інститут інформаційно-комунікаційних технологій та електронної інженерії Національного університету «Львівська політехніка».

М 74 **Моделі** та методи побудови розподілених інформаційно-комунікаційних систем / М. І. Бешлей, М. М. Климаш, О. М. Шпур, Г. В. Бешлей. Полтава : ЛітЕра, 2026. 220 с.

Рекомендовано до друку Вченою радою
Національного університету "Львівська політехніка"
(протокол № 36 від 23 червня 2026 р.)

ISBN 978-617-95543-2-2
DOI: 10.51587/9786-1795-54322-2026-36

Монографія присвячена дослідженню моделей і методів побудови розподілених інформаційно-комунікаційних систем. Розглянуто технологічні засади їх побудови з використанням хмарних і периферійних обчислень, віртуалізації, контейнеризації та мікросервісної архітектури. Досліджено методи реактивного та проактивного автомасштабування обчислювальних ресурсів, зокрема підходи до прогнозування навантаження. Значну увагу приділено адаптивному балансуванню навантаження в Edge-Cloud середовищі та ефективному використанню ресурсів периферійних вузлів. Запропоновано модель і метод інтелектуального управління розміщенням мікросервісів з урахуванням поточного та прогнозованого стану обчислювальних вузлів. Наведено результати експериментальних досліджень запропонованих рішень.

Видання призначене для науковців, інженерів, викладачів, аспірантів і здобувачів вищої освіти за спеціальностями G5 «Електроніка, електронні комунікації, приладобудування та радіотехніка» та F6 «Інформаційні системи і технології», а також фахівців, які працюють у галузі розподілених систем, хмарних і периферійних обчислень, мікросервісних технологій та управління обчислювальними ресурсами.

УДК 621.391

ISBN 978-617-95543-2-2

© Бешлей М., 2026
© Климаш М., 2026
© Шпур О., 2026
© Бешлей Г., 2026

З М І С Т

ВСТУП **5**

Розділ І **ТЕОРЕТИЧНІ І ТЕХНОЛОГІЧНІ ЗАСАДИ ПОБУДОВИ РОЗПОДІЛЕНИХ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ СИСТЕМ** **7**

1.1. Поняття та особливості розподілених інформаційно-комунікаційних систем 7

1.2. Хмарні технології та віртуалізація як основа побудови розподілених інформаційно-комунікаційних систем 12

1.3. Мікросервісна архітектура та оркестрація контейнеризованих сервісів 22

1.4. Методи горизонтального та вертикального автомасштабування ресурсів розподілених систем на основі Kubernetes 31

1.5. Моніторинг та оцінювання стану ресурсів розподілених інформаційно-комунікаційних систем 35

1.6. Обґрунтування необхідності проактивного масштабування ресурсів у розподілених інформаційно-комунікаційних системах ... 43

Розділ ІІ **СИСТЕМАТИЗАЦІЯ МОДЕЛЕЙ І МЕТОДІВ ПОБУДОВИ МІКРОСЕРВІСНИХ РОЗПОДІЛЕНИХ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ СИСТЕМ** **47**

2.1. Особливості та обмеження архітектурних підходів до побудови розподілених інформаційно-комунікаційних систем 47

2.2. Обмін повідомленнями та моніторинг ресурсів у мікросервісних розподілених інформаційно-комунікаційних систем 60

2.3. Концептуальні моделі та методи управління мікросервісами у розподілених інформаційно-комунікаційних системах 64

2.4. Систематизація моделей і методів побудови розподілених інформаційно-комунікаційних систем 79

РОЗДІЛ ІІІ **МЕТОДИ РЕАКТИВНОГО ТА ПРОАКТИВНОГО АВТОМАСШТАБУВАННЯ РЕСУРСІВ У РОЗПОДІЛЕНИХ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ СИСТЕМАХ** **85**

3.1. Методика експериментального дослідження реактивного масштабування ресурсів у Kubernetes-середовищі 85

3.2. Експериментальне дослідження ефективності вертикального та горизонтального масштабування ресурсів РІКС 88

3.3. Методи машинного навчання для прогнозування навантаження та проактивного масштабування ресурсів 106

3.4. Метод проактивного масштабування обчислювальних ресурсів РІКС на основі моделей LSTM та Holt–Winters 109

3.5. Підготовка часових даних та формування вибірок для навчання LSTM-моделі 120
3.6. Дослідження ефективності проактивного автомасштабування обчислювальних ресурсів 126

РОЗДІЛ IV МОДЕЛІ ТА МЕТОДИ АДАПТИВНОГО УПРАВЛІННЯ НАВАНТАЖЕННЯМ У РОЗПОДІЛЕНОМУ EDGE-CLOUD СЕРЕДОВИЩІ 133

4.1. Архітектурні моделі Edge-Cloud систем розподілу обчислень та офлоадингу 133
4.2. Методи балансування навантаження в хмарних та периферійних обчислювальних системах 137
4.3. Структурно-функціональна модель досліджуваної Edge-Cloud системи 143
4.4. Модель формування навантаження та алгоритм вибору дій у системі Brownout 148
4.5. Метод адаптивного управління навантаженням у розподіленому Edge-Cloud середовищі 155
4.6. Експериментальне середовище та методика оцінювання ресурсно-адаптивного балансування в Edge-Cloud системі 164
4.7. Дослідження ефективності адаптивного балансування навантаження в Edge-Cloud середовищі 172

РОЗДІЛ V МОДЕЛЬ ТА МЕТОД ІНТЕЛЕКТУАЛЬНОГО УПРАВЛІННЯ РОЗМІЩЕННЯМ МІКРОСЕРВІСІВ У РОЗПОДІЛЕНИХ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ СИСТЕМАХ 183

5.1. Концептуальна модель інтелектуального управління розміщенням мікросервісів у розподілених інформаційно-комунікаційних системах 183
5.2. Метод прогнозування стану працездатності вузлів на основі аналізу логів 191
5.3. Дослідження ефективності методу інтелектуального управління розміщенням мікросервісів у розподілених інформаційно-комунікаційних системах 194
5.4. Методологія побудови розподілених інформаційно-комунікаційних систем на основі запропонованих рішень 202

ВИСНОВКИ 207
ПОДЯКА 211
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 212

ВСТУП

Розподілені інформаційно-комунікаційні системи (РІКС) широко використовуються для надання цифрових сервісів, оброблення та передавання даних, а також організації взаємодії між користувачами, пристроями й програмними застосунками. Їх особливістю є розподіл обчислювальних і мережевих ресурсів між різними вузлами, які спільно виконують завдання та забезпечують роботу сервісів. Такий підхід дає змогу гнучко використовувати доступні ресурси, масштабувати систему та наближати оброблення даних до кінцевих користувачів.

Розвиток хмарних і периферійних обчислень, віртуалізації, контейнеризації та мікросервісної архітектури значно розширив можливості побудови РІКС. Платформи оркестрації, зокрема Kubernetes, забезпечують автоматизоване розгортання, масштабування та відновлення мікросервісів. Водночас зі збільшенням кількості вузлів і сервісів ускладнюється управління ресурсами, особливо за умов постійної зміни навантаження. Це зумовлює необхідність розроблення методів, які дають змогу ефективно використовувати наявні ресурси та своєчасно адаптувати систему до зміни умов її функціонування.

Одним із таких напрямів є автомасштабування ресурсів. Реактивні методи змінюють обсяг ресурсів відповідно до поточного навантаження, тоді як проактивні дають змогу прогнозувати його зміну та завчасно підготувати необхідні обчислювальні ресурси. Важливе значення має також балансування навантаження між вузлами Edge-Cloud середовища, що дає змогу уникати перевантаження окремих вузлів і раціональніше використовувати їх обчислювальні можливості.

Окремим завданням є ефективне розміщення мікросервісів у розподіленій системі. Вибір вузла для виконання сервісу впливає на використання ресурсів, мережевий трафік і затримку оброблення запитів. Тому під час управління розміщенням доцільно враховувати поточний стан інфраструктури, взаємодію між мікросервісами та прогнозовані зміни стану обчислювальних вузлів.

Монографію присвячено дослідженню та розвитку моделей і методів побудови розподілених інформаційно-комунікаційних систем. Розглянуто теоретичні та технологічні засади РІКС, підходи до побудови мікросервісних систем, методи реактивного і проактивного автомасш-

табування ресурсів, прогнозування навантаження та адаптивного балансування в Edge-Cloud середовищі. Значну увагу приділено також інтелектуальному управлінню розміщенням мікросервісів з урахуванням прогнозованого стану вузлів.

Запропоновані в монографії моделі та методи спрямовані на ефективне використання обчислювальних і мережевих ресурсів, зменшення перевантажень та підвищення стабільності функціонування РІКС в умовах динамічної зміни навантаження.

Розділ І

ТЕОРЕТИЧНІ І ТЕХНОЛОГІЧНІ ЗАСАДИ ПОБУДОВИ РОЗПОДІЛЕНИХ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ СИСТЕМ

1.1. Поняття та особливості розподілених інформаційно-комунікаційних систем

Інформаційно-комунікаційні системи є технологічною основою процесів збирання, передавання, оброблення, зберігання та використання інформації. Їх функціонування ґрунтується на взаємодії комунікаційної інфраструктури, обчислювальних засобів, програмного забезпечення та інформаційних ресурсів. У міру розширення функціональних можливостей таких систем змінюються не лише їх масштаби, а й принципи організації: обчислювальні ресурси, дані та програмні сервіси дедалі частіше розміщуються на різних фізичних і логічних вузлах та спільно використовуються для виконання прикладних завдань.

Така організація зумовлена необхідністю забезпечення ефективного доступу до інформаційних ресурсів для територіально розподілених користувачів, обслуговування значних і нерівномірних потоків даних, зменшення часу надання сервісів та раціонального використання доступних мережевих і обчислювальних ресурсів. У результаті традиційне розмежування між комунікаційною мережею, обчислювальною інфраструктурою та прикладними сервісами поступово втрачає однозначність. Вони формують взаємопов'язане середовище, у якому характеристики одного компонента безпосередньо впливають на ефективність функціонування інших.

У цьому контексті важливого значення набуває поняття розподіленої системи [1]. У загальному випадку розподіленою є система, функціональні компоненти та ресурси якої розміщені на множині взаємопов'язаних вузлів і координовано використовуються для виконання спільних завдань. Взаємодія між такими вузлами здійснюється шляхом обміну інформацією через комунікаційне середовище, а результат функціонування системи формується внаслідок узгодженої роботи її окремих компонентів [2].

Принциповою ознакою розподіленої системи є не саме територіальне рознесення обладнання, а розподіл ресурсів, функцій і процесів

між декількома взаємодіючими компонентами. Тому поняття розподіленості доцільно розглядати ширше, ніж географічне розташування вузлів. Компоненти системи можуть бути розміщені як у різних географічних точках, так і в межах одного центру оброблення даних, але при цьому бути логічно відокремленими та виконувати різні функції.

Залежно від способу організації компонентів можна виділити фізичну, логічну та функціональну розподіленість. Фізична розподіленість визначається розміщенням ресурсів на різних апаратних вузлах. Логічна пов'язана з поділом програмних та інформаційних ресурсів незалежно від їх фізичного розташування. Функціональна розподіленість передбачає декомпозицію процесів системи та виконання окремих функцій різними її компонентами. Для реальних інформаційно-комунікаційних систем характерним є одночасне поєднання зазначених видів розподіленості.

На відміну від класичної розподіленої обчислювальної системи, для інформаційно-комунікаційної системи принципове значення має комунікаційна складова. Виконання прикладного завдання потребує не лише наявності необхідних обчислювальних ресурсів, але й забезпечення передавання інформації між джерелами даних, обчислювальними вузлами та кінцевими користувачами із заданими характеристиками якості. Тому мережеві та обчислювальні ресурси в таких системах доцільно розглядати як взаємопов'язані складові єдиного процесу надання сервісів.

Виходячи з цього, під розподіленою інформаційно-комунікаційною системою (РІКС) у межах цієї монографії будемо розуміти сукупність взаємопов'язаних фізично та/або логічно розподілених мережевих, обчислювальних, програмних та інформаційних ресурсів, узгоджена взаємодія яких забезпечує передавання, оброблення, зберігання та використання інформації, а також надання цифрових сервісів відповідно до заданих вимог щодо якості обслуговування, продуктивності, надійності та безпеки.

Таке визначення дозволяє розглядати РІКС як цілісне мережево-обчислювальне середовище, у якому комунікаційні, обчислювальні та інформаційні процеси не є ізольованими. Їх взаємозалежність визначає структуру системи, принципи розміщення сервісів, характер розподілу ресурсів та способи управління ними. Узагальнену структуру розподіленої інформаційно-комунікаційної системи наведено на рис. 1.1.

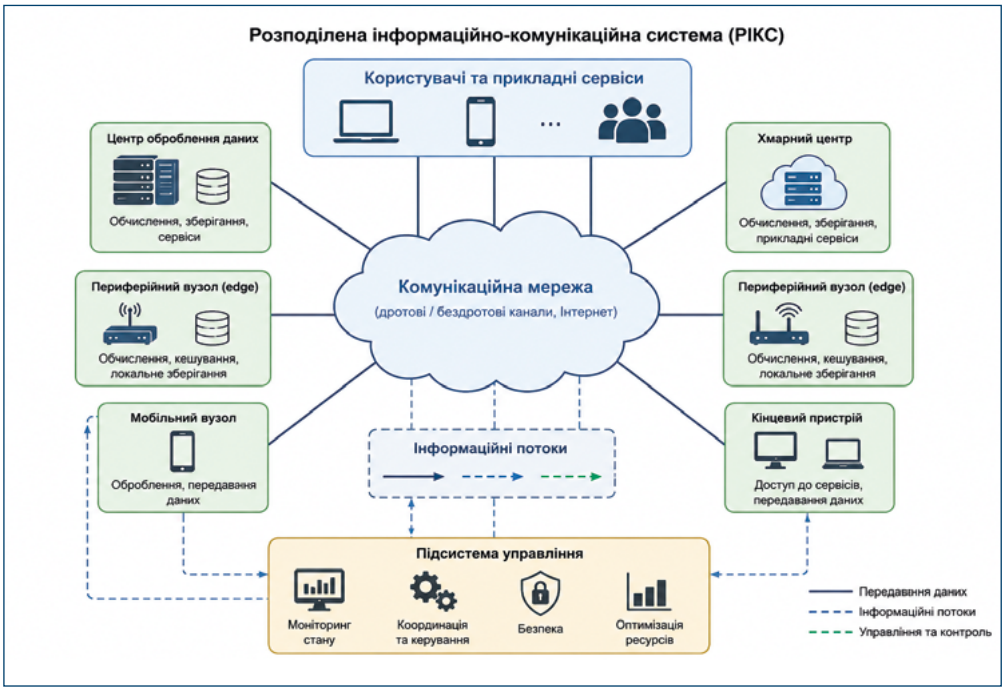


Рис. 1.1. Узагальнена структура розподіленої інформаційно-комунікаційної системи

Узагальнено розподілену інформаційно-комунікаційну систему можна представити як:

$$S = \{N, L, R, A, D, M\}, \qquad (1.1)$$

де N – множина фізичних і віртуальних вузлів системи; L – множина фізичних та логічних зв'язків між ними; R – сукупність мережевих, обчислювальних і сховищних ресурсів; A – множина прикладних функцій та сервісів; D – сукупність даних та інформаційних потоків; M – засоби моніторингу, координації та управління.

До складу множини вузлів N можуть входити центри оброблення даних, фізичні сервери, віртуальні машини, контейнери, мережеві пристрої, периферійні обчислювальні вузли, мобільні та кінцеві пристрої. Залежно від призначення системи окремий вузол може виконувати функції передавання, оброблення або зберігання інформації, а також поєднувати декілька функцій.

Однією з основних особливостей РІКС є розподіленість ресурсів і функцій. Обчислювальні потужності, засоби зберігання даних та програмні сервіси можуть бути розміщені на різних вузлах системи. Такий

підхід створює можливість використовувати ресурси декількох компонентів для виконання спільних завдань, перерозподіляти навантаження та розміщувати сервіси ближче до користувачів або джерел інформації.

Водночас розподіленість зумовлює необхідність забезпечення узгодженої взаємодії між окремими компонентами. Для виконання певного завдання недостатньо лише наявності вільного обчислювального ресурсу. Необхідно також забезпечити передавання до нього необхідних даних із відповідними показниками затримки, пропускної здатності та надійності. Саме тому характерною особливістю РІКС є взаємозалежність мережевих та обчислювальних ресурсів.

Наприклад, вибір вузла з найвищою обчислювальною продуктивністю не завжди забезпечує найменший час виконання запиту. Якщо такий вузол розташований на значній мережевій відстані від користувача, затримка передавання даних може суттєво вплинути на загальний час надання сервісу. Водночас найближчий до користувача вузол може мати недостатню продуктивність або бути перевантаженим. Отже, вибір ресурсів у РІКС потребує одночасного врахування характеристик комунікаційної та обчислювальної складових.

Таким чином, зміна характеристик мережі або стану обчислювального вузла безпосередньо впливає на кінцеві показники надання сервісу. Це визначає необхідність комплексного підходу до розподілу мережевих та обчислювальних ресурсів. Характерною особливістю РІКС є також гетерогенність компонентів. У межах однієї системи можуть функціонувати вузли з різною обчислювальною продуктивністю, обсягом пам'яті, характеристиками мережевих інтерфейсів, програмним забезпеченням і функціональним призначенням. Така неоднорідність є природною для розподілених систем, оскільки вони можуть об'єднувати ресурси центрів оброблення даних, хмарних платформ, периферійних вузлів та кінцевих пристроїв [4].

Використання неоднорідних ресурсів розширює функціональні можливості системи, проте одночасно ускладнює процеси їх розподілу. Під час вибору вузла для розміщення або виконання певного сервісу необхідно враховувати не лише факт наявності вільного ресурсу, а і його характеристики та відповідність вимогам конкретного завдання.

Ще однією важливою властивістю РІКС є динамічність стану. У процесі функціонування змінюються інтенсивність інформаційних потоків,

кількість активних користувачів, рівень завантаження обчислювальних вузлів, доступна пропускна здатність каналів зв'язку та обсяг вільних ресурсів. Окремі компоненти можуть підключатися до системи, тимчасово ставати недоступними або виходити з ладу.

Тому стан РІКС не є сталим і може бути представлений як функція часу:

$$X(t)=F(N(t),L(t),R(t),A(t),D(t)) \qquad (1.2)$$

Динамічність системи означає, що рішення, прийняте для певного стану РІКС, не обов'язково залишатиметься ефективним після зміни навантаження або доступності ресурсів. У зв'язку з цим важливого значення набуває адаптивність, тобто здатність системи змінювати спосіб використання ресурсів відповідно до поточних або прогнозованих умов функціонування.

Адаптація може здійснюватися шляхом перерозподілу навантаження між вузлами, зміни маршрутів передавання даних, масштабування сервісів, перенесення програмних компонентів на інші обчислювальні ресурси та залучення резервних вузлів. Метою таких дій є підтримання заданих характеристик функціонування системи за зміни її внутрішнього стану або зовнішніх умов.

Важливою особливістю РІКС є масштабованість, тобто здатність системи забезпечувати необхідні показники функціонування за збільшення кількості користувачів, вузлів, сервісів та обсягів оброблюваної інформації. Розподілена організація створює можливість горизонтального масштабування шляхом додавання нових вузлів або екземплярів сервісів та розподілу навантаження між ними.

Поряд із масштабованістю важливого значення набуває відмовостійкість. Розподілення функцій між декількома компонентами створює можливість резервування критичних ресурсів і сервісів. У разі відмови окремого вузла його функції можуть бути перенесені на інший доступний ресурс, що дозволяє зберегти працездатність системи або мінімізувати рівень погіршення її характеристик.

Для забезпечення відмовостійкості застосовуються механізми резервування, реплікації даних і сервісів, виявлення відмов, автоматичного перемикання та відновлення. Водночас збільшення кількості компонентів у розподіленій системі підвищує складність її контролю, тому

ефективність зазначених механізмів значною мірою залежить від засобів моніторингу та управління.

Окремою властивістю РІКС є прозорість доступу до розподілених ресурсів. Для користувача фізичне місце розташування сервісу або даних не повинно визначати спосіб взаємодії з ними. Система має забезпечувати доступ до необхідної функції незалежно від того, на якому саме вузлі вона реалізована, а вибір конкретного ресурсу здійснювати з урахуванням поточного стану інфраструктури.

Функціонування РІКС безпосередньо пов'язане із забезпеченням якості обслуговування. Різні сервіси можуть висувати різні вимоги до затримки передавання даних, пропускної здатності, допустимого рівня втрат, доступності та обчислювальної продуктивності. Відповідно, розподіл ресурсів повинен здійснюватися не лише з урахуванням їх наявності, але й відповідно до вимог конкретних сервісів.

Не менш важливою характеристикою є безпека та стійкість функціонування РІКС. Наявність значної кількості вузлів, мережевих з'єднань та інтерфейсів взаємодії розширює множину потенційних точок впливу на систему. Тому захист інформації, контроль доступу, виявлення аномальної активності, моніторинг стану компонентів та забезпечення можливості відновлення після порушень повинні враховуватися під час побудови та експлуатації розподіленої системи [5].

Таким чином, розподіленим інформаційно-комунікаційним системам властиві розподіленість ресурсів і функцій, гетерогенність компонентів, взаємозалежність мережевих та обчислювальних процесів, динамічність стану, адаптивність, масштабованість, відмовостійкість, прозорість доступу до ресурсів, а також необхідність забезпечення заданого рівня якості обслуговування та безпеки.

1.2. Хмарні технології та віртуалізація як основа побудови розподілених інформаційно-комунікаційних систем

Важливим етапом розвитку РІКС стало поширення хмарних обчислень (Cloud Computing). Хмарний підхід забезпечив можливість надання обчислювальних потужностей, сховищ даних і програмних сервісів як

динамічно доступних ресурсів. Фізична інфраструктура при цьому абстрагується від користувача, а необхідний обсяг ресурсів може виділятися відповідно до поточного навантаження [6].

Хмарна модель істотно змінила принципи побудови інформаційно-комунікаційних систем. Замість жорсткої прив'язки програмного сервісу до конкретного фізичного обладнання стало можливим динамічне створення, масштабування, переміщення та видалення віртуальних ресурсів. Це забезпечило більш ефективне використання обчислювальної інфраструктури та створило передумови для гнучкого управління її ресурсами.

Хмарні технології сьогодні є основою розвитку сучасних інформаційних систем, оскільки забезпечують швидкий доступ до обчислювальних ресурсів, сервісів і сховищ даних через мережу Інтернет. Використання хмарних платформ дозволяє користувачам працювати з інформацією незалежно від місця перебування та типу пристрою [7]. На рис. 1.2 показано загальну модель взаємодії користувачів із хмарним середовищем.



Рис. 1.2. Модель взаємодії користувачів з хмарним середовищем

Хмарне середовище об'єднує різні інформаційні сервіси, зокрема електронну пошту, документи, мультимедійні файли, календарі та засоби

онлайн-комунікації. Доступ до цих ресурсів може здійснюватися зі смартфонів, планшетів, ноутбуків або персональних комп'ютерів. Важливою складовою хмарних платформ є механізми захисту інформації, які забезпечують конфіденційність, цілісність і контроль доступу до даних.

Хмарні обчислення є сучасним підходом до організації обчислювальних ресурсів, який поєднує мережеві технології, розподілені обчислення, віртуалізацію, балансування навантаження та механізми високої доступності. Основною особливістю хмарних обчислень є можливість отримання обчислювальних ресурсів і програмних сервісів через мережу Інтернет за потребою користувача.

Однією з ключових переваг хмарних технологій є модель оплати лише за фактично використані ресурси. Це дозволяє ефективніше використовувати обчислювальні потужності та зменшувати витрати на підтримку власної інфраструктури. Хмарні обчислення реалізуються на основі трьох основних моделей надання сервісів, наведених на рис. 1.3.

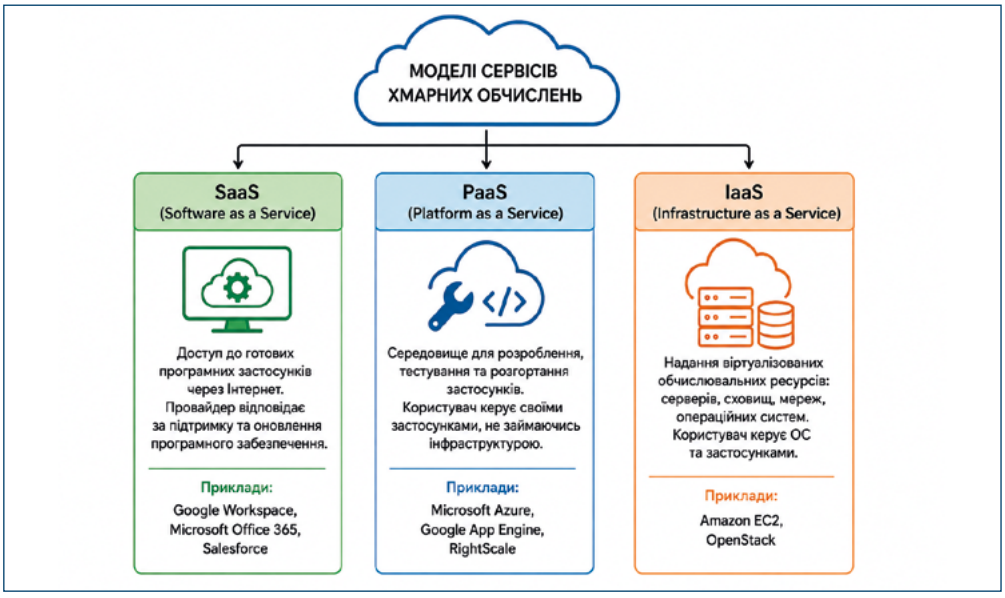


Рис. 1.3. Основні моделі сервісів хмарних обчислень

Software as a Service (SaaS) передбачає надання користувачеві доступу до готових програмних застосунків через мережу Інтернет без необхідності встановлення та обслуговування відповідного програмного забезпечення на локальних пристроях. Управління програмною та ба-

зовою інфраструктурою здійснює постачальник хмарних послуг, тоді як користувач отримує доступ до функціональних можливостей застосунку. Прикладами сервісів, що реалізують модель SaaS, є Google Workspace, Microsoft 365 та Salesforce [8].

Platform as a Service (PaaS) забезпечує користувачеві програмне середовище та необхідні інструменти для розроблення, тестування, розгортання й супроводу застосунків. При цьому управління базовою обчислювальною та мережевою інфраструктурою покладається на постачальника хмарних послуг, що дає змогу розробнику зосередитися безпосередньо на створенні та функціонуванні програмних сервісів. Прикладами PaaS-рішень є Google App Engine та Microsoft Azure App Service [9].

Infrastructure as a Service (IaaS) передбачає надання користувачеві віртуалізованих обчислювальних, мережевих і сховищних ресурсів. Користувач отримує можливість створювати та конфігурувати віртуальні машини, керувати операційними системами, мережевими параметрами та виділеними ресурсами відповідно до потреб прикладних сервісів. Така модель забезпечує гнучке нарощування або вивільнення ресурсів залежно від поточного навантаження. Прикладами IaaS-рішень є Amazon EC2 та OpenStack [10].

Однією з ключових переваг хмарних обчислень є еластичність, тобто здатність системи автоматично змінювати обсяг доступних ресурсів залежно від поточного навантаження. Такий підхід дозволяє ефективніше використовувати апаратні ресурси та зменшувати витрати, оскільки користувач сплачує лише за фактично використані потужності. Еластичність забезпечує адаптацію інфраструктури до змінних умов роботи та підвищує стабільність функціонування сервісів.

Розвиток хмарних технологій став результатом еволюції обчислювальних систем і методів віртуалізації. Спочатку використовувалися багатокористувацькі операційні системи та мейнфрейми, які дозволяли декільком користувачам працювати з одним фізичним сервером. Подальший розвиток віртуалізації забезпечив створення ізольованих програмних середовищ, що стало основою сучасних хмарних платформ [11].

Наступним етапом розвитку стала контейнеризація, яка значно спростила процес розгортання та перенесення застосунків між різними середовищами. Контейнери дозволяють запускати програми в ізольова-

ному середовищі без необхідності використання окремої повноцінної операційної системи. Перші механізми ізоляції процесів були реалізовані ще в UNIX-системах, а подальший розвиток контейнерних технологій став можливим завдяки появі cgroups та просторів імен у Linux. На відміну від віртуальних машин, контейнери не містять окремої операційної системи, а використовують ядро ОС хост-системи. Завдяки цьому Docker-контейнери споживають менше ресурсів, швидше запускаються та забезпечують кращу масштабованість.

Контейнери особливо ефективні для розгортання мікросервісних застосунків у розподілених інформаційних системах. При цьому контейнери та віртуальні машини можуть використовуватися спільно, доповнюючи один одного в хмарній інфраструктурі. Порівняння архітектури віртуальних машин та Docker-контейнерів зображено на рис. 1.4.

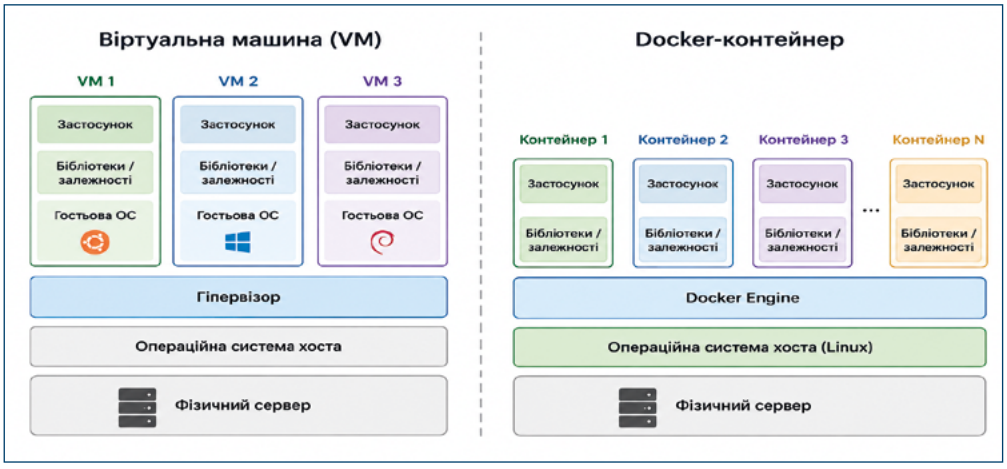


Рис. 1.4. Порівняння архітектури віртуальних машин та Docker-контейнерів

Контейнери мають низькі накладні витрати, швидко запускаються та дозволяють розміщувати велику кількість застосунків на одному сервері. У порівнянні з віртуальними машинами вони потребують менше ресурсів і забезпечують вищу ефективність використання обчислювальної інфраструктури. Завдяки цьому контейнеризація стала важливою складовою сучасних хмарних платформ та систем автоматичного масштабування [12].

Значний поштовх розвитку контейнеризації надала поява платформи Docker, яка стандартизувала процес створення та запуску контей-

нерів. Це сприяло широкому використанню мікросервісної архітектури, у якій окремі компоненти застосунку функціонують незалежно один від одного та можуть масштабуватися окремо залежно від навантаження. Основні принципи контейнеризації в хмарних середовищах наведено на рис. 1.5.

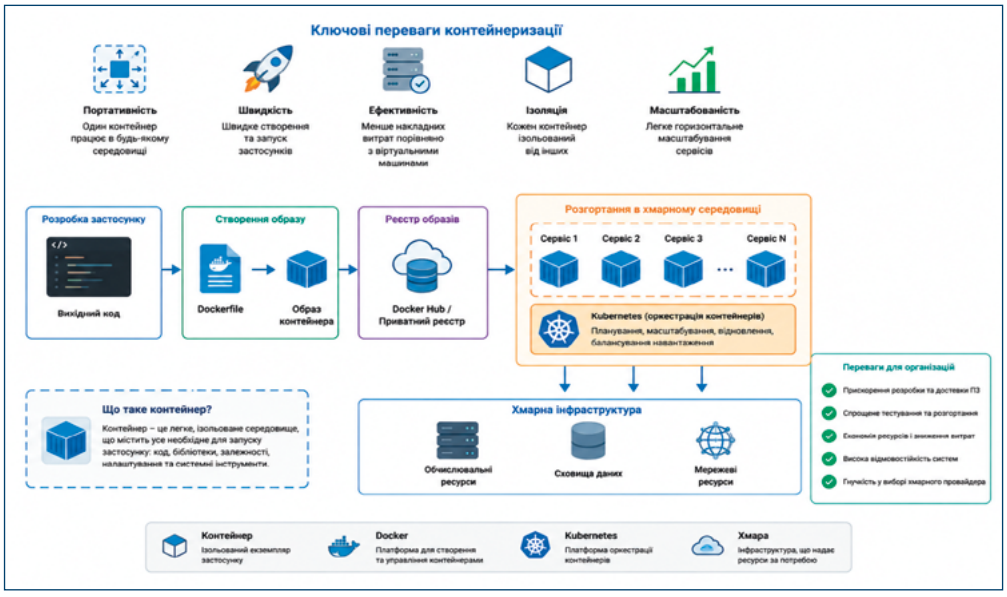


Рис. 1.5. Архітектура та основні принципи контейнеризації у хмарних обчисленнях

У сучасних хмарних середовищах кількість контейнерів може досягати сотень і тисяч екземплярів, що істотно ускладнює їх ручне адміністрування та зумовлює необхідність використання засобів автоматизованого керування контейнеризованими застосунками. Для вирішення цих завдань застосовуються платформи оркестрації, серед яких широкого поширення набула Kubernetes [13]. Вона забезпечує автоматизоване розгортання та масштабування сервісів, балансування навантаження, контроль стану контейнерів, керування їх життєвим циклом і відновлення працездатності сервісів у разі виникнення відмов.

Поєднання хмарних технологій, віртуалізації, контейнеризації та засобів оркестрації створює технологічну основу для побудови гнучких, масштабованих і відмовостійких РІКС. Контейнеризований підхід дає змогу розподіляти програмні компоненти між множиною обчислювальних вузлів, оперативно змінювати кількість їх екземплярів та перероз-

поділяти ресурси відповідно до поточного навантаження. Особливого значення ці можливості набувають у системах із динамічною інтенсивністю запитів, де потреба в обчислювальних і мережевих ресурсах може істотно змінюватися в часі [14].

Таким чином, хмарна модель не обмежується перенесенням обчислень на віддалену інфраструктуру, а передбачає принципово інший підхід до надання, використання та управління ресурсами. Її характерними рисами є еластичність і масштабованість, автоматизація процесів управління, відмовостійкість, можливість динамічного виділення ресурсів та їх використання відповідно до фактичних потреб. Основні характеристики хмарних обчислень, що мають визначальне значення для побудови та функціонування РІКС, узагальнено на рис. 1.6.



Рис. 1.6. Основні характеристики хмарних обчислень

Однією з ключових характеристик хмарного середовища є еластичність, яка полягає у можливості динамічно збільшувати або зменшувати обсяг виділених ресурсів відповідно до зміни навантаження. Тісно пов'язаною з нею є масштабованість, що характеризує здатність системи підтримувати необхідний рівень продуктивності при зміні кількості користувачів, інтенсивності інформаційних потоків або обсягів оброблюваних даних. На відміну від традиційної інфраструктури,

де нарощування ресурсів часто потребує встановлення додаткового фізичного обладнання, хмарні середовища дозволяють виконувати такі операції програмними засобами.

Важливою властивістю є також автоматизація управління ресурсами. Процеси розгортання, конфігурування, моніторингу, балансування навантаження та масштабування можуть виконуватися автоматизовано на основі визначених політик і поточного стану системи. Це створює передумови для переходу від статичного виділення ресурсів до адаптивного управління ними відповідно до зміни умов функціонування РІКС.

Відмовостійкість хмарних систем забезпечується можливістю резервування та розподілення сервісів і даних між декількома обчислювальними вузлами або територіально відокремленими центрами оброблення даних. У разі відмови окремого компонента його функції можуть бути перенесені на інші доступні ресурси, що сприяє підтриманню безперервності надання сервісів.

Економічна перевага хмарної моделі пов'язана з можливістю використання ресурсів відповідно до фактичної потреби та застосуванням моделей оплати за їх споживання. Це дозволяє зменшити необхідність попереднього резервування значних обчислювальних потужностей і підвищити ефективність використання інфраструктури.

Водночас розподілений характер хмарної інфраструктури підвищує вимоги до безпеки інформації та сервісів. Необхідним є забезпечення конфіденційності, цілісності та доступності даних, автентифікації й авторизації користувачів, захищеної взаємодії між компонентами, а також постійного моніторингу стану системи.

Розглянуті властивості хмарних технологій безпосередньо впливають на принципи побудови розподілених інформаційно-комунікаційних систем. Можливість динамічного виділення та масштабування ресурсів, віртуалізація обчислювальної інфраструктури, контейнеризація й автоматизована оркестрація створюють умови для розподілення функцій системи між множиною фізично та логічно відокремлених вузлів. При цьому оброблення і зберігання даних, виконання прикладних сервісів та передавання інформаційних потоків можуть здійснюватися на різних рівнях інфраструктури від кінцевих і периферійних пристроїв до регіональних вузлів та хмарних центрів оброблення даних. Це зумов-

лює формування багаторівневої архітектури РІКС, у якій мережеві, обчислювальні, програмні та інформаційні ресурси функціонують як взаємопов'язані складові єдиного середовища.

Узагальнену багаторівневу архітектуру розподіленої інформаційно-комунікаційної системи наведено на рис. 1.7. Її основу становить сукупність територіально та логічно розподілених обчислювальних вузлів, мережевих ресурсів, сховищ даних і програмних сервісів, взаємодія між якими здійснюється через комунікаційну інфраструктуру.

Розподіленість РІКС проявляється на декількох взаємопов'язаних рівнях. На клієнтському рівні формуються запити користувачів і кінцевих пристроїв, які через розподілені вузли доступу та комунікації надходять до відповідних сервісів. Прикладні функції реалізуються множиною сервісів або мікросервісів, розміщених на різних обчислювальних вузлах, а дані можуть зберігатися та оброблятися в територіально розподілених базах даних, сховищах і системах кешування. Інфраструктурний рівень, своєю чергою, об'єднує фізичні та віртуальні сервери, контейнеризовані середовища і мережеві ресурси, розміщені в різних центрах оброблення даних, хмарних та периферійних середовищах.

Взаємодія між зазначеними рівнями забезпечує передавання інформаційних потоків, доступ до сервісів та спільне використання розподілених ресурсів. Окрему функціональну складову становить підсистема управління та моніторингу, яка забезпечує збір інформації про стан компонентів, контроль використання ресурсів, керування конфігурацією, виявлення відмов та координацію роботи розподілених вузлів. На основі отриманих даних можуть прийматися рішення щодо перерозподілу ресурсів, балансування навантаження, розміщення сервісів та їх масштабування відповідно до поточних умов функціонування системи.

Таким чином, багаторівнева архітектура РІКС відображає одночасно фізичну розподіленість ресурсів і функціональну розподіленість процесів передавання, оброблення та зберігання інформації, а також надання цифрових сервісів. Така організація створює передумови для забезпечення масштабованості, еластичності, відмовостійкості та необхідної якості обслуговування в умовах динамічної зміни навантаження.

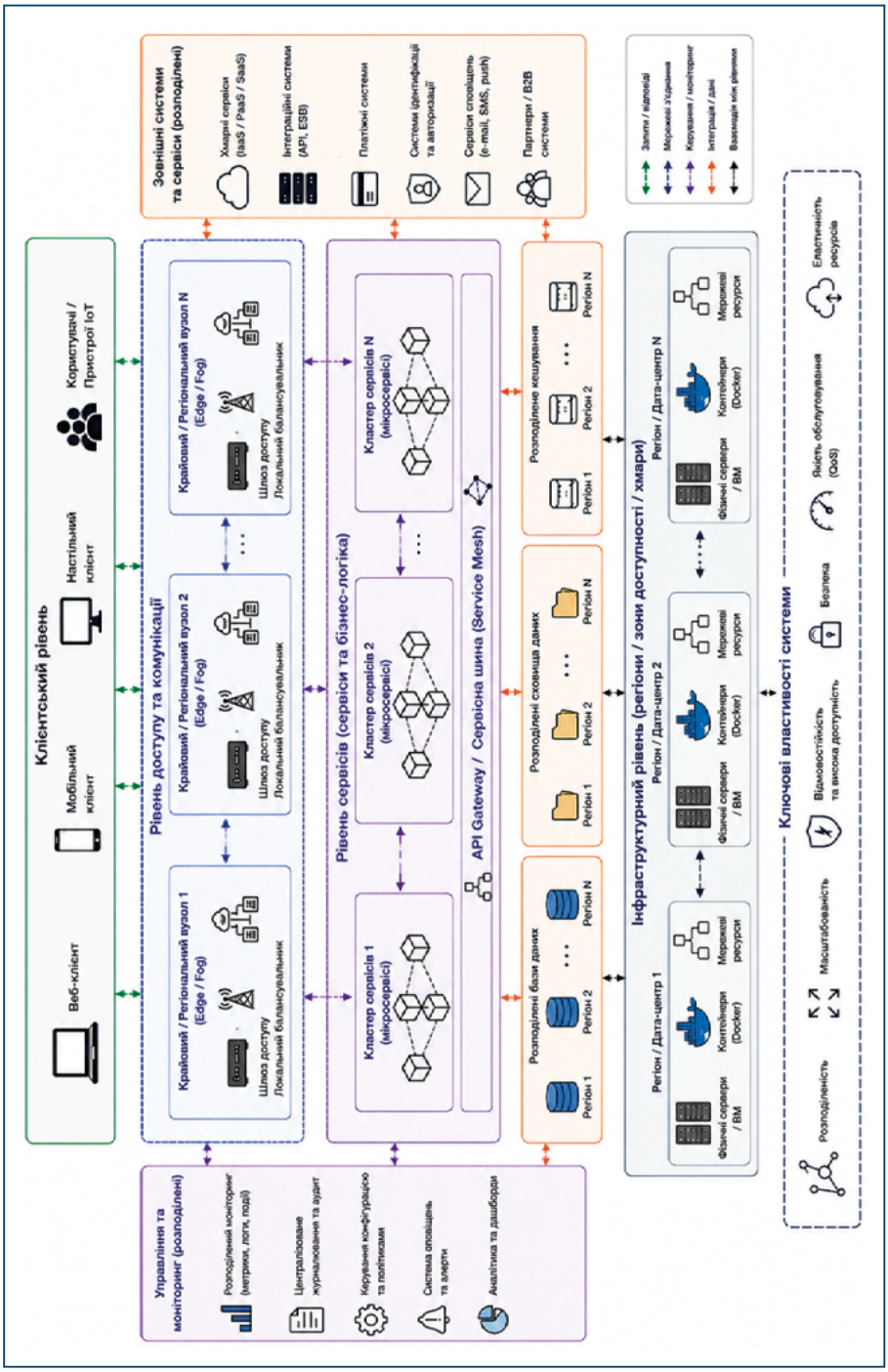


Рис. 1.7. Узагальнена багаторівнева архітектура розподіленої інформаційно-комунікаційної системи

1.3. Мікросервісна архітектура та оркестрація контейнеризованих сервісів

Мікросервісна архітектура є підходом до побудови програмних систем, за якого функціональність застосунку декомпозується на сукупність відносно незалежних сервісів. Кожен мікросервіс реалізує визначену функцію або групу взаємопов'язаних функцій і може розроблятися, розгортатися, оновлюватися та масштабуватися незалежно від інших компонентів системи. Взаємодія між мікросервісами здійснюється через визначені програмні інтерфейси (API) або механізми обміну повідомленнями, що забезпечує слабку зв'язаність окремих компонентів і дає змогу використовувати для їх реалізації різні програмні технології [15].

На відміну від монолітної архітектури, у якій основні функціональні компоненти застосунку об'єднані в єдине програмне середовище, мікросервісний підхід передбачає їх функціональне та логічне відокремлення. У розподіленому середовищі окремі мікросервіси можуть виконуватися на різних фізичних або віртуальних обчислювальних вузлах, що безпосередньо відповідає принципам побудови РІКС. Така організація дає змогу незалежно керувати ресурсами окремих компонентів, перерозподіляти їх між вузлами та адаптувати конфігурацію системи відповідно до зміни навантаження.

Важливою перевагою мікросервісної архітектури є можливість незалежного масштабування окремих сервісів. У разі збільшення навантаження на певну функціональну складову системи додаткові ресурси можуть виділятися саме для відповідного мікросервісу без необхідності масштабування всього застосунку. Це створює передумови для більш раціонального використання обчислювальних ресурсів і забезпечує гнучкість системи за нерівномірного розподілу навантаження між її функціональними компонентами [16].

Практична реалізація мікросервісної архітектури значною мірою пов'язана з використанням технологій контейнеризації. Розміщення окремих мікросервісів у контейнерах забезпечує ізоляцію середовища їх виконання, спрощує перенесення між обчислювальними вузлами та дає

змогу оперативно створювати або видаляти додаткові екземпляри сервісів. Саме поєднання мікросервісної архітектури та контейнеризації сформувало технологічну основу для побудови гнучких хмарних і розподілених застосунків.

Водночас зі збільшенням кількості мікросервісів і їх екземплярів істотно ускладнюються процеси управління розподіленим програмним середовищем. Необхідно визначати вузли для розміщення контейнерів, контролювати стан запущених сервісів, забезпечувати їх мережеву взаємодію, балансування навантаження, масштабування та відновлення у разі відмов. Крім того, динамічне створення і видалення екземплярів сервісів потребує постійного контролю стану обчислювальних ресурсів та узгодження їх використання з поточними потребами застосунків.

Для автоматизації зазначених процесів використовуються системи оркестрації контейнеризованих сервісів. Оркестрація передбачає координоване управління розгортанням, розміщенням, масштабуванням і життєвим циклом контейнерів у розподіленому обчислювальному середовищі. Системи оркестрації забезпечують автоматизований вибір доступних обчислювальних вузлів, підтримання заданої кількості екземплярів сервісів, контроль їх стану, балансування навантаження та відновлення працездатності після відмов.

Однією з найбільш поширених платформ оркестрації контейнеризованих застосунків є Kubernetes. Її використання дає змогу об'єднати множину фізичних або віртуальних обчислювальних вузлів у кластер і забезпечити централізоване керування розміщеними на них контейнеризованими сервісами. Kubernetes автоматизує процеси розгортання контейнерів, контролю їх стану, балансування навантаження, масштабування та відновлення, створюючи таким чином механізми адаптивного управління ресурсами розподіленого середовища.

Особливого значення використання систем оркестрації набуває в РІКС із динамічним характером навантаження. Зміна інтенсивності запитів користувачів та інформаційних потоків призводить до зміни потреб окремих сервісів у процесорних ресурсах, оперативній пам'яті, пропускній здатності мережі та інших ресурсах. За таких умов статичне виділення ресурсів може спричиняти як перевантаження окремих ком-

понентів, так і неефективне використання надлишково зарезервованих потужностей. Тому механізми моніторингу, розміщення сервісів і автоматичного масштабування є важливими складовими управління контейнеризованими РІКС.

Порівняння монолітного та мікросервісного підходів до організації застосунків наведено на рис. 1.8. У монолітній архітектурі функціональні компоненти об'єднані в межах одного застосунку та переважно масштабуються як єдине ціле [17]. Мікросервісна архітектура передбачає декомпозицію застосунку на окремі сервіси, які можуть бути розподілені між різними обчислювальними вузлами та масштабуватися незалежно відповідно до характеру навантаження.

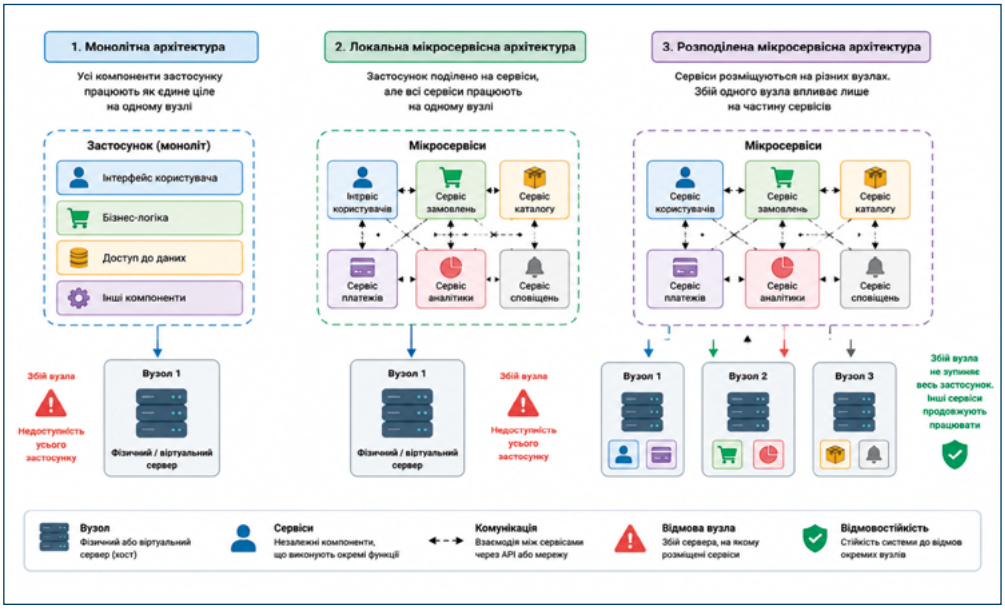


Рис. 1.8. Порівняння архітектурних підходів до розгортання мікросервісних застосунків

Контейнеризація стала основою розвитку мікросервісної архітектури, оскільки забезпечує ізоляцію застосунків, портативність та однакові умови виконання на різних платформах. Контейнери дозволяють швидко запускати сервіси та ефективно використовувати ресурси серверів. Завдяки невеликому розміру контейнерних образів і швидкому часу запуску стало можливим оперативне масштабування застосунків відповідно до зміни навантаження [18].

Зі збільшенням кількості контейнерів зростає складність їх адміністрування. Для забезпечення стабільної роботи систем необхідно контролювати конфігурації, мережеві взаємодії, стан сервісів і використання ресурсів. Тому значна увага приділяється автоматизації керування інфраструктурою за допомогою таких інструментів, як Helm, Terraform та інших платформ декларативного керування.

Kubernetes є платформою оркестрації контейнерів, яка забезпечує автоматизоване керування контейнеризованими застосунками у розподілених середовищах. Кластер Kubernetes складається з робочих вузлів (worker nodes), на яких виконуються застосунки, та керуючої площини (control plane), що відповідає за координацію, моніторинг і прийняття рішень щодо роботи всієї системи. Для користувача кластер виступає єдиною платформою, яка автоматично забезпечує балансування навантаження, масштабування та відмовостійкість сервісів. Архітектура Kubernetes сформувалася на основі внутрішніх систем компанії Google, зокрема Borg та Omega, які використовувалися для управління великими обчислювальними кластерами. Завдяки цьому Kubernetes від самого початку був орієнтований на автоматизацію, масштабованість та високу надійність. Сьогодні платформа підтримується більшістю провідних хмарних провайдерів, серед яких Amazon Web Services, Google Cloud та Microsoft Azure [19].

Kubernetes побудований відповідно до принципів мікросервісної архітектури. Система складається з окремих компонентів, кожен із яких виконує власну функцію та взаємодіє через стандартизовані API. Такий підхід забезпечує модульність, гнучкість та можливість незалежного оновлення окремих компонентів без порушення роботи всього кластеру. Важливою особливістю Kubernetes є розширюваність. Платформа підтримує додавання власних типів ресурсів через механізм CRD (Custom Resource Definitions) та інтеграцію із зовнішніми сервісами. Це дозволяє використовувати Kubernetes не лише для керування контейнерами, а й для централізованого управління різними елементами інфраструктури.

У Kubernetes фізичні або віртуальні сервери називаються вузлами (nodes), а сукупність вузлів утворює кластер. Користувач задає бажаний стан системи декларативно, після чого Kubernetes автоматично підтри-

мує цей стан, аналізуючи поточну конфігурацію та виконуючи необхідні дії для її коригування [20]. Основними об'єктами Kubernetes є Pod-и та Service-и. Pod є найменшою одиницею розгортання та може містити один або декілька контейнерів, які працюють спільно й використовують спільний мережевий простір та ресурси. Огляд основних об'єктів Kubernetes та взаємозв'язків між ними наведено на рис. 1.9.

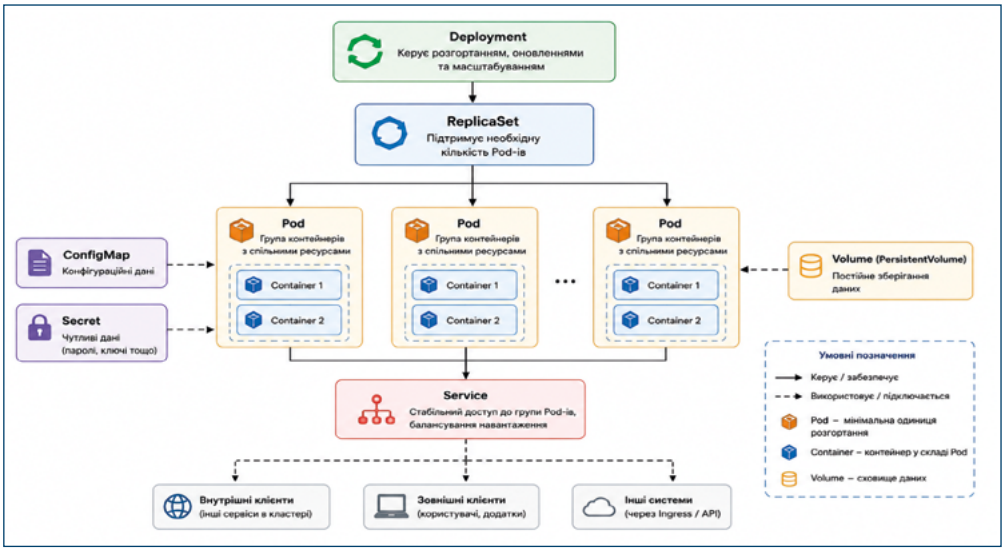


Рис. 1.9. Основні об'єкти платформи Kubernetes та схема їх взаємодії

Для керування ресурсами Kubernetes використовує механізми requests та limits. Requests визначають гарантований мінімальний обсяг ресурсів для контейнера, тоді як limits задають максимально допустимий рівень використання процесора та пам'яті. Це дозволяє ефективно розподіляти навантаження між вузлами та уникати перевантаження системи.

Підтримка необхідної кількості Pod-ів здійснюється за допомогою об'єктів ReplicaSet та Deployment. Deployment забезпечує автоматичне оновлення застосунків, масштабування сервісів і відновлення після помилок. Завдяки цьому можна виконувати оновлення без зупинки роботи сервісів та швидко повертатися до попередніх версій у разі виникнення проблем.

Оскільки Pod-и можуть створюватися та видалятися динамічно, їхні IP-адреси змінюються. Для забезпечення стабільного доступу ви-

користовується об'єкт Service, який виконує функції service discovery та балансування навантаження між Pod-ами. Для застосунків, що потребують збереження стану, використовується StatefulSet. Він забезпечує сталі ідентифікатори, контрольований порядок запуску та підтримку постійних сховищ даних, що є важливим для баз даних та інших stateful-сервісів.

Керуюча площина (Control Plane) відповідає за управління кластером та включає API Server, Scheduler, Controller Manager та базу даних etcd. Площина даних (Data Plane) виконує робочі навантаження на вузлах. Архітектуру Kubernetes наведено на рис. 1.10.

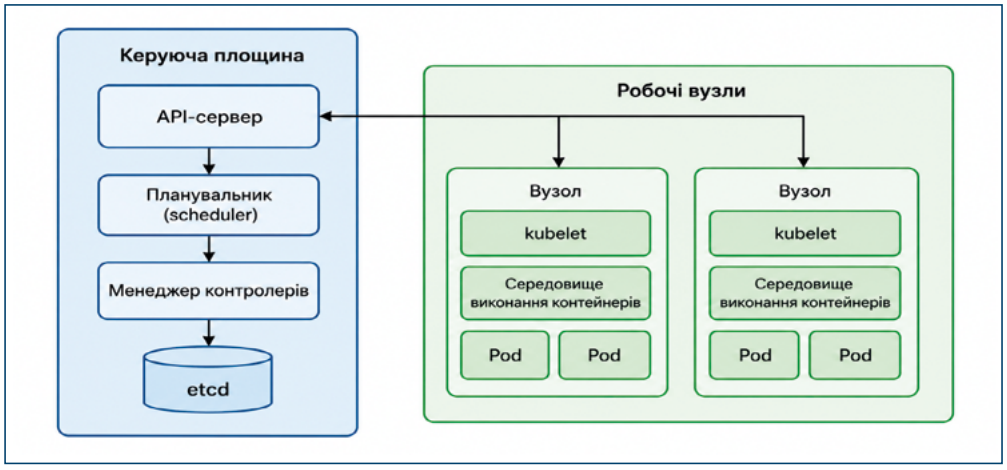


Рис. 1.10. Архітектурна модель Kubernetes

На кожному вузлі працює компонент kubelet, який взаємодіє з control plane, запускає контейнери та контролює їх стан. Для збору інформації про використання ресурсів застосовується cAdvisor, який надає дані про навантаження процесора, пам'яті, мережі та дискової підсистеми.

Важливу роль у Kubernetes відіграють контролери, які реалізують механізм control loops. Вони постійно порівнюють поточний стан системи з бажаним та автоматично виконують необхідні дії для синхронізації. Центральною точкою доступу до всіх компонентів є API Server, а зберігання конфігурацій та стану кластера забезпечує розподілене сховище etcd. Для моніторингу поточного навантаження використовується Metrics Server, який надає метрики для механізмів автоматичного масш-

табування, зокрема Horizontal Pod Autoscaler (HPA). Для зберігання історичних даних і побудови аналітики зазвичай застосовуються системи моніторингу, такі як Prometheus [21].

Таким чином, Kubernetes є комплексною платформою оркестрації, яка поєднує принципи контейнеризації, мікросервісної архітектури та автоматизованого управління ресурсами. Використання Kubernetes дозволяє створювати масштабовані, відмовостійкі та гнучкі хмарні системи, здатні ефективно адаптуватися до змін навантаження.

Основні принципи розгортання контейнерів у середовищі Kubernetes наведено на рис. 1.11.

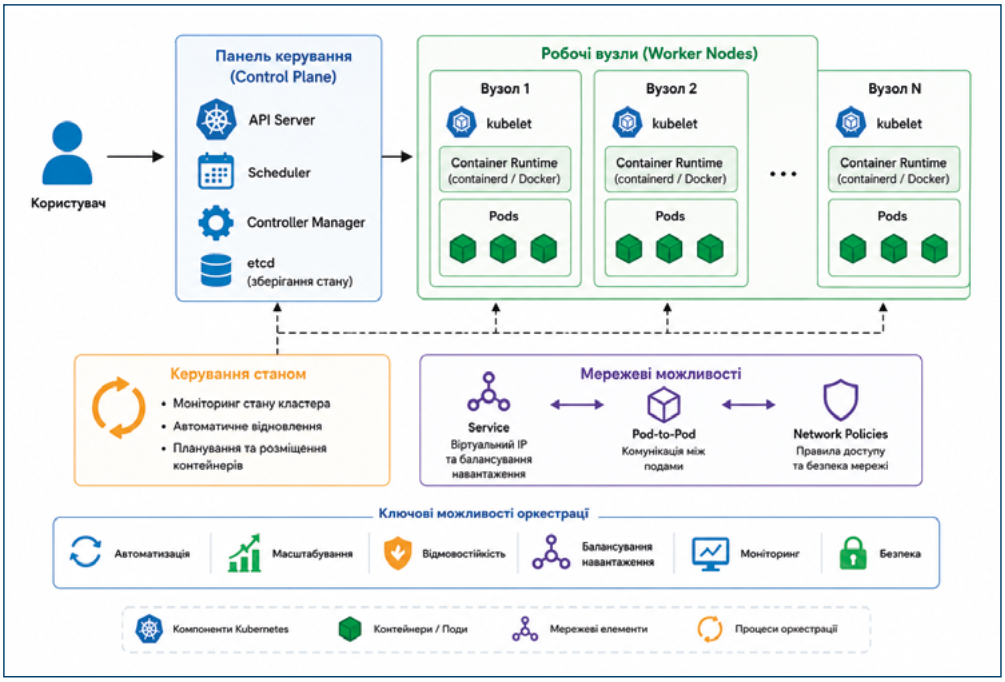


Рис. 1.11. Розгортання контейнеризованих застосунків у Kubernetes

Платформа автоматично підтримує необхідну кількість екземплярів сервісів, контролює їх роботу та забезпечує перезапуск контейнерів у випадку помилок.

Однією з важливих функціональних можливостей Kubernetes є автоматичне масштабування контейнеризованих сервісів відповідно до зміни навантаження та стану доступних ресурсів. Для цього платформа підтримує механізми горизонтального масштабування подів (Horizontal

Pod Autoscaler, HPA), вертикального масштабування ресурсів контейнерів (Vertical Pod Autoscaler, VPA), а також масштабування, ініційованого зовнішніми подіями або прикладними метриками. Застосування таких механізмів дає змогу динамічно адаптувати обсяг залучених обчислювальних ресурсів до поточних потреб сервісів і підтримувати необхідний рівень їх продуктивності [22].

Функціонування мікросервісів у розподіленому контейнеризованому середовищі потребує також організації надійної мережевої взаємодії між окремими компонентами. У Kubernetes для цього використовуються мережеві механізми та спеціалізовані програмні засоби, які забезпечують адресацію і маршрутизацію трафіку, балансування навантаження, керування мережевими політиками та захищену взаємодію між сервісами. Залежно від архітектури системи ці функції можуть реалізовуватися за допомогою мережевих плагінів і технологій сервісної сітки (service mesh) [24].

Важливою складовою управління контейнеризованою інфраструктурою є моніторинг її поточного стану. Збір і аналіз метрик використання процесорних ресурсів, оперативної пам'яті, мережевої активності, кількості запитів та інших показників дають змогу оцінювати навантаження на окремі сервіси й обчислювальні вузли. Для реалізації таких функцій можуть застосовуватися засоби моніторингу та візуалізації, зокрема Prometheus і Grafana [25]. Отримана інформація формує основу для прийняття рішень щодо балансування навантаження, перерозподілу ресурсів і автоматичного масштабування сервісів.

Таким чином, поєднання мікросервісної архітектури, контейнеризації та засобів оркестрації створює технологічну основу для побудови масштабованих і адаптивних РІКС. Водночас ефективність функціонування таких систем значною мірою визначається не лише можливістю автоматизованого розгортання сервісів, а й здатністю своєчасно оцінювати поточний стан інфраструктури, визначати необхідний обсяг ресурсів та адаптувати їх розподіл відповідно до зміни навантаження. Це зумовлює необхідність застосування механізмів моніторингу, автоматичного масштабування та раціонального розміщення контейнеризованих сервісів у розподіленому обчислювальному середовищі.

1.4. Методи горизонтального та вертикального автомасштабування ресурсів розподілених систем на основі Kubernetes

Автоматичне масштабування ресурсів є одним із основних механізмів адаптації розподілених інформаційно-комунікаційних систем до динамічної зміни навантаження. Його застосування передбачає автоматизовану зміну обсягу залучених обчислювальних ресурсів або кількості екземплярів сервісів відповідно до поточного стану системи та інтенсивності інформаційних потоків. Такий підхід дає змогу підтримувати необхідний рівень продуктивності та якості обслуговування за одночасного підвищення ефективності використання ресурсів [26].

Необхідність автоматичного масштабування зумовлена динамічним характером функціонування РІКС. Інтенсивність запитів користувачів, обсяги переданих і оброблюваних даних, завантаження мережевих каналів та обчислювальних вузлів можуть істотно змінюватися в часі. За статичного виділення ресурсів це може призводити до їх дефіциту в періоди пікового навантаження або, навпаки, до надлишкового резервування обчислювальних потужностей за низької інтенсивності використання системи. Тому механізми масштабування повинні забезпечувати узгодження доступного обсягу ресурсів із поточними потребами сервісів.

У контейнеризованих РІКС реалізація автоматичного масштабування тісно пов’язана із засобами оркестрації, які забезпечують створення, видалення та перерозподіл екземплярів сервісів між обчислювальними вузлами. Одним із середовищ реалізації таких механізмів є Kubernetes, у якому керування станом системи здійснюється відповідно до декларативної моделі. Для контейнеризованих сервісів задається бажаний стан, а засоби оркестрації забезпечують його підтримання шляхом виконання відповідних операцій над компонентами системи.

Залежно від способу зміни доступних ресурсів розрізняють горизонтальне та вертикальне масштабування. Горизонтальне масштабування (scale-out/scale-in) передбачає зміну кількості паралельно функціонуючих екземплярів сервісу, тоді як вертикальне масштабування (scale-up/

scale-down) полягає у зміні обсягу ресурсів, виділених окремому екземпляру. Вибір відповідного способу визначається архітектурою застосунку, характером навантаження, доступністю ресурсів та вимогами до продуктивності системи.

Горизонтальне масштабування передбачає збільшення або зменшення кількості екземплярів сервісу залежно від поточного навантаження. У разі його зростання створюються додаткові екземпляри, між якими розподіляється потік запитів, а після зменшення навантаження надлишкові екземпляри можуть бути вилучені. Такий підхід особливо ефективний для мікросервісних застосунків без збереження локального стану (stateless), оскільки дає змогу оперативно змінювати кількість активних екземплярів без істотної зміни їх конфігурації.

У середовищі Kubernetes горизонтальне масштабування може реалізовуватися за допомогою механізму Horizontal Pod Autoscaler (HPA), який змінює кількість Pod-ів відповідно до визначених показників стану системи [28]. Як вхідні параметри можуть використовуватися показники завантаження процесора та пам'яті, а також прикладні й зовнішні метрики. У загальному випадку рішення щодо масштабування може ґрунтуватися на сукупності характеристик, що відображають стан обчислювальних і мережевих ресурсів РІКС. Принцип горизонтального масштабування ресурсів наведено на рис. 1.12.

У Kubernetes горизонтальне масштабування реалізується за допомогою механізму Horizontal Pod Autoscaler (HPA). HPA аналізує метрики використання CPU, пам'яті та інших ресурсів і автоматично змінює кількість Pod-ів залежно від навантаження. У разі зростання навантаження створюються нові екземпляри сервісу, а під час зменшення активності їх кількість скорочується для економії ресурсів.

Попри високу еластичність, горизонтальне масштабування має певні обмеження. Для stateful-застосунків, таких як бази даних або системи з локальним станом, масштабування ускладнюється необхідністю синхронізації та реплікації даних. Крім того, для ефективного розподілу навантаження потрібні механізми балансування трафіку та сервісні mesh-рішення, зокрема Istio або Linkerd.

Вертикальне масштабування (scale-up/scale-down) передбачає зміну обсягу ресурсів, виділених одному Pod-у або контейнеру. У цьому випадку

збільшується або зменшується кількість CPU, оперативної пам'яті чи інших ресурсів без створення нових екземплярів застосунку. У Kubernetes цей підхід реалізується через Vertical Pod Autoscaler (VPA).

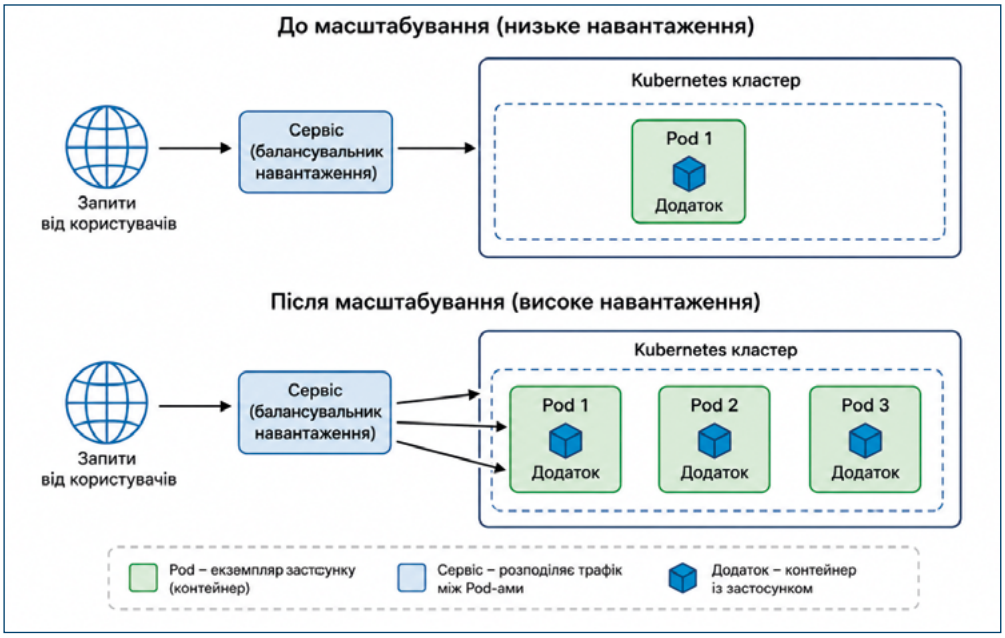


Рис. 1.12. Схема горизонтального масштабування ресурсів

VPA аналізує використання ресурсів і автоматично коригує параметри requests та limits. На відміну від HPA, який реагує на короткострокові зміни навантаження, вертикальне масштабування частіше орієнтується на довготривалі тенденції використання ресурсів. Такий підхід є ефективним для монолітних застосунків, систем аналітики, JVM-застосунків або моделей машинного навчання, які складно масштабувати горизонтально. Принцип вертикального масштабування ресурсів наведено на рис. 1.13.

Основним обмеженням вертикального масштабування є фізичні ресурси вузла. Кількість CPU та пам'яті, доступна одному Pod-у, не може перевищувати можливості сервера або типу хмарного інстансу. Саме тому горизонтальне масштабування вважається більш еластичним підходом для високонавантажених distributed-систем.

У розподілених інформаційно-комунікаційних системах горизонтальне та вертикальне масштабування можуть застосовуватися як окремо,

так і в межах комбінованого підходу до управління ресурсами [29]. У такому разі рішення щодо зміни кількості екземплярів сервісу та обсягу ресурсів, виділених кожному з них, приймаються узгоджено з урахуванням поточного навантаження, доступної ресурсної ємності обчислювальних вузлів та встановлених вимог до продуктивності. Поєднання різних механізмів масштабування розширює можливості адаптації РІКС до зміни умов функціонування та дає змогу ефективніше використовувати доступні ресурси.

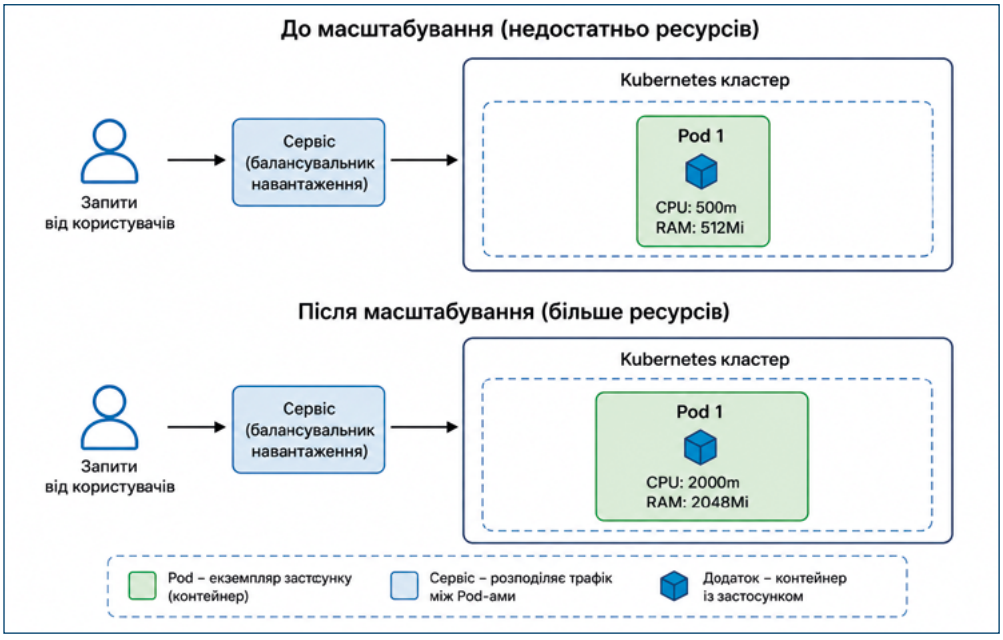


Рис. 1.13. Схема вертикального масштабування ресурсів

Важливою перевагою автоматичного масштабування є можливість узгодження обсягу залучених ресурсів із фактичною потребою в них. У періоди підвищеного навантаження система може збільшувати ресурсне забезпечення окремих сервісів або кількість їх екземплярів, тоді як зі зменшенням інтенсивності навантаження надлишкові ресурси можуть вивільнятися. Такий механізм дає змогу зменшити непродуктивне резервування обчислювальних потужностей і водночас забезпечити необхідний рівень продуктивності за пікових навантажень.

Автоматичне масштабування тісно пов’язане з іншими механізмами адаптивного управління РІКС. Зокрема, у контейнеризованих середовищах зміна кількості екземплярів сервісу потребує їх розміщення на до-

ступних обчислювальних вузлах з урахуванням поточного використання ресурсів. У разі відмови окремих компонентів засоби оркестрації забезпечують відновлення необхідного стану системи шляхом перезапуску або повторного розміщення контейнеризованих сервісів. Таким чином, масштабування, розміщення сервісів і механізми самовідновлення утворюють взаємопов'язані складові управління ресурсами розподіленого середовища.

Необхідною умовою реалізації автоматичного масштабування є отримання актуальної інформації про стан системи. Для цього здійснюється моніторинг завантаження процесорних ресурсів, використання оперативної пам'яті, інтенсивності мережевого трафіку, кількості запитів, затримок обслуговування та інших показників, що характеризують функціонування РІКС. У контейнеризованих і хмарних середовищах для збору та опрацювання відповідних метрик можуть використовуватися спеціалізовані системи моніторингу, зокрема Prometheus, а також засоби моніторингу, інтегровані в інфраструктуру хмарних платформ.

Водночас використання лише поточних значень метрик означає, що рішення щодо масштабування переважно приймається після фіксації зміни стану системи. За різкого зростання навантаження затримка між його виявленням, прийняттям рішення та фактичним введенням додаткових ресурсів може призводити до тимчасового погіршення продуктивності та якості обслуговування. Тому перспективним напрямом розвитку механізмів управління ресурсами РІКС є використання методів прогнозування, які дають змогу оцінювати майбутню зміну навантаження та завчасно формувати керуючі впливи щодо необхідного масштабування [30].

1.5. Моніторинг та оцінювання стану ресурсів розподілених інформаційно-комунікаційних систем

Моніторинг стану розподілених інформаційно-комунікаційних систем ґрунтується на зборі та аналізі кількісних показників, що характеризують використання ресурсів, продуктивність окремих компонентів і

якість надання сервісів. Отримані метрики дають змогу оцінювати поточний стан системи, виявляти перевантаження та аномальні режими функціонування, визначати дефіцит ресурсів і формувати інформаційну основу для прийняття рішень щодо їх перерозподілу та автоматичного масштабування.

Важливу групу становлять ресурсні метрики, які характеризують стан обчислювальних, мережевих і систем зберігання даних. До них належать рівень використання процесорних ресурсів, обсяг зайнятої та доступної оперативної пам'яті, інтенсивність операцій введення-виведення, використання дискового простору, пропускна здатність мережевих інтерфейсів, обсяг переданих даних та інші показники. Їх комплексний аналіз дає змогу визначати ступінь завантаження окремих вузлів і сервісів, виявляти потенційні вузькі місця та оцінювати доступну ресурсну ємність розподіленої інфраструктури.

Поряд із ресурсними показниками важливе значення мають сервісні метрики, які безпосередньо характеризують функціонування прикладних компонентів РІКС. До них можуть належати інтенсивність надходження запитів, час їх оброблення, затримка надання сервісу, частота виникнення помилок та доступність окремих компонентів. Спільне використання ресурсних і сервісних метрик забезпечує більш повне оцінювання стану системи, оскільки дає змогу встановлювати взаємозв'язок між рівнем використання ресурсів та фактичною якістю обслуговування користувачів. Зібрані в процесі моніторингу показники переважно формуються у вигляді часових рядів, у яких кожному значенню метрики відповідає певний момент часу. Накопичення таких даних дає змогу досліджувати динаміку навантаження, виявляти періодичні закономірності й аномальні зміни, а також використовувати історичні спостереження для прогнозування майбутнього стану системи. Це має особливе значення для реалізації проактивних методів управління ресурсами, за яких рішення щодо масштабування можуть прийматися не лише на основі поточних значень метрик, а й з урахуванням прогнозованої зміни навантаження.

Оцінювання якості функціонування сервісів може здійснюватися із застосуванням індикаторів рівня сервісу (Service Level Indicator, SLI), цільових рівнів сервісу (Service Level Objective, SLO) та угод про рівень сер-

вісу (Service Level Agreement, SLA). Їх використання дає змогу формалізувати вимоги до доступності, продуктивності та інших характеристик обслуговування і враховувати відповідні обмеження під час прийняття рішень щодо управління ресурсами РІКС.

Для автоматизованого збору, зберігання та аналізу метрик у контейнеризованих середовищах можуть використовуватися спеціалізовані системи моніторингу. Одним із поширених засобів є Prometheus, який забезпечує збір і зберігання метрик у вигляді часових рядів та їх подальше опрацювання. Для візуалізації накопичених даних і формування інформаційних панелей може використовуватися Grafana, що забезпечує графічне представлення динаміки контрольованих показників та спрощує аналіз стану окремих компонентів і системи загалом. Prometheus переважно використовує модель збору даних типу pull, за якої сервер моніторингу періодично отримує значення метрик із визначених джерел. Для інтеграції компонентів, які безпосередньо не надають метрики у відповідному форматі, використовуються експортери. Вони забезпечують отримання показників від операційних систем, баз даних, мережевих компонентів та інших програмних засобів і їх подальше передавання до системи моніторингу. За необхідності можуть застосовуватися спеціалізовані експортери для отримання метрик, специфічних для конкретної РІКС.

У контейнеризованому середовищі засоби моніторингу можуть бути інтегровані з платформою Kubernetes, що дає змогу централізовано отримувати інформацію про стан вузлів, контейнеризованих сервісів і використовуваних ними ресурсів. Накопичені метрики надалі можуть використовуватися не лише для оперативного контролю стану інфраструктури, а й як вхідні дані для механізмів автоматичного масштабування, алгоритмів прогнозування навантаження та методів адаптивного управління ресурсами РІКС. Архітектуру системи моніторингу з використанням Prometheus та Grafana наведено на рис. 1.14.

Для організації моніторингу контейнеризованих сервісів у середовищі Kubernetes важливим є забезпечення контрольованої мережевої взаємодії між компонентами системи та джерелами метрик. Керування мережевим доступом може здійснюватися за допомогою політик Network Policies, які визначають допустимі з’єднання між компонентами

кластера та забезпечують необхідний рівень ізоляції під час отримання моніторингових даних.

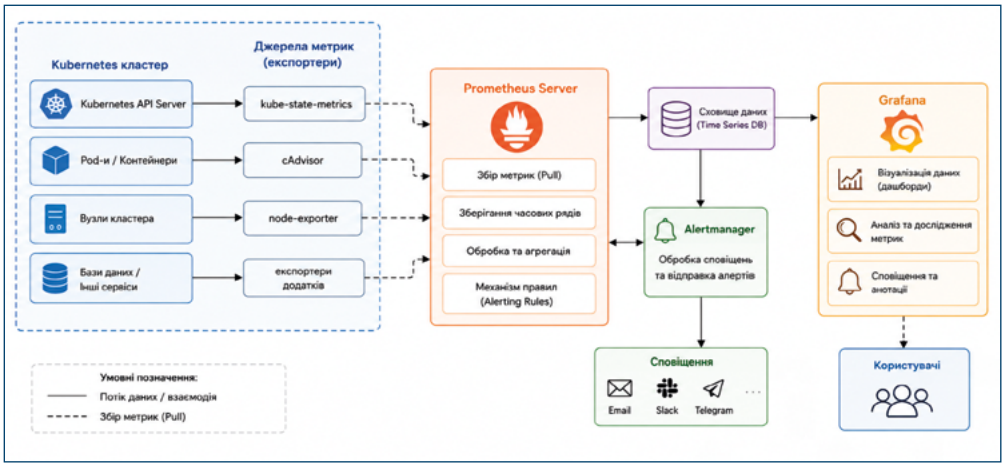


Рис. 1.14. Архітектура системи моніторингу на базі Prometheus і Grafana

Автоматизація виявлення джерел метрик забезпечується механізмами service discovery, підтримуваними Prometheus. Їх використання дає змогу динамічно формувати та оновлювати перелік об'єктів моніторингу відповідно до поточного стану контейнеризованого середовища. Це має особливе значення для РІКС, у яких унаслідок масштабування, перерозподілу ресурсів або відновлення після відмов кількість і розміщення екземплярів сервісів можуть змінюватися в процесі функціонування. Інтеграція Prometheus із Kubernetes забезпечує автоматизоване виявлення відповідних об'єктів кластера та підключення необхідних джерел метрик відповідно до заданої конфігурації моніторингу.

Для комплексного оцінювання стану РІКС доцільно здійснювати збір метрик на різних рівнях її інфраструктури. На рівні контейнеризованого середовища контролюються показники стану контейнерів і Pod-ів, а також використання виділених їм обчислювальних ресурсів. На рівні фізичних або віртуальних обчислювальних вузлів здійснюється моніторинг завантаження процесора, використання оперативної пам'яті, дискової підсистеми та мережевої активності. Для отримання відповідних показників можуть застосовуватися спеціалізовані експортери, які забезпечують уніфіковане подання метрик для їх подальшого збирання та опрацювання засобами Prometheus.

Поєднання метрик різних рівнів дає змогу сформувати комплексне уявлення про поточний стан розподіленої інфраструктури, оцінювати характер використання ресурсів та виявляти потенційні перевантаження окремих компонентів. Накопичені значення метрик можуть бути представлені у вигляді часових рядів і використані як для оперативного моніторингу, так і для подальшого аналізу динаміки навантаження та прийняття рішень щодо адаптивного управління ресурсами РІКС. Приклад візуалізації динаміки використання процесорних ресурсів та оперативної пам'яті контейнеризованими сервісами засобами Grafana наведено на рис. 1.15.



Рис. 1.15. Скріншот Grafana з відображенням детальної інформації про використання процесора та пам'яті контейнерів

Для розширеного оцінювання стану контейнеризованих сервісів поряд із базовими ресурсними показниками можуть використовуватися спеціалізовані метрики, що характеризують функціонування окремих програмних компонентів РІКС. Зокрема, для систем обміну повідомленнями доцільним є моніторинг стану черг, інтенсивності надходження та оброблення повідомлень, кількості активних завдань і тривалості їх виконання. Аналіз таких показників дає змогу оцінювати характер роз-

поділу навантаження між мікросервісами, виявляти накопичення необроблених запитів та визначати потенційні перевантаження окремих компонентів системи.

Для отримання прикладних метрик, безпосередньо пов'язаних із логікою функціонування сервісів, можуть застосовуватися спеціалізовані експортери, які взаємодіють із програмними інтерфейсами застосунків та формують метрики у форматі, придатному для подальшого опрацювання засобами Prometheus. Залежно від характеру контрольованого показника можуть використовуватися різні типи метрик, зокрема Counter для накопичувальних величин, Gauge для показників, значення яких змінюються в часі, та Histogram для аналізу розподілу тривалості виконання операцій або оброблення запитів. Такий підхід дає змогу доповнити ресурсні показники інформацією про фактичний стан і продуктивність прикладних сервісів.

Поєднання ресурсних і прикладних метрик є особливо важливим для реалізації механізмів адаптивного управління ресурсами РІКС. Використання лише показників завантаження процесора або оперативної пам'яті не завжди достатньо відображає фактичний рівень навантаження на сервіс. Тому під час формування рішень щодо масштабування можуть додатково враховуватися інтенсивність запитів, довжина черги повідомлень, кількість активних або очікуючих завдань, час їх виконання та інші показники, що характеризують функціонування конкретного сервісу.

У середовищі Kubernetes використання таких метрик у механізмах горизонтального автомасштабування потребує їх інтеграції з відповідними інтерфейсами метрик платформи. Для цього може застосовуватися Prometheus Adapter, який забезпечує представлення зібраних Prometheus метрик через інтерфейси Kubernetes та створює можливість їх використання механізмом Horizontal Pod Autoscaler (HPA).

У результаті рішення щодо зміни кількості екземплярів контейнеризованого сервісу можуть прийматися не лише на основі стандартних ресурсних показників, а й з урахуванням прикладних характеристик поточного навантаження. Узагальнену схему формування та передавання метрик для автоматичного масштабування контейнеризованих сервісів наведено на рис. 1.16.



Рис. 1.16. Схема передачі метрик, що використовуються для масштабування: стрілками позначено логічний рух даних, а помаранчевими стрілками – необроблені HTTP-метрики

Рис. 1.16 демонструє повний процес використання кастомних метрик для автоматичного горизонтального масштабування застосунків у Kubernetes. На відміну від стандартного HPA, який може використовувати лише завантаження CPU або пам'яті, запропонована схема дозволяє масштабувати Pod-и на основі будь-яких бізнесових або прикладних показників, наприклад кількості HTTP-запитів, активних користувачів, довжини черги повідомлень, часу відповіді сервісу чи кількості транзакцій за секунду. Робота системи починається на рівні джерел метрик. Джерелами можуть виступати безпосередньо мікросервіси та Pod-и Kubernetes, які генерують інформацію про власний стан і навантаження. Крім того, метрики можуть формуватися спеціальними кастомними експортерами, що збирають дані із зовнішніх систем або прикладних сервісів. Також джерелом можуть бути сторонні інформаційні системи, бази даних, брокери повідомлень, системи моніторингу черг або інші компоненти інфраструктури. Усі ці елементи публікують свої метрики у форматі, придатному для подальшого збору.

Наступним компонентом є Prometheus, який виконує роль централізованої системи моніторингу. Він періодично опитує всі джерела за принципом *pull model*, тобто самостійно звертається до визначених endpoint-ів і отримує актуальні значення метрик. Після збору да-

них Prometheus зберігає їх у вигляді часових рядів, що дозволяє аналізувати динаміку зміни навантаження та використовувати історичні значення для моніторингу й візуалізації. Оскільки Kubernetes не може напряму працювати з внутрішнім форматом Prometheus, наступним елементом архітектури є Prometheus Adapter. Цей компонент виконує роль проміжного шлюзу між Prometheus і Kubernetes. Adapter звертається до Prometheus за допомогою мови запитів PromQL, отримує необхідні значення метрик та перетворює їх у формат Kubernetes Metrics API. Таким чином кастомні метрики стають доступними для компонентів Kubernetes так само, як і стандартні показники використання процесора чи пам'яті. Після перетворення метрики передаються до Kubernetes API Server через інтерфейс Custom Metrics API. API Server виступає центральною точкою взаємодії всіх компонентів Kubernetes і забезпечує доступ до кастомних метрик для механізмів автоматичного масштабування. Далі в роботу вступає Horizontal Pod Autoscaler (HPA). HPA регулярно отримує значення необхідних метрик через Kubernetes Metrics API та порівнює їх із цільовими пороговими значеннями, заданими адміністратором. Наприклад, якщо середній час відповіді перевищує встановлений поріг або кількість запитів на Pod стає занадто великою, HPA приймає рішення про необхідність масштабування.

На основі отриманих даних HPA формує команду для зміни кількості реплік застосунку. Ця команда передається до об'єкта Deployment або ReplicaSet, які відповідають за керування життєвим циклом Pod-ів. У разі зростання навантаження Kubernetes автоматично створює нові Pod-и, а при його зменшенні видаляє надлишкові екземпляри, оптимізуючи використання ресурсів кластера.

Таким чином, весь процес можна описати наступною послідовністю:

1. Мікросервіси, експортери або зовнішні системи генерують метрики.
2. Prometheus збирає та зберігає ці метрики.
3. Prometheus Adapter отримує потрібні показники через PromQL.
4. Adapter перетворює їх у формат Kubernetes Custom Metrics API.
5. Kubernetes API Server надає доступ до цих метрик.
6. HPA аналізує поточні значення та порівнює їх із заданими порогами.
7. Deployment або ReplicaSet збільшує або зменшує кількість Pod-ів.

8. Система автоматично адаптується до поточного навантаження.

Основною перевагою такого підходу є те, що масштабування відбувається не лише за рівнем завантаження процесора чи пам'яті, а за реальними показниками роботи сервісу. Це дозволяє більш точно реагувати на зміну навантаження, підвищувати продуктивність застосунків, зменшувати час відповіді та ефективніше використовувати ресурси Kubernetes-кластера.

Таким чином, інтеграція Prometheus, Grafana та системи експортерів забезпечує комплексний моніторинг контейнеризованих сервісів у Kubernetes. Використання низькорівневих і високорівневих метрик створює основу для аналізу навантаження, прогнозування поведінки системи та реалізації механізмів автоматичного масштабування ресурсів у хмарних середовищах.

1.6. Обґрунтування необхідності проактивного масштабування ресурсів у розподілених інформаційно-комунікаційних системах

Хмарні технології є однією з основних технологічних платформ для побудови та функціонування сучасних розподілених інформаційно-комунікаційних систем. Їх використання забезпечує еластичність, масштабованість, високу доступність сервісів та можливість динамічного розподілу обчислювальних ресурсів відповідно до поточних потреб системи. Водночас поширення контейнеризації та мікросервісної архітектури істотно ускладнює процеси управління ресурсами, оскільки інтенсивність запитів, інформаційних потоків і навантаження на окремі компоненти РІКС може суттєво змінюватися в часі. Навантаження в РІКС характеризується високою динамічністю та може містити трендові, сезонні, циклічні й випадкові складові. Зміна інтенсивності запитів і потоків даних безпосередньо впливає на використання процесорних ресурсів, оперативної пам'яті, систем зберігання даних і пропускної здатності мережевих з'єднань. Різке збільшення навантаження за недостатньої кількості доступних ресурсів може спричинити зростання часу обслуговування запитів, накопичення черг, зниження продуктивності та порушення встановлених вимог до якості обслугову-

вання. Водночас резервування ресурсів відповідно до максимального очікуваного навантаження призводить до їх неповного використання в періоди зниження активності системи. Для адаптації ресурсного забезпечення до зміни навантаження застосовуються механізми автоматичного масштабування. Значна частина традиційних механізмів реалізує реактивний принцип, за якого рішення щодо зміни кількості або обсягу ресурсів приймається після фіксації відповідної зміни контрольованих метрик. Зокрема, перевищення встановленого порогового значення використання процесорних ресурсів, оперативної пам'яті або іншого показника може ініціювати горизонтальне чи вертикальне масштабування відповідного сервісу. Основним обмеженням реактивного підходу є наявність часової затримки між виникненням зміни навантаження та фактичним наданням необхідних ресурсів. Цей інтервал включає час, необхідний для збору й опрацювання метрик, виявлення умови масштабування, формування керуючого рішення, створення та ініціалізації додаткових екземплярів сервісу, а також їх включення в процес обслуговування запитів. У разі стрімкого зростання навантаження система протягом цього інтервалу може функціонувати в умовах дефіциту ресурсів, що призводить до збільшення затримок та зниження якості обслуговування.

Іншим способом зменшення ризику перевантаження є попереднє резервування додаткових ресурсів. Однак такий підхід призводить до надлишкового ресурсного забезпечення та знижує ефективність використання обчислювальної інфраструктури. Особливо суттєво ця проблема проявляється у великих мікросервісних РІКС, де одночасно функціонує значна кількість контейнеризованих сервісів. Принципи реактивного та проактивного масштабування ресурсів РІКС наведено на рис. 1.17.

Додаткова складність управління ресурсами пов'язана з асинхронним характером взаємодії компонентів сучасних РІКС. Використання брокерів повідомлень, черг завдань та подієво-орієнтованої архітектури може призводити до нерівномірного розподілу навантаження між окремими сервісами. У таких умовах зміна потреби в ресурсах не завжди безпосередньо відображається лише показниками використання процесора або оперативної пам'яті. Для прийняття обґрунтованих рішень щодо масштабування доцільно додатково враховувати сервісні метрики, зокрема

інтенсивність надходження запитів, довжину черг, кількість активних завдань та час їх виконання.



Рис. 1.17. Принцип роботи реактивного та проактивного масштабування

Крім того, навантаження РІКС часто характеризується часовими закономірностями, які можуть бути виявлені на основі накопичених моніторингових даних. Реактивні механізми не використовують інформацію про майбутню зміну стану системи, оскільки керуючі рішення формуються переважно на основі поточних або нещодавно отриманих значень метрик. Це зумовлює доцільність застосування проактивного масштабування, за якого потреба в ресурсах визначається з урахуванням прогнозованого навантаження.

Основна ідея проактивного масштабування полягає у завчасному формуванні керуючого рішення щодо зміни ресурсного забезпечення до моменту виникнення потенційного дефіциту ресурсів. Для цього використовуються історичні значення метрик, представлені у вигляді часових рядів, на основі яких здійснюється прогнозування майбутнього стану системи. Отриманий прогноз може використовуватися для визначення необхідної кількості екземплярів сервісу або обсягу ресурсів, які повинні бути доступними в певний момент часу.

У разі прогнозованого збільшення навантаження проактивний механізм може завчасно ініціювати створення додаткових екземплярів кон-

тейнеризованого сервісу або збільшення обсягу виділених йому ресурсів. Це дає змогу компенсувати часові затримки, пов'язані з процесом масштабування, та підготувати систему до обслуговування очікуваного навантаження. У разі прогнозованого зниження навантаження кількість активних екземплярів або обсяг зарезервованих ресурсів може бути зменшено, що сприяє підвищенню ефективності використання обчислювальної інфраструктури. Для прогнозування навантаження можуть використовуватися статистичні методи аналізу часових рядів та методи машинного навчання. До поширених підходів належать моделі ARIMA та Holt–Winters, а також рекурентні нейронні мережі LSTM і GRU [31-33]. Статистичні моделі дають змогу враховувати трендові та сезонні компоненти часових рядів, тоді як рекурентні нейронні мережі забезпечують моделювання складніших нелінійних часових залежностей. Вибір конкретного методу прогнозування повинен визначатися характером навантаження, горизонтом прогнозування, необхідною точністю та обчислювальною складністю моделі.

Для реалізації проактивного масштабування контейнеризованих РІКС можуть використовуватися платформи оркестрації, зокрема Kubernetes. Інтеграція Kubernetes із засобами моніторингу, зовнішніми джерелами метрик і спеціалізованими контролерами дає змогу включати результати прогнозування в контур управління ресурсами. У такому випадку прогнозована величина навантаження використовується для формування рішення щодо необхідної кількості екземплярів сервісу або обсягу ресурсів, після чого відповідне рішення реалізується засобами оркестрації. Перспективна архітектура системи проактивного масштабування ресурсів РІКС зображена на рис.1.18.

Застосування проактивного масштабування дає змогу підвищити ефективність управління ресурсами завдяки узгодженню їх доступного обсягу з прогнозованими потребами сервісів. Це сприяє зменшенню ймовірності виникнення дефіциту ресурсів у періоди пікового навантаження та одночасно обмежує надлишкове резервування в періоди низької активності. У результаті створюються передумови для підвищення продуктивності, забезпечення встановлених показників якості обслуговування та раціональнішого використання обчислювальної інфраструктури.

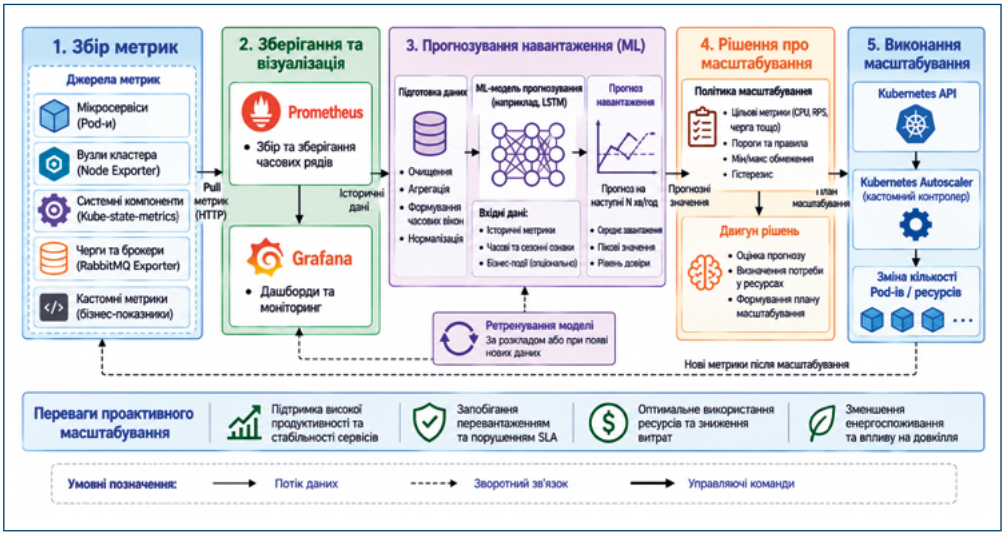


Рис. 1.18. Перспективна архітектура системи проактивного масштабування ресурсів РІКС

Отже, перехід від суто реактивного до прогнозно-орієнтованого управління ресурсами є важливим напрямом розвитку механізмів масштабування РІКС [35-38]. Поєднання моніторингу, аналізу часових рядів, прогнозування навантаження та автоматизованого прийняття рішень створює основу для побудови адаптивних методів управління ресурсами, здатних враховувати не лише поточний, а й прогнозований стан розподіленої системи.

РОЗДІЛ ІІ

СИСТЕМАТИЗАЦІЯ МОДЕЛЕЙ І МЕТОДІВ ПОБУДОВИ МІКРОСЕРВІСНИХ РОЗПОДІЛЕНИХ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ СИСТЕМ

2.1. Особливості та обмеження архітектурних підходів до побудови розподілених інформаційно-комунікаційних систем

Розвиток РІКС супроводжується трансформацією архітектурних підходів до побудови програмних сервісів та організації їх взаємодії. Вибір архітектури безпосередньо впливає на масштабованість системи, ефективність використання обчислювальних ресурсів, надійність, гнучкість розгортання та можливості адаптації до зміни навантаження. У цьому контексті доцільним є порівняльний аналіз монолітної та мікросервісної архітектур як поширених підходів до побудови програмної складової РІКС.

Монолітна архітектура передбачає об'єднання основних функціональних компонентів застосунку, зокрема логіки оброблення даних, користувацького інтерфейсу та механізмів доступу до даних, у межах єдиної програмної системи. Її компоненти тісно пов'язані між собою та, як правило, розгортаються і масштабуються як єдине ціле [40]. Узагальнену структуру вебсервісу з монолітною архітектурою наведено на рис. 2.1.

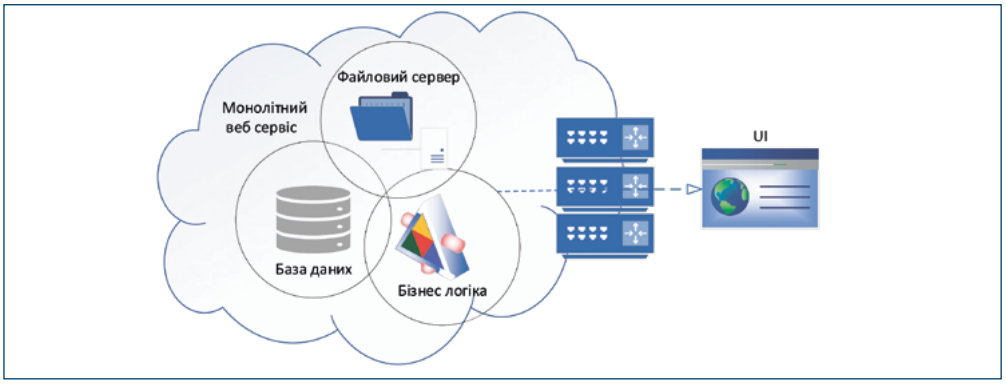


Рис. 2.1. Приклад побудови вебсервісу із монолітною архітектурою

Щоб внести зміни до такого типу програми, потрібно оновити весь стек, отримавши доступ до кодової бази, а також створити та розгорнути

оновлену версію інтерфейсу на стороні сервісу. Такий підхід забезпечує простоту реалізації, цілісність виконання та зручність централізованого управління, що задовольняє вимоги функціонування для маломасштабних або локальних систем.

До переваг монолітної архітектури можна віднести:

- ✓ Просте розгортання – один виконуваний файл або каталог спрощує розгортання.
- ✓ Розробка – коли додаток створено з однією кодовою базою, його легше розробляти.
- ✓ Продуктивність – у централізованій базі коду та репозиторії один API часто може виконувати ту саму функцію, яку численні API виконують з мікросервісами.
- ✓ Спрощене тестування – оскільки монолітний додаток є єдиним централізованим блоком, наскрізне тестування можна виконувати швидше, ніж у розподіленому додатку.
- ✓ Легке налагодження – оскільки весь код знаходиться в одному місці, легше виконати запит і знайти проблему.

Монолітні додатки можуть бути досить ефективними, доки вони не стають занадто великими, і масштабування не стає проблемою. Зі зростанням складності розподілених середовищ, підвищення вимог до масштабованості, надійності й безперервності обслуговування, обмеження монолітної архітектури стали дедалі очевиднішими.

До недоліків моноліту можна віднести:

- Повільніша швидкість розробки – велика монолітна програма робить розробку складнішою та повільнішою.
- Масштабованість – не можливо масштабувати окремі компоненти.
- Надійність – якщо в будь-якому модулі виникла помилка, це може вплинути на доступність усієї програми.
- Перешкода для впровадження технологій – будь-які зміни у програмному файлі або мові впливають на весь застосунок, часто роблячи зміни дорогими та трудомісткими.
- Відсутність гнучкості – моноліт обмежений технологіями, які вже використовуються в моноліті.
- Розгортання – невелика зміна в монолітному застосунку вимагає повторного розгортання всього моноліту.

З метою подолання обмежень монолітного підходу та підвищення гнучкості побудови розподілених систем широкого застосування набула мікросервісна архітектура, що передбачає декомпозицію програмного застосунку на сукупність функціонально відокремлених і відносно незалежних сервісів [42]. Кожний мікросервіс реалізує визначену функціональність, взаємодіє з іншими компонентами через стандартизовані програмні інтерфейси та може незалежно розгортатися, оновлюватися і масштабуватися. Така організація забезпечує можливість адаптивного виділення ресурсів для окремих компонентів відповідно до зміни їх навантаження, що є особливо важливим для функціонування РІКС у динамічних умовах. Узагальнену структуру вебсервісу з мікросервісною архітектурою наведено на рис. 2.2.

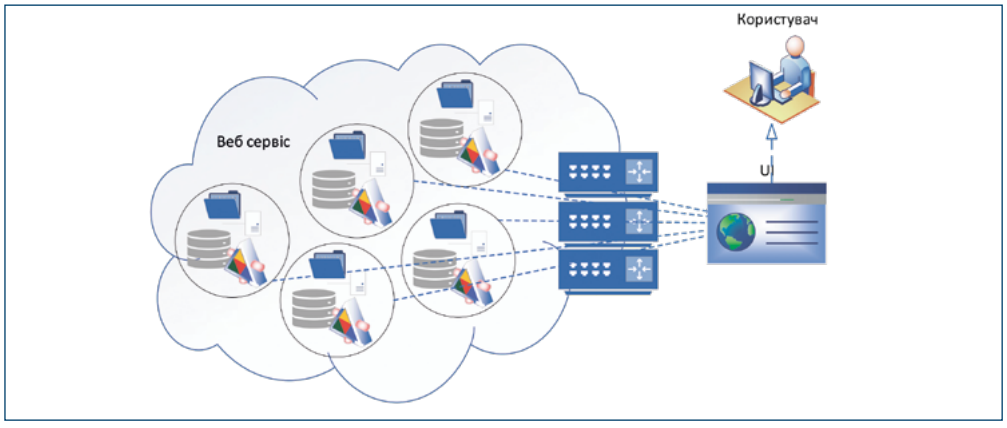


Рис. 2.2. Приклад побудови вебсервісу із мікросервісною архітектурою

Застосунки на основі мікросервісів розділяють програмне забезпечення на компоненти, які можна замінювати та оновлювати незалежно один від одного. Вони взаємодіють через мережу, зазвичай використовуючи універсальні протоколи, наприклад REST API, які не залежать від конкретних технологій. Це забезпечує їх слабке зв'язування, дозволяючи замінювати чи перебудовувати окремий компонент без впливу на решту застосунку.

До недоліків мікросервісів можна віднести:

- Складність процесу розгортання – мікросервіси додають більше складності порівняно з монолітною архітектурою, оскільки існує більше сервісів у більшій кількості місць, створених кількома командами. Якщо процес розгортання і, відповідно оновлення,

не керується належним чином, це призводить до уповільнення швидкості роботи та низької операційної продуктивності.

- Експоненціальні витрати на інфраструктуру – кожен новий мікросервіс може мати власну вартість набору тестів, схем розгортання, інфраструктури хостингу, інструментів моніторингу тощо.
- Додаткові організаційні витрати – командам потрібно додати ще один рівень комунікації та співпраці для координації оновлень та інтерфейсів.
- Проблеми узгодженості – кожен мікросервіс має власний набір журналів, що впливає на загальну працездатність і узгодженість системи. Крім того, один компонент сервісу може виконуватися на кількох машинах, що ще більше ускладнює налагодження.
- Відсутність стандартизації – без спільної платформи управління може спостерігатися нагромадження мікросервісів, написаних різними мовами програмування, стандартів ведення лог-журналів та моніторингу, що призведе до порушення їхнього функціонування.
- Відсутність чіткого розподілу відповідальності – зі впровадженням більшої кількості сервісів зростає кількість команд, які їх використовують. З часом стає важко зрозуміти, які доступні сервіси команда може використовувати та до кого звернутися за підтримкою.

На відміну від монолітних рішень, у яких усі компоненти інтегровані в єдиний модуль, архітектура системи із мікросервісами (MSA) базується на поділі системи на незалежні, самодостатні рівні. Ефективність мікросервісної архітектури визначається низкою ключових принципів:

- Декомпозиція – поділ великої системи на дрібні, функціонально визначені сервіси. Це підвищує керованість проєкту, оскільки команди розробників можуть працювати з окремими модулями, не охоплюючи всю систему загалом.
- Автономність – кожен сервіс розглядається як незалежна одиниця з власним життєвим циклом, управлінням, API, логікою та базою даних. Це дає змогу розгортати й оновлювати сервіси без впливу на інші частини системи.
- Слабке зв'язування – взаємодія сервісів реалізується через стандартизовані API (синхронні чи асинхронні), що зменшує взаємозалежність і підвищує надійність у разі збоїв.

- Агрегація API – використання чітко визначених інтерфейсів для комунікації між сервісами. Це дає можливість поєднувати модулі, створені на різних мовах програмування, у єдину систему без технологічних обмежень.
- Децентралізоване управління – кожен сервіс самостійно керує власними ресурсами, що дозволяє масштабувати їх вибірково, відповідно до навантажень.
- Гнучкість – можливість швидкої адаптації сервісів до нових вимог і змін ринку, що підвищує життєздатність проєкту.
- Стійкість до збоїв – система здатна мінімізувати вплив відмови одного сервісу на інші завдяки механізмам повторних спроб, автоматичного відновлення чи аварійного перемикання.
- Масштабованість – сервіси можна масштабувати незалежно один від одного за допомогою горизонтального масштабування, балансування навантаження чи кешування, що дозволяє системі динамічно реагувати на змінювані умови експлуатації.
- Контроль версій і висока доступність – сервіси повинні бути доступними впродовж максимально можливого часу, а процеси їх оновлення та розгортання мають бути безперервними. Контроль версій API забезпечує керованість змін і сумісність різних компонентів системи.

Таким чином, MSA формує гнучке й надійне середовище для побудови масштабованих програмних рішень, здатних ефективно працювати в умовах постійних змін і високих вимог до продуктивності.

На рис. 2.3 наведений узагальнений вигляд архітектури системи, в якій функціонує мікросервісний додаток.

В її основі функціонують наступні програмні компоненти, які і забезпечують роботу такої системи:

- Виявлення компонентів сервісів (Service Discovery).

Додатки на базі мікросервісів зазвичай працюють у віртуалізованому чи контейнеризованому середовищі. Кількість екземплярів (компонентів) сервісів та їхня локація динамічно змінюються. Відтак системі необхідно розуміти, як знайти ці компоненти, та як вони називаються, аби направляти правильні запити до цільового мікросервіса. Саме тут стає корисним підхід Service Discovery.

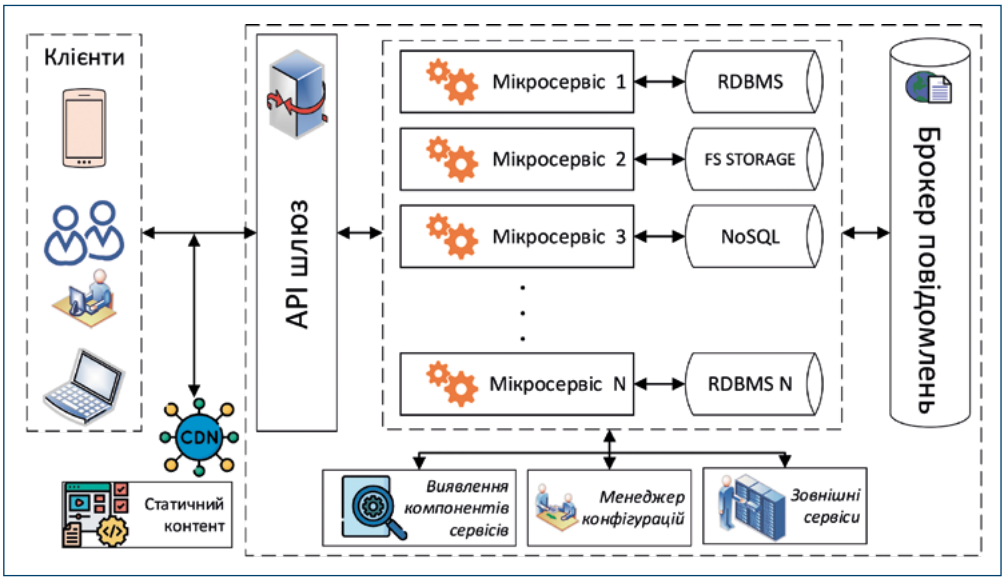


Рис. 2.3. Узагальнений вигляд архітектури системи, яка реалізована за допомогою мікросервісів

Йдеться про механізм автоматичного виявлення та реєстрації доступних в мережі сервісів. Service Discovery працює як реєстр для відстеження адрес всіх компонентів, аби надати змогу сервісам знаходити одне одного без необхідності жорсткої прив'язки до конкретних IP-адрес чи портів. Це робить масштабування та управління мікросервісами більш гнучкими та зручними. Зазвичай для реалізації Service Discovery застосовуються спеціалізовані інструменти, такі як Consul, Etcd, ZooKeeper, або ж вбудовані можливості комплексних хмарних платформ.

- Балансувальник навантаження (Load Balancer) та/або брокер повідомлень (Message Broker).

Однією з найважливіших причин появи та розвитку мікросервісної архітектури стала потреба в горизонтальному масштабуванні ПЗ. Сучасні додатки створюють декілька екземплярів сервісу, аби впоратися з величезним трафіком запитів. Але тут ключове значення має ефективний розподіл запитів між екземплярами. Тому балансувальник навантаження (Load Balancer) чи брокер повідомлень (Message Broker) – це компонент, що розподіляє вхідний мережевий трафік між декількома екземплярами чи компонентами складного сервісу, аби забезпечити оптимальну завантаженість й забезпечити швидкодію та відмовостій-

кість системи. Load Balancer може працювати на рівні додатку (наприклад, HTTP-запитів), або на рівні мережі, наприклад, TCP-з'єднань.

- Шлюз (API Gateway).

API Gateway у мікросервісній архітектурі – це серверний компонент, який виступає єдиною точкою входу для клієнтів, забезпечуючи управління, моніторинг, аутентифікацію, авторизацію, маршрутизацію і перетворення запитів до різних сервісів. Саме цей шлюз дає змогу об'єднати різні мікросервіси в єдине ціле для зовнішніх клієнтів, приховуючи складність внутрішньої структури системи. Відтак він має величезне значення для побудови якісного UX та безпеки мікросервісного середовища. API Gateway також може виконувати додаткові функції, такі як кешування, обмеження швидкості, перетворення даних і забезпечення безпеки. Це спрощує розробку клієнтських застосунків, оскільки клієнти взаємодіють тільки з шлюзом, тоді як внутрішні мікросервіси можна змінити або масштабувати без впливу на зовнішніх клієнтів. Можливості API Gateway надають такі інструменти як NGINX, MuleSoft Anypoint Platform, Kong, Apigee, Spring Cloud Gateway інструменти веб-сервісів Amazon тощо.

- Моніторинг сервісів (Service Monitoring).

Аби управління сервісом було ефективним, його показники мають бути наочними та доступними для сприйняття. Саме це завдання вирішує компонент моніторингу мікросервісів – Service Monitoring. Вони покликані забезпечити процес збору, аналізу та візуалізації даних про роботу мікросервісів. По суті, це свого роду частина системи управління та моніторингу показників роботи мікросервісами. Моніторинг має критичне значення для забезпечення продуктивності, доступності, надійності та безпеки системи. Він дає змогу оперативно виявляти проблеми, відстежувати ключові метрики, вживати заходів для поліпшення роботи системи та ухвалювати якісні стратегічні рішення, засновані на даних. Для реалізації Service Monitoring у мікросервісній архітектурі використовують різні інструменти, такі як Prometheus, Grafana, ELK Stack, Datadog, New Relic тощо. Такі платформи дають змогу відстежувати метрики продуктивності, журнали подій, трасування запитів та інші дані, необхідні для забезпечення ефективної роботи мікросервісів у контейнеризованих та розподілених середовищах.

Врешті роботу усіх сервісів необхідно скоординувати та інтегрувати в єдиний процес для втілення певної бізнес-логіки. Саме для цього у мікросервісній архітектурі передбачені механізми оркестратори (Service Orchestration). Вони охоплюють управління послідовністю викликів мікросервісів, передачу даних між ними, обробку помилок і забезпечення цілісності даних тощо. В рамках Service Orchestration певний центральний сервіс надсилає команди іншим мікросервісам, забезпечує їхню координацію та контролює результати взаємодії, аби вони відповідали бізнес-логіці. Оркестування дозволяє додатку бути цілісним та ефективним, попри розподілену внутрішню логіку. Воно спрощує залежності між мікросервісами, усуває конфлікти, дозволяє вдосконалити бізнес-логіку додатку як таку.

Це лише базові компоненти мікросервісної архітектури, які так чи інакше характерні для усіх систем, що працюють з такого роду сервісами. На практиці кожен компонент можна додатково деталізувати, визначаючи окремі елементи. У контексті функціонування в мережевому середовищі MSA зазвичай структурується у кілька рівнів (рис.2.4), кожен з яких виконує специфічні функції:

- Рівень представлення (Presentation Layer).

Даний рівень забезпечує взаємодію користувача з системою через інтерфейси (веб, мобільні застосунки, API-шлюзи). Відповідає за авторизацію, автентифікацію, маршрутизацію запитів і адаптацію даних. Часто реалізується через API Gateway, який виконує роль єдиної точки входу до мікросервісів.

- Рівень бізнес-логіки (Business Logic Layer).

Цей рівень складається безпосередньо з мікросервісів, кожен з яких виконує конкретні бізнес-функції (обробка транзакцій, аналітика, обробка потокових даних тощо). Мікросервіси спілкуються між собою через стандартизовані протоколи, переважно REST, gRPC, GraphQL або асинхронні системи повідомлень (Kafka, RabbitMQ, NATS). Для забезпечення стійкості застосовуються шаблони service discovery, балансування навантаження, а також circuit breakers для запобігання каскадним відмовам.

- Рівень даних (Data Layer).

По суті рівень даних є найскладнішим, оскільки забезпечує роботу з базами даних, які можуть бути розподіленими або спеціалізованими під конкретний мікросервіс (підхід database-per-service). Використовуються

як SQL, так і NoSQL системи (PostgreSQL, MongoDB, Cassandra, Redis). Він повинен гарантувати узгодженість і цілісність даних у розподіленому середовищі, часто із застосуванням event sourcing або CQRS (Command Query Responsibility Segregation – це архітектурний шаблон, який розділяє операції запису даних (команди) від операцій читання даних (запити), дозволяючи оптимізувати кожну з них окремо для підвищення продуктивності, масштабованості та безпеки програми).

- Рівень інфраструктури та комунікацій (Infrastructure Layer).

Забезпечує оркестрацію контейнерів (Docker, Kubernetes), управління сервісами, моніторинг і автоматичне масштабування. Даний рівень включає інструменти Service Mesh (Istio, Linkerd), що забезпечують безпечну маршрутизацію, балансування навантаження, трасування запитів і політики доступу. У мережевому контексті тут особливо важливі SDN (Software-Defined Networking) та NFV (Network Function Virtualization), які дозволяють гнучко управляти маршрутизацією трафіку та використанням ресурсів на рівні Edge/Fog вузлів.

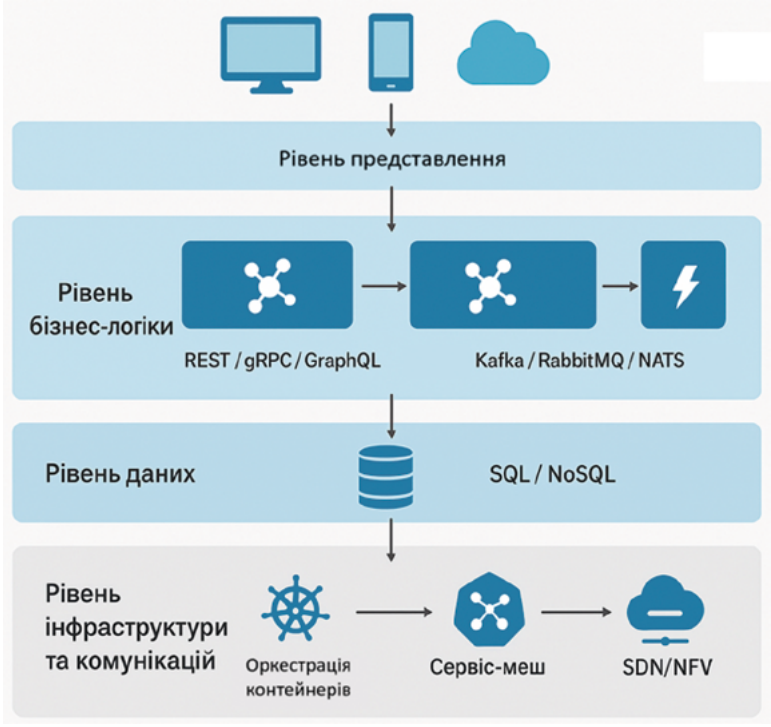


Рис. 2.4. Схема мережевої організації мікросервісної інфраструктури

Архітектуру мікросервісів можна розділити на внутрішню та зовнішню архітектуру.

Внутрішня мікросервісна архітектура (Internal Microservices Architecture) – це підхід до побудови програмних систем, у якому застосунок структурується як сукупність малих, незалежних, слабко зв'язаних сервісів, які взаємодіють між собою в межах системи. Вона володіє певними своїми характеристиками та складовими:

1. Незалежні сервіси. Кожен мікросервіс розробляється з метою виконання конкретної функції. Сервіси можуть бути автономно розроблені, розгорнуті і масштабовані, незалежно від інших компонентів системи.
2. Слабке зв'язування (Loose Coupling). Взаємодія між сервісами відбувається через чітко визначені інтерфейси (API) або через черги повідомлень. Це мінімізує прямі залежності та дозволяє варіювати технології, що застосовуються в різних сервісах.
3. Обмежені контексти (Bounded Contexts). Кожен сервіс відповідає за певну працездатність загального сервісу у межах чітко окресленого контексту. Сервіс володіє своїми даними та логікою.
4. API-шлюз (API Gateway). Використовується як точка входу для зовнішніх або клієнтських запитів, направляючи їх до відповідних внутрішніх мікросервісів.
5. Сервісне виявлення (Service Discovery). Механізми, які дозволяють сервісам динамічно ідентифікувати місце розташування інших сервісів і встановлювати з ними зв'язок без необхідності жорсткого конфігурування.
6. Контейнеризація. Мікросервіси часто пакуються і розгортаються в контейнерах (наприклад, Docker), що забезпечує портативність, ізоляцію середовища виконання та ефективніше використання інфраструктурних ресурсів.
7. Оркестрація. Застосовуються засоби автоматичного керування життєвим циклом мікросервісів – розгортання, масштабування, оновлення, моніторинг тощо.
8. Децентралізоване управління даними. Кожен сервіс оперує власним сховищем даних. Це сприяє автономності сервісів і зменшує конкуренцію за загальні ресурси, а також дозволяє

уникнути єдиного централізованого вузла у питаннях зберігання даних.

Немає встановленого стандарту для внутрішньої архітектури, оскільки кожен сервіс чи застосунок мають свої власні характеристики. Ця частина є найскладнішою у проектуванні MSA.

Зовнішня архітектура (Outer Architecture) – це набір інфраструктурних компонентів, що забезпечують узгоджену взаємодію мікросервісів із зовнішніми клієнтами та між собою (рис. 2.5). Її основною метою є створення уніфікованого середовища управління, комунікації та автоматизації, що забезпечує стійкість і масштабованість системи.

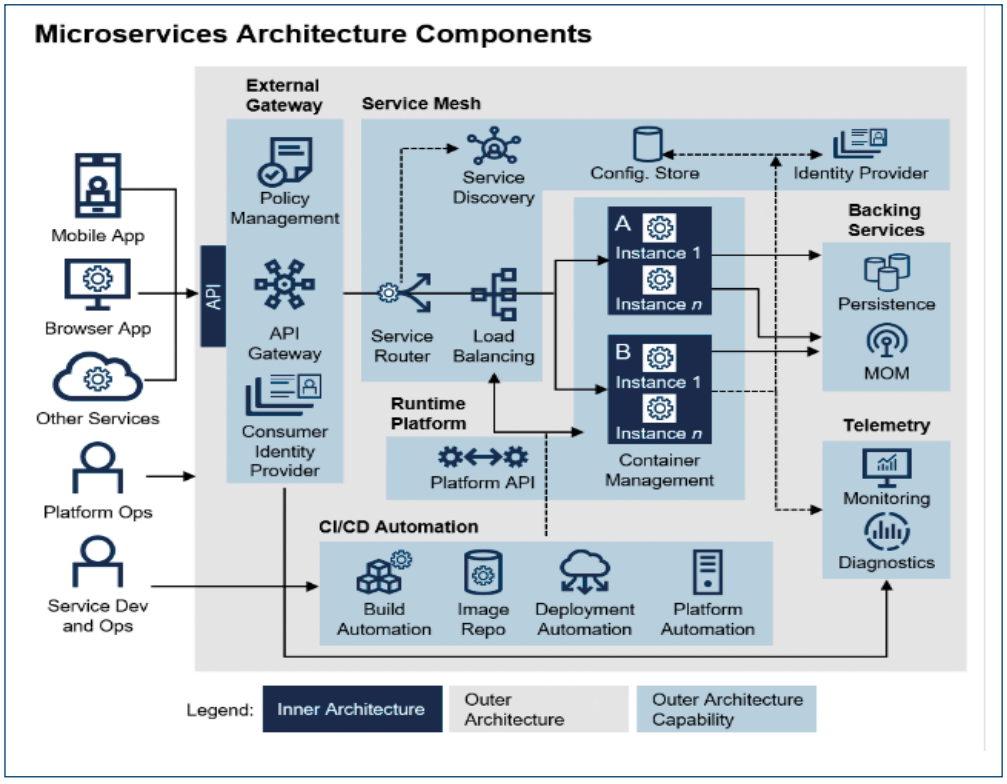


Рис. 2.5. Принцип взаємодії мікросервісів із зовнішніми клієнтами та між собою (зовнішня архітектура) [42]

Науковці у своїх працях [43-47] виокремлюють шість ключових складових зовнішньої архітектури:

1. **Зовнішній шлюз (External Gateway)**. Зовнішній шлюз виступає точкою входу для всіх зовнішніх клієнтів. Він виконує функції мар-

шрутизації запитів до відповідних мікросервісів; автентифікації та авторизації; контролю доступу і застосування політик безпеки; обмеження швидкості (rate limiting) та запобігання перевантаженню.



Рис. 2.6. Приклад роботи External Gateway

Таким чином, шлюз слугує брандмауером та одночасно балансувальником навантаження для вхідного трафіку.

2. **Сервісна сітка (Service Mesh – SM)** – це інфраструктурний рівень, що управляє взаємодією між мікросервісами. Вона забезпечує безпечну та надійну комунікації між сервісами, завдяки можливості налаштування різних методів доступу до середовища обміну даними; автоматичне виявлення місця розгортання сервісів та маршрутизація запитів до них; контроль за процесом передавання трафіку (формування транспортного потоку, circuit breaking, retry-логіка); спостережуваність (observability) через збір метрик та трасування.
3. **Управління контейнерами (Container Management).** Даний компонент відповідає за життєвий цикл контейнерів, у яких розгортаються мікросервіси. У поєднанні із SM це по суті, система оркестрації мікросервісами, оскільки основним завданням цього рівня архітектури є автоматичне розгортання, масштабування та міграція контейнерів; ізоляція середовищ виконання; ефективне використання обчислювальних ресурсів та забезпечення відмовостійкості. У сучасних системах, найчастіше цю функцію виконує Kubernetes або аналогічні платформи оркестрації.

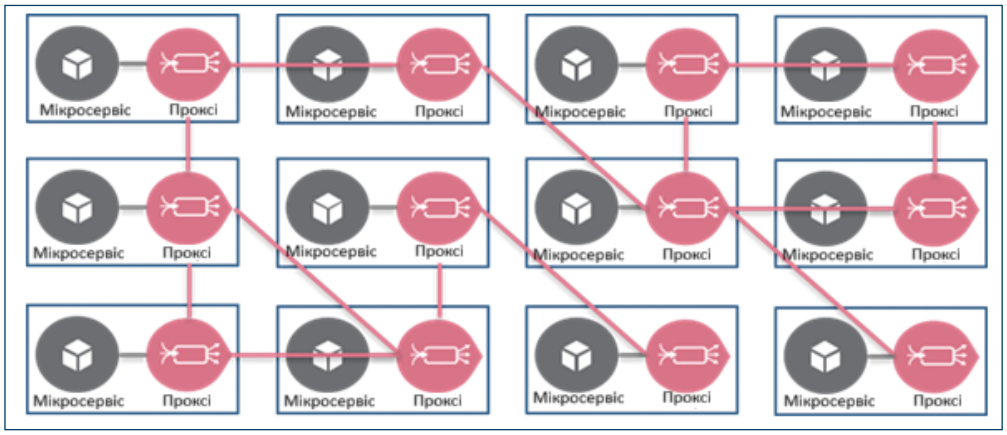


Рис. 2.7. Приклад функціонування Service Mesh [42]

4. **Допоміжні сервіси або сервіси підтримки роботи мікросервісів (Backing Services).** Цей рівень архітектури є допоміжним. По суті, це сервіси, які можна використовувати через мережу для роботи всього мікросервісу. Це комплексне поняття, яке стосується підключених ресурсів, що взаємодіють із елементарним сервісом, таких як бази даних, системи кешування та служби SMTP. Зазвичай комунікація між цими сервісами є асинхронною. Про типи повідомлень, які використовуються для роботи і розгортання мікросервісних застосувань буде описано нижче.
5. **Телеметрія (Telemetry)**. Саме цей рівень охоплює механізми збору, обробки та аналізу даних про роботу системи. У MSA значна кількість мікросервісів працює в розподіленому середовищі, тому індивідуальний моніторинг стану кожного сервісу та реагування на проблеми є надзвичайно складним та трудомістким. Телеметрія контролює служби та налаштовує середовище для реагування на проблеми, що виникають для кожної служби, здійснює моніторинг стану сервісів та логування. Завдяки телеметрії досягається спостережуваність (observability) системи, що дозволяє своєчасно виявляти аномалії та оптимізувати роботу.
6. **Автоматизація CI/CD (CI/CD Automation)** забезпечує безперервну інтеграцію і доставку нових версій мікросервісів. Автоматизація безперервної інтеграції, безперервної доставки та

безперервного розгортання є критично важливим компонентом будь-якої системи MSA. Автоматизація цього процесу включає: автоматичне тестування, побудову контейнерів; контроль версій і т.п. Завдяки CI/CD забезпечується оперативне оновлення сервісів без порушення доступності та працездатності системи.

Таким чином, зовнішня архітектура виступає керуючим і підтримувальним шаром мікросервісної системи, який інтегрує ключові інструменти для безпеки, масштабованості та автоматизації. Вона забезпечує надійність роботи внутрішніх мікросервісів та прозору взаємодію з клієнтами, знижуючи складність процесом управління системою в цілому.

2.2. Обмін повідомленнями та моніторинг ресурсів у мікросервісних розподілених інформаційно-комунікаційних систем

Обмін повідомленнями у мікросервісних системах є ключовим елементом забезпечення взаємодії між окремими сервісами в розподіленому середовищі. У контексті мережної інфраструктури мікросервісів цей процес охоплює як синхронні, так і асинхронні методи комунікації, що дозволяють підтримувати гнучкість, масштабованість та стійкість системи до збоїв [48].

Синхронний обмін повідомленнями (Synchronous Messaging) передбачає, що відправник повідомлення очікує на відповідь від отримувача перед тим, як продовжити обробку. У мікросервісних мережах поширені такі підходи до реалізації синхронного обміну повідомленнями:

- REST (Representational State Transfer) – архітектурний стиль, який використовує протокол HTTP для взаємодії між сервісами через запити (GET, POST, PUT, DELETE). REST забезпечує просту інтеграцію та сумісність між сервісами, проте може бути обмеженим щодо продуктивності при великих обсягах даних.
- gRPC (Google Remote Procedure Call) – високопродуктивний фреймворк від Google, що використовує HTTP/2 для передачі двійкових даних та підтримує строгі контракти через gRPC IDL (Interface Definition Language). gRPC дозволяє реалізувати

ефективну комунікацію між сервісами з низькою затримкою та підтримкою потокових даних.

- GraphQL – мова запитів для API, яка дозволяє клієнту визначати точну структуру запитуваних даних. Це зменшує обсяг передаваних даних і оптимізує продуктивність при складних взаємодіях між мікросервісами.
- Thrift – фреймворк, що забезпечує кросмовну взаємодію сервісів через Thrift IDL, підтримуючи різні протоколи передачі та формати серіалізації, що робить його придатним для гетерогенних середовищ.

Використання такого способу комунікації, у стилі запит-відповідь, більше підходить для інтерактивних сервісів та клієнтів, які потребують обробки в режимі реального часу, наприклад таких як зовнішні API.

Асинхронний обмін повідомленнями (Asynchronous Messaging) дозволяє сервісу відправляти повідомлення незалежно від отримання відповіді, що значно підвищує масштабованість та стійкість системи до затримок або відмов окремих компонентів. Для таких сценаріїв мікросервіси можуть використовувати протоколи обміну повідомленнями, такі як NATS, Kafka, AMQP, STOMP та MQTT.

Kafka – розподілений брокер повідомлень, який забезпечує стійке та надійне зберігання потоків подій із можливістю обробки в реальному часі.

NATS – легкий, високопродуктивний брокер повідомлень, оптимізований для простих сценаріїв публікації/підписки та мікросервісних комунікацій.

AMQP (Advanced Message Queuing Protocol) – протокол для організації черг повідомлень, що гарантує надійну доставку та підтримку складних маршрутизаторів і обробки черг.

На відміну від більшості традиційних протоколів обміну повідомленнями, Kafka та NATS пропонують невимушену, але розподілену інфраструктуру обміну повідомленнями, яка більше підходить для створення мікросервісів.

STOMP (Simple/Streaming Text Oriented Messaging Protocol) – текстово орієнтований протокол для обміну повідомленнями, простий у впровадженні та інтеграції з різними брокерами.

MQTT (Message Queuing Telemetry Transport) – легковаговий протокол для публікації/підписки, що широко застосовується у IoT та розподілених системах з обмеженими ресурсами.

Організація обміну повідомленнями у мережах мікросервісної архітектури поєднує синхронні та асинхронні методи взаємодії, підкріплені чіткими контрактами сервісів (Service Contracts). Вони визначають інтерфейси сервісів та структуру повідомлень. У системах, де працюють з мікросервісами найчастіше використовують:

- OpenAPI/Swagger – стандартизований опис REST API, що дозволяє автоматично генерувати документацію, клієнтські SDK та забезпечує чітке визначення кінцевих точок та форматів даних.
- gRPC IDL – опис інтерфейсів у gRPC, що дозволяє строго визначити методи, параметри та типи даних між сервісами.
- GraphQL Schema – декларативний опис типів даних і запитів у GraphQL, що забезпечує узгодженість та контроль над структурами відповіді.
- Thrift IDL – мова опису сервісів у Thrift, що підтримує різні мови програмування та формати серіалізації для крос-платформної взаємодії. Це забезпечує високу гнучкість, ефективність передачі даних та надійність роботи розподіленої системи в цілому.

Їх використання забезпечує високу гнучкість, ефективність передачі даних та надійність роботи розподіленої системи в цілому.

У сучасних гетерогенних обчислювальних та мережевих середовищах мікросервісні додатки функціонують одночасно на Edge-вузлах, у проміжних обчислювальних вузлах (Fog) та в централізованих хмарних ЦОД. Така багаторівнева архітектура забезпечує можливість обробки даних у близькості до їхнього джерела, зменшуючи затримки та підвищуючи ефективність використання ресурсів. Водночас вона накладає значні вимоги до управління ресурсами та параметрами мережі, адже обчислювальні потужності вузлів, пропускна здатність каналів, затримки та надійність мережі постійно змінюються залежно від навантаження та поведінки користувачів.

Мікросервісні системи характеризуються високою динамічністю та модульністю: кожен сервіс відповідає за окрему функцію та може бути масштабований, переміщений або перезапущений незалежно від ін-

ших. Такі властивості, разом із непередбачуваним характером трафіку та запитами користувачів на ті чи інші його складові, роблять традиційні статичні методи адміністрування ресурсів і налаштування мережі недостатньо ефективними. Відсутність гнучкого моніторингу та адаптивного управління може призвести до деградації продуктивності сервісів, збільшення затримок обробки запитів та зростання ризику відмов окремих компонентів системи. У зв'язку з цим особливої уваги потребує розробка комплексних систем моніторингу та управління параметрами мереж, спеціально орієнтованих на мікросервісні додатки. Їх інтеграція у мережі дозволить у режимі реального часу збирати дані про стан обчислювальних вузлів, пропускну здатність каналів, затримки та рівень завантаження контейнерів. Крім того, з іншого боку, завдяки використанню програмного підходу до конфігурації таких систем (SDN у поєднанні з NFV чи ін.) є можливість виявляти аномалії, прогнозувати потенційні відмови та коригувати параметри системи проактивно. Крім того, інтеграція алгоритмів адаптивного балансування та оркестрації дозволяє ефективно розподіляти сервіси між різнорівневими середовищами (Edge–Fog–Cloud), оптимізуючи використання обчислювальних ресурсів, мінімізуючи затримки та підвищуючи стабільність роботи системи навіть за умов динамічних змін навантаження.

Важливим елементом таких систем є прогнозування поведінки мережі та навантаження на сервіси, що забезпечує можливість динамічного визначення пріоритетів ресурсів і своєчасного масштабування сервісів. Це особливо критично для застосунків із високими вимогами до якості обслуговування та безперервності обробки даних, таких як відеоаналітика, IoT-системи, сервіси реального часу, що функціонують лише при розгортанні мереж мобільного зв'язку 5-го покоління.

Таким чином, розробка систем моніторингу та управління параметрами мереж із мікросервісними додатками є не лише необхідною умовою підтримки стабільної роботи сучасних сервісів, а й ключовим фактором для забезпечення високої продуктивності, надійності та адаптивності у динамічних, масштабованих та багаторівневих обчислювальних середовищах. Вона створює основу для інтелектуальної оркестрації, підвищення ефективності використання ресурсів та забезпечення безперервності обслуговування користувачів у реальному часі.

2.3. Концептуальні моделі та методи управління мікросервісами у розподілених інформаційно-комунікаційних системах

Розвиток мікросервісної архітектури та її застосування у розподілених інформаційно-комунікаційних системах зумовили формування різних концептуальних моделей і підходів до організації розгортання, функціонування та управління мікросервісами. Їх особливості визначаються архітектурою обчислювального середовища, способом розподілу мережевих і обчислювальних ресурсів, характером міжсервісної взаємодії та вимогами до якості надання сервісів. У сучасних РІКС такі підходи реалізуються із використанням технологій віртуалізації, контейнеризації, оркестрації, програмно-конфігурованих мереж, периферійних і хмарних обчислень.

Важливим напрямом розвитку таких систем є інтеграція мікросервісної архітектури з інфраструктурою мобільних мереж нового покоління, у якій необхідно забезпечувати динамічне розміщення сервісів, розподіл ресурсів та підтримання заданих показників якості обслуговування. У цьому контексті доцільно розглянути концептуальні підходи, що ґрунтуються на мережевому слайсингу, технологіях SDN/NFV та периферійних обчисленнях і створюють основу для гнучкого управління мікросервісами в умовах змінного навантаження та неоднорідності ресурсів.

П’яте та шосте покоління мобільних мереж (5–6G) підтримує широкий спектр складних сервісів, що висувають суворі та різноманітні вимоги до кінцевої затримки, пропускної здатності, надійності та інших показників продуктивності. Для забезпечення гнучкого розгортання, масштабування та ефективного функціонування сервісів у мобільних мережах п’ятого покоління застосовується концепція мережевого слайсингу (network slicing), яка передбачає формування на основі спільної фізичної інфраструктури логічно ізольованих мережевих зрізів (network slices) з визначеними характеристиками та вимогами до ресурсів. Кожний мережевий зріз може охоплювати ресурси мережі радіодоступу (Radio Access Network, RAN), ядра мережі (Core Network, CN) та транспортної мережі, забезпечуючи їх узгоджене використання відповідно до вимог конкретного сервісу. (Рис. 2.8). Ця концепція стає можливою зав-

дяки гнучкості та програмованості, яку надають технології SDN (Software Defined Networking) та NFV (Network Function Virtualization). Незважаючи на значні переваги мережевого слайсингу, сервіси з високими вимогами до часу відгуку, такі як безпілотне управління транспортними засобами, відеоаналітика та потокові застосунки, потребують інтеграції периферійних обчислень з множинним доступом (multi-access edge computing (MEC)) у 5G-мережі та хмарну інфраструктуру.

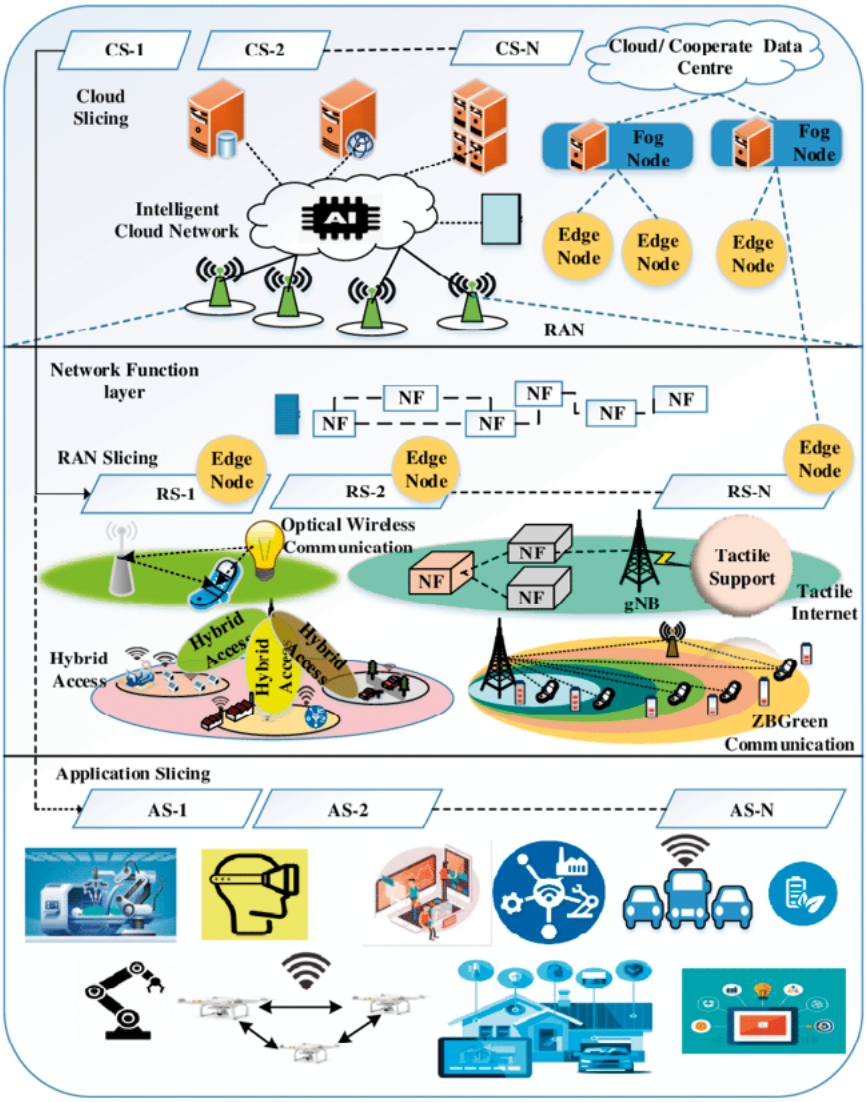


Рис. 2 8. Архітектура мережі мобільного зв'язку на основі віртуалізованої мережі-слайсингу [49]

MEC дозволяє перенести обчислювальні ресурси ближче до користувачів та IoT-сенсорів, де генеруються дані, що суттєво зменшує затримку у порівнянні з класичною mobile cloud computing (MCC). Однак MEC обмежений обсягами ресурсів і може легко перевантажуватися, тому доцільне використання багаторівневої обчислювальної та мережевої інфраструктури, де критичні сервіси обробляються на MEC, а менш чутливі до затримок – у віддаленій хмарі.

Мікросервісні застосунки у такому середовищі потребують оркестрації ресурсів на рівні окремих функцій, оскільки продуктивність застосунку залежить від сумісного використання обчислювальних, мережевих та інших ресурсів різними мікросервісами (resource coupling). У такій багаторівневій структурі виникає ряд проблем: визначення, на яких вузлах або рівнях розгортати конкретні функції, та ефективна організація їхньої комунікації з урахуванням цих взаємозв'язків. Рішення цих завдань критично впливає на кінцеві показники застосунку, такі як загальна затримка, точність, ефективність використання ресурсів і вартість обслуговування.

Розгортання та оптимізація застосунків у такій складній інфраструктурі є надзвичайно складним завданням. Крім того, стан різних ресурсів (наприклад, доступних обчислювальних і мережевих) є високодинамічним через значну варіативність в обчислювальному середовищі (багатокористувацькість, гетерогенність тощо), а також у мережі (перешкоди 5G NR, перевантаження каналів, втрата пакетів тощо). Тому завдання розгортання та оптимізації застосунків у багаторівневій обчислювальній і мережевій інфраструктурі стає ще складнішим, якщо враховувати динамічний стан ресурсів у реальному часі.

Одним із варіантів вирішення цієї проблеми є автоматизоване управління наданням мікросервісних застосунків у динамічному, гетерогенному, багаторівневому обчислювальному середовищі 5G-мережі з урахуванням взаємозв'язків між ресурсами, коли відомі лише вимоги на рівні застосунку, але відсутні детальні вимоги на рівні окремих функцій [50]. Їх принцип роботи ґрунтується на інтеграції моніторингу доступних ресурсів, прогнозування продуктивності та оптимального розподілу обчислювальних і мережевих ресурсів для задоволення кінцевих вимог додатка.

У більшості випадків, фізична інфраструктура представляється обчислювальними вузлами, розподіленими по кількох рівнях, на яких можливе розділення ресурсів (compute slicing) для обслуговування різних додатків. Кожен вузол характеризується набором доступних ресурсів і рівнем, до якого він належить, наприклад, пристрій IoT, віддалений Edge, локальний Edge або центральна хмара. Вузли забезпечують обчислювальні та мережеві ресурси, які можуть динамічно призначатися різним функціям або мікросервісам додатка. Додаток моделюється як граф мікросервісів або функцій з визначеними між ними зв'язками, що відображають залежності потоків даних. Для кожного додатка визначаються кінцеві вимоги до продуктивності, такі як затримка та пропускна здатність. На рівні функцій кожна з них має власні часткові показники продуктивності, а також можливі обмеження щодо рівня інфраструктури, на якому вона повинна виконуватися (наприклад, локальне виконання на пристрої користувача).

Ключовим аспектом таких систем управління є врахування взаємозв'язку ресурсів, що виділяються для різних функцій (Рис. 2.9). Для кожної пари ресурсів визначається мінімальна кількість, яку слід призначити функції, щоб досягти заданого показника продуктивності додатка, враховуючи ресурси, вже виділені іншим функціям. Основна мета полягає у мінімізації сумарного використання обчислювальних і мережевих ресурсів при забезпеченні вимог до продуктивності мікросервісів, таких як затримка, пропускна здатність та QoS.

Цей підхід дозволяє врахувати, як використання обчислювальних ресурсів впливає на ефективність мережевих ресурсів, і навпаки – як зміни у виділених ресурсах однієї функції впливають на інші функції.

Серед існуючих платформ для оркестрації контейнеризованих мікросервісів Kubernetes виділяється як найбільш універсальне і широко впроваджене рішення. Kubernetes надає механізми автоматичного масштабування, балансування навантаження, відновлення збоїв, управління конфігураціями та оновленнями без простою сервісів. Крім того, його архітектура дозволяє інтегрувати власні програмні модулі планування та моніторингу, що робить платформу придатною для динамічних Edge-Cloud середовищ із мінливими умовами навантаження та обмеженими ресурсами. Однак стандартна стратегія оркестрації Kubernetes має

обмеження, зокрема, вона не враховує реальне використання ресурсів під час планування розміщення контейнерів, що може призводити до зниження продуктивності через конкуренцію за ресурси. Для подолання цих обмежень у роботі Angelo Marchese та Orazio Tomarchio [51] пропонують розширення Kubernetes за допомогою стратегії оркестрації, орієнтованої на динамічне навантаження. Її основою є використання агентів моніторингу для збору телеметричних даних про використання ресурсів (CPU, пам'ять, мережа і т.п.) як на рівні вузлів, так і на рівні контейнерів (Рис. 2.10). Ці дані дають змогу сформувати актуальне уявлення про поточний стан інфраструктури та функціонування мікросервісів. Планувальник Kubernetes доповнено спеціалізованим плагіном, який використовує телеметричні дані про поточне завантаження процесора (CPU), використання оперативної пам'яті, дискових ресурсів і мережевої інфраструктури для прийняття обґрунтованих рішень щодо розміщення та перерозподілу мікросервісів.

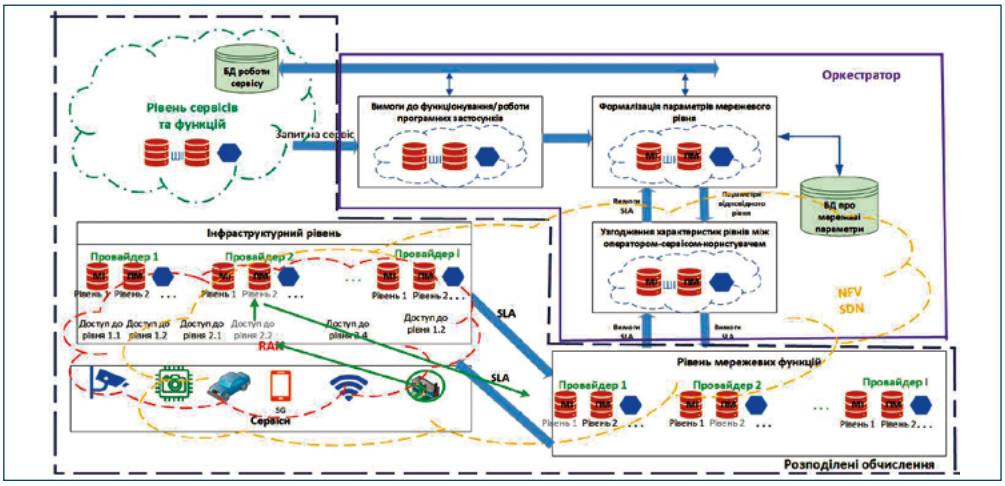


Рис. 2.9. Концептуальна модель управління наданням мікросервісних застосунків у мережах 5-6G

Планувальник функціонує у вигляді функції (абстракції) для керування життєвим циклом кожного компоненту (функції) мікросервісу у площині управління Kubernetes та розширює можливості стандартного планувальника шляхом використання спеціального плагіна LoadAware Scoring. Для кожного компоненту P мікросервісу k, який підлягає розміщенню в контейнері, плагін призначає оцінку кожного кандидата-вузла

у кластері, що пройшов фазу фільтрації, виконуючи функцію оцінювання (scoring function). Згенеровані плагіном оцінки комбінуються з результатами інших плагінів оцінювання стандартного планувальника Kubernetes. На етапі оцінювання викликається функція Score() плагіна LoadAware, яка визначає оцінку для кожного вузла кластера n під час планування розміщення компоненту *P* мікросервісу *k* (Рис. 2.11).

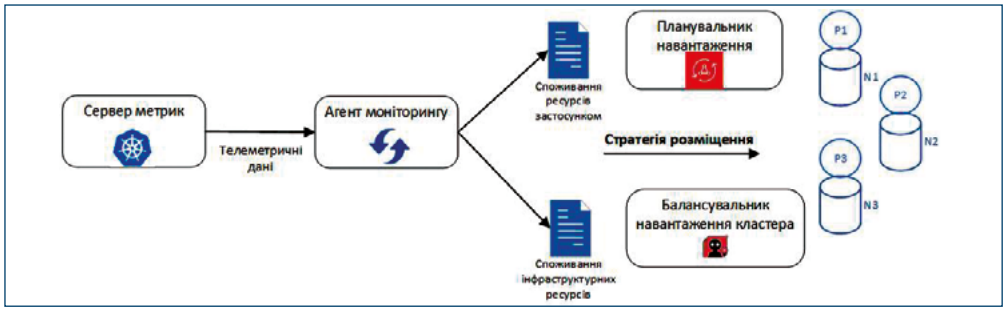


Рис. 2.10. Архітектура запропонованого у [51] плагіна для удосконалення роботи планувальника Kubernetes

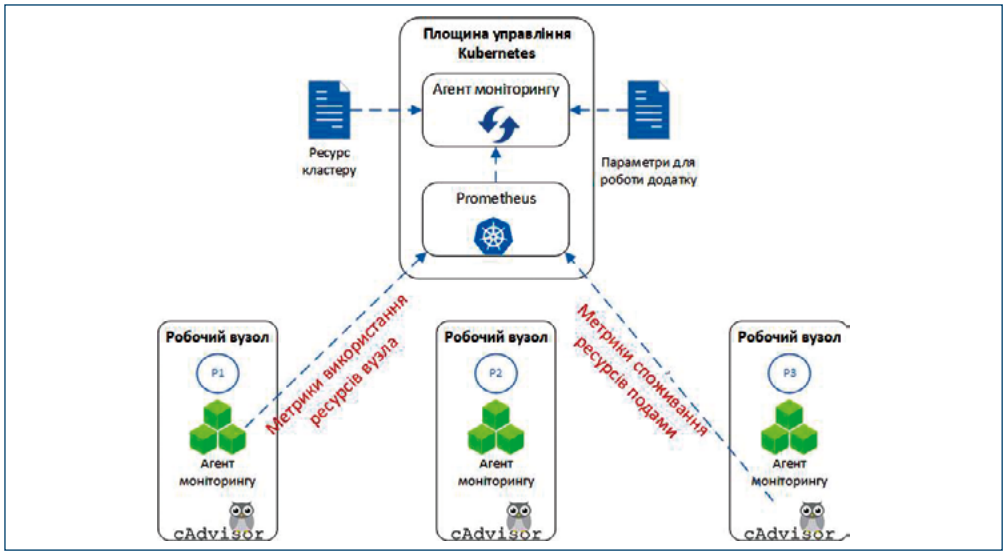


Рис. 2.11. Система управління (моніторингу) фреймворком

Оцінка вузла n для розміщення компоненту *P* мікросервісу *k* визначається за рівнянням (1).

$$scope(p,n_i)=(1-std(cpu_{r_n}, mem_{r_n}, net_{r_n}, disk_{r_n}))*MaxNodeScore, \quad (2.1)$$

де *MaxNodeScore* – максимальна кількість вузлів для розгортання;

cpu_{r_n} – ступінь завантаження процесорних ресурсів вузла після можливого розміщення нового компоненту P. Він оцінюється:

$$cpu_{r_n} = \frac{cpu_p + cpu_n}{\sum cpu_n}, \qquad (2.2)$$

де – прогнозоване робоче використання CPU, необхідне для роботи компоненту P (визначене за допомогою відповідної функції у Deployment); cpu_p – фактичне сумарне використання CPU на вузлі n; $\sum cpu_n$ – загальна кількість доступних (allocatable) CPU на вузлі n. mem_{r_n} – ступінь використання пам'яті вузла після можливого розміщення нового компоненту P. Він оцінюється:

$$mem_{r_n} = \frac{mem_p + mem_n}{\sum mem_n}, \qquad (2.3)$$

де mem_p – прогнозована робоча RAM, необхідна для роботи компоненту P (визначене за допомогою відповідної функції у Deployment); mem_n – фактичне сумарне використання RAM на вузлі n; $\sum mem_n$ – загальний обсяг доступної пам'яті на вузлі n. net_{r_n} – ступінь використання пропускної здатності каналу після можливого розміщення нового компоненту P. Він оцінюється:

$$net_{r_n} = \frac{net_p + net_n}{\sum net_n}, \qquad (2.4)$$

де net_p – прогнозоване використання пропускної здатності каналу, необхідного для передавання компоненту P (визначене за допомогою відповідної функції у Deployment); net_n – фактичне сумарне використання пропускної здатності каналу, необхідного для передавання компоненту P від вузла n; $\sum net_n$ – загальний обсяг пропускної здатності каналу, яка доступна вузлу n. $disk_{r_n}$ – ступінь використання пропускної здатності дискової підсистеми (I/O throughput) після можливого розміщення нового компоненту P. Визначають її аналогічно до попередніх параметрів.

Ключова ідея даної стратегії планування полягає у мінімізації взаємного впливу між компонентами мікросервісу, які спільно використовують ресурси, що дозволяє знизити кількість порушень вимог QoS, пов'язаних із затримкою відповіді застосунків. Водночас підвищується рівень використання ресурсів кластера завдяки можливості розміщення більшої кількості застосунків на тих самих вузлах. Для досягнення цього

компоненти P, які конкурують за один і той самий тип ресурсу, розподіляються по різних вузлах, де це можливо.

На відміну від стандартних плагінів планувальника Kubernetes, що призначають вузлам оцінки на основі статичних вимог до ресурсів (задекларованих розробниками застосунків), плагін LoadAware використовує телеметрію поточного використання ресурсів у реальному часі. Це забезпечує можливість динамічного коригування рішень щодо розміщення залежно від актуальних моделей споживання ресурсів компонентами P, які можуть змінюватися з часом.

Ще однією особливістю такого підходу до управління процесом розгортання мікросервісами є Дескеджер, орієнтований на навантаження (Load-Aware Descheduler). Це додатковий компонент, який періодично оцінює ефективність розміщення контейнерів та ініціює їх переміщення між вузлами, якщо це може покращити використання ресурсів та знизити час відгуку. Для оцінки ефективності переміщення контейнера використовується функція оцінки *dscore(p)*, яка визначається як:

$$dscore(p) = \max_{n_j \in c_{n_s}} \left(score\left(p, n_i\right) \right) - score\left(p, n_i\right), \qquad (2.5)$$

де p – контейнер, який використовується для розгортання одного компоненту (функції) мікросервісу, c_{n_s} – множина всіх вузлів у кластері, n_j – поточний вузол, на якому розміщено контейнер p, $score(p, n_i)$ – оцінка придатності вузла n для розміщення компоненту P мікросервісу k, яка враховує реальне використання ресурсів.

Контейнер p підлягає переміщенню, якщо:

$$dscore(p) > 0 \qquad (2.6)$$

Це означає, що переміщення контейнера на інший вузол з вищою оцінкою дозволить зменшити конкуренцію за ресурси та покращити продуктивність.

Комбіноване застосування динамічного планування та адаптивного дескеджування забезпечує ефективніше використання ресурсів кластеру, зменшення часу відгуку мікросервісів і покращення масштабованості системи при зростанні кількості розгорнутих контейнерів та навантаження на інфраструктуру. Однак для такого підходу потрібен додатковий моніторинг і складні алгоритми для правильного прогнозування

навантажень, високі вимоги до інфраструктури для забезпечення безперебійної роботи в реальному часі.

Автори роботи [52] розв'язують схожу проблему як задачу графової кластеризації, де вузли і ребра графа позначають відповідно споживання ресурсів і афінності між сервісами. Використовується нечітка логіка для покращення процесу класифікації і оптимізації розміщення мікросервісів в середовищах Kubernetes.

Пропонується комбінований підхід, який поєднує fuzzy-кластеризацію мікросервісів із евристичним пакуванням (Рис. 2.12). Fuzzy-кластеризація дозволяє кожному мікросервісу належати до кількох кластерів одночасно, з певною ймовірністю p_{ik}, де i – мікросервіс, а k – кластер. Такий підхід формує ймовірнісну матрицю членства $p=[p_{ik}]$, яка визначає ймовірність належності мікросервісу до конкретного кластеру:

$$\sum_{k=1}^{K} p_{ik}, \forall i \in V, \tag{2.7}$$

де V – множина всіх мікросервісів, а K – число кластерів. Членство визначається з урахуванням афінності між мікросервісами, яка оцінюється за двома метриками: Requests Per Second (RPS), що показує інтенсивність запитів від одного сервісу до іншого $Ns_{i\to s_j}$ протягом інтервалу часу x, та Weighted Bidirectional Affinity (WBA), яка враховує як кількість повідомлень, так і обсяг даних між сервісами:

$$RPS_{s_i \to s_j} = \frac{\sum_{t=1}^{x} N_{s_i \to s_j}(t)}{x}, \tag{2.8}$$

$$WBA_{a,b} = w\frac{m_{a,b}}{m} + (1-w)\frac{d_{a,b}}{d}, \tag{2.9}$$

де $m_{a,b}$ та $d_{a,b}$ – кількість повідомлень та обсяг даних між сервісами a і b, m – загальна кількість обмінюваних повідомлень, а d – загальний обсяг обміну даними у байтах, а w – вага ($w \in R$, $0 \le w \le 1$), що відображає значущість кожного компонента метрики спорідненості (тобто розміру даних та кількості повідомлень). Завдяки fuzzy-підходу сервіси, що інтенсивно взаємодіють, можуть бути розподілені по кількох вузлах, що забезпечує більш гнучке та масштабоване розміщення.

Після формування fuzzy-кластерів застосовується евристичне пакування для оптимального розподілу мікросервісів по вузлах кластеру. Ця стадія враховує ресурси вузлів (CPU, RAM) та міжвузловий трафік (egress

traffic). Загальна вартість розміщення кластера формується як сума витрат на обчислювальні ресурси та мережевий трафік:

$$Cost=\sum_{i=1}^{n}(CostCPU_i+CostRAM_i)+\sum i,j_{\in N}t_e\,(i\rightarrow j)cost_{egress}, \quad (2.10)$$

де $CostCPU_i$ та $CostRAM_i$ визначаються як добуток виділених ресурсів на їх вартість, відповідно до тарифного плану провайдера обчислювальних ресурсів, $t_e(i\rightarrow j)$ – обсяг міжвузлового трафіку між сервісами i та j, а n – кількість вузлів. Функція $t_e(i\rightarrow j)$ дорівнює нулю для сервісів на одному вузлі і приймає значення трафіку, якщо сервіси знаходяться на різних вузлах.

Цей підхід дозволяє одночасно зменшити число активних вузлів, оптимізувати час відгуку та знизити витрати на інфраструктуру. Серйозним перевагою є також масштабованість та адаптивність: fuzzy-кластеризація забезпечує можливість реплікації мікросервісів на різних вузлах, а інтеграція з Horizontal Pod Autoscaler (HPA) дозволяє автоматично підлаштовувати кількість реплік під змінне навантаження, підтримуючи стабільну продуктивність.

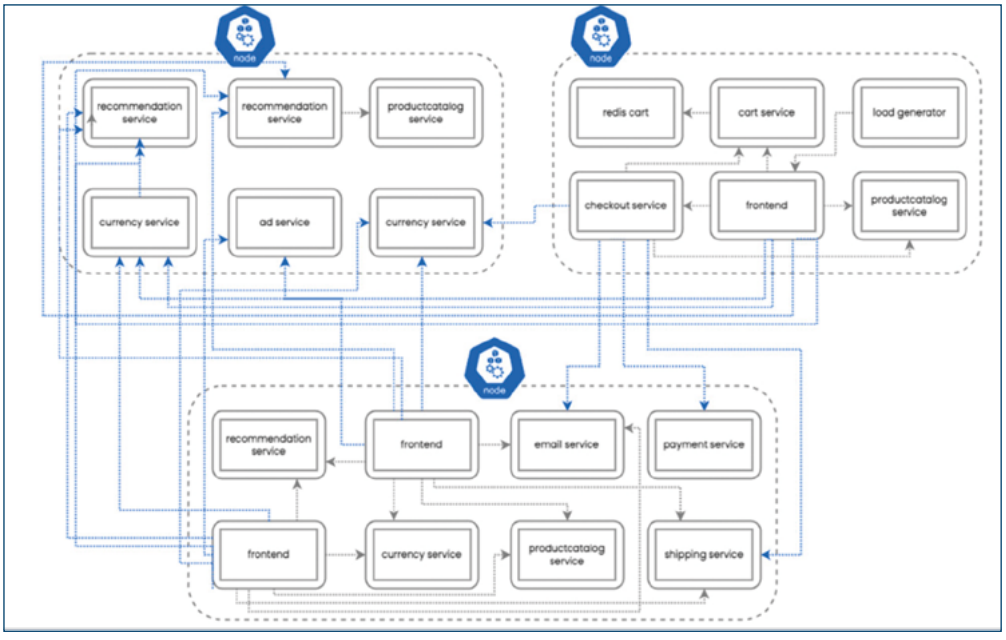


Рис. 2.12. Розміщення мікросервісів за нечітким підходом із увімкненим HPA [52]

Експериментальна перевірка показала, що комбінація fuzzy-кластеризації та евристичного пакування забезпечує оптимальне використ-

тання ресурсів, високу масштабованість та мінімізацію затримок у мікросервісних системах. Проте, в динамічно змінних умовах не завжди можна точно визначити нечіткі множини для розміщення мікросервісів, що може призвести до неефективного використання ресурсів. Потрібно постійно коригувати параметри нечіткої логіки.

Вирішенням цих проблем стала поява проміжних рівнів обчислень – Fog та Edge, які дозволяють переносити частину обробки ближче до джерел даних. Така багаторівнева архітектура (Cloud–Fog–Edge) забезпечує більш рівномірний розподіл навантаження між центром і периферією, знижує латентність доступу, зменшує трафік до хмари та підвищує надійність функціонування систем. Особливо актуальним такий підхід стає в умовах функціонування мікросервісних додатків. Однак ефективне використання цієї архітектури потребує створення механізмів інтелектуального управління обчислювальними процесами, що враховують неоднорідність ресурсів, динамічність мережевих умов і просторово-часовий характер потоків даних, оскільки ускладнюються процеси їх розгортання, моніторингу та оркестрації. У межах Cloud-Fog інфраструктур ця проблема загострюється, оскільки мікросервіси можуть розміщуватись на різних рівнях обчислювальної ієрархії – від віддаленої хмари до периферійних вузлів із обмеженими ресурсами. Це вимагає побудови адаптивних систем управління, здатних у реальному часі приймати рішення щодо міграції, масштабування та взаємодії мікросервісів з урахуванням доступності ресурсів, мережевих затримок, енергоспоживання та поточної завантаженості вузлів.

Традиційні методи управління, орієнтовані переважно на централізовані хмарні середовища, виявляються неефективними в умовах децентралізованої Fog-архітектури, де ключовими чинниками стають автономність, контекстна адаптивність і швидкість реагування на зміни стану системи. Зокрема, виникає необхідність у мультиагентних та інтелектуальних підходах до оркестрації мікросервісів, які забезпечують оптимальне використання ресурсів і підтримують баланс між продуктивністю, енергетичною ефективністю та якістю обслуговування (QoS/QoE).

Сучасні дослідження [53–55] акцентують увагу на необхідності децентралізованих і контекстно-орієнтованих методів управління, здатних адаптувати поведінку мікросервісів до поточних умов мережі.

Такі підходи спираються на використання штучного інтелекту (AI) та машинного навчання (ML) для прогнозування навантаження, оптимізації розміщення контейнерів і вибору вузлів із мінімальною затримкою. Наприклад, застосування підкріплювального навчання (Reinforcement Learning) дає змогу формувати політику розподілу мікросервісів залежно від поточного стану системи, забезпечуючи баланс між продуктивністю та споживанням ресурсів.

Інший напрямок досліджень стосується адаптивної оркестрації на основі QoS/QoE, де метою є забезпечення стабільного рівня якості обслуговування користувачів. У таких підходах, як у [56], враховуються часові обмеження сервісів, типи трафіку та критичність завдань. Розробляються моделі прийняття рішень, які дозволяють динамічно мігрувати мікросервіси між Fog- і Cloud-рівнями відповідно до прогнозованих змін затримок і завантаження мережі.

Значна кількість праць також присвячена енергоефективному управлінню в Cloud-Fog інфраструктурах [57, 58]. У таких системах, особливо в периферійних вузлах із обмеженим живленням, важливо забезпечити оптимальне використання енергоресурсів без втрати продуктивності. З цією метою використовуються гібридні оптимізаційні алгоритми (наприклад, поєднання евристичних методів із навчанням на історичних даних), які визначають оптимальні конфігурації розміщення та параметри масштабування.

Окрему увагу в останні роки приділено проблемі fault tolerance та самовідновлення у мікросервісних системах. Роботи [57, 58] пропонують використання імовірнісних моделей і механізмів адаптивного моніторингу для своєчасного виявлення аномалій та відмов вузлів. Такі підходи інтегруються з оркестраторами і дозволяють здійснювати реактивне або проактивне відновлення сервісів без порушення загальної цілісності системи.

Попри значний прогрес, більшість існуючих рішень залишаються частково централізованими, не враховують динамічну змінність топології Fog-рівня, а також взаємозалежність між сервісами в контексті їх спільного функціонування. Це обмежує можливості масштабування та стійкість систем до збоїв у реальних сценаріях розподілених мереж. Тому актуальним залишається завдання розробки інтелектуальних

децентралізованих моделей управління, що поєднують переваги локальної автономності Fog-вузлів із можливістю глобальної оптимізації на рівні хмарної координації.

У відповідь на ці виклики проведено низку досліджень, спрямованих на створення інтелектуальних систем управління розміщенням мікросервісів, які поєднують обчислення на рівнях Cloud і Fog. Одним із найпомітніших рішень є система MicroFog [59], що пропонує універсальний підхід до організації управління в розподіленому середовищі.

Система MicroFog реалізує концепцію управління (CE)(Рис. 2.13), у межах якої обчислювальні вузли різних Fog-кластерів взаємодіють через гнучкий механізм оркестрації. Основна ідея полягає у створенні модульного, масштабованого середовища, здатного автоматично розміщувати та мігрувати мікросервіси між вузлами з урахуванням доступних ресурсів, топології мережі, затримок і поточного навантаження. На відміну від традиційних хмарних рішень, MicroFog забезпечує підтримку мульти-кластерних і мульти-доменно-розподілених сценаріїв, де управління здійснюється не централізовано, а через узгоджену взаємодію Fog-контролерів.

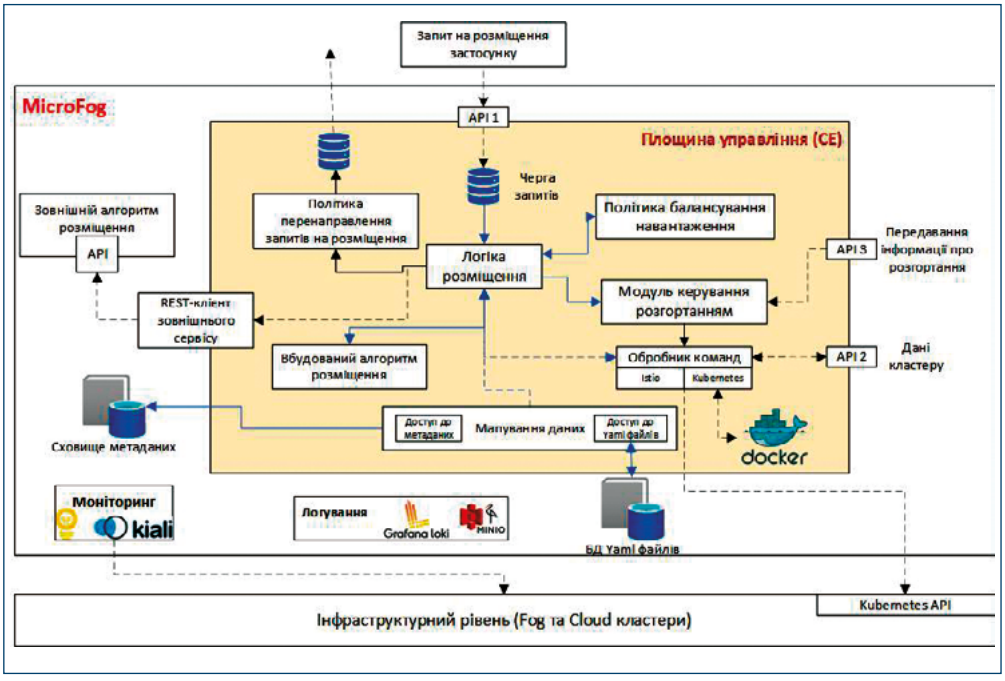


Рис. 2.13. Логіка управління розміщенням мікросервісних IoT додатків

Робота такої системи управління мікросервісними IoT додатками передбачає декілька етапів :

1. Приймання та організація запитів на розміщення.

Постачальники додатків надсилають запит на розміщення застосунку (PR- Placement Requests) до MicroFog-CE через REST API у форматі JSON. Кожен запит містить інформацію про додаток, QoS-вимоги, обмеження на розміщення, точки входу трафіку та інші релевантні параметри. Запити потрапляють у чергу PRQueue, де обробляються залежно від обраного режиму: у secondary режимі всі накопичені PR обробляються одночасно, тоді як у primary режимі кожен PR обробляється негайно після надходження. У разі завантаження потоку обробки запит очікує в черзі до його звільнення.

2. Отримання та підготовка метаданих.

На цьому етапі CE збирає інформацію про наявні ресурси кластерів (CPU, RAM, доступність вузлів) та топологічні дані, включаючи сусідні Fog та Cloud кластери. Модель додатка запитується зі Сховища метаданих і містить інформацію про склад мікросервісів, композитні сервіси, шляхи даних, потоки даних та вимоги до ресурсів. У режимі primary CE формує глобальну картину всієї інфраструктури, отримуючи дані з усіх кластерів. У secondary режимі кожен CE збирає локальні дані та обмінюється PR із суміжними кластерами, забезпечуючи узгоджене розміщення у мультикластерному середовищі.

3. Виконання алгоритму розміщення.
4. Розгортання додатків.

Після визначення розміщення CE формує інформацію для розгортання у двох основних категоріях:

Базова інформація для розгортання: включає відображення Pod на вузол, ingress-кластери для додатків, ресурси Istio Gateway та Virtual Service для маршрутизації трафіку.

Інформація для балансування навантаження: формування підмножин та ваг для Virtual Services і Destination Rules відповідно до обраної політики балансування.

Неповні PR пересилаються до сусідніх кластерів відповідно до політики пересилання, що забезпечує узгоджене завершення розміщення у всіх кластерах.

5. Балансування навантаження.

MicroFog-CE підтримує стандартні методи Istio (round-robin, random, least request, weighted) та дозволяє застосовувати власні політики балансування. Вага кожного екземпляра мікросервісу може динамічно коригуватися на основі результатів розміщення, що гарантує рівномірний розподіл трафіку. У централізованому режимі інформація для балансування формується одночасно для всіх кластерів, у розподіленому – передається паралельно після розміщення всіх відповідних мікросервісів.

6. Моніторинг та логування.

Система інтегрує Prometheus, Grafana, Kiali та Loki для збору метрик, трасування запитів та централізованого логування. Це дозволяє оцінювати продуктивність, контролювати коректність розміщення та створює основу для потенційного застосування алгоритмів на основі машинного навчання для динамічного розміщення мікросервісів.

7. Масштабованість та гнучкість.

CE реалізовано як мікросервіс на Quarkus, що дозволяє легко масштабувати його у Kubernetes і Istio-середовищах. Кожен CE конфігурується автономно та взаємодіє з іншими через REST API. Розгортання додатків здійснюється за допомогою YAML-файлів з MinIO Object Store та Docker-образів із Docker Registry. Таке поєднання забезпечує гнучке і автоматизоване управління розміщенням та балансуванням навантаження у мультикластерних Fog-Cloud середовищах.

Такий підхід дає змогу реалізувати адаптивне та масштабоване управління сервісами IoT-додатків, забезпечуючи при цьому мінімальні затримки обробки й ефективне використання ресурсів. Архітектура MicroFog поєднує логіку планування розміщення з контролем виконання сервісів на рівні Fog-вузлів, що дає можливість одночасно враховувати обчислювальні та мережеві характеристики. Результати, наведені у [59], свідчать, що використання такої архітектури дозволяє зменшити середній час відгуку та покращити рівномірність розподілу навантаження між кластерами у порівнянні з централізованими моделями оркестрації.

Водночас, навіть попри очевидні переваги, MicroFog виявляє низку обмежень, що відкривають простір для подальших досліджень.

Зокрема, у межах платформи недостатньо розвинені механізми прогнозування поведінки мережі, енергетичної оптимізації та самовідновлення у разі відмов вузлів. Крім того, питання динамічного вибору оптимальної політики міграції мікросервісів з урахуванням зміни затримок, завантаження каналів і варіацій QoS залишається відкритим.

2.4. Систематизація моделей і методів побудови розподілених інформаційно-комунікаційних систем

Розвиток мікросервісної архітектури та її поширення у РІКС зумовили появу значної кількості моделей і методів, орієнтованих на вирішення окремих завдань розміщення, планування ресурсів, масштабування, міграції, балансування навантаження, моніторингу та управління. Відмінності у функціональному призначенні таких рішень, типах урахованих ресурсів, механізмах прийняття рішень і середовищах функціонування ускладнюють їх безпосереднє порівняння. Тому для визначення характерних особливостей відомих рішень та встановлення перспективних напрямів їх подальшого розвитку доцільно здійснити систематизацію моделей і методів побудови мікросервісних РІКС.

Систематизацію пропонується здійснювати за п’ятьма основними ознаками: функціональним призначенням, типом урахованих ресурсів, способом управління, середовищем функціонування та методологічним підходом до прийняття рішень. Такий багатокритеріальний поділ дає змогу охопити як функціональну складову управління мікросервісами, так і особливості ресурсного забезпечення та практичної реалізації відповідних механізмів. Узагальнену систематизацію моделей і методів побудови мікросервісних розподілених інформаційно-комунікаційних систем наведено на рис. 2.14.

За функціональним призначенням моделі та методи доцільно розділити відповідно до основних завдань, які виникають у процесі функціонування мікросервісної системи. До них належать розміщення мікросервісів, планування ресурсів, масштабування, міграція, балансування навантаження, а також моніторинг та управління. Такий поділ відображає

життєвий цикл мікросервісів у розподіленому середовищі, а саме від вибору початкового місця їх розгортання до подальшої адаптації конфігурації системи відповідно до зміни навантаження та доступності ресурсів.

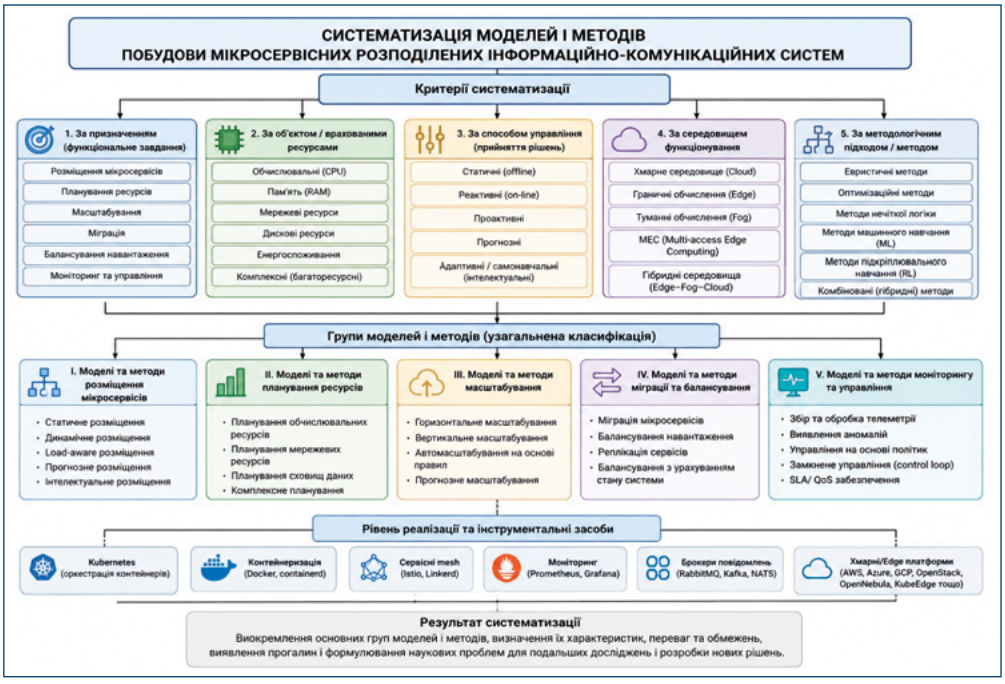


Рис. 2.14. Систематизація моделей і методів побудови мікросервісних РІКС

За типом урахованих ресурсів можна виокремити моделі, орієнтовані на обчислювальні ресурси процесора, оперативну пам'ять, мережеві та дискові ресурси, енергоспоживання, а також комплексні багаторесурсні моделі. Найпростіші рішення враховують один домінуючий параметр, наприклад завантаження CPU або обсяг доступної оперативної пам'яті. Проте для розподілених мікросервісних систем такого представлення часто недостатньо, оскільки продуктивність окремого мікросервісу визначається одночасним впливом декількох видів ресурсів. Зокрема, переміщення компонента на вузол із достатніми обчислювальними ресурсами не гарантує покращення продуктивності, якщо між взаємодіючими мікросервісами виникають значні мережеві затримки або недостатньою є пропускна здатність каналів. Тому перспективним є застосування комплексних моделей, у яких рішення приймаються на основі сукупності обчислювальних, мережевих та інших параметрів.

За способом управління та прийняття рішень можна виділити статичні, реактивні, проактивні, прогнозні та адаптивні (інтелектуальні) моделі й методи. Статичні методи ґрунтуються на попередньо визначених правилах і параметрах та практично не враховують поточних змін стану середовища. Їхньою перевагою є відносна простота реалізації, однак ефективність таких рішень зменшується в умовах динамічної зміни навантаження.

Реактивні методи використовують актуальні показники стану системи та формують керуючі впливи після виявлення зміни контрольованих параметрів або перевищення встановлених порогових значень. До цієї групи можна віднести механізми автоматичного масштабування, які збільшують або зменшують кількість екземплярів мікросервісу залежно від поточного завантаження ресурсів. Основним обмеженням реактивного управління є наявність часової затримки між виникненням зміни стану системи, її виявленням та виконанням відповідної керуючої дії.

Проактивні та прогнозні методи спрямовані на випереджальне прийняття рішень. Вони використовують накопичені телеметричні дані та моделі прогнозування для оцінювання майбутнього навантаження, можливого дефіциту ресурсів або погіршення показників якості обслуговування. Це дає змогу виконувати масштабування, міграцію чи перерозподіл ресурсів до виникнення критичного стану системи.

Подальшим розвитком цього напряму є адаптивні та інтелектуальні методи, які поєднують поточну телеметрію, історичні дані та алгоритми інтелектуального аналізу для динамічного формування керуючих рішень. На відміну від статичних механізмів, такі рішення здатні адаптуватися до зміни характеристик навантаження і стану інфраструктури без необхідності постійного ручного коригування параметрів управління.

Важливою ознакою систематизації є також середовище функціонування мікросервісів. За цією ознакою можна виділити моделі для хмарних середовищ (Cloud), граничних обчислень (Edge), туманних обчислень (Fog), інфраструктури MEC (Multi-access Edge Computing) та комбінованих середовищ Edge–Fog–Cloud. У централізованих хмарних системах основна увага приділяється ефективному використанню значних пулів обчислювальних ресурсів і масштабуванню сервісів. Для Edge- та MEC-середовищ критичними стають затримка передавання даних, об-

меженість локальних ресурсів і близькість мікросервісу до джерела даних або користувача. У гібридних Edge–Fog–Cloud середовищах задача додатково ускладнюється необхідністю вибору не тільки конкретного обчислювального вузла, але й відповідного рівня інфраструктури для виконання окремого мікросервісу.

За методологічним підходом до формування керуючих рішень можна виокремити евристичні, оптимізаційні методи, методи нечіткої логіки, машинного навчання, підкріплювального навчання та комбіновані методи. Евристичні алгоритми забезпечують відносно низьку обчислювальну складність і можуть бути ефективними для оперативного прийняття рішень. Оптимізаційні методи дають змогу формалізувати задачу розподілу ресурсів або розміщення мікросервісів шляхом визначення цільової функції та системи обмежень. Водночас зі збільшенням кількості мікросервісів, вузлів і контрольованих параметрів складність пошуку оптимального рішення істотно зростає.

Методи нечіткої логіки є доцільними за умов невизначеності або неможливості встановлення однозначних меж між станами системи. Методи машинного навчання дають змогу виявляти приховані залежності між характеристиками навантаження, ресурсним станом вузлів і продуктивністю мікросервісів. Методи підкріплювального навчання орієнтовані на формування політики управління на основі взаємодії із середовищем та отримання винагороди за прийняті рішення. Окрему групу становлять комбіновані методи, які інтегрують декілька механізмів прийняття рішень, наприклад прогнозування навантаження засобами машинного навчання з подальшою оптимізацією розміщення мікросервісів.

Відповідно до функціонального призначення розглянуті рішення можна узагальнити у п’ять основних груп. Перша група охоплює моделі та методи розміщення мікросервісів. До неї належать статичне, динамічне, орієнтоване на навантаження (load-aware), прогнозне та інтелектуальне розміщення. Основним завданням таких методів є визначення вузла або рівня інфраструктури, який найбільшою мірою відповідає ресурсним вимогам конкретного мікросервісу.

Друга група представлена моделями та методами планування ресурсів. Вони охоплюють планування обчислювальних і мережевих ресурсів, ресурсів систем зберігання даних та комплексне багаторесурсне плану-

вання. На відміну від методів розміщення, тут основна увага приділяється визначенню необхідного обсягу ресурсів та механізмів їх розподілу між мікросервісами.

До третьої групи належать моделі та методи масштабування. Вони забезпечують адаптацію ресурсного забезпечення мікросервісів до зміни інтенсивності навантаження. Основними механізмами є горизонтальне і вертикальне масштабування, автоматичне масштабування на основі встановлених правил та прогнозне масштабування. Останнє є особливо перспективним для сервісів із суттєвими коливаннями інтенсивності запитів, оскільки дає змогу завчасно підготувати необхідні ресурси.

Четверта група об'єднує моделі та методи міграції і балансування. Їх застосування спрямоване на перерозподіл мікросервісів та навантаження між доступними вузлами. До цієї групи належать механізми міграції контейнерів, балансування навантаження, реплікації сервісів та балансування з урахуванням поточного стану системи. Такі механізми особливо важливі в динамічних гетерогенних середовищах, де доступність і завантаженість ресурсів можуть істотно змінюватися в часі.

П'ята група охоплює моделі та методи моніторингу й управління. Їх функціонування ґрунтується на зборі та обробленні телеметрії, виявленні аномалій, застосуванні політик управління, формуванні замкненого контуру керування (closed-loop management), а також контролі виконання вимог SLA та QoS. Ця група має особливе значення для реалізації адаптивного управління, оскільки саме телеметричні дані формують інформаційну основу для прийняття рішень щодо масштабування, міграції та перерозподілу ресурсів.

Практична реалізація зазначених моделей і методів здійснюється із застосуванням відповідних програмних та інфраструктурних засобів. Базовим компонентом оркестрації контейнеризованих мікросервісів є Kubernetes, тоді як Docker та containerd забезпечують їх контейнеризацію. Для управління міжсервісною взаємодією можуть використовуватися сервісні сітки Istio та Linkerd, а збір і візуалізація телеметрії можуть здійснюватися за допомогою Prometheus і Grafana. Асинхронна взаємодія мікросервісів підтримується брокерами повідомлень RabbitMQ, Kafka та NATS. Розгортання систем може здійснюватися у хмарних, Edge та гібридних інфраструктурах.

Таким чином, проведена систематизація показує, що моделі й методи побудови мікросервісних РІКС доцільно розглядати не ізольовано, а як взаємопов'язану сукупність механізмів розміщення, планування, масштабування, міграції, балансування, моніторингу та управління. Простежується поступовий перехід від статичних і реактивних механізмів до проактивних, прогнозних та інтелектуальних рішень, здатних враховувати поточний і прогнозований стан гетерогенної інфраструктури.

Водночас збільшення кількості контрольованих параметрів, динамічність навантаження, взаємозалежність обчислювальних і мережевих ресурсів, а також розподілення мікросервісів між Cloud-, Fog-, Edge- та MEC-рівнями суттєво ускладнюють процес прийняття керуючих рішень. Це обумовлює необхідність подальшого розвитку моделей і методів, здатних комплексно враховувати стан ресурсів, прогнозувати зміну навантаження та адаптивно визначати раціональні варіанти розміщення, масштабування і міграції мікросервісів. Виявлені особливості та обмеження відомих рішень формують підґрунтя для розроблення відповідних моделей і методів управління ресурсами РІКС.

РОЗДІЛ ІІІ

МЕТОДИ РЕАКТИВНОГО ТА ПРОАКТИВНОГО АВТОМАСШТАБУВАННЯ РЕСУРСІВ У РОЗПОДІЛЕНИХ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ СИСТЕМАХ

3.1. Методика експериментального дослідження реактивного масштабування ресурсів у Kubernetes-середовищі

Ефективність функціонування розподілених інформаційно-комунікаційних систем значною мірою визначається здатністю їх обчислювальної інфраструктури адаптуватися до динамічних змін навантаження. Для мікросервісних і контейнеризованих середовищ характерними є нерівномірність потоків запитів, періодичне виникнення пікових навантажень та зміна потреб окремих сервісів у процесорних ресурсах і оперативній пам'яті. Використання статично визначеної конфігурації за таких умов може призводити як до дефіциту ресурсів і погіршення продуктивності сервісів, так і до надлишкового резервування та нераціонального використання обчислювальної інфраструктури.

Адаптація ресурсного забезпечення РІКС до таких змін може здійснюватися на основі реактивного та проактивного підходів [60]. Реактивне керування передбачає зміну обсягу доступних ресурсів відповідно до фактично зафіксованого стану системи, тоді як проактивне ґрунтується на прогнозуванні майбутньої динаміки навантаження та завчасному прийнятті відповідних керуючих рішень. Поєднання цих підходів створює передумови для підтримання необхідного рівня продуктивності сервісів за одночасного раціонального використання ресурсів розподіленої інфраструктури.

Дослідження доцільно розпочати з реактивних механізмів, оскільки саме вони забезпечують базову адаптацію ресурсного забезпечення відповідно до поточного стану системи та формують основу для подальшого переходу до прогнозного керування. У контейнеризованих середовищах така адаптація безпосередньо пов'язана зі зміною кількості екземплярів сервісу або обсягу ресурсів, доступних кожному з них. Тому на першому етапі дослідження необхідно оцінити ефективність

відповідних механізмів масштабування та визначити їх вплив на функціонування РІКС за різної інтенсивності навантаження.

Одним із ключових завдань під час побудови РІКС є забезпечення відповідності обсягу виділених обчислювальних ресурсів поточній інтенсивності навантаження. Для контейнеризованих сервісів це завдання може вирішуватися шляхом динамічної зміни кількості екземплярів застосунку або обсягу ресурсів, доступних окремим контейнерам. У Kubernetes ці підходи реалізуються відповідно механізмами горизонтального та вертикального автомасштабування.

Для оцінювання ефективності зазначених механізмів сформовано методику експериментального дослідження, яка передбачає послідовне тестування контейнеризованого застосунку за різних способів керування обчислювальними ресурсами. Основною метою дослідження є визначення впливу горизонтального та вертикального автомасштабування на продуктивність сервісу, використання обчислювальних ресурсів і здатність системи адаптуватися до динамічної зміни інтенсивності вхідного навантаження

Експериментальне дослідження передбачає три основні режими реактивного функціонування системи: без застосування автомасштабування, із використанням горизонтального автомасштабування та із застосуванням вертикального автомасштабування. Для забезпечення коректності порівняння в усіх режимах використовується однакова модель прикладного сервісу та аналогічні сценарії формування навантаження. У процесі експериментів здійснюється моніторинг використання обчислювальних ресурсів і основних показників продуктивності сервісу за різної інтенсивності вхідного навантаження. Це дає змогу оцінити особливості функціонування системи для кожного з досліджуваних режимів та визначити ефективність застосування механізмів автомасштабування. Узагальнену послідовність проведення експериментального дослідження наведено на рис. 3.1.

Початкове тестування виконувалося для застосунку, що функціонував в одному Pod. Генерація навантаження здійснювалася за допомогою інструменту Apache JMeter. Під час експериментів фіксувалися такі показники, як середній час відповіді, кількість оброблених запитів та пропускна здатність сервісу.

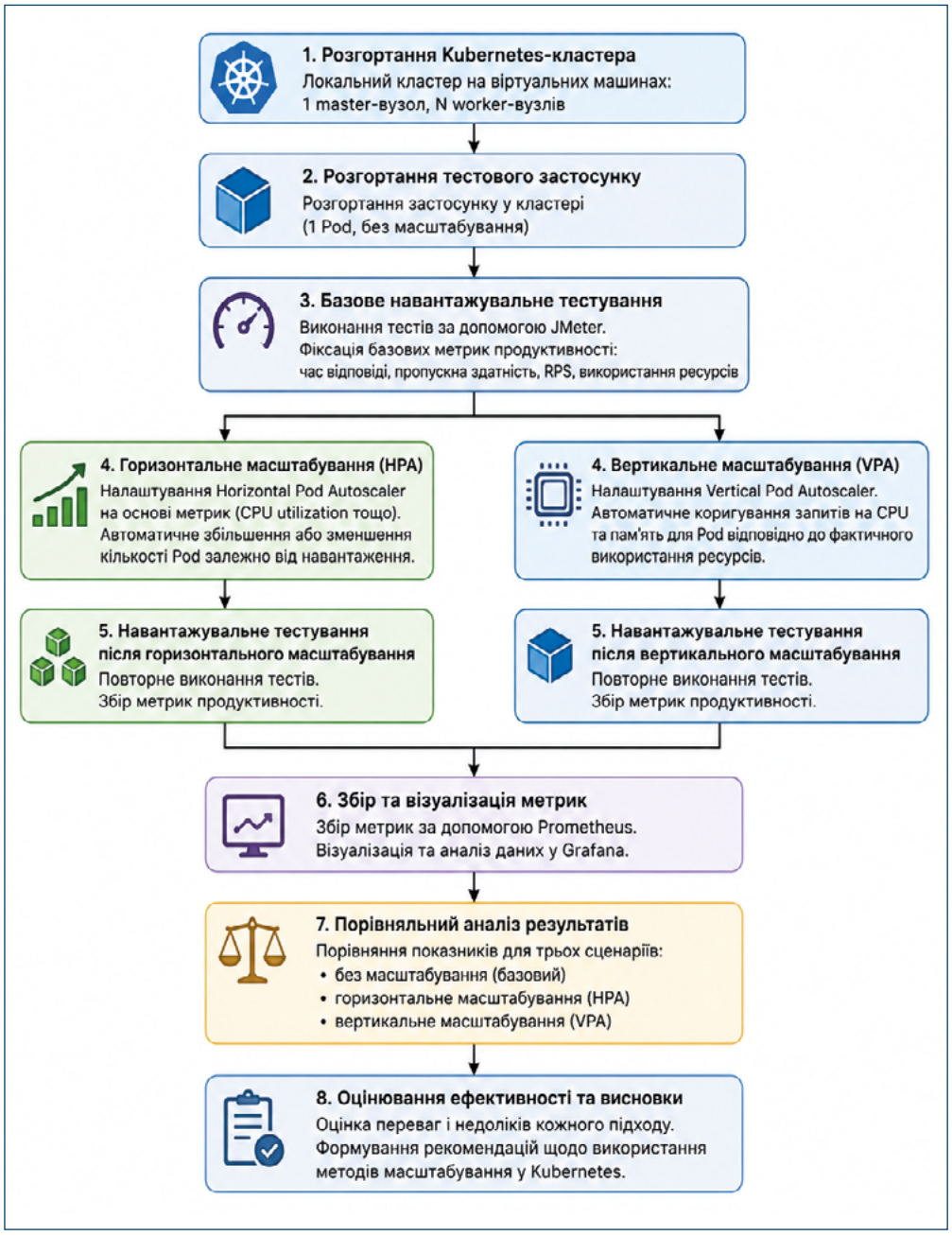


Рис. 3.1. Структурна схема експериментального дослідження механізмів автомасштабування ресурсів у Kubernetes-середовищі

Горизонтальне масштабування реалізовувалося шляхом автоматичної зміни кількості Pod за допомогою механізму Horizontal Pod

Autoscaler (HPA). Рішення про масштабування приймалося на основі показників використання ресурсів, насамперед завантаження процесора. Після налаштування HPA проводилася повторна серія навантажувальних випробувань для визначення ефективності автоматичного збільшення та зменшення кількості екземплярів сервісу.

Вертикальне автомасштабування досліджується із застосуванням Vertical Pod Autoscaler (VPA), який дає змогу адаптувати ресурсне забезпечення контейнерів відповідно до фактичного використання CPU та оперативної пам'яті [61]. Після налаштування VPA виконується аналогічна серія навантажувальних випробувань, що забезпечує зіставність отриманих результатів із базовим режимом та горизонтальним масштабуванням.

На завершальному етапі здійснюється порівняльний аналіз результатів експериментів. Оцінювання проводиться за показниками продуктивності сервісу, стабільності його функціонування та ефективності використання обчислювальних ресурсів. Це дає змогу визначити особливості горизонтального й вертикального автомасштабування та оцінити доцільність їх застосування за різних режимів навантаження.

Сформована методика створює основу для подальшого дослідження механізмів адаптивного керування обчислювальними ресурсами контейнеризованих сервісів у розподілених інформаційно-комунікаційних системах.

3.2. Експериментальне дослідження ефективності вертикального та горизонтального масштабування ресурсів РІКС

Для оцінювання ефективності механізмів масштабування контейнеризованого сервісу в розподіленому Kubernetes-середовищі проведено серію експериментальних досліджень за умов контрольованої зміни інтенсивності вхідного навантаження. Для генерації навантаження використано Apache JMeter, а моніторинг використання обчислювальних ресурсів здійснювався засобами Prometheus та Grafana. Дослідження охоплювало три сценарії функціонування системи: базову конфігурацію застосунку без масштабування, вертикальне збільшення ресурсів

контейнера та горизонтальне збільшення кількості екземплярів сервісу. Отримані результати дають змогу оцінити вплив кожного підходу на продуктивність і ресурсний стан системи.

Оскільки досліджуваний сервіс реалізовано у вигляді REST API, тестове навантаження формувалося шляхом надсилання HTTP-запитів до визначеної кінцевої точки. Конфігурацію API-ресурсу, використаного для формування потоку тестових запитів, наведено на рис. 3.2.

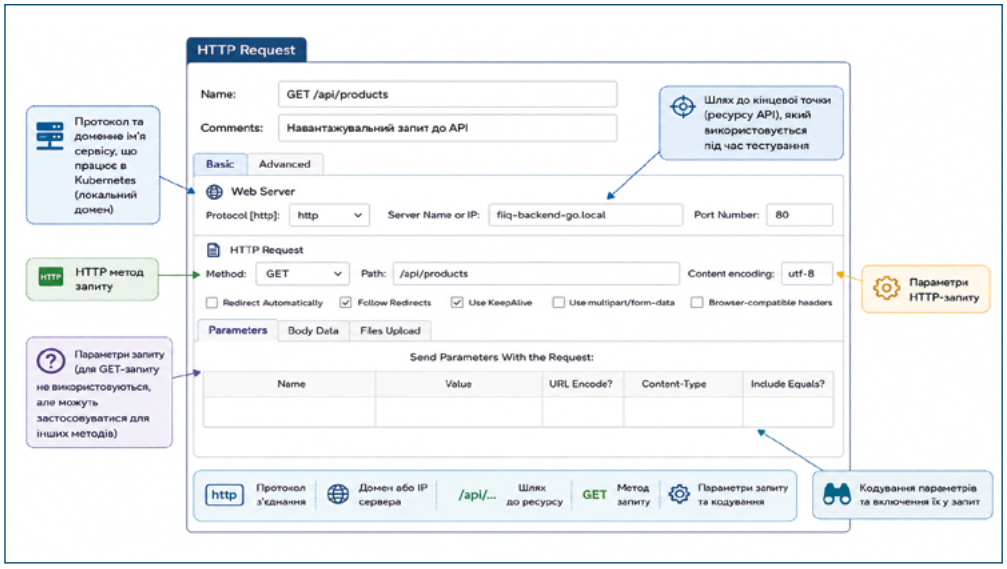


Рис. 3.2. Конфігурація REST API досліджуваного сервісу для формування тестового навантаження

Для визначення граничного навантаження сервісу кількість одночасних користувачів поступово збільшувалася, починаючи з 1 000, до моменту порушення стабільної роботи або виникнення помилок під час оброблення запитів. Після завершення базового тестування аналогічні випробування проводилися для сценаріїв вертикального та горизонтального масштабування. Основними показниками оцінювання були середній час відповіді сервісу, частка помилкових запитів та пропускна здатність системи.

Для імітації одночасної роботи заданої кількості користувачів, які формують запити до API, в Apache JMeter використовується механізм групи паралельних потоків (Thread Group), де кожен потік моделює поведінку окремого користувача. Цільову кількість потоків встановлено

на рівні 6 000, а період нарощування навантаження (Ramp-up period) – 20 с. Це означає, що протягом перших 20 с кількість активних потоків поступово збільшується від початкового значення до 6 000, що дає змогу уникнути миттєвого створення максимального навантаження та відтворити поступове зростання інтенсивності звернень до API. Після досягнення заданого рівня навантаження одночасна робота 6 000 користувачів підтримується протягом наступних 10 с, унаслідок чого загальна тривалість тестового сценарію становить 30 с. Для дослідження поведінки системи за різної інтенсивності навантаження тестування виконується послідовно з цільовою кількістю користувачів від 1 000 до 6 000 із кроком 1 000. Такий підхід дає змогу оцінити зміну показників продуктивності системи зі збільшенням кількості одночасних запитів та визначити рівень навантаження, за якого починається погіршення характеристик її функціонування. Приклад конфігурації відповідного профілю навантаження наведено на рис. 3.3.

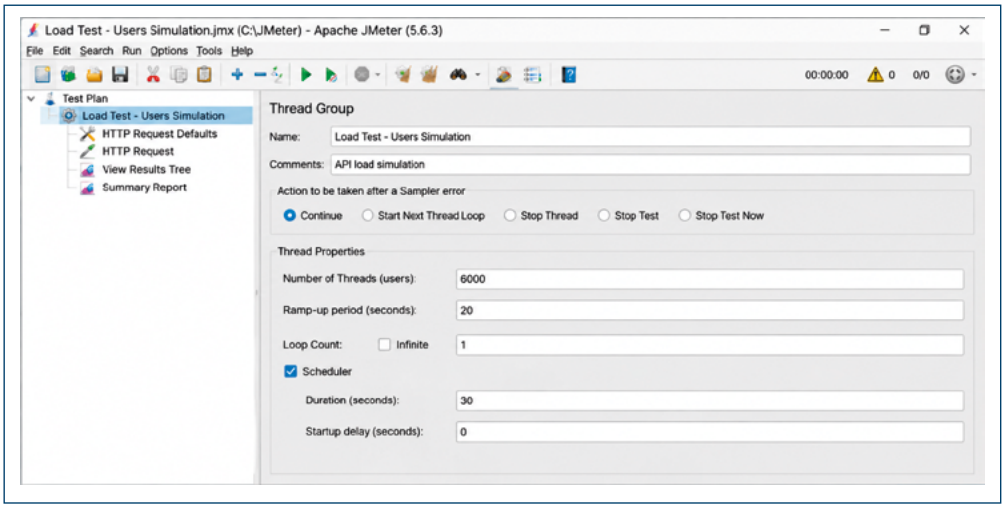


Рис. 3.3. Конфігурація профілю навантаження в Apache JMeter

Початковий етап дослідження проводився для застосунку, який функціонував в одному Pod із базовими параметрами процесорних ресурсів та оперативної пам'яті. Після розгортання сервісу за допомогою kubectl перевірялася коректність створення необхідних об'єктів Kubernetes. Структуру ресурсів, задіяних у базовому сценарії експериментального дослідження, наведено на рис. 3.4.

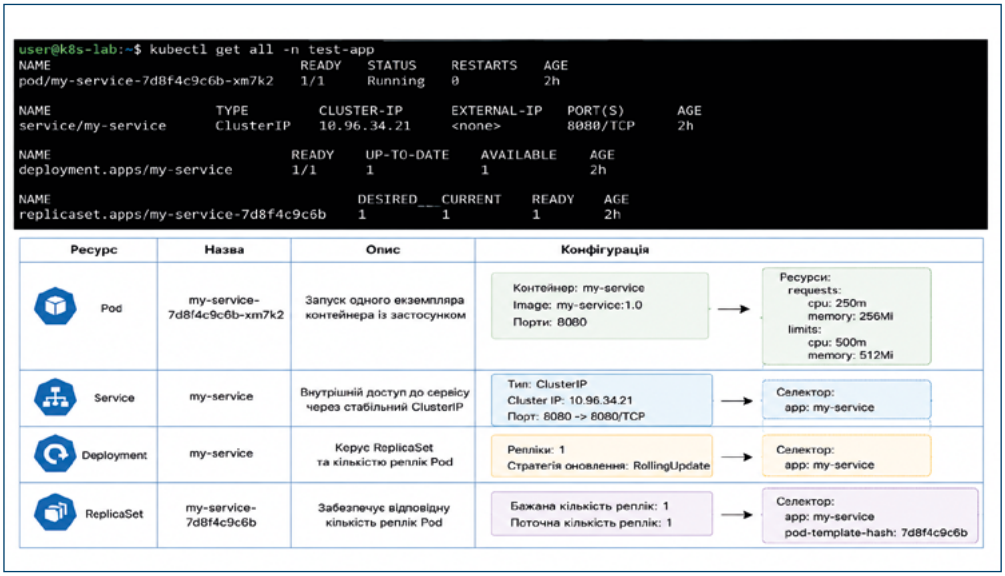


Рис. 3.4. Конфігурація ресурсів Kubernetes у базовому сценарії експериментального дослідження

Аналіз отриманих результатів показав поступове збільшення часу відповіді сервісу зі зростанням кількості одночасних користувачів. За навантаження 6 000 користувачів спостерігалася значна кількість невдалих HTTP-запитів, що свідчило про вичерпання доступних ресурсів контейнера. Для встановлення причини порушення працездатності було проаналізовано стан Pod після завершення тестування. Встановлено, що контейнер завершив роботу через нестачу оперативної пам'яті (OOMKilled), після чого був автоматично перезапущений засобами Kubernetes. Відповідний стан контейнера наведено на рис. 3.5.

Результати моніторингу використання процесорних ресурсів у базовому сценарії наведено на рис. 3.6. За навантаження 1 000 користувачів використання CPU становить 0,08 ядра, що не перевищує запитуваного значення 0,1 ядра. Зі збільшенням кількості користувачів до 2 000 і 3 000 спостерігається зростання використання процесора відповідно до 0,17 та 0,19 ядра. При 4 000 одночасних користувачів використання CPU досягає встановленого ліміту 0,2 ядра та залишається на цьому рівні при подальшому збільшенні навантаження до 6 000 користувачів. Отримані результати свідчать про вичерпання доступного процесорного ресурсу контейнера, внаслідок чого подальше зростання навантаження вже не

супроводжується збільшенням доступної обчислювальної потужності. Це створює передумови для зниження продуктивності мікросервісу та обґрунтовує необхідність застосування механізмів масштабування при подальшому збільшенні кількості користувачі.

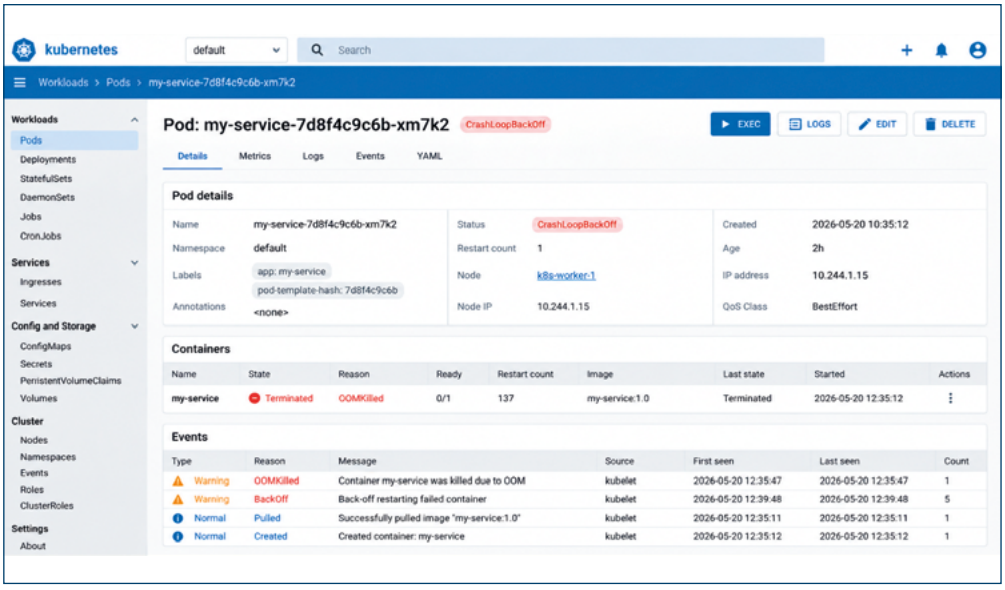


Рис. 3.5. Стан Pod після вичерпання доступного обсягу оперативної пам'яті

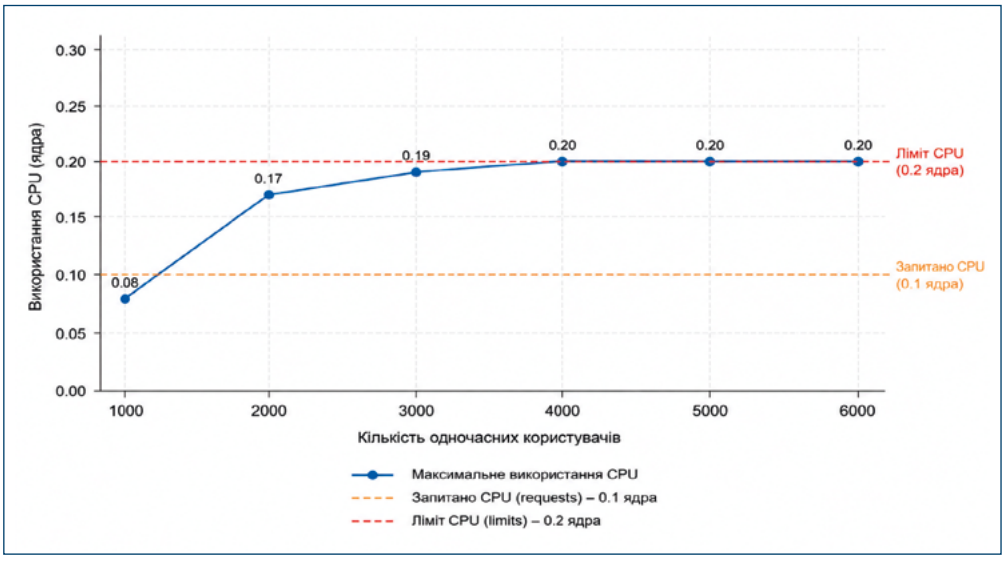


Рис. 3.6. Максимальне використання процесорних ресурсів у базовому сценарії

Аналогічна залежність спостерігалася для оперативної пам'яті (рис. 3.7). За навантаження 1 000 користувачів використання пам'яті становить 92 MiB, що є нижчим за запитуваний обсяг 128 MiB. При збільшенні кількості користувачів до 2 000 використання пам'яті зростає до 118 MiB і також залишається в межах запитуваного значення. При навантаженні 6 000 користувачів використання пам'яті досягає 242 MiB, наближаючись до встановленого ліміту 256 MiB. Таким чином, результати демонструють практично послідовне зростання потреби в оперативній пам'яті зі збільшенням навантаження. Наближення використання пам'яті до граничного значення при 6 000 користувачах свідчить про вичерпання доступного запасу ресурсів базової конфігурації та підвищення ризику завершення роботи контейнера внаслідок нестачі пам'яті (OOMKilled).

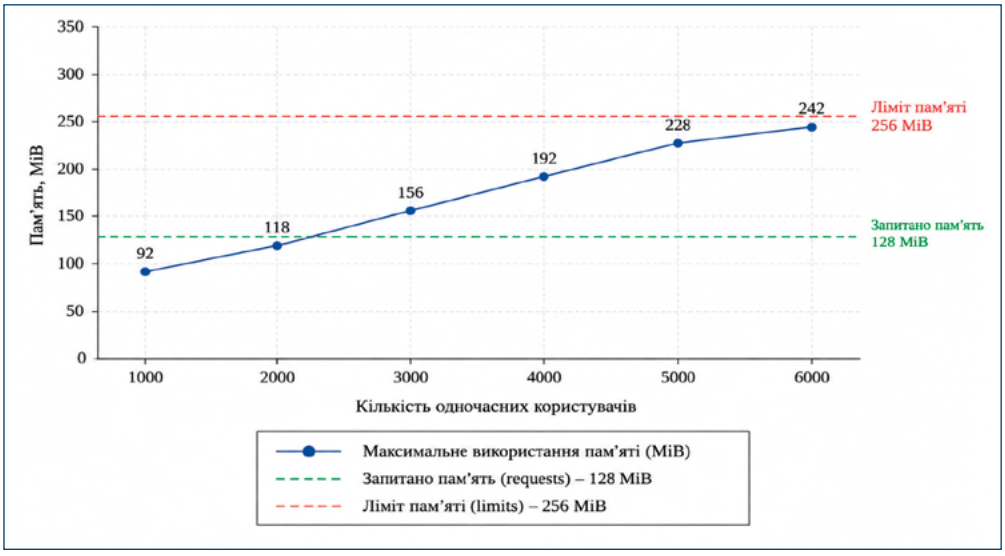


Рис. 3.7. Динаміка використання оперативної пам'яті в базовому сценарії

Наступним етапом стало дослідження вертикального масштабування, яке передбачало збільшення значень requests та limits для процесорних ресурсів і оперативної пам'яті контейнера. Такий підхід забезпечує збільшення доступних для застосунку ресурсів без зміни кількості Pod.

На відміну від базового сценарію, після збільшення обсягу доступних ресурсів сервіс успішно обробляв навантаження до 1 0000 одночасних користувачів без виникнення помилок. Отриманий результат свідчить

про позитивний вплив вертикального збільшення ресурсів на здатність контейнеризованого сервісу обробляти зростаючий потік запитів.

Максимальні значення використання процесорних ресурсів у сценарії вертикального масштабування наведено на рис. 3.8.

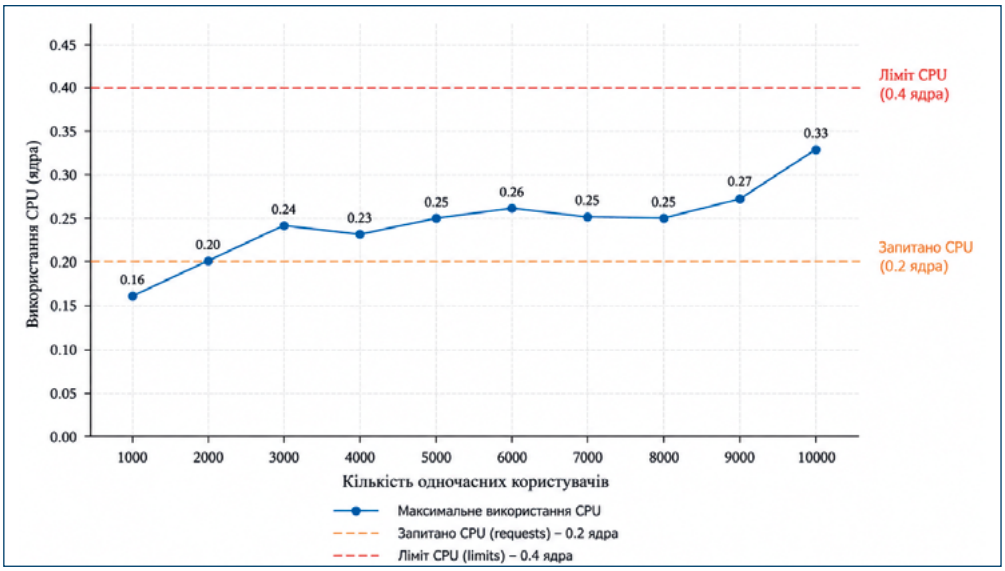


Рис. 3.8. Максимальне використання процесорних ресурсів за вертикального масштабування

Динаміку використання оперативної пам'яті наведено на рис. 3.9. Отримані результати показали, що збільшення доступного обсягу пам'яті дало змогу уникнути аварійного завершення контейнера навіть за високої інтенсивності навантаження.

Останній сценарій дослідження передбачав горизонтальне масштабування сервісу шляхом збільшення кількості Pod з одного до декількох екземплярів. Розподіл вхідного навантаження між Pod дав змогу збільшити сумарну обчислювальну потужність сервісу без зміни ресурсних параметрів окремого контейнера.

На рис. 3.10 наведено максимальне використання процесорних ресурсів двома Pod під час навантажувального тестування із застосуванням горизонтального масштабування. На відміну від базового сценарію, в якому весь потік запитів обслуговував один Pod, у цьому випадку застосунок REST API функціонує у двох екземплярах, а вхідний трафік розподіляється між ними засобами сервісу Kubernetes. Тому наведені

графіки дають змогу оцінити не лише рівень використання CPU, а й рівномірність розподілу навантаження між двома Pod.

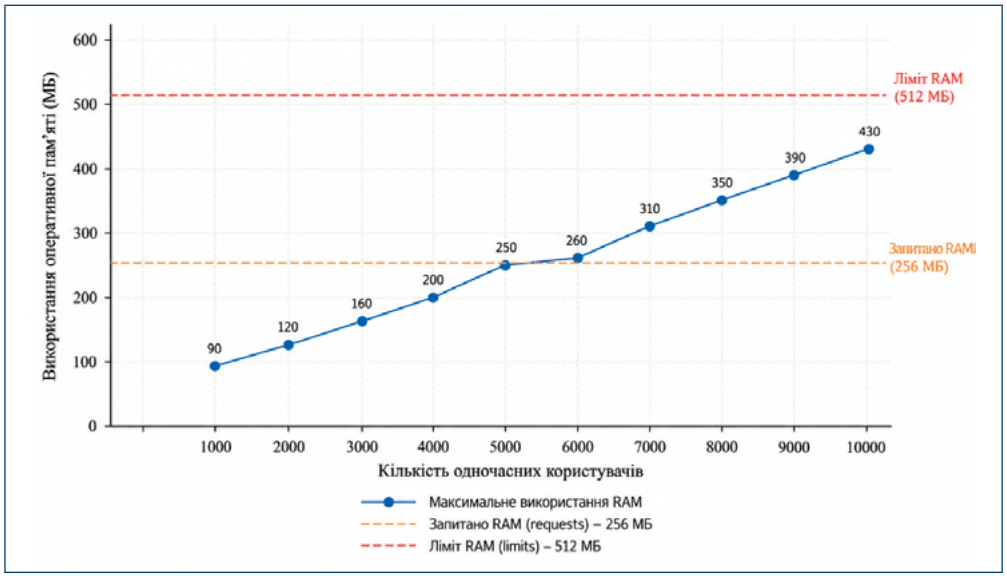


Рис. 3.9. Динаміка використання оперативної пам'яті за вертикального масштабування

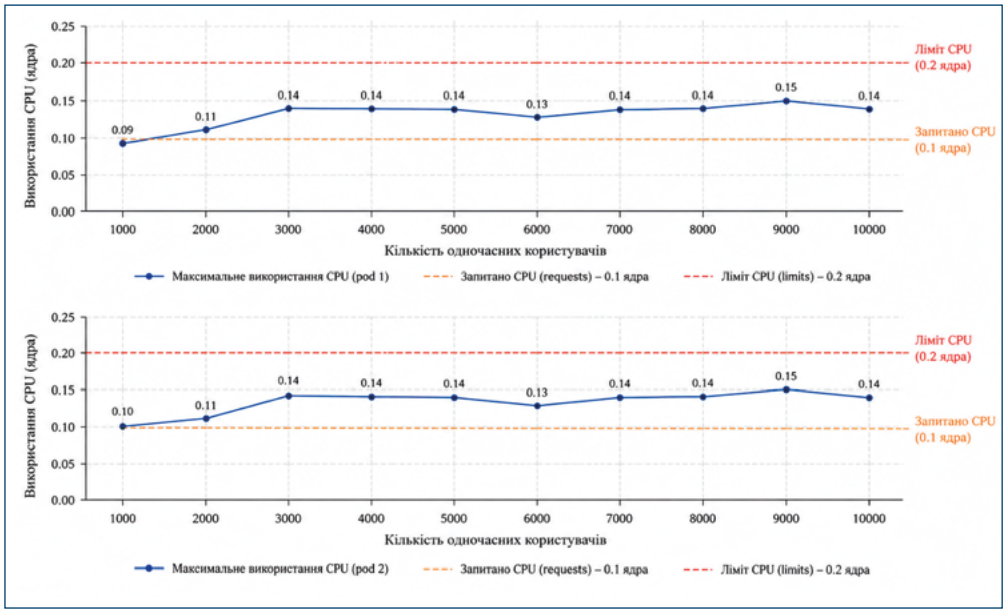


Рис. 3.10. Розгортання додаткових Pod у сценарії горизонтального масштабування

Для Pod 1 за навантаження 1 000 одночасних користувачів максимальне використання CPU становить приблизно 0,09 ядра, тобто є дещо нижчим за запитуване значення requests, установлене на рівні 0,1 ядра. При збільшенні навантаження до 2 000 користувачів використання CPU зростає приблизно до 0,11 ядра, а при 3 000–5 000 користувачів стабілізується на рівні близько 0,14 ядра. За 6 000 користувачів спостерігається незначне зниження до 0,13 ядра, після чого використання CPU знову перебуває в межах 0,14–0,15 ядра. Навіть за максимального досліджуваного навантаження 1 0000 користувачів значення становить близько 0,14 ядра та не наближається до встановленого ліміту 0,2 ядра.

Аналогічна залежність спостерігається для Pod 2. За 1 000 користувачів використання CPU становить близько 0,10 ядра, а при подальшому збільшенні навантаження переважно перебуває в діапазоні 0,11–0,15 ядра. Характер зміни процесорного навантаження практично повторює результати для Pod 1: локальні збільшення та зменшення використання CPU виникають в обох екземплярах, однак істотного перевантаження одного з них не спостерігається.

Порівняння двох графіків показує, що після горизонтального масштабування навантаження розподіляється між Pod 1 і Pod 2 достатньо рівномірно. Різниця між максимальними значеннями використання CPU двома екземплярами є незначною, а зі збільшенням кількості користувачів до 1 0000 не спостерігається стійкого зростання використання процесора до граничного рівня. Значення CPU для обох Pod переважно знаходяться між запитуваним рівнем 0,1 ядра та встановленим лімітом 0,2 ядра.

Цей результат особливо показовий порівняно з базовою конфігурацією з одним Pod. У базовому сценарії використання CPU зі збільшенням навантаження досягало ліміту 0,2 ядра вже приблизно при 4 000 користувачах і надалі залишалося на граничному рівні. Після збільшення кількості екземплярів застосунку до двох процесорне навантаження розподіляється між ними, унаслідок чого навіть при 1 0000 одночасних користувачів кожен Pod використовує лише близько 0,14–0,15 ядра CPU. Отже, горизонтальне масштабування усуває локальне вичерпання процесорного ресурсу окремого Pod та формує запас обчислювальної потужності для обслуговування зростаючого навантаження.

На рис. 3.11 наведено максимальне використання оперативної пам'яті двома Pod під час навантажувального тестування горизонтального масштабування в діапазоні від 1 000 до 1 0000 одночасних користувачів. Як і у випадку використання процесорних ресурсів, характер зміни обсягу оперативної пам'яті для Pod 1 і Pod 2 є практично однаковим, що свідчить про рівномірний розподіл навантаження між двома екземплярами застосунку.

За навантаження від 1 000 до 5 000 одночасних користувачів максимальне використання оперативної пам'яті кожним Pod залишається нижчим за запитуваний обсяг requests, установлений на рівні 128 МБ. Зокрема, для Pod 1 використання пам'яті зростає приблизно від 30 до 115 МБ, а для Pod 2 – від 28 до 116 МБ. При збільшенні кількості користувачів до 6 000 значення перевищує рівень requests і становить близько 140 МБ для обох Pod.

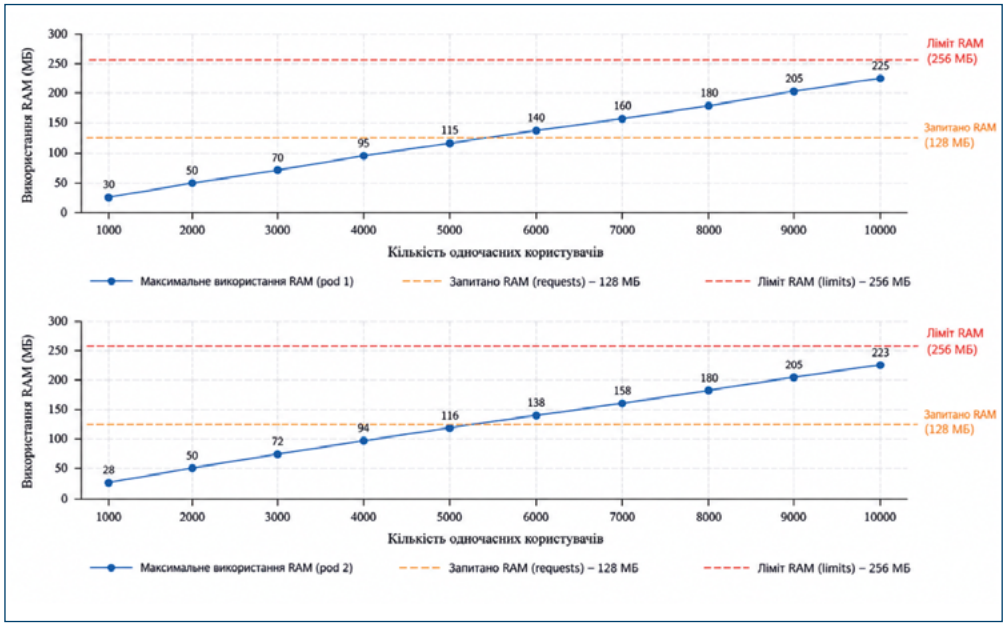


Рис. 3.11. Залежність максимального використання оперативної пам'яті двома Pod від кількості одночасних користувачів при горизонтальному масштабуванні

Подальше збільшення навантаження супроводжується поступовим зростанням використання оперативної пам'яті. При 7 000–9 000 користувачів воно збільшується приблизно від 158–160 до 205 МБ, а за

максимального досліджуваного навантаження 1 0000 користувачів досягає близько 225 МБ для Pod 1 та 223 МБ для Pod 2. Водночас ці значення залишаються нижчими за встановлений ліміт limits, що становить 256 МБ. Отже, навіть за максимального навантаження обидва Pod зберігають певний запас оперативної пам'яті до досягнення граничного значення.

Практично однакові значення використання RAM для Pod 1 і Pod 2 в усьому досліджуваному діапазоні підтверджують збалансований розподіл навантаження між екземплярами застосунку. Порівняно з базовим сценарієм, у якому один Pod уже при 6 000 користувачів використовував близько 242 МБ і наближався до ліміту 256 МБ, горизонтальне масштабування дає змогу розподілити ресурсне навантаження між двома Pod та суттєво розширити діапазон навантаження, за якого система функціонує без досягнення встановленого обмеження оперативної пам'яті. Таким чином, отримані результати підтверджують ефективність горизонтального масштабування з погляду використання ресурсів пам'яті та підвищення здатності системи обслуговувати більшу кількість одночасних користувачів.

На рис. 3.12 наведено результати порівняння частки помилкових HTTP-запитів для базового сценарію, вертикального та горизонтального масштабування. За базової конфігурації при навантаженні від 1 000 до 5 000 одночасних користувачів помилкових HTTP-запитів практично не спостерігається. Однак при збільшенні навантаження до 6 000 користувачів частка помилок різко зростає до 79,97 %, що свідчить про втрату здатності базової конфігурації стабільно обслуговувати такий рівень навантаження. Причиною цього є вичерпання доступної оперативної пам'яті Pod та подальше завершення роботи контейнера.

Натомість при вертикальному та горизонтальному масштабуванні в усьому досліджуваному діапазоні від 1 000 до 1 0000 одночасних користувачів частка помилкових HTTP-запитів залишається близькою до нуля. Отже, збільшення ресурсів окремого Pod у разі вертикального масштабування та розподіл навантаження між двома Pod у разі горизонтального масштабування дали змогу усунути ресурсне обмеження, виявлене під час базового тестування.

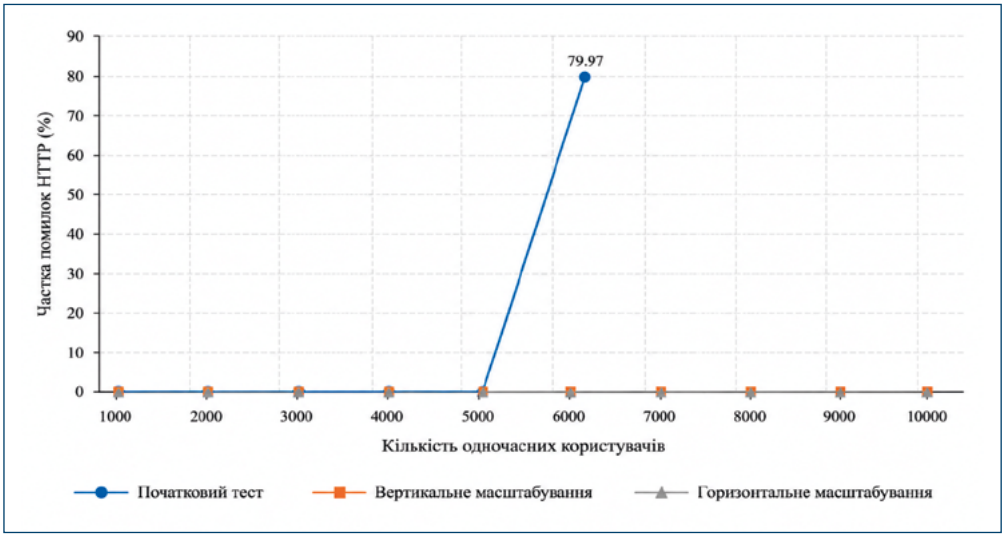


Рис. 3.12. Залежність частки помилкових HTTP-запитів від кількості одночасних користувачів за різних сценаріїв масштабування

Особливо показовим є навантаження 6 000 користувачів: базова конфігурація вже не забезпечує стабільної роботи REST API, тоді як обидва сценарії масштабування продовжують обслуговувати запити без суттєвого зростання частки помилок. Це підтверджує доцільність застосування механізмів масштабування для забезпечення стійкої роботи контейнеризованих мікросервісів за умов зростання інтенсивності навантаження.

На рис. 3.13 наведено залежність середнього часу відгуку REST API від кількості одночасних користувачів для базової конфігурації, вертикального та горизонтального масштабування. Зі збільшенням інтенсивності навантаження для всіх досліджуваних сценаріїв спостерігається зростання часу відгуку, однак його характер та абсолютні значення суттєво відрізняються.

У базовому сценарії збільшення кількості користувачів від 1 000 до 5 000 супроводжується поступовим зростанням середнього часу відгуку приблизно від 20 до 105 тис. мс. При навантаженні 6 000 користувачів значення залишається близьким до 100 тис. мс, однак цей результат необхідно розглядати разом із даними про помилкові HTTP-запити. Як було показано на рис. 3.12, за такого навантаження частка помилок у базовій конфігурації різко зростає до 79,97 %, що свідчить

про досягнення системою критичного рівня навантаження та втрату можливості стабільно обслуговувати вхідні запити. Тому відсутність подальшого зростання середнього часу відгуку в цьому випадку не є ознакою стабілізації продуктивності.

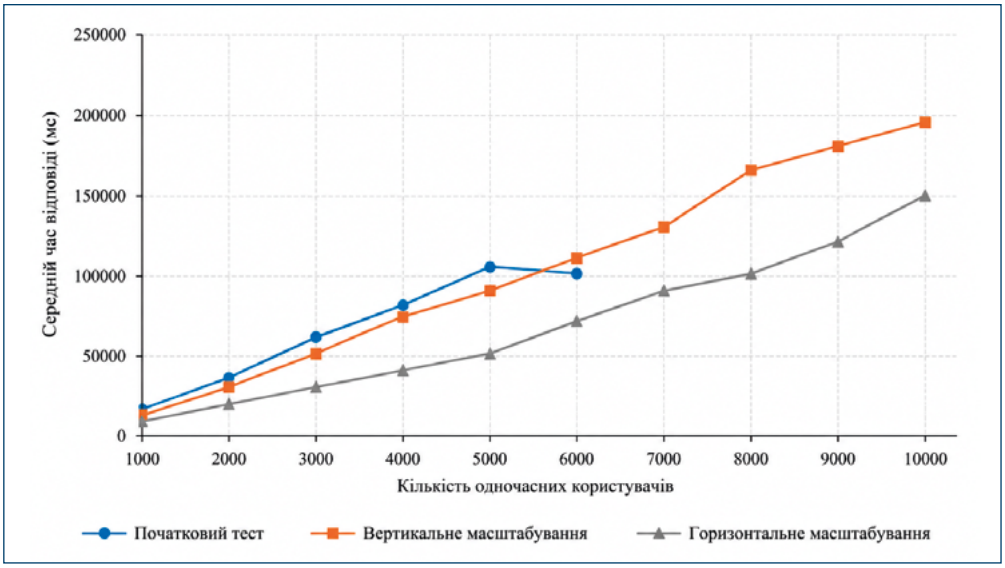


Рис. 3.13. Середній час відгуку для базового, вертикального та горизонтального тестів масштабування ресурсів

Вертикальне масштабування забезпечує працездатність застосунку в усьому досліджуваному діапазоні навантаження до 1 0000 одночасних користувачів. Водночас збільшення ресурсів окремого Pod не усуває зростання часу відгуку. При збільшенні кількості користувачів середній час відгуку поступово зростає приблизно від 15 тис. мс при 1 000 користувачів до 110 тис. мс при 6 000 та до 190 тис. мс при 1 0000 користувачів. Отже, вертикальне масштабування дає змогу уникнути ресурсної відмови, характерної для базової конфігурації, однак за високої інтенсивності запитів спостерігається накопичення затримки їх обслуговування.

Найменші значення середнього часу відгуку отримано для горизонтального масштабування. Розподіл навантаження між двома Pod забезпечує паралельне обслуговування вхідних запитів і, відповідно, зменшує навантаження на кожний окремий екземпляр REST API. При 5 000 користувачів середній час відгуку становить близько 50 тис. мс,

що майже вдвічі менше порівняно з вертикальним масштабуванням. При подальшому збільшенні навантаження час відгуку також зростає, однак залишається нижчим: при 1 0000 користувачів він становить близько 150 тис. мс порівняно з приблизно 190 тис. мс для вертикального масштабування.

Слід зазначити, що абсолютні значення часу відгуку, особливо за навантаження 5 000–1 0000 одночасних користувачів, є значними. Це вказує на роботу досліджуваного застосунку в умовах високого та граничного навантаження, коли інтенсивність надходження запитів наближається до або перевищує можливості їх оперативного обслуговування. У таких умовах збільшується час очікування запитів на оброблення, що призводить до накопичення затримки. Тому результати цього експерименту доцільно використовувати насамперед для порівняльного оцінювання ефективності різних способів масштабування, а не як показник досягнення прийнятного абсолютного часу відгуку.

Таким чином, вертикальне масштабування підвищує ресурсну спроможність окремого Pod і дає змогу уникнути відмови при зростанні навантаження, тоді як горизонтальне масштабування додатково забезпечує розподіл запитів між декількома екземплярами застосунку. За результатами проведеного експерименту саме горизонтальне масштабування забезпечує найменший середній час відгуку в досліджуваному діапазоні навантаження. Водночас значні абсолютні значення цього показника при максимальній кількості користувачів свідчать про необхідність подальшого масштабування системи або оптимізації механізмів оброблення запитів для забезпечення заданих вимог до затримки.

На рис. 3.14 наведено порівняння пропускної здатності REST API для базової конфігурації, вертикального та горизонтального масштабування залежно від кількості одночасних користувачів. Отримані результати показують, що характер зміни пропускної здатності суттєво залежить від способу масштабування застосунку.

Для базової конфігурації пропускна здатність у діапазоні від 1 000 до 6 000 одночасних користувачів залишається практично незмінною і становить близько 30 запитів/с. Відсутність її подальшого зростання зі збільшенням кількості користувачів свідчить про досягнення межі про-

дуктивності початкової конфігурації. При цьому результат для 6 000 користувачів необхідно розглядати разом із попередньо отриманими показниками використання оперативної пам'яті та частки помилкових HTTP-запитів. За такого навантаження відбувається вичерпання доступної пам'яті Pod, а частка помилкових запитів різко зростає. Отже, збереження пропускної здатності на рівні близько 30 запитів/с у цьому випадку не означає стабільної роботи системи, оскільки значна частина запитів уже не обслуговується коректно.

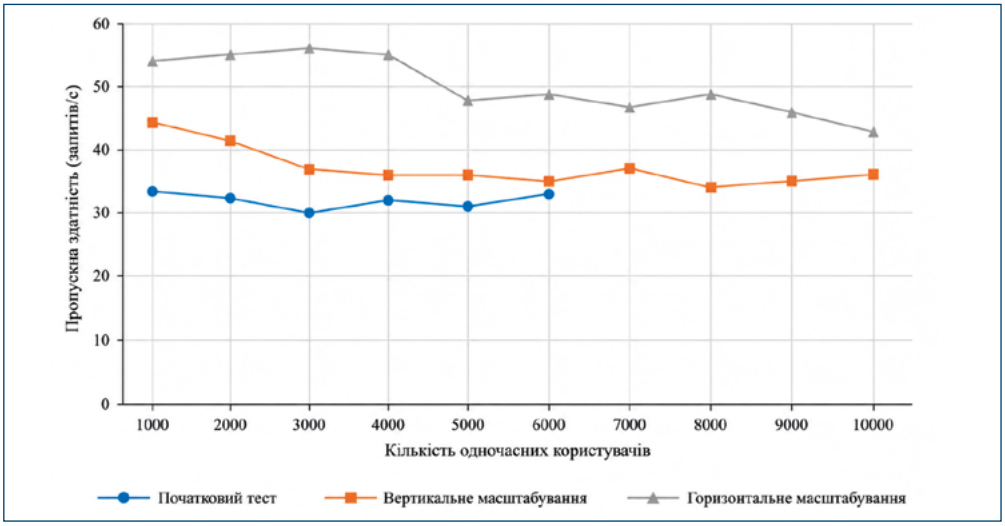


Рис. 3.14. Залежність пропускної здатності REST API від кількості одночасних користувачів за різних сценаріїв масштабування

При вертикальному масштабуванні на початковому етапі тестування пропускна здатність становить близько 44,3 запитів/с, тобто перевищує відповідний показник базової конфігурації. Зі збільшенням кількості користувачів вона поступово зменшується приблизно до 35 запитів/с і надалі, у діапазоні 4 000–1 0000 користувачів, залишається відносно стабільною. Це показує, що збільшення ресурсів окремого Pod дає змогу усунути критичне ресурсне обмеження та забезпечити роботу REST API за більшого навантаження, однак саме по собі не приводить до пропорційного збільшення кількості запитів, які можуть бути опрацьовані за одиницю часу.

Найвищі значення пропускної здатності в усьому досліджуваному діапазоні навантаження спостерігаються при горизонтальному

масштабуванні. На відміну від вертикального підходу, збільшення кількості Pod забезпечує можливість паралельного обслуговування запитів декількома екземплярами REST API. Розподіл вхідного трафіку між ними зменшує навантаження на окремий Pod та дає змогу системі обробляти більшу кількість запитів за одиницю часу. Отриманий результат узгоджується також із результатами аналізу середнього часу відгуку на рис. 3.13, де горизонтальне масштабування забезпечувало нижчі значення затримки порівняно з іншими досліджуваними конфігураціями.

На рис. 3.15 наведено приріст пропускної здатності REST API, отриманий при застосуванні горизонтального масштабування порівняно з вертикальним. Зі збільшенням кількості одночасних користувачів від 1 000 до 3 000 приріст пропускної здатності зростає з 10 до 19 запитів/с і зберігає максимальне значення 19 запитів/с при 4 000 користувачів. Це свідчить про те, що в цьому діапазоні розподіл навантаження між декількома Pod забезпечує суттєвий виграш порівняно зі збільшенням ресурсів одного Pod.

При подальшому збільшенні навантаження спостерігається коливання приросту пропускної здатності в межах 10–15 запитів/с, а при 1 0000 користувачів він зменшується до 7 запитів/с. Така тенденція вказує на поступове наближення горизонтально масштабованої конфігурації до межі її ресурсних можливостей. Отримані результати підтверджують, що горизонтальне масштабування забезпечує вищу пропускну здатність у всьому досліджуваному діапазоні, однак самого збільшення кількості Pod недостатньо для необмеженого зростання продуктивності РІКС, що обґрунтовує необхідність адаптивного вибору моменту та способу масштабування залежно від поточного і прогнозованого стану системи.

Узагальнення результатів проведених експериментів показує, що як вертикальне, так і горизонтальне масштабування дають змогу розширити діапазон навантаження, за якого контейнеризований застосунок зберігає працездатність. У базовій конфігурації основним обмежувальним фактором стала оперативна пам’ять: при 6 000 одночасних користувачів її використання наблизилося до встановленого ліміту, що призвело до завершення роботи контейнера та різкого зростання частки помилкових HTTP-запитів. Збільшення доступних ресурсів при верти-

кальному масштабуванні усунуло це обмеження та забезпечило роботу застосунку до 1 0000 користувачів.

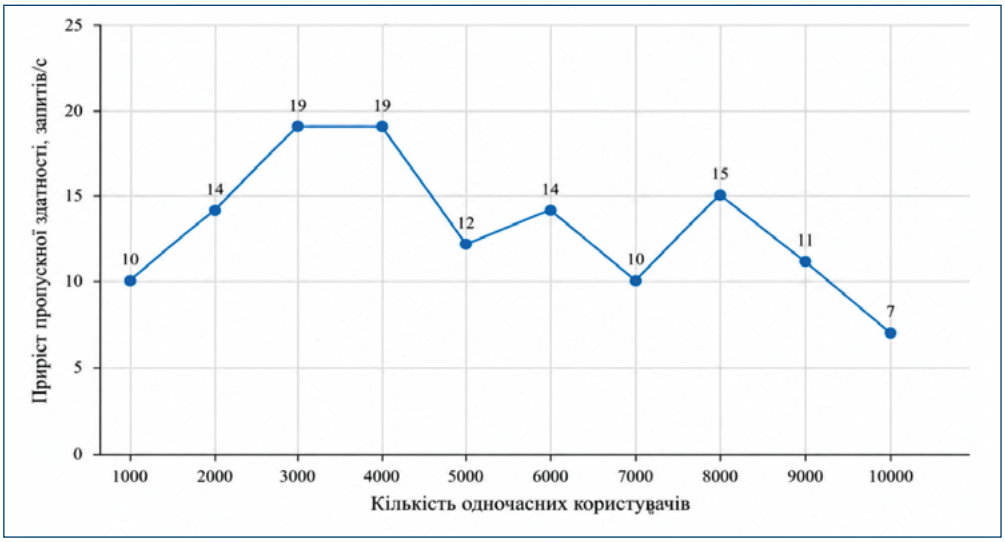


Рис. 3.15. Залежність приросту пропускної здатності REST API при горизонтальному масштабуванні порівняно з вертикальним від кількості одночасних користувачів

Горизонтальне масштабування, крім усунення ресурсного обмеження окремого Pod, забезпечило розподіл навантаження між двома екземплярами REST API. Результати моніторингу використання CPU та оперативної пам'яті показали близькі значення для обох Pod, що свідчить про достатньо рівномірний розподіл навантаження. Водночас саме горизонтальна конфігурація продемонструвала найвищу пропускну здатність і найменший середній час відгуку серед досліджуваних сценаріїв.

Таким чином, результати експериментального дослідження показують, що вертикальне масштабування насамперед дає змогу подолати обмеження ресурсів окремого Pod, тоді як горизонтальне масштабування забезпечує додатковий ефект за рахунок паралельного обслуговування запитів декількома екземплярами застосунку. Для досліджуваного REST API горизонтальний підхід виявився ефективнішим за сукупністю показників пропускної здатності, середнього часу відгуку та розподілу ресурсного навантаження.

Отримані результати показали, що статичне виділення ресурсів обмежує здатність системи адаптуватися до збільшення кількості користу-

вачів і при досягненні критичного навантаження може призводити до вичерпання ресурсів окремого контейнера, зростання затримки та появи помилок під час обслуговування запитів.

Застосування вертикального масштабування дало змогу розширити допустимий діапазон навантаження шляхом збільшення ресурсів окремого Pod, тоді як горизонтальне масштабування забезпечило розподіл навантаження між декількома екземплярами застосунку. За результатами проведених експериментів горизонтальний підхід продемонстрував кращі показники пропускної здатності та середнього часу відгуку, а також забезпечив рівномірніший розподіл використання процесорних ресурсів і оперативної пам'яті між Pod.

У контексті побудови розподілених інформаційно-комунікаційних систем отримані результати підтверджують доцільність використання адаптивного керування ресурсами як одного з механізмів забезпечення масштабованості, продуктивності та стійкості системи до зміни інтенсивності навантаження. Розподіл обчислювального навантаження між декількома екземплярами мікросервісів дає змогу не лише уникнути перевантаження окремих вузлів, а й ефективніше використовувати доступні ресурси розподіленої інфраструктури.

Водночас результати показують, що самого збільшення ресурсів або кількості Pod недостатньо для гарантованого забезпечення заданих показників якості обслуговування. Значне зростання часу відгуку за високого навантаження свідчить про необхідність урахування поточного стану системи, інтенсивності вхідного трафіку та ресурсного навантаження під час прийняття рішень щодо масштабування. Це обґрунтовує необхідність застосування в РІКС методів динамічного та адаптивного керування ресурсами, здатних своєчасно визначати момент масштабування та вибирати відповідний спосіб зміни ресурсної конфігурації системи.

Особливо перспективним у цьому контексті є перехід від реактивного до проактивного масштабування. На відміну від реактивного підходу, за якого зміна ресурсів виконується після фіксації зростання навантаження або досягнення встановлених порогових значень, проактивний підхід передбачає прогнозування майбутньої зміни навантаження та завчасне виділення необхідних ресурсів. Це дає змогу скоротити часовий

інтервал між виникненням потреби в додаткових ресурсах та їх фактичним наданням, запобігти перевантаженню окремих компонентів і зменшити ймовірність різкого зростання часу відгуку. Таким чином, поєднання моніторингу поточного стану РІКС із прогнозуванням динаміки навантаження створює основу для побудови проактивних механізмів керування ресурсами, здатних адаптувати конфігурацію системи ще до досягнення критичних значень її параметрів.

3.3. Методи машинного навчання для прогнозування навантаження та проактивного масштабування ресурсів

Методи машинного навчання є одним із перспективних інструментів інтелектуального керування ресурсами розподілених інформаційно-комунікаційних систем. Їх застосування дає змогу аналізувати накопичені дані про функціонування системи, виявляти приховані закономірності у зміні навантаження та формувати прогноз його подальшої динаміки. На відміну від традиційних алгоритмів із наперед визначеними правилами, моделі машинного навчання встановлюють залежності безпосередньо на основі історичних даних, що є особливо важливим для систем із динамічним і нерівномірним характером навантаження.

Основою машинного навчання є формування математичної моделі, яка відображає залежність між сукупністю вхідних ознак та цільовими показниками. Після навчання така модель може використовуватися для прогнозування майбутніх значень або класифікації нових спостережень. У задачах керування ресурсами хмарних і розподілених систем машинне навчання застосовується для прогнозування використання процесорних ресурсів, оперативної пам’яті, інтенсивності мережевого трафіку та визначення необхідності масштабування обчислювальної інфраструктури.

За характером формування навчальної моделі основними підходами є навчання з учителем (Supervised Learning), навчання без учителя (Unsupervised Learning) та навчання з підкріпленням (Reinforcement Learning). Їх узагальнену структуру та можливі напрями застосування для керування ресурсами розподілених систем наведено на рис. 3.16.



Рис. 3.16. Основні підходи машинного навчання для інтелектуального керування ресурсами розподілених систем

Навчання з учителем передбачає використання набору даних із відомими цільовими значеннями. У процесі навчання визначаються залежності між вхідними характеристиками та відповідними результатами, на основі яких формується модель для прогнозування нових значень. Такий підхід доцільно застосовувати для прогнозування навантаження, коли доступні історичні дані про використання обчислювальних ресурсів і відповідні часові характеристики.

Навчання без учителя використовується для аналізу даних без попередньо визначених міток. Основними завданнями при цьому є виявлення прихованої структури даних, кластеризація та ідентифікація аномальних станів. У розподілених інформаційно-комунікаційних системах такі методи можуть застосовуватися для виявлення нетипових профілів навантаження, групування сервісів за характером використання ресурсів та аналізу поведінки користувачів.

Навчання з підкріпленням ґрунтується на взаємодії програмного агента із середовищем, під час якої агент виконує певні дії та отри-

має відповідну винагороду залежно від їх результату. На основі накопиченого досвіду формується політика керування, спрямована на максимізацію довгострокової винагороди. У хмарних і контейнеризованих середовищах цей підхід може використовуватися для динамічного розподілу ресурсів та адаптивного формування політик масштабування.

Для прогнозування навантаження застосовують різні методи машинного навчання. Одним із базових є лінійна регресія, що дає змогу апроксимувати залежності між вхідними показниками та прогнозованою величиною. Її перевагами є відносна простота реалізації та невисока обчислювальна складність, однак ефективність такого підходу знижується за наявності складних нелінійних залежностей у часових рядах навантаження.

Для моделювання складніших залежностей можуть використовуватися нейронні мережі, багатошарова структура яких забезпечує можливість апроксимації нелінійних процесів. Такі моделі застосовуються для аналізу часових рядів, мережевого трафіку та прогнозування майбутнього рівня використання обчислювальних ресурсів.

Метод опорних векторів (Support Vector Machine, SVM), а для задач прогнозування неперервних величин – його регресійна модифікація Support Vector Regression (SVR), також може використовуватися для аналізу навантаження. Застосування ядрових функцій дає змогу враховувати нелінійні залежності між ознаками, що розширює можливості використання цього підходу для прогнозування поведінки системи.

Ефективність моделей машинного навчання значною мірою залежить від вибору їх гіперпараметрів. Залежно від типу моделі до них можуть належати швидкість навчання, структура та кількість прихованих шарів нейронної мережі, коефіцієнти регуляризації, параметри ядрових функцій тощо. Для пошуку раціональної конфігурації можуть застосовуватися методи перебору за сіткою (Grid Search) та випадкового пошуку (Random Search).

У задачах прогнозування навантаження особливе значення мають моделі, орієнтовані на опрацювання часових рядів. Використання попередніх значень контрольованих показників дає змогу оцінювати їх майбутню динаміку та завчасно виявляти тенденції до збільшення або

зменшення навантаження. Отриманий прогноз може бути використаний системою керування для прийняття рішення про зміну обсягу ресурсів до моменту фактичного виникнення перевантаження.

Таким чином, застосування машинного навчання розширює можливості адаптивного керування ресурсами розподілених інформаційно-комунікаційних систем. Прогнозування майбутнього навантаження створює передумови для переходу від реактивного до проактивного масштабування, за якого необхідні обчислювальні ресурси можуть бути підготовлені до очікуваної зміни інтенсивності запитів. Узагальнену послідовність такого процесу наведено на рис. 3.17.



Рис. 3.17. Структурна схема прогнозного керування ресурсами на основі машинного навчання

3.4. Метод проактивного масштабування обчислювальних ресурсів РІКС на основі моделей LSTM та Holt–Winters

У РІКС прогнозування навантаження є важливою складовою ефективного керування обчислювальними ресурсами. Використання мікросервісної архітектури, контейнеризації та платформ оркестрації, зокрема Kubernetes, зумовлює необхідність не лише реагувати на поточні зміни навантаження, а й завчасно оцінювати його майбутню динаміку. Прогнозування дає змогу підготувати необхідні ресурси до очікуваного

збільшення інтенсивності запитів і зменшити ймовірність перевантаження сервісів.

Навантаження хмарного сервісу доцільно розглядати як часовий ряд, тобто послідовність значень контрольованого показника, впорядкованих у часі. Такими показниками можуть бути кількість HTTP-запитів, використання процесорних ресурсів і оперативної пам'яті, інтенсивність мережевого трафіку, кількість активних користувачів або довжина черги повідомлень. Аналіз часових рядів дає змогу виявляти тренди, періодичні коливання та характерні цикли зміни навантаження, які можуть бути використані для прогнозування майбутнього стану системи.

Для прогнозування часових рядів можуть застосовуватися класичні статистичні моделі, зокрема ARIMA та SARIMA. Вони є ефективними для часових рядів із достатньо вираженою статистичною структурою, однак характер навантаження реальних хмарних сервісів часто визначається складними нелінійними залежностями. На його динаміку можуть впливати поведінка користувачів, особливості бізнес-процесів, оновлення програмного забезпечення та зовнішні події. За таких умов доцільним є застосування методів машинного та глибинного навчання.

Одним із підходів до опрацювання послідовних даних є рекурентні нейронні мережі (Recurrent Neural Networks, RNN), які враховують інформацію про попередні стани послідовності. Це дає змогу моделювати часові залежності між значеннями навантаження. Водночас для класичних RNN характерною є проблема зникання або вибуху градієнта, що ускладнює навчання моделі за наявності довготривалих залежностей.

Для зменшення впливу зазначених обмежень використовуються мережі довгої короткочасної пам'яті (Long Short-Term Memory, LSTM). Архітектура LSTM містить стан комірки та керовані вентилі, за допомогою яких регулюється збереження, оновлення та передавання інформації між часовими кроками. Завдяки цьому модель здатна враховувати довготривалі залежності в часовому ряді, зокрема повторювані добові та тижневі цикли зміни навантаження.

Водночас для навантаження інформаційно-комунікаційних сервісів характерними можуть бути не лише нелінійні зміни, а й виражені трен-

дові та сезонні складові. Для їх урахування доцільним є використання методу Holt–Winters, який ґрунтується на експоненційному згладжуванні часового ряду та передбачає окреме оцінювання його рівня, тренду й сезонності. На відміну від LSTM, цей метод не потребує побудови складної нейромережевої моделі та може забезпечувати достатню точність прогнозування для навантажень із регулярною часовою структурою.

Вхідними даними LSTM-моделі є послідовності історичних значень контрольованих метрик. Наприклад, для прогнозування кількості запитів до сервісу можуть використовуватися спостереження за попередні години або дні. Перед формуванням навчальної вибірки доцільно проаналізувати часовий ряд для виявлення трендів, періодичних піків, спадів навантаження та інших закономірностей, які характеризують роботу досліджуваного сервісу.

Перед навчанням LSTM-моделі виконується попередня підготовка даних, що передбачає їх очищення, нормалізацію або масштабування та формування навчальної і тестової вибірок. Масштабування значень до заданого діапазону сприяє стабільнішому процесу навчання нейронної мережі та зменшує вплив різних масштабів вхідних ознак.

Під час формування навчальної і тестової вибірок необхідно зберігати часову послідовність спостережень. На відміну від багатьох задач класифікації, випадкове перемішування часових даних може призвести до використання інформації з майбутніх моментів часу під час навчання моделі. Тому початкова частина часового ряду використовується для навчання, а наступний часовий інтервал – для тестування. Такий поділ відповідає практичному сценарію, за якого прогноз формується виключно на основі доступних історичних даних.

Важливим параметром формування вхідних послідовностей є довжина вікна спостереження, яка визначає кількість попередніх часових значень, використаних для прогнозування наступного стану. Наприклад, для погодинного часового ряду вікно зі 168 спостережень охоплює попередні сім діб, що дає змогу враховувати добову та тижневу періодичність навантаження.

Типова модель прогнозування може містити один або декілька LSTM-шарів і вихідний повнозв’язний шар (Dense), який формує прогнозоване значення навантаження. Навчання виконується шляхом мінімізації

похибки між фактичними та прогнозованими значеннями. Для задач регресійного прогнозування як функція втрат може використовуватися середньоквадратична помилка (Mean Squared Error, MSE), а для оптимізації параметрів моделі – алгоритм Adam.

Після завершення навчання модель оцінюється на тестовій частині часового ряду шляхом зіставлення прогнозованих і фактичних значень навантаження. Отримані результати дають змогу визначити здатність моделі відтворювати динаміку досліджуваного процесу та оцінити можливість її подальшого використання для прогнозного керування ресурсами. Особливості LSTM та Holt–Winters визначають доцільність їх спільного використання в системі проактивного керування ресурсами. LSTM дає змогу моделювати складні нелінійні залежності у часовому ряді, тоді як Holt–Winters ефективно враховує трендову та сезонну складові навантаження. Тому використання обох моделей дає можливість формувати альтернативні прогнози та вибирати модель, яка точніше відтворює поточний характер зміни навантаження.

Для цього прогнозування може виконуватися паралельно за допомогою LSTM та Holt–Winters, після чого точність отриманих прогнозів оцінюється на контрольному часовому інтервалі. Як критерій вибору доцільно використовувати середньоквадратичну помилку (Mean Squared Error, MSE). Модель із меншим значенням MSE визначається як більш придатна для поточного режиму навантаження, а сформований нею прогноз використовується для прийняття рішення щодо масштабування ресурсів.

Такий підхід формує основу гібридного методу проактивного масштабування обчислювальних ресурсів РІКС на основі моделей LSTM та Holt–Winters. На відміну від реактивного масштабування, де керуюча дія формується після досягнення встановленого порогового значення контрольованої метрики, прогнозування дає змогу оцінити майбутню потребу в ресурсах і розпочати масштабування завчасно. У Kubernetes-середовищі це створює можливість підготувати необхідну кількість Pod до очікуваного збільшення навантаження, зменшити ризик виникнення дефіциту ресурсів і скоротити період роботи сервісу в перевантаженому стані

Послідовність формування та використання гібридного методу для масштабування ресурсів РІКС наведено на рис. 3.18.

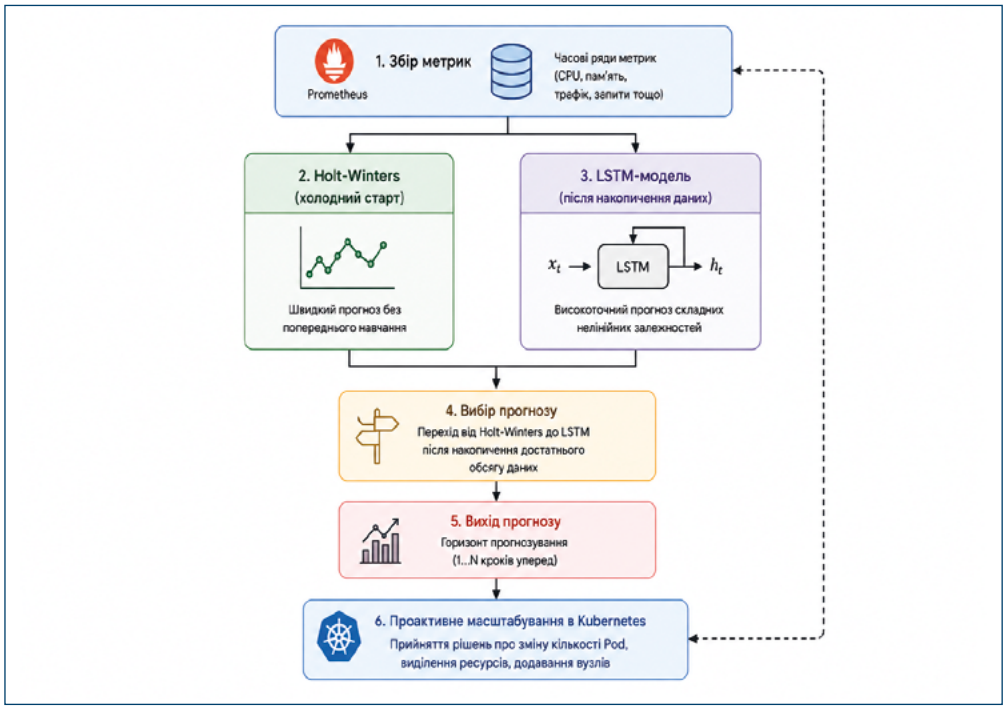


Рис. 3.18. Послідовність формування LSTM-моделі для прогнозування навантаження хмарного сервісу

Окремим завданням є багатокрокове прогнозування, за якого необхідно визначити навантаження не лише для наступного часового кроку, а й для певного майбутнього інтервалу. За рекурсивного підходу перше прогнозоване значення формується на основі фактичної історичної послідовності, після чого отримані прогнозні значення можуть використовуватися для формування наступних прогнозів. Це дає змогу оцінювати навантаження на декілька часових кроків наперед, однак зі збільшенням горизонту прогнозування можливе накопичення похибки.

Прогнозовані значення навантаження можуть бути використані як вхідні дані для механізму проактивного масштабування Kubernetes. Якщо прогноз свідчить про майбутнє зростання інтенсивності запитів, система керування може завчасно ініціювати збільшення кількості реплік Pod, коригування ресурсного забезпечення контейнерів або, за наявності відповідного механізму керування кластером, збільшення кількості робочих вузлів. За прогнозованого зменшення навантаження надлишкові ресурси можуть бути поступово вивільнені.

Інтеграція LSTM-моделі із засобами моніторингу Prometheus та візуалізації Grafana дає змогу сформувати прогнозно-аналітичний рівень системи керування ресурсами. Prometheus забезпечує накопичення часових рядів контрольованих метрик, результати прогнозування можуть використовуватися механізмом прийняття рішень, а Grafana – для візуального аналізу фактичних і прогнозованих показників. У такій архітектурі рішення щодо масштабування можуть враховувати не лише поточний стан ресурсів, а й очікувану зміну навантаження.

Якість прогнозування LSTM-моделі залежить від характеристик вхідних даних, структури нейронної мережі та вибраних гіперпараметрів. До основних параметрів належать довжина вхідного вікна, кількість LSTM-шарів і нейронів, швидкість навчання, кількість епох та розмір пакета (batch size). Надмірне ускладнення архітектури може призвести до перенавчання, тоді як недостатня складність моделі може не забезпечити відтворення істотних закономірностей часового ряду. Тому вибір конфігурації має супроводжуватися оцінюванням якості прогнозу на даних, які не використовувалися під час навчання.

Таким чином, LSTM-модель може бути використана як складова прогнозного керування ресурсами розподіленої хмарної інфраструктури. Аналіз історичних часових рядів дає змогу оцінювати майбутню динаміку навантаження та завчасно ініціювати масштабування контейнеризованих сервісів. У поєднанні з Kubernetes і засобами моніторингу такий підхід створює основу для переходу від реактивного до проактивного керування обчислювальними ресурсами, спрямованого на підвищення стабільності сервісів та ефективності використання інфраструктури.

Функціонування LSTM-комірки визначається послідовністю операцій над поточним вхідним вектором x_t, попереднім прихованим станом h_{t-1} та станом пам'яті c_{t-1}. На першому етапі вентиль забування визначає частину інформації попереднього стану пам'яті, яку необхідно зберегти:

$$f_t = \sigma\,(W_f\,[h_{t-1}, x_t] + b_f) \qquad (3.1)$$

де W_f і b_f – матриця ваг та вектор зміщення вентиля забування; σ – сигмоїдна функція активації. Далі вхідний вентиль визначає частину нової інформації, яка буде записана до стану пам'яті:

$$i_t = \sigma\,(W_i\,[h_{t-1}, x_t] + b_i) \qquad (3.2)$$

Кандидат на оновлення стану пам'яті формується відповідно до виразу

$$\tilde{c}_t = \tanh\,(W_c\,[h_{t-1}, x_t] + b_c) \qquad (3.3)$$

де функція tanh( ) забезпечує нелінійне перетворення значень. Поточний стан пам'яті визначається шляхом поєднання збереженої частини попереднього стану та нової інформації:

$$c_t = f_t\,c_{t-1}, x_t + i_c\,\tilde{c}_t \qquad (3.4)$$

У виразі 3.4 поєднання складових стану пам'яті здійснюється із застосуванням операції поелементного множенн. Такий механізм забезпечує вибіркове збереження інформації про попередні стани часового ряду та її оновлення відповідно до поточних вхідних даних.

На наступному етапі вихідний вентиль визначає інформацію, яка використовується для формування прихованого стану:

$$o_t = \sigma\,(W_o\,[h_{t-1}, x_t] + b_o) \qquad (3.5)$$

а прихований стан LSTM-комірки обчислюється як

$$h_t = o_t\,\tanh\,(c_t) \qquad (3.6)$$

Отже, LSTM-модель формує прогноз з урахуванням накопичених закономірностей часового ряду та може використовуватися для відтворення складної нелінійної динаміки навантаження. Водночас застосування LSTM потребує попереднього навчання та достатньої кількості історичних спостережень. Це створює обмеження на початковому етапі функціонування нового сервісу, коли необхідна історія метрик ще не сформована.

Для забезпечення прогнозування в умовах обмеженого обсягу історичних даних у гібридній моделі використовується метод Holt-Winters. Він ґрунтується на експоненційному згладжуванні та враховує рівень, тренд і сезонну складову часового ряду. Для адитивної моделі Holt-Winters поточний рівень визначається як:

$$L_t = a\,(y_t - S_{t-m}) + (1-a)\,(L_{t-1} + T_{t-1}) \qquad (3.7)$$

де y_t – фактичне значення навантаження в момент часу t; L_t – оцінка рівня часового ряду; T_t – трендова складова; S_{t-m} – сезонна складова з періодом m; a- коефіцієнт згладжування рівня.

Трендова складова характеризує напрям та інтенсивність зміни навантаження і визначається як

$$T_t = \beta\,(L_t - L_{t-1}) + (1-\beta)\,T_{t-1} \qquad (3.8)$$

де β – коефіцієнт згладжування тренду. Сезонна складова оновлюється відповідно до виразу

$$S_t = \gamma\,(y_t - L_t) + (1-\gamma)\,S_{t-m} \qquad (3.9)$$

де γ визначає швидкість адаптації моделі до зміни сезонного профілю навантаження. Прогноз навантаження на h часових кроків уперед для адитивної моделі Holt-Winters може бути представлений як

$$\hat{y}_{t+h} = L_t + hT_t + S_{t-m+h} \qquad (3.10)$$

Таким чином, Holt-Winters формує прогноз на основі поточного рівня, тренду та сезонності часового ряду без необхідності тривалого попереднього навчання. Це робить його доцільним для використання на етапі «холодного старту», хоча для коректного оцінювання сезонної складової все одно необхідна певна початкова історія спостережень.

Поєднання Holt-Winters та LSTM дає змогу реалізувати дворівневий механізм прогнозування навантаження. На початковому етапі використовується Holt-Winters, що забезпечує формування прогнозу за обмеженої кількості накопичених даних. Після формування достатньої історії спостережень прогнозування виконується LSTM-моделлю, здатною враховувати складніші нелінійні залежності. Перехід між моделями може визначатися обсягом накопичених даних або результатами порівняння похибки прогнозування.

Прогнозоване значення навантаження передається до механізму проактивного керування ресурсами Kubernetes. За очікуваного збільшення навантаження рішення про масштабування може бути сформоване до фактичного досягнення критичного рівня використання ресурсів, що скорочує час реакції системи та зменшує ризик погіршення якості обслуговування. За прогнозованого спаду навантаження надлишкові ресурси можуть бути вивільнені.

Отже, гібридний підхід LSTM–Holt-Winters поєднує переваги статистичного та нейромережевого прогнозування і забезпечує можливість адаптації механізму керування до обсягу доступних історичних даних. Його використання створює основу для проактивного масштабування контейнеризованих сервісів, спрямованого на підтримання необхідної продуктивності та раціональне використання обчислювальних ресурсів Kubernetes-кластера.

Для реалізації прогнозного керування ресурсами використовується архітектура проактивного автомасштабування вебзастосунків у Kubernetes, заснована на прогнозуванні часових рядів [62]. Структурно-функціональна схема проактивного автомасштабування вебзастосунків у Kubernetes на основі прогнозування навантаження показано на рис. 3.19.

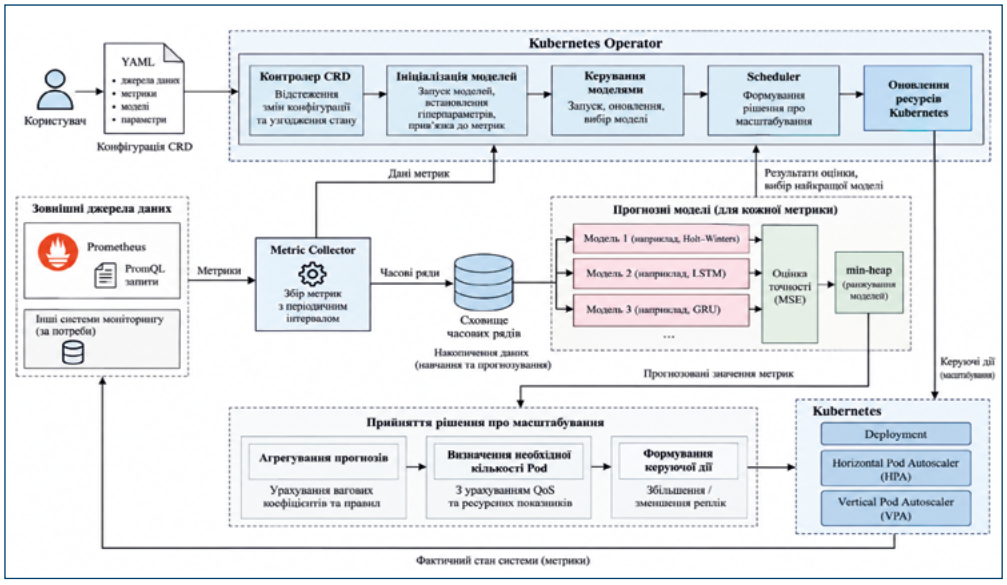


Рис. 3.19. Структурно-функціональна схема проактивного автомасштабування вебзастосунків у Kubernetes на основі прогнозування навантаження

Її програмна реалізація передбачає використання Custom Resource Definition (CRD) разом зі спеціалізованим Kubernetes Operator. Такий підхід розширює можливості традиційних засобів автомасштабування завдяки переходу від реакції на перевищення встановлених порогових значень до прогнозного керування ресурсами з урахуванням показників якості обслуговування.

CRD виконує роль декларативного інтерфейсу конфігурування системи. За допомогою YAML-конфігурації визначаються джерела даних, набір контрольованих метрик, зокрема завантаження CPU, RTT, показники QoE та інтенсивність сервісних запитів (Service Request), а також моделі прогнозування, призначені для відповідних метрик [63]. Важливою особливістю такого підходу є можливість асоціювати з однією ме-

трикою декілька прогнозних моделей. Завдяки цьому система не обмежується використанням наперед заданого алгоритму, а може постійно оцінювати точність альтернативних моделей і вибирати ту, яка краще відповідає поточному характеру навантаження.

Загальна організація запропонованого механізму відповідає замкненому контуру керування з урахуванням QoS і включає процеси моніторингу, прийняття рішень та виконання керуючих дій, координовані Kubernetes Operator.

Після створення або зміни об'єкта CRD Kubernetes Operator запускає процес узгодження фактичного стану системи з описаною конфігурацією. У межах цього процесу аналізуються специфікації метрик і моделей прогнозування, після чого формується необхідний стан компонентів системи. Operator ініціалізує визначені моделі, встановлює їхні гіперпараметри та логічно пов'язує кожну модель з відповідною контрольованою метрикою.

Для забезпечення можливості використання різнорідних реалізацій прогнозних моделей передбачено уніфікований інтерфейс їх виклику. Взаємодія може здійснюватися з використанням TCP або Unix Domain Socket. Завдяки такій організації прогнозні моделі можуть функціонувати як незалежні програмні сервіси, залишаючись при цьому складовими єдиного контуру керування, координованого Kubernetes Operator.

Збирання контрольованих показників виконується централізованим компонентом Metric Collector, параметри якого задаються у CRD. Відповідно до встановленого інтервалу цей компонент звертається до зовнішніх систем моніторингу, зокрема Prometheus, використовуючи визначені користувачем PromQL-запити. Отримані значення накопичуються у вигляді часових рядів та надалі використовуються прогнозними моделями як під час навчання, так і в процесі формування прогнозів у режимі реального часу.

Централізований збір метрик забезпечує синхронізований доступ усіх моделей до однакових вхідних даних. Водночас усувається необхідність незалежного звернення кожної прогнозної моделі до системи моніторингу, що дає змогу уникнути дублювання запитів і забезпечити узгодженість даних, які використовуються для порівняння результатів прогнозування.

Для кожної контрольованої метрики паралельно оцінюється декілька моделей-кандидатів. Їх ефективність визначається на основі похибки прогнозування, наприклад середньоквадратичної помилки (Mean Squared Error, MSE), розрахованої між прогнозованими та фактичними значеннями. Для впорядкування моделей за величиною похибки використовується структура даних min-heap, яка забезпечує безпосередній доступ до моделі з найменшим поточним значенням помилки для відповідної метрики.

Після накопичення достатнього обсягу історичних спостережень прогнозні моделі проходять початковий етап оцінювання. На цьому етапі вони формують прогнози лише в режимі моделювання, тобто отримані результати ще не використовуються для виконання реальних операцій масштабування. Такі прогнози необхідні для визначення початкової точності моделей і формування їхнього первинного ранжування у відповідних структурах min-heap.

Під час подальшого функціонування системи спеціалізована корутина оновлення періодично виконує повторне навчання прогнозних моделей та коригує їх положення в min-heap відповідно до поточних значень похибки. Такий механізм забезпечує адаптацію до зміни характеру навантаження та можливих змін статистичних властивостей вхідних даних. У результаті модель, яка була найточнішою на початковому етапі, не обов'язково залишається пріоритетною протягом усього часу функціонування системи.

Основна корутина Scheduler функціонує у синхронізованих часових інтервалах, узгоджених із горизонтом прогнозування моделей. Під час кожного циклу прогнозування для кожної метрики вибирається модель, яка на поточний момент займає найвищу позицію за точністю. На її основі визначається прогнозоване значення відповідного показника для наступного інтервалу керування. Отримані прогнози перетворюються на сигнали масштабування відповідно до визначених користувачем вагових коефіцієнтів та правил агрегування.

Прогнозовані значення контрольованих метрик надалі об'єднуються у комплексне рішення щодо масштабування, на основі якого визначається необхідна кількість реплік Pod для наступного інтервалу керування. Kubernetes Operator реалізує сформоване рішення шляхом оновлення

відповідних об'єктів Kubernetes. Таким чином, зміна кількості ресурсів відбувається на основі очікуваних значень QoS та ресурсних показників, а не лише після фактичного перевищення встановлених порогів.

Після виконання операції масштабування Scheduler переходить у стан очікування до наступної синхронізованої точки прогнозування, після чого цикл збирання та аналізу даних, вибору прогнозної моделі й формування керуючого рішення повторюється. Така організація формує замкнений контур проактивного керування, у якому результати роботи масштабованого застосунку знову відображаються у контрольованих метриках і використовуються під час наступних циклів прогнозування.

Використання ієрархічного механізму керування прогнозними моделями на основі min-heap у поєднанні з корутинною організацією планування забезпечує адаптивне прогнозне автомасштабування ресурсів. Структура min-heap дає змогу оперативно отримувати модель із найменшою поточною похибкою для кожної метрики, тоді як періодичне перенавчання забезпечує пристосування прогнозного механізму до зміни характеристик навантаження. У результаті запропонований підхід орієнтований на використання в масштабованих Kubernetes-середовищах, для яких важливими є підтримання показників QoS, адаптація до динамічного навантаження та завчасне формування рішень щодо необхідної кількості обчислювальних ресурсів в РІКС.

3.5. Підготовка часових даних та формування вибірок для навчання LSTM-моделі

Одним із ключових етапів побудови системи прогнозного керування ресурсами є підготовка даних для навчання моделі. Якість та спосіб формування вибірок безпосередньо впливають на точність прогнозування навантаження і, відповідно, на ефективність прийняття рішень щодо проактивного масштабування ресурсів у Kubernetes-середовищі.

У межах дослідження використано часовий ряд погодинної кількості запитів до вебсервісу протягом п'яти тижнів. Такий часовий інтервал дає змогу дослідити характер зміни навантаження та враху-

вати добову і тижневу періодичність активності користувачів. Основні етапи підготовки часових даних для подальшого навчання LSTM-моделі наведено на рис. 3.20.



Рис. 3.20. Послідовність підготовки часових даних для навчання LSTM-моделі

На початковому етапі виконується завантаження та попередній аналіз вихідних даних. Досліджуваний часовий ряд містить один основний показник – кількість запитів до сервісу протягом кожної години. Для визначення характерних закономірностей зміни навантаження здійснюється візуальний аналіз даних на різних часових інтервалах.

Добову динаміку навантаження наведено на рис. 3.21. Вона характеризується зростанням кількості запитів у періоди підвищеної активності користувачів та її зменшенням у нічний час.

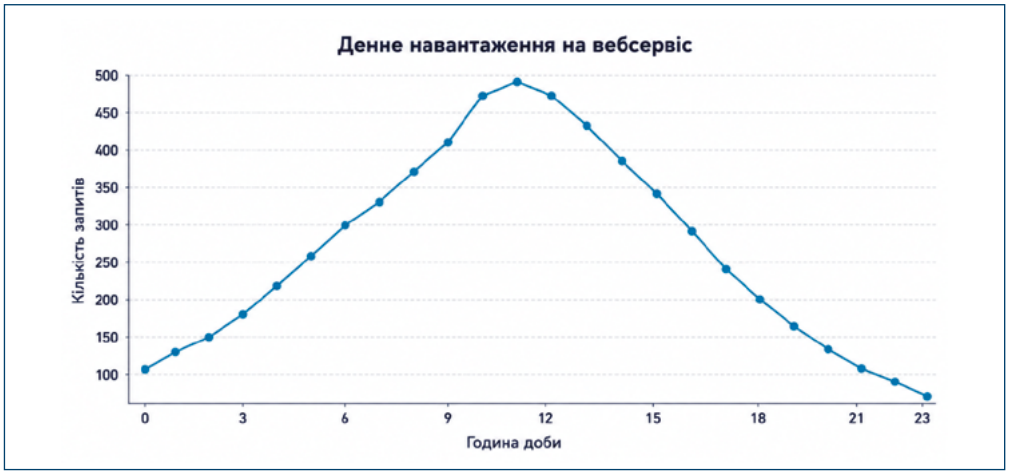


Рис. 3.21. Добова динаміка кількості запитів до вебсервісу

Для оцінювання повторюваності навантаження протягом тижня проаналізовано тижневий часовий інтервал (рис. 3.22). Отримані дані демонструють повторюваність добових профілів та зменшення інтенсивності запитів у вихідні дні.



Рис. 3.22. Тижнева динаміка навантаження вебсервісу

Зміну навантаження за весь п'ятитижневий період спостереження наведено на рис. 3.23. Часовий ряд характеризується повторюваними тижневими циклами та поступовим збільшенням кількості запитів наприкінці досліджуваного періоду.



Рис. 3.23. Динаміка навантаження вебсервісу протягом п'яти тижнів

Перед формуванням вхідних послідовностей виконується стандартизація значень за допомогою StandardScaler. У цьому випадку дані перетворюються до нульового середнього та одиничного стандартного відхилення, що сприяє стабільнішому навчанню нейронної мережі. Після попередньої обробки часовий ряд розділяється на навчальну та тестову частини зі збереженням хронологічного порядку спостережень. Для навчання моделі використовуються дані перших чотирьох тижнів, тоді як останній тиждень відводиться для оцінювання якості прогнозування. Такий поділ виключає випадкове перемішування часових спостережень і відповідає реальному сценарію, за якого прогноз майбутнього навантаження формується на основі попередньо накопичених даних. Принцип формування навчальної та тестової вибірок наведено на рис. 3.24.

Для побудови вхідних послідовностей застосовується механізм ковзного вікна (lookback). Розмір вікна становить 168 годин, що відповідає одному тижню спостережень. Кожна вхідна послідовність містить значення навантаження за попередні 168 годин, а цільовим є значення для наступного часового кроку. Такий підхід дає змогу враховувати добову та тижневу періодичність досліджуваного часового ряду.

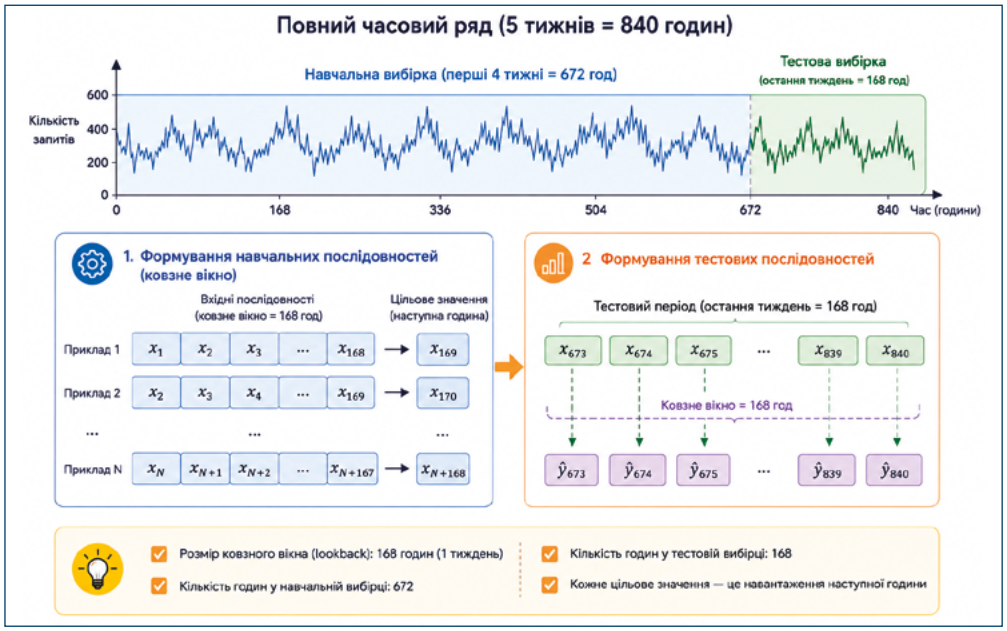


Рис. 3.24. Формування навчальної та тестової вибірок із використанням ковзного вікна

На основі сформованих послідовностей будується LSTM-модель прогнозування навантаження. Її архітектура містить LSTM-шар із 256 нейронами та вихідний повнозв'язний шар (Dense), який формує прогнозоване значення навантаження. Структуру досліджуваної моделі наведено на рис. 3.25.

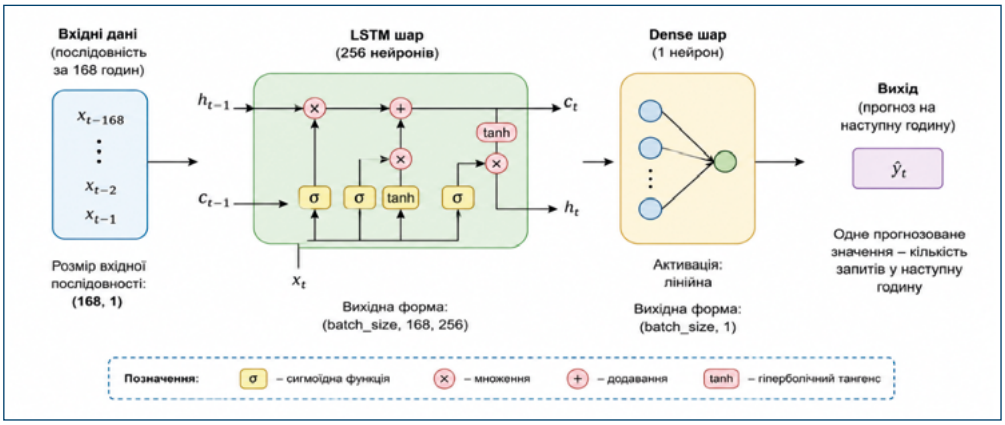


Рис. 3.25. Структура LSTM-моделі для прогнозування навантаження вебсервісу

Для навчання LSTM-моделі використовується оптимізатор Adam, а як функція втрат – середньоквадратична помилка (MSE). Зменшення значення функції втрат у процесі навчання характеризує збіжність моделі та покращення відповідності між прогнозованими і фактичними значеннями навчальної вибірки.

Після завершення навчання модель перевіряється на тестовій вибірці. Тестові послідовності формуються за тим самим принципом ковзного вікна, після чого отримані прогнозовані значення зіставляються з фактичними даними. Результати такого порівняння наведено на рис. 3.26.

Отримані результати показують, що LSTM-модель відтворює основну динаміку часового ряду та прогнозує характер зміни навантаження на тестовому інтервалі. Кількісне оцінювання точності при цьому доцільно здійснювати за відповідними метриками похибки прогнозування.

На завершальному етапі виконується прогнозування навантаження на наступний тиждень. Для цього застосовується рекурсивний багатокроковий підхід: кожне отримане прогнозоване значення додається до вхідної послідовності та використовується для формування прогнозу на

наступному часовому кроці. Такий спосіб дає змогу сформувати прогноз на весь заданий часовий горизонт, хоча зі збільшенням його тривалості можливе накопичення похибки.

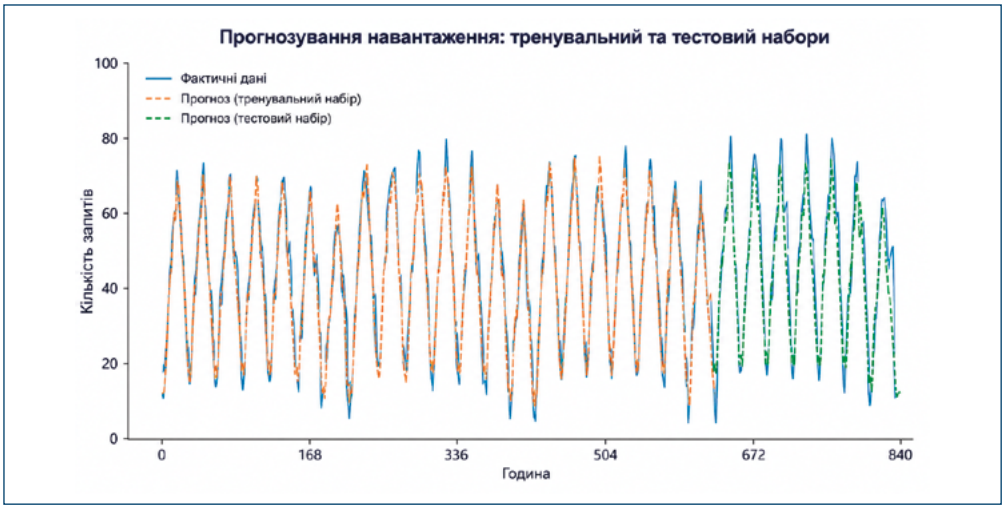


Рис. 3.26. Порівняння фактичного та прогнозованого LSTM-моделлю навантаження

Результати прогнозування навантаження на наступний тиждень наведено на рис. 3.27. Сформований прогноз відображає характерну циклічність часового ряду та продовжує виявлену за історичними даними динаміку навантаження.



Рис. 3.27. Прогноз динаміки навантаження вебсервісу на наступний тиждень

Проведене дослідження показує можливість використання сформованої LSTM-моделі для прогнозування динаміки навантаження контейнеризованого сервісу. Отримані прогнозні значення можуть бути використані як вхідна інформація для механізму проактивного масштабування, який завчасно змінює обсяг доступних ресурсів відповідно до очікуваного стану навантаження в Kubernetes-кластері.

3.6. Дослідження ефективності проактивного автомасштабування обчислювальних ресурсів

Для оцінювання ефективності механізму проактивного автомасштабування ресурсів у середовищі Kubernetes проведено експериментальне дослідження, в основу якого покладено прогнозування часових рядів навантаження. Основною метою експерименту є порівняння запропонованого проактивного підходу з традиційним реактивним масштабуванням, а також визначення здатності прогнозної моделі зменшувати перевантаження сервісу за умов динамічної зміни інтенсивності запитів.

Для проведення дослідження розгорнуто Kubernetes-кластер, що складався з одного керуючого вузла (control-plane node) та трьох робочих вузлів (worker nodes). Усі вузли мали однакові обчислювальні характеристики: два процесорні ядра та 2 ГБ оперативної пам'яті. Така конфігурація забезпечила збалансоване й контрольоване експериментальне середовище та дала змогу досліджувати особливості роботи механізмів масштабування без впливу надлишкових обчислювальних ресурсів. Кластер функціонував під керуванням Ubuntu 23.04 із використанням Kubernetes версії 1.26.3 та Docker 23.0.6 як середовища виконання контейнерів.

У кластері було розгорнуто тестовий вебсервіс у вигляді Kubernetes Deployment, доступ до якого організовано за допомогою ресурсів Service та Ingress. Вебсервіс реалізовував RESTful API для оброблення HTTP-запитів і був спеціально налаштований таким чином, щоб у разі перевищення попередньо визначеного порогу пропускної здатності повертати серверні помилки. Це дало змогу однозначно фіксувати моменти виникнення перевантаження. Рівень помилок сервісу (Service Error Rate) визначався як частка невдало оброблених запитів протягом заданого часового інтервалу та використовувався як безпосередній показник перевантаження сервісу.

В експериментах помилковими вважалися відповіді з HTTP-кодами класу 5xx. Відповідно, менше значення Service Error Rate свідчить про здатність механізму автомасштабування завчасно забезпечити необхідний обсяг ресурсів і запобігти деградації сервісу під час різкого збільшення трафіку. Для збирання та зберігання метрик використовувався Prometheus версії 2.47, за допомогою якого реєструвалися інтенсивність запитів, кількість активних Pod, рівень помилок та опосередковані показники QoS. Загальну структуру експериментального середовища для дослідження проактивного автомасштабування наведено на рис. 3.28.

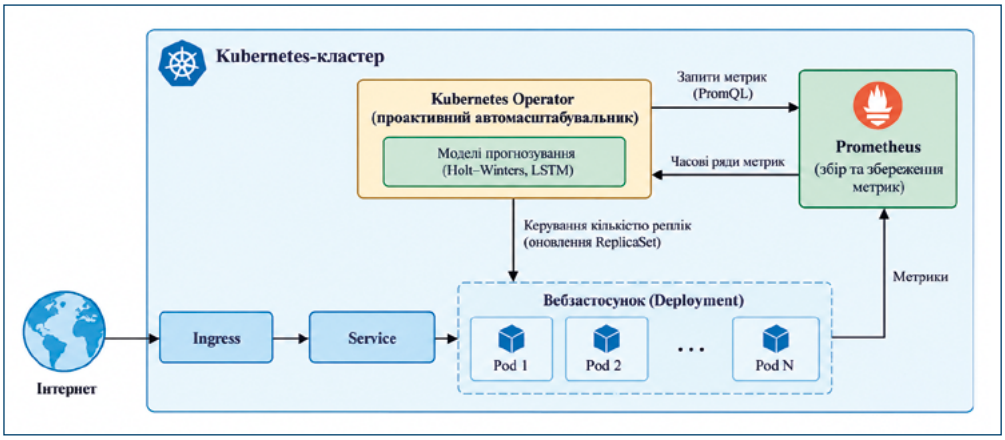


Рис. 3.28. Структурна схема експериментального середовища для дослідження проактивного автомасштабування вебзастосунку в Kubernetes [62]

Експеримент передбачав два окремі запуски з використанням однакового профілю вхідного навантаження. Під час першого запуску застосовувався стандартний реактивний підхід, за якого кількість Pod змінювалася лише після перевищення встановлених порогових значень контрольованих метрик. Під час другого запуску використовувався проактивний механізм масштабування, реалізований на основі Kubernetes Operator. У цьому випадку рішення щодо зміни кількості Pod приймалося завчасно на основі прогнозу часового ряду навантаження з урахуванням затримки, необхідної для ініціалізації нових Pod.

Профіль навантаження сформовано таким чином, щоб відтворити характерні режими функціонування реального вебзастосунку. Він включав початковий стабільний період із низькою інтенсивністю запитів, етап поступового збільшення навантаження, короткочасний різкий сплеск

трафіку типу flash crowd, період тривалого високого навантаження, подальше швидке зменшення інтенсивності запитів та завершальну фазу з випадковими коливаннями. В експерименті передбачалося, що кожен Pod здатний обробляти фіксовану кількість запитів за одиницю часу. Тому гранична пропускна здатність сервісу динамічно змінювалася залежно від кількості активних Pod, а стан перевантаження виникав у разі перевищення вхідним навантаженням їхньої сукупної обчислювальної спроможності. Сформований сценарій дав змогу дослідити поведінку механізмів масштабування як за нормального режиму функціонування, так і за критичних пікових навантажень.

Експеримент з автомасштабування проводився з часовою дискретністю в одну хвилину, що дало змогу відобразити затримки реактивного масштабування на відповідному часовому масштабі. Оскільки вихідний набір даних не містив вимірювань із хвилинною дискретністю, погодинний прогноз навантаження використовувався як базова оцінка, після чого його деталізовано до хвилинного рівня шляхом рівномірного розподілу кількості запитів у межах кожної години та внесення незначних коливань. Такий спосіб дав змогу узгоджено поєднати реальні доступні дані, використані для навчання прогнозної моделі, з аналізом роботи механізму автомасштабування Kubernetes, зберігши загальні тенденції зміни навантаження. Результати експериментального дослідження дають змогу простежити суттєві відмінності між реактивним і проактивним способами керування ресурсами. Порівняння фактичного навантаження з виділеною обчислювальною потужністю показало, що за реактивного масштабування збільшення доступних ресурсів відбувається із запізненням відносно зростання інтенсивності запитів. Унаслідок цього виникають часові інтервали, протягом яких фактичне навантаження перевищує доступну пропускну здатність сервісу.

За проактивного підходу ресурси виділяються завчасно відповідно до прогнозованої динаміки навантаження, що істотно скорочує або повністю усуває такі інтервали перевантаження. Відповідні результати наведено на рис. 3.29.

Динаміку зміни кількості активних Pod протягом експерименту наведено на рис. 3.30. У разі застосування проактивного механізму рішення про масштабування формується на основі прогнозу наванта-

ження для наступного часового інтервалу. Це дає змогу завчасно підготувати необхідну кількість Pod до моменту фактичного збільшення трафіку. За реактивного підходу нові Pod додаються лише після виявлення перевищення встановлених порогових значень, що проявляється у запізненні зміни кількості активних реплік відносно динаміки навантаження.

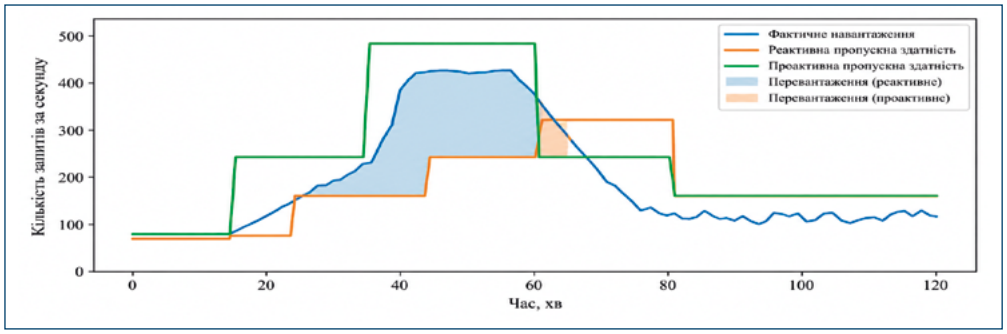


Рис. 3.29. Порівняння фактичного навантаження та виділеної обчислювальної потужності за реактивного і проактивного автомасштабування

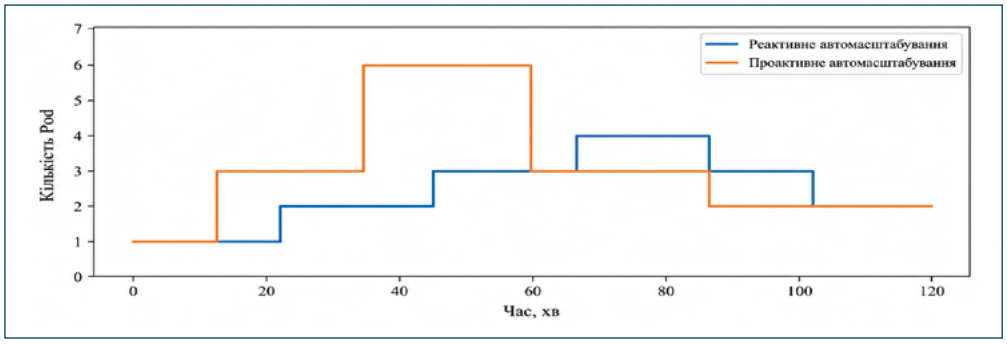


Рис. 3.30. Динаміка кількості активних Pod за реактивного та проактивного автомасштабування

Окрему увагу приділено аналізу показника Service Error Rate, який характеризує частку запитів, що не були успішно оброблені вебсервісом. Динаміку цього показника протягом експерименту наведено на рис. 3.31. За реактивного масштабування під час періодів високого навантаження спостерігаються триваліші інтервали підвищеного рівня помилок. Натомість проактивний механізм характеризується суттєво нижчим рівнем помилок, оскільки додаткові ресурси виділяються до моменту виникнення критичного перевантаження.

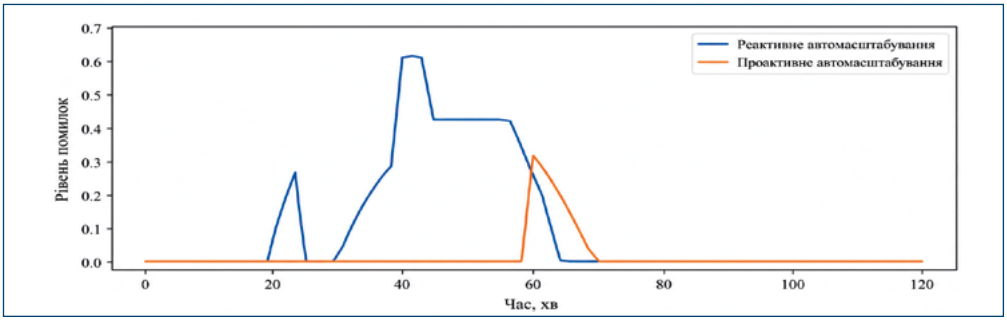


Рис. 3.31. Динаміка рівня помилок сервісу за реактивного та проактивного автомасштабування

Для комплекснішого оцінювання впливу перевантаження проаналізовано накопичення порушень SLA у часі. На рис. 3.32 наведено відповідні залежності для обох досліджуваних механізмів автомасштабування. По горизонтальній осі відображено тривалість експерименту в хвилинах, а по вертикальній – накопичене перевантаження сервісу, представлене загальною кількістю запитів, які не були оброблені через недостатню кількість доступних ресурсів [62].

Для реактивного автомасштабування крива накопичених порушень SLA різко зростає приблизно між 35-ю та 60-ю хвилинами експерименту. Це свідчить про те, що протягом зазначеного інтервалу фактичне навантаження систематично перевищувало доступну пропускну здатність сервісу. Незважаючи на виявлення механізмом реактивного масштабування зростання навантаження, додаткові Pod ставали доступними із затримкою, пов'язаною зі збиранням метрик, формуванням керуючого рішення та запуском контейнерів. Унаслідок цього сервіс протягом декількох часових інтервалів функціонував у стані перевантаження, а кожний такий інтервал збільшував сумарну кількість порушень SLA. Приблизно після 60-ї хвилини крива переходить у плато, що свідчить про завершення масштабування та припинення подальшого накопичення перевантажень. Проте негативний ефект, накопичений протягом попереднього періоду, залишається суттєвим.

Для проактивного автомасштабування спостерігається інший характер зміни накопичених порушень SLA. На початковій частині експерименту їх кількість залишається практично незмінною, оскільки необхідні ресурси виділяються завчасно відповідно до прогнозу наван-

таження. Лише приблизно на 60-й хвилині спостерігається незначне зростання накопиченого показника, що може відповідати короткочасному або частковому перевищенню фактичним навантаженням прогнозованого рівня. Надалі крива знову переходить у горизонтальну ділянку. Такий характер залежності свідчить, що протягом більшої частини експерименту сервіс функціонує в межах доступної обчислювальної потужності, а випадки перевантаження є короткочасними та незначними.

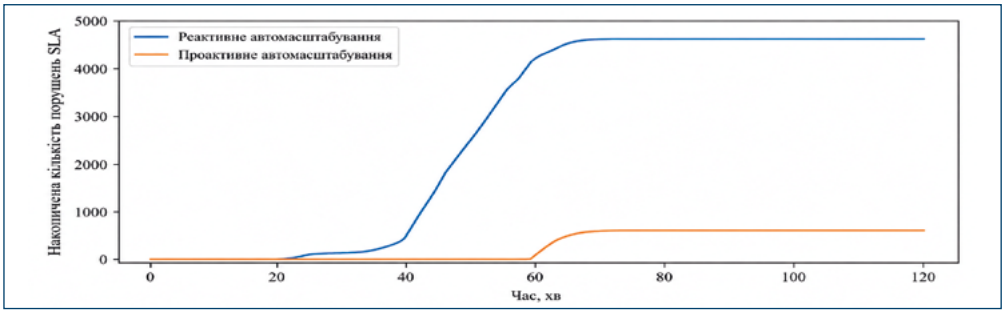


Рис. 3.32. Накопичення порушень SLA за реактивного та проактивного автомасштабування

Узагальнене порівняння загальної кількості порушень SLA протягом усього експерименту наведено на рис. 3.33. Значення для реактивного автомасштабування відповідає кінцевому рівню накопиченої кривої та характеризує значну кількість необроблених запитів, що виникла внаслідок запізнілого виділення додаткових ресурсів. Для проактивного підходу відповідний показник є у декілька разів нижчим, що безпосередньо свідчить про суттєве зменшення сумарного обсягу порушень SLA. Отримана відмінність показує, що використання прогнозування навантаження дає змогу компенсувати затримки розгортання Pod у Kubernetes та зменшити негативний вплив перевантаження на показники QoS.

У практичних Kubernetes-середовищах реактивне автомасштабування не може миттєво реагувати на зміну навантаження через сукупність технологічних затримок [30–32]. Зростання інтенсивності запитів виявляється із певним запізненням, оскільки метрики збираються й агрегуються дискретно, після чого контролер HPA періодично перевіряє виконання умов масштабування. Навіть після прийняття відповідного рішення новому Pod потрібен додатковий час для планування його розміщення, отримання образу контейнера, запуску застосунку та

проходження перевірки готовності перед початком оброблення запитів. Крім того, механізми стабілізації HPA навмисно обмежують швидкість зміни кількості реплік для запобігання надто частим коливанням масштабу системи.

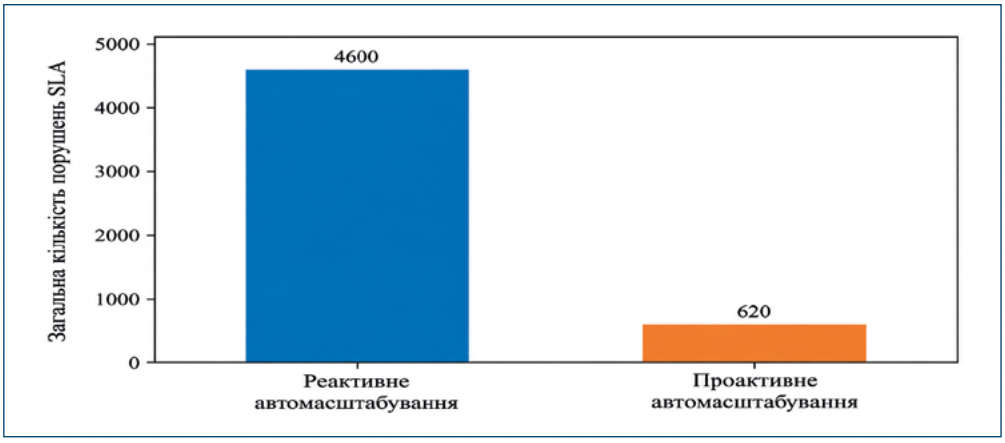


Рис. 3.33. Порівняння загальної кількості порушень SLA для реактивного та проактивного автомасштабування

Унаслідок сукупної дії зазначених чинників часовий інтервал між фактичним збільшенням навантаження та введенням у роботу повністю готових Pod у реальних кластерах може становити від десятків секунд до декількох хвилин. За різких сплесків трафіку така затримка створює умови для тимчасового перевантаження сервісу. Проактивний механізм частково компенсує цю особливість завдяки тому, що рішення щодо виділення ресурсів формується не після фактичного виникнення перевантаження, а на основі прогнозованої зміни навантаження.

Проведене експериментальне дослідження дає змогу оцінити не лише факт виникнення перевантаження, але і його накопичувальний вплив протягом часу функціонування сервісу. Реактивне масштабування призводить до суттєвого накопичення порушень SLA навіть за відносно коротких інтервалів пікового навантаження. Натомість застосування проактивного автомасштабування на основі прогнозування часових рядів істотно зменшує цей ефект і забезпечує стабільніше функціонування вебсервісу протягом експерименту.

РОЗДІЛ IV

МОДЕЛІ ТА МЕТОДИ АДАПТИВНОГО УПРАВЛІННЯ НАВАНТАЖЕННЯМ У РОЗПОДІЛЕНОМУ EDGE-CLOUD СЕРЕДОВИЩІ

4.1. Архітектурні моделі Edge-Cloud систем розподілу обчислень та офлоадингу

Хмарні обчислення (Cloud Computing) є одним із базових технологічних підходів до побудови розподілених інформаційно-комунікаційних систем, що передбачає надання обчислювальних ресурсів, засобів зберігання даних і прикладних сервісів через мережеву інфраструктуру. Використання хмарної моделі забезпечує динамічне виділення ресурсів відповідно до поточних потреб користувачів і сервісів, їх масштабування та спільне використання в межах розподіленого середовища. Це дає змогу зменшити залежність від локальної апаратно-програмної інфраструктури та підвищити ефективність використання доступних обчислювальних ресурсів.

Подальший розвиток розподіленої моделі надання обчислювальних ресурсів пов'язаний із концепцією периферійних хмарних обчислень (Edge-Cloud), яка передбачає розміщення частини обчислювальних потужностей і сервісів на периферії мережі, у безпосередній близькості до кінцевих пристроїв. Така організація дає змогу виконувати оброблення даних поблизу джерел їх формування, зменшуючи обсяг передавання інформації до віддалених центрів обробки даних та скорочуючи затримки під час надання сервісів [63].

Поєднання централізованих хмарних і периферійних обчислювальних ресурсів формує багаторівневе розподілене середовище, у якому функції оброблення, зберігання та передавання даних можуть розподілятися між різними рівнями інфраструктури. Edge-Cloud при цьому не замінює традиційну хмарну модель, а доповнює її, формуючи обчислювальний континуум між централізованими центрами обробки даних і користувацьким обладнанням (User Equipment, UE) (рис. 4.1). Розміщення обчислювальних ресурсів ближче до UE забезпечує зменшення затримки та підвищення оперативності оброблення даних, тоді як цен-

тралізована хмарна інфраструктура надає значно більші обчислювальні потужності та можливості для виконання ресурсомістких завдань [64]. Така багаторівнева організація створює основу для побудови розподілених інформаційно-комунікаційних систем із гнучким розподілом обчислювальних, мережевих і сховищних ресурсів відповідно до вимог сервісів та поточного стану інфраструктури.

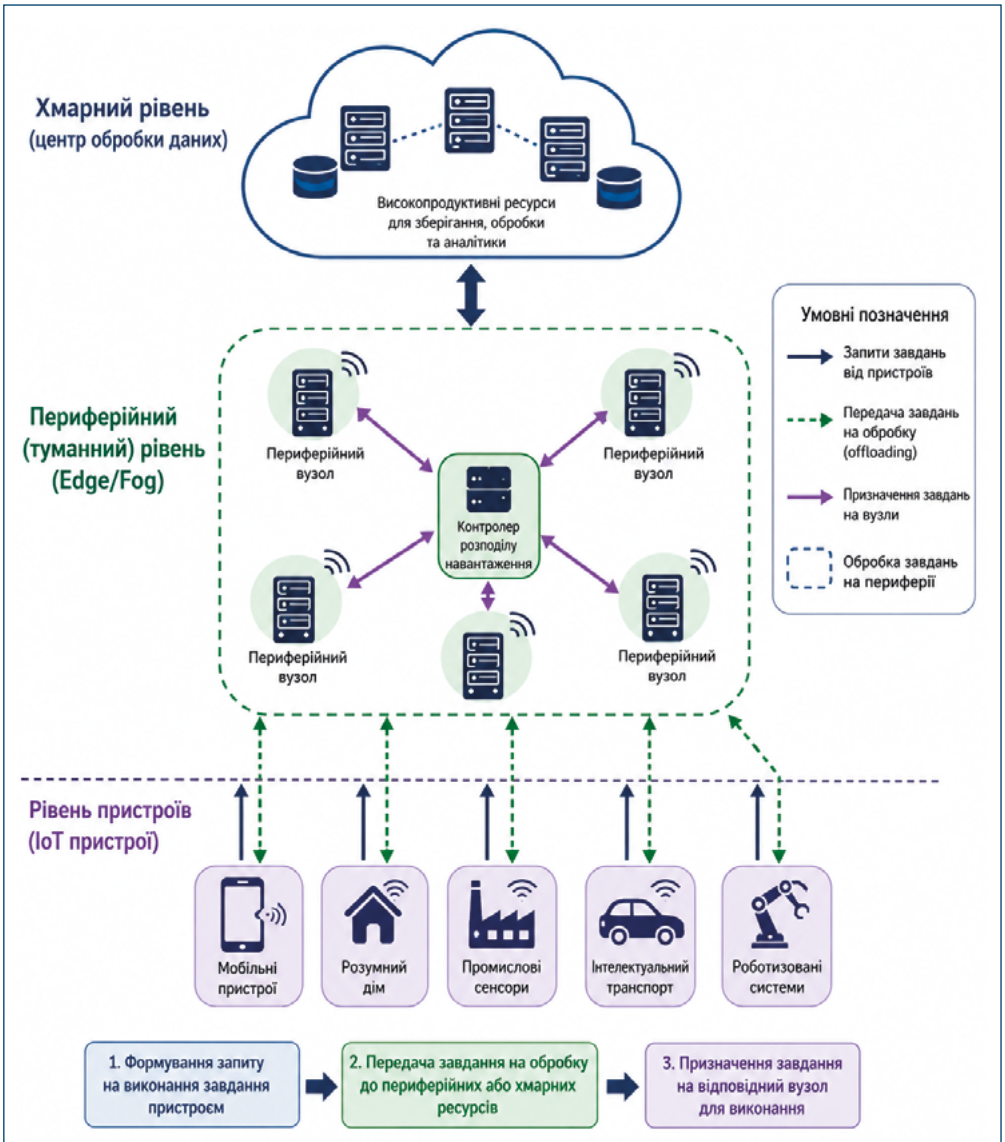


Рис. 4.1. Edge-Cloud архітектура

Офлоадинг (offloading) є одним із ключових механізмів організації обчислювального процесу в розподілених інформаційно-комунікаційних системах і передбачає передавання обчислювальних завдань між різними рівнями інфраструктури залежно від їх складності, доступності ресурсів та вимог до якості обслуговування. Якщо ресурсів кінцевого пристрою (User Equipment, UE) недостатньо для виконання певного завдання, його оброблення може бути перенесене на периферійний рівень. Незважаючи на додаткові витрати часу на передавання даних, використання продуктивніших периферійних ресурсів у багатьох випадках дає змогу скоротити загальний час виконання завдання [65]. За аналогічним принципом ресурсоємні завдання, які не можуть бути ефективно опрацьовані на периферійному рівні, можуть передаватися до централізованої хмарної інфраструктури.

Управління розподілом завдань у такому середовищі здійснюється із застосуванням механізмів балансування навантаження (Load Balancing), які забезпечують вибір обчислювального ресурсу відповідно до його поточного стану та характеристик завдання. Спільне використання механізмів офлоадингу та балансування навантаження дає змогу динамічно перерозподіляти обчислення між кінцевими пристроями, периферійними вузлами та хмарними ресурсами, запобігаючи перевантаженню окремих компонентів системи.

На рис. 4.2 наведено узагальнену багаторівневу архітектуру розподіленої інформаційно-комунікаційної системи, що охоплює рівень кінцевих пристроїв, периферійний (Edge/Fog) та хмарний рівні. На рівні пристроїв формуються запити на виконання завдань від мобільних пристроїв, компонентів розумного будинку, промислових сенсорів, інтелектуального транспорту, роботизованих систем та інших IoT-пристроїв. Залежно від обчислювальної складності завдання та доступних локальних ресурсів його виконання може здійснюватися безпосередньо на пристрої або передаватися на вищий рівень інфраструктури.

Периферійний рівень представлений сукупністю Edge/Fog-вузлів, розташованих у мережевій близькості до джерел формування даних. На цьому рівні здійснюються приймання та оброблення переданих завдань, а також їх розподіл між доступними вузлами відповідно до поточного навантаження. Розміщення обчислювальних ресурсів поблизу кінцевих

пристроїв дає змогу зменшити обсяг трафіку, що передається до централізованої хмари, та забезпечити необхідні часові характеристики для сервісів, чутливих до затримок. У разі недостатності периферійних ресурсів частина завдань може бути перенесена на хмарний рівень [66].

Хмарний рівень об'єднує високопродуктивні та масштабовані обчислювальні ресурси, призначені для виконання ресурсоємних операцій, централізованого зберігання й оброблення значних обсягів даних. Таким чином, взаємодія трьох рівнів забезпечує гнучкий розподіл обчислювального навантаження відповідно до доступності ресурсів, характеристик завдань і вимог сервісів, формуючи основу для ефективного функціонування розподілених інформаційно-комунікаційних систем.

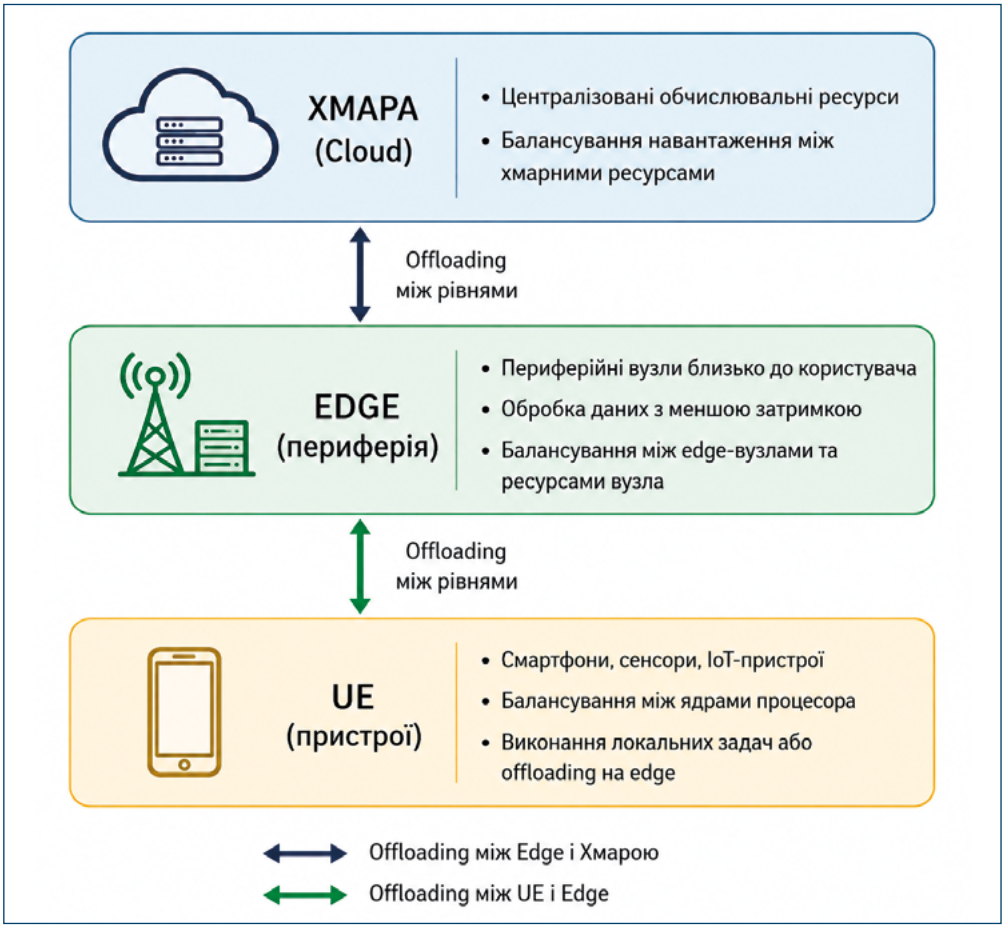


Рис. 4.2. Архітектура взаємодії Cloud–Edge–UE з використанням механізму офлоадингу

Нижній рівень архітектури формують кінцеві користувацькі пристрої (User Equipment, UE), до яких належать смартфони, сенсори, IoT-пристрої та інше обладнання, що є джерелом даних і обчислювальних завдань. Частина операцій може виконуватися безпосередньо на пристрої з використанням доступних локальних ресурсів. Балансування навантаження між обчислювальними компонентами UE сприяє підвищенню продуктивності та раціональному використанню енергетичних ресурсів. У разі недостатньої обчислювальної потужності або недоцільності локального виконання завдання передається на периферійний вузол за допомогою механізму офлоадингу.

Таким чином, взаємодія рівнів UE–Edge-Cloud забезпечує динамічний розподіл обчислювальних завдань у межах розподіленої інформаційно-комунікаційної системи. Рішення щодо місця виконання завдання може прийматися з урахуванням поточного завантаження обчислювальних ресурсів, стану мережі, складності завдання та вимог сервісу до затримки й продуктивності. Поєднання механізмів офлоадингу та балансування навантаження дає змогу адаптивно використовувати ресурси різних рівнів, запобігати їх перевантаженню та забезпечувати необхідні показники якості надання сервісів.

4.2. Методи балансування навантаження в хмарних та периферійних обчислювальних системах

Балансування навантаження є одним із ключових механізмів ефективного використання ресурсів розподілених інформаційно-комунікаційних систем. У хмарних і периферійних обчислювальних середовищах воно передбачає розподіл вхідних запитів або обчислювальних завдань між доступними екземплярами сервісів та обчислювальними вузлами з метою запобігання перевантаженню окремих компонентів і забезпечення необхідного рівня продуктивності системи.

Кожне завдання потребує певного обсягу обчислювальних, пам'ятєвих і мережевих ресурсів, зокрема процесорного часу, оперативної пам'яті та пропускної здатності каналів зв'язку. Водночас фактичні ресурсні потреби можуть змінюватися в процесі функціонування системи та не завжди можуть бути точно визначені заздалегідь. За таких

умов вибір оптимального варіанта призначення завдань доступним ресурсам перетворюється на складну задачу оптимізації, яка для багатьох постановок належить до класу NP-складних [67].

Неточність оцінювання та прогнозування потреб у ресурсах може призводити до двох протилежних станів. Надмірне виділення ресурсів (over-provisioning) виникає, коли для забезпечення сервісу резервується більше обчислювальних потужностей, ніж фактично необхідно. Такий підхід підвищує запас продуктивності, проте знижує ефективність використання інфраструктури та збільшує витрати на її експлуатацію. Недостатнє виділення ресурсів (under-provisioning), навпаки, характеризується дефіцитом ресурсів за зростання навантаження, що може спричинити збільшення часу відгуку, погіршення якості обслуговування або тимчасову недоступність сервісів [68].

Методи балансування навантаження відрізняються способом вибору ресурсу для оброблення чергового запиту. До найпростіших належать Round Robin, за якого запити послідовно розподіляються між доступними екземплярами сервісу в циклічному порядку, та Random Load Balancing, що передбачає випадковий вибір ресурсу для обслуговування запиту [69]. Перевагою таких методів є простота реалізації та невеликі обчислювальні витрати, однак вони не враховують фактичний стан ресурсів і відмінності в інтенсивності окремих завдань.

Більш досконалі методи використовують інформацію про поточний стан системи, зокрема завантаження процесора, використання оперативної пам’яті, кількість активних з’єднань, довжину черг, пропускну здатність мережі та час відгуку сервісів. Це дає змогу динамічно змінювати розподіл запитів відповідно до поточного навантаження та ефективніше використовувати доступні ресурси. Для периферійних обчислювальних систем додаткового значення набувають мережева затримка, відстань до користувача, доступність вузлів та обмеженість їхніх обчислювальних ресурсів [70].

Практичну реалізацію механізмів розподілу навантаження в сучасних хмарних середовищах доцільно розглянути на прикладі Kubernetes – платформи оркестрації контейнеризованих застосунків. У такому середовищі прикладний сервіс може бути представлений декількома екземплярами (Pod), між якими розподіляються вхідні запити. Меха-

нізми балансування у взаємодії із засобами оркестрації та масштабування дають змогу адаптувати кількість доступних екземплярів до зміни інтенсивності навантаження та підтримувати необхідний рівень доступності сервісів.

На рис. 4.3 наведено узагальнену структуру розподілу навантаження в хмарному середовищі на основі Kubernetes, яка ілюструє взаємодію вхідного потоку запитів, балансувальника навантаження та множини екземплярів сервісу.

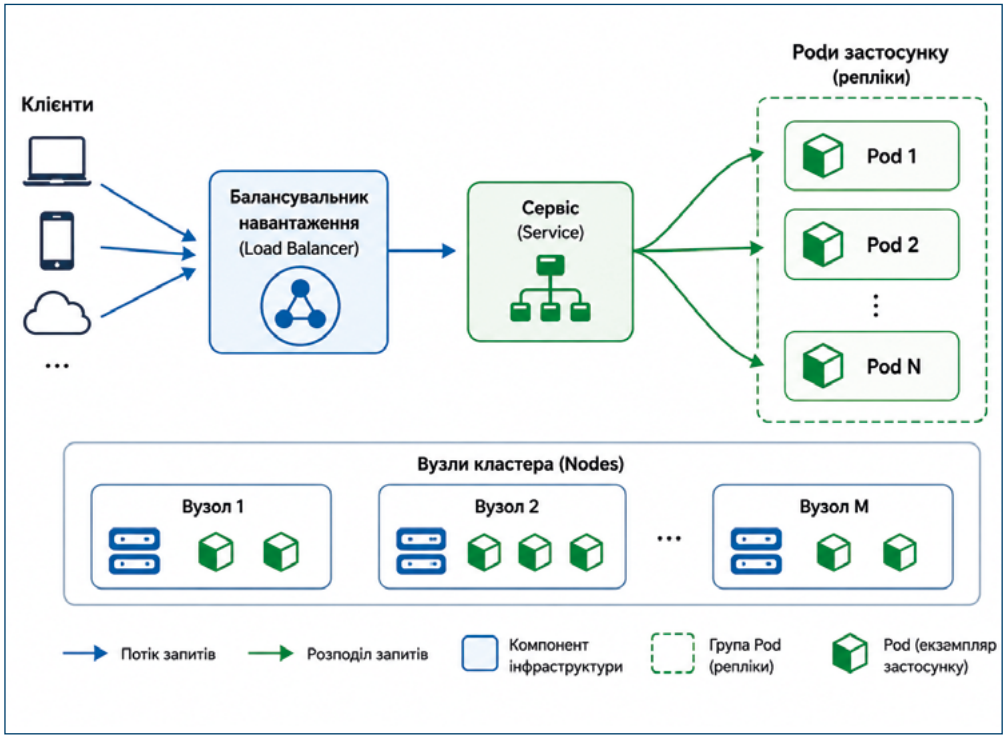


Рис. 4.3. Схема балансування навантаження між екземплярами застосунку в Kubernetes-кластері

Архітектура Kubernetes ґрунтується на принципах контейнеризації, оркестрації та автоматизованого керування обчислювальними ресурсами, що забезпечує масштабоване розгортання та підтримку працездатності застосунків у кластерному середовищі. Основною одиницею розгортання в Kubernetes є Pod, який об'єднує один або декілька контейнерів із спільними мережевими та, за потреби, сховищними ресурсами. Для забезпечення необхідної продуктивності застосунок може одночасно виконуватися у

вигляді декількох реплік Pod, розміщених на одному або різних вузлах кластера.

Логічний доступ до групи Pod забезпечується за допомогою об'єкта Service, який формує стабільну точку доступу до застосунку та забезпечує спрямування трафіку до доступних його екземплярів. Завдяки цьому клієнту не потрібно володіти інформацією про фактичне розташування окремих Pod або відстежувати зміни їхнього складу. Вхідні запити можуть розподілятися між репліками застосунку, що дає змогу уникати концентрації навантаження на одному екземплярі та підвищувати доступність сервісу.

Розміщення Pod на обчислювальних вузлах визначається планувальником Kubernetes з урахуванням доступності ресурсів, заданих вимог і встановлених політик планування. Водночас кількість реплік застосунку може динамічно змінюватися відповідно до поточного стану системи. Зокрема, механізми автоматичного горизонтального масштабування дають змогу збільшувати або зменшувати кількість Pod на основі контрольованих метрик, таких як використання процесора та пам'яті, а також інших ресурсних або прикладних показників. Це забезпечує адаптацію обчислювальної інфраструктури до зміни інтенсивності робочого навантаження [71].

У хмарних середовищах вузли Kubernetes-кластера зазвичай функціонують на фізичних або віртуалізованих обчислювальних ресурсах центрів обробки даних. Концентрація значних обчислювальних потужностей у таких центрах створює сприятливі умови для централізованого керування, моніторингу, резервування та масштабування ресурсів. Водночас територіальна віддаленість хмарної інфраструктури від кінцевих користувачів може збільшувати мережеву затримку та час доставки даних. Цей чинник є особливо важливим для застосунків реального часу, мультимедійних сервісів, систем дистанційного керування та IoT-застосунків, для яких часові характеристики передавання й оброблення даних безпосередньо впливають на якість надання сервісів.

За таких умов ефективність функціонування сервісів визначається не лише кількістю доступних обчислювальних ресурсів, а й способом розподілу навантаження між ними. Механізми Kubernetes забезпечують абстракцію доступу до множини реплік застосунку та дають змогу роз-

поділяти мережевий трафік між доступними Pod [72]. У поєднанні з автоматичним масштабуванням це створює механізм адаптації системи до динамічних змін навантаження: збільшення кількості реплік розширює доступну обчислювальну потужність, тоді як механізми балансування забезпечують розподіл вхідного потоку запитів між активними екземплярами сервісу.

Узагальнена структура організації балансування навантаження в Kubernetes-кластері відображає взаємодію між вхідним потоком запитів, сервісом та множиною Pod, які реалізують окремі екземпляри застосунку. Така організація забезпечує відокремлення логіки доступу до сервісу від фізичного розміщення його екземплярів та створює основу для динамічного масштабування і розподілу навантаження в хмарному середовищі.

У периферійних обчислювальних середовищах принципи контейнеризації та оркестрації можуть застосовуватися аналогічно до хмарної інфраструктури: прикладні сервіси розгортаються у вигляді множини екземплярів на доступних Edge-вузлах. Однак, на відміну від централізованої хмарної інфраструктури, периферійні обчислювальні ресурси є територіально розподіленими та розміщуються у безпосередній близькості до джерел формування трафіку. Така організація скорочує шлях передавання даних між користувацькими пристроями (UE) та обчислювальними ресурсами, що дає змогу зменшити мережеву затримку та забезпечити необхідні часові характеристики сервісів реального часу, промислових IoT-систем, мультимедійних застосунків, систем доповненої реальності та автономного керування [16, 17].

Водночас окремі Edge-вузли характеризуються суттєво меншими обчислювальними та сховищними ресурсами порівняно з централізованими центрами обробки даних. Це зумовлює необхідність ефективного розподілу навантаження не лише між екземплярами застосунку в межах одного вузла, а й між територіально розподіленими Edge-вузлами. У такому середовищі можна виокремити локальне балансування, за якого запити розподіляються між доступними екземплярами сервісу на одному вузлі, та міжвузловий розподіл навантаження (inter-edge offloading), що передбачає перенесення частини завдань на інший периферійний вузол із достатнім резервом ресурсів.

Якщо доступних ресурсів периферійного середовища недостатньо для виконання завдання із заданими параметрами якості обслуговування, воно може бути передане до централізованої хмарної інфраструктури. Такий механізм Edge-to-Cloud offloading доцільно застосовувати насамперед для ресурсоємних обчислень, для яких переваги використання потужніших хмарних ресурсів компенсують додаткові витрати на передавання даних. У результаті рішення про місце виконання завдання має враховувати не лише доступність обчислювальних ресурсів, а й мережеву затримку, обсяг даних, інтенсивність навантаження та вимоги конкретного сервісу.

На рис. 4.4 наведено узагальнену структуру балансування навантаження в периферійному середовищі з використанням Kubernetes. Кожен Edge-вузол може обслуговувати локальний потік запитів і розподіляти його між розгорнутими на ньому екземплярами застосунку. За зміни навантаження або дефіциту локальних ресурсів частина завдань може бути перенесена на інший доступний Edge-вузол або, за необхідності, до хмарного рівня. Таким чином, на відміну від централізованого хмарного середовища, під час прийняття рішення щодо розподілу навантаження необхідно враховувати не лише стан окремих Pod і вузлів, а й їхнє мережеве розташування та характеристики каналів зв'язку.

Децентралізований характер Edge-інфраструктури зумовлює необхідність координації локальних механізмів балансування. Вхідні потоки можуть формуватися незалежно в різних сегментах мережі, тому централізація всіх запитів в одній точці балансування може призвести до збільшення затримки, додаткового мережевого трафіку та утворення потенційного вузького місця. Локальне прийняття рішень дає змогу оперативніше реагувати на зміну навантаження, тоді як міжвузлова координація забезпечує використання вільних ресурсів сусідніх Edge-вузлів.

Отже, розподіл навантаження в Edge-середовищі доцільно розглядати як ієрархічний процес, що охоплює три послідовні рівні прийняття рішень: локальне балансування між екземплярами сервісу, перенесення завдань між Edge-вузлами та передавання ресурсоємних завдань до централізованої хмари. Вибір відповідного рівня оброблення визначається поточним станом обчислювальних і мережевих ресурсів та вимогами сервісу. Такий підхід створює основу для адаптивного керування ресур-

сами розподілених інформаційно-комунікаційних систем за умов динамічного навантаження.

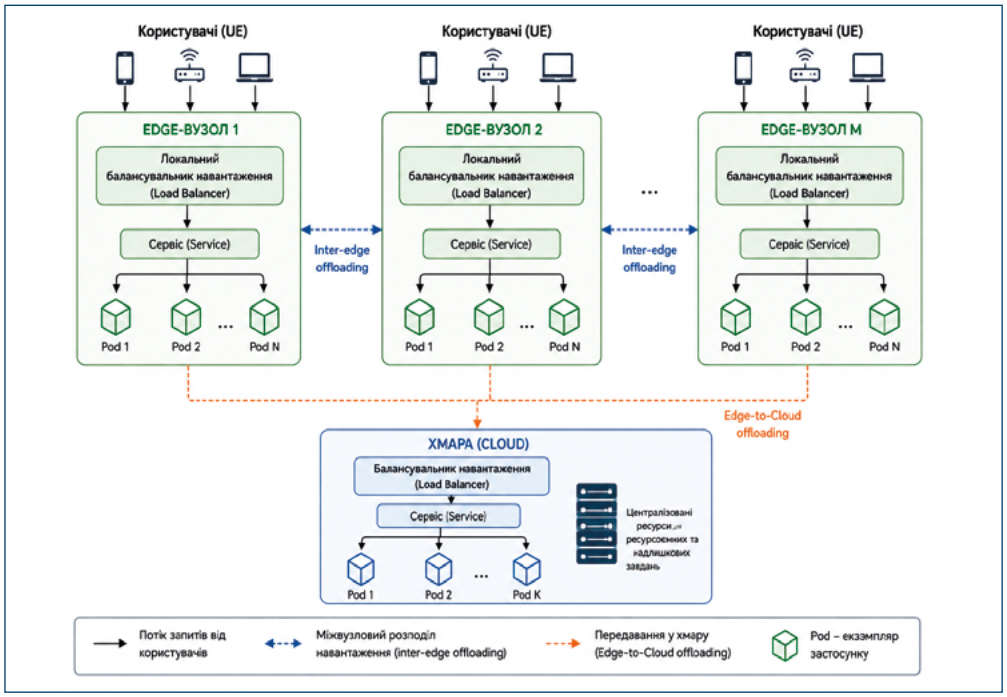


Рис. 4.4. Структурна схема балансування навантаження в периферійному середовищі на основі Kubernetes

4.3. Структурно-функціональна модель досліджуваної Edge-Cloud системи

Для дослідження методів розподілу навантаження сформовано структурно-функціональну модель Edge-Cloud системи, що відтворює взаємодію територіально розподілених периферійних обчислювальних вузлів за умов динамічного надходження користувацьких запитів. Узагальнену структуру досліджуваного середовища наведено на рис. 4.5.

Вхідне навантаження формується потоками запитів від користувацьких пристроїв (User Equipment, UE) та розподіляється між доступними Edge-вузлами. Кожен вузол забезпечує локальне оброблення запитів і реалізує механізми керування навантаженням відповідно до поточного стану обчислювальних ресурсів. Водночас між окремими вузлами існу-

ють мережеві затримки, тому перенесення завдання з одного Edge-вузла на інший супроводжується додатковими часовими витратами. Ця особливість враховується під час оцінювання доцільності міжвузлового перенесення навантаження.

Модель також передбачає можливість надходження зовнішніх запитів, які формуються іншими компонентами розподіленого середовища та не належать до основного потоку експериментального генератора навантаження. Наявність локального та зовнішнього навантаження дає змогу відтворити динамічні умови функціонування Edge-системи та дослідити поведінку алгоритмів за різного ступеня завантаження її вузлів. Архітектурні принципи побудови експериментального середовища сформовано з урахуванням підходу Resource Aware Proxy (RAP) [73], адаптованого для порівняльного дослідження різних механізмів балансування навантаження.

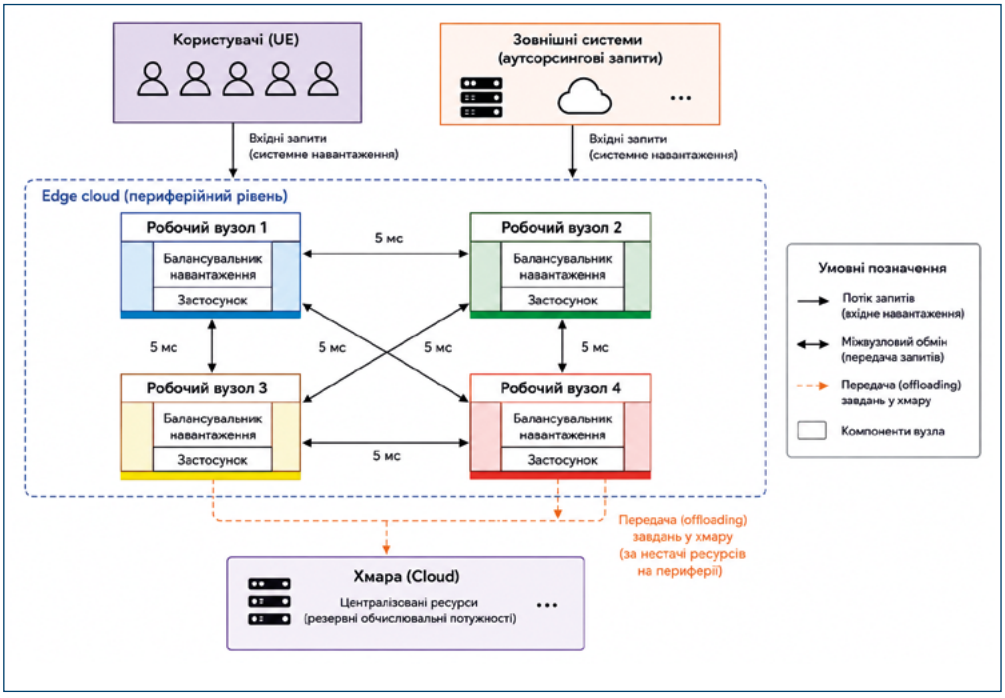


Рис. 4.5. Структурно-функціональна модель досліджуваної Edge-Cloud системи

Кожен Edge-вузол реалізовано як автономний обчислювальний компонент із власними засобами оброблення запитів, моніторингу стану

ресурсів і керування навантаженням. Деталізовану структурно-функціональну схему такого вузла наведено на рис. 4.6.

У межах Kubernetes Edge-вузол містить Pod, до складу якого входять два функціонально взаємопов'язані контейнери. Перший контейнер реалізує прикладний сервіс і безпосередньо обробляє вхідні запити. Другий містить проксі-компонент, який здійснює моніторинг стану застосунку та реалізує логіку прийняття рішень щодо розподілу навантаження. Обмін поточними показниками між компонентами здійснюється через спільний том (shared volume), до якого застосунок періодично записує інформацію про використання ресурсів і параметри свого поточного стану.

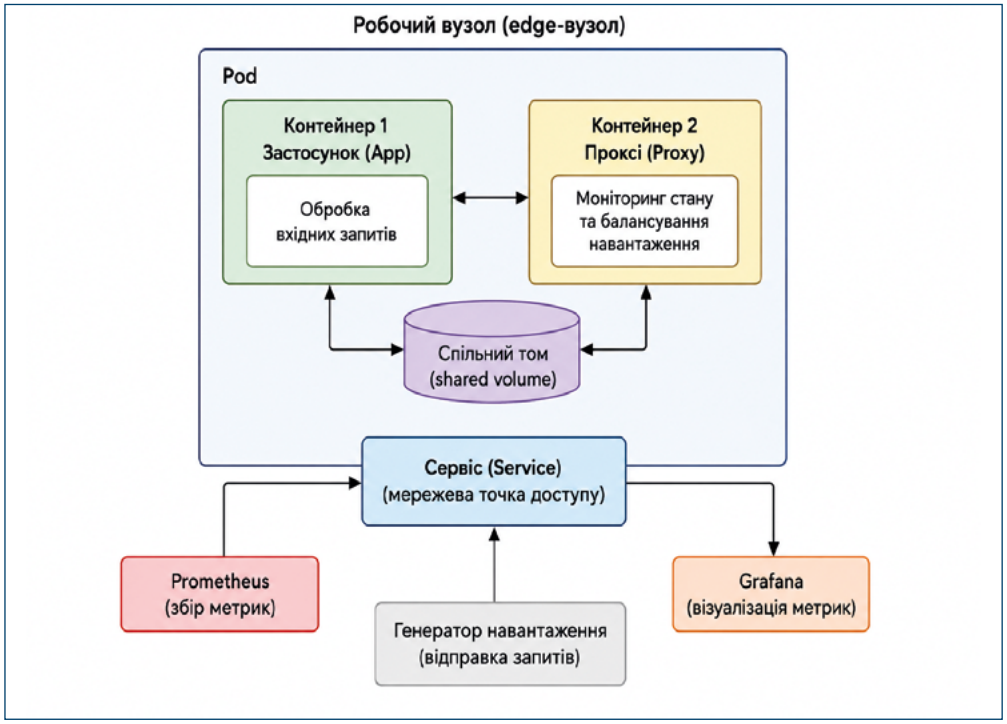


Рис. 4.6. Структурно-функціональна схема робочого вузла

Для забезпечення зовнішньої взаємодії використовується Kubernetes Service, через який здійснюється обмін даними між Pod та іншими компонентами експериментального середовища. Зокрема, генератор навантаження формує вхідний потік запитів, а система моніторингу Prometheus забезпечує періодичне збирання метрик із Edge-вузлів. Отримані показ-

ники характеризують поточне використання обчислювальних ресурсів і можуть використовуватися алгоритмами балансування під час прийняття рішень щодо локального виконання або перенесення завдань на інші вузли.

Накопичені Prometheus часові ряди використовуються також засобами Grafana для візуалізації та подальшого аналізу поведінки системи. Таким чином, підсистема моніторингу виконує дві функції: забезпечує алгоритми керування актуальною інформацією про стан ресурсів та формує експериментальні дані для оцінювання ефективності досліджуваних методів.

Для забезпечення відтворюваності експериментів обчислювальні ресурси контейнерів задаються на рівні конфігурації Kubernetes. Для прикладного контейнера встановлюються обмежені ресурси, що дає змогу штучно формувати стани дефіциту обчислювальної потужності за високої інтенсивності навантаження. Проксі-компоненту, навпаки, надається достатній ресурсний резерв, щоб його власні обмеження не впливали на результати оцінювання алгоритмів балансування.

Модель прикладного сервісу

Як приклад прикладного сервісу використано модель обробника підключень користувацького обладнання – UE-handler. Вона відтворює обчислювальну складову обслуговування запитів від мобільних пристроїв, ноутбуків, планшетів та інших UE, взаємодія яких у реальній мережі здійснюється через мережу радіодоступу (Radio Access Network, RAN) [74-77].

Для формування контрольованого навантаження UE-handler доповнено генератором запитів, який моделює активність користувачів. Залежно від поточного обсягу доступних ресурсів запит може оброблятися із застосуванням різних алгоритмів, що відрізняються обчислювальною складністю та якістю отриманого результату. За достатнього ресурсного забезпечення використовується точніший, але ресурсомісткіший алгоритм, тоді як у стані підвищеного навантаження система може перейти до спрощеного способу оброблення.

Вибір UE-handler як прикладного сценарію не обмежує загальність запропонованої моделі. Його використання дає змогу представити механізми балансування та адаптивного керування ресурсами на прикладі

конкретного телекомунікаційного сервісу, тоді як структурно-функціональні принципи моделі можуть бути застосовані до інших сервісів Edge-Cloud середовища.

UE-handler реалізовано як HTTP-сервіс на основі фреймворку Flask. Основним програмним інтерфейсом є endpoint /action, через який здійснюється виконання набору взаємозамінних алгоритмів (actions). Кожний із них характеризується певною обчислювальною складністю та відповідним рівнем якості результату. Такий підхід дає змогу досліджувати вплив доступності ресурсів на вибір способу оброблення запиту.

Для адаптації застосунку до зміни ресурсного стану реалізовано механізм brownout, який передбачає тимчасове спрощення окремих функцій сервісу за умов високого навантаження замість повної відмови в обслуговуванні. Рівень сервісу використовується як критерій вибору алгоритму: система прагне застосувати варіант із найвищою доступною якістю, за якого водночас виконуються встановлені вимоги до продуктивності. Це підхід, який базується на принципах керування і дає змогу динамічно знижувати навантаження, деактивуючи або обмежуючи необов'язкові компоненти застосунку. Brownout дозволяє контролювати рівень якості обслуговування, щоб досягати визначених цільових параметрів, наприклад певної максимальної затримки. Спочатку ця технологія застосовувалася у традиційних хмарних системах разом зі стратегіями балансування навантаження, однак у даному дослідженні її використання перенесено в контекст периферійної хмари (Edge computing), де вимоги до адаптивності є значно жорсткішими.

Поточний стан застосунку, зокрема показники використання ресурсів та параметри механізму brownout, періодично оновлюється й передається проксі-компоненту через спільний том. На основі цих даних проксі отримує актуальну інформацію про стан сервісу та використовує її у відповідних циклах керування навантаженням.

Для оцінювання часу обслуговування запитів до /action використовуються механізми Flask before_request та after_request. Перший фіксує момент початку оброблення запиту, а другий визначає тривалість його виконання перед формуванням відповіді. Отримане значення часу обслуговування (service time) використовується механізмами адаптивного

керування, зокрема контролером brownout та відповідними алгоритмами балансування навантаження.

Використання вебзастосунку як експериментального сервісу узгоджується з підходами, застосованими в попередніх дослідженнях методів керування навантаженням [78, 79]. Flask обрано завдяки простоті реалізації, невеликим накладним витратам і можливості контрольованого формування різних режимів оброблення запитів. Оскільки основною метою експериментів є порівняльне оцінювання досліджуваних алгоритмів за однакових умов, абсолютна продуктивність конкретного вебсервера не є визначальним фактором.

4.4. Модель формування навантаження та алгоритм вибору дій у системі Brownout

Для дослідження адаптивного керування навантаженням використовується endpoint/action, через який реалізовано виконання взаємозамінних дій (actions) із різною обчислювальною складністю. Саме запити до цього endpoint враховуються під час оцінювання стану прикладного сервісу та формування керувальних рішень. Інші запити не беруть безпосередньої участі у механізмі вибору дій і розглядаються як фонове навантаження системи.

Обчислювальне навантаження формується програмно шляхом виконання циклічних операцій із послідовним читанням даних із текстового файлу. Кількість виконаних ітерацій визначається типом вибраної дії та задає відносний обсяг ресурсів, необхідних для оброблення запиту. Такий спосіб дає змогу контрольовано змінювати обчислювальну складність операцій і формувати різні режими завантаження застосунку.

Поєднання циклічних обчислень із операціями читання даних дає змогу уникнути миттєвого завантаження процесора до граничного рівня та сформувати більш плавний профіль використання ресурсів. Водночас операції введення-виведення вносять певну варіативність у тривалість виконання запитів, зокрема через механізми кешування даних операційною системою. Така варіативність наближає експериментальне навантаження до реальних умов, у яких тривалість оброблення запитів за-

лежить від їхнього змісту, стану ресурсів, операцій введення-виведення та інших чинників.

Для самоналаштовуваної системи така невизначеність є важливою характеристикою експериментального середовища, оскільки алгоритм керування не повинен спиратися на наперед точно відомий час виконання кожної операції. Натомість рішення формуються динамічно на основі спостережуваного стану системи та фактичних показників її функціонування.

У реальних розподілених інформаційно-комунікаційних системах окремі запити можуть суттєво відрізнятися за обчислювальною складністю. Оброблення одного запиту може передбачати лише отримання вже сформованих даних, тоді як іншого – виконання декількох звернень до бази даних, агрегацію інформації, складні обчислення або залучення додаткових сервісів. Відповідно, навіть за однакової кількості запитів фактичне навантаження на процесор, оперативну пам'ять та мережеві ресурси може бути різним.

Наприклад, у вебсервісі запит користувача може оброблятися у спрощеному режимі шляхом повернення основної інформації з кешу. Розширений варіант того самого запиту може додатково передбачати звернення до бази даних, фільтрацію та агрегацію результатів. Ще складніший режим може включати персоналізацію відповіді, виконання аналітичних операцій або застосування моделі машинного навчання. У такому випадку всі три варіанти забезпечують виконання основної функції сервісу, проте потребують різного обсягу ресурсів і формують різну цінність для користувача.

Саме така особливість використовується в самоадаптивних системах (self-adaptive systems), у яких режим обслуговування може динамічно змінюватися відповідно до поточного стану ресурсів. Системі при цьому не обов'язково наперед точно знати тривалість виконання кожного запиту. Рішення приймається на основі контрольованих показників її стану, зокрема завантаження процесора, використання оперативної пам'яті, інтенсивності вхідного потоку запитів або інших доступних метрик. Це дає змогу зменшувати обчислювальну складність обслуговування в періоди перевантаження та, навпаки, використовувати доступний ресурсний резерв для надання розширеної функціональності.

Для моделювання такої поведінки в досліджуваному застосунку реалізовано три типи дій – Action 1, Action 2 та Action 3. Вони відповідають різним рівням обчислювальної складності та цінності результату.

Action 1 характеризується найменшою ресурсомісткістю та відповідає спрощеному режиму обслуговування. Його використання дає змогу скоротити обсяг виконуваних операцій і тим самим зменшити навантаження на систему. Наприклад, замість формування детального аналітичного результату сервіс може повернути лише основні дані, використати попередньо сформований результат із кешу або пропустити виконання необов'язкових додаткових операцій. Такий режим є доцільним за високого завантаження, коли пріоритетом стає збереження доступності сервісу та можливості обслуговувати максимальну кількість запитів.

Action 2 відповідає базовому режиму роботи та передбачає виконання основного набору операцій, необхідних для повноцінного обслуговування запиту. Наприклад, сервіс може виконати звернення до бази даних, опрацювати отримані значення та сформувати стандартну відповідь користувачеві. Цей режим використовується за нормального навантаження, коли доступних ресурсів достатньо для виконання основної функціональності без необхідності її спрощення.

Action 3 характеризується найбільшою обчислювальною складністю та забезпечує розширений рівень обслуговування. До основної операції можуть додаватися, наприклад, поглиблений аналіз даних, формування рекомендацій, додаткові запити до баз даних, персоналізація результату або виконання обчислювально складнішого алгоритму. Така дія потребує більше процесорного часу та інших ресурсів, але водночас забезпечує більшу цінність отриманого результату. Тому її доцільно виконувати за низького або помірного навантаження, коли система має достатній ресурсний резерв.

Отже, логіку адаптації можна представити таким чином: низьке навантаження → Action 3; нормальне навантаження → Action 2; високе навантаження → Action 1. При цьому перехід до менш ресурсомісткої дії не означає повного припинення обслуговування користувача. Система продовжує виконувати основну функцію, але тимчасово зменшує обсяг додаткових обчислень. Саме в цьому полягає принцип Brownout: за-

міст відмови сервісу в умовах дефіциту ресурсів частина необов'язкової функціональності спрощується або обмежується.

Такий підхід можна проілюструвати на прикладі інформаційно-аналітичного сервісу. За невеликої кількості користувачів система може сформувати розширену відповідь із персоналізованими рекомендаціями (Action 3). Зі збільшенням навантаження вона переходить до стандартного формування результату без додаткової аналітики (Action 2). Якщо ж навантаження наближається до критичного рівня, користувачеві може бути надана лише основна інформація (Action 1). У результаті зменшується обчислювальна складність окремого запиту, що дає змогу вивільнити частину ресурсів для обслуговування інших запитів і знизити ризик перевантаження системи.

Для оцінювання ефективності такого механізму недостатньо враховувати лише кількість успішно оброблених запитів, оскільки запити, виконані в різних режимах, мають неоднакову корисність. Тому використовується показник згенерованої цінності (generated value), який враховує як кількість успішно оброблених запитів за одиницю часу, так і тип виконаної дії. Кожному типу дії ставиться у відповідність ваговий коефіцієнт: менш ресурсомістка Action 1 має нижчу цінність, базова Action 2 – вищу, а розширена Action 3 – найбільшу.

Наприклад, дві конфігурації можуть успішно обробляти однакові 100 запитів за секунду, але в першій більшість запитів виконується в режимі Action 1, а в другій – Action 3. За показником пропускної здатності ці конфігурації виглядатимуть однаково, однак друга забезпечує користувачам функціонально повніший результат. Показник згенерованої цінності дає змогу врахувати цю відмінність і оцінювати не лише продуктивність системи, а й корисність забезпеченого нею рівня обслуговування.

Таким чином, експериментальна модель дає змогу дослідити компроміс між ресурсомісткістю виконуваних операцій, продуктивністю системи та цінністю наданого сервісу. Механізм Brownout динамічно вибирає рівень обслуговування залежно від поточного стану ресурсів, забезпечуючи збереження працездатності системи за високого навантаження та використання розширеної функціональності за наявності достатнього ресурсного резерву. Параметри реалізованих типів дій та відповідні їм характеристики наведено в табл. 4.1.

Таблиця 4.1

Дії та їх відповідні навантаження й згенеровані цінності

Дія	Відносне навантаження	Згенерована цінність
Action 1	0.5	0.75
Action 2	1.0	1.0
Action 3	1.5	1.3

Зі збільшенням обчислювальної складності дії зростає цінність отриманого результату, однак одночасно збільшується навантаження на ресурси. Така залежність формує компроміс між якістю обслуговування та ресурсними витратами, який є основою для прийняття адаптивних рішень системою Brownout. Вибрані співвідношення параметрів ґрунтуються на принципі диференціації вартості виконання альтернативних дій та адаптовані до моделі з трьома рівнями сервісу.

Алгоритм вибору дії

Керування режимом оброблення запитів здійснюється за допомогою параметра рівня сервісу (service level), значення якого нормується в діапазоні від 0 до 1. Цей параметр визначає, з якою ймовірністю для оброблення чергового запиту буде вибрано одну з трьох дій, що відрізняються рівнем ресурсомісткості та цінністю результату. Зміна рівня сервісу дає змогу регулювати частку менш або більш ресурсомістких дій відповідно до поточного завантаження системи. За високого навантаження перевага надається спрощеному режиму оброблення, тоді як за наявності достатнього ресурсного резерву частіше можуть виконуватися дії з більшою обчислювальною складністю та вищою цінністю результату.

На відміну від класичного механізму Brownout [80-829], який передбачає два стани опціональної функціональності, її виконання або невиконання , у досліджуваній моделі використано три режими оброблення запитів. Такий підхід забезпечує поступову зміну обчислювального навантаження та дає змогу гнучкіше адаптувати ресурсоспоживання сервісу до поточного стану систем.

Для оцінювання часових характеристик роботи системи використовується 95-й перцентиль затримки обслуговування. Він дає змогу оцінити затримку для переважної більшості запитів і водночас врахувати

випадки погіршення часу обслуговування, які можуть бути непомітними при використанні лише середнього значення.

Отримані значення затримки зберігаються у ковзному вікні фіксованого розміру. Після надходження нового значення воно додається до поточної вибірки, а найстаріше значення видаляється. Завдяки цьому для оцінювання стану системи постійно використовуються останні результати спостережень. На основі оновленої вибірки обчислюється 95-й перцентиль затримки, який використовується контролером під час визначення необхідного рівня сервісу. Рівень сервісу задається нормованим значенням у межах від нуля до одиниці та визначає співвідношення між трьома режимами оброблення запитів. Низьке значення рівня сервісу збільшує ймовірність використання Action 1, тобто спрощеного та найменш ресурсомісткого режиму. Середнє значення відповідає переважному використанню Action 2 як базового режиму. Зі збільшенням рівня сервісу зростає ймовірність вибору Action 3, яка забезпечує розширене оброблення запиту, але потребує більшого обсягу обчислювальних ресурсів.

Вибір конкретної дії для кожного запиту здійснюється ймовірнісно. Якщо рівень сервісу нижчий за середнє значення, алгоритм розподіляє запити між Action 1 та Action 2. Чим нижчим є рівень сервісу, тим більша частка запитів обробляється у спрощеному режимі. Наприклад, за суттєвого зростання навантаження система може частіше використовувати Action 1, зменшуючи таким чином обсяг обчислень, необхідний для обслуговування запитів.

Якщо рівень сервісу перевищує середнє значення, вибір здійснюється між Action 2 та Action 3. У міру збільшення доступного ресурсного резерву частка запитів, для яких використовується Action 3, поступово зростає. Таким чином, за низького навантаження система може частіше виконувати розширений набір операцій і забезпечувати вищу цінність результату.

Важливою особливістю такого підходу є те, що зміна рівня сервісу не призводить до різкого перемикання всіх запитів з одного режиму на інший. Натомість поступово змінюється співвідношення між кількістю запитів, оброблених за допомогою Action 1, Action 2 та Action 3. Це забезпечує плавну адаптацію ресурсоспоживання до поточного стану сис-

теми та є основною відмінністю реалізованого механізму від класичного Brownout із двома станами.

Таким чином, контролер формує замкнений цикл адаптивного керування. Затримки оброблення запитів накопичуються у ковзному вікні, на їх основі оцінюється поточний стан системи, відповідно коригується рівень сервісу, після чого змінюється співвідношення між доступними режимами оброблення запитів. Це дає змогу зменшувати ресурсомісткість сервісу в періоди підвищеного навантаження та використовувати розширений режим обслуговування за наявності достатнього резерву обчислювальних ресурсів.

Алгоритм адаптивного вибору режиму оброблення запитів залежно від поточного рівня сервісу наведено на рис. 4.7. Умовні позначення, використані символи та їх пояснення подано нижче:

- service_level – поточний рівень сервісу, нормований у діапазоні від 0 до 1;
- r – випадкове число в діапазоні від 0 до 1, яке використовується для ймовірнісного вибору дії;
- Action 1 – спрощений режим оброблення запиту з найменшим навантаженням;
- Action 2 – базовий режим оброблення запиту;
- Action 3 – розширений режим оброблення запиту з найбільшим навантаженням і вищою цінністю результату;
- d – значення затримки обслуговування запиту;
- $d_1, d_2, \ldots, d_n$ – останні значення затримки, що зберігаються у ковзному вікні;
- d_new – нове значення затримки, яке додається до ковзного вікна;
- P95 – 95-й перцентиль затримки, розрахований за значеннями поточного ковзного вікна.

Ймовірності вибору дій визначаються рівнем сервісу. За service_level = 0 система завжди обирає Action 1, за service_level = 0,5 – Action 2, а за service_level = 1 – Action 3. Для проміжних значень імовірності змінюються лінійно. Відповідно, Action 2 домінує в центральній частині діапазону та виконує роль основного режиму роботи системи, тоді як Action 1 і Action 3 забезпечують адаптацію відповідно до дефіциту або надлишку доступних ресурсів.

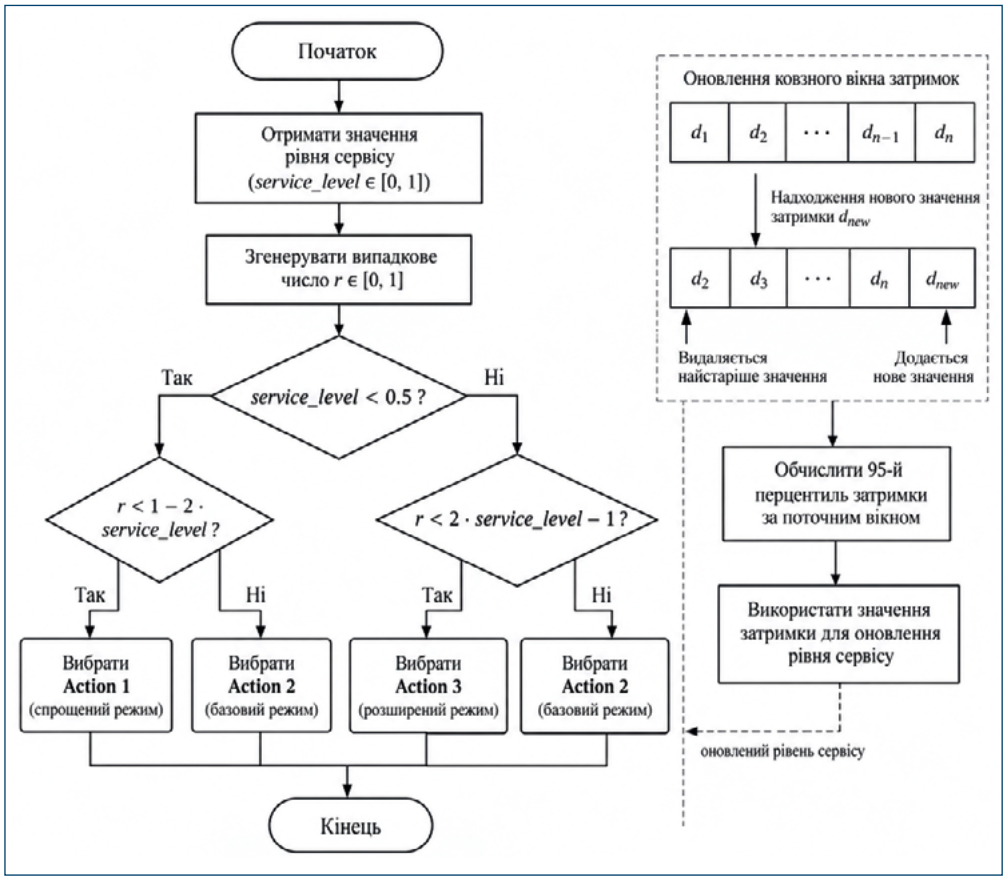


Рис. 4.7. Блок-схема алгоритму адаптивного вибору режиму оброблення запитів на основі рівня сервісу

4.5. Метод адаптивного управління навантаженням у розподіленому Edge-Cloud середовищі

Ефективність функціонування Edge-Cloud системи значною мірою визначається здатністю механізмів керування оперативно реагувати на нерівномірність і динамічні зміни навантаження. На відміну від централізованої хмарної інфраструктури, Edge-вузли мають обмежені обчислювальні ресурси та безпосередньо обслуговують потоки запитів від кінцевих користувачів. За таких умов перевантаження окремого вузла може спричинити збільшення часу обслуговування та погіршення показників якості надання сервісу.

Для адаптивного розподілу запитів використано ресурсно-орієнтований підхід, за якого рішення щодо місця оброблення запиту приймається на основі актуального стану обчислювальних ресурсів. Загальна логіка передбачає пріоритет локального виконання: доки поточний Edge-вузол має достатній ресурсний резерв, запити обробляються локально. У разі перевищення встановлених порогових значень частина навантаження перенаправляється на інший вузол із кращим ресурсним станом.

Архітектура проксі та моніторинг стану системи

Функції балансування реалізовано у проксі-компоненті, розгорнутому в окремому контейнері від прикладного сервісу. Проксі виконує роль проміжного рівня між джерелом запитів та застосунком і визначає місце оброблення кожного нового запиту. Залежно від стану ресурсів запит спрямовується до локального застосунку або передається на інший Edge-вузол.

Реалізація проксі ґрунтується на Flask і функціонує в просторі користувача (user space). Концептуально такий підхід базується на RAP, запропонованому Nguyen та ін. [73]. На відміну від оригінальної реалізації з модифікованим KubeProxy, у сформованому експериментальному середовищі функції проксі реалізовано на прикладному рівні, що дало змогу досліджувати та модифікувати алгоритми балансування без внесення змін у системні компоненти Kubernetes.

Для прийняття рішень проксі використовує як локальні показники стану застосунку, так і інформацію про інші Edge-вузли. Збирання та агрегація відповідних метрик здійснюються за допомогою Prometheus. Локальні показники застосунку передаються до проксі через спільний том (shared volume), 1, що забезпечує швидке отримання актуальної інформації без додаткового мережевого обміну.

Набір метрик, які використовуються для моніторингу та реалізації алгоритмів балансування, наведено в табл. 4.2.

Таблиця 4.2

Додаткові метрики вимірювання, які відстежує проксі

Метрика	Опис	Тип
1	2	3
brownout	Поточний рівень сервісу (у межах 0–1)	Gauge

Продовження табл. 4.2

1	2	3
cpu_usage	Використання CPU у % від виділених ресурсів	Gauge
ram_usage	Використання пам'яті у % від виділених ресурсів	Gauge
handled	Кількість запитів, оброблених локально (без аутсорсингу)	Counter
usage	Кількість запитів, що повинні бути оброблені + поточні активні запити	Counter
outsourced	Кількість запитів, переданих іншим вузлам	Counter
received	Кількість запитів, отриманих безпосередньо від генератора навантаження	Counter

Метрики brownout, cpu_usage та ram_usage характеризують стан прикладного сервісу та передаються проксі через спільний том. Решта показників формуються безпосередньо проксі та використовуються для оцінювання характеру розподілу запитів і результативності алгоритмів балансування. Значення локальних параметрів оновлюються з більшою частотою, ніж інформація про віддалені вузли, що забезпечує швидшу реакцію на зміни стану власного застосунку.

Для розмежування первинних і перенаправлених запитів реалізовано дві точки доступу: /action та /actionProxy. Запити до /action проходять через механізм балансування, тоді як /actionProxy використовується для запитів, уже перенаправлених з іншого вузла, і забезпечує їх безпосереднє передавання локальному застосунку. Така організація запобігає повторному перенаправленню одного й того самого запиту між Edge-вузлами та виключає можливість утворення циклів.

Ресурсно-адаптивний алгоритм RAP

У розподіленому Edge-середовищі навантаження між окремими вузлами може розподілятися нерівномірно. У певний момент один вузол може інтенсивно використовувати процесорні ресурси або оперативну пам'ять, тоді як інші вузли системи мають достатній ресурсний резерв. За таких умов оброблення всіх запитів виключно на вузлі, до якого вони надійшли, може призводити до локального перевантаження, збільшення затримки та зниження продуктивності сервісу. Тому важливим завданням є динамічний перерозподіл запитів з урахуванням поточного ресурсного стану Edge-вузлів.

Як основу механізму адаптивного балансування використано алгоритм RAP. Його принцип полягає у періодичному оцінюванні стану обчислювальних ресурсів доступних Edge-вузлів та визначенні вузла, найбільш придатного для приймання додаткового навантаження. Якщо локальний вузол має достатній запас ресурсів, запит обробляється безпосередньо на ньому. Якщо використання ресурсів досягає встановленого порогового рівня, нові запити можуть бути спрямовані на інший вузол із кращим ресурсним станом.

Такий підхід дає змогу зберегти переваги локального Edge-оброблення та одночасно використовувати вільні ресурси інших компонентів розподіленої системи. Локальне оброблення залишається пріоритетним, оскільки не потребує додаткового передавання даних між вузлами. Перенаправлення активується лише за недостатнього ресурсного резерву локального вузла.

В оригінальному RAP інформація про використання ресурсів отримується засобами Kubernetes. У запропонованій реалізації джерелом глобальної інформації про стан Edge-вузлів є система моніторингу Prometheus. Вона забезпечує централізований збір метрик з усіх вузлів, після чого отримані значення використовуються проксі для оцінювання їх поточного завантаження.

Основними контрольованими параметрами є використання процесора та оперативної пам'яті. Для кожного вузла визначаються усереднені значення цих показників за останній часовий інтервал. У межах експериментального дослідження інтервал усереднення прийнято рівним 5 с. Таке значення дає змогу зменшити вплив короткочасних коливань ресурсного навантаження та водночас зберегти достатню швидкість реакції на його стійкі зміни.

Використання усереднених значень є важливим для стабільної роботи алгоритму. Короткочасне зростання використання CPU не завжди означає, що вузол дійсно потребує розвантаження. Якщо рішення приймати лише за миттєвими значеннями, система може надто часто змінювати напрям передавання запитів. Усереднення за невеликий часовий інтервал дає змогу зменшити вплив таких коливань та приймати рішення на основі більш стійкої оцінки ресурсного стану.

Разом із глобальним моніторингом у реалізованій системі використовується окремий механізм контролю локального вузла. Інформа-

ція про його поточний стан оновлюється з більшою частотою та передається через спільний том shared volume. Це дає змогу швидше виявляти локальне перевантаження, не очікуючи завершення наступного глобального циклу збору метрик Prometheus.

Отже, механізм RAP містить два взаємопов'язані контури керування. Глобальний контур періодично аналізує ресурсний стан усіх доступних Edge-вузлів і визначає найкращий вузол для можливого перенаправлення запитів. Локальний контур контролює стан поточного вузла та визначає необхідність такого перенаправлення. Поділ цих функцій дає змогу поєднати глобальну інформацію про стан розподіленої системи зі швидкою реакцією на локальні зміни навантаження.

Робота глобального контрольного циклу починається зі збору Prometheus актуальних метрик CPU та RAM для доступних Edge-вузлів. Після отримання даних виконується перевірка кожного вузла на відповідність установленим обмеженням. Якщо використання процесора або оперативної пам'яті досягає заданого порогового значення, такий вузол виключається з множини кандидатів для приймання додаткових запитів. Це запобігає перенаправленню навантаження на вузли, які вже перебувають у стані високого завантаження.

Для вузлів, які пройшли попередню перевірку, виконується оцінювання ресурсного стану. Оскільки для вибору враховується декілька показників, їх значення нормалізуються та об'єднуються в інтегральну оцінку. Нормалізація забезпечує приведення різних ресурсних характеристик до порівнянного вигляду, а використання вагових коефіцієнтів дає змогу врахувати важливість кожного ресурсу для роботи конкретного застосунку.

Інтегральна ресурсна оцінка i-го Edge-вузла визначається як

$$S_i = \sum_{r \in R} w_r N_{i,r}, \tag{4.1}$$

де S_i є інтегральною оцінкою ресурсного стану i-го Edge-вузла; R є множиною ресурсів, які враховуються під час вибору вузла; $N_{i,r}$ є нормалізованою оцінкою доступності ресурсу r на i-му вузлі; W_r є ваговим коефіцієнтом відповідного ресурсу.

Вагові коефіцієнти визначають внесок окремих ресурсів у загальну оцінку вузла. Наприклад, для застосунків із високою інтенсивністю обчислень більшу вагу може мати доступність CPU, тоді

як для сервісів, що опрацьовують значні обсяги даних, важливішим може бути резерв оперативної пам'яті. Таким чином, механізм вибору може бути налаштований відповідно до ресурсних особливостей конкретного сервісу.

Для коректної інтерпретації інтегральної оцінки нормалізоване значення має характеризувати доступність відповідного ресурсу. Більше значення показника відповідає більшому ресурсному резерву вузла. Після розрахунку оцінок для всіх допустимих кандидатів вибирається вузол із максимальним значенням. Він зберігається як *best_node* та використовується як пріоритетний вузол для перенаправлення запитів до наступного оновлення інформації про стан системи.

Роботу алгоритму доцільно представити у вигляді двох взаємопов'язаних процедур. Перша процедура періодично визначає найкращий вузол на основі глобальної інформації про використання ресурсів. Друга виконується під час надходження кожного нового запиту та визначає, чи потрібно використовувати знайдений вузол. Узагальнену послідовність роботи ресурсно-адаптивного методу наведено на рис. 4.8.

Як показано на рис. 4.8, запит користувача спочатку надходить до проксі локального Edge-вузла. Після цього перевіряється поточне використання його процесорних ресурсів та оперативної пам'яті. Якщо обидва показники залишаються нижчими за встановлені порогові значення, запит передається локальному застосунку та обробляється без залучення інших вузлів.

Якщо локальний вузол не має достатнього ресурсного резерву, проксі використовує інформацію про best_node, отриману під час останнього глобального контрольного циклу. Запит передається на визначений Edge-вузол, де виконується його оброблення. Після завершення операції результат повертається через проксі користувачеві. Таким чином, зміна місця фактичного виконання запиту не впливає на логіку взаємодії користувача із сервісом.

Важливою особливістю цього підходу є те, що повний аналіз стану всіх Edge-вузлів не виконується для кожного запиту. Найкращий вузол визначається періодично та зберігається до наступного циклу оновлення. Під час надходження запиту необхідно лише перевірити актуальний стан локального вузла та визначити можливість локального

оброблення. Це зменшує кількість додаткових операцій, які необхідно виконати безпосередньо під час обслуговування запиту.

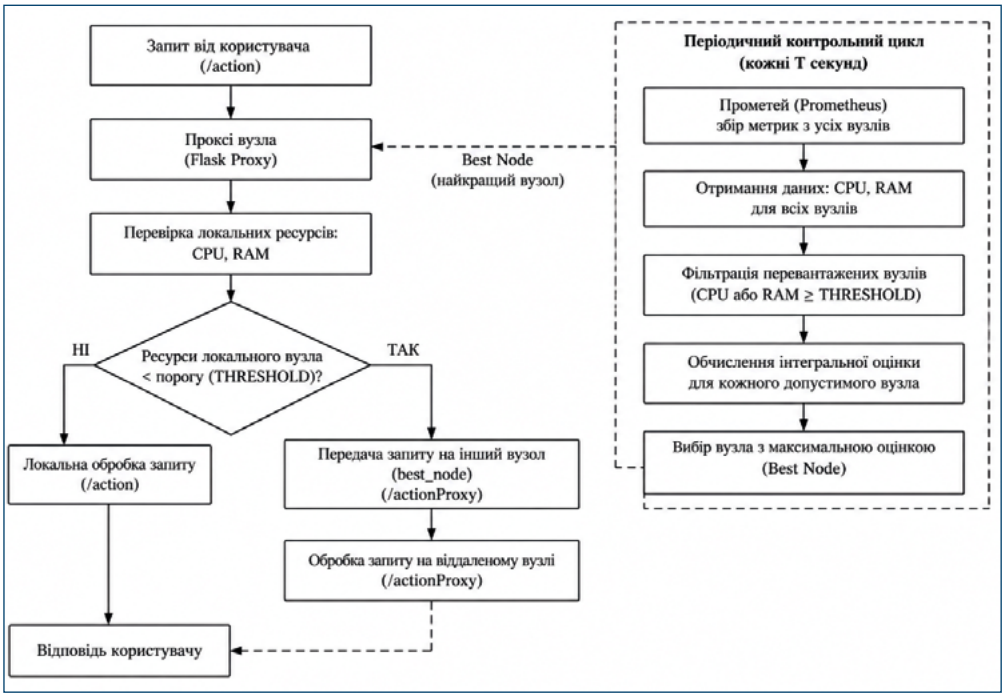


Рис. 4.8. Блок-схема ресурсно-адаптивного алгоритму вибору Edge-вузла для оброблення запитів

Функція CALCULATE_BEST_NODE() реалізує глобальну частину алгоритму. На початку її виконання отримуються актуальні показники використання CPU та оперативної пам'яті для доступних Edge-вузлів. Початково як best_node задається локальний вузол. Далі послідовно аналізуються інші вузли системи.

Для кожного вузла перевіряється відповідність показників CPU та RAM установленому порогу THRESHOLD. Якщо хоча б один показник досягає порогового значення, вузол не розглядається як кандидат для приймання додаткового навантаження. Для вузлів із достатнім ресурсним резервом виконується нормалізація контрольованих показників та розраховується інтегральна оцінка. Якщо отримане значення перевищує найкращу знайдену оцінку, відповідний вузол зберігається як новий best_node. Після перевірки всіх кандидатів функція повертає вузол із найкращим ресурсним станом.

Функція GET_NODE_TO_HANDLE_REQUEST() реалізує локальну частину алгоритму та виконується під час надходження нового запиту. Спочатку перевіряється поточне використання CPU та оперативної пам'яті локального вузла. Якщо обидва показники залишаються нижчими за встановлений поріг, функція вибирає локальний вузол. Якщо хоча б один із ресурсів досягає критичного рівня, для оброблення запиту вибирається раніше визначений best_node.

Таким чином, локальне оброблення має пріоритет перед перенаправленням. Наявність у системі вузла з більшим обсягом вільних ресурсів сама по собі не є підставою для передавання на нього запиту. Перенаправлення здійснюється лише тоді, коли локальний вузол не може продовжувати приймати навантаження без ризику погіршення продуктивності. Це дає змогу уникати необов'язкового мережевого обміну та використовувати ресурси інших Edge-вузлів лише за необхідності.

Поєднання двох процедур забезпечує ресурсно-адаптивне балансування навантаження в розподіленому Edge-середовищі. Періодичний глобальний цикл визначає найбільш придатний вузол для приймання додаткових запитів, а локальний контроль забезпечує оперативне прийняття рішення щодо необхідності перенаправлення. У результаті система може реагувати на нерівномірний розподіл навантаження, уникати передавання запитів на перевантажені вузли та ефективніше використовувати доступний ресурсний резерв розподіленої інфраструктури.

Для кращого розуміння принципу роботи RAP розглянемо простий приклад. Нехай система складається з трьох Edge-вузлів, а допустимий рівень використання ресурсів становить 80 %. Запит користувача надходить на перший вузол. У цей момент його CPU завантажений на 85 %, тому вузол уже не повинен приймати додаткове навантаження.

У такому випадку проксі не обробляє новий запит на першому вузлі, а перевіряє, який із інших вузлів був визначений як найбільш придатний для його перенаправлення. За даними Prometheus, другий вузол використовує 45 % CPU і 50 % оперативної пам'яті, а третій вузол відповідно 65 % і 75 %. Обидва вузли можуть прийняти запит, оскільки використання їх ресурсів не перевищує встановленого порогу. Однак другий вузол має більший запас вільних ресурсів, тому

алгоритм визначає його як найкращий вузол для приймання додаткового навантаження.

Новий запит перенаправляється з першого на другий Edge-вузол, де й виконується його оброблення. Після завершення оброблення отриманий результат повертається користувачеві. Таким чином, замість подальшого навантаження вже завантаженого першого вузла система використовує вільні ресурси другого вузла.

Якщо згодом навантаження на першому вузлі зменшиться і використання його CPU та оперативної пам'яті знову стане нижчим за 80 %, наступні запити оброблятимуться безпосередньо на ньому. Отже, логіка RAP є такою: поки локальний Edge-вузол має достатньо ресурсів, запити обробляються локально; якщо він перевантажується, нові запити передаються на інший вузол, який на цей момент має більший ресурсний резерв.

Отже, RAP поєднує локальне та глобальне прийняття рішень. Локальна перевірка забезпечує швидку реакцію на стан поточного вузла, тоді як періодичне ранжування інших Edge-вузлів дає змогу використовувати інформацію про ресурсний стан розподіленої системи загалом. Це зменшує необхідність виконання ресурсоємного глобального пошуку для кожного окремого запиту та водночас забезпечує адаптацію маршрутизації до зміни навантаження.

Таким чином, у роботі запропоновано метод адаптивного керування навантаженням у розподілених Edge-Cloud системах, який поєднує два механізми адаптації. Перший, на основі RAP, визначає доцільність локального оброблення запиту або його перенаправлення на інший Edge-вузол з урахуванням поточного використання CPU та оперативної пам'яті. Другий, на основі Brownout, дає змогу змінювати рівень сервісу та відповідно ресурсомісткість оброблення запитів залежно від поточного навантаження. Завдяки їх спільному використанню система може не лише перерозподіляти запити між Edge-вузлами, а й зменшувати навантаження безпосередньо на локальному вузлі шляхом вибору менш ресурсомісткого режиму обслуговування. Це зменшує потребу в міжвузловому перенесенні запитів, сприяє раціональнішому використанню доступних обчислювальних ресурсів та дає змогу підтримувати необхідні часові характеристики обслуговування за нерівномірного і змінного навантаження.

4.6. Експериментальне середовище та методика оцінювання ресурсно-адаптивного балансування в Edge-Cloud системі

Для експериментального оцінювання алгоритмів балансування навантаження сформовано середовище, у якому екземпляри застосунку, розгорнуті на Edge-вузлах, функціонують під керованим HTTP-навантаженням. Генерування трафіку здійснюється за допомогою Locust [83], розгорнутого на окремому вузлі. Locust це інструмент для генерування штучного навантаження та навантажувального тестування програмних систем. Він дає змогу імітувати одночасну роботу великої кількості користувачів, які надсилають HTTP-запити до досліджуваного сервісу. За допомогою Locust можна задавати кількість віртуальних користувачів, інтенсивність їх появи та сценарії надсилання запитів, а також оцінювати час відповіді, пропускну здатність і кількість помилок. Таке розміщення генератора дає змогу забезпечити порівнювані умови мережевої взаємодії з усіма екземплярами застосунку в досліджуваному кластері.

Кожний експеримент виконується за фіксованої кількості одночасних користувачів і триває 5 хвилин. Кількість активних користувачів поступово збільшується від нуля до заданого максимального значення протягом перших 24 с тестування. Це забезпечує поступове формування навантаження та стабілізацію контрольних циклів досліджуваних алгоритмів. Перші 30 с кожного експерименту визначено як початковий період стабілізації (warm-up), і їх не враховано під час подальшого опрацювання результатів.

Поведінка віртуального користувача передбачає початкове встановлення взаємодії з одним із чотирьох Edge-вузлів, після чого протягом усього експерименту генерується один запит за секунду до кінцевої точки / action. За малої затримки обслуговування теоретично досяжна пропускна здатність системи наближається до кількості одночасно активних користувачів, оскільки кожний із них формує один запит за секунду.

Усі запити мають однаковий пріоритет і не відрізняються за ресурсними вимогами. Це дає змогу мінімізувати вплив характеристик окре-

мих типів трафіку та зосередити оцінювання на поведінці алгоритмів балансування навантаження.

Експериментальне середовище побудовано на основі спрощеної Edge-Cloud архітектури з використанням Kubernetes. За основу структурної організації середовища взято модель, запропоновану Nguyen та ін. [73], адаптовану до умов проведеного дослідження. Чотири екземпляри застосунку розгорнуто на чотирьох окремих робочих вузлах, тоді як генератор навантаження функціонує на виділеному вузлі. Структуру експериментального середовища наведено на рис. 4.9.

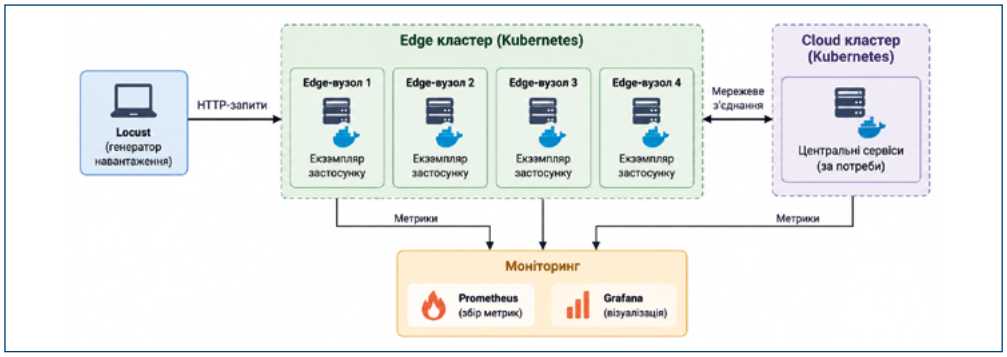


Рис. 4.9. Структура експериментального Edge-Cloud середовища для оцінювання алгоритмів балансування навантаження

На етапі попередніх експериментів контрольовані показники візуалізувалися засобами Grafana, що дало змогу спостерігати за динамікою навантаження та виявляти нетипову поведінку системи. Під час основної серії експериментів значення метрик отримували безпосередньо через API Prometheus і зберігали у CSV-файлах для подальшого опрацювання. Паралельне використання Grafana забезпечувало візуальний контроль коректності перебігу експериментів.

Тестове середовище розгорнуто в кластері OpenShift в окремому просторі імен (namespace). Інфраструктура кластера містила 13 робочих та 3 керуючі вузли на базі серверів ProLiant DL360 Gen10 з процесорами Intel Xeon Gold 6252 2,10 GHz, оперативною пам'яттю 6 × 32 GB та SSD-сховищем 960 GB у конфігурації RAID-1 із додатковим сховищем 2 × 1,92 TB.

Для кількісного порівняння досліджуваних алгоритмів використано показники, що характеризують результативність обслуговування запи-

тів та часові характеристики системи. Основні метрики експериментального оцінювання наведено в табл. 4.3.

Таблиця 4.3

Метрики оцінювання алгоритмів адаптивного балансування навантаження

Метрика	Одиниця вимірювання	Характеристика
Згенерована цінність (Generated Value)	бал/с	Інтегральний показник кількості успішно оброблених запитів з урахуванням рівня сервісу, вибраного механізмом Brownout
Середня затримка	мс	Середній час оброблення запиту та формування відповіді
USP (Under SetPoint)	%	Частка завершених запитів із затримкою, меншою за встановлене цільове значення
Пропускна здатність (Throughput)	запитів/с	Кількість запитів, успішно оброблених системою за одиницю часу

Показник Generated Value характеризує обсяг наданого сервісу з урахуванням рівня обслуговування, вибраного механізмом Brownout. На відміну від підходу Klein та ін. [13], де результативність сервісу пов'язується з отриманим прибутком, у цьому дослідженні показник використовується для узагальненого кількісного оцінювання фактично наданого рівня сервісу без безпосередньої прив'язки до його економічної вартості.

Застосування Brownout передбачає можливість оброблення запитів із різними рівнями сервісу, кожному з яких відповідає певний ваговий коефіцієнт. Тому максимально можлива згенерована цінність визначається як

$$V_{max} = N_u \cdot r_u \cdot k_{max} \qquad (4.2)$$

де N_u – кількість одночасно активних користувачів; r_u – частота формування запитів одним користувачем, запитів/с; k_{max} – максимальний коефіцієнт цінності вибраного рівня сервісу. У проведених експериментах $r_u = 1$ *запит / с*, а $k_{max} = 1{,}3$ за використання Brownout і $1{,}0$ для базового рівня сервісу.

Згенерована цінність виражається в умовних балах і характеризує корисний результат оброблення запитів з урахуванням рівня наданого сервісу. Один бал відповідає цінності одного успішно обробленого за-

питу на базовому рівні сервісу, для якого ваговий коефіцієнт прийнято рівним 1,0. Для розширеного рівня сервісу з коефіцієнтом 1,3 один успішно оброблений запит відповідає 1,3 бала. Таким чином, цей показник дає змогу враховувати не лише кількість оброблених запитів, а й цінність рівня сервісу, з яким вони були виконані.

Наприклад, для 48 одночасних користувачів за умови мінімальної затримки та використання максимального рівня сервісу теоретична верхня межа становить:

$$V_{max} = 48 \cdot 1 \cdot 1{,}3 = 62{,}4 \; бал / с. \qquad (4.3)$$

Значення 62,4 характеризує не кількість оброблених запитів, а їх сумарну цінність з урахуванням рівня сервісу/

Для оцінювання ефективності запропонованих механізмів адаптивного керування та балансування навантаження використовується сукупність показників, які характеризують продуктивність системи, часові характеристики обслуговування запитів, цінність отриманого результату та інтенсивність перенесення навантаження між Edge-вузлами. До основних показників належать пропускна здатність, згенерована цінність (Generated Value), середня затримка, частка запитів, виконаних у межах установленої часової вимоги (User Satisfaction Percentage, USP), а також частка міжвузлового перенесення навантаження (inter-edge offloading).

Пропускна здатність характеризує кількість запитів, які система успішно обробляє за одиницю часу, та вимірюється в запитах за секунду. Цей показник дає змогу оцінити здатність системи обслуговувати вхідне навантаження за наявного обсягу обчислювальних ресурсів. Чим більшою є пропускна здатність, тим більшу кількість запитів система може успішно обробити протягом заданого проміжку часу.

Однак під час використання Brownout самої пропускної здатності недостатньо для повної оцінки результатів. Причина полягає в тому, що запити можуть оброблятися з різними рівнями сервісу, яким відповідають різні значення ресурсомісткості та цінності результату. Тому додатково використовується показник згенерованої цінності Generated Value. Він характеризує сумарну цінність успішно оброблених запитів з урахуванням того, який режим обслуговування був використаний для кожного з них.

У конфігурації без Brownout усі запити обробляються в базовому режимі Action 2, для якого коефіцієнт цінності дорівнює 1,0. За таких умов одному успішно обробленому запиту відповідає один умовний бал цінності. Тому числове значення Generated Value збігається з числовим значенням пропускної здатності. Наприклад, якщо система успішно обробляє 40 запитів за секунду, її пропускна здатність становить 40 запитів/с, а згенерована цінність становить 40 умовних балів/с.

За використання Brownout ці показники можуть відрізнятися. Частина запитів може оброблятися у спрощеному режимі Action 1, частина у базовому режимі Action 2, а частина у розширеному режимі Action 3. Кожному режиму відповідає власний коефіцієнт цінності. Тому Generated Value залежить не лише від кількості успішно оброблених запитів, а й від співвідношення між різними режимами їх обслуговування. Наприклад, виконання запиту в режимі Action 3 з коефіцієнтом цінності 1,3 забезпечує більший внесок у Generated Value, ніж виконання такого самого запиту в базовому режимі Action 2 з коефіцієнтом 1,0.

Отже, пропускна здатність показує, скільки запитів система успішно обробила, а Generated Value характеризує, яку сумарну цінність було отримано в результаті їх оброблення. Розділення цих показників є важливим для коректного порівняння конфігурацій із Brownout та без його використання.

Для оцінювання часових характеристик роботи системи використовується середня затримка обслуговування запитів. Вона характеризує середній час від моменту надходження запиту до отримання результату його оброблення. Зростання середньої затримки може свідчити про збільшення навантаження на обчислювальні ресурси, накопичення запитів або виникнення додаткових затримок унаслідок їх перенаправлення між Edge-вузлами.

Разом із середньою затримкою використовується показник User Satisfaction Percentage (USP), який характеризує частку запитів, оброблених у межах установленої часової вимоги. Для цього задається цільове значення затримки, яке позначається як setpoint. Якщо час оброблення конкретного запиту не перевищує setpoint, такий запит вважається виконаним із дотриманням заданої часової вимоги. USP визначає відсоток таких запитів від їх загальної кількості.

Наприклад, якщо цільове значення затримки встановлено на рівні 300 мс і протягом експерименту було оброблено 1 000 запитів, з яких 950 отримали відповідь не пізніше ніж через 300 мс, значення USP становить 95 %. Це означає, що для 95 % запитів система забезпечила встановлену часову вимогу. Чим вищим є значення USP, тим стабільніше система забезпечує необхідний час обслуговування користувацьких запитів.

Використання USP дає змогу доповнити оцінку за середньою затримкою. Середнє значення може бути прийнятним навіть тоді, коли певна частина запитів обробляється значно довше за допустимий час. USP безпосередньо показує, для якої частки запитів установлена вимога щодо затримки була фактично виконана. Тому цей показник є важливим для оцінювання стабільності обслуговування в умовах динамічної зміни навантаження.

Для Edge-Cloud систем значення USP має особливе значення, оскільки запити можуть оброблятися не лише на локальному вузлі, а й перенаправлятися на інші Edge-вузли. Таке перенесення навантаження дає змогу використовувати вільні обчислювальні ресурси інших вузлів, однак потребує додаткового мережевого обміну. У результаті загальний час обслуговування окремих запитів може збільшуватися. Тому ефективність балансування необхідно оцінювати не лише за кількістю успішно виконаних запитів, а й за часткою запитів, для яких забезпечено допустимий час відповіді.

Для детальнішого контролю часових характеристик використовується також 95-й перцентиль затримки. Він визначає значення часу відповіді, яке не перевищують 95 % оброблених запитів. Наприклад, якщо 95-й перцентиль дорівнює 280 мс, це означає, що 95 % запитів були оброблені за час не більше 280 мс, а для 5 % запитів час відповіді був більшим.

На відміну від середньої затримки, 95-й перцентиль краще відображає погіршення часу обслуговування для частини запитів у періоди підвищеного навантаження. Саме тому цей показник використовується контролером Brownout для визначення поточного стану системи. Такий підхід відповідає принципам оцінювання часових характеристик, використаним у роботах Nylander та ін. [84] і Dürango та ін. [85].

У проведених експериментах цільове значення затримки setpoint встановлено на рівні 300 мс. Вибір цього значення ґрунтується на аналізі мережевих даних Ericsson та узгоджується з параметрами, використаними в роботах Klein та ін. [84] і Dürango та ін. [85]. Контролер Brownout порівнює поточне значення 95-го перцентиля з установленим setpoint. Якщо 95-й перцентиль перевищує 300 мс, це свідчить про погіршення часових характеристик обслуговування та необхідність зменшення ресурсомісткості виконуваних операцій. Якщо затримка залишається нижчою за цільове значення, система може підтримувати поточний рівень сервісу або, за наявності достатнього ресурсного резерву, поступово його підвищувати.

Таким чином, USP і 95-й перцентиль використовуються для різних цілей. USP є підсумковим показником якості обслуговування та показує частку запитів, виконаних у межах установленої часової вимоги. 95-й перцентиль використовується механізмом Brownout для поточного контролю затримки та прийняття рішення щодо зміни рівня сервісу.

Для аналізу роботи ресурсно-адаптивного балансування додатково враховується частка міжвузлового перенесення навантаження, або inter-edge offloading. Цей показник визначає частку запитів, які замість локального оброблення були передані на інший Edge-вузол. Він дає змогу оцінити, наскільки активно алгоритм RAP використовує ресурсний резерв інших вузлів системи.

Частка inter-edge offloading не розглядається як самостійний критерій ефективності. Збільшення кількості перенаправлених запитів не обов'язково означає покращення роботи системи. Якщо локальний вузол перевантажений, перенесення частини запитів на інші вузли може зменшити його завантаження та запобігти подальшому зростанню затримки. Водночас надмірне перенаправлення збільшує обсяг мережевого обміну між Edge-вузлами та може негативно впливати на час відповіді. Тому inter-edge offloading аналізується разом із пропускною здатністю, Generated Value, середньою затримкою та USP.

Перед проведенням основної серії експериментів виконано попереднє налаштування параметрів Brownout та алгоритму ресурсно-адаптивного балансування RAP. Частоту виконання контролера Brownout встановлено на рівні двох разів за секунду, що відповідає інтервалу 500

мс. Для аналізу поточного стану системи використовується контрольне вікно тривалістю 5 с.

У межах цього вікна накопичуються актуальні значення затримки обслуговування запитів. Після надходження нового вимірювання воно додається до поточної вибірки, а застарілі значення поступово вилучаються. На основі даних, що містяться в контрольному вікні, визначається 95-й перцентиль затримки, який порівнюється з установленим цільовим значенням 300 мс. Таким чином, рішення Brownout ґрунтується не на одиничному вимірюванні, а на сукупності останніх спостережень за поведінкою системи.

Параметри контрольного циклу визначено за результатами попередніх експериментів. Подальше скорочення інтервалу отримання даних із Prometheus призводило до пропусків окремих вимірювань та появи часових розривів у послідовностях метрик. Водночас надмірне збільшення тривалості контрольного вікна зменшувало б швидкість реакції механізму на зміну навантаження. Використання п’ятисекундного вікна забезпечує необхідний компроміс між стабільністю оцінювання поточного стану системи та швидкістю реакції контролера.

Для експериментів без використання Brownout встановлюється фіксований базовий режим Action 2, якому відповідає коефіцієнт цінності 1,0. У такій конфігурації рівень сервісу не змінюється залежно від поточного навантаження, а всі запити обробляються з однаковою ресурсомісткістю. Цей режим використовується як базовий для порівняння з адаптивною конфігурацією.

За активації Brownout рівень сервісу змінюється відповідно до поточного стану системи. У періоди високого навантаження збільшується частка запитів, які обробляються у спрощеному режимі Action 1. Такий режим потребує менше обчислювальних ресурсів і дає змогу зменшити навантаження на систему. За нормального стану переважає базовий режим Action 2. Якщо система має достатній ресурсний резерв, збільшується частка запитів, які обробляються в розширеному режимі Action 3. Цей режим потребує більше ресурсів, однак забезпечує вищу цінність результату.

У результаті використання Brownout створюється можливість адаптивно змінювати ресурсомісткість обслуговування залежно від поточ-

ного навантаження. У поєднанні з RAP це формує два взаємодоповнювальні механізми адаптації. RAP визначає можливість перенесення запиту на інший Edge-вузол за недостатності локальних ресурсів, тоді як Brownout змінює рівень сервісу та, відповідно, обчислювальну складність оброблення запитів.

Таким чином, оцінювання експериментальних результатів здійснюється комплексно. Пропускна здатність характеризує кількість успішно оброблених запитів, Generated Value враховує цінність отриманого результату залежно від рівня сервісу, середня затримка відображає часові витрати на оброблення, USP визначає частку запитів, виконаних у межах установленої часової вимоги, а inter-edge offloading характеризує інтенсивність перенесення запитів між Edge-вузлами. Спільний аналіз цих показників дає змогу оцінити вплив Brownout і RAP на продуктивність, часові характеристики та ефективність використання ресурсів розподіленої Edge-Cloud системи.

4.7. Дослідження ефективності адаптивного балансування навантаження в Edge-Cloud середовищі

Частоту виконання контрольного циклу алгоритмів балансування навантаження встановлено на рівні 500 мс. Аналогічний інтервал використовувався Klein та ін. [13] під час дослідження EPBH. Подальше скорочення цього інтервалу в експериментальному середовищі обмежувалося частотою отримання метрик із Prometheus, оскільки надмірна інтенсивність запитів могла спричиняти пропуски окремих вимірювань та часові розриви у послідовностях контрольованих показників. Для ресурсно-адаптивного алгоритму використано додатковий локальний цикл із частотою 4 рази за секунду. На відміну від основного циклу, він призначений для частішого оновлення інформації про власні ресурси Edge-вузла. Це дає змогу швидше враховувати короткочасні зміни локального навантаження без необхідності відповідного збільшення частоти повного циклу прийняття рішень. Порогове значення використання ресурсів для RAP встановлено на рівні 90 %. Вибір такого значення ґрунтується на результатах попередніх експе-

риментів та параметрах, використаних Nguyen та ін. [73]. Зменшення порога призводило до надмірного міжвузлового перенесення запитів, тоді як його збільшення підвищувало ймовірність перевантаження локального Edge-вузла.

Основними ресурсними показниками алгоритму є використання процесора та оперативної пам'яті. Водночас експериментальне навантаження сформовано як переважно CPU-інтенсивне, що дає змогу дослідити поведінку алгоритму за контрольованого дефіциту процесорних ресурсів. Відповідно вагові коефіцієнти RAP встановлено як $w_{RAM} = 1$, $w_{CPU} = 2$.

Отже, стан процесорних ресурсів має вдвічі більшу вагу під час формування інтегральної оцінки вузла порівняно з використанням оперативної пам'яті. Для контрольованого відтворення станів різного рівня завантаження встановлено ресурсні обмеження контейнерів Kubernetes. Контейнеру досліджуваного застосунку виділено 500 millicores CPU, що забезпечує досягнення граничного рівня використання процесорних ресурсів за інтенсивного потоку запитів. Проксі-контейнеру виділено одне процесорне ядро, щоб його ресурсні обмеження не впливали на виконання функцій контролера. Ліміти оперативної пам'яті для обох контейнерів встановлено на рівні, достатньому для виключення RAM як основного чинника перевантаження в цій серії експериментів.

Для дослідження поведінки алгоритмів використано три рівні сумарного навантаження – 48, 60 та 72 одночасні користувачі. Навантаження 48 користувачів відповідає базовому режиму, за якого стандартний RAP переважно функціонує без вираженого перевантаження. Сценарії з 60 та 72 користувачами формують вищі рівні завантаження і використовуються для оцінювання здатності алгоритмів реагувати на дефіцит локальних ресурсів.

Для кожного рівня навантаження досліджено рівномірний і нерівномірний розподіл користувачів між чотирма Edge-вузлами. У першому випадку використовується співвідношення X:X:X:X, у другому навантаження одного з вузлів збільшується втричі та формується співвідношення 3X:X:X:X. Використані експериментальні сценарії наведено в табл. 4.4. Наприклад, для загальної кількості 48 користувачів рівномірний розподіл становить 12:12:12:12, а нерівномірний 24:8:8:8.

Таблиця 4.4

Сценарії розподілу користувацького навантаження між Edge-вузлами

Сценарій	Загальна кількість користувачів	Node 1	Node 2	Node 3	Node 4	Схема розподілу
1	48	12	12	12	12	X:X:X:X
2	60	15	15	15	15	X:X:X:X
3	72	18	18	18	18	X:X:X:X
4	48	24	8	8	8	3X:X:X:X
5	60	30	10	10	10	3X:X:X:X
6	72	36	12	12	12	3X:X:X:X

Такий набір сценаріїв дає змогу дослідити алгоритми як за симетричного розподілу навантаження, так і за локального перевантаження одного Edge-вузла. Порівняння результатів для 48, 60 і 72 користувачів забезпечує оцінювання поведінки системи за послідовного переходу від базового режиму до більш інтенсивного навантаження, а сценарії 3X:X:X:X дають змогу окремо оцінити ефективність механізму міжвузлового перенесення запитів.

Для оцінювання ефективності розглянутих методів адаптивного керування навантаженням проведено серію експериментальних досліджень, спрямованих на порівняння базового ресурсно-адаптивного алгоритму RAP та його модифікованого варіанта з інтегрованим механізмом brownout. Основну увагу зосереджено на визначенні впливу адаптивної зміни рівня обслуговування на пропускну здатність системи, згенеровану цінність, часові характеристики обробки запитів та інтенсивність їх перенесення між Edge-вузлами.

Експериментальне оцінювання виконувалося для Edge-Cloud системи, сформованої з чотирьох робочих вузлів відповідно до архітектури та параметрів середовища, визначених у попередніх підрозділах. Для підвищення достовірності отриманих результатів кожний сценарій навантаження виконувався повторно, після чого розраховувалися усереднені значення досліджуваних показників. Середня затримка визначалася на основі часу обслуговування запитів усіма вузлами протягом відповідного тестового інтервалу.

Порівняльне дослідження проводилося для сформованих у підрозділі 4.7 сценаріїв, які передбачали зміну загальної кількості одночасних користувачів та характеру розподілу вхідного потоку між Edge-вузлами. Розглядалися навантаження, сформовані 48, 60 і 72 одночасними користувачами. Для кожного рівня досліджувалися два варіанти розподілу: рівномірний X:X:X:X та нерівномірний 3X:X:X:X, за якого один із вузлів отримував утричі більшу частку вхідного потоку. Такий набір сценаріїв дає змогу дослідити поведінку алгоритмів як за поступового зростання загального навантаження, так і за виникнення локального перевантаження окремого Edge-вузла.

Принципова відмінність між досліджуваними варіантами полягає у способі адаптації системи до зміни доступних обчислювальних ресурсів. У базовому варіанті RAP застосунок функціонує з фіксованим рівнем сервісу та використовує Action 2, для якої коефіцієнт згенерованої цінності дорівнює одиниці. За таких умов кількість успішно оброблених запитів за одиницю часу чисельно відповідає згенерованій цінності. У модифікованому варіанті RAP+brownout рівень сервісу може адаптивно змінюватися відповідно до поточного стану ресурсів, тому пропускна здатність і згенерована цінність характеризують різні аспекти функціонування системи.

Порівняння пропускної здатності та згенерованої цінності для досліджуваних алгоритмів за різних сценаріїв навантаження наведено на рис. 4.10.

Аналіз отриманих результатів показує, що інтеграція механізму brownout забезпечує підвищення згенерованої цінності для всіх досліджуваних сценаріїв. У більшості випадків позитивний ефект спостерігається також щодо пропускної здатності системи. Винятком є сценарій із 48 одночасними користувачами за рівномірного розподілу X:X:X:X, для якого базовий RAP забезпечує дещо вищу пропускну здатність. Це пояснюється тим, що за відносно невисокого та рівномірного навантаження доступних ресурсів Edge-вузлів достатньо для обробки запитів без додаткової адаптації рівня сервісу.

Зі збільшенням кількості користувачів характер співвідношення між пропускною здатністю та згенерованою цінністю змінюється. За навантаження 48 одночасних користувачів у конфігурації RAP+brownout

пропускна здатність є нижчою за згенеровану цінність. Це пов'язано з можливістю виконання запитів із підвищеним рівнем сервісу, коли доступний запас ресурсів дає змогу використовувати більш ресурсомісткі функції застосунку.

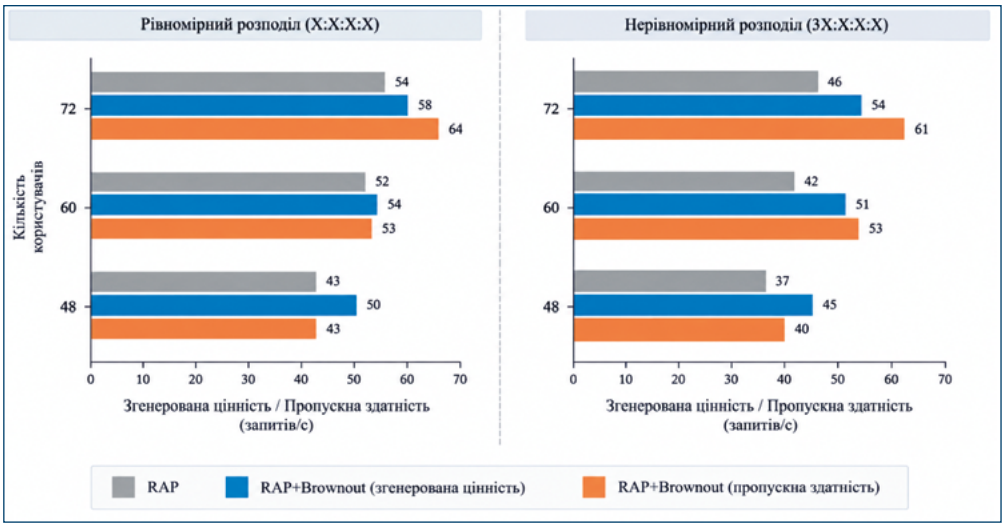


Рис. 4.10. Порівняння пропускної здатності та згенерованої цінності для RAP і RAP+brownout за різних сценаріїв навантаження

За подальшого зростання навантаження механізм brownout адаптує рівень обслуговування до фактичного стану обчислювальних ресурсів. Система частіше переходить до менш ресурсомістких варіантів виконання запитів, що зменшує обчислювальні витрати на їх обробку та дає змогу обслужити більшу кількість запитів за одиницю часу. У результаті за високого навантаження пропускна здатність може перевищувати показник згенерованої цінності. Таким чином, зміна співвідношення між цими показниками відображає адаптацію системи до дефіциту доступних ресурсів.

Особливий інтерес становлять результати для нерівномірного розподілу навантаження 3X:X:X:X. За такого сценарію один Edge-вузол отримує втричі більший потік запитів порівняно з кожним із трьох інших вузлів. Це створює локальне ресурсне перевантаження навіть тоді, коли в межах усієї системи ще наявний певний запас обчислювальної потужності. Саме за таких умов найбільш виразно проявляється відмінність між базовим RAP і його модифікацією з brownout.

За навантаження 48 одночасних користувачів застосування RAP+ brownout забезпечує приблизно на 15 % більшу згенеровану цінність за рівномірного розподілу вхідного потоку. Для нерівномірної схеми 3X:X:X:X перевага збільшується приблизно до 20 %. Отже, ефективність поєднання ресурсно-адаптивного балансування з динамічною зміною рівня сервісу особливо помітна за нерівномірного завантаження Edge-вузлів.

Такий результат пояснюється тим, що механізм brownout доповнює міжвузловий розподіл запитів можливістю адаптувати складність їх локальної обробки. У разі наближення вузла до стану перевантаження система може тимчасово скоротити виконання необов'язкових ресурсомістких операцій. Унаслідок цього зменшується обсяг обчислювальних ресурсів, необхідних для обробки одного запиту, а вузол отримує можливість обслужити більшу частину вхідного потоку без негайного перенесення запитів на інші вузли.

Отже, RAP+brownout фактично реалізує два взаємопов'язані механізми адаптації. Перший полягає у перерозподілі навантаження між Edge-вузлами з урахуванням стану їхніх ресурсів, тоді як другий передбачає зміну ресурсомісткості локального обслуговування шляхом вибору відповідної дії brownout. Поєднання цих механізмів розширює можливості системи щодо реагування на динамічні та нерівномірні зміни навантаження.

Водночас пропускна здатність не може розглядатися як єдиний критерій ефективності адаптивного балансування. Для Edge-Cloud систем важливим є не лише збільшення кількості оброблених запитів, але й дотримання заданих часових характеристик їх обслуговування. Тому наступним етапом експериментального оцінювання є аналіз середньої затримки та показника USP, який характеризує частку запитів, виконаних швидше за встановлене цільове значення 300 мс.

Результати порівняння часових характеристик базового RAP та RAP+brownout наведено на рис. 4.11.

За невисокого рівня навантаження відмінності між досліджуваними підходами є незначними. Зокрема, для 48 одночасних користувачів за рівномірного розподілу запитів базовий RAP демонструє кращі результати як за пропускною здатністю, так і за середньою затримкою. За та-

ких умов обчислювальних ресурсів вузлів достатньо для обслуговування сформованого потоку, тому додаткова адаптація рівня сервісу не забезпечує суттєвого виграшу.

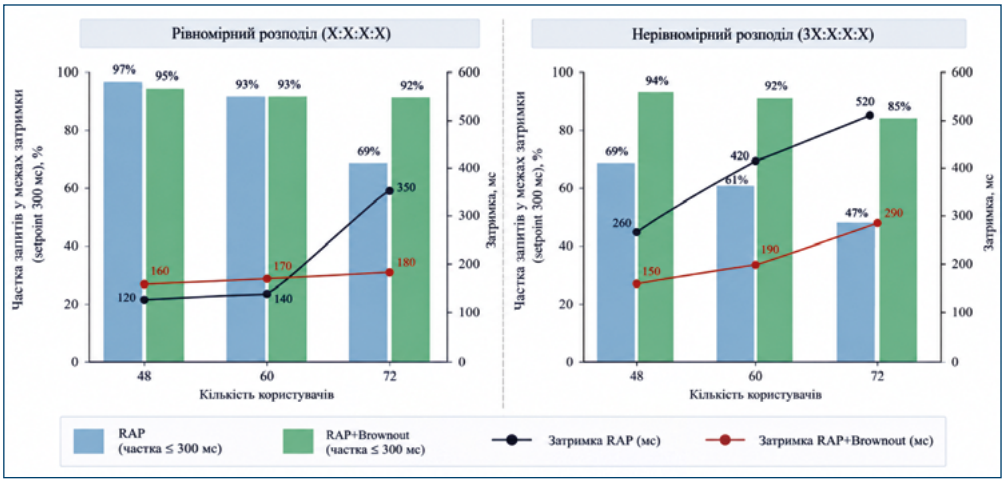


Рис. 4.11. Порівняння середньої затримки та частки запитів у межах цільового значення для RAP і RAP+brownout

Зі збільшенням кількості одночасних користувачів поведінка базового RAP погіршується. Зростає середній час обслуговування запитів, а частка запитів, виконаних у межах заданого порога затримки, зменшується. Це свідчить про поступове наближення окремих вузлів до стану ресурсного насичення. За таких умов одного лише механізму перенесення частини запитів на інші Edge-вузли недостатньо для стабільного підтримання заданих часових характеристик.

Інша поведінка спостерігається для RAP+brownout. За підвищеного навантаження адаптивна зміна рівня сервісу дає змогу зменшити обчислювальні витрати на обробку окремих запитів і тим самим стримувати зростання затримки. У досліджених сценаріях середня затримка залишалася нижчою за встановлене цільове значення 300 мс, а частка запитів, оброблених у межах цього порога, зберігалася на високому рівні.

Отримані результати свідчать, що brownout виконує функцію додаткового адаптивного механізму, який активується за наближення системи до стану перевантаження. Замість негайного перенесення значної частини запитів на інші вузли система спочатку може зменшити ресур-

сумісткість їх локальної обробки. Це особливо важливо для Edge-Cloud середовищ, де міжвузлова передача запитів може супроводжуватися додатковою мережевою затримкою.

Окремим показником, який характеризує поведінку алгоритму балансування, є частка запитів, перенесених між Edge-вузлами. Міжвузлове перенесення дає змогу використовувати вільні ресурси сусідніх вузлів, однак його надмірна інтенсивність може збільшувати мережевий трафік і час обслуговування. У досліджуваній реалізації RAP рішення щодо перенесення запиту приймається у разі перевищення встановленого порогового рівня використання ресурсів локального вузла.

Результати порівняння інтенсивності міжвузлового перенесення запитів наведено на рис. 4.12.

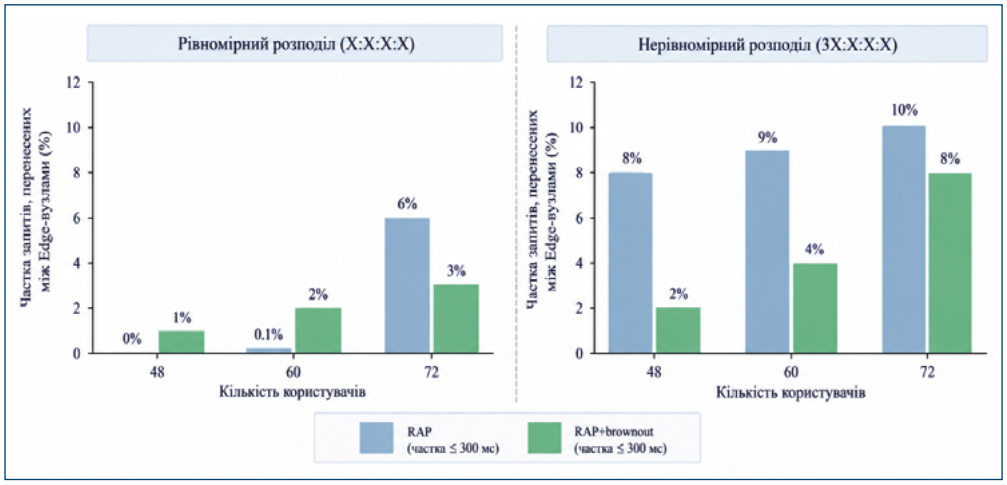


Рис. 4.12. Частка запитів, перенесених між Edge-вузлами за використання RAP і RAP+brownout

Аналіз результатів показує, що в більшості досліджуваних сценаріїв використання RAP+brownout зменшує потребу в перенесенні запитів на сусідні вузли. Така поведінка безпосередньо пов'язана зі здатністю механізму brownout змінювати ресурсомісткість локального обслуговування. За зростання навантаження вузол може перейти до менш ресурсомісткої дії та продовжити локальну обробку більшої частини вхідних запитів, не досягаючи порогового рівня використання ресурсів.

Водночас ця залежність не є однаково вираженою для всіх режимів. Для сценаріїв із 48 та 60 одночасними користувачами за рівномірного розподілу навантаження різниця між алгоритмами щодо частки перенесених запитів є менш однозначною. За таких умов ресурси розподіляються між вузлами відносно рівномірно, тому потреба у додатковій адаптації та міжвузловому перенесенні загалом є меншою.

Найбільш показовими є результати для схеми 3X:X:X:X. Вузол, який отримує втричі більший потік запитів, швидше наближається до встановленого порога використання ресурсів. Для базового RAP це призводить до активнішого перенесення запитів на сусідні вузли. У конфігурації RAP+brownout частина навантаження може бути компенсована локально шляхом переходу до менш ресурсомісткого режиму обслуговування. У результаті зменшується кількість запитів, які необхідно передавати іншим Edge-вузлам. Зниження інтенсивності міжвузлового перенесення має важливе значення для розподілених інформаційно-комунікаційних систем. З одного боку, воно зменшує додаткове навантаження на мережеву інфраструктуру, а з іншого – скорочує вплив міжвузлової затримки на загальний час обслуговування. Тому ефективність RAP+brownout проявляється не лише у збільшенні кількості оброблених запитів, але й у більш раціональному використанні локальних обчислювальних ресурсів Edge-вузлів.

Проведене експериментальне дослідження підтверджує доцільність поєднання ресурсно-адаптивного алгоритму RAP із механізмом brownout. На відміну від базового RAP, у якому основним способом реагування на локальне перевантаження є перенесення частини запитів на інші вузли, модифікований підхід додатково передбачає адаптацію рівня сервісу відповідно до поточного ресурсного стану. Це забезпечує більш гнучке керування навантаженням та розширює діапазон умов, у яких система може підтримувати необхідні характеристики обслуговування. Разом із тим застосування brownout передбачає певний компроміс між повнотою функціональності сервісу та ефективністю використання ресурсів. Перехід до менш ресурсомісткої дії може супроводжуватися тимчасовим обмеженням необов’язкових функцій застосунку. Тому оцінювання такого підходу доцільно здійснювати не

лише за кількістю успішно оброблених запитів, а з урахуванням згенерованої цінності та часових характеристик їх обслуговування. У цьому контексті згенерована цінність є узагальненим показником, який дає змогу враховувати відмінності між рівнями сервісу. Максимізація лише пропускної здатності могла б призводити до переважного використання найменш ресурсомістких дій навіть тоді, коли система має достатній запас ресурсів для забезпечення повнішого рівня обслуговування. Використання згенерованої цінності дає змогу оцінити не тільки кількість виконаних запитів, але й рівень сервісу, з яким вони були оброблені. Отже, ефективність адаптивного балансування в досліджуваній Edge-Cloud системі визначається сукупністю взаємопов'язаних характеристик: пропускною здатністю, згенерованою цінністю, середньою затримкою, часткою запитів у межах заданого порога та інтенсивністю міжвузлового перенесення. Комплексне використання цих показників дає змогу коректніше оцінити поведінку алгоритмів за різних режимів функціонування системи. Результати експериментів показують, що переваги RAP+brownout найбільш виразно проявляються зі зростанням інтенсивності та нерівномірності вхідного навантаження. За нормального режиму функціонування відмінність між базовим та модифікованим алгоритмами може бути незначною, тоді як за наближення окремих Edge-вузлів до стану перевантаження додатковий механізм адаптації рівня сервісу забезпечує суттєвіший ефект.

Таким чином, інтеграція механізму brownout із ресурсно-адаптивним балансуванням дає змогу одночасно враховувати стан обчислювальних ресурсів, характер розподілу навантаження та допустимий рівень обслуговування. Такий підхід забезпечує більш стійке функціонування системи за динамічних змін вхідного потоку, зменшує необхідність надмірного перенесення запитів між вузлами та сприяє підтриманню заданих часових характеристик. Отримані результати підтверджують перспективність комбінованого адаптивного керування навантаженням для побудови розподілених інформаційно-комунікаційних систем на основі Edge-Cloud архітектури.

Балансування навантаження у запропонованому підході здійснюється на декількох рівнях. На локальному Edge-вузлі механізм Brownout адаптує рівень сервісу відповідно до поточного наванта-

ження. Якщо ресурсів локального вузла недостатньо, RAP забезпечує перенаправлення частини запитів на інші Edge-вузли з доступним ресурсним резервом. У випадку, коли можливості Edge-рівня вичерпано, Cloud-рівень може використовуватися як додатковий обчислювальний ресурс для оброблення надлишкового навантаження. Така організація забезпечує послідовне використання локальних, розподілених Edge та хмарних ресурсів залежно від поточного стану системи.

РОЗДІЛ V

МОДЕЛЬ ТА МЕТОД ІНТЕЛЕКТУАЛЬНОГО УПРАВЛІННЯ РОЗМІЩЕННЯМ МІКРОСЕРВІСІВ У РОЗПОДІЛЕНИХ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ СИСТЕМАХ

5.1. Концептуальна модель інтелектуального управління розміщенням мікросервісів у розподілених інформаційно-комунікаційних системах

Розглянуті у попередньому розділі методи адаптивного управління навантаженням дають змогу реагувати на поточні зміни стану Edge-вузлів шляхом перерозподілу запитів та зміни рівня їх обслуговування. Водночас для ефективного функціонування РІКС важливо не лише реагувати на вже наявні зміни навантаження, а й завчасно визначати можливе погіршення стану окремих вузлів. Особливого значення це набуває у мікросервісних системах, де загальна продуктивність залежить від місця розгортання окремих сервісів, інтенсивності їх взаємодії та стану мережевих і обчислювальних ресурсів. Неоптимальне розміщення взаємопов'язаних мікросервісів може призводити до збільшення міжсервісних затримок, обсягу мережевого трафіку та витрат обчислювальних ресурсів. Тому наступним напрямом удосконалення механізмів управління РІКС є використання прогнозної інформації про стан вузлів під час прийняття рішень щодо розміщення мікросервісів. На відміну від підходів, що враховують лише поточний стан ресурсів, прогнозне управління дає змогу завчасно оцінити можливість перевантаження вузла та підготувати відповідні керуючі дії до суттєвого погіршення характеристик обслуговування. Для формування прогнозу доцільно використовувати накопичену інформацію про функціонування системи, зокрема системні журнали, дані NetFlow і SNMP, логи контейнерів Docker/Kubernetes та телеметрію SDN-контролера. На основі цих даних можна оцінювати динаміку використання CPU, оперативної пам'яті, пропускної здатності каналів, затримки та інших показників, що характеризують стан вузлів.

У цьому розділі розглянуто методи інтелектуального управління розміщенням мікросервісів у РІКС, які поєднують аналіз логів і телеметрії, прогнозування стану вузлів та оцінювання спорідненості мікросервісів. На основі отриманої інформації система може завчасно визначати вузли з ризиком перевантаження та приймати рішення щодо розміщення нових екземплярів мікросервісів, перерозподілу навантаження, масштабування або міграції вже розгорнутих компонентів. Такий підхід забезпечує перехід від управління на основі лише поточного стану системи до прогнозного прийняття рішень щодо розміщення мікросервісів.

Оскільки кожен сервіс працює незалежно, важливо мінімізувати міжсервісні затримки, оптимізувати маршрутизацію запитів та зменшувати обсяг мережевого трафіку. Недостатньо оптимізоване управління взаємодією між мікросервісами може призвести до значного зниження продуктивності системи, збільшення часу відповіді сервісів та підвищених витрат на інфраструктуру. Важливим аспектом оптимізації є визначення ступеня спорідненості між мікросервісами, що дозволяє покращити їхню організацію та розподіл. Такий параметр взаємозалежності особливо важливий в контексті відокремлення мікросервісних додатків у кластери або ж забезпечення мінімізації затримки шляхом їх розгортання за критерієм мінімальної близькості до кінцевого користувача на Edge-вузлах. Дослідження [87-90] показують, що спорідненість між сервісами можна визначити на основі аналізу мережевого трафіку, логів взаємодії, затримок між сервісами та частоти викликів. Це дозволяє ідентифікувати сервіси, які тісно взаємодіють між собою, і, відповідно, групувати їх для зменшення затримок та покращення продуктивності системи. Крім того, ефективна кластеризація мікросервісів сприяє оптимальному використанню обчислювальних ресурсів, зменшенню міжмережевого трафіку та підвищенню надійності розподілених систем.

Зі збільшенням кількості мікросервісів у системі управління нею стає дедалі складнішим. Виклики включають оркестрацію сервісів, динамічне балансування навантаження, моніторинг та управління збоями. Це стає особливо важливим в умовах сталого розвитку та функціонування адаптивної мережевої інфраструктури, де про-

блема автоматизації розміщення мікросервісів та адаптації їх робочих процесів до змінних умов та постійних міграцій виходить на перший план.

Саме тому необхідно розробити механізм автоматизації розміщення мікросервісів та адаптації їх робочих процесів до змінних умов шляхом аналізу спорідненості мікросервісів, що дасть змогу підвищити ефективність їх управління та забезпечити підтримку необхідного рівня QoS/QoE, особливо в умовах мінімізації затримок та обмеженості обчислювальних ресурсів на крайових вузлах мережі [91-93]. З постійним розвитком технологій та змінними умовами експлуатації інфраструктури, зокрема у контексті Edge Computing та хмарних середовищ, стає необхідним забезпечити сталість та адаптивність системи. Автоматизація дозволяє не тільки налаштовувати розгортання мікросервісів відповідно до наявних ресурсів і вимог, але й забезпечити безперервну адаптацію до нових умов, забезпечуючи стабільне функціонування системи в умовах постійних змін. Загалом, автоматизація процесу розгортання мікросервісних додатків із підтримкою QoS/QoE є необхідною для забезпечення ефективності, надійності та масштабованості сучасних розподілених систем. Вона дозволяє зменшити складність управління, підвищити продуктивність і стабільність, а також забезпечити високий рівень QoE навіть у складних та динамічних умовах [94, 95].

У цьому контексті активно застосовуються підходи CI/CD (Continuous Integration / Continuous Delivery), які дозволяють автоматизувати процеси тестування, збірки та оновлення мікросервісів. Інтеграція з DevOps-практиками сприяє узгодженій взаємодії між командами розробки та експлуатації, пришвидшуючи доставку оновлень і мінімізуючи людські помилки. Одним із ключових технологічних компонентів CI/CD-пайплайнів є Docker – система контейнеризації, яка забезпечує ізольоване та повторюване середовище для розгортання мікросервісів. Проте, попри всі переваги, ці підходи мають і певні обмеження. CI/CD-процеси вимагають детального налаштування, ретельного моніторингу та підтримки. У великих мікросервісних системах виникають труднощі із узгодженістю оновлень, управлінням залежностями та тестуванням інтеграційних сценаріїв.

Крім того, Docker-контейнери, хоча й спрощують масштабування і портативність, не вирішують проблему оптимального розміщення сервісів у реальному середовищі з обмеженими ресурсами (наприклад, на Edge-вузлах), де ключовими є затримки, пропускна здатність і локальна доступність даних. Сам по собі Docker не враховує мережеву топологію, спорідненість між сервісами чи їх географічну близькість, що обмежує ефективність при розгортанні в розподілених середовищах із динамічними вимогами.

Автори робіт [96-99] використовують різні стратегії розгортання, щоб спробувати зменшити час відповіді (тобто суму часу обробки кожної служби та часу передачі даних). У той же час Yin, L. [99] з колегами, розглядаючи роль контейнерів, представили нову модель планування завдань. Вони оптимізували кількість одночасних завдань у вузлах туману та запропонували механізм перерозподілу, щоб зменшити затримку завдань, і збільшити паралелізм завдань у вузлах туману. X. Li [100] та інші запропонували структуру гібридних обчислень і розробили стратегію інтелектуального планування ресурсів. На основі жадібної стратегії та порогової стратегії з обмеженнями затримки вони розробили двофазний алгоритм для планування обчислювальних ресурсів на крайньому рівні, що покращує продуктивність системи в реальному часі, задоволення та енергоефективність. Однак ці існуючі методи мають обмеження в розгляді динамічних впливів міжвузлових затримок зв'язку та мінливого двонаправленого трафіку між вихідними та низхідними мікросервісами з кількома репліками. Невирішиним і залишається питання автоматизованого управління, що пришвидшить процес розгортання реплік.

Визначення ступеня спорідненості між мікросервісами на основі аналізу мережевого трафіку, логів взаємодії, затримок та частоти викликів є важливою темою в сучасних дослідженнях мікросервісних архітектур. Shuilang He та ін. [101] пропонують стратегію розгортання мікросервісів, яка оптимізує їхнє розміщення, аналізуючи коефіцієнт взаємодії між сервісами. Однак їх стратегія не гнучка до постійних змін у навантаженні. Це впливає на точність визначення взаємодії між сервісами, що є важливим у контексті функціонування та масштабування великих розподілених систем. Автори роботи [102]

дещо інакше підходять до цього питання. Вони пропонують систему TraDE, яка забезпечує адаптивне перепланування контейнерів мікросервісів відповідно до змін у мережі. Це дозволяє підтримувати бажану продуктивність всередині кластеру, автоматично змінюючи розподіл мікросервісів на основі поточних змін. Запропонований механізм адаптації вимагає постійного моніторингу умов середовища, що дозволяє швидко реагувати на зміни у навантаженні та ресурсах. Однак він залежить від зовнішніх змін у мережі, що може збільшити навантаження на інфраструктуру через часті зміни в розміщенні мікросервісів. Враховуючи досягнення науковців у вирішенні проблеми управління автоматизованим розгортанням мікросервісних додатків із підтримкою QOS/QOE пропонуємо свій варіант вирішення. Для ефективного та сталого розвитку розподілених інфокомунікаційних мереж пропонуємо вносити зміни лише в систему управління мережею, яка у більшості таких мереж реалізується за допомогою SDN-контролерів.

На сьогоднішній день важко уявити систему, яка працює з мікросервісними додатками без програмного управління. Передавання, управління та збір статистичної інформації здійснюється SDN-контролером шляхом пошуку записів у групових таблицях потоків від одного або/до декількох інтерфейсів чи додатків. Коли мова іде про мікросервісні додатки, контролер здійснює перенаправлення потоків запитів до необхідної аплікації. В такому випадку саме контролер буде відповідальним за розгортання цих додатків.

На рис. 5.1 представлена концептуальна модель розробленої архітектури управління мережею, яка поєднує принципи SDN з аналітикою логів для адаптивної оптимізації розподілу трафіку, розгортання сервісів та управління ресурсами. Основною метою архітектури є динамічне балансування навантаження, зменшення затримок у Cloud-Edge-Fog мережах та автоматизація процесу міграції мікросервісів з урахуванням їх спорідненості та географічної близькості до користувачів.

Центральне місце у структурі посідає площина управління SDN, що приймає запити на перенаправлення потоків, аналізує стан мережі та координує взаємодію між обчислювальними рівнями (Cloud, Fog, Edge). Важливим компонентом архітектури є підсистема аналізу логів [103], яка

збирає дані з SDN-контролерів і систем моніторингу, здійснюючи їх обробку за допомогою алгоритмів машинного навчання. Отримані закономірності дозволяють прогнозувати зміни навантаження, виявляти аномалії у маршрутизації та оптимізувати перерозподіл ресурсів. У зв'язку з цим пропонується метод, який поєднує семантичну обробку неструктурованих логів, прогнозування стану вузлів та інтегральну оцінку стану працездатності вузлів, що дозволяє адаптивно управляти ресурсами та розгортанням мікросервісів. На основі результатів лог-аналітики площина управління коригує розподіл кешу, приймає рішення про масштабування мікросервісів або перенесення їхніх компонентів ближче до користувача.

Окрему увагу приділено управлінню мережею на рівні сервісів, яке включає механізми кешування контенту, управління мобільністю та резервування ресурсів. Виділено модуль підтримки якості обслуговування (QoE), що контролює затримки, керує пріоритетами трафіку та забезпечує відповідність параметрам QoS (Quality of Service).

Блок «Управління мобільністю» в межах даного рівня слідкуватиме за розгортанням компонентів мікросервісного додатку за критерієм географічної близькості до користувача, моніторитиме розташування даного додатку в межах кластеру і час його перебування.

При цьому блок «Резервування ресурсів» буде відповідати за контроль і аналіз наявних вільних програмно-апаратних ресурсів, і по-суті, виконуватиме роль Resource discoverer`a, (так як це у описано у [104]) включаючи випадки розгортання функцій NFV. Його завдання полягає в оцінці доступних фізичних, віртуальних, апаратних і телекомунікаційних ресурсів. Аналізуючи пропускну здатність каналів та доступність компонентів сервісу, він ухвалює рішення щодо надання послуги або перенесення її компонентів на менш завантажений сервер. Для цього він надсилає запити до Площини управління, яка визначає необхідність міграції компонентів для балансування навантаження та оптимального використання ресурсів.

У нижній частині зображення розташовані віртуалізовані мережеві ресурси, спектральне віртуалізоване управління та інші рівні абстракції, що забезпечують ефективний розподіл навантаження та автоматичну оптимізацію розгортання мікросервісних додатків.

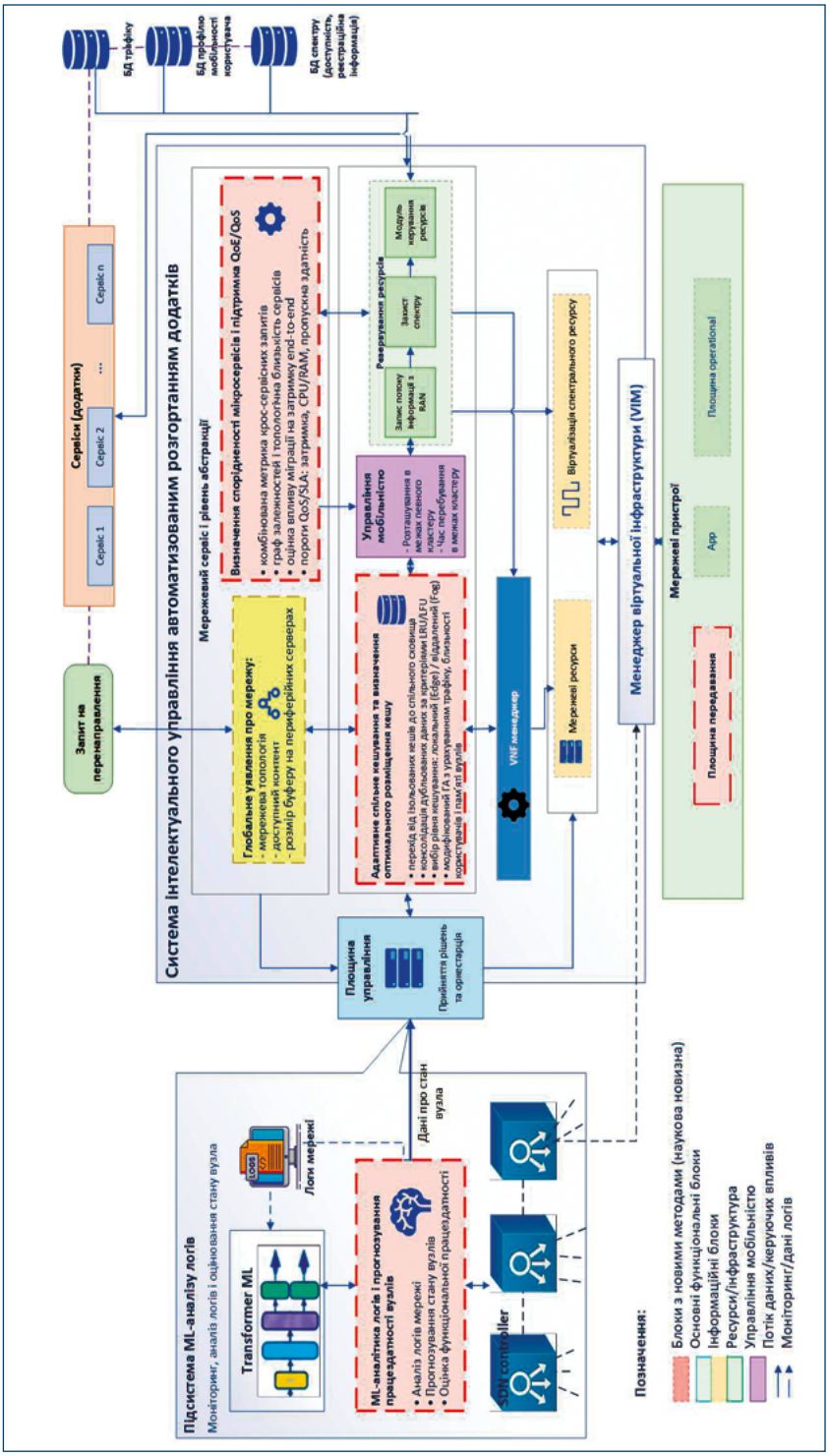


Рис. 5.1. Концептуальна архітектура системи інтелектуального управління розгортанням мікросервісних додатків з прогнозуванням працездатності вузлів

Коли користувач ініціює запит на доступ до мікросервісу (наприклад, викликає API або відкриває вебдодаток), цей запит надходить до пограничних шлюзів та балансувальник навантаження. На цьому етапі аналізується географічне місце розташування користувача, а також визначається найближчий Edge-вузол, здатний обробити запит із мінімальною затримкою і надати цей сервіс. Вся ця інформація надходить на Площину управління у вигляді лог-файлів і записується у відповідну Підсистему. Площина управління перевіряє чи відповідний мікросервіс уже запущений чи ні, звіряючи інформацію за допомогою таблиць потоків. Якщо відповідний мікросервіс уже запущений на оптимальному Edge-вузлі, запит перенаправляється безпосередньо на нього. В іншому випадку запит спрямовується на Рівень керування, де перевіряється можливість додаткового розгортанння. При цьому необхідні для користувача параметри якості записуються у Модуль підтримки якості обслуговування та QoE. Після отримання запиту система управління аналізує низку параметрів для вибору оптимального Edge-вузла: фізична відстань до користувача (на основі інформації отриманої із блоку «Глобальне уявлення про мережу»); затримка мережі між вузлом і клієнтом (на основі інформації про рівень QoS); поточне навантаження на обчислювальні ресурси (CPU, RAM, GPU); резервні обчислювальні потужності для масштабування; наявність уже розгорнутих екземплярів мікросервісу.

На основі цих даних система приймає рішення щодо запуску мікросервісу на конкретному Edge-вузлі. Якщо вузол відповідає всім критеріям, він отримує команду на автоматичне розгортання сервісу. Після успішного розгортання мікросервісу система балансування навантаження оновлює свої конфігурації, додаючи новий вузол до таблиці потоків, а система моніторингу починає збирати статистику щодо його продуктивності та стабільності роботи вузлів мережі.

Оскільки ресурси Edge-вузлів можуть бути і є обмеженими (пам’ять, потужність процесора, пропускна здатність), важливо розподіляти навантаження між мікросервісами так, щоб мінімізувати затримки та обсяг трафіку, який передається. Відповідно це потребує ефективнішого функціонування Рівня керування запропонованої моделі архітектури системи інтелектуального управління, та дасть можливість оптимізувати вико-

ристання мережевих і обчислювальних ресурсів та підвищити продуктивність, гнучкість і масштабованість усієї інфраструктури. Проте проактивна робота цього рівня вимагає точного математичного оцінювання поточного та прогнозного стану кожного обчислювального елемента.

5.2. Метод прогнозування стану працездатності вузлів на основі аналізу логів

На першому етапі здійснюється збір і попередня обробка логів. Логи надходять із різних джерел: системних журналів (syslog), потоків NetFlow, SNMP-запитів, логів контейнерів Docker/Kubernetes і телеметрії SDN-контролера. Вони проходять етап нормалізації, що включає видалення дублікатів, приведення часових міток до єдиного формату та класифікацію подій за типами, зокрема CPU, RAM, пропускною здатністю, затримкою та I/O. Подальше фільтрування дозволяє виокремити найбільш інформативні записи, що впливають на завантаженість вузлів.

Для кожного запису логів формується векторне представлення, яке інтегрує текстові та числові компоненти. Текстові повідомлення перетворюються у вектори за допомогою сучасних методів embedding (Word2Vec, BERT або Transformer), тоді як числові метрики нормалізуються відносно історичних значень. Додатково кодуються часові характеристики записів, що дає змогу враховувати сезонні або періодичні закономірності у завантаженості вузлів.

Нехай маємо $L_i = \{l_1, l_2, \ldots l_n\}$ множину логів вузла i. Кожен запис логу l_i, який потрапляє і зберігається в Підсистемі ML-аналізу логів формується з текстової частини (дані про мікросервіс) tx_i, мережевих параметрів вузла NPN_i (CPU, RAM, пропускна здатність, затримка тощо) та часової мітки tx_i – дані про час запису події: $l_i = (tx_i, NPN_i, t_i) \in R^m$

Для роботи із блоком ML переведемо формат логу у векторну форму. Для цього за допомогою embedding-функції φ текстову частину логу представимо як функцію $f_{tx_i} = \varphi(tx_i) \in R^d$, де d – розмірність векторного простору; інформацію про мережеві параметри вузли нормалізуємо відносно середнього μ_i та стандартного відхилення σ_i показників для даного вузла $f_{NPN_i} = \frac{NPN_i - \mu_i}{\sigma_i} \in R^m$; часова складова кодується як числовий

параметр або синусоїдальна трансформація для відображення циклічності доби

$$P - f_{t_i} = \left[\sin\left(\frac{2\pi t_i}{P}\right), \cos\left(\frac{2\pi t_i}{P}\right)\right] \in R^2.$$

Об'єднання цих компонентів формує векторне представлення лог запису X_i = [f_{tx_i} , f_{NPN_i} , f_{t_i}], що дозволить об'єднати семантичну інформацію та кількісні показники вузла в єдиному векторному просторі для подальшого аналізу.

На другому етапі на основі сформованих векторів будується прогноз стану вузла i на наступні моменти часу $(t + k)$:

$$\hat{S}_i(t + k) = F_o(X_1, X_2, \quad , X_i), \tag{5.1}$$

де F_o – модель машинного навчання, яка використовується для прогнозування в даній підсистемі. Для систем, які об'єднують функціонування Cloud – Edge –Fog рівнів найчастіше використовують або рекурентні нейронні моделі, наприклад, LSTM, або моделі Transformer. LSTM дозволяє враховувати короткострокові залежності, Transformer – довгострокові послідовності подій, що важливо для динамічних Cloud–Edge–Fog середовищ. Для прогнозування стану вузлів, на основі історії використання ресурсів мережевих вузлів кожного із рівнів, де розгортаються мікросервісні компоненти додатків, використаємо модель Transformer.

На третьому етапі кожну прогнозовану метрику нормуємо відносно її максимально допустимого для даного вузла значення:

$$S_{i,j}(t+k) = \frac{\hat{S}_{i,j}(t+k)}{S_{i,j}^{\max}}, S_{i,j}(t+k) \in [0;1] \tag{5.2}$$

де $S_{i,j}$ $(t + k)$ – прогнозоване значення j-ї метрики вузла i на момент часу; $S_{i,j}^{\max}$ – максимально допустиме значення цієї метрики (номінальна потужність процесора, обсяг оперативної пам'яті, пропускна здатність каналу тощо).

Інтегральне прогнозоване завантаження вузла визначається як зважена згортка нормованих метрик:

$$L_i(t+k) = \sum_j w_j S_{i,j}(t+k), \sum_j w_j = 1, \tag{5.3}$$

де w_j – ваговий коефіцієнт j-ї метрики, який відображає її внесок у загальне навантаження вузла. Умова нормування вагових коефіцієнтів гарантує, що $L_i = (t + k) \in [0;1]$.

Окремим фактором ризику є нетиповість поведінки метрик вузла, яка не зводиться до рівня їх завантаження. Відповідний показник ано-

мальності, тобто показник відхилення реальних значень метрик від спрогнозованих, можна визначити як:

$$A_i(t) = \sigma\left(\alpha\left(\Delta\hat{S}_i(t) - \Delta\hat{S}_0(t)\right)\right), A_i(t) \in (0;1), \quad (5.4)$$

де $\Delta\hat{S}_i(t)$ – різниця (евклідова відстань) між реальним і спрогнозованим станом вузла на момент часу t, $\Delta\hat{S}_0(t)$- характерне (медіанне) значення похибки прогнозу, визначене на навчальній вибірці; α – коефіцієнт чутливості системи, σ – логістична функція.

Інтегральний показник ризику втрати працездатності вузла NSI (Node State Index) формується як опукла комбінація прогнозованого завантаження та показника аномальності:

$$NSI_i(t+k) = (1 - w_a)L_i(t+k) + w_a A_i(t), \quad (5.5)$$

де $w_a = \in (0;1)$ – ваговий коефіцієнт, який визначає, наскільки сильно нетиповість поведінки метрик вузла впливає на загальну оцінку ризику порівняно з рівнем його завантаження.

Ухвалення керуючих рішень здійснюється за єдиним інтегральним показником, для якого визначено чотири діапазони значень із відповідними керуючими діями (табл. 5.1).

Таблиця 5.1

Чотирирівнева порогова стратегія керування станом вузла

Рівень	Діапазон значень NSI	Керуюча дія системи
1	$NSI_i = (t+k) < 0{,}3$	Ризик втрати працездатності низький. Керуючі дії не виконуються, вузол залишається в пулі балансування без змін
2	$0{,}3 \le NSI_i(t+k) < 0{,}6$	Ризик помірний. Система збільшує частоту зчитування логів вузла та розпочинає планування перерозподілу навантаження без виконання міграції мікросервісів
3	$0{,}6 \le NSI_i(t+k) < 0{,}85$	Ризик підвищений. Система ініціює адаптивні дії: розподіл нових сесій на інші вузли, активування спільного кешування або мінімізацію його вмісту
4	$NSI_i(t+k) \ge 0{,}85$	Ризик критичний. Дані про стан вузла передаються на SDN-контролер, який формує правило міграції чи перезапуску мікросервісів або виключення вузла з пулу балансування

У нашому випадку вузол вважається таким, що потребує розвантаження, за умови $NSI_i(t+k) \ge 0{,}85$. Якщо значення показника менше за порогове, вузол вважається стабільним і працездатним; якщо не менше – формується правило щодо перерозподілу навантаження, міграції мікро-

сервісів або масштабування. Залежно від умов SLA та параметрів QoS вагові коефіцієнти w_j, w_a і порогові значення можуть змінюватися та відображатися у лог-файлах, тому підсистема аналізу логів оперативно коригує алгоритм оцінювання стану функціональної працездатності вузлів мережі. Блок-схему цього алгоритму наведено на рис. 5.2.

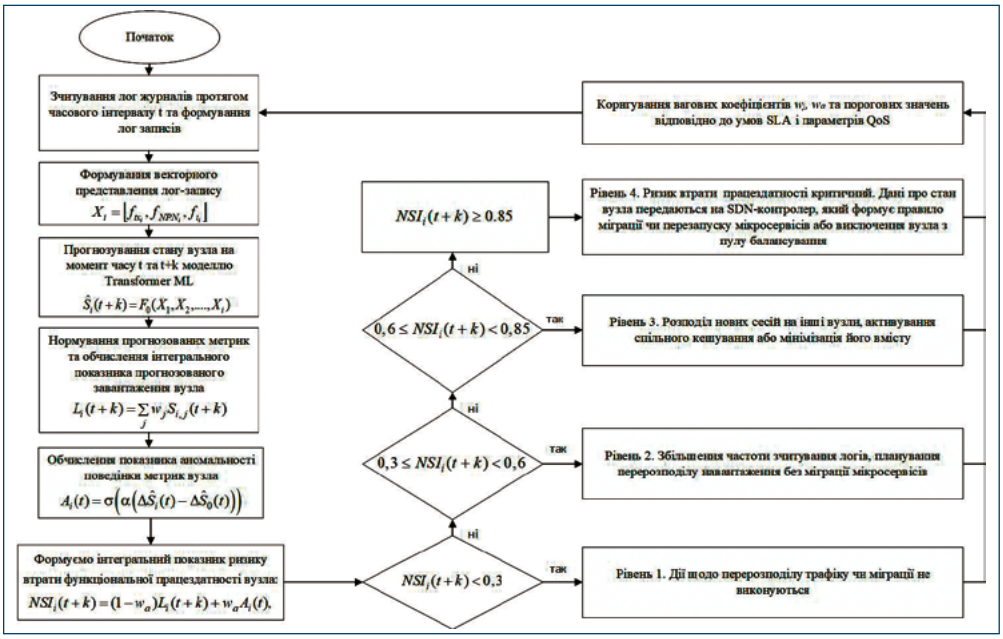


Рис. 5.2. Блок-схема алгоритму оцінки стану працездатності вузлів мережі

5.3. Дослідження ефективності методу інтелектуального управління розміщенням мікросервісів у розподілених інформаційно-комунікаційних системах

Для дослідження ефективності запропонованих рішень розроблено імітаційну модель мережі із SDN управлінням і функціонуванням мікросервісних додатків. Розроблена модель симуляції базується на дискретно-подієвому підході, який добре підходить для моделювання мереж завдяки своїй здатності точно відтворювати послідовність і час подій. Цей підхід дозволяє моделі імітувати взаємодію на рівні пакетів, процеси ухвалення рішень у комутаторах та поширення керуючих повідомлень ме-

режею з високою часовою точністю. Дискретно-подієва природа забезпечує ефективне використання обчислювальних ресурсів, оскільки події обробляються лише тоді, коли це необхідно, а не через фіксовані проміжки часу. Ця модель розроблена за допомогою мови програмування Java, використовуючи об'єктно-орієнтований підхід для моделювання компонентів мережі, таких як хости, комутатори, контролер та потоки трафіку. Обробка подій виконується через спеціальний системний планувальник подій, який дозволяє симулювати мережеві процеси з точністю до мікросекунд. Взаємодія між елементами мережі здійснюється через симуляцію протоколів SDN, таких як OpenFlow.

Основними компонентами моделі, що моделюють роботу мережевих елементів, є хост, або кінцевий пристрій (наприклад, пристрої користувачів чи Edge-пристрій), к омутатор, мережевий інтерфейс, канал зв'язку, ПКМ-контролер та пакет. Окрім мережевих елементів, модель враховує компоненти програмно-конфігурованих мереж, такі як таблиця потоків, потік, правила комутації і т.д.

Процес обробки пакету починається з моменту коли пакет повністю прийнятий мережним інтерфейсом. У такому випадку, мережний інтерфейс записує пакет у вхідну чергу пакетів пристрою, яка представляє собою FIFO-чергу, в яку попадають усі вхідні пакети. Основним компонентом комутатора є комутаційний модуль, який зчитує наступний пакет з вхідної черги, та використовує заголовок пакету для пошуку правила в таблиці потоків. Кожне правило складається з трьох компонентів, критерії відповідності, список дій, інформація про потік. Критерії відповідності, містять зразок заголовка пакету (Ethernet, IP та TCP), що дає можливість комутатору порівняти заголовок вхідного пакету з критерієм відповідності. Якщо заголовок вхідного пакету співпадає з критерієм відповідності, отже правило знайдено. Кожне правило містить список дії, які комутатор повинен виконати над пакетом, серед яких розроблені такі дії як: направлення пакету у визначений вихідний інтерфейс, відкидання пакету, пересилання пакету на контролер, оновлення метрик правила (кількість пакетів та обсяг байтів оброблених цим правилом). Таким чином, визначивши правило, комутатор виконує над пакетом весь список дій зазначений у цьому правилі у порядку, в якому їх додав до правила контролер. У випадку передачі пакету до наступного мережного елемента через один

з фізичних інтерфейсів комутатора, комутаційний блок записує пакет у вихідну чергу відповідного мережевого інтерфейсу. Мережний інтерфейс зчитує пакети з вихідної черги і пересилає їх наступному мережному елементу через канал зв'язку. При цьому, якщо канал зв'язку зайнятий, комутатор не починає передачу, а вичікує певний час (згенерований випадковим чином відповідно до алгоритмів CSMA/CD) після чого знову прослуховує канал, і так до тих пір допоки канал не стане вільним для передачі. Тривалість обробки пакету комутатором з моменту прийняття пакету вхідних інтерфейсом і до моменту запису пакету у вихідний інтерфейс моделюється шляхом розрахунку на основі максимальних значень продуктивності комутаційної матриці з урахуванням визначеної кількості мережевих інтерфейсів комутатора. При цьому в цю тривалість вносяться випадкові часові відхилення, що робить модель ближчою до реальних умов функціонування Ethernet-мереж.

Варто зазначити, що правила також містять такі поля, як пріоритет і тайм-аут. За допомогою поля пріоритету контролер має можливість створити два чи більше правил з однаковими критеріями відповідності та різним списком дій, визначивши кожному правилу пріоритет від 0 до 2^{16}. Комутатор таким чином знайде всі правила, проте вибере те, яке має найвищий пріоритет. Окрім того, кожне правило має тайм-аут, який визначається контролером при встановленні правила, і має одиницю вимірювання – секунди. Кожну секунду комутатор оновлює лічильник тайм-ауту всіх правил, віднімаючи від кожного одну секунду, і видаляє ті правила, тайм-аут яких досягнув значення нуль. Насамкінець, кожне правило містить ідентифікатор потоку, що дає можливість контролеру збирати статистику обсягів переданої інформації з кожного комутатора по кожному правилу, і таким чином здійснювати моніторинг завантаження каналів та комутаторів.

Система моделювання побудована за принципами мікросервісної архітектури та включає сервіси auth, analytics, devops, logging, dashboard, inventory, alerting та fraud, які взаємодіють між собою через комутаційні вузли switch1–switch9 та представлена на рис. 5.3. Кожен сервіс може генерувати запити до будь-якого іншого з випадковим інтервалом часу, який змінюється під час моделювання. Це дозволяє відтворити реалістичну динаміку навантаження, властиву системам із багатьма одночасно активними компонентами. Для проведення дослідження встановимо:

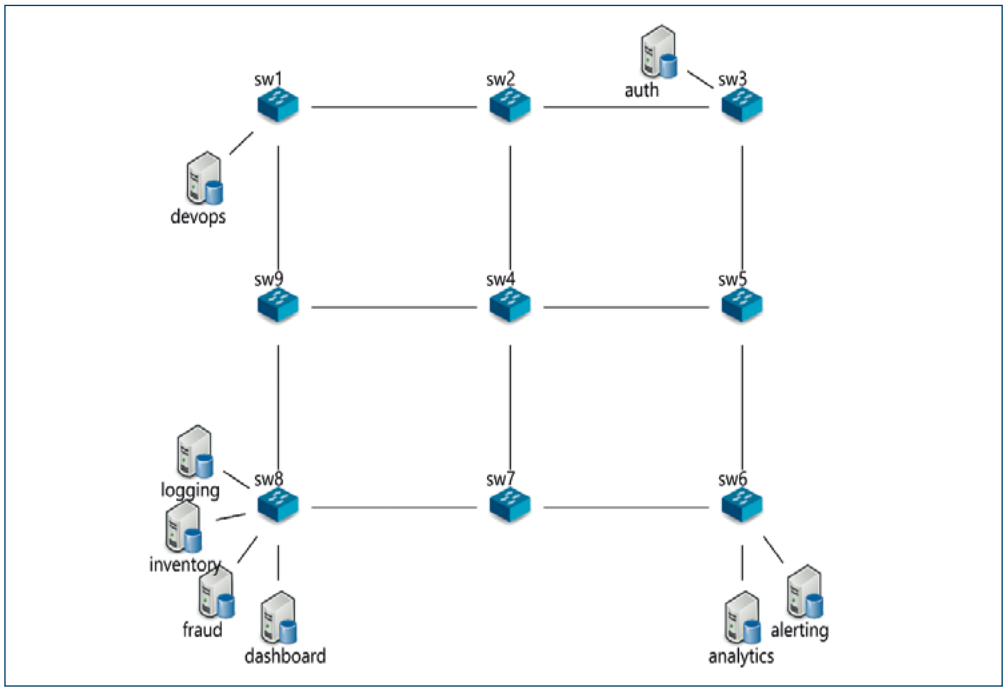


Рис. 5.3. Топологія досліджуваної мережі із розгорнутими мікросервісами

1. Пропускна здатність усіх каналів однакова і рівна 1 Мбіт/с.
2. Затримка вимірюється як різниця в часі між моментом, коли сервіс надсилає запит до іншого сервісу, і моментом отримання відповіді. Затримка враховує такі компоненти, як передачу запиту з прикладного рівня на мережевий, очікування у вихідних буферах на відправку, передачу по каналу відповідно до швидкості каналу, затримку очікування відповідно до алгоритму CSMA/CD, затримку очікування на обробку у вхідних чергах комутаторів та серверів, затримку комутації.
3. Якщо пакет втрачається, тоді такий запит відкидається, статистика втрат не врахована в результатах.
4. Міграція сервісів відбувається миттєво, нехтуючи часом необхідним для перенесення та розгортання сервісу на новий сервер. Тривалість реконфігурації мережі SDN-контролером також ігнорується і відбувається миттєво.
5. Один запит відповідає одному IP пакету який на при передачі через мережу разом з заголовком кадру Ethernet становить 1538 байт.

6. Розташування сервісів вибирається випадковим чином.
7. Рішення про міграцію сервісу (тобто інтервал зчитування лог-файлів) приймається кожні 2 секунди на основі прогнозування стану працездатності вузла та перерахунку спорідненості.

Дослідження проводилося у два етапи. У першій частині моделювання система працювала без застосування розробленої архітектури. У другій частині була використана архітектура із впровадженням методів прогнозування стану працездатності вузла на основі аналізу логів та оцінки спорідненості компонентів мікросервісу [105], що дозволяють адаптивно змінювати розміщення сервісів відповідно до поточного стану мережі. На кожному з етапів моделювання проводилося протягом 320 секунд.

У базовій конфігурації середня інтенсивність трафіку становила приблизно 127 запитів/с, з коливаннями від 33 до 171 запиту. Коефіцієнт варіації – 0,27, що свідчить про помірну варіабельність трафіку, характерну для динамічних мікросервісних систем. Протягом експерименту зафіксовано понад 100 локальних піків активності (рис. 5.4), що підтверджує наявність періодичних сплесків навантаження.



Рис. 5.4. Сумарна інтенсивність навантаження на систему (без застосування розробленої архітектури управління автоматизованим розгортанням)

Як видно із рис. 5.5 середня міжсервісна затримка становить 218 мс, медіанне значення – 210 мс, із коливаннями від 48 до 740 мс. Коефіцієнт варіації – 0,28, що свідчить про помірну мінливість затримки, а позитивний тренд її зростання вказує на перевантаження окремих каналів у моменти пікових звернень. Приблизно 49 % запитів оброблялися в межах допустимого порогу 200 мс, решта перевищували цей показник.



Рис. 5.5. Середнє значення затримки між сервісами за секунду без застосування розробленої архітектури управління автоматизованим розгортанням

Середній рівень використання каналів становив 0,42 (у шкалі 0–1), медіанне значення – 0,40, при цьому спостерігалися значні коливання між окремими лінками. Коефіцієнт варіації – 0,61, що свідчить про нерівномірне навантаження, яке призводить до появи вузьких місць у мережі та збільшення затримок (рис. 5.6).

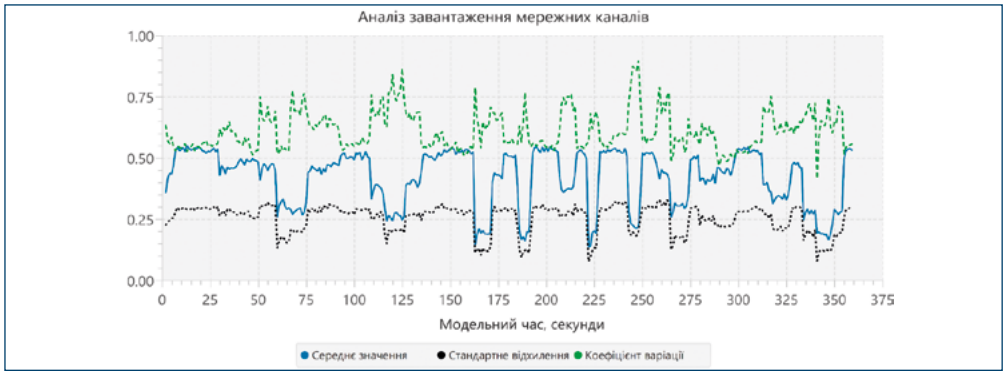


Рис. 5.6. Аналіз завантаженості мережних каналів без застосування розробленої архітектури управління автоматизованим розгортанням

Після впровадження розробленої архітектури управління автоматизованим розгортанням середня інтенсивність трафіку становила близько 129 запитів/с, у діапазоні від 45 до 188 запитів (рис. 5.7). Коефіцієнт варіації залишився на рівні 0,28, однак кількість локальних піків зменшилася на 12 % до 94.



Рис. 5.7. Сумарна інтенсивність навантаження на систему із застосування розробленої архітектури управління автоматизованим розгортанням

Із врахуванням споріденості мікросервісів середня затримка між сервісами знизилась до 192 мс, медіанне значення – 170 мс, при цьому коефіцієнт варіації становив 0,23. Частка запитів із затримкою менше 200 мс зросла до 64 %. Зменшення розкиду значень свідчить про зниження впливу окремих перевантажених лінків, що підтверджує ефективність методу прогнозування стану працездатності вузлів (рис. 5.8).

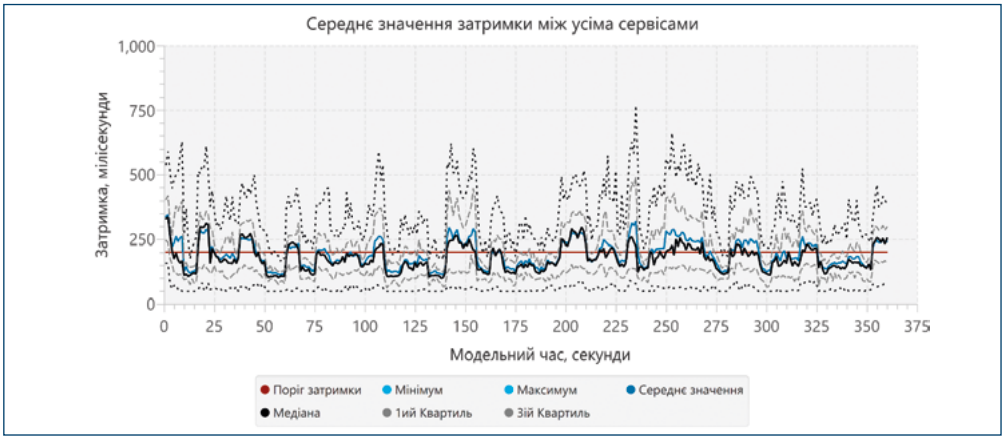


Рис. 5.8. Середнє значення затримки між сервісами із врахуванням запропонованих рішень щодо розгортання

Завдяки реалізації запропонованої системи управління ефективність використання каналів збільшилося на 8 % і середнє значення їх використанням становило 0,45, медіанне значення – 0,44, а коефіцієнт варіації зменшився на 9 % до 0,55. Це означає, що завантаження каналів

стало більш рівномірним, а ресурси мережі використовуються ефективніше (рис. 5.9).

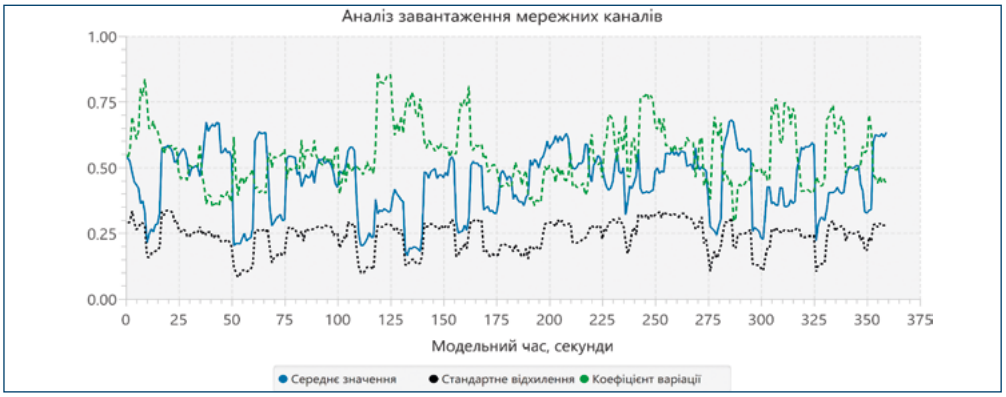


Рис. 5.9. Результати завантаженості мережних каналів після реалізації запропонованої системи управління

Розподіл затримок показав, що після впровадження архітектури середнє значення зменшилось із 218 мс до 192 мс, а медіана – з 210 мс до 170 мс (рис. 5.10). Частка запитів із затримкою нижче 200 мс зросла з 49,3 % до 64,2 %, що підтверджує зменшення кількості перевищень порогу допустимого часу обробки (рис. 5.11). Таким чином, система демонструє покращення приблизно на 15 % у швидкодії та стабільності обробки запитів.

Таким чином, у розділі запропоновано модель та метод інтелектуального управління розміщенням мікросервісів у розподілених інформаційно-комунікаційних системах, що поєднують моніторинг стану мережевих та обчислювальних ресурсів, аналіз логів, прогнозування стану вузлів і оцінювання спорідненості мікросервісів. На відміну від статичного розміщення, запропонований підхід дає змогу адаптивно змінювати місце виконання мікросервісів відповідно до поточного та прогнозованого стану інфраструктури.

Використання прогнозної інформації дає змогу завчасно виявляти вузли з ризиком перевантаження та приймати рішення щодо перерозподілу навантаження, масштабування або міграції мікросервісів до погіршення характеристик обслуговування. Врахування спорідненості компонентів при виборі місця їх розміщення, своєю чергою, сприяє скороченню міжсервісних затримок і більш рівномірному використанню мережевих ресурсів.

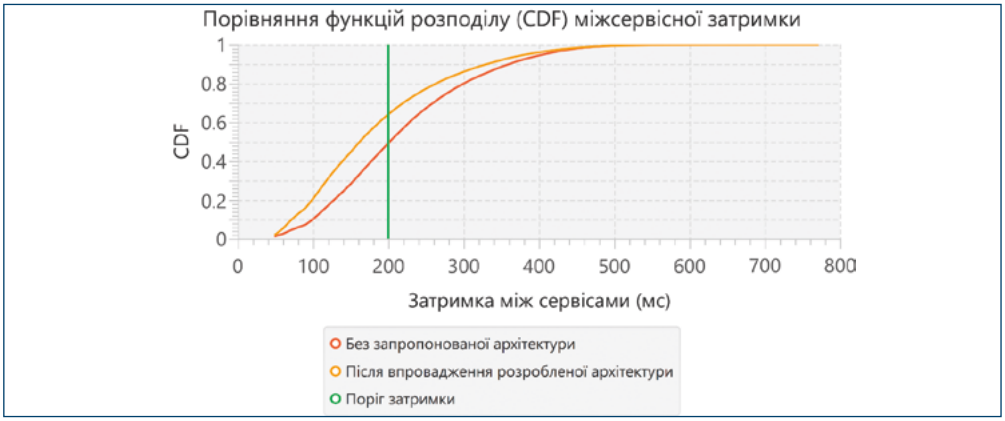


Рис. 5.10. Порівняння CDF розподілу затримок до та після впровадження архітектури

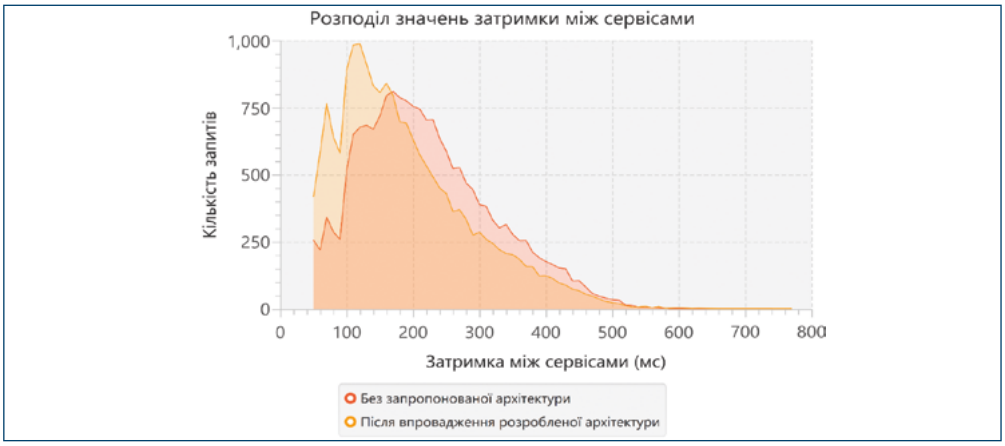


Рис. 5.11. Гістограми розподілу затримок між сервісами без і зі застосування розробленої архітектури управління автоматизованим розгортанням

Результати імітаційного моделювання підтвердили ефективність запропонованого підходу. Середня міжсервісна затримка зменшилася з 218 до 192 мс, медіанне значення – з 210 до 170 мс, а частка запитів із затримкою до 200 мс зросла з 49,3 % до 64,2 %. Одночасно середній рівень використання каналів збільшився до 0,45, а коефіцієнт варіації їх завантаження зменшився до 0,55, що свідчить про більш рівномірний розподіл мережевого навантаження.

Отже, поєднання прогнозування стану вузлів із врахуванням спорідненості мікросервісів дає змогу перейти від реагування на вже сформо-

ване перевантаження до завчасного прийняття рішень щодо розміщення компонентів. Це підвищує адаптивність РІКС до динамічних змін навантаження, сприяє ефективнішому використанню доступних ресурсів та забезпечує стабільніші часові характеристики взаємодії між мікросервісами.

5.4. Методологія побудови розподілених інформаційно-комунікаційних систем на основі запропонованих рішень

Проведені дослідження дали змогу сформувати методологію побудови розподілених інформаційно-комунікаційних систем, яка об'єднує запропоновані в попередніх розділах моделі та методи управління ресурсами, навантаженням і розміщенням мікросервісів.

Основою методології є використання інформації про поточний та прогнозований стан системи для прийняття рішень щодо її адаптації до зміни умов функціонування. Методологія передбачає побудову РІКС на основі мікросервісної архітектури, контейнеризації та засобів оркестрації Kubernetes. Функціональні компоненти системи реалізуються у вигляді окремих мікросервісів, які можуть незалежно розгортатися, масштабуватися та переміщуватися між обчислювальними вузлами. Використання Edge та Cloud ресурсів дає змогу розподіляти обчислення між різними рівнями інфраструктури відповідно до поточного навантаження, доступності ресурсів і вимог до якості обслуговування. Початковим етапом методології є визначення вимог до РІКС та формування її архітектури. На цьому етапі визначають склад мікросервісів, взаємозв'язки між ними, структуру обчислювального середовища та необхідні ресурси. Також встановлюють основні вимоги до функціонування системи, зокрема до затримки, пропускної здатності, доступності та ефективності використання обчислювальних ресурсів.

Наступним етапом є організація моніторингу стану системи. Для обчислювальних вузлів і мікросервісів здійснюється збір даних про використання процесора та оперативної пам'яті, інтенсивність надходження запитів, час їх обслуговування, пропускну здатність та інші характеристики. Додатково аналізуються системні журнали, які містять інформацію про події та зміни стану компонентів системи. Отримані дані ви-

користовуються для оцінювання поточного стану РІКС та формування вхідних даних для прогнозування. Важливою складовою методології є прогнозування зміни навантаження. Запропонований у розділі 3 метод проактивного автомасштабування використовує моделі LSTM та Holt Winters для прогнозування майбутньої потреби в обчислювальних ресурсах. За недостатньої кількості історичних даних може використовуватися модель Holt Winters. Після накопичення необхідного обсягу даних застосовується LSTM модель. Вибір прогнозної моделі здійснюється на основі оцінювання похибки прогнозування.

Отриманий прогноз використовується для завчасної зміни кількості обчислювальних ресурсів. Якщо прогнозується зростання навантаження, система збільшує доступні ресурси до моменту фактичного перевантаження. За прогнозованого зменшення навантаження надлишкові ресурси можуть бути вивільнені. Це дає змогу зменшити затримку реакції системи на зміну навантаження та підвищити ефективність використання ресурсів.

Наступною складовою методології є адаптивне управління навантаженням у розподіленому Edge-Cloud середовищі. Запропонований ресурсно-адаптивний механізм балансування враховує поточний стан обчислювальних вузлів під час вибору місця оброблення запиту. Якщо локальний вузол має достатній обсяг доступних ресурсів, запит обробляється локально. У разі перевищення встановлених порогових значень навантаження запит може бути переданий іншому доступному Edge-вузлу. Разом із ресурсно-адаптивним балансуванням використовується механізм Brownout. Він забезпечує адаптацію ресурсомісткості оброблення запитів відповідно до поточного стану системи. У запропонованій моделі передбачено три режими оброблення запитів, які характеризуються різним рівнем обчислювального навантаження та функціональної цінності. Вибір відповідного режиму здійснюється з урахуванням часових характеристик обслуговування. Це дає змогу зменшувати навантаження на систему в періоди перевантаження без повної відмови від обслуговування запитів.

Поєднання ресурсно-адаптивного балансування та Brownout дає змогу вирішувати два взаємопов'язані завдання. Механізм балансування визначає вузол, на якому доцільно обробити запит, а Brownout визначає режим його оброблення. Такий підхід забезпечує адапта-

цію як розподілу навантаження між вузлами, так і ресурсоспоживання окремих сервісів. Окремою складовою методології є інтелектуальне управління розміщенням мікросервісів. На відміну від балансування запитів між уже розгорнутими екземплярами сервісів, цей механізм визначає доцільність розміщення або переміщення мікросервісів між обчислювальними вузлами. Рішення приймається з урахуванням поточного та прогнозованого стану вузлів. Для прогнозування стану вузлів використовується запропонований у розділі 5 метод аналізу системних журналів. Виявлення ознак можливого погіршення працездатності вузла дає змогу завчасно змінити розміщення мікросервісу. Якщо результати прогнозування вказують на можливе погіршення стану поточного вузла, мікросервіс може бути розгорнутий на іншому вузлі з достатніми ресурсами. Такий підхід дає змогу зменшити вплив можливих відмов на доступність сервісів.

Таким чином, сформована методологія передбачає три основні рівні управління РІКС. Перший рівень пов'язаний з управлінням обсягом обчислювальних ресурсів на основі прогнозування навантаження. Другий рівень забезпечує адаптивний розподіл запитів між вузлами та зміну режиму їх оброблення. Третій рівень відповідає за розміщення мікросервісів з урахуванням прогнозованого стану обчислювальних вузлів. Спільне використання цих механізмів забезпечує комплексне управління ресурсами, навантаженням і розміщенням сервісів. Після виконання керуючих дій здійснюється повторне оцінювання стану системи. Аналізуються використання ресурсів, затримка, пропускна здатність, доступність сервісів та інші показники. Отримані результати знову надходять до системи моніторингу та використовуються під час наступного циклу прийняття рішень. Таким чином формується замкнений цикл управління, який забезпечує постійну адаптацію РІКС до зміни умов функціонування.

Загальну структуру запропонованої методології доцільно представити у вигляді послідовності таких етапів: визначення вимог і формування архітектури, моніторинг стану системи, аналіз та прогнозування, вибір механізму управління, виконання керуючих дій та оцінювання отриманих результатів. Результати оцінювання використовуються як вхідні дані для наступного циклу управління. Схему методології побудови РІКС на основі запропонованих рішень наведено на рис. 5.12.

Запропонована методологія об'єднує реактивні та проактивні механізми управління. Реактивні механізми забезпечують адаптацію до поточних змін навантаження та стану ресурсів. Прогнозні моделі дають змогу оцінити майбутні зміни та завчасно виконати необхідні керуючі дії. У результаті система отримує можливість не лише реагувати на перевантаження або погіршення стану вузлів, але й запобігати їх негативному впливу на роботу сервісів.

Отже, сформована методологія узагальнює запропоновані в монографії рішення та визначає порядок їх використання під час побудови РІКС. Її основою є комплексне управління обчислювальними ресурсами, навантаженням і розміщенням мікросервісів з використанням поточних та прогнозованих даних про стан системи. Застосування такого підходу дає змогу підвищити ефективність використання ресурсів, зменшити затримку обслуговування, забезпечити необхідну продуктивність та підвищити стійкість РІКС.

ВИСНОВКИ

У монографії розвинуто теоретичні положення та запропоновано моделі та методи побудови розподілених інформаційно-комунікаційних систем, орієнтованих на ефективне використання обчислювальних ресурсів, адаптацію до динамічного навантаження та забезпечення стійкого функціонування мікросервісів. Основну увагу зосереджено на поєднанні технологій контейнеризації та оркестрації Kubernetes, проактивного автомасштабування, адаптивного управління навантаженням в Edge-Cloud середовищі та інтелектуального управління розміщенням мікросервісів.

Проведений аналіз сучасних підходів до побудови РІКС дав змогу визначити основні особливості використання хмарних і периферійних обчислень, віртуалізації, контейнеризації та мікросервісної архітектури. Встановлено, що динамічний характер навантаження, неоднорідність обчислювальних ресурсів та розподіленість компонентів потребують застосування адаптивних механізмів управління. Традиційні реактивні підходи не завжди дають змогу своєчасно реагувати на швидкі зміни стану системи, що обґрунтовує необхідність використання прогнозування та проактивного управління.

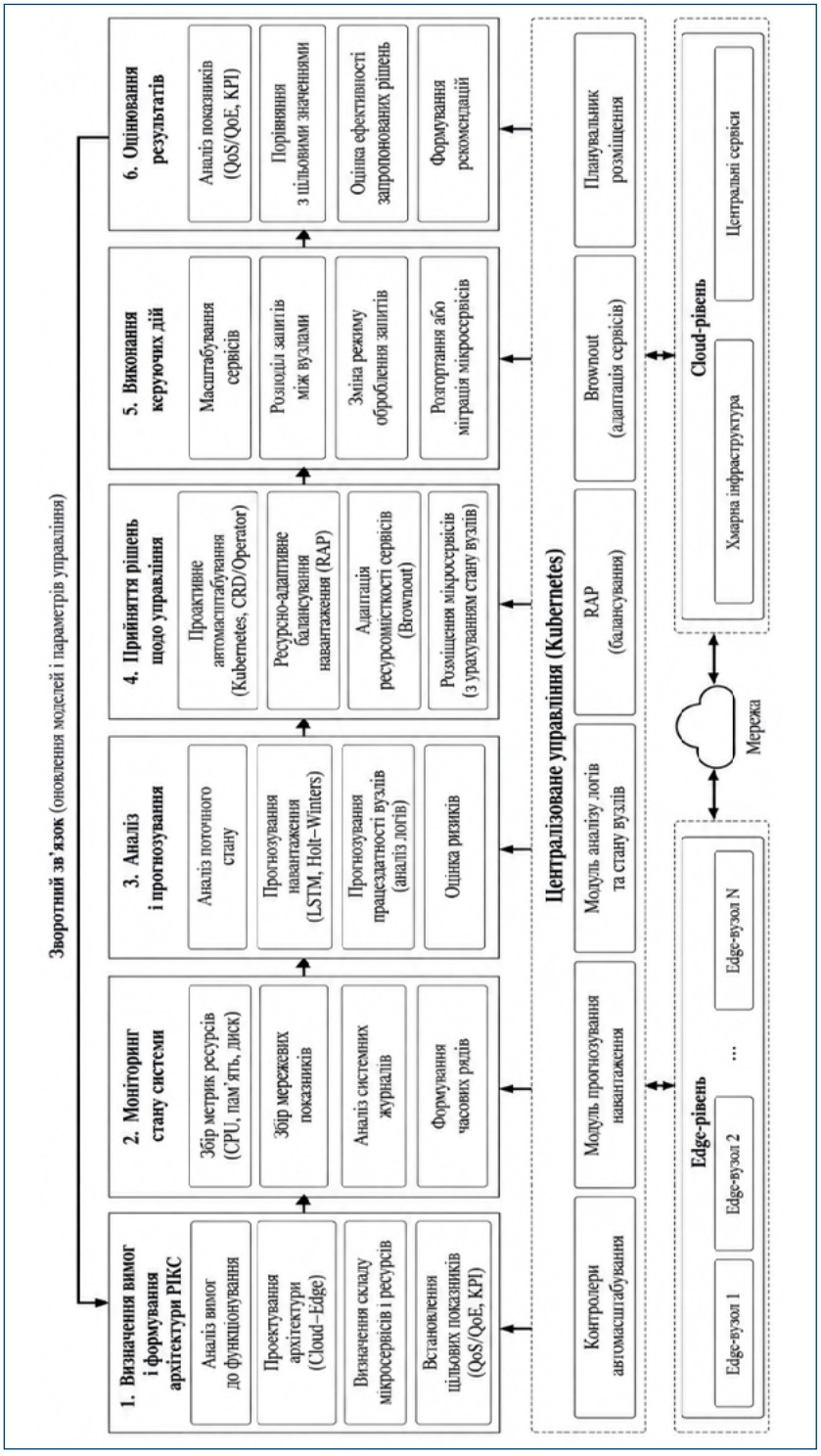


Рис. 5.12. Методологія побудови розподілених інформаційно-комунікаційних систем на основі запропонованих рішень

Систематизовано моделі та методи побудови мікросервісних РІКС з урахуванням особливостей взаємодії сервісів, обміну повідомленнями, моніторингу ресурсів та управління мікросервісами. Визначено взаємозв'язок між архітектурою системи, механізмами моніторингу, управлінням ресурсами та показниками якості обслуговування. Це стало основою для подальшого розроблення методів адаптивного управління РІКС.

Досліджено реактивні механізми вертикального та горизонтального масштабування ресурсів у Kubernetes-середовищі. Експериментальні результати показали, що збільшення ресурсів окремого контейнера дає змогу підвищити його здатність обслуговувати навантаження, однак має обмеження, пов'язані з ресурсами обчислювального вузла. Горизонтальне масштабування забезпечує розподіл навантаження між декількома екземплярами сервісу та дає змогу ефективніше адаптувати систему до збільшення кількості запитів. Водночас реактивний характер таких механізмів зумовлює затримку між виникненням потреби в додаткових ресурсах та їх фактичним наданням.

Для усунення зазначеного обмеження запропоновано метод проактивного автомасштабування обчислювальних ресурсів на основі прогнозування часових рядів. У методі використано моделі LSTM та Holt Winters, що дає змогу прогнозувати майбутнє навантаження та завчасно визначати потребу у зміні ресурсів. Вибір прогнозної моделі здійснюється з урахуванням точності отриманого прогнозу. Запропонований підхід забезпечує перехід від реагування на вже зафіксоване перевантаження до завчасного масштабування відповідно до очікуваної зміни навантаження.

Розроблено підхід до реалізації проактивного автомасштабування в Kubernetes з використанням механізмів CRD та спеціалізованого оператора. Така реалізація забезпечує інтеграцію прогнозування з процесом керування контейнеризованими застосунками та дає змогу автоматизувати прийняття рішень щодо масштабування. Проведені дослідження підтвердили доцільність використання прогнозних моделей для підвищення ефективності управління ресурсами РІКС.

Для розподіленого Edge-Cloud середовища запропоновано ресурсно адаптивний метод до управління навантаженням, який враховує поточний стан обчислювальних вузлів. Механізм балансування забезпечує вибір вузла для оброблення запиту залежно від доступності його ресурсів.

За достатнього ресурсного резерву запит обробляється локально, а у разі перевантаження може бути переданий іншому доступному вузлу. Такий підхід дає змогу враховувати нерівномірність завантаження Edge-вузлів та ефективніше використовувати наявні обчислювальні ресурси.

Для додаткової адаптації системи до зміни навантаження використано механізм Brownout, який забезпечує зміну режиму оброблення запитів відповідно до поточного стану системи. Запропонована модель передбачає використання декількох режимів обслуговування з різною ресурсомісткістю та функціональною цінністю. Поєднання Brownout із ресурсно-адаптивним балансуванням дає змогу одночасно визначати доцільний вузол оброблення запиту та адаптувати обсяг ресурсів, необхідних для його виконання. Експериментальне дослідження підтвердило ефективність такого підходу за нерівномірного розподілу навантаження між Edge-вузлами.

Запропоновано модель та метод інтелектуального управління розміщенням мікросервісів у РІКС. На відміну від підходів, які враховують переважно поточне використання ресурсів, запропоноване рішення передбачає оцінювання прогнозованого стану обчислювальних вузлів. Для цього використовується аналіз системних журналів, на основі якого визначаються ознаки можливого погіршення працездатності вузла. Отримана інформація використовується під час прийняття рішення щодо подальшого розміщення або переміщення мікросервісів.

Застосування прогнозування стану вузлів дає змогу завчасно реагувати на можливе погіршення їх працездатності та зменшувати вплив таких подій на доступність мікросервісів. У разі виявлення відповідних ознак система може прийняти рішення про розміщення сервісу на іншому вузлі з достатніми ресурсами. Таким чином, управління розміщенням мікросервісів доповнює механізми автомасштабування та балансування навантаження і формує ще один рівень адаптивного управління РІКС.

На основі отриманих результатів сформовано методологію побудови розподілених інформаційно-комунікаційних систем, яка об’єднує запропоновані в монографії рішення. Методологія охоплює визначення вимог та формування архітектури системи, моніторинг її стану, аналіз і прогнозування, прийняття керуючих рішень, виконання відповідних дій та оцінювання отриманих результатів. Результати оцінювання вико-

ристовуються у наступному циклі управління, що забезпечує постійну адаптацію системи до зміни умов функціонування.

Особливістю сформованої методології є комплексне використання трьох рівнів управління. Перший рівень забезпечує визначення необхідного обсягу обчислювальних ресурсів на основі прогнозування навантаження. Другий рівень відповідає за розподіл запитів між обчислювальними вузлами та адаптацію режиму їх оброблення. Третій рівень забезпечує управління розміщенням мікросервісів з урахуванням поточного та прогнозованого стану вузлів. Взаємодія цих рівнів дає змогу узгоджено управляти ресурсами, навантаженням та розміщенням сервісів.

Отримані результати мають практичне значення для побудови та модернізації РІКС, що функціонують в умовах змінного навантаження та неоднорідності обчислювальних ресурсів. Запропоновані рішення можуть бути використані у контейнеризованих Cloud та Edge середовищах для підвищення ефективності використання ресурсів, скорочення часу реакції на зміну навантаження, зменшення затримки обслуговування та підвищення стійкості функціонування мікросервісів.

Таким чином, у монографії сформовано комплексний підхід до побудови РІКС, у якому реактивні механізми доповнюються прогнозними та адаптивними методами управління. Поєднання проактивного автомасштабування, ресурсно-адаптивного балансування, Brownout та інтелектуального управління розміщенням мікросервісів створює основу для побудови РІКС, здатних адаптуватися до зміни навантаження, стану ресурсів та умов функціонування. Запропоновані моделі, методи та методологія спрямовані на забезпечення ефективного використання обчислювальних ресурсів, необхідного рівня якості обслуговування, масштабованості та стійкості розподілених інформаційно-комунікаційних систем.

ПОДЯКА

Автори висловлюють щиру подяку колегам та науковим партнерам, які сприяли проведенню досліджень та підготовці цієї монографії.

Окремі результати, представлені в монографії, отримано в межах виконання науково-дослідних робіт «Розроблення проактивних методів та засобів забезпечення резильєнтності розподілених інформаційно-комунікаційних систем критичної інфраструктури» (2026–2028 рр.), номер державної реєстрації 0126U001331; «Розроблення фізично-узгоджених мультимодальних моделей штучного інтелекту для роботизованих систем» (2026–2028 рр.); «Розроблення моделей функціонування інформаційної системи спеціального призначення з використанням інтелектуальних методів передавання та захисту інформації» (2024–2026 рр.), номер державної реєстрації 0225U002634; «Розробка методів і засобів побудови мобільних кіберфізичних систем із штучним інтелектом для соціальних, безпекових та оборонних завдань» (2025–2027 рр.), номер державної реєстрації 0125U 000645.

Під час підготовки монографії автори використовували ChatGPT (OpenAI) як допоміжний інструмент для мовного та стилістичного редагування, структурування окремих фрагментів тексту та підготовки графічних матеріалів. Усі наукові результати, моделі, методи, експериментальні дані, їх аналіз і висновки є результатом роботи авторів. Матеріали, опрацьовані із застосуванням штучного інтелекту, перевірені та відредаговані авторами, які несуть відповідальність за остаточний зміст монографії.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Yenhalychev S., Leunenko O., Davydov V. Development an intelligent task scheduling method in heterogeneous distributed information systems. *Eastern-European Journal of Enterprise Technologies*. 2025. Vol. 3, No. 9 (135). P. 6–18. DOI: 10.15587/1729-4061.2025.329263
2. Zinchenko A. Yu. Design of distributed information systems based on using the technology of loosely coupled components. *Systems and Technologies*. 2023. Vol. 63, No. 1. P. 5–14. DOI: 10.32782/2521-6643-2022.1-63.1
3. Tsegelyk G. G., Krasniuk R. P. The mathematical modeling of optimal data processing in distributed information systems. *System Research and Information Technologies*. 2018. No. 2. DOI: 10.20535/SRIT.2308-8893.2018.2.07
4. Maia A., Boutouchent A., Kardjadja Y., Gherari M., Gelal Soyak E., Saqib M., Boussekar K., Cilbir I., Habibi S., Ouledsidi Ali S., Ajib W., Elbiaze H., Erçetin O., Ghamri-Doudane Y., Glitho R. A survey on integrated computing, caching, and communication in the cloud-to-edge continuum. *Computer Communications*. 2024. Vol. 219. P. 128–152. DOI: 10.1016/j.comcom.2024.03.005
5. Rasouli N., Klein C., Elmroth E. Resource Management for Mission-Critical Applications in Edge Computing: Systematic Review on Recent Research and Open Issues. *ACM Computing Surveys*. 2026. Vol. 58, No. 3. Art. 71. P. 1–37. DOI: 10.1145/3762181
6. Beshley H., Seliuchenko M., Bodnar S., Beshley M., Klymash M. Development of a platform for researching automatic container scaling and load balancing in distributed systems. *Information and Communication Technologies, Electronic Engineering*. 2024. Vol. 4, No. 2. P. 38–48. DOI: 10.23939/ictee2024.02.038
7. Al-Dhuraibi Y., Paraiso F., Djarallah N., Merle P. Elasticity in Cloud Computing: State of the Art and Research Challenges. *IEEE Transactions on Services Computing*. 2018. Vol. 11, No. 2. P. 430–447. DOI: 10.1109/TSC.2017.2711009
8. Mell P., Grance T. *The NIST Definition of Cloud Computing. Gaithersburg* : National Institute of Standards and Technology, 2011. 7 p. (NIST Special Publication 800-145). DOI: 10.6028/NIST.SP.800-145
9. Muñoz-Escoí F. D., Bernabeu-Aubán J. M. A survey on elasticity management in PaaS systems. *Computing*. 2017. Vol. 99, No. 7. P. 617–656. DOI: 10.1007/s00607-016-0507-8
10. Manvi S. S., Shyam G. K. Resource management for Infrastructure as a Service (IaaS) in cloud computing: A survey. *Journal of Network and Computer Applications*. 2014. Vol. 41. P. 424–440. DOI: 10.1016/j.jnca.2013.10.004
11. Silva V. G. da, Kirikova M., Alksnis G. Containers for Virtualization: An Overview. *Applied Computer Systems*. 2018. Vol. 23, No. 1. P. 21–27. DOI: 10.2478/acss-2018- 0003
12. Jarkas O., Ko R., Dong N., Mahmud R. A Container Security Survey: Exploits, Attacks, and Defenses. *ACM Computing Surveys*. 2025. Vol. 57, No. 7. Art. 170. P. 1–36. DOI: 10.1145/3715001
13. Carrión C. Kubernetes as a Standard Container Orchestrator: A Bibliometric Analysis. *Journal of Grid Computing*. 2022. Vol. 20. Art. 42. DOI: 10.1007/s10723-022-09629-8

14. Kumar B., Verma A., Verma P. Critical insights into runtime scheduling, image, storage, and networking challenges in modern Kubernetes environments. *Computer Science Review*. 2026. Vol. 59. Art. 100851. DOI: 10.1016/j.cosrev.2025.100851

15. Di Francesco P., Lago P., Malavolta I. Architecting with microservices: A systematic mapping study. *Journal of Systems and Software*. 2019. Vol. 150. P. 77–97. DOI: 10.1016/j.jss.2019.01.001

16. Waseem M., Liang P., Shahin M., Di Salle A., Márquez G. Deployment and communication patterns in microservice architectures: A systematic literature review. *Journal of Systems and Software*. 2021. Vol. 180. Art. 111014. DOI: 10.1016/j.jss.2021.111014

17. Mohamad Sasudin M. I., Mohd Rusli H., Kama N. Monolith Application to Microservices Model Driven Analysis Migration: State-of-The-Art Techniques. *Open International Journal of Informatics*. 2022. Vol. 10, No. 2. DOI: 10.11113/oiji2022.10n2.231

18. Watada J., Roy A., Kadikar R., Pham H., Xu B. Emerging Trends, Techniques and Open Issues of Containerization: A Review. *IEEE Access*. 2019. Vol. 7. P. 152443–152472. DOI: 10.1109/ACCESS.2019.2945930

19. Casalicchio E., Iannucci S. The state-of-the-art in container technologies: Application, orchestration and security. *Concurrency and Computation: Practice and Experience*. 2020. Vol. 32, No. 17. e5668. DOI: 10.1002/cpe.5668

20. Rodriguez M. A., Buyya R. Container-based cluster orchestration systems: A taxonomy and future directions. *Software: Practice and Experience*. 2019. Vol. 49, No. 5. P. 698–719. DOI: 10.1002/spe.2660

21. Pozdniakova O., Mažeika D., Cholomskis A. SLA-Adaptive Threshold Adjustment for a Kubernetes Horizontal Pod Autoscaler. *Electronics*. 2024. Vol. 13, No. 7. Art. 1242. DOI: 10.3390/electronics13071242

22. Niazi M., Abbas S., Soliman A.-H., Alyas T., Asif S., Faiz T. Vertical Pod Autoscaling in Kubernetes for Elastic Container Collaborative Framework. *Computers, Materials & Continua*. 2023. Vol. 74, No. 1. P. 591–606. DOI: 10.32604/cmc.2023.032474

23. Faustino D., Gonçalves N., Portela M., Silva A. R. Stepwise migration of a monolith to a microservice architecture: Performance and migration effort evaluation. *Performance Evaluation*. 2024. Vol. 164. Art. 102411. DOI: 10.1016/j.peva.2024.102411

24. Elkhatib Y., Povedano Poyato J. An Evaluation of Service Mesh Frameworks for Edge Systems. *Proceedings of the 6th International Workshop on Edge Systems, Analytics and Networking*. 2023. P. 19–24. DOI: 10.1145/3578354.3592867

25. Murcia J. M. B., Zarca A. M., Braghin S., Skármeta A. Cross-domain telemetry architecture: dynamic observability in the computing continuum. *Computing*. 2026. Vol. 108. Art. 73. DOI: 10.1007/s00607-026-01660-x

26. Pomeluiko D., Yeromina N., Petrov S., Ishchenko N., Voloshchuk O. Kubernetes Cluster Scaling Methods in Cloud Environments. *Control, Navigation and Communication Systems. Academic Journal*. 2025. No. 2. P. 175–181. DOI: 10.26906/SUNZ.2025.2.175

27. Bittencourt L. F., Rodrigues-Filho R., Spillner J., De Turck F., Santos J., da Fonseca N. L. S., Rana O., Parashar M., Foster I. The computing continuum: Past, present,

and future. *Computer Science Review*. 2025. Vol. 58. Art. 100782. DOI: 10.1016/j.cosrev.2025.100782

28. Rolik O. I., Omelchenko V. V. Proactive Horizontal Scaling Method for Kubernetes. *Radio Electronics, Computer Science, Control*. 2024. No. 1. P. 215–225. DOI: 10.15588/1607-3274-2024-1-20
29. Qu C., Calheiros R. N., Buyya R. Auto-Scaling Web Applications in Clouds: A Taxonomy and Survey. *ACM Computing Surveys*. 2018. Vol. 51, No. 4. Art. 73. DOI: 10.1145/3148149
30. Vu D.-D., Tran M.-N., Kim Y. Predictive Hybrid Autoscaling for Containerized Applications. *IEEE Access*. 2022. Vol. 10. P. 109768–109778. DOI: 10.1109/ACCESS.2022.3214985
31. Kumar J., Goomer R., Singh A. K. Long Short Term Memory Recurrent Neural Network (LSTM-RNN) Based Workload Forecasting Model for Cloud Datacenters. *Procedia Computer Science*. 2018. Vol. 125. P. 676–682. DOI: 10.1016/j.procs.2017.12.087.
32. Bi J., Li S., Yuan H., Zhou M. C. Integrated deep learning method for workload and resource prediction in cloud systems. *Neurocomputing*. 2021. Vol. 424. P. 35–48. DOI: 10.1016/j.neucom.2020.11.011
33. Singh P., Gupta P., Jyoti K. TASM: Technocrat ARIMA and SVR model for workload prediction of web applications in cloud. *Cluster Computing*. 2019. Vol. 22, No. 2. P. 619–633. DOI: 10.1007/s10586-018-2868-6
34. Messias V., Estrella J., Ehlers R., Santana M., Santana R., Reiff-Marganiec S. Combining time series prediction models using genetic algorithm to autoscaling web applications hosted in the cloud infrastructure. *Neural Computing and Applications*. 2016. Vol. 27, No. 8. P. 2383–2406. DOI: 10.1007/s00521-015-2133-3
35. Toka L., Dobreff G., Fodor B., Sonkoly B. Machine Learning-Based Scaling Management for Kubernetes Edge Clusters. *IEEE Transactions on Network and Service Management*. 2021. Vol. 18, No. 1. P. 958–972. DOI: 10.1109/TNSM.2021.3052837
36. Imdoukh M., Ahmad I., Alfailakawi M. G. Machine Learning-Based Auto-Scaling for Containerized Applications. *Neural Computing and Applications*. 2020. Vol. 32. P. 9745–9760. DOI: 10.1007/s00521-019-04507-z
37. Ahmad H., Treude C., Wagner M., Szabo C. Towards Resource-Efficient Reactive and Proactive Auto-Scaling for Microservice Architectures. *Journal of Systems and Software*. 2025. Vol. 225. Art. 112390. DOI: 10.1016/j.jss.2025.112390
38. Intelligent Autoscaling of Microservices in the Cloud for Real-Time Applications. *IEEE Access*. 2021. Vol. 9. P. 35464–35476. DOI: 10.1109/ACCESS.2021.3061890
39. Abgaz Y. M., McCarren A., Elger P., Solan D., Lapuz N., Bivol M., Jackson G. M., Yilmaz M., Buckley J., Clarke P. M. Decomposition of Monolith Applications Into Microservices Architectures: A Systematic Review. *IEEE Transactions on Software Engineering*. 2023. Vol. 49, No. 8. P. 4213–4242. DOI: 10.1109/TSE.2023.328729
40. Abgaz Y. M., McCarren A., Elger P., Solan D., Lapuz N., Bivol M., Jackson G. M., Yilmaz M., Buckley J., Clarke P. M. Decomposition of Monolith Applications Into Microservices Architectures: A Systematic Review. *IEEE Transactions on Software Engineering*. 2023. Vol. 49, No. 8. P. 4213–4242. DOI: 10.1109/TSE.2023.3287297

41. Söylemez M., Tekinerdogan B., Kolukısa Tarhan A. Feature-Driven Characterization of Microservice Architectures: A Survey of the State of the Practice. *Applied Sciences*. 2022. Vol. 12, No. 9. Art. 4424. DOI: 10.3390/app12094424

42. NNINZOO. Tistory. URL: https://nninzoo.tistory.com/151

43. Goetsch K. Outer Architecture. *Microservices for Modern Commerce*. O'Reilly Media, 2017. URL: https://www.oreilly.com/library/view/microservices-for-modern/9781492049135/ch04.html

44. Butti R. Enterprise Microservices Design [Part 3: Outer Architecture Zone]. *Medium*. 2017. URL: https://ravikiran763.medium.com/enterprise-microservices-design-part-3-outer-architecture-zone-cc4c1f2ce4dc

45. Butti R. Enterprise Microservices Design [Part 2: Inner Architecture Zone]. *Medium*. 2017. URL: https://ravikiran763.medium.com/enterprise-microservices-design-part-2-inner-architecture-zone-801ffe839f02

46. Nadgowda A. Navigating the Ins and Outs of a Microservice Architecture (MSA). *InfoQ*. 2017. URL: https://www.infoq.com/articles/navigating-microservices-architecture/

47. Richardson C. Microservice Architecture Patterns. *Microservices.io*. 2021. URL: https://microservices.io/patterns/microservices.html

48. Indrasiri K. Microservices in Practice: Key Architectural Concepts of an MSA. *WSO2*. 2019. URL: https://wso2.com/whitepapers/microservices-in-practice-key-architectural-concepts-of-an-msa/

49. Dogra A., Jha R. K., Jain S. A Survey on Beyond 5G Network With the Advent of 6G: Architecture and Emerging Technologies. *IEEE Access*. 2020. Vol. 9. P. 67512–67547. DOI: 10.1109/ACCESS.2020.3031234

50. Zhang J., Zhang Y., Liu X., Wang L. 5G Network Slicing: Architecture, Service Management, and Orchestration. *Applied Sciences*. 2021. Vol. 11, No. 24. Art. 11914

51. Marchese A., Tomarchio O. Enhancing the Kubernetes Platform with a Load-Aware Orchestration Strategy. *SN Computer Science*. 2025. Vol. 6, No. 1. Art. 224. DOI: 10.1007/s42979-025-03712-z

52. Medina J. R., Sánchez M. A., Martínez F. J. Kubernetes-Based Orchestration for Efficient Microservice Deployment in Cloud Environments. *Electronics*. 2024. Vol. 13, No. 1. Art. 65

53. Canali C., Di Modica G., Lancellotti R., Rossi S., Scotece D. A Validated Performance Model for Micro-services Placement in Fog Systems. *SN Computer Science*. 2023. Vol. 4, No. 1. Art. 417. DOI: 10.1007/s42979-023-01847-5

54. Babar M. A., Khan M. A., Zomaya A. Y. The Role of Microservice Approach in Edge Computing: Opportunities and Challenges. *Journal of Computer Science and Technology*. 2023. Vol. 40, No. 3. P. 1–17. DOI: 10.1016/j.jcst.2023.03.001

55. Ashraf M., Shiraz M., Abbasi A., Alqahtani O., Badshah G., Lasisi A. Microservice Application Scheduling in Multi-Tiered Fog-Computing-Enabled IoT. *Sensors*. 2023. Vol. 23, No. 16. Art. 7142. DOI: 10.3390/s23167142

56. Choi H.-J., Kim J.-H., Lee J.-H., Han J.-Y., Kim W.-S. Adaptive Microservice Architecture and Service Orchestration Considering Resource Balance to Support

Multi-User Cloud VR. *Electronics*. 2025. Vol. 14, No. 7. Art. 1249. DOI: 10.3390/electronics14071249

57. Mohammadi V., Rahmani A. M., Darwesh A., Sahafi A. Fault Tolerance in Fog-Based Social Internet of Things. *Knowledge-Based Systems*. 2023. Vol. 265. Art. 110376. DOI: 10.1016/j.knosys.2023.110376
58. Zhang P., Xiang L., Song Z. et al. Adaptive Load Balancing and Fault-Tolerant Microservices Architecture for High-Availability Web Systems Using Docker and Spring Cloud. *Discover Applied Sciences*. 2025. Vol. 7. Art. 705. DOI: 10.1007/s42452-025-07320-7
59. Pallewatta S., Kostakos V., Buyya R. MicroFog: A Framework for Scalable Placement of Microservices-Based IoT Applications in Federated Fog Environments. *Journal of Systems and Software*. 2024. Vol. 209. Art. 111910. DOI: 10.1016/j.jss.2023.111910
60. Ahmad H., Treude C., Wagner M., Szabo C. Towards Resource-Efficient Reactive and Proactive Auto-Scaling for Microservice Architectures. *Journal of Systems and Software*. 2025. Vol. 225. Art. 112390. DOI: 10.1016/j.jss.2025.112390
61. Niazi M., Abbas S., Soliman A.-H., Alyas T., Asif S., Faiz T. Vertical Pod Autoscaling in Kubernetes for Elastic Container Collaborative Framework. *Computers, Materials & Continua*. 2023. Vol. 74, No. 1. P. 591–606. DOI: 10.32604/cmc.2023.032474
62. Beshley H., Vilhutskyi S., Beshley M., Greguš M. AI-Based Proactive Autoscaling Technique for Web Applications in Kubernetes. 2026 IEEE 18th International Conference on Advanced Trends in Radioelectronics, Telecommunications and Computer Engineering (TCSET). 2026. P. 1–6. DOI: 10.1109/TCSET65181.2026.11461141
63. Przystupa K., Beshley M. IBN and Service Management for 5G and Beyond. London : Routledge, 2025. 336 p. DOI: 10.1201/9781003651819
64. Seliuchenko M., Beshley M., Shpur O., Seliuchenko N., Klymash M., Beshley H. Software-Defined Multi-Access Edge/Cloud Computing for 5G/6G Time-Critical Services. 2023 IEEE 17th International Conference on the Experience of Designing and Application of CAD Systems (CADSM). Jaroslaw, Poland, 22–25 February 2023. P. 1–4. DOI: 10.1109/CADSM58174.2023.10076524
65. Lin H., Zeadally S., Chen Z., Labiod H., Wang L. A Survey on Computation Offloading Modeling for Edge Computing. Journal of Network and Computer Applications. 2020. Vol. 169. Art. 102781. DOI: 10.1016/j.jnca.2020.102781
66. Beshley M., Klymash M., Hamal M., Shkoropad Y., Branytskyy A. Method for Estimating Service Delay in Edge and Cloud Computing Architecture. 2020 IEEE 15th International Conference on Advanced Trends in Radioelectronics, Telecommunications and Computer Engineering (TCSET). Lviv-Slavske, Ukraine, 25–29 February 2020. P. 915–919. DOI: 10.1109/TCSET49122.2020.235570
67. Fard M. V., Sahafi A., Rahmani A. M., Mashhadi P. S. Resource Allocation Mechanisms in Cloud Computing: A Systematic Literature Review. IET Software. 2020. Vol. 14, No. 6. P. 685–700. DOI: 10.1049/iet-sen.2019.0338
68. AI-Driven Resource Allocation in Cloud Computing: A Systematic Review Revealing Critical Sustainability and Evaluation Gaps. *Computing*. 2026. DOI: 10.1007/s00607-026-01655-8

69. Jyoti A., Yadav A., Kumar D., Mamoria P., Yadav V. Classification & Technological Analysis of Load Balancing in Cloud Computing. *Recent Patents on Engineering.* 2026. Vol. 20, No. 1. Art. e18722121358630. DOI: 10.2174/0118722121358630241220044608

70. Rasouli N., Klein C., Elmroth E. Resource Management for Mission-Critical Applications in Edge Computing: Systematic Review on Recent Research and Open Issues. *ACM Computing Surveys*. 2026. Vol. 58, No. 3. Art. 71. P. 1–37. DOI: 10.1145/3762181

71. Carrión C. Kubernetes Scheduling: Taxonomy, Ongoing Issues and Challenges. *ACM Computing Surveys*. 2023. Vol. 55, No. 7. Art. 138. P. 1–37. DOI: 10.1145/3539606

72. Nguyen T.-T., Yeom Y.-J., Kim T., Park D.-H., Kim S. Horizontal Pod Autoscaling in Kubernetes for Elastic Container Orchestration. *Sensors*. 2020. Vol. 20, No. 16. Art. 4621. DOI: 10.3390/s20164621

73. Nguyen Q. M., Pham L. A., Kim T. Load-Balancing of Kubernetes-Based Edge Computing Infrastructure Using Resource Adaptive Proxy. *Sensors*. 2022. Vol. 22, No. 8. Art. 2869. DOI: 10.3390/s22082869

74. Beshley H., Beshley M., Medvetskyi M., Pyrih Y. QoS-Aware Optimal Radio Resource Allocation Method for Machine-Type Communications in 5G LTE and Beyond Cellular Networks. *Wireless Communications and Mobile Computing*. 2021. Vol. 2021. Art. 9966366. P. 1–18. DOI: 10.1155/2021/9966366

75. Beshley M., Kryvinska N., Yaremko O., Beshley H. A Self-Optimizing Technique Based on Vertical Handover for Load Balancing in Heterogeneous Wireless Networks Using Big Data Analytics. *Applied Sciences*. 2021. Vol. 11, No. 11. Art. 4737. P. 1–24. DOI: 10.3390/app11114737

76. Su J., Beshley M., Przystupa K., Kochan O., Rusyn B., Stanisławski R., Yaremko O., Majka M., Beshley H., Demydov I., Pyrih Y., Kahalo I. 5G Multi-Tier Radio Access Network Planning Based on Voronoi Diagram. *Measurement*. 2022. Vol. 192. Art. 110814. P. 1–17. DOI: 10.1016/j.measurement.2022.110814

77. Beshley M., Kryvinska N., Beshley H. Energy-Efficient QoE-Driven Radio Resource Management Method for 5G and Beyond Networks. *IEEE Access*. 2022. Vol. 10. P. 131691–131710. DOI: 10.1109/ACCESS.2022.3229897

78. Yim J. W. QoS for Best-Effort Batch Jobs in Container-Based Cloud. *Concurrency and Computation: Practice and Experience*. 2023. Vol. 35. Art. e6422. DOI: 10.1002/cpe.6422

79. Hadeed W., Abdullah D. B. Load Balancing Mechanism for Edge-Cloud-Based Priorities Containers. *International Journal of Wireless and Microwave Technologies*. 2022. Vol. 12, No. 5. P. 1–9. DOI: 10.5815/ijwmt.2022.05.01

80. Xu M., Buyya R. Energy Efficient Scheduling of Cloud Application Components with Brownout. *IEEE Transactions on Sustainable Computing*. 2017. Vol. 2, No. 2. P. 60–74. DOI: 10.1109/TSUSC.2017.2661339

81. Xu M., Buyya R. Brownout Approach for Adaptive Management of Resources and Applications in Cloud Computing Systems: A Taxonomy and Future Directions. *ACM Computing Surveys*. 2019. Vol. 52, No. 1. Art. 8. P. 1–27.

82. Klein C., Maggio M., Årzén K.-E., Hernández-Rodriguez F. Brownout: Building More Robust Cloud Applications. *Proceedings of the 36th International Conference*

on Software Engineering. Hyderabad, India, 31 May–7 June 2014. P. 700–711. DOI: 10.1145/2568225.256832

83. Locust. An open source load testing tool. URL: https://locust.io/
84. Nylander T., Klein C., Årzén K.-E., Maggio M. BrownoutCC: Cascaded Control for Bounding the Response Times of Cloud Applications. *Proceedings of the 2018 American Control Conference (ACC)*. Milwaukee, WI, USA, 27–29 June 2018. P. 3354–3361. DOI: 10.23919/ACC.2018.8431282
85. Dürango J., Dellkrantz M., Maggio M., Klein C., Papadopoulos A. V., Hernández-Rodriguez F., Elmroth E., Årzén K.-E. Control-Theoretical Load-Balancing for Cloud Applications with Brownout. *2014 IEEE 53rd Annual Conference on Decision and Control (CDC). Los Angeles*, CA, USA, 15–17 December 2014. P. 5320–5327. DOI: 10.1109/CDC.2014.7040221
86. García Gil S., Murillo J. M., Galán-Jiménez J. Optimizing IoT Microservices Placement for Latency Reduction in UAV-Assisted Wireless Networks. *2023 IEEE 20th International Conference on Mobile Ad Hoc and Smart Systems (MASS)*. 2023. P. 658–663.
87. Daoud M., El Mezouari A., Faci N., Benslimane D., Maamar Z. et al. A Multi-Model Based Microservices Identification Approach. *Journal of Systems Architecture*. 2021. Vol. 118. Art. 102200. DOI: 10.1016/j.sysarc.2021.102200
88. Luo S., Xu H., Lu C., Ye K., Xu G., Zhang L., He J., Xu C. An In-Depth Study of Microservice Call Graph and Runtime Performance. *IEEE Transactions on Parallel and Distributed Systems*. 2022. Vol. 33, No. 12. P. 3901–3914. DOI: 10.1109/TPDS.2022.3174631
89. Wen Y., Cheng G., Deng S., Yin J. Microservice Workflow Modeling for Affinity Scheduling to Improve the QoS. *Web and Big Data*. 2023. P. 313–328. DOI: 10.1007/978-3-031-25158-0_24
90. Nicolas-Plata A., Gonzalez-Compean J. L., Sosa-Sosa V. J. A Service Mesh Approach to Integrate Processing Patterns into Microservices Applications. *Cluster Computing*. 2024. Vol. 27. P. 7417–7438. DOI: 10.1007/s10586-024-04342-5
91. Beshley M., Kryvinska N., Beshley H., Yaremko O., Klymash Y. Virtual Router Design and Modeling for Future Networks with QoS Guarantees. *Electronics*. 2021. Vol. 10, No. 10. Art. 1139. P. 1–26.
92. Beshley M., Kryvinska N., Beshley H., Medvetskyi M., Barolli L. Centralized QoS Routing Model for Delay/Loss Sensitive Flows at the SDN-IoT Infrastructure. *Computers, Materials & Continua*. 2021. Vol. 69, No. 3. P. 3727–3748. DOI: 10.32604/cmc.2021.018625
93. Huang F., Beshley M., Seliuchenko M., Kochan O., Beshley H., Bodnar S. QoS-Driven Auto-Scaling Docker Containers Testbed with Dynamic Weighted Load Balancing. *2023 IEEE 5th International Conference on Advanced Information and Communication Technologies (AICT)*. Lviv, Ukraine, 2023. P. 48–51. DOI: 10.1109/AICT61584.2023.10452680
94. Luntovskyy A., Beshley M. Designing HDS under Considering of QoS Robustness and Security for Heterogeneous IBN. *Future Intent-Based Networking: On the QoS Robust and Energy Efficient Heterogeneous Software Defined Networks. Lecture Notes in Electrical Engineering*. 2022. Vol. 831. P. 19–37.

95. Beshley M., Kryvinska N., Beshley H., Panchenko O., Medvetskyi M. Traffic Engineering and QoS/QoE Supporting Techniques for Emerging Service-Oriented Software-Defined Network. *Journal of Communications and Networks*. 2024. Vol. 26, No. 1. P. 99–114. DOI: 10.23919/JCN.2023. 000065

96. Brogi A., Forti S., Guerrero C., Lera I. How to Place Your Apps in the Fog: State of the Art and Open Challenges. *Software: Practice and Experience*. 2020. Vol. 50, No. 5. P. 719–740. DOI: 10.1002/spe.2766

97. Smet P., Dhoedt B., Simoens P. Docker Layer Placement for On-Demand Provisioning of Services on Edge-Clouds. *IEEE Transactions on Network and Service Management*. 2018. Vol. 15, No. 3. P. 1161–1174. DOI: 10.1109/TNSM.2018.2844187

98. Samanta A., Tang J. Dyme: Dynamic Microservice Scheduling in Edge Computing Enabled IoT. *IEEE Internet of Things Journal*. 2020. Vol. 7, No. 7. P. 6164–6174.

99. Yin L., Luo J., Luo H. Tasks Scheduling and Resource Allocation in Fog Computing Based on Containers for Smart Manufacturing. *IEEE Transactions on Industrial Informatics*. 2018. Vol. 14, No. 10. P. 4712–4721. DOI: 10.1109/TII.2018.2851241

100. Li X., Wan J., Dai H.-N., Imran M., Xia M., Celesti A. A Hybrid Computing Solution and Resource Scheduling Strategy for Edge Computing in Smart Manufacturing. *IEEE Transactions on Industrial Informatics*. 2019. Vol. 15, No. 7. P. 4225–4234. DOI: 10.1109/TII.2019.2899679

101. He S., Xiong A., Zhang M., Li C. Microservice Deployment Strategy Based on Interaction Weight in Edge Environment. *Research Square*. 2024. P. 1–13. DOI: 10.21203/rs.3.rs-3916188/v1

102. Chen M., Islam M., Read M., Buyya R. TraDE: Network and Traffic-Aware Adaptive Scheduling for Microservices Under Dynamics. *arXiv*. 2024. arXiv:2411.05323. DOI: 10.48550/arXiv.2411.05323

103. Peleh N., Shpur O., Klymash M. Intelligent Detection of DDoS Attacks in SDN Networks. *Future Intent-Based Networking: On the QoS Robust and Energy Efficient Heterogeneous Software Defined Networks. Lecture Notes in Electrical Engineering*. 2022. Vol. 831. P. 210–222.

104. Shpur O., Klymash M., Seliuchenko M., Strykhaliuk B., Lavriv O. Improving the Quality of Composite Services Through Improvement of Cloud Infrastructure Management. *International Journal of Computer Science and Information Security*. 2015. Vol. 13, No. 9. P. 36–44. ISSN 1947-5500.

105. Shpur O., Seliuchenko M., Klymash M. Control of Automated Deployment of Microservice Applications with QoS/QoE Support for Sustainable Development of Information and Communication Networks. *Networks and Sustainability. Lecture Notes in Electrical Engineering*. 2025. Vol. 1473. P. 685–705. DOI: 10.1007/978-3-032-02272-1_31

Монографія

МОДЕЛІ ТА МЕТОДИ ПОБУДОВИ РОЗПОДІЛЕНИХ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ СИСТЕМ

БЕШЛЕЙ Микола Іванович ,
КЛИМАШ Михайло Миколайович,
ШПУР Ольга Миколаївна,
БЕШЛЕЙ Галина Володимирівна

Виготовлення оригінал-макета:
Лілія Ребрик

Формат: 70x100/16. Ум. друк. арк. 17.88
Наклад 100 прим. Зам. № ___

Видавець:
Видавництво «ЛітЕра»
ФОП Ребрик Лілія Володимирівна
вул. Героїв України, 9-106, Полтава-40, 36040.
Свідоцтво суб'єкта видавничої справи
серія ДК № 6024 від 19.02.2018 р.

Виготовлювач: