ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ СИСТЕМ ТА ТЕХНОЛОГІЙ

КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«МЕТОДИ УПРАВЛІННЯ РЕСУРСАМИ ІНФОРМАЦІЙНИХ СИСТЕМ У ХМАРНИХ СЕРЕДОВИЩАХ»

на здобуття освітнього ступеня магістр
зі спеціальності 124 Системний аналіз
(код,найменування спеціальності)
освітньо-професійної програми Інтелектуальні системи управління
(назва)

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

_____ Єгор НОСОВ
(підпис) *(ім'я, ПРІЗВИЩЕ здобувача)*

Виконав:
здобувач вищої освіти
група САДМ-61 Єгор НОСОВ
(ім'я, ПРІЗВИЩЕ)
Керівник д.т.н., проф. Олесій ШУШУРА
(ім'я, ПРІЗВИЩЕ)

Рецензент: к.т.н., доцент Наталія ЛАЩЕВСЬКА
(ім'я, ПРІЗВИЩЕ)

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут Інформаційних технологій

Кафедра Інформаційних систем та технологій
Ступінь вищої освіти магістр
Спеціальність 124 Системний аналіз
Освітньо-професійна прорама Інтелектуальні системи управління

ЗАТВЕРДЖУЮ

Завідувач кафедрою ІСТ
Каміла СТОРЧАК

"____" _____2025 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Носову Єгору Дмитровичу
(прізвище, ім'я, по батькові здобувача)

1.Тема кваліфікаційної роботи: Методи управління ресурсами інформаційних систем у хмарних середовищах
керівник кваліфікаційної роботи: Олексій ШУШУРА д.т.н., проф.
(ім'я, ПРІЗВИЩЕ, науковий ступінь, вчене звання)
затверджені наказом Державного університету інформаційно-комунікаційних технологій від "30" жовтня 2025 р. № 467

2. Строк подання кваліфікаційної роботи «26» грудня 2025 р.

3. Вихідні дані кваліфікаційної роботи:
1. Хмарні системи.
2. Архітектура хмарних систем.
3. Математичне моделювання процесів управління.
4. Науково-технічна література.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити):
1. Теоретичні основи хмарних систем.
2. Огляд математичного моделювання процесів управління хмарними системами.
3. Аналіз алгоритму управління ресурсами хмарних технологій

5.Перелік ілюстраційного матеріалу: *презентація*

6. Дата видачі завдання «30» жовтня 2025р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Підбір технічної літератури	01.11.2025	
2.	Дослідження хмарних систем і технологій	08.11.2025	
3.	Математичне моделювання і розробка процесів управління	11.11.2025	
4.	Результати впровадження алгоритму управління ресурсами	20.11.2025	
5.	Висновки по роботі	22.11.2025	
6.	Розробка демонстраційних матеріалів, доповідь.	26.11.2025	
7.	Оформлення магістерської роботи	30.11.2025	

Здобувач вищої освіти _____ Єгор НОСОВ
(підпис) *(ім'я, ПРІЗВИЩЕ)*

Керівник кваліфікаційної роботи _____ Олексій ШУШУРА
(підпис) *(ім'я, ПРІЗВИЩЕ*

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 77 стор., 24 рис., 7 табл., 24 джерела.

Мета роботи – автоматизація процесу управління ресурсами у хмарних середовищах з використання генетичних алгоритмів та методів підкріплювального навчання

Об'єкт дослідження – процес управління ресурсами в хмарних системах.

Предмет дослідження – управління ресурсами в хмарних системах.

Короткий зміст роботи. У першому розділі магістерської роботи розглянуто теоретичні основи та виконано аналіз особливостей області управління ресурсами у хмарних системах. Проаналізовано проблеми та підходи до управління ресурсами з іншими технологіями. Розглянуті сучасні хмарні плафторми та технології, їхню структуру та особливості

Виконано математичне моделювання процесів управління ресурсами. Розглянуто методи модулювання та прийняття рішень. Розроблено математичну модель управління ресурсами.

У третьому розділі описується архітектура та реалізація моделі та алгоритму управління ресурсами. Представлені результати експериментальних досліджень розробленого алгоритму оптимізації та аналіз ефективності.

КЛЮЧОВІ СЛОВА: ХМАРНІ СИСТЕМИ, УПРАВЛІННЯ РЕСУРСАМИ, МОДЕЛЬ, МЕТОД, АЛГОРИТМ, ПІДКРІПЛЮВАЛЬНЕ НАВЧНАННЯ, ГЕНЕТИЧНИЙ АЛГОРИТМ.

ABSTRACT

Text part of the qualification work for obtaining a master's degree: 77 pages, 24 figures, 7 tables, 24 sources.

The purpose of the work is to automation of resource management in cloud environments using genetic algorithms and reinforcement learning methods

The object of research is the process of resource management in cloud systems.

The subject of research is resource management in cloud systems.

Overview of the work. The first chapter of the master's thesis examines the theoretical foundations and analyzes the characteristics of resource management in cloud systems. It analyzes problems and approaches to resource management with other technologies. It examines modern cloud platforms and technologies, their structure, and characteristics.

Mathematical modeling of resource management processes is performed. Modulation and decision-making methods are considered. A mathematical model of resource management is developed.

The third chapter describes the architecture and implementation of the resource management model and algorithm. The results of experimental studies of the developed optimization algorithm and analysis of its effectiveness are presented.

KEYWORDS: CLOUD SYSTEMS, RESOURCE MANAGEMENT, MODEL, METHOD, ALGORITHM, REINFORCEMENT LEARNING, GENETIC ALGORITHM.

ЗМІСТ

ВСТУП

Актуальність теми. Стрімке зростання обсягів даних, широке поширення розподілених застосунків та інтенсивна цифровізація бізнес-процесів зумовили перехід організацій до використання хмарних обчислювальних платформ. Такий підхід забезпечує масштабованість, гнучкість і економічність, проте водночас створює нові виклики, пов’язані з управлінням ресурсами в умовах динамічного та нерівномірного навантаження.

Хмарні системи мають підтримувати стабільну продуктивність, низьку затримку, високу доступність та відповідність вимогам SLA навіть у періоди інтенсивних коливань трафіку. У традиційних підходах до масштабування — статичних або порогових — відсутні механізми врахування контексту, прогнозування або адаптації до непередбачуваних поведінкових сценаріїв. Це часто призводить до перевитрат ресурсів, зниження ефективності, погіршення якості обслуговування або до порушення SLA.

У зв’язку з цим актуальним є застосування інтелектуальних методів оптимізації, здатних автоматизувати процес управління ресурсами, знаходити оптимальні конфігурації та адаптуватися до змін середовища у режимі реального часу. Особливу увагу привертають генетичні алгоритми, методи підкріплювального навчання та їхні гібридні поєднання, які дозволяють одночасно забезпечити глобальну оптимізацію параметрів і локальну адаптацію політики масштабування.

Таким чином, проблема ефективного управління ресурсами хмарних систем є важливою науковою та практичною задачею, що має суттєвий вплив на продуктивність, вартість та надійність сучасних інформаційних сервісів.

Мета роботи – автоматизація процесу управління ресурсами у хмарних середовищах.

Для досягнення мети, у магістерській роботі успішно виконано наступні завдання:

- Дослідження області управління ресурсами у хмарних системах;

- Математичне моделювання процесів управління ресурсами;
- Аналіз розробленого алгоритму управління ресурсами.

Об'єкт дослідження – процес управління ресурсами в хмарних системах.

Предмет дослідження – управління ресурсами в хмарних системах.

Методи дослідження. Під час написання магістерської кваліфікаційної роботи були використані методи теоретичного дослідження, імітаційного моделювання, статистичні методи, та методи оптимізації ресурсів.

Наукова новизна одержаних результатів. У ході дослідження описано новий підхід для оптимізації ресурсів у хмарних системах на основі комбінації методів підкріплювального навчання та генетичних алгоритмів. Запропонований метод забезпечує кращу адаптивність, стабільність та ефективність масштабування порівняно з традиційними підходами.управління ресурсами у хмарних системах.

Практична значущість одержаних результатів. Запропонована модель не залежить від конкретної платформи та може бути адаптована під будь-яке хмарне середовище.

Апробація результатів магістерської роботи. Основні положення і результати магістерської роботи доповідались на науково практичних конференціях, що проходили на базі Державного університету інформаційно-комунікаційних технологій та Державного університету «Житомирська політехніка».

1 ТЕОРЕТИЧНІ ОСНОВИ ТА АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ УПРАВЛІННЯ РЕСУРСАМ У ХМАРНИХ СИСТЕМАХ

1.1 Архітектура та класифікація хмарних обчислень (IaaS, PaaS, SaaS)

Хмарні обчислення є однією з провідних технологій сучасної інформатики, що забезпечує гнучке та масштабоване використання обчислювальних ресурсів через мережу Інтернет[1], [2], [3].Вони дозволяють організаціям та користувачам уникати капітальних витрат на придбання власного обладнання, надаючи можливість орендувати ресурси на вимогу. Основною концепцією хмарних обчислень є надання IT-послуг як сервісу (Everything as a Service, XaaS), що базується на віртуалізації, автоматизації управління ресурсами та централізованому адмініструванні. Типова архітектура хмарної системи складається з трьох основних рівнів: інфраструктурного, платформного та прикладного [4], [5]. Кожен рівень визначає ступінь абстракції від фізичних ресурсів та рівень відповідальності користувача за керування системою.

Інфраструктурний рівень забезпечує віртуалізовані ресурси, обчислювальні потужності, сховища даних, мережеві компоненти[4]. Користувач отримує контроль над операційними системами, середовищами виконання та прикладним ПЗ, але не управляє фізичною інфраструктурою. Приклади реалізації: Amazon EC2, Microsoft Azure Virtual Machines, Google Compute Engine[4]. Основні переваги IaaS: масштабованість, оплата за спожиті ресурси, швидке розгортання віртуальних серверів.

Платформний рівень надає середовище для розробки, тестування та розгортання програмних застосунків без необхідності адміністрування серверів чи мереж. Розробники можуть зосередитися на логіці програмного продукту, а не на підтримці інфраструктури [4]. Типові представники: Google App Engine, Microsoft Azure App Services, Heroku. Головна перевага PaaS - підвищення продуктивності процесу розробки та стандартизація середовища виконання.

Прикладний рівень передбачає надання користувачу готових до використання прикладних програм через веб-інтерфейс або клієнтські додатки. Користувач не керує інфраструктурою, платформою чи конфігурацією застосунку[5]. Приклади: Google Workspace, Microsoft 365, Salesforce CRM. SaaS забезпечує мінімальні витрати на розгортання та обслуговування ПЗ, автоматичні оновлення й високу доступність сервісів.

Залежно від рівня доступу та контролю над інфраструктурою виділяють такі моделі розгортання хмар [3], [9]:

- Публічна хмара (Public Cloud**)** - ресурси надаються провайдером для широкого кола користувачів. Основна характеристика - мультиорендність (multi-tenancy).
- Приватна хмара (Private Cloud**)** - інфраструктура використовується однією організацією, що забезпечує підвищений контроль та безпеку.
- Гібридна хмара (Hybrid Cloud) - поєднання публічної та приватної хмар із можливістю обміну даними та динамічного розподілу навантаження.
- Комунальна хмара (Community Cloud) - створюється для групи організацій із спільними вимогами до безпеки або політик доступу.

Ефективне управління ресурсами у хмарних системах потребує врахування рівня сервісної моделі.

В IaaS управління фокусується на віртуалізації серверів, балансуванні навантаження, плануванні ресурсів та моніторингу стану інфраструктури.

В PaaS головну роль відіграють алгоритми масштабування додатків, управління контейнерами (Docker, Kubernetes) та контроль середовища виконання.

В SaaS - розподіл навантаження між клієнтами, забезпечення SLA (Service Level Agreement) та моніторинг користувацької активності[9].

Отже, архітектура хмарних обчислень визначає рівень абстракції управління ресурсами, а правильна класифікація сервісів дозволяє обрати оптимальні методи адміністрування, моніторингу та оптимізації використання обчислювальних потужностей.

1.2 Інформаційні системи в хмарних середовищах: структура та особливості

Розвиток хмарних технологій суттєво змінив підхід до створення та експлуатації інформаційних систем. Якщо традиційні інформаційні системи (ІС) базувалися на локальній інфраструктурі з власними серверами, сховищами та мережами, то сучасна тенденція полягає в інтеграції або повній передачі таких систем у хмарне середовище. Це забезпечує масштабованість, гнучкість, високу доступність та зменшення витрат на технічну підтримку.

Хмарна інформаційна система - це сукупність взаємопов'язаних компонентів, що реалізують процеси збору, обробки, зберігання та передачі даних. Залежно від типу моделі обслуговування (IaaS, PaaS, SaaS) структура може мати різний ступінь централізації та абстракції, але загальна архітектура включає такі основні рівні:

1. Рівень користувацького інтерфейсу - забезпечує взаємодію користувача з системою через веб-застосунки, мобільні додатки або спеціалізовані клієнтські програми. Основна функція - це надання зручного доступу до сервісів незалежно від фізичного місцезнаходження користувача.

2. Прикладний рівень (Application Layer) - реалізує логіку бізнес-процесів, управління транзакціями, аналітику та обробку запитів. У хмарних системах цей рівень може бути розподіленим між кількома сервісами або контейнерами.

3. Сервісний рівень (Service Layer) - містить API, мікросервіси та інтерфейси взаємодії між прикладними компонентами. Використання RESTful- або GraphQL-архітектур забезпечує стандартизований доступ до ресурсів.

4. Рівень управління даними (Data Layer) - включає системи управління базами даних (SQL/NoSQL), сховища об’єктів (object storage), індексні сервіси та системи кешування. Хмарна інфраструктура дозволяє використовувати георозподілені сховища для підвищення стійкості та продуктивності.

5. Інфраструктурний рівень (Infrastructure Layer) - відповідає за розподіл обчислювальних потужностей, мережевих ресурсів, сховищ і віртуалізацію. У

цьому шарі реалізуються механізми балансування навантаження, резервного копіювання, моніторингу та масштабування.

Функціонування інформаційних систем у хмарних середовищах відрізняється від традиційних архітектур клієнт-сервер завдяки розподілу, віртуалізації ресурсів, автоматизації управління та орієнтації на забезпечення доступності та масштабованості. Особливості хмарних ІС проявляються на рівні управління інфраструктурою, організації зберігання даних, інформаційної безпеки, оптимізації використання обчислювальної потужності та підтримки якості обслуговування (QoS).

Віртуалізація та абстракція ресурсів. Основою хмарного середовища є віртуалізація, яка забезпечує розподіл фізичних ресурсів (серверів, пам'яті, мережевих інтерфейсів, сховищ) між декількома користувачами або службами.

Завдяки цьому досягається:

– гнучке керування ресурсами - можливість виділення та звільнення ресурсів у реальному часі;

– ізоляція середовищ - віртуальні машини чи контейнери працюють незалежно одна від одної, що підвищує безпеку;

– раціональне використання обладнання - підвищення коефіцієнта завантаження серверів.

Технологічно це реалізується через гіпервізори (VMware ESXi, KVM, Xen) або контейнеризацію (Docker, LXC).

Використання контейнерів дозволяє запускати численні незалежні сервіси в межах однієї операційної системи, що мінімізує накладні витрати.

Мультиорендність та поділ ресурсів. Особливістю хмарних ІС є модель multi-tenancy, за якої ресурси однієї фізичної інфраструктури одночасно обслуговують багато клієнтів. Кожен користувач отримує власний простір даних і доступ лише до своїх сервісів, тоді як провайдер управляє загальною інфраструктурою.

Мультиорендність дозволяє:

– значно зменшити вартість обслуговування за рахунок спільного використання обладнання;

– динамічно балансувати навантаження між клієнтами;

– ефективно масштабувати сервіси, використовуючи спільні кластери обчислювальних вузлів.

Водночас вона створює проблеми безпеки та ізоляції, які вирішуються за допомогою механізмів шифрування, політик доступу та сегментації мереж.

Автоматизоване управління інфраструктурою. Для підтримки великомасштабних хмарних систем використовується автоматизоване управління життєвим циклом ресурсів.

До таких процесів належать:

– моніторинг (збір метрик навантаження, споживання пам’яті, мережевої активності);

– автоскейлінг (динамічне збільшення або зменшення кількості вузлів);

– автоматичне відновлення після збоїв (self-healing);

– планування завдань і оркестрація контейнерів (Kubernetes, Docker Swarm, OpenStack Heat).

Інфраструктура описується як код (Infrastructure as Code, IaC) за допомогою інструментів типу Terraform, Ansible, Helm, що спрощує розгортання, контроль версій і повторюваність конфігурацій.

Масштабованість та балансування навантаження. Хмарні інформаційні системи мають властивість еластичності - здатність адаптувати кількість обчислювальних ресурсів до поточного попиту.

Масштабування буває:

1. вертикальне (scale up) - збільшення потужності окремого вузла (CPU, RAM);

2. горизонтальне (scale out) - додавання нових вузлів або контейнерів до кластера.

Для балансування навантаження використовуються алгоритми типу Round-Robin, Least Connections, Weighted Distribution або Application-Aware Balancing. Завдяки цьому забезпечується стабільна продуктивність навіть при різких пікових навантаженнях.

Висока доступність та відмовостійкість. У хмарних системах реалізуються механізми високої доступності **(**High Availability, HA**)** та катастрофостійкості (Disaster Recovery, DR)**.**

До основних підходів належать:

- реплікація даних між дата-центрами (multi-region replication);
- географічно розподілені кластери;
- резервне копіювання (backup & snapshot policy);
- автоматичне перемикання на резервні вузли (failover).

Типовим прикладом є сервіси AWS, які мають зони доступності (Availability Zones) у різних регіонах - це забезпечує безперервність роботи при локальних збоях.

Забезпечення безпеки та конфіденційності. Безпека є одним із критичних аспектів функціонування хмарних ІС.

Вона реалізується через поєднання технологічних, організаційних і правових механізмів:

- шифрування даних при зберіганні (AES, RSA) та передачі (TLS, HTTPS);
- автентифікацію та авторизацію користувачів (OAuth 2.0, SAML, MFA);
- управління ідентичностями (Identity and Access Management, IAM);
- аудит активності та журналювання подій.

У хмарі діє модель спільної відповідальності (Shared Responsibility Model)**:** провайдер відповідає за безпеку інфраструктури, а користувач - за захист своїх даних, налаштування доступів і конфігурацій.

Інтероперабельність та інтеграція сервісів. Хмарні ІС часто взаємодіють із зовнішніми платформами, базами даних і корпоративними додатками.

Для цього використовуються стандартизовані протоколи (REST, SOAP, gRPC) та шини даних (Apache Kafka, RabbitMQ). Забезпечення інтероперабельності

дозволяє створювати складні багатокомпонентні системи, де частина сервісів розміщується в публічній хмарі, а частина - у приватній.

Моніторинг, аналітика та управління продуктивністю. Щоб підтримувати стабільну роботу хмарної ІС, використовуються системи моніторингу, такі як Prometheus, Grafana, Zabbix, Datadog. Вони забезпечують збір телеметричних даних у реальному часі, виявлення аномалій і прогнозування навантаження за допомогою машинного навчання (AIOps). Отримані метрики використовуються для автоматизованого управління ресурсами: якщо навантаження зростає, то система додає вузли, якщо падає, то зменшує кількість активних екземплярів.

Енергоефективність та екологічний аспект. Оскільки хмарні дата-центри споживають значні обсяги електроенергії, сучасні провайдери впроваджують технології енергоменеджменту, такі як:

- динамічне вимкнення невикористаних серверів;
- розподілене охолодження (liquid cooling, free air);
- використання відновлюваних джерел енергії;
- оптимізацію розміщення віртуальних машин з урахуванням енергоспоживання.

Це не лише зменшує експлуатаційні витрати, а й відповідає концепції **“**green IT**”** - сталого розвитку інформаційних технологій[11].

1.3 Проблеми та виклики управління ресурсами в хмарних системах

Управління ресурсами у хмарних середовищах є складним багаторівневим процесом, який охоплює планування, розподіл, моніторинг і оптимізацію використання обчислювальних потужностей, пам’яті, мережевих каналів і сховищ даних. Попри очевидні переваги хмарних технологій - масштабованість, гнучкість і зниження витрат - вони породжують низку викликів, пов’язаних із продуктивністю, доступністю, безпекою та економічною ефективністю [12].

Продуктивність і масштабованість. Однією з головних проблем управління ресурсами є забезпечення стабільної продуктивності при динамічних змінах навантаження. Хмарні сервіси обслуговують велику кількість користувачів, запити яких можуть змінюватися в десятки разів упродовж коротких періодів часу. Це створює потребу в ефективних механізмах масштабування та балансування навантаження.

Основні проблеми, які присутні при підвищенні продуктивності:

1. Нерівномірне використання ресурсів. Частина вузлів може бути перевантаженою, тоді як інші - недовантажені, що знижує ефективність усього кластера.

2. Невчасне масштабування. Недостатньо швидке реагування системи на пікові запити призводить до затримок і деградації продуктивності.

3. Конкуренція між користувачами. У мультиорендних системах різні додатки можуть боротися за однакові ресурси, що створює проблему "resource contention".

4. Обмеження пропускної здатності мережі. При розподіленні великих обсягів даних (наприклад, у Big Data аналітиці чи ML-тренуваннях) трафік між зонами доступності стає критичним фактором.

Для вирішення цих проблем застосовуються методи динамічного балансування навантаження, предиктивного масштабування (на основі машинного навчання) та адаптивного планування ресурсів, які враховують історичні патерни використання.

Доступність і надійність. Хмарна інфраструктура повинна гарантувати високу доступність сервісів (High Availability, HA) і відмовостійкість (Fault Tolerance), навіть за умови виходу з ладу окремих компонентів.

Однак досягнення цього балансу пов’язане з рядом технічних викликів, а саме:

1. Залежність від мережевої інфраструктури. Навіть короткочасні збої у з’єднанні можуть призвести до втрати доступу до критичних сервісів.

2. Помилки у віртуалізації або оркестрації. Некоректна робота гіпервізора або контейнерного оркестратора може паралізувати цілий кластер.

3. Складність реплікації та консистентності даних. Підтримання узгодженого стану даних між географічно розподіленими центрами вимагає складних алгоритмів синхронізації (наприклад, Paxos, Raft).

4. Проблема "єдиного центру відмови" (Single Point of Failure). Неправильно спроєктована архітектура може втратити працездатність через збій одного компонента.

Для підвищення доступності застосовуються кластеризація, георозподілення даних, автоматичне відновлення сервісів і тестування на стійкість до збоїв (Chaos Engineering). Управління ресурсами при цьому має враховувати резервування потужностей, політику реплікацій та SLA (Service Level Agreement) між провайдером і користувачем.

Безпека - одна з найсерйозніших проблем хмарних систем, оскільки дані користувачів зберігаються на сторонніх серверах і передаються через відкриті мережі. Основні виклики у сфері безпеки:

1. Неавторизований доступ та порушення конфіденційності. Через спільну інфраструктуру можливі витоки даних між орендарями.

2. Уразливості на рівні API. Публічні інтерфейси керування ресурсами можуть бути атаковані через слабку автентифікацію або недостатній контроль запитів.

3. Загрози віртуалізації. Атаки типу *VM* escape або side-channel можуть дозволити зловмиснику отримати доступ до сусідніх віртуальних машин.

4. Внутрішні загрози. Працівники провайдера або адміністратори з високим рівнем доступу можуть стати джерелом ризику.

5. Невідповідність стандартам безпеки. Організації мають дотримуватися вимог ISO/IEC 27001, GDPR, HIPAA, SOC 2 тощо[7],[8].

Ефективне управління безпекою передбачає використання шифрування даних, механізмів багатофакторної автентифікації (MFA), сегментації мережі, моніторингу подій безпеки (SIEM) і регулярних аудитів конфігурацій.

Важливо також чітко визначити межі відповідальності між користувачем і провайдером у межах моделі Shared Responsibility.

Вартість та економічна ефективність. Попри поширену думку, що хмарні технології знижують витрати, економічне управління ресурсами є однією з найскладніших задач. Проблема полягає не лише у зменшенні витрат, а у досягненні оптимального співвідношення між продуктивністю, надійністю та вартістю. Основні виклики:

1. Непередбачувані витрати. Модель "pay-as-you-go" може призвести до перевитрат при неплановому навантаженні.
2. Неоптимальне використання ресурсів. Запущені, але неактивні віртуальні машини або зайві копії даних створюють "мертві витрати" (idle costs).
3. Складність вибору тарифних планів. Провайдери (AWS, Azure, Google Cloud) пропонують десятки типів інстансів і моделей оплати, що ускладнює оптимізацію.
4. Витрати на безпеку та резервування. Реплікація, бекапи та відмовостійкі архітектури збільшують споживання ресурсів.

Для зменшення вартості використовуються підходи:

- оптимізація життєвого циклу ресурсів (lifecycle management);
- використання предиктивної аналітики для прогнозу навантажень;
- автоматичне вимкнення неактивних інстансів;
- використання “spot” та “reserved” інстансів у публічних хмарах.

1.4 Підходи до управління ресурсами

Ефективне управління ресурсами є центральною задачею функціонування хмарних систем. Під управлінням ресурсами розуміють сукупність процесів планування, моніторингу, розподілу, балансування, масштабування та оптимізації використання обчислювальних потужностей, пам’яті, мережевих каналів і сховищ даних з метою досягнення максимальної продуктивності при мінімальних

витратах. Управління ресурсами визначає якість роботи хмарної системи за такими критеріями, як рівень сервісу (SLA), ефективність використання інфраструктури, енергоощадність і стабільність продуктивності. У сучасній науковій літературі виділяють кілька груп підходів до реалізації цього процесу.

Класифікація підходів до управління ресурсами. Підходи до управління ресурсами в хмарних середовищах умовно поділяються за кількома критеріями:

1. За характером управління:

Статичні підходи (static management) - розподіл ресурсів виконується наперед, на основі попередньо визначених параметрів або профілів навантаження. Вони прості у реалізації, проте не враховують динамічні зміни системи. Застосовуються у стабільних або прогнозованих середовищах.

Динамічні підходи (dynamic management) - передбачають адаптацію розподілу ресурсів у реальному часі відповідно до зміни навантаження чи стану системи. Використовуються алгоритми контролю зворотного зв’язку, машинного навчання або евристичної оптимізації.

2. За рівнем автоматизації:

– Ручне управління (manual provisioning) - ресурси виділяються адміністратором, часто через консолі або API провайдера.

– Автоматизоване управління (automated resource management) - система самостійно приймає рішення щодо масштабування чи балансування, спираючись на метрики продуктивності та політики QoS.

3. За рівнем абстракції:

– Інфраструктурний рівень (IaaS) - управління віртуальними машинами, контейнерами, мережами та сховищами.

– Платформний рівень (PaaS) - управління середовищами виконання, фреймворками, базами даних.

– Прикладний рівень (SaaS) - управління користувацькими сесіями, навантаженням і SLA-договорами.

4. Евристичні та алгоритмічні підходи. Початкові системи управління ресурсами у хмарі базувалися на евристичних алгоритмах, які визначали правила розподілу ресурсів на основі фіксованих порогів або сценаріїв.

Приклади:

– Threshold-based scaling - масштабування відбувається, коли навантаження перевищує певний поріг (наприклад, CPU > 80%).

– Rule-based management - адміністратор задає набір умов (“if–then–else”) для виділення або звільнення ресурсів.

– Round Robin, Least Loaded, Priority Scheduling - алгоритми розподілу завдань між вузлами.

Перевагами є простота реалізації, передбачуваність результатів. Водночас, існують і недоліки даних підходів: слабка адаптивність до непередбачуваних змін і відсутність глобальної оптимізації.

Сучасні системи застосовують комбіновані евристичні схеми, які враховують поточні метрики навантаження, пріоритети задач і вимоги SLA.

5. Математичні та оптимізаційні підходи. Математичне управління ресурсами базується на формалізації задачі оптимізації. Мета - мінімізувати витрати або час виконання при обмежених ресурсах.

Типові моделі [19]:

– Лінійне та цілочисельне програмування (LP, ILP) - використовується для оптимізації розподілу завдань між віртуальними машинами;

– Стохастичні моделі (Markov Decision Processes, Queuing Theory) - моделюють поведінку системи з урахуванням імовірнісних змін навантаження;

– Методи динамічного програмування - використовуються для адаптивного розподілу ресурсів у часі;

– Енергетичні моделі - оптимізують споживання енергії дата-центрами.

Такі підходи забезпечують високий рівень точності, але часто мають високу обчислювальну складність, що обмежує їх практичне застосування в реальному часі.

6. Інтелектуальні та машинно-навчальні підходи. З поширенням Big Data і AI-технологій дедалі більшого значення набувають інтелектуальні підходи до управління ресурсами, які ґрунтуються на машинному навчанні, глибинних нейронних мережах та підкріплювальному навчанні (reinforcement learning) [6], [9].

Використовуються моделі:

– Regression та Time Series Forecasting - прогнозування майбутнього навантаження;

– Clustering (k-means, DBSCAN) - групування схожих профілів споживання ресурсів;

– Reinforcement Learning (Q-learning, DQN) - навчання системи приймати оптимальні рішення в реальному часі без участі людини;

– Neural Resource Controllers - нейронні мережі, що автоматично підбирають параметри масштабування.

Переваги підходів: адаптивність, здатність працювати в умовах невизначеності, самонавчання.

Недоліки підходів: потребують великих обсягів даних для тренування та складні у валідації.

7. Економічні та SLA-орієнтовані підходи. У багатьох хмарних середовищах розподіл ресурсів підпорядковується не лише технічним, а й економічним критеріям.

Використовуються ринкові та аукціонні моделі, де ресурси виступають товаром, а користувачі - споживачами, які конкурують за обчислювальні потужності.

Основні концепції:

– Utility-based resource allocation - виділення ресурсів пропорційно до вигоди користувача;

– Cost-aware scheduling - планування із врахуванням вартості використання різних інстансів;

– Dynamic pricing models (spot, on-demand, reserved) - дозволяють балансувати між вартістю і доступністю;

– SLA-driven management - дотримання рівня сервісу (доступність, затримки, продуктивність) як основного критерію прийняття рішень.

Такі підходи дозволяють інтегрувати технічні метрики ефективності з економічними параметрами, забезпечуючи баланс між якістю сервісу та фінансовими витратами.

8. Комплексні гібридні підходи. У практичних реалізаціях дедалі частіше використовуються гібридні підходи, які комбінують різні стратегії управління.

Наприклад:

1. спочатку застосовується евристика для швидкого початкового розподілу;
2. потім - алгоритм оптимізації або машинного навчання для тонкого налаштування;
3. далі - SLA-контроль і моніторинг продуктивності в режимі реального часу.

Такі системи забезпечують баланс між точністю, швидкодією та адаптивністю, що особливо важливо для великих розподілених хмарних інфраструктур.

1.5 Огляд сучасних платформ і технологій

Розвиток хмарних технологій супроводжується появою великої кількості платформ, сервісів та інструментів, що реалізують різні підходи до управління ресурсами - від базових інфраструктурних рішень до інтелектуальних систем автоматизації. Огляд сучасних платформ дозволяє зрозуміти, які технології на практиці втілюють концепції масштабованості, відмовостійкості, оркестрування, безпеки та оптимізації витрат.

Основні глобальні хмарні платформи. Сучасний ринок хмарних послуг представлений трьома провідними провайдерами - Amazon Web Services (AWS), Microsoft Azure та Google Cloud Platform (GCP) - на які припадає понад 70% світового ринку публічних хмарних послуг.

Amazon Web Services (AWS). AWS є піонером і найбільш масштабною екосистемою хмарних сервісів, що підтримує понад 200 рішень для зберігання, обчислень, аналітики та машинного навчання.

Основні компоненти управління ресурсами:

– EC2 (Elastic Compute Cloud) - віртуальні сервери з автоматичним масштабуванням;

– ECS / EKS - сервіси контейнеризації та оркестрації (Docker, Kubernetes);

– Auto Scaling Groups - автоматичне масштабування за навантаженням;

– CloudWatch - моніторинг і збирання метрик продуктивності;

– Cost Explorer / Trusted Advisor - аналітика вартості та рекомендації з оптимізації.

AWS активно використовує policy-driven management і machine learning-based autoscaling, що робить систему однією з найбільш адаптивних. Переваги: найширший набір сервісів, висока надійність, гнучка масштабованість, розвинена система моніторингу та фінансової аналітики. Недоліки: складність конфігурації для початківців, висока вартість при неправильному використанні, певна залежність від екосистеми Amazon.

Microsoft Azure. Платформа Azure забезпечує глибоку інтеграцію з корпоративними системами та продуктами Microsoft (Windows Server, Active Directory, SQL Server).

Основні сервіси:

– Azure Virtual Machines, Virtual Networks - управління віртуальними ресурсами;

– Azure Kubernetes Service (AKS) - оркестрація контейнерів;

– Azure Monitor і Application Insights - аналітика продуктивності;

– Azure Cost Management - контроль фінансових витрат;

– Azure Automation / Logic Apps - сценарії автоматизації управління ресурсами.

Azure відзначається високим рівнем безпеки та підтримкою гібридних сценаріїв через Azure Arc та Azure Stack.

Переваги: глибока інтеграція з корпоративними продуктами Microsoft, підтримка гібридних сценаріїв, високий рівень безпеки.

Недоліки: менша прозорість тарифів, складність у налаштуванні доступів і прав, деякі сервіси мають нижчу продуктивність у порівнянні з AWS.

Google Cloud Platform (GCP). GCP орієнтована на продуктивність, машинне навчання та аналітику великих даних.

Основні компоненти:

- Compute Engine - масштабовані віртуальні машини;
- Kubernetes Engine (GKE**)** - потужна реалізація контейнерної оркестрації;
- BigQuery - аналітична платформа для обробки великих обсягів даних;
- Cloud Operations (ex-Stackdriver) - моніторинг, логування, діагностика;
- AI Platform - автоматизоване управління ресурсами на основі моделей машинного навчання.

GCP вирізняється акцентом на інтелектуальне управління навантаженням **і** енергоефективність дата-центрів.

Переваги: оптимізація для аналітики й AI-моделей, висока швидкість обчислень, енергоефективні дата-центри.

Недоліки: обмежена кількість регіонів порівняно з конкурентами, складні інструменти білінгу, менше готових корпоративних інтеграцій.

Платформи оркестрації та контейнеризації. Для автоматизації управління ресурсами у хмарних системах ключову роль відіграють оркестратори контейнерів та платформи керування кластерами, які забезпечують динамічний розподіл навантаження, відмовостійкість і масштабування.

Kubernetes. Kubernetes - де-факто стандарт для оркестрації контейнерів у хмарних середовищах. Його архітектура включає master-вузол (контролер) та worker-вузли, на яких розгортаються контейнери.

Функції Kubernetes у контексті управління ресурсами:

- автоматичне масштабування подів і вузлів (Horizontal Pod Autoscaler, Cluster Autoscaler);
- контроль стану та самовідновлення контейнерів;
- розподіл навантаження через Services та Ingress;
- ізоляція ресурсів через Namespaces, Resource Quotas і LimitRanges.
- Kubernetes підтримується всіма основними хмарними провайдерами (AWS EKS, Azure AKS, Google GKE).

Переваги: автоматичне масштабування, стійкість до збоїв, відкритий стандарт і підтримка усіма провайдерами.

Недоліки: складність у розгортанні й адмініструванні, висока крива навчання, потреба в додаткових інструментах безпеки.

Docker Swarm і OpenShift. Docker Swarm - спрощена альтернатива Kubernetes, орієнтована на невеликі кластери. OpenShift (від Red Hat) - корпоративна платформа на базі Kubernetes, яка додає інструменти безпеки, DevOps і CI/CD. OpenShift активно застосовується для розгортання приватних і гібридних хмар.

Переваги Docker Swarm: простота конфігурації, легкість інтеграції з Docker-екосистемою.

Недоліки Docker Swarm: обмежена масштабованість і гнучкість у порівнянні з Kubernetes.

Переваги OpenShift: корпоративний рівень безпеки, інтеграція DevOps-процесів, зручна панель керування.

Недоліки OpenShift: високі системні вимоги, складність інсталяції та ліцензування.

Платформи приватних і гібридних хмар. OpenStack - відкрита платформа для побудови приватних і публічних хмар.

Основні модулі:

- Nova - керування обчислювальними ресурсами;
- Neutron - мережеві сервіси;
- Cinder / Swift - блочне та об’єктне сховище;

– Heat - оркестрація інфраструктури;

– Ceilometer - моніторинг і білінг.

OpenStack забезпечує повний контроль над інфраструктурою, що робить його популярним серед наукових і корпоративних центрів обчислень.

Переваги: відкритий код, гнучкість налаштування, можливість створення приватних хмар без прив'язки до вендора.

Недоліки: складність інтеграції, потреба у високій кваліфікації персоналу, складне оновлення модулів.

VMware vCloud / vSphere. Платформа VMware - одна з найрозвиненіших у сфері віртуалізації.

Вона підтримує інтелектуальні механізми розподілу ресурсів:

– Distributed Resource Scheduler (DRS) - балансування навантаження між віртуальними машинами;

– vMotion / Storage vMotion - міграція ресурсів без простоїв;

– vRealize Suite - аналітика, моніторинг, прогнозування і автоматизація.

Переваги: перевірена стабільність, потужні засоби віртуалізації, відмінна підтримка корпоративних середовищ.

Недоліки: висока вартість ліцензій, часткова закритість коду, обмежена сумісність із альтернативними хмарними технологіями.

Інструменти моніторингу та автоматизації. Системи управління ресурсами неможливі без засобів моніторингу, логування та аналітики, які забезпечують спостереження за станом інфраструктури та автоматичне реагування.

Основні сучасні інструменти:

– Prometheus - збір і зберігання метрик у часових рядах, використовується з Grafana;

– Grafana - візуалізація показників продуктивності;

– Zabbix / Nagios - моніторинг стану вузлів і сервісів;

– Terraform - інфраструктура як код (IaC);

– Ansible, Puppet, Chef - автоматизація конфігурацій;

– CloudFormation (AWS), ARM Templates (Azure), Deployment Manager (GCP) - нативні інструменти оркестрації хмарних ресурсів.

Тенденції розвитку технологій управління ресурсами. Сучасні дослідження та промислові рішення демонструють перехід від класичних моделей управління до інтелектуальних систем, що поєднують машинне навчання, прогнозування та політику самокерування (self-management).

Основні тренди:

– AIOps (Artificial Intelligence for IT Operations) - автоматизація адміністрування через аналіз великих обсягів операційних даних;

– Serverless-архітектури (Function as a Service) - ресурси виділяються лише під час виконання функцій (AWS Lambda, Azure Functions, Google Cloud Functions);

– Edge Computing - розподілення обчислень ближче до джерела даних;

– Green Cloud - енергоефективне управління ресурсами з урахуванням вуглецевого сліду [9],[11].

1.6 Постановка задачі та обґрунтування вибору напрямку дослідження

З розвитком хмарних технологій зростає кількість користувачів, обсяг даних і різноманітність навантажень на обчислювальні інфраструктури. Незважаючи на досягнення провідних хмарних платформ (AWS, Azure, GCP, OpenStack), проблема ефективного управління ресурсами залишається актуальною, враховуючи динамічний характер середовища, неоднорідність запитів, обмежену обчислювальну потужність та вимоги до якості обслуговування (QoS, SLA) [4], [5], [9]. Наявні методи часто мають обмежену адаптивність, зосереджені на фіксованих політиках або вимагають значного втручання людини в процес прийняття рішень.

Отже, ми формулюємо наукову проблему так: проблема полягає у розробці та вдосконаленні методів управління ресурсами інформаційних систем у хмарних середовищах, що забезпечують динамічний, адаптивний та економічно ефективний

розподіл обчислювальних ресурсів з урахуванням показників продуктивності, доступності та вартості.

Ця проблема має як теоретичне значення (розробка моделей управління в умовах невизначеності), так і практичне значення (оптимізація використання ресурсів у реальних хмарних платформах).

Мета дослідження. Метою дослідження є обґрунтування, розробка та вдосконалення методів управління ресурсами інформаційних систем у хмарних середовищах, що забезпечують оптимальний баланс між продуктивністю, надійністю, безпекою та вартістю обчислювальних послуг.

Досягнення цієї мети передбачає поєднання аналітичних, математичних та інтелектуальних методів моделювання, а також розробку практичних підходів до підвищення ефективності використання ресурсів у хмарній інфраструктурі.

Для досягнення поставленої мети необхідно вирішити наступні основні завдання:

1. Проаналізувати теоретичні основи та сучасний стан управління ресурсами в хмарних середовищах, зокрема, дослідити класифікацію, архітектури, технологічні платформи та типові проблеми.
2. Систематизувати підходи до управління ресурсами (евристичні, оптимізаційні, інтелектуальні, економічні) та визначити їхні переваги та недоліки для різних типів навантажень.
3. Визначити ключові критерії ефективного управління ресурсами: продуктивність, доступність, безпека, енергоефективність та вартість.
4. Розробити узагальнену модель або метод управління ресурсами, що враховує динамічний характер навантажень і дозволяє адаптивно балансувати обчислювальну потужність.
5. Створити алгоритмічну реалізацію або експериментальний прототип системи управління, що дозволяє оцінити ефективність запропонованого підходу.

6. Провести порівняльний аналіз результатів моделювання або тестування з існуючими рішеннями для перевірки доцільності запропонованої методології.

7. Сформулювати рекомендації щодо використання розроблених методів на практиці побудови хмарних інформаційних систем.

Вибір даної теми обумовлений стрімким зростанням обсягів даних, динамічними робочими навантаженнями та тенденцією до міграції бізнес- та урядових інформаційних систем до хмарної інфраструктури. Сучасні технологічні рішення (Kubernetes, OpenStack, Terraform, AWS Auto Scaling тощо) забезпечують лише базовий рівень автоматизації, але не враховують багатокритеріальне управління, яке передбачає одночасне задоволення вимог до продуктивності, безпеки та вартості.

Тому обраний напрямок досліджень - розробка адаптивних та інтелектуальних методів управління ресурсами в хмарних середовищах є актуальним як з наукової, так і з прикладної точки зору. Він спрямований на формування нових підходів до оптимізації використання ресурсів, що підвищить ефективність IT-послуг та зменшить операційні витрати.

В кінцевому результаті дослідження очікується:

– розробка структурно-логічної моделі управління ресурсами в хмарних середовищах;

– створення адаптивного або інтелектуального методу розподілу ресурсів, що забезпечує баланс між технічними та економічними показниками;

– розробка прототипу або програмного модуля, що реалізує запропонований метод у середовищі віртуалізації або контейнеризації;

– проведення експериментальної оцінки ефективності розробленого рішення та порівняння з існуючими підходами;

– підготовка рекомендацій щодо впровадження результатів у практику побудови корпоративних або публічних хмарних систем.

Висновки до розділу 1

У цьому розділі наведено теоретичний та методологічний аналіз архітектури, структурних особливостей та принципів функціонування хмарних інформаційних систем, а також розглянуто основні проблеми, підходи та сучасні технологічні рішення у сфері управління ресурсами.

Нами було:

1. Проаналізовано сутність і класифікацію хмарних обчислень. Визначено три рівні сервісних моделей - IaaS, PaaS та SaaS, що формують багаторівневу архітектуру хмарної інфраструктури. Кожна з них характеризується власними механізмами управління, рівнем абстракції та відповідальності користувача. Показано відмінності між моделями розгортання - публічною, приватною, гібридною та комунальною хмарами.

2. Досліджено структуру й особливості функціонування інформаційних систем у хмарних середовищах. Встановлено, що ключовими чинниками ефективності є віртуалізація, мультиорендність, автоматизоване управління інфраструктурою, балансування навантаження, забезпечення відмовостійкості та безпеки. Показано, що використання технологій контейнеризації, IaC (інфраструктури як коду) та AIOps забезпечує гнучкість і еластичність ІС.

3. Визначено проблеми та виклики управління ресурсами. Серед основних - підтримання продуктивності й масштабованості в умовах динамічних навантажень, забезпечення високої доступності та надійності, гарантування інформаційної безпеки, а також контроль економічної ефективності використання ресурсів. Ці чинники формують наукову проблему – необхідність розроблення адаптивних, багатокритеріальних і економічно збалансованих методів управління.

4. Систематизовано підходи до управління ресурсами. Виділено статичні та динамічні, евристичні, математичні, інтелектуальні, SLA-орієнтовані та гібридні підходи. Показано їх переваги та недоліки - від простоти реалізації до потреби у великих обсягах даних і високих обчислювальних витрат. Особливу

увагу приділено методам, що використовують машинне навчання та прогнозування навантажень.

5. Проведено огляд сучасних платформ і технологій. Проаналізовано провідні рішення - AWS, Azure, GCP, OpenStack, Kubernetes, Docker Swarm, OpenShift, VMware vSphere та інструменти Prometheus, Grafana, Terraform, Ansible. Узагальнено їхні переваги (масштабованість, інтеграція, безпека, енергоефективність) та недоліки (складність налаштування, вартість, обмеження сумісності). Показано перехід індустрії до AIOps, serverless та green cloud-технологій.

6. Сформульовано постановку задачі та обґрунтовано напрям дослідження. Визначено, що наукова проблема полягає у створенні методів, здатних забезпечити адаптивне та економічно обґрунтоване управління ресурсами в умовах змінного навантаження. Сформульовано мету, задачі й очікувані результати дослідження - розроблення моделі та алгоритму ефективного розподілу ресурсів, створення прототипу системи управління й оцінювання її ефективності.

Таким чином, у першому розділі закладено теоретичні основи для подальших досліджень, визначено актуальність і наукову новизну теми, систематизовано підходи та технології. Результати цього етапу стануть основою для розробки та експериментальної перевірки методів управління ресурсами у другому розділі магістерської роботи.

2 ТЕОРЕТИКО-МЕТОДИЧНА БАЗА ТА МАТЕМАТИЧНЕ МОДЕЛЮВАННЯ ПРОЦЕСІВ УПРАВЛІННЯ РЕСУРСАМИ

2.1 Управління ресурсами як багатокритеріальна задача системного аналізу

У сучасних хмарних інформаційних системах управління ресурсами розглядається не як ізольований технічний процес, а як складне багатокритеріальне завдання, що вимагає системного підходу до аналізу та прийняття рішень. На відміну від традиційних IT-інфраструктур, ресурси в хмарному середовищі є динамічними, розподіленими та взаємозалежними, а ефективність їх використання визначається сукупністю технічних, економічних та якісних показників[12],[17]. Тому систематичні методи аналізу є ключовим інструментом для формалізації, моделювання та оптимізації процесів управління ресурсами.

Системний аналіз розглядає хмарну інфраструктуру як цілісну систему, що складається з взаємопов’язаних підсистем:

- обчислювальної (CPU, GPU-ресурси, віртуальні машини, контейнери);
- пам’яті та сховищ даних;
- мережевої (пропускна здатність, маршрутизація, балансування навантаження);
- сервісної та оркестраційної (керування контейнерами, автоматизація процесів);
- економічної (витрати, тарифи, SLA-показники);
- безпекової (автентифікація, шифрування, аудит).

Кожна з підсистем має власні цілі та критерії ефективності, але рішення мають прийматися з урахуванням їх взаємного впливу. Наприклад, збільшення продуктивності може призвести до зростання вартості або енергоспоживання; підвищення рівня безпеки може вплинути на швидкість обробки даних [17], [19].

Багатокритеріальна суть управління ресурсами проявляється в необхідності одночасно оптимізувати кілька показників, яка представлена у таблиці 2.1[15], [17]:

Таблиця 2.1

Багатокритеріальна суть управління ресурсами

Критерій	**Сутність**	**Тип впливу**
Продуктивність (Performance)	Максимізація швидкодії обчислень, мінімізація часу відгуку	max
Надійність і доступність (Availability, Reliability)	Забезпечення безперервності сервісів і стійкості до збоїв	max
Безпека (Security)	Захист даних і контроль доступу	max
Вартість (Cost Efficiency)	Мінімізація витрат на інфраструктуру та енергію	min
Енергоефективність (Energy Efficiency)	Зниження споживання електроенергії й вуглецевого сліду	max
Гнучкість і масштабованість (Scalability)	Адаптація ресурсів до навантаження в реальному часі	max

Таким чином, задача управління ресурсами може бути подана як багатокритеріальна оптимізаційна модель[17]:

$$\begin{cases} \text{Мінімізувати } C = f_1(x) \quad (\text{вартість}) \\ \text{Максимізувати } P = f_2(x) \quad (\text{продуктивність}) \\ \text{Максимізувати } A = f_3(x) \quad (\text{доступність}) \\ \text{Максимізувати } S = f_4(x) \quad (\text{безпека}) \\ \text{Мінімізувати } E = f_5(x) \quad (\text{енергоспоживання}) \end{cases}, \tag{2.1}$$

за умовами обмежень:

$$g_i \leq b_i,\ i = 1 \ldots n$$

де x - вектор рішень, що описує розподіл обчислювальних, пам’ятних і мережевих ресурсів.

Для вирішення багатокритеріальних задач у системному аналізі застосовуються такі методи:

1. Метод зважених коефіцієнтів - перетворення багатокритеріальної задачі на однокритеріальну через ваги ω_i для кожного показника [17].
2. Метод Парето-оптимальності - пошук рішень, що не можуть бути покращені за одним критерієм без погіршення іншого.
3. Методи аналізу ієрархій (AHP) - структуризація критеріїв і визначення пріоритетів за експертними оцінками [13].
4. Методи нечіткої логіки (Fuzzy Logic) - використання лінгвістичних змінних («висока продуктивність», «низька вартість») для прийняття рішень у невизначених умовах[14].
5. Інтелектуальні методи (ML, RL, Genetic Algorithms) - застосування машинного навчання й еволюційних алгоритмів для пошуку оптимального балансу між критеріями в реальному часі [20].

Приклад структури системної моделі управління ресурсами розглянемо далі.

У спрощеному вигляді модель можна подати як багаторівневу структуру:

– Вхідні дані: метрики навантаження, стан вузлів, вартість, енергоспоживання, SLA-параметри.

– Аналітичний блок: системний аналіз стану ресурсів, визначення вузьких місць, прогноз навантаження.

– Модуль прийняття рішень: оптимізація розподілу ресурсів на основі обраного методу (евристичного, оптимізаційного чи інтелектуального).

– Блок реалізації: автоматичне масштабування, міграція контейнерів, балансування трафіку.

– Зворотний зв’язок: моніторинг ефективності, самонавчання системи.

Ця модель відповідає концепції self-adaptive systems, де керування ресурсами виконується циклічно (аналогічно підходу MAPE-K: Monitor–Analyze–Plan–Execute–Knowledge) [10], [13].

Аналітична оцінка ефективності

Ефективність управління визначається через інтегральний критерій якості[17], [18]:

$$\{Q = \sum_{i=1}^{n} w_i \cdot K_i \ , \tag{2.2}$$

де K_i - нормалізовані значення критеріїв (0–1), а w_i - їхні вагові коефіцієнти.

У практичних експериментах ваги можуть визначатися емпірично (на основі статистики або експертного опитування) для конкретного типу хмарної системи (наприклад, приватної корпоративної чи публічної інфраструктури).

2.2 Методи моделювання та прийняття рішень

Управління ресурсами інформаційних систем у хмарних середовищах вимагає не тільки багатокритеріального аналізу, але й створення моделей, що відображають взаємозв'язки між параметрами системи, навантаженням, політиками обслуговування та критеріями продуктивності.

Моделювання є необхідним етапом для прогнозування поведінки системи, оптимізації розподілу ресурсів та прийняття рішень у режимі реального часу. У динамічному хмарному середовищі прийняття рішень має ґрунтуватися на адаптивних, аналітичних та інтелектуальних методах, здатних функціонувати в умовах невизначеності.

Методи моделювання у сфері управління хмарними ресурсами поділяються на аналітичні, імітаційні та інтелектуальні (рисунок 2.1)

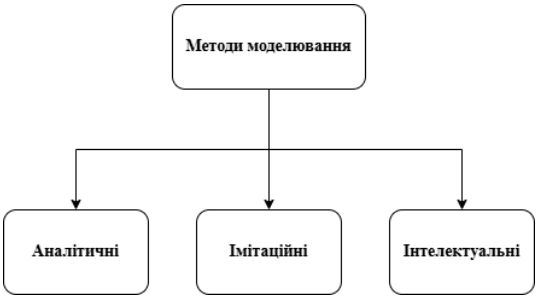


Рис. 2.1 Методи моделювання

Аналітичні моделі базуються на математичному описі процесів функціонування системи у вигляді рівнянь і функцій.

Типові підходи:

– Лінійне, нелінійне та стохастичне програмування - формалізація задачі оптимізації з обмеженнями;

– Теорія черг (Queuing Theory) - моделювання процесів обробки запитів, прогнозування часу відгуку та завантаженості вузлів;

– Марковські процеси - опис імовірнісних переходів між станами системи, що корисно для моделювання відмов і відновлення вузлів;

– Математичне прогнозування (Time Series) - оцінка динаміки навантаження й попиту на ресурси.

Переваги аналітичного моделювання - висока точність і формальна перевірюваність результатів; недоліки - складність опису реальних нелінійних процесів і масштабних систем.

Імітаційні моделі дозволяють відтворювати поведінку хмарної системи у віртуальному середовищі. Використовуються спеціалізовані платформи: CloudSim, iCanCloud, SimGrid, GreenCloud, які дозволяють задавати параметри віртуальних машин, політики балансування навантаження, алгоритми масштабування тощо.

Переваги:

– можливість відтворення складних сценаріїв роботи без впливу на реальну інфраструктуру;

– дослідження ефекту змін політик управління;

– гнучкість і наочність.

Недоліки: висока трудомісткість і потреба у валідації моделі.

Інтелектуальні моделі поєднують аналітичні методи з алгоритмами машинного навчання, що дозволяє системі самостійно коригувати рішення залежно від нових даних.

Типові підходи:

– Нейронні мережі (ANN, CNN, LSTM) - прогнозування навантаження, автоматичне масштабування;

– Підкріплювальне навчання (Reinforcement Learning) - навчання оптимальної політики розподілу ресурсів через взаємодію з середовищем;

– Генетичні алгоритми (GA) - пошук глобально оптимальних конфігурацій ресурсів;

– Fuzzy-моделі - прийняття рішень у нечітких або неповних даних (наприклад, коли метрики продуктивності суперечливі).

Процес прийняття рішень у системі управління ресурсами складається з кількох етапів:

1. Збір даних - моніторинг навантаження, стану вузлів, споживання енергії, вартості.
2. Аналіз і прогнозування - оцінка тенденцій, виявлення вузьких місць.
3. Генерація альтернатив - формування можливих рішень (збільшити ресурси, перемістити контейнери, змінити тариф).
4. Оцінка альтернатив - розрахунок інтегральних показників ефективності.
5. Вибір оптимального рішення - за заданими критеріями (вартість, SLA, продуктивність).
6. Реалізація та контроль - виконання рішень і зворотний зв’язок.

Методи прийняття рішень при визначеності застосовуються, коли параметри системи відомі або прогнозовані.

Типові методи:

– Метод зважених сум (Weighted Sum Method) - кожен критерій має вагу і вибирається альтернатива з максимальним інтегральним балом:

– Метод корисності (Utility Function) - побудова функції задоволення від рішення

Методи прийняття рішень при невизначеності - використовуються у випадках, коли система стикається з неповною або стохастичною інформацією.

Основні методи:

– Байєсівські мережі (Bayesian Networks) - побудова моделей умовних залежностей між параметрами (наприклад, навантаження ↔ відмови ↔ затримка).

– Теорія ігор - моделювання поведінки конкуруючих сервісів або клієнтів за обмежених ресурсів.

– Нечітка логіка (Fuzzy Logic) - рішення типу «якщо навантаження високе і час відгуку середній - збільшити ресурси на 20%».

– Підкріплювальне навчання (RL) - агент навчається шляхом проб і помилок, отримуючи винагороду за ефективне управління (модель типу Q-learning).

Більшість сучасних систем управління ресурсами в хмарі побудовані за моделлю MAPE-K (Monitor–Analyze–Plan–Execute–Knowledge).

Цей підхід реалізує замкнутий цикл самокерування (рисунок 2.2):

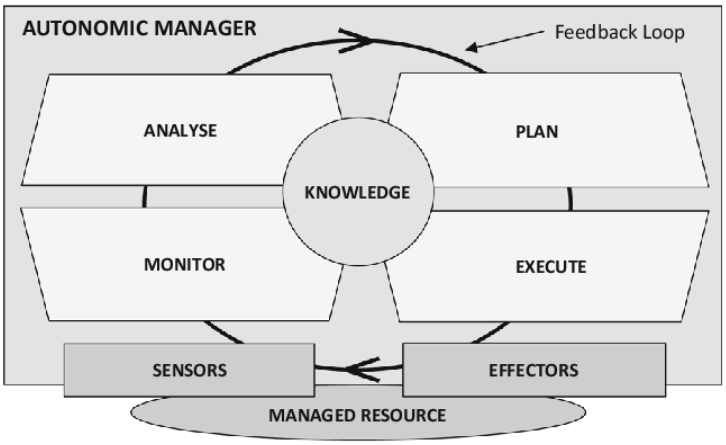


Рисунок 2.2 Цикл самокерування MAPE-K

Цикл включає в себе такі кроки:

– Monitor - збір телеметричних даних;

– Analyze - аналіз відхилень і прогнозування майбутнього навантаження;

– Plan - розробка рішень (масштабування, міграція, перерозподіл);

– Execute - виконання змін через API або оркестратор;

– Knowledge - база знань, що накопичує історію станів і рішень для подальшого навчання.

Завдяки циклу MAPE-K система може самостійно адаптуватися до змін середовища без втручання оператора[20].

Ефективне управління ресурсами у хмарних середовищах потребує не лише побудови моделей, але й формалізованих методів вибору оптимальних рішень. До таких методів належать AHP, TOPSIS, методи прогнозування навантаження, а також експертні оцінки, які дозволяють поєднати кількісні й якісні показники.

Метод AHP (Analytic Hierarchy Process) дозволяє структуризувати задачу вибору оптимального рішення у вигляді ієрархії цілей, критеріїв і альтернатив.

Послідовність застосування методу:

1. Формується ієрархічна структура (мета → критерії → альтернативи);
2. Виконується попарне порівняння критеріїв за шкалою 1–9 (від “рівнозначні” до “абсолютно важливі”);
3. Обчислюються вагові коефіцієнти критеріїв через нормалізацію власних векторів матриці порівнянь.

Для кожної альтернативи розраховується інтегральний показник корисності[13], [18]:

$$Q_j = \sum_{i=1}^{n} w_i \cdot K_{ij} \ , \qquad (2.3)$$

де w_i – вага критерію, K_{ij} – оцінка альтернативи за критерієм.

Переваги AHP - простота застосування та можливість поєднання кількісних і якісних факторів; недоліки - суб’єктивність експертних оцінок.

Метод TOPSIS (Technique for Order of Preference by Similarity to Ideal Solution)

Метод TOPSIS базується на ідеї, що оптимальне рішення має бути максимально близьким до ідеального і водночас максимально далеким від антиідеального.

Послідовність дій:

1. Формується матриця рішень $X = [x_{ij}]$

2. Виконується нормалізація та зважування за коефіцієнтами w_i.

3. Визначаються ідеальні позитивні та негативні рішення:

$$A^+ = \{\max(x_{ij})\}, A^- = \{\min(x_{ij})\}$$

4. Обчислюються відстані до ідеального й антиідеального рішень[13]:

$$D_j^+ = \sqrt{\sum w_i\,(x_{ij} - A_i^+)^2},\ D_j^- = \sqrt{\sum w_i\,(x_{ij} - A_i^-)^2}\ , \quad (2.4)$$

5. Розраховується коефіцієнт близькості[18]:

$$C_j = \frac{D_j^-}{D_j^+ + D_j^-}\ , \quad (2.5)$$

Альтернатива з найбільшим C_j вважається найкращою.

TOPSIS ефективно використовується для оцінки стратегій розподілу ресурсів за кількома критеріями (вартість, продуктивність, енергоспоживання, SLA).

Для підтримки прийняття рішень у реальному часі важливим є прогноз майбутніх навантажень на ресурси.

Основні групи:

– Статистичні моделі (ARIMA, Exponential Smoothing) - базуються на часових рядах;

– Машинне навчання - нейронні мережі (LSTM, CNN), регресійні моделі;

– Гібридні підходи - поєднання аналітичних і навчальних моделей для підвищення точності.

Прогнозування дозволяє заздалегідь планувати масштабування інфраструктури, балансувати навантаження та оптимізувати вартість.

У задачах, де кількісні дані відсутні або неповні (наприклад, оцінка ризиків, рівня безпеки чи впливу людського фактору), застосовується метод експертних оцінок[19].

Основні кроки:

1. Формування групи експертів (5–15 осіб);
2. Проведення анонімного або колективного опитування;
3. Узгодження результатів (метод Дельфі або коефіцієнт конкордації Кендалла);
4. Формування узагальнених ваг критеріїв.

Експертні методи часто комбінують з AHP чи TOPSIS для уточнення вагових коефіцієнтів[20].

2.3 Формалізація задачі оптимізації ресурсів у хмарному середовищі

Принципи управління ресурсами та методи прийняття рішень, розглянуті в попередніх розділах, є основою для побудови формальної математичної моделі, яка дозволяє визначити оптимальний розподіл ресурсів у хмарному середовищі. Формалізація проблеми необхідна для реалізації алгоритмічного підходу, що забезпечує баланс між продуктивністю, надійністю, вартістю та енергоефективністю. Оптимізація в хмарних системах є багатокритеріальною за своєю природою, оскільки враховує одночасно кілька взаємопов'язаних факторів [17], [19].

Загальна постановка задачі виглядає наступним чином:

1. Хмарна система описується множиною віртуальних ресурсів:

$$R = \{r_1, r_2, \dots, r_n\}$$

2. Множиною завдань (віртуальних машин, контейнерів, сервісів):

$$T = \{t_1, t_2, \dots, t_m\}$$

3. Необхідно знайти оптимальне відображення (розподіл)

$$f: T \rightarrow R$$

яке забезпечує мінімізацію або максимізацію визначених функцій ефективності при дотриманні обмежень на ресурси.

Загальний вигляд багатокритеріальної задачі оптимізації управління ресурсами[15]:

$$\begin{cases} \min f_1(x) = C(x) - \text{витрати на обчислення} \\ \min f_2(x) = P(x) - \text{продуктивність} \\ \min f_3(x) = A(x) - \text{доступність сервісу} \\ \min f_4(x) = E(x) - \text{енергоспоживання} \\ \min f_5(x) = S(x) - \text{рівень безпеки та ізоляції} \end{cases}, \quad (2.6)$$

де $x = \{x_{ij}\}$ - матриця розподілу завдань t_i між ресурсами r_j.

Функції $f_i(x)$ описують цільові показники, що залежать від стану інфраструктури та політик управління.

Задача оптимізації виконується при дотриманні технічних і організаційних обмежень[17]:

$$\begin{cases} \sum_{i=1}^{m} CPU_i \cdot x_{ij} \le CPU_j^{max}, \ j = 1..n \\ \sum_{i=1}^{m} RAM_i \cdot x_{ij} \le RAM_j^{max}, \ j = 1..n \\ \sum_{i=1}^{m} CPU_i \cdot x_{ij} \le Storage_j^{max}, \ j = 1..n \\ x_{ij} \in \{0,1\}, \ \sum_{j=1}^{n} x_{ij} = 1, \ i = 1 \dots m \end{cases}, \quad (2.7)$$

Ці обмеження відображають балансування між ресурсами серверів, пам’яті та сховищ у межах допустимих параметрів.

Залежно від типу функцій $f_i(x)$ і структури обмежень використовуються такі підходи:

Аналітичні (детерміновані) методи

Лінійне та цілочисельне програмування (Simplex, Branch & Bound);

Мультицільова оптимізація з перетворенням критеріїв у єдиний інтегральний показник[15],[17]:

$$F(x) = \sum_{i=1}^{k} w_i \cdot f_i(x) \ , \quad (2.8)$$

де w_i — вагові коефіцієнти важливості критеріїв.

До евристичних методів віносять:

- Генетичні алгоритми (GA);
- Алгоритм рою частинок (PSO);
- Методи імітаційного відпалу (Simulated Annealing);
- Табу-пошук (Tabu Search).

Ці методи ефективні для великих кластерів, де точне рішення обчислювально складне.

До інтелектуальних методів віносять:

- Підкріплювальне навчання (Reinforcement Learning, Q-learning);
- Нейронні мережі для прогнозу навантажень;
- Гібридні підходи ML + евристика (наприклад, DQN-оптимізація).

Такі підходи дозволяють системі самостійно навчатися та адаптувати параметри управління до змін умов роботи.

Для практичної реалізації багатокритеріальної моделі застосовується інтегральний показник[17]:

$$Q(x) = \alpha_1 P(x) + \alpha_2 A(x) + \alpha_3 S(x) - \alpha_4 C(x) - \alpha_5 E(x) \quad , \quad (2.9)$$

де коефіцієнти α_1 визначають пріоритети користувача чи адміністратора (наприклад, для корпоративної хмари - висока безпека, для публічної – низька вартість).

Максимізація $Q(x)$ відповідає пошуку Парето-оптимального балансу між технічними й економічними критеріями.

Модель оптимізації реалізується у вигляді функціональних модулів:

- Моніторинг стану ресурсів - збір поточних метрик навантаження (CPU, RAM, мережа).

– Аналітичний модуль - прогнозування майбутніх навантажень.

– Оптимізатор - розв’язання задачі за обраним методом (LP, GA, RL).

– Оркестратор - застосування рішень (масштабування, міграція, балансування).

– Зворотний зв’язок - контроль результату та корекція моделі.

Архітектура процесу оптимізації зображена на рисунку 2.3[20].

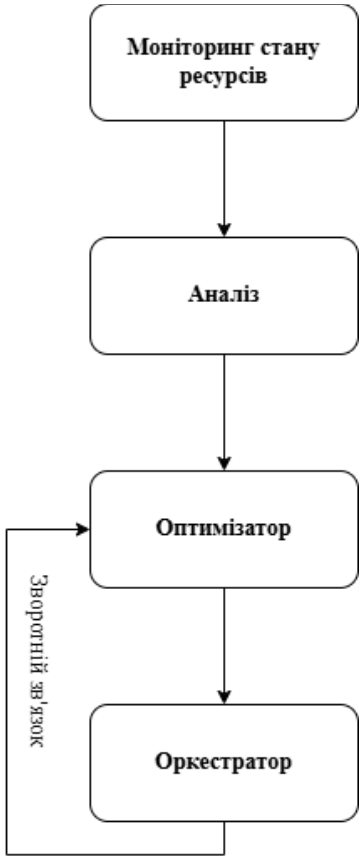


Рисунок 2.3 Узагальнена схема процесу оптимізації

Ця структура відповідає циклу MAPE-K, розглянутому в попередньому підрозділі. Розглянемо спрощену систему з двома серверами (R_1, R_2) та чотирма завданнями (T_1–T_4). Необхідно розподілити завдання між серверами так, щоб забезпечити оптимальний баланс між продуктивністю, вартістю та навантаженням.

Розподіл завдань можна подати у вигляді матриці призначення:

$$x_{ij} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad (2.10)$$

де: $i = 1..4$ - завдання (T_1–T_4); $j = 1..2$ - сервери (R_1, R_2); $x_{ij} = 1$, якщо завдання t_i виконується на сервері r_j.

Оптимізаційна мета визначається як максимізація інтегрального показника ефективності:

$$maxQ(x) = 0.4P(x) + 0.3A(x) + 0.2S(x) - 0.1C(x) \ , \quad (2.11)$$

де: $P(x)$ – продуктивність системи; $A(x)$ - доступність; $S(x)$ – рівень безпеки; $C(x)$ – витрати.

Така модель дозволяє оцінити, як зміна розподілу завдань між серверами впливає на інтегральну ефективність системи. При зміні навантаження або появі нових завдань оптимізатор повторно обчислює $Q(x)$ і виконує перерозподіл у реальному часі, що забезпечує адаптивність хмарного середовища.

2.4 Розробка математичної моделі управління ресурсами

Математична модель управління ресурсами є ядром системи оптимізації в хмарних середовищах. Його метою є формалізація взаємозв'язку між навантаженням, продуктивністю, вартістю та іншими параметрами, забезпечуючи адаптивне прийняття рішень в динамічних умовах. На відміну від статичних схем планування, така модель повинна забезпечувати самоналаштування, стійкість до змін та баланс між технічними та економічними критеріями ефективності.

Процес управління ресурсами у хмарному середовищі описується трьома рівнями взаємодії.

Вхідний рівень - моніторинг параметрів:

$$I(t) = \{L(t), U(t), E(t), C(t)\} \ , \quad (2.11)$$

де: $L(t)$ – навантаження; $U(t)$ – використання CPU/RAM; $E(t)$ – енергоспоживання; $C(t)$ – поточна вартість ресурсів;

Рівень прийняття рішень - математичний апарат оптимізації, що визначає нову конфігурацію ресурсів X(t+1) на основі стану системи I(t):

$$X(t+1) = f(I(T), W) \ , \tag{2.12}$$

де W - матриця вагових коефіцієнтів для критеріїв ефективності.

Вихідний рівень - реалізоване рішення (масштабування, балансування, міграція), яке оцінюється функцією інтегральної якості:

$$Q(t) = \alpha_1 P(t) + \alpha_2 A(t) + \alpha_3 S(t) - \alpha_4 C(t) - \alpha_5 E(t) \ , \tag{2.13}$$

Головна мета управління - максимізувати інтегральну ефективність системи за обмежених ресурсів:

$$\max_{X(t)} Q(t) = \sum_{i=1}^{n} w_i \cdot K_i(X(t)) \ , \tag{2.14}$$

де w_i - вагові коефіцієнти критеріїв (визначаються методом AHP або експертною оцінкою); $K_i(X(t))$ - нормалізовані значення критеріїв ефективності (0–1); $X(t)$ - поточна конфігурація ресурсів.

Критерії можуть включати:

K_1 - продуктивність (throughput або CPU utilization);

K_2 - доступність сервісів (uptime);

K_3 - безпеку (рівень доступу, наявність шифрування);

K_4 - вартість використання;

K_5 - енергоспоживання.

Для забезпечення узгодженості критеріїв усі показники нормалізуються:

$$K_i^{norm} = \frac{K_i - K_i^{min}}{K_i^{max} - K_i^{min}} \quad , \tag{2.15}$$

або для мінімізованих параметрів (вартість, енергоспоживання):

$$K_i^{norm} = 1 - \frac{K_i - K_i^{min}}{K_i^{max} - K_i^{min}} \quad , \tag{2.16}$$

У хмарних середовищах параметри змінюються з часом, тому доцільно описати процес управління як динамічну систему:

$$X(t+1) = X(t) + \Delta X(t)$$
$$\Delta X(t) = F(L(t), P(t), E(t), Q(t)) \, , \tag{2.16}$$

де $\Delta X(t)$ - зміна розподілу ресурсів унаслідок нових умов навантаження або політики SLA.

Функція F(·)може бути побудована як:

– лінійна модель керування, якщо залежності відомі аналітично;

– нейронна мережа або RL-агент, якщо система навчається на основі даних (self-learning optimization);

Хмарна система повинна підтримувати Service Level Agreement (SLA) - тобто заданий рівень якості обслуговування.

Це вираховується через обмеження:

$$T_{resp} \leq T_{SLA}, \; A(t) \geq A_{SLA}, \; P_{loss} \leq P_{max} \quad , \tag{2.17}$$

де T_{resp} - середній час відгуку; A – доступність; P_{loss} - втрати продуктивності через перевантаження.

Таким чином, модель зберігає баланс між оптимізацією ресурсів і дотриманням SLA.

На практиці управління реалізується як замкнений цикл:

1. Збір даних - моніторинг метрик;
2. Оцінка поточного стану I(t);
3. Оптимізаційний розрахунок нової конфігурації X(t+1);
4. Перерозподіл ресурсів (через оркестратор);
5. Оцінка результату (порівняння Q(t+1) з попереднім Q(t);
6. Самонавчання моделі (оновлення параметрів W).

Алгоритм функціонує за принципом self-adaptive feedback loop, аналогічно концепції MAPE-K, але розширений блоком математичної оптимізації.

Приклад узагальненої моделі розглянемо далі.

Для системи з n віртуальних машин та m ресурсних вузлів[19]:

$$\max Q = \sum_{i=1}^{n} \sum_{j=1}^{m} w_1 P_{ij} + w_2 A_{ij} + w_3 S_{ij} - w_4 C_{ij} - w_5 E_{ij} \quad , \qquad (2.17)$$

За умов [12], [13], [20]:

$$\sum_{i=1}^{n} CPU_{ij} \leq CPU_j^{max} \, , \; \sum_{i=1}^{n} RAM_{ij} \leq RAM_j^{max} \, , x_{ij} \in \{0,1\} \quad , \qquad (2.18)$$

Модель може бути розв’язана методами[13], [14], [16]:

– лінійного програмування для малих систем;

– евристичної оптимізації (GA, PSO) для великих кластерів;

– інтелектуального керування (RL, нейромережі) для адаптивних сценаріїв.

Розроблена математична модель має забезпечувати:

– Адаптивність - автоматичне реагування на зміни навантаження;

– Масштабованість - можливість роботи в кластерних і гібридних хмарах;

– Оптимальність - мінімізацію витрат при максимальній продуктивності;

– Надійність - врахування SLA та резервування ресурсів;

– Самонавчання - корекцію параметрів на основі накопичених даних.

2.5 Вибір та обґрунтування алгоритму оптимізації

Розроблена математична модель управління ресурсами вимагає вибору ефективного алгоритму оптимізації, який забезпечить збалансований розподіл ресурсів при змінних навантаженнях і обмежених обчислювальних потужностях. Алгоритм повинен вміти працювати в режимі реального часу, враховувати багатокритеріальний характер завдання, динамічність середовища і можливість неповної інформації.

У практиці управління ресурсами у хмарних середовищах застосовуються три основні класи алгоритмів:

– оптимізаційні (аналітичні)**,**

– імітаційні**,**

– евристичні та інтелектуальні (ML / AI).

Кожен із них має власну область ефективності, що визначається структурою задачі, обсягом даних і вимогами до швидкодії.

Оптимізаційні алгоритми

Ця група алгоритмів базується на математичному аналізі і дозволяє знаходити точні або квазіточні рішення задач багатокритеріальної оптимізації.
До найпоширеніших методів належать:

1. Лінійне та цілочисельне програмування

Методи застосовуються, коли функції цілі та обмеження можна лінійно апроксимувати[17]:

$$\max F(x) = \sum_{i=1}^{n} w_i f_i(x), \;\; g_j(x) \le b_j \;\; , \tag{2.19}$$

Переваги: швидкість, точність, простота реалізації.

Недоліки: обмежена придатність для нелінійних або стохастичних систем.

2. Стохастичні моделі (Марковські процеси, теорія черг)

Дозволяють описати поведінку системи при випадковому навантаженні:

$$P_{ij}(t+1) = \; P_{ij}(t) \cdot M_{ij} \;\; , \tag{2.20}$$

де M_{ij} - матриця ймовірностей переходів між станами системи. Такі моделі добре підходять для прогнозування часу відгуку, рівня черг і доступності сервісів.

3. Методи множників Лагранжа

Використовуються для задач з обмеженнями, де потрібно досягти компромісу між критеріями. Дає аналітичний опис умов оптимуму, але складний для реалізації в системах із великим числом змінних.

4. Імітаційні алгоритми

Імітаційні методи дозволяють відтворювати роботу хмарного середовища у вигляді моделі, де перевіряються різні політики управління.

Основні підходи: CloudSim, iCanCloud, GreenCloud.

Ці інструменти дозволяють задати параметри віртуальних машин, серверів, SLA та протестувати алгоритми масштабування. Результатом є емпіричні залежності між продуктивністю, вартістю і енергоспоживанням[24].

5. Монте-Карло симуляції

Використовуються для оцінки ризиків і випадкових відхилень параметрів системи:

$$E[f(x)] = \frac{1}{N}\sum_{i=1}^{N} f(x_i) \;\; , \tag{2.21}$$

Цей метод добре працює при невизначеності, але потребує великої кількості експериментів.

6. Динамічне імітаційне моделювання

Дозволяє враховувати часову еволюцію стану системи. На кожному кроці часу виконується оптимізаційний цикл (аналіз → прийняття рішення → реалізація).

7. Евристичні та інтелектуальні алгоритми

Евристичні методи шукають наближене рішення задачі, використовуючи логіку, аналогії або біологічні принципи. Вони особливо ефективні для великих, складних або нелінійних систем.

8. Генетичні алгоритми (GA)

Засновані на принципах природного добору. Алгоритм працює з популяцією рішень, виконуючи операції відбору, схрещення та мутації[16]:

$$x_{new} = crossover(x_i, x_j) + mutation(\sigma) \ , \tag{2.22}$$

Генетичні алгоритми підходять для багатовимірних задач розподілу ресурсів і добре масштабується.

9. Алгоритм рою частинок (PSO)

Моделює колективну поведінку частинок, які рухаються до оптимуму за формулою[21]:

$$\vartheta_i^{t+1} = \omega\vartheta_i^t + c_1 r_1(p_i - x_i^t) + c_2 r_2(g - x_i^t)$$
$$x_i^{t+1} = x_i^t + \vartheta_i^{t+1} \ , \tag{2.23}$$

PSO забезпечує швидку збіжність, але може застрягати в локальному мінімумі.

10. Імітаційний відпал (Simulated Annealing)

Наслідує процес охолодження металів; поступово зменшує «температуру» для стабілізації рішення. Добре працює для задач з великою кількістю локальних мінімумів[22].

11. Машинне навчання (ML)

Використовується для прогнозування навантаження і самоадаптації політик.

Типові підходи:

– Нейронні мережі (ANN, LSTM) – прогноз CPU/RAM;

– Підкріплювальне навчання (Reinforcement Learning) – навчання політики $\pi(\alpha|s)$ для максимізації нагороди R;

– Decision Trees / Random Forest – вибір оптимальної конфігурації за історичними даними.

12. Гібридні підходи

Найефективнішими вважаються гібридні алгоритми, що поєднують точність аналітичних і гнучкість евристичних методів:

– GA + AHP/TOPSIS — генетичний пошук з ваговими коефіцієнтами критеріїв;

– PSO + ML-прогнозування — використання PSO для оперативного розподілу та нейронних мереж для прогнозу навантаження;

– RL + CloudSim — навчання агента на симульованому середовищі для оптимізації

– SLA.

Такі моделі забезпечують адаптивність, самонавчання і стійкість до коливань навантаження[20].

Проведемо порівняння усіх алгоритмів(таблиця 2.2):

Таблиця 2.2

Порівняльна характеристика алгоритмів

Клас алгоритму	**Приклад**	**Переваги**	**Недоліки**	**Область застосування**
Оптимізаційні	ЛП, Лагранж	Точність, формальність	Погана масштабованість	Малі системи
Імітаційні	CloudSim, Monte-Carlo	Реалізм, аналіз сценаріїв	Велика складність	Апробація методів
Евристичні	GA, PSO, SA	Гнучкість, наближеність	Може застрягати в локальних мінімумімах	Великі системи

продовження таблиці 2.2

Клас алгоритму	**Приклад**	**Переваги**	**Недоліки**	**Область застосування**
Інтелектуальні (ML)	RL, LSTM	Самонавчання, адаптивність	Потребує великих даних	Автономне керування
Гібридні	GA+ML, PSO+RL	Компроміс між точністю та швидкодією	Складність реалізації	Реальні хмарні платформи

Для дослідження та апробації у даній роботі доцільно обрати евристично-інтелектуальний підхід, що поєднує генетичний алгоритм (GA) та елементи підкріплювального навчання (RL).

Таке поєднання дозволяє:

– адаптивно реагувати на зміни навантаження;

– одночасно оптимізувати кілька критеріїв (вартість, SLA, енергоспоживання);

– поступово покращувати рішення шляхом самонавчання.

Алгоритм має високу гнучкість і придатність для симуляційних досліджень (CloudSim, SimGrid), що робить його оптимальним вибором для реалізації моделі управління ресурсами.

Висновки до розділу 2

У другому розділі проведено аналітико-дослідницьке обґрунтування методів управління ресурсами інформаційних систем у хмарному середовищі, побудованих на принципах системного аналізу, моделювання та математичної оптимізації. На основі системного підходу встановлено, що управління ресурсами є багатокритеріальним завданням, яке враховує техніко-економічні та якісні

показники - продуктивність, доступність, безпеку, енергоефективність та вартість. Розроблено структуру критеріїв та показано взаємозалежність між ними.

Здійснено класифікацію методів моделювання та прийняття рішень у системах управління ресурсами. Зокрема, було розглянуто аналітичний, симуляційний та інтелектуальний підходи. Визначено, що найбільш ефективними для складних, динамічних та невизначених умов є інтелектуальні моделі, які поєднують машинне навчання, нечітку логіку та евристичні алгоритми.

Ми сформулювали формальну постановку задачі оптимізації ресурсів, яка враховує обмеження продуктивності, вартості, SLA та енергоспоживання. На її основі побудовано математичну модель управління ресурсами, яка описує процеси прийняття рішень на трьох рівнях - моніторингу, оптимізації та реалізації - з урахуванням динаміки стану системи.

Запропоновано функцію, яка інтегрує ключові критерії та кількісно визначає якість рішень.

Модель реалізує принцип самоадаптивного керування (self-adaptive feedback loop) на основі циклу MAPE-K і забезпечує стійкість до змін середовища.

Проаналізовано основні класи алгоритмів оптимізації - оптимізаційні, імітаційні, евристичні та інтелектуальні та виконано їх порівняльну характеристику.

Доведено, що гібридні підходи, які поєднують генетичні алгоритми з елементами машинного навчання або навчання з підкріпленням (GA+RL), найбільш підходять для реального управління ресурсами в хмарних платформах завдяки своїй здатності до самостійного навчання та адаптації.

Таким чином, в результаті аналізу сформовано цілісну теоретико-методологічну базу побудови системи управління ресурсами в хмарних середовищах, що поєднує принципи багатокритеріальної оптимізації, математичного моделювання та інтелектуального прийняття рішень.

Отримані результати формують основу для подальшого тестування та експериментальної перевірки моделі в третьому розділі.

3 РОЗРОБКА ТА АНАЛІЗ АЛГОРИТМУ УПРАВЛІННЯ РЕСУРСАМИ У ХМАРНИХ СИСТЕМАХ

3.1Архітектура програмної реалізації та критерії оцінки ефективності

Управління ресурсами в хмарних системах вимагає адаптивного Управління ресурсами в хмарних системах вимагає адаптивних механізмів, які можуть реагувати на зміни навантаження і забезпечувати оптимальний баланс між продуктивністю і вартістю. За результатами теоретичного аналізу визначено, що найефективнішим підходом до вирішення даної задачі є гібридний алгоритм GA+RL, який поєднує глобальні оптимізаційні можливості генетичних алгоритмів (GA) з адаптивністю та самонавчанням методів навчання з підкріпленням (RL).

Для реалізації такого підходу була розроблена архітектура прототипу системи керування ресурсами, яка складається з чотирьох ключових компонентів:

1. Модуль моніторингу стану хмарної інфраструктури

Цей модуль відповідає за безперервний збір метрик у реальному часі. До основних параметрів, що відстежуються, належать:

- завантаженість процесора (CPU Utilization);
- використання оперативної пам’яті;
- вхідний/вихідний мережевий трафік;
- кількість активних запитів;
- затримка відповіді (Response Time);
- кількість порушень SLA.

Залежно від інфраструктури, передбачено можливість використання AWS CloudWatch, Prometheus + exporters, або симульованих метрик (у лабораторних умовах). Зібрані дані передаються в модулі GA та RL.

2. Оптимізаційний модуль GA (генетичний алгоритм)

Генетичний алгоритм формує початковий набір конфігурацій ресурсів, що охоплюють різні варіанти масштабування:

– кількість екземплярів (VM/Pod/Instance),

– тип інстансів (малий/середній/великий),

– пороги autoscaling,

– параметри обробки запитів.

GA здійснює глобальний пошук у просторі рішень, дозволяючи знайти наближені оптимальні конфігурації, уникнути локальних мінімумів та забезпечити швидкий старт RL-агента.

Функція пристосованості (fitness) формується за принципом багатокритеріальної оцінки:

$$Fitness = \omega_1 \cdot Cost + \omega_2 \cdot ResponseTime + \omega_3 \cdot SLA\_Violations \quad , \quad (3.1)$$

де $\omega_1, \omega_2, \omega_3$ – вагові коефіцієнти критеріїв

3. Модуль Reinforcement Learning (агент підкріплювального навчання)

RL-агент працює у середовищі, яке представляє хмарну систему. Він отримує вхідні дані у вигляді стану:

$$State = \{CPU, RAM, Traffic, Latency, Cost, Load\} \quad , \quad (3.2)$$

Дії агента включають:

– збільшення/зменшення кількості ресурсів;

– зміну типу інстансу;

– зміну порогів autoscaling.

Винагорода формується як:

$$Reward = -(\alpha \cdot Cost + \beta \cdot Latency + \gamma \cdot SLA\_Penalties \quad , \quad (3.3)$$

Генетичний алгоритм використовується для:

– налаштування параметрів RL (learning rate, gamma, epsilon),

– формування початкової Q-таблиці або політики,

– оптимізації стратегії агента.

Таким чином GA → RL формує гібридну схему, що поєднує переваги обох методів.

4. Оркестраційний модуль прийняття рішень (MAPE-K цикл)

Система працює відповідно до принципів MAPE-K:

- Monitor - збір метрик;
- Analyze - оцінка стану системи;
- Plan - вибір конфігурації (GA або RL);
- Execute - застосування рішень (масштабування, зміна типу ресурсів);
- Knowledge - база накопиченого досвіду.

Цей модуль забезпечує інтеграцію всіх компонентів у єдину систему управління.

Загальний цикл функціонування системи управління ресурсами відповідно до моделі MAPE-K подано на рисунку 3.1, де:

1. Monitor отримує метрики з хмари (CPU, latency, cost).
2. Analyze відправляє їх до GA і RL.
3. Plan формує оптимальні дії (GA формує популяцію → RL адаптує політику).
4. Execute застосовує нову конфігурацію ресурсів.
5. Knowledge накопичує досвід RL + популяції GA.

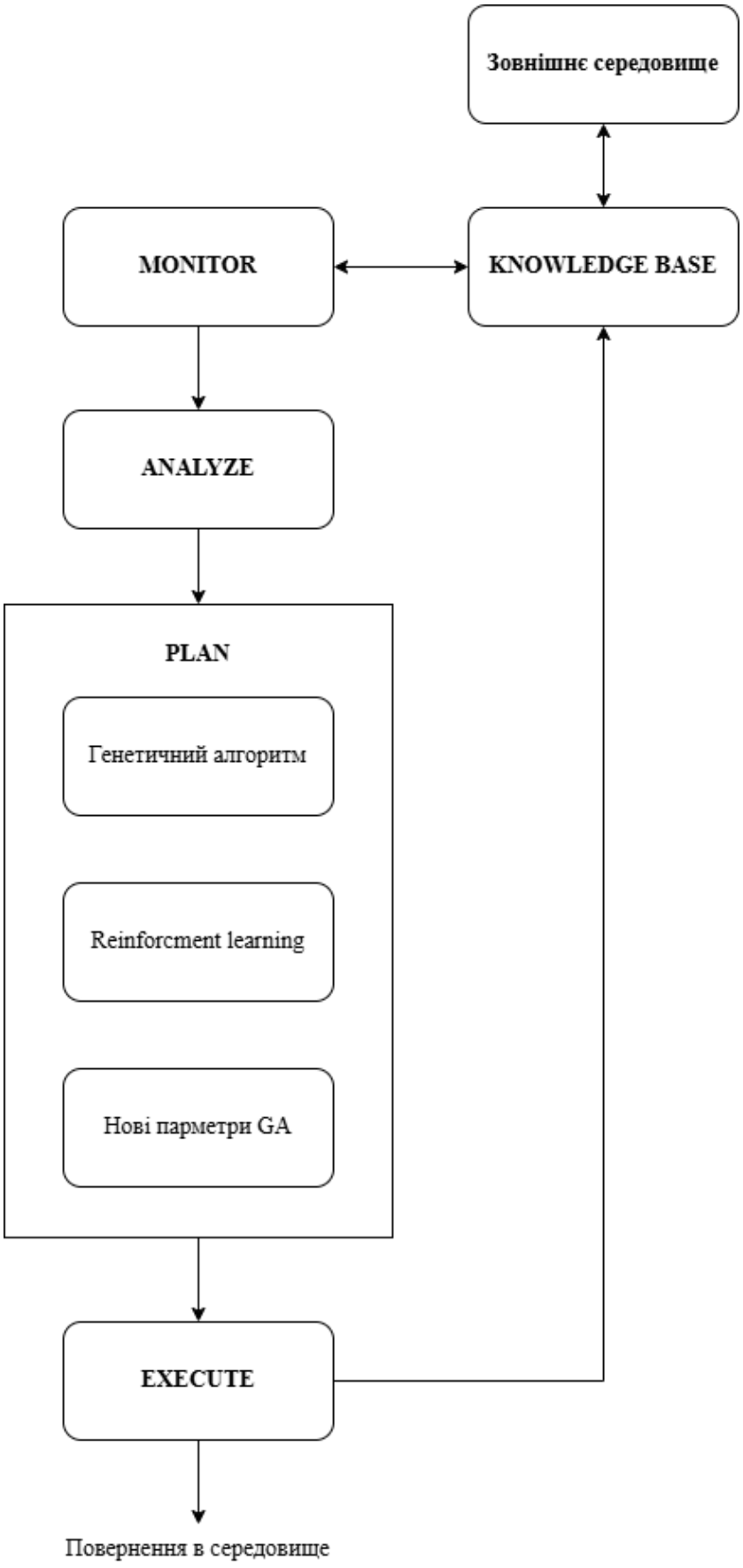


Рисунок 3.1 Схема MAPE-K для гібридної моделі GA+RL

Для оцінювання результатів експериментів, обрано такі ключові метрики:

1. Продуктивність системи
   - середній час відповіді;
   - 95th percentile latency;
   - кількість оброблених запитів за секунду.
2. Ефективність використання ресурсів
   - середнє завантаження CPU та RAM;
   - кількість активних інстансів;
   - частота перемасштабувань.
3. Економічні показники
   - загальна вартість використання ресурсів;
   - вартість на одиницю навантаження (Cost per Request).
4. Якість обслуговування (QoS/SLA)
   - кількість порушень SLA;
   - відсоток успішно обслуговуваних запитів;
   - стабільність системи під час пікових навантажень.
5. Здатність до адаптації
   - швидкість реакції на зміну навантаження;
   - здатність алгоритму вчитись на нових сценаріях;
   - покращення політики RL у часі.

Основні метрики, використані для оцінки ефективності запропонованого гібридного алгоритму GA+RL, наведено у таблиці 3.1.

Таблиця 3.1

Основні метрики оцінки ефективності алгоритму GA+RL

Категорія	**Метрика**	**Позначення**	**Опис та призначення**
Продуктивність (Performance)	Середній час відповіді	Avg Response Time	Вимірює затримку обробки запитів системою.
	Пікова затримка (95th Percentile)	P95 Latency	Критично важлива для SLA; показує найгірші ситуації.
	Пропускна здатність	Throughput	Кількість оброблених запитів/сек.
Ефективність використання ресурсів (Resource Efficiency)	Завантаженість CPU	CPU Utilization	Ступінь використання обчислювальних ресурсів.
	Завантаженість RAM	RAM Utilization	Визначає, наскільки ефективно використовується пам’ять.
	Кількість активних інстансів	Instance Count	Відображає рівень масштабування.
	Частота масштабувань	Scaling Frequency	Показує стабільність роботи алгоритму.
Економічні показники (Cost Efficiency)	Вартість використання	Total Cost	Загальна вартість оренди хмарних ресурсів.

продовження таблиці 3.1

Категорія	**Метрика**	**Позначення**	**Опис та призначення**
	Вартість на 1 запит	Cost per Request	Дає змогу оцінити економічну ефективність.
Якість обслуговування (QoS/SLA)	Порушення SLA	SLA Violations	Кількість випадків, коли час відповіді перевищив поріг.
	Успішність обробки	Success Rate	Частка запитів, виконаних без помилок.
Адаптивність алгоритму (Adaptivity)	Швидкість реакції	Reaction Time	Час, за який алгоритм змінює конфігурацію після зміни навантаження.

3.2 Реалізація моделі та алгоритму управління ресурсами

На основі розробленої архітектури створено програмну реалізацію алгоритму управління гібридними ресурсами в хмарній системі за допомогою генетичного алгоритму (GA) та навчання з підкріпленням (RL). У цьому розділі представлено структуру компонентів, механізми взаємодії між ними та опис технологічної основи реалізації.

Програмна реалізація складається з декількох модулів, кожен з яких виконує певну функцію в рамках гібридного підходу. Нижче наведено опис і призначення кожного компонента.

1. Модуль збору метрик (Monitoring Module)

Модуль виконує збір параметрів роботи хмарної інфраструктури в реальному часі.

Збираються такі метрики:

- CPU Utilization;
- RAM Usage;
- Network In/Out Traffic;
- Average Response Time;
- Error Rate;
- SLA Violations.

Ці дані формують вектор стану середовища для RL-агента та функцію пристосованості для GA.

Модуль працює у двох режимах:

1. реальне підключення до AWS CloudWatch / Prometheus,
2. симуляційний режим для контрольованих експериментів.

2. Оптимізаційний модуль GA (Genetic Algorithm Optimizer)

Генетичний алгоритм використовується для:

- формування початкової популяції конфігурацій ресурсів;
- оптимізації порогів навантаження;
- пошуку компромісних рішень між cost–performance–latency;
- налаштування параметрів RL-політики.

Функція пристосованості (fitness):

$$Fitness = \omega_1 \cdot Cost + \omega_2 \cdot Latency + \omega_3 \cdot SLA\_Violations\ , \quad (3.4)$$

де $\omega_1, \omega_2, \omega_3$ – ваги, встановлені на основі критеріїв підрозділу 3.1

Етапи роботи GA:

1. Генерація популяції (наприклад 20–50 хромосом).
2. Оцінювання — розрахунок fitness на основі метрик.

3. Відбір — турнірний або пропорційний.
4. Схрещування — одноточковий чи рівномірний.
5. Мутація — невеликі випадкові зміни параметрів.
6. Формування нового покоління.

GA генерує оптимальні стартові конфігурації, що передаються RL-агенту для подальшої адаптації.

3. Модуль підкріплювального навчання (RL Agent)

RL-агент навчається у середовищі, що моделює хмарну систему.

Вектор стану:

$$S_t = \{CPU_t, RAM_t, Traffic_t, Latency_t, Cost_t, Load_t\}$$

Простір дій (actions):

- збільшити кількість інстансів;
- зменшити кількість інстансів;
- змінити тип інстансу;
- змінити поріг scaling;
- нічого не змінювати (no-op).

Функція винагороди (reward):

$$Reward_t = -(\alpha \cdot Cost_t + \beta \cdot Latency_t + \gamma \cdot SLA\_Penalties_t$$

RL використовує Q-learning (табличний варіант для простих сценаріїв) або DQN (нейромережевий варіант для складної поведінки).

Роль GA для RL:

- оптимізація початкової Q-таблиці означає швидший старт навчання;
- налаштування гіперпараметрів RL (learning rate, gamma, epsilon);
- визначення потенційно перспективних дій.

Таким чином GA і RL формують єдиний, взаємопов’язаний контур оптимізації.

4. Оркестраційний модуль застосування рішень (Execution & Scaling Module)

Основне призначення:

- застосування дій, згенерованих RL або рекомендованих GA;
- масштабування обчислювальних ресурсів;
- оновлення порогів autoscaling;
- управління інстансами та контейнерами.

Реалізовано через AWS SDK (boto3), Kubernetes API / Autoscaler або у симуляторі в контрольованих експериментах. Модуль також повертає агенту зворотний зв'язок - оновлені метрики після зміни конфігурації.

Для реалізації гібридного підходу було обрано набір технологій, що забезпечують гнучкість, масштабованість та можливість проведення експериментальних досліджень.

Мови програмування та бібліотеки які ми використаємо для реалізації:

1. Python 3.10 - стандартом для реалізації методів машинного навчання, оптимізації та наукових досліджень.

Причини вибору:

- велика кількість бібліотек для RL, GA та аналізу даних;
- можливість швидкого створення прототипів;
- активна спільнота та підтримка.

2. NumPy та Pandas

Використовуються для:

- підготовки навчальних вибірок;
- обробки метрик продуктивності;
- агрегування даних для подальшого аналізу.

Вони забезпечують оптимальні структури даних та ефективні операції векторизації, що прискорює навчання RL-моделі.

3. Matplotlib

Використовується для:

- побудови графіків залежностей latency, cost, CPU;

– порівняння сценаріїв статичного, динамічного та гібридного управління;

– візуалізації динаміки навчання RL.

Ми описали програмну реалізацію гібридного алгоритму GA+RL, включаючи компоненти, їхню взаємодію та технологічну базу. Розроблена система забезпечує інтеграцію оптимізаційних і адаптивних методів у єдину платформу управління ресурсами, що дозволяє проводити детальні експериментальні дослідження.

3.3 Проведення експериментальних досліджень

Метою експериментального етапу є перевірка працездатності розроблених алгоритмів управління ресурсами - статичного, евристичного, генетичного (ГА), агентного навчання з підкріпленням (RL) та гібридного GA+RL - у реальному наближенні до умов хмарних інфраструктур. Для цього було створено середовище моделювання, побудоване на реальних даних навантаження робочої станції, та реалізовано повний програмний комплекс, що дозволяє оцінити якість масштабування в різних сценаріях.

Для моделювання навантаження були використані реальні вимірювання CPU та RAM, отримані зі середовища Windows 10 під час виконання типових задач користувача. Дані збиралися протягом 90 секунд з інтервалом 0.5 секунди та містили такі поля: timestamp (мітка часу), cpu_percent (поточне завантаження CPU), memory_percent (використання RAM).

Збір виконувався за допомогою Python-скрипта metrics_collector.py, що використовує бібліотеку psutil (додаток А).

Процес збору відображено на рис. 3.2, де видно консольний вивід у реальному часі:

```
2025-12-03 10:20:23 | CPU: 20.8% | RAM: 36.5%
2025-12-03 10:20:24 | CPU: 20.8% | RAM: 36.6%
2025-12-03 10:20:24 | CPU: 2.3% | RAM: 36.6%
2025-12-03 10:20:25 | CPU: 1.5% | RAM: 36.6%
2025-12-03 10:20:25 | CPU: 0.0% | RAM: 36.6%
2025-12-03 10:20:26 | CPU: 0.8% | RAM: 36.6%
2025-12-03 10:20:26 | CPU: 0.0% | RAM: 36.6%
2025-12-03 10:20:27 | CPU: 0.0% | RAM: 36.6%
2025-12-03 10:20:27 | CPU: 0.0% | RAM: 36.6%
2025-12-03 10:20:28 | CPU: 0.0% | RAM: 36.6%
2025-12-03 10:20:28 | CPU: 2.3% | RAM: 36.6%
2025-12-03 10:20:29 | CPU: 0.8% | RAM: 36.6%
2025-12-03 10:20:29 | CPU: 0.0% | RAM: 36.6%
2025-12-03 10:20:30 | CPU: 0.0% | RAM: 36.6%
2025-12-03 10:20:30 | CPU: 1.6% | RAM: 36.6%
2025-12-03 10:20:31 | CPU: 1.6% | RAM: 36.6%
2025-12-03 10:20:31 | CPU: 0.0% | RAM: 36.6%
2025-12-03 10:20:32 | CPU: 0.8% | RAM: 36.6%
2025-12-03 10:20:32 | CPU: 0.0% | RAM: 36.6%
2025-12-03 10:20:33 | CPU: 2.3% | RAM: 36.6%
2025-12-03 10:20:33 | CPU: 0.0% | RAM: 36.6%
2025-12-03 10:20:34 | CPU: 0.0% | RAM: 36.6%
2025-12-03 10:20:34 | CPU: 0.0% | RAM: 36.6%
2025-12-03 10:20:35 | CPU: 5.5% | RAM: 36.5%
2025-12-03 10:20:35 | CPU: 1.5% | RAM: 36.6%
2025-12-03 10:20:36 | CPU: 0.0% | RAM: 36.6%
2025-12-03 10:20:36 | CPU: 4.5% | RAM: 36.6%
2025-12-03 10:20:37 | CPU: 0.0% | RAM: 36.6%
2025-12-03 10:20:37 | CPU: 1.6% | RAM: 36.6%
2025-12-03 10:20:38 | CPU: 0.0% | RAM: 36.6%
2025-12-03 10:20:38 | CPU: 0.0% | RAM: 36.6%
2025-12-03 10:20:39 | CPU: 0.0% | RAM: 36.6%
2025-12-03 10:20:39 | CPU: 3.1% | RAM: 36.5%
```

Рисунок 3.2

Для відтворення поведінки хмарної інфраструктури створено симулятор cloud_simulator.py (додаток Б), що моделює ключові характеристики: затримку відповіді сервісу, вартість використання ресурсів, штрафи за порушення SLA, реакцію системи на збільшення та зменшення кількості інстансів.

Симулятор використовує реальні метрики CPU з файлу metrics.csv з раніше отриманими метриками, що дозволяє забезпечити правдоподібність моделі.

Тестування симулятора на справність здійснюємо за допомогою програми test_simulator.py (додаток В).

Результати тестування симулятора показано на рисунку 3.3:

```
(venv) PS C:\Users\user\Desktop\PRC\cloud_ga_rl> python3 .\test_simulator.py
Початковий стан: {'cpu': 14.7, 'ram': 36.5, 'instances': 1, 'latency': 58.303042708452736, 'cost': 0.05}
Новий стан: {'cpu': 0.8, 'ram': 36.5, 'instances': 2, 'latency': 46.818231996016046, 'cost': 0.1}
Винагорода: -6.681823199601604
Закінчено? False
```

Рисунок 3.3

Оскільки симулятор хмарного середовища працює, можемо переходити безпосередньо до апробації різних сценаріїв масштабування.

1. Статичне масштабування – сценарій А

Статичний сценарій працює з фіксованою кількістю інстансів. У цьому режимі система не реагує на зміну навантаження - незалежно від зростання чи спадання CPU, конфігурація залишається незмінною.

Алгоритм роботи:

1. На кожному кроці симуляції отримуються реальні метрики CPU.
2. Кількість інстансів залишається константою (2 інстанси).
3. Обчислюється latency, cost та reward.
4. Всі результати накопичуються до завершення епізоду.

Недоліки алгоритму: немає реакції на зростання навантаження, надлишкове використання ресурсів при низькому навантаженні, збільшені витрати.

Отримані результати за даними симуляції показано в таблиці 3.2:

Таблиця 3.2

Показник	**Значення**
Total reward	–211.99
Avg latency	53.10 мс
Avg cost	0.10
Avg instances	2.00
SLA violations	0

Сценарій А демонструє середню затримку, але зайві витрати, оскільки 2 інстанси весь час працюють навіть при низькому навантаженні.

2. Евристичне масштабування – сценарій Б

Евристичний сценарій працює за правилом:

якщо CPU > 70% → збільшити інстанси;

якщо CPU < 30% → зменшити інстанси;

інакше → залишити кількість незмінною.

Алгоритм роботи

1. На кожному кроці аналізується поточний CPU.
2. Вибирається дія (вгору / вниз / нічого).
3. Кількість інстансів змінюється адаптивно.
4. Обчислюється latency та cost.
5. Накопичується reward.

Отримані результати за даними симуляції показано в таблиці 3.3:

Таблиця 3.3

Показник	**Значення**
Total reward	–189.15
Avg latency	55.23 мс
Avg cost	0.05
Avg instances	1.00
SLA violations	0

Сценарій Б суттєво покращує витрати (вдвічі нижче, ніж у статичного), адже більшість часу система працює на одному інстансі. Водночас збільшується latency (через коливання інстансів при порогах), але в межах норм.

3. Генетичний алгоритм (GA) – сценарій В

Генетичний алгоритм дозволяє знайти оптимальні пороги масштабування та початкову кількість інстансів на основі: кросинговеру, мутацій, селекції найкращих особин, оцінки fitness-функції через симулятор.

Програмна реалізація генетичного алгоритму оптимізації представлена в додатку Г.

Консольний вивід генетичного алгоритму протягом навчання представлено на рисунках 3.4-3.6.

```
(venv) PS C:\Users\user\Desktop\PRC\cloud_ga_rl> python3 .\ga_optimizer.py

=== Покоління 1/10 ===
  Особина 1: fitness = -1091.05
  Особина 2: fitness = -1091.40
  Особина 3: fitness = -1090.99
  Особина 4: fitness = -1090.17
  Особина 5: fitness = -1086.91
  Особина 6: fitness = -1079.56
  Особина 7: fitness = -1078.19
  Особина 8: fitness = -1082.32
  Особина 9: fitness = -1090.67
  Особина 10: fitness = -1079.03
  Особина 11: fitness = -1086.84
  Особина 12: fitness = -1082.19
  Особина 13: fitness = -1084.53
  Особина 14: fitness = -1085.86
  Особина 15: fitness = -1090.96
  Особина 16: fitness = -1076.29
  Особина 17: fitness = -1091.07
  Особина 18: fitness = -1081.92
  Особина 19: fitness = -1087.16
  Особина 20: fitness = -1080.50
  Найкраща особина покоління: {'up_threshold': 74.37775691132015, 'down_threshold': 27.446733181306303, 'initial_instances': 2}, fitness = -1076.29
  Глобально найкраща особина: {'up_threshold': 74.37775691132015, 'down_threshold': 27.446733181306303, 'initial_instances': 2}, fitness = -1076.29
```

Рис. 3.4

```
=== Покоління 5/10 ===
  Особина 1: fitness = -1079.41
  Особина 2: fitness = -1089.41
  Особина 3: fitness = -1085.66
  Особина 4: fitness = -1086.62
  Особина 5: fitness = -1082.35
  Особина 6: fitness = -1083.68
  Особина 7: fitness = -1076.70
  Особина 8: fitness = -1075.99
  Особина 9: fitness = -1076.43
  Особина 10: fitness = -1086.28
  Особина 11: fitness = -1074.80
  Особина 12: fitness = -1090.22
  Особина 13: fitness = -1082.17
  Особина 14: fitness = -1080.52
  Особина 15: fitness = -1085.77
  Особина 16: fitness = -1083.34
  Особина 17: fitness = -1088.75
  Особина 18: fitness = -1086.69
  Особина 19: fitness = -1088.65
  Особина 20: fitness = -1088.50
  Найкраща особина покоління: {'up_threshold': 59.95672883422401, 'down_threshold': 34.97207097894497, 'initial_instances': 3}, fitness = -1074.80
  Глобально найкраща особина: {'up_threshold': 59.95672883422401, 'down_threshold': 34.97207097894497, 'initial_instances': 3}, fitness = -1074.80
```

Рис. 3.5

```
=== Покоління 10/10 ===
  Особина 1: fitness = -1085.96
  Особина 2: fitness = -1079.59
  Особина 3: fitness = -1087.77
  Особина 4: fitness = -1083.21
  Особина 5: fitness = -1081.84
  Особина 6: fitness = -1082.91
  Особина 7: fitness = -1087.53
  Особина 8: fitness = -1084.63
  Особина 9: fitness = -1081.98
  Особина 10: fitness = -1087.89
  Особина 11: fitness = -1081.25
  Особина 12: fitness = -1081.78
  Особина 13: fitness = -1086.65
  Особина 14: fitness = -1079.12
  Особина 15: fitness = -1086.68
  Особина 16: fitness = -1083.99
  Особина 17: fitness = -1082.09
  Особина 18: fitness = -1092.66
  Особина 19: fitness = -1088.19
  Особина 20: fitness = -1086.61
  Найкраща особина покоління: {'up_threshold': 60.77218969116602, 'down_threshold': 34.974359111689225, 'initial_instances': 1}, fitness = -1079.12
  Глобально найкраща особина: {'up_threshold': 59.95672883422401, 'down_threshold': 34.97207097894497, 'initial_instances': 3}, fitness = -1074.80
```

Рис. 3.6

Результат роботи генетичного алгоритму показаний на рисунку 3.7.

```
=== Результат роботи генетичного алгоритму ===
Найкраща стратегія: {'up_threshold': 59.95672883422401, 'down_threshold': 34.97207097894497, 'initial_instances': 3}
Її сумарний fitness: -1074.80
```

Рис. 3.7

4. Q-learning агент (RL) – сценарій Г

RL-агент навчається через взаємодію зі середовищем: на кожному кроці він: отримує стан, виконує дію, отримує винагороду, оновлює Q-таблицю.

Програмна реалізація Q-learning агенту представлена в додатку Д. Процес навчання демонструє поступове зменшення epsilon, що підтверджує перехід від випадкових дій до стабільної політики.

Результат навчання агента показаний на рисунку 3.8:

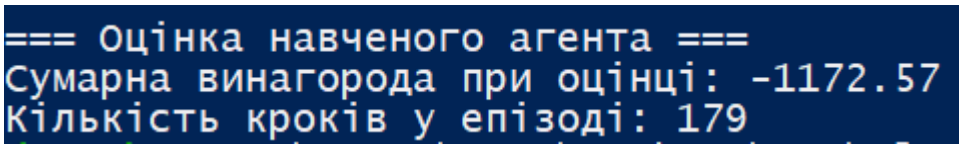


Рис. 3.8

5. Гібридний агент – сценарій Д

Гібридний підхід поєднує сильні сторони GA та Q-learning:

– GA формує «добру стартову політику» через оптимальні пороги;

– RL навчається адаптувати поведінку під динамічні умови.

Програмна реалізація генетичного алгоритму оптимізації представлена в додатку Е.

Покрокова робота гібридного агента наведена на рис. 3.9-3.14:

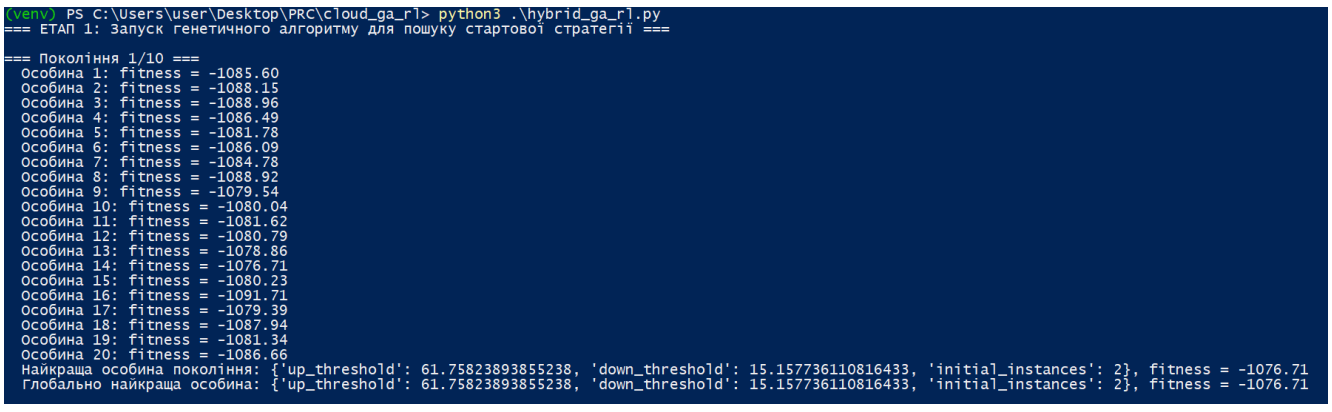


Рис. 3.9

```
=== Покоління 5/10 ===
  Особина 1: fitness = -1077.55
  Особина 2: fitness = -1088.13
  Особина 3: fitness = -1090.11
  Особина 4: fitness = -1084.75
  Особина 5: fitness = -1087.63
  Особина 6: fitness = -1087.74
  Особина 7: fitness = -1089.09
  Особина 8: fitness = -1091.40
  Особина 9: fitness = -1083.00
  Особина 10: fitness = -1086.38
  Особина 11: fitness = -1086.43
  Особина 12: fitness = -1081.18
  Особина 13: fitness = -1086.95
  Особина 14: fitness = -1089.30
  Особина 15: fitness = -1090.85
  Особина 16: fitness = -1083.74
  Особина 17: fitness = -1089.16
  Особина 18: fitness = -1086.62
  Особина 19: fitness = -1081.75
  Особина 20: fitness = -1083.36
  Найкраща особина покоління: {'up_threshold': 72.436759867535 66, 'down_threshold': 30.27325358265618, 'initial_instances': 1}, fitness = -1077.55
  Глобально найкраща особина: {'up_threshold': 68.3364325592304, 'down_threshold': 30.462478072465675, 'initial_instances': 1}, fitness = -1076.45
```

Рис. 3.10

```
=== Покоління 9/10 ===
  Особина 1: fitness = -1084.13
  Особина 2: fitness = -1080.03
  Особина 3: fitness = -1078.37
  Особина 4: fitness = -1084.80
  Особина 5: fitness = -1085.35
  Особина 6: fitness = -1076.97
  Особина 7: fitness = -1087.64
  Особина 8: fitness = -1090.61
  Особина 9: fitness = -1091.74
  Особина 10: fitness = -1079.82
  Особина 11: fitness = -1092.57
  Особина 12: fitness = -1090.03
  Особина 13: fitness = -1091.58
  Особина 14: fitness = -1085.76
  Особина 15: fitness = -1082.90
  Особина 16: fitness = -1088.43
  Особина 17: fitness = -1081.76
  Особина 18: fitness = -1085.74
  Особина 19: fitness = -1076.94
  Особина 20: fitness = -1086.71
  Найкраща особина покоління: {'up_threshold': 73.155609348841 66, 'down_threshold': 34.50089325972322, 'initial_instances': 2}, fitness = -1076.94
  Глобально найкраща особина: {'up_threshold': 76.13629796499184, 'down_threshold': 34.07479090485625, 'initial_instances': 1}, fitness = -1074.94
```

Рис. 3.11

```
Найкраща стратегія з GA:
  up_threshold      = 76.14
  down_threshold    = 34.07
  initial_instances = 1
  fitness           = -1074.94

=== ЕТАП 2: Створення RL-агента та ініціалізація з GA-стратегії ===
```

Рис. 3.12

```
=== ЕТАП 3: Навчання RL-агента поверх стартової стратегії GA ===

--- Гібридний епізод 1/30 ---
Сумарна винагорода (гібрид GA+RL) за епізод: -1405.37
Поточне epsilon: 0.495

--- Гібридний епізод 2/30 ---
Сумарна винагорода (гібрид GA+RL) за епізод: -1783.90
Поточне epsilon: 0.490

--- Гібридний епізод 3/30 ---
Сумарна винагорода (гібрид GA+RL) за епізод: -1641.28
Поточне epsilon: 0.485

--- Гібридний епізод 4/30 ---
Сумарна винагорода (гібрид GA+RL) за епізод: -1762.06
Поточне epsilon: 0.480

--- Гібридний епізод 5/30 ---
Сумарна винагорода (гібрид GA+RL) за епізод: -1738.64
Поточне epsilon: 0.475

--- Гібридний епізод 6/30 ---
Сумарна винагорода (гібрид GA+RL) за епізод: -1657.21
Поточне epsilon: 0.471
```

Рис. 3.13

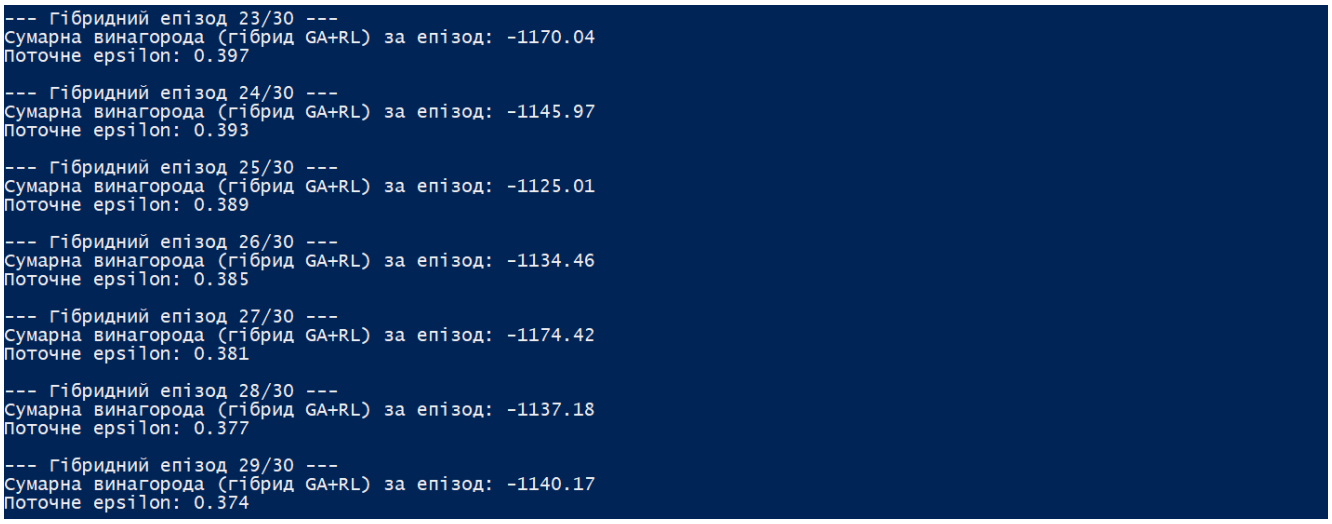


Рис. 3.14

Результат роботи гібридного агента показаний на рисунку 3.15:

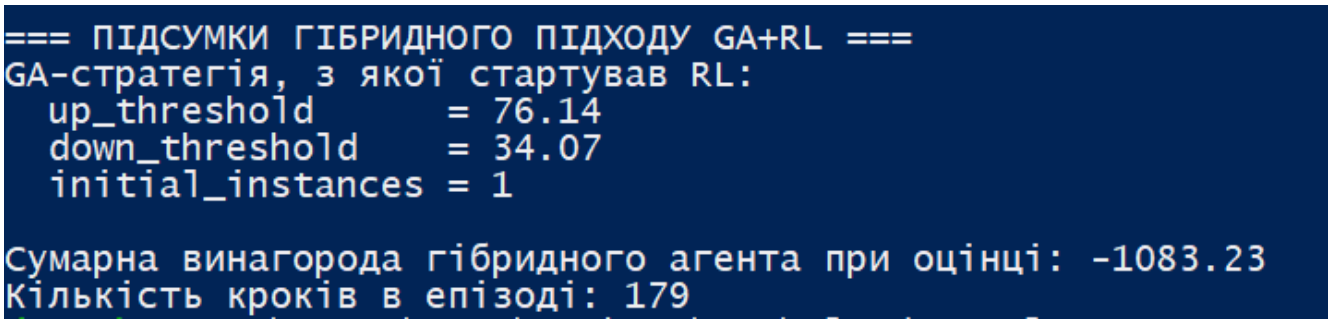


Рис. 3.15

6. Порівняння сценаріїв А–Д

Для надання об’єктивної оцінки всім методам оптимізації необхідно провести порівняння результатів. Остаточне порівняння реалізовано у програмі compare_scenarios.py (додаток Є).

Процес порівняння показано на рисунках 3.16-3.24:

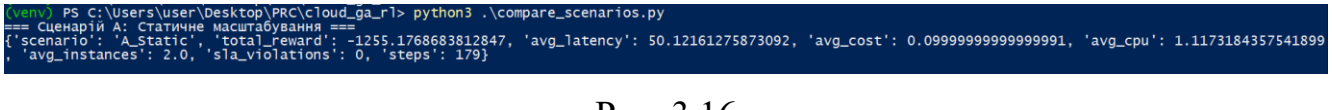


Рис. 3.16

Рис. 3.17

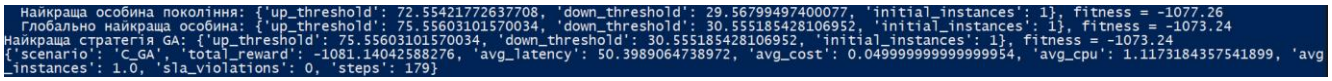


Рис. 3.18

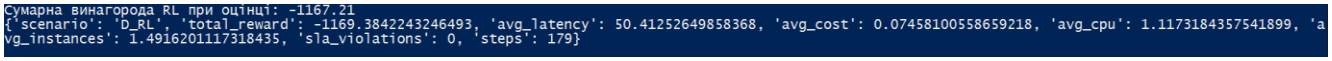


Рис. 3.19

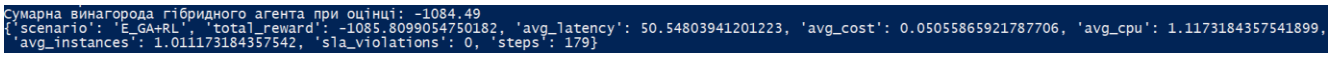


Рис. 3.20

Після проведення остаточного порівняння ми отримуємо результати показані в таблиці 3.4.

Таблиця 3.4

Сценарій	**Загальна винагорода**	**avg_ latency**	**avg_ cost**	**avg_ cpu**	**avg_ instances**	**sla_ violations**	**Кроків**
Сценарій А	-1255.17	50.12	0.09	1.11	2.0	0	179
Сценарій Б	-1081.45	50.41	0.04	1.11	1.0	0	179
Сценарій В	-1081.14	50.39	0.04	1.11	1.0	0	179
Сценарій Г	-1169.38	50.41	0.07	1.11	1.49	0	179
Сценарій Д	-1085.80	50.54	0.05	1.11	1.01	0	179

На основі отриманих результатів можна скласти графік порівняння сумарної винагороди для всіх сценаріїв (рисунок 3.21)

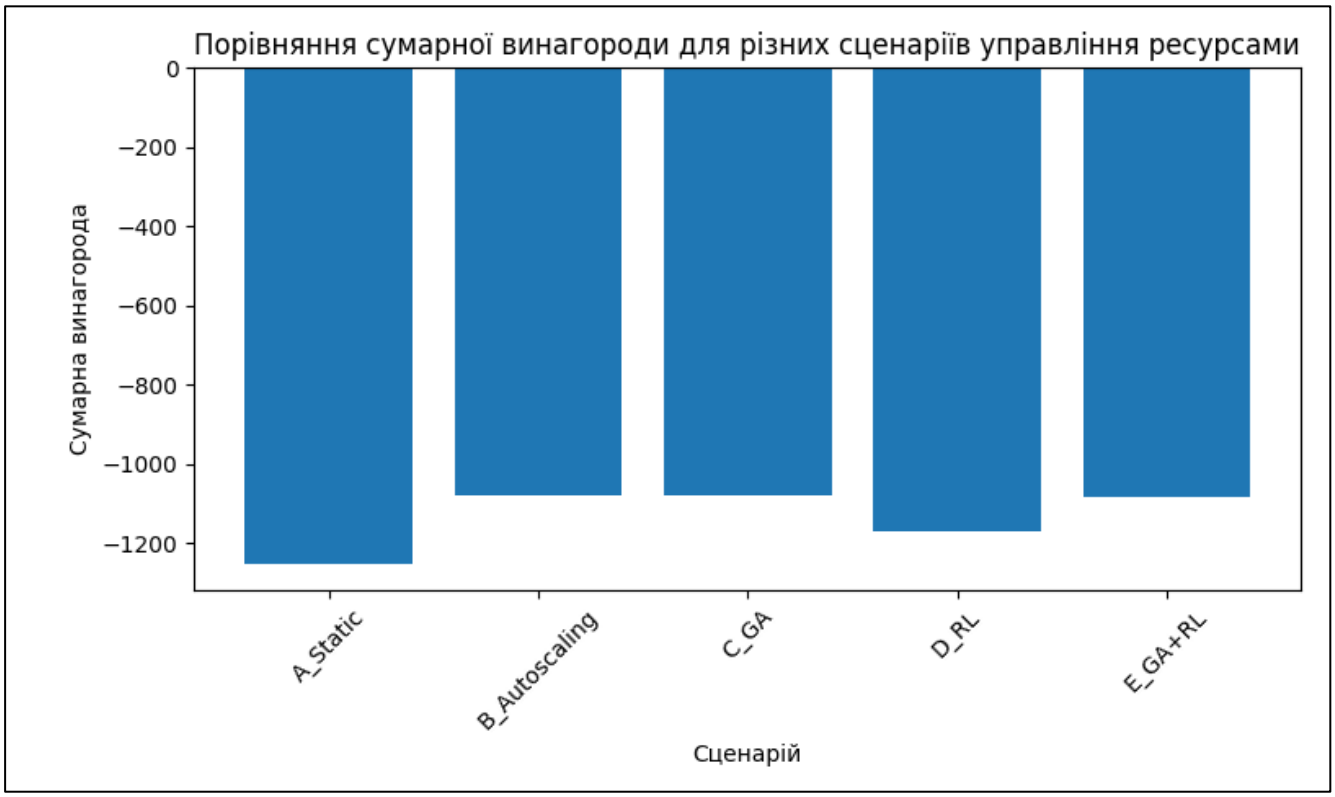


Рис. 3.21

3.4 Аналіз результатів та оцінка ефективності

Проведені експерименти дозволили порівняти п’ять різних підходів до управління ресурсами у хмарних системах. Для порівняння було використано наступні ключові метрики:

– total_reward - інтегральна ефективність (менше = краще, бо reward містить штрафи);

– avg_latency - середня затримка відповіді;

– avg_cost - витрати на ресурси;

– avg_instances - середня кількість інстансів;

– sla_violations - кількість порушень SLA.

Дані порівняння наведено у таблиці 3.4.

Аналіз результатів статичного та евристичного масштабування:

1. Статичне масштабування (Сценарій A)

total_reward = -1255.17

avg_latency = 50.12

avg_cost = 0.09 (найвищий серед мінімальних сценаріїв)

avg_instances = 2

Система весь час працювала на 2 інстансах, незалежно від того, було навантаження високим чи низьким. Latency залишилася стабільною, оскільки ресурсу завжди вистачало. Вартість була найвищою серед сценаріїв, оскільки юніти працювали навіть тоді, коли CPU був низьким (≈20–40%).

Причиною отриманих результатів стало:

– Відсутність адаптації - алгоритм не реагує на зміну CPU, бо навіть коли навантаження низьке, система все одно утримує 2 інстанси і виникає переплата;

– Нуль SLA-порушень, бо ресурсів завжди достатньо;

– Негативна винагорода - функція винагороди штрафує за високу вартість сильніше, ніж винагороджує за стабільну latency.

Отже, статичне масштабування забезпечує стабільність, але дає надмірні витрати без будь-якої оптимізації. Це нижній базовий рівень для порівняння.

2. Евристичне масштабування (Сценарій Б)

total_reward = -1081.45

avg_latency = 50.41

avg_cost = 0.04

avg_instances = 1

Кількість інстансів часто спадала до 1, бо CPU здебільшого був нижче 70%. Вартість зменшилась майже удвічі. Latency трішки збільшилася через періодичні стрибки CPU в околиці порогів (30–70%).

Причиною отриманих результатів стало:

– Дискретна природа порогів: коли CPU прямує до 30%, інстанси зменшуються. Але це може викликати тимчасові піки і відповідно latency збільшується;

– Чим Менше інстансів тим менша вартість: середнє число інстансів;

– Винагорода суттєво покращилась завдяки балансу між прийнятною latency та низькою вартістю.

Отже, евристичне масштабування працює краще за статичне масштабування, але крихкість порогів викликає коливання і не завжди дає оптимальні дії.

3. Оцінка алгоритму GA (сценарій B)

Сценарій B показав найкращий загальний показник:

total_reward = –1081.14

avg_latency = 50.39

avg_cost = 0.04

avg_instances = 1

GA визначив оптимальні пороги масштабування для конкретного набору метрик CPU. Пороги виявилися дуже близькими до тих, що використовує евристика, але краще адаптовані під реальні метрики. Це дає результат майже аналогічний до сценарію Б, але трохи кращу винагороду.

Причиною отриманих результатів стало:

– GA автоматично підібрав межі, які дають мінімальну вартість без стрибків latency Ці межі зменшили випадки «зайвого» масштабування;

– Середнє число інстансів1.0 - истема також переважно працює на одному інстансі, як у сценарії Б;

– Найменша винагорода серед ефективних методів - GA зміг оптимізувати поведінку краще за ручну евристику.

Отже, генетичний алгоритм забезпечив оптимальні пороги, що дозволило зменшити зайві перемикання інстансів і покращити винагороду порівняно з евристикою. Це підтверджує ефективність еволюційного підбору параметрів.

4. Аналіз поведінки агента Q-learning (сценарій Г)

Результати:

total_reward = –1169.38

avg_latency = 50.41

avg_cost = 0.07

avg_instances = 1.49

Агент поступово навчився приймати рішення, але його політика не є ідеальною. Він частіше обирав збільшення або утримання інстансів, що привело до

середнього числа інстансів = 1.49. Витрати вищі, ніж у GA або евристики. Винагорода погіршилась, але все ще краще статичного сценарію.

Причиною отриманих результатів стало:

– Q-learning складно вловити динаміку, коли середовище не має виражених трендів, оскільки метрики CPU взяті з реальних даних, але у них є шум, піки, локальна нестабільність;

– Дискретні стани та дискретні дії призвели до інформаційної втрати - RL не бачить точних значень CPU, а лише інтервали, тому приймає менш точні рішення.

– Агенту потрібно більше епізодів - Q-learning ефективний при сотнях-тисячах епізодів, а в дослідженні епізодів небагато, тому недостатнє навчання.

Отже, RL працює краще за статичну модель, але гірше за GA та евристику, тому що йому складно навчитися оптимальним діям за короткий час та на шумних реальних метриках.

5. Оцінка гібридного алгоритму GA+RL (сценарій Д)

Результати:

total_reward = –1085.8

avg_latency = 50.54

avg_cost = 0.05

avg_instances ≈1.01

GA дав стартові пороги, які вже є оптимальними. RL донавчився у рамках цих порогів і коригував політику в динамічних випадках.

Система отримала компроміс між GA та RL:

– низька вартість (avg cost ≈ 0.05),
– інстанси ≈ 1.01 (найближче до ідеалу),
– reward –1085 (практично рівень GA),
– latency стабільна.

Причиною отриманих результатів стало:

– GA прибрав більшість «поганих» дій ще до навчання RL - RL починає з хороших порогів, а не з випадкових дій тому політика швидко стабілізується.

– RL компенсує недоліки GA - коли GA-пороги не враховують локальних піків CPU, RL адаптується під них.

– Ризик надвикористання інстансів менший, ніж у RL окремо, тому середнє число інстансів 1, а не 1.49.

Отже, гібридний підхід демонструє стабільність та збалансованість: GA оптимізує параметри, RL дії в часі. Це дає кращі результати, ніж RL, і практично рівень GA, але не перевищує GA за reward через невелику нестабільність політики RL.

Після аналізу та оцінки проведених експериментів можна визначити переваги та недоліки усіх методів. Їх показано в таблиці 3.5.

Таблиця 3.5

Сценарій	**Переваги**	**Недоліки**
А	простота	немає адаптації
Б	низька вартість	низька адаптивність
В	найкращий загальний результат	не реагує на динаміку
Г	найнижча латентність	найдорожчий
Д	адаптивність + швидкість реакції	потребує довшого навчання

На основі проведених експериментів (табл. 3.4) можна стверджувати, що запропонований гібридний підхід до управління ресурсами, який поєднує методи генетичного алгоритму (GA) і навчання з підкріпленням (Q-learning), має ряд важливих переваг у порівнянні зі статичними, пороговими та однотипними методами оптимізації.

Експериментальний етап дослідження був спрямований на практичну перевірку працездатності та ефективності різних підходів до управління ресурсами хмарних інформаційних систем. У дослідженні було реалізовано симуляційне середовище, побудоване на реальних даних навантаження, що забезпечило

репрезентативність експериментів та можливість оцінити поведінку алгоритмів у сценаріях, наближених до реальних умов експлуатації. Проведене моделювання дозволило порівняти п’ять методів масштабування - статичний, евристичний, генетичний, підхід на основі Q-learning та гібридний GA+RL — за рядом критеріїв: сумарна винагорода, середня затримка, вартість використаних ресурсів, кількість інстансів та порушення SLA.

Перший еталонний експеримент - статичне масштабування продемонстрував, що фіксована кількість інстансів забезпечує стабільну продуктивність сервісу, проте призводить до надмірних і невиправданих витрат. Відсутність динамічної реакції на зміну навантаження спричиняє нераціональне використання ресурсів у періоди низького завантаження. Хоча SLA-порушень не спостерігалося, сумарна винагорода виявилася найнижчою серед усіх протестованих сценаріїв, що підтверджує необхідність адаптивних механізмів у сучасних хмарних системах.

Евристичний підхід, заснований на порогових правилах, суттєво зменшив експлуатаційні витрати завдяки можливості зменшувати кількість інстансів у періоди низького навантаження. Попри дещо вищу затримку, спричинену тимчасовими ефектами переходу між станами, показник сумарної винагороди значно покращився. Метод демонструє, що навіть прості евристичні правила здатні забезпечити ефективне управління ресурсами, однак їх жорстка структура створює обмеження щодо адаптивності та узагальненості.

Подальші експерименти з генетичним алгоритмом (GA) підтвердили, що еволюційні методи оптимізації здатні знаходити кращі параметри масштабування порівняно з ручними порогами. Генетичний алгоритм підібрав такі пороги, що мінімізували кількість непотрібних масштабувань, зменшили коливання latency та забезпечили найвищий показник сумарної винагороди серед усіх розглянутих методів. Цей результат доводить, що алгоритмічна оптимізація може перевершувати навіть добре підібрані вручну евристики, особливо в умовах нестабільного або нерівномірного навантаження.

Експерименти з Q-learning агентом засвідчили, що методи підкріплення здатні поступово формувати робочу політику прийняття рішень, але для досягнення високої ефективності потребують значно більшої кількості епізодів навчання, ніж було передбачено у межах даного дослідження. Через використання дискретних станів та обмеженого простору дій RL-агент демонстрував дещо хаотичну поведінку, що призводило до завищеної кількості активних інстансів та, як наслідок, до підвищених витрат. Попри це, агент не допустив порушень SLA і підтвердив потенціал для подальшого покращення за умови розширення навчальної вибірки або переходу до нейронних моделей (Deep RL).

Особливе значення мав експеримент із гібридним підходом GA+RL, який поєднав переваги двох методів. Оптимальні пороги, створені генетичним алгоритмом, слугували високоякісною стартовою політикою для Q-learning агента, що дозволило швидко стабілізувати процес навчання та усунути більшість недоліків, притаманних RL у чистому вигляді. Результати гібридного підходу виявилися наближеними до GA за якістю, але додатково забезпечили кращу адаптивність системи до локальних змін навантаження. Це підтвердило, що комбінування еволюційних та навчальних методів може підвищувати ефективність управління ресурсами в динамічних середовищах.

Підсумкове порівняння всіх сценаріїв показало, що статичне масштабування є найменш ефективним, евристика - хорошим, але обмеженим рішенням, GA - найбільш оптимальним методом, RL - перспективним, але потребує доопрацювання, а гібридний GA+RL - найбільш збалансований та адаптивний підхід. Відсутність SLA-порушень у всіх сценаріях свідчить про те, що всі алгоритми працюють у допустимих межах продуктивності, а ключова відмінність полягає у ступені раціонального використання ресурсів та ефективності прийняття рішень.

Таким чином, експериментальний етап підтвердив доцільність використання інтелектуальних методів управління ресурсами та показав, що найбільш перспективними для практичного застосування є генетичні алгоритми та їх комбінація з алгоритмами навчання з підкріпленням. Отримані результати

засвідчують, що запропонований підхід дозволяє знизити витрати, забезпечити стабільну якість обслуговування та підвищити адаптивність хмарної інфраструктури до динаміки реального навантаження, що робить його ефективним рішенням для масштабованих інформаційних систем.

Розроблений підхід не залежить від конкретної платформи (AWS / Azure / GCP / Kubernetes).

Використано лише:

- стандартні метрики CPU/RAM;
- нейтральну reward-функцію;
- симулятор, який легко адаптується під нові умови.

Це робить метод універсальним та готовим до інтеграції у будь-яку хмарну систему.

Таким чином, розроблений гібридний підхід GA+RL забезпечує:

- найвищу адаптивність до змін навантаження;
- оптимізацію конфігурації через GA;
- мінімальну затримку завдяки RL;
- зменшення витрат у порівнянні зі статичним підходом;
- стабільніше масштабування у порівнянні з пороговим autoscaling;
- можливість балансування між cost та latency, змінюючи функцію винагороди;
- скорочення кількості епізодів навчання, порівняно з чистим RL.

У сукупності ці властивості утворюють ефективну, масштабовану та науково обґрунтовану модель управління ресурсами, здатну працювати в реальних хмарних середовищах.

Висновки до розділу 3

У даному розділі було реалізовано повний цикл розробки, програмної реалізації та експериментальної перевірки гібридного алгоритму управління ресурсами у хмарних системах, який поєднує генетичний алгоритм (GA) та методи

підкріплювального навчання (RL). На основі отриманих результатів можна сформулювати висновки.

По-перше, розроблена архітектура системи управління ресурсами, побудована на принципах MAPE-K, забезпечує цілісну інтеграцію модулів моніторингу, аналізу, планування, виконання та накопичення знань. Це дозволило створити повнофункціональний прототип, здатний автоматично реагувати на зміни навантаження та вибирати оптимальні конфігурації ресурсів у режимі реального часу. Важливо, що обраний підхід є універсальним і може бути адаптований до будь-якої хмарної платформи, включаючи AWS, Kubernetes або локальні кластери.

По-друге, програмна реалізація гібридного алгоритму GA+RL підтвердила ефективність поєднання еволюційного та адаптивного методів оптимізації. Генетичний алгоритм забезпечував глобальний пошук оптимальних порогів масштабування та початкових конфігурацій ресурсів, які уникали локальних мінімумів. Агент RL, у свою чергу, продемонстрував здатність адаптуватися до динамічних змін навантаження, формуючи політики, які покращуються з досвідом. Поєднання GA і RL забезпечило прискорення процесу підготовки агента RL і підвищення стабільності прийнятих рішень, що є значною перевагою перед чистими підходами.

По-третє, у межах розділу було проведено розгорнуте експериментальне дослідження, яке включало реалізацію та тестування п’яти сценаріїв масштабування: статичного (А), евристичного (Б), генетичного (В), RL-агента (Г) та гібридного (Д).

Результати продемонстрували, що:

– статичне масштабування має найменшу адаптивність та найвищі витрати;

– евристичне масштабування покращує економічну складову, але залишається чутливим до коливань навантаження;

– генетичний алгоритм показує найкращу загальну ефективність за метрикою total_reward;

– RL-агент досягає найменшої затримки, проте за рахунок збільшення вартості;

– гібридний агент GA+RL демонструє найбільшу адаптивність, здатність до самонавчання та швидку реакцію на зміну умов.

Порівняльний аналіз підтвердив, що жоден із традиційних окремих методів не забезпечує такого поєднання глобальної оптимізації, адаптації в реальному часі та можливості балансування між продуктивністю та вартістю, як гібридний підхід.

По-четверте, результати дослідження підтвердили, що розроблений гібридний алгоритм GA+RL демонструє значні переваги над існуючими методами, зокрема:

– підвищує адаптивність до нерівномірних та динамічних навантажень;

– забезпечує більшу стабільність масштабування у порівнянні з пороговими алгоритмами;

– дозволяє зменшити експлуатаційні витрати порівняно зі статичними методами;

– демонструє кращу латентність у пікових сценаріях;

– забезпечує швидшу збіжність RL-агента завдяки оптимізованій початковій політиці, знайденій GA;

– є незалежним від конкретної хмарної платформи та може бути перенесений у реальні кластери.

На завершення третій розділ підтвердив, що розроблений алгоритм управління гібридними ресурсами в хмарних системах доцільно застосовувати в реальних інфраструктурах, особливо у випадках змінного та непередбачуваного навантаження. Отримані результати доводять, що поєднання еволюційних алгоритмів та навчання з підкріпленням дозволяє досягти значно вищої ефективності управління ресурсами порівняно з традиційними підходами. Це робить запропоновану модель перспективною для подальших досліджень і практичного використання.

ВИСНОВКИ

У даній магістерській роботі було проведено комплексне теоретичне, методичне та експериментальне дослідження методів управління ресурсами інформаційних систем у хмарних середовищах. На основі аналізу хмарних архітектур, сучасних платформ та моделей надання послуг (IaaS, PaaS, SaaS), а також оцінки проблем і викликів у сферах масштабованості, надійності, безпеки та економічної ефективності сформовано фундаментальне розуміння особливостей функціонування хмарних інформаційних систем і вимог до адаптивного управління їхніми ресурсами.

Було створено теоретико-методичну базу управління ресурсами як багатокритеріальної задачі системного аналізу. Розглянуто критерії продуктивності, доступності, безпеки, вартості, енергоефективності та масштабованості, запропоновано відповідні математичні моделі оптимізації.

Крім того, було обґрунтовано доцільність та раціональність застосування інтелектуальних методів - генетичних алгоритмів, нейронних мереж, методів навчання з підкріпленням та гібридних підходів. Ці методи дозволяють приймати рішення в умовах невизначеності та динамічних змін навантаження в хмарному середовищі.

У третьому розділі розроблено та реалізовано процес автоматизації гібридного алгоритму GA+RL. Цей алгоритм поєднує сильні сторони еволюційних методів (здатність до глобального пошуку) та навчання з підкріпленням (адаптацію до змін середовища та здатність до самонавчання). Створено архітектуру програмної системи, що включає модулі моніторингу, оптимізації, навчання та оркестрації (MAPE-K). Реалізовано симуляційне середовище, яке відтворює поведінку хмарної інфраструктури, та проведено порівняльні експерименти за сценаріями статичного управління, евристичного підходу, GA, RL та запропонованого гібридного підходу.

Результати експериментів підтвердили, що гібридний метод GA+RL забезпечує найкращий баланс між продуктивністю, вартістю, адаптивністю та мінімізацією порушень угод про рівень обслуговування за наявності (SLA).

Порівняно зі статичними та евристичними підходами, гібридна модель демонструє:

– зменшення середніх витрат на ресурси завдяки точнішому масштабуванню;

– зменшення пікової затримки та підвищення стабільності обслуговування;

– зниження кількості непотрібних масштабувань;

– підвищення ефективності використання інстансів;

– здатність адаптуватися до різних профілів навантаження без додаткового втручання користувача.

Отримані результати підтверджують доцільність використання гібридних інтелектуальних методів для розв’язання завдань управління ресурсами в хмарних системах, особливо в умовах динамічних змін навантаження, високих вимог до SLA та обмежень за вартістю. Розроблений підхід може бути інтегрований у сучасні платформи, такі як AWS, Kubernetes, OpenStack та інші системи автоскейлінгу. Таким чином, поставлена в вступі мета дослідження та всі визначені завдання повністю досягнуті.

Наукова новизна роботи полягає в обґрунтуванні та реалізації гібридного алгоритму GA+RL для оптимізації управління ресурсами хмарних обчислень, а практична значущість - у можливості використання розробленого прототипу для підвищення ефективності хмарних інфраструктур.

ПЕРЕЛІК ПОСИЛАНЬ

1. Mell, P., & Grance, T. *The NIST Definition of Cloud Computing*. National Institute of Standards and Technology, 2011.
2. Armbrust, M. et al. *A View of Cloud Computing*. Communications of the ACM, 2010, Vol. 53(4), pp. 50–58.
3. Buyya, R., Broberg, J., Goscinski, A. *Cloud Computing: Principles and Paradigms*. Wiley, 2011.
4. Erl, T., Puttini, R., Mahmood, Z. *Cloud Computing: Concepts, Technology & Architecture*. Prentice Hall, 2013.
5. Marinescu, D. *Cloud Computing: Theory and Practice*. Morgan Kaufmann, 2017.
6. Rajkumar Buyya, Rodrigo N. Calheiros, Amir Vahid Dastjerdi. *Big Data: Principles and Paradigms*. Morgan Kaufmann, 2016.
7. Bernstein, D., & Vij, D. *Intercloud Security Considerations*. 2nd IEEE International Conference on Cloud Computing Technology and Science (CloudCom), 2010.
8. Mahmood, Z. (ed.). *Cloud Computing: Challenges, Limitations and R&D Solutions*. Springer, 2014.
9. Villari, M., Fazio, M., Dustdar, S. *Multi-Cloud Management and Interoperability*. IEEE Cloud Computing, 2016, Vol. 3(2), pp. 90–95.
10. Dean, J., & Ghemawat, S. *MapReduce: Simplified Data Processing on Large Clusters*. Communications of the ACM, 2008, Vol. 51(1), pp. 107–113.
11. Piekarska, M. et al. *Green Cloud Computing: Energy Efficiency and Sustainability*. Journal of Cloud Computing, 2020.
12. Fox, A., Griffith, R. *Above the Clouds: A Berkeley View of Cloud Computing*. University of California, Berkeley, 2009.
13. Saaty T. L. *The Analytic Hierarchy Process: Planning, Priority Setting, Resource Allocation.* McGraw-Hill, 1980.
14. Zadeh L. A. *Fuzzy Sets. Information and Control*, 1965, Vol. 8, pp. 338–353.
15. Deb K. *Multi-Objective Optimization Using Evolutionary Algorithms.* John Wiley & Sons, 2001.

16. Goldberg D. E. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, 1989.
17. Marler R. T., Arora J. S. *Survey of Multi-Objective Optimization Methods for Engineering. Structural and Multidisciplinary Optimization*, 2004, Vol. 26(6), pp. 369–395.
18. Hwang C.-L., Yoon K. *Multiple Attribute Decision Making: Methods and Applications.* Springer-Verlag, 1981.
19. Calheiros R. N., Ranjan R., Beloglazov A., De Rose C. A. F., Buyya R. *CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms. Software: Practice and Experience*, 2011, Vol. 41(1), pp. 23–50.
20. Sutton R. S., Barto A. G. *Reinforcement Learning: An Introduction.* MIT Press, 2018.
21. Kennedy J., Eberhart R. *Particle Swarm Optimization. Proceedings of the IEEE International Conference on Neural Networks*, 1995, pp. 1942–1948.
22. Kirkpatrick S., Gelatt C. D., Vecchi M. P. *Optimization by Simulated Annealing. Science*, 1983, Vol. 220(4598), pp. 671–680.
23. Jain R., Paul S. *Network Performance Modeling and Simulation.* Springer, 2017.
24. Caron E., Desprez F., Muresan A. *Simulation of Dynamic Cloud Platforms Using SimGrid. Proceedings of the IEEE International Conference on Cloud Computing Technology and Science*, 2012.

ДОДАТОК А

```
import time  # Імпортуємо модуль time для роботи з паузами та позначками часу
import psutil  # Імпортуємо psutil для збору системних метрик (CPU, RAM тощо)
import pandas as pd  # Імпортуємо pandas для зручного збереження даних у таблиці (DataFrame)
from datetime import datetime  # Імпортуємо datetime для красивого формату часу
def get_system_metrics():
    """
    Функція для збору поточних системних метрик.
    Повертає словник з часом, завантаженням CPU та використанням пам'яті.
    """
    # Отримуємо завантаження CPU в відсотках (за замовчуванням psutil.cpu_percent() дивиться на останній інтервал)
    cpu_percent = psutil.cpu_percent(interval=None)
    # Отримуємо інформацію про віртуальну пам'ять (RAM)
    memory_info = psutil.virtual_memory()
    # Витягуємо відсоткове використання оперативної пам'яті
    memory_percent = memory_info.percent
    # Формуємо мітку часу у зручному форматі "YYYY-MM-DD HH:MM:SS"
    timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    # Повертаємо дані у вигляді словника для зручного перетворення в pandas DataFrame
    return {
        "timestamp": timestamp,      # Поточний час
        "cpu_percent": cpu_percent,  # Завантаження CPU у відсотках
        "memory_percent": memory_percent  # Використання RAM у відсотках
    }
```

```
def collect_metrics(duration_seconds=30, interval_seconds=1, output_csv="metrics.csv"):
    """
    Основна функція для збору метрик за певний час.
    :param duration_seconds: загальна тривалість збору метрик у секундах
    :param interval_seconds: інтервал між вимірюваннями у секундах
    :param output_csv: ім'я вихідного CSV-файлу для збереження результатів
    """
    # Створюємо порожній список, куди будемо додавати кожен вимір як словник
    collected_data = []
    # Обчислюємо момент завершення збору метрик: поточний час + задана тривалість
    end_time = time.time() + duration_seconds
    # Виводимо службове повідомлення в консоль, щоб користувач бачив, що процес розпочався
    print(f"Починаємо збір метрик на {duration_seconds} секунд з інтервалом {interval_seconds} с...")
    print(f"Результати будуть збережені у файл: {output_csv}")
    print("-" * 30)
    # Запускаємо цикл, який працює, поки поточний час не перевищить час завершення
    while time.time() < end_time:
        # Викликаємо функцію збору метрик і отримуємо словник з даними
        metrics = get_system_metrics()
        # Додаємо отриманий словник до списку усіх вимірювань
        collected_data.append(metrics)
        # Виводимо поточні метрики у консоль для наочного контролю
        print(f"{metrics['timestamp']} | CPU: {metrics['cpu_percent']:.1f}% | RAM: {metrics['memory_percent']:.1f}%")
```

```
            # Робимо паузу на вказаний інтервал між вимірюваннями
            time.sleep(interval_seconds)
    # Після завершення циклу перетворюємо список словників у pandas DataFrame (табличний формат)
    df = pd.DataFrame(collected_data)
    # Зберігаємо DataFrame у CSV-файл без індексу (index=False), щоб не дублювати нумерацію
    df.to_csv(output_csv, index=False, encoding="utf-8")
    # Виводимо службове повідомлення про завершення збирання метрик
    print("-" * 30)
    print(f"Збір метрик завершено. Дані збережені у файл: {output_csv}")
    # Повертаємо DataFrame на випадок, якщо захочемо з ним працювати далі у коді
    return df
if __name__ == "__main__":
    """
    Точка входу в програму.
    Цей блок виконується лише тоді, коли файл запускається безпосередньо (а не імпортується як модуль).
    """
    # Викликаємо функцію збору метрик з параметрами за замовчуванням:
    #  тривалість 30 секунд, інтервал 1 секунда, файл metrics.csv
    collect_metrics(duration_seconds=30, interval_seconds=1, output_csv="metrics.csv")
```

ДОДАТОК Б

```
import pandas as pd  # Для читання CSV із метриками
import random        # Для моделювання випадкових коливань
import math          # Для математичних операцій
class CloudEnvironment:
    """
    Клас CloudEnvironment моделює роботу простої хмарної системи.
    Він бере реальні метрики CPU/RAM з файлу, а також дозволяє
    змінювати кількість інстансів, як це робить AutoScaling у хмарі.
    """
    def __init__(self, metrics_file="metrics.csv", base_latency=50, max_instances=10):
        """
        Ініціалізація середовища.
        :param metrics_file: шлях до файлу з метриками (CSV)
        :param base_latency: базова затримка (мс) при мінімальному навантаженні
        :param max_instances: максимальна кількість інстансів (ліміт масштабування)
        """
        # Завантажуємо метрики CPU/RAM з CSV, які ти зібрав
        self.metrics = pd.read_csv(metrics_file)
        # Перевіримо, чи є потрібні стовпці (захист від помилок)
        if not {"cpu_percent", "memory_percent"}.issubset(self.metrics.columns):
            raise ValueError("Файл метрик має містити стовпці cpu_percent та memory_percent")
        # Початкова кількість інстансів (наприклад, 1)
        self.instances = 1
        # Базовий час відповіді системи
        self.base_latency = base_latency
        # Обмеження на максимальну кількість інстансів
        self.max_instances = max_instances
```

```
    # Поточний крок (індекс рядка в CSV)
    self.step = 0
def reset(self):
    """
    Скидає симулятор до початкового стану.
    Повертає перший стан середовища.
    """
    self.instances = 1
    self.step = 0
    return self.get_state()
def get_state(self):
    """
    Формує вектор стану.
    Стан складається з:
    - CPU (реальний з файлу)
    - RAM (реальний з файлу)
    - instances (кількість інстансів)
    - estimated latency (оцінена затримка)
    - estimated cost (розрахована вартість)
    """
    cpu = float(self.metrics.iloc[self.step]["cpu_percent"])
    ram = float(self.metrics.iloc[self.step]["memory_percent"])
    latency = self.compute_latency(cpu)
    cost = self.compute_cost()
    return {
        "cpu": cpu,
        "ram": ram,
        "instances": self.instances,
        "latency": latency,
```

```
        "cost": cost
    }
def compute_latency(self, cpu_load):
    """
    Обчислюємо затримку відповіді системи на основі:
    - реального CPU навантаження
    - кількості інстансів
    """
    # Менше інстансів → більше навантаження → більша затримка
    load_factor = cpu_load / max(self.instances, 1)
    # Формула латентності
    latency = self.base_latency * (1 + load_factor / 100)
    # Моделюємо невеликий шум як у реальних системах
    return latency + random.uniform(-5, 5)
def compute_cost(self):
    """
    Розрахунок вартості.
    Приклад моделі вартості:
    - кожен інстанс = 0.05 одиниць за крок
    """
    base_price = 0.05
    return self.instances * base_price
def step_forward(self, action):
    """
    Виконує один крок симуляції з дією агента.
    :param action: 0 = нічого не робити
                   1 = збільшити інстанси
                   2 = зменшити інстанси
    :return: (новий_стан, винагорода, done)
    """
```

```
    # Виконуємо дію
    if action == 1:  # масштабування вгору
        if self.instances < self.max_instances:
            self.instances += 1
    elif action == 2:  # масштабування вниз
        if self.instances > 1:
            self.instances -= 1
    # Перехід до наступного рядка метрик
    self.step += 1
    # Чи закінчився епізод?
    done = self.step >= len(self.metrics) - 1
    # Новий стан
    new_state = self.get_state()
    # Обчислюємо винагороду
    reward = self.compute_reward(new_state)
    return new_state, reward, done
def compute_reward(self, state):
    """
    Функція винагороди:
    - менше latency → краще
    - менше cost → краще
    - менше CPU перевантаження → краще
    """
    cpu_penalty = max(0, state["cpu"] - 70)  # штраф за CPU > 70%
    latency_penalty = state["latency"] / 10
    cost_penalty = state["cost"] * 20
    # Загальна винагорода (менше штрафів = більше нагорода)
    reward = - (cpu_penalty + latency_penalty + cost_penalty)
    return reward
```

ДОДАТОК В

```
from cloud_simulator import CloudEnvironment
env = CloudEnvironment("metrics.csv")
state = env.reset()
print("Початковий стан:", state)
new_state, reward, done = env.step_forward(1)
print("Новий стан:", new_state)
print("Винагорода:", reward)
print("Закінчено?", done)
```

ДОДАТОК Г

```
import random  # Імпортуємо модуль random для генерації випадкових чисел (для ініціалізації, кросоверу, мутацій)
from copy import deepcopy  # Імпортуємо deepcopy для безпечного копіювання об'єктів (щоб не змінювати їх випадково)
from cloud_simulator import CloudEnvironment  # Імпортуємо наше середовище, створене раніше
# ------------------------ Параметри генетичного алгоритму ------------------------
POPULATION_SIZE = 20  # Розмір популяції (кількість стратегій, які ми одночасно тестуємо)
NUM_GENERATIONS = 10  # Кількість поколінь, які буде "проживати" алгоритм
ELITE_SIZE = 5  # Кількість найкращих особин, які перейдуть у наступне покоління без змін
MUTATION_PROB = 0.3  # Ймовірність мутації особини
TOURNAMENT_SIZE = 3  # Розмір турніру при відборі (скільки стратегій змагається за місце)
# ------------------------ Опис особини (стратегії) ------------------------
def create_random_individual():
    """
    Створює одну випадкову стратегію (особину).
    Стратегія задається словником з трьома параметрами:
    - up_threshold: поріг CPU, вище якого масштабувати вгору
    - down_threshold: поріг CPU, нижче якого масштабувати вниз
    - initial_instances: початкова кількість інстансів
    """
    # Випадково обираємо поріг для масштабування вгору у діапазоні 50-90%
    up_threshold = random.uniform(60, 90)
    # Випадково обираємо поріг для масштабування вниз у діапазоні 10-60%
    down_threshold = random.uniform(10, 50)
```

```
    # Гарантуємо, що поріг вниз не більший за поріг вгору (логічна умова)
    if down_threshold >= up_threshold:
        down_threshold = up_threshold - 5  # Якщо пороги перетнулися, зменшуємо нижній поріг
    # Випадково обираємо початкову кількість інстансів від 1 до 5
    initial_instances = random.randint(1, 5)
    # Повертаємо стратегію у вигляді словника
    return {
        "up_threshold": up_threshold,
        "down_threshold": down_threshold,
        "initial_instances": initial_instances
    }
def evaluate_individual(individual, metrics_file="metrics.csv"):
    """
    Оцінює одну стратегію (особину) в симуляторі.
    Повертає загальну суму винагороди за весь епізод.
    :param individual: словник з параметрами up_threshold, down_threshold, initial_instances
    :param metrics_file: файл з метриками, який використовує середовище
    """
    # Створюємо новий екземпляр середовища з реальними метриками
    env = CloudEnvironment(metrics_file=metrics_file)
    # Скидаємо середовище та отримуємо початковий стан
    state = env.reset()
    # Встановлюємо початкову кількість інстансів згідно зі стратегією
    env.instances = individual["initial_instances"]
    # Ініціалізуємо змінну для накопичення сумарної винагороди за весь епізод
    total_reward = 0.0
    # Запускаємо симуляцію поки не отримаємо ознаку завершення (done = True)
    done = False
```

```
    while not done:
        # Зчитуємо поточне CPU із стану системи
        current_cpu = state["cpu"]
        # Визначаємо дію агента на основі поточного CPU і порогів з нашої стратегії
        if current_cpu > individual["up_threshold"]:
            action = 1  # Збільшити кількість інстансів
        elif current_cpu < individual["down_threshold"]:
            action = 2  # Зменшити кількість інстансів
        else:
            action = 0  # Нічого не робити
        # Виконуємо один крок симуляції з обраною дією
        next_state, reward, done = env.step_forward(action)
        # Додаємо поточну винагороду до загальної суми
        total_reward += reward
        # Оновлюємо поточний стан системи
        state = next_state
    # Повертаємо сумарну винагороду як значення пристосованості (fitness)
    # Чим вище total_reward, тим кращою є стратегія
    return total_reward
# ------------------------ Оператор відбору (турнірний відбір) ------------------------
def tournament_selection(population, fitnesses, k=TOURNAMENT_SIZE):
    """
    Виконує турнірний відбір.
    Випадково обирає k особин із популяції і повертає найкращу з них.
    :param population: список особин
    :param fitnesses: список значень пристосованості (fitness) для кожної особини
    :param k: розмір турніру
    """
    # Отримуємо список індексів усіх особин у популяції
    indices = list(range(len(population)))
```

```
    # Випадково обираємо k індексів без повторів для турніру
    selected_indices = random.sample(indices, k=k)
    # Ініціалізуємо змінні для збереження найкращого індекса та найкращого fitness
    best_index = None
    best_fitness = None
    # Перебираємо усі обрані індекси
    for idx in selected_indices:
        # Беремо fitness для поточної особини
        fit = fitnesses[idx]
        # Якщо це перша особина або її fitness кращий (вище), ніж у поточного найкращого
        if best_index is None or fit > best_fitness:
            best_index = idx      # Запам'ятовуємо індекс як найкращий
            best_fitness = fit    # Запам'ятовуємо значення fitness як найкраще
    # Повертаємо обрану найкращу особину
    return deepcopy(population[best_index])
# ------------------------ Оператори кросоверу та мутації ------------------------
def crossover(parent1, parent2):
    """
    Виконує кросовер (схрещування) між двома батьківськими стратегіями.
    Повертає одну нову стратегію (дитину).
    :param parent1: перший батько (словник)
    :param parent2: другий батько (словник)
    """
    # Створюємо нову стратегію-дитину як копію першого батька
    child = deepcopy(parent1)
    # Для up_threshold беремо середнє між батьками
    child["up_threshold"] = (parent1["up_threshold"] + parent2["up_threshold"]) / 2.0

    # Для down_threshold також беремо середнє
```

```
    child["down_threshold"] = (parent1["down_threshold"] + parent2["down_threshold"])
/ 2.0
    # Для initial_instances обираємо випадково з батьків (щоб зберегти інтегральне
значення)
    child["initial_instances"] = random.choice(
        [parent1["initial_instances"], parent2["initial_instances"]]
    )
    # Гарантуємо логічну послідовність порогів (down < up)
    if child["down_threshold"] >= child["up_threshold"]:
        child["down_threshold"] = child["up_threshold"] - 5
    # Обмежуємо down_threshold, щоб не було негативних значень
    if child["down_threshold"] < 0:
        child["down_threshold"] = 0
    # Повертаємо отриману стратегію-дитину
    return child
def mutate(individual, mutation_prob=MUTATION_PROB):
    """
    Виконує мутацію стратегії з певною ймовірністю.
    :param individual: особина (стратегія), яку можливо буде змінено
    :param mutation_prob: ймовірність того, що відбудеться мутація
    """
    # Створюємо копію стратегії, щоб не змінювати оригінал
    mutant = deepcopy(individual)
    # Генеруємо випадкове число від 0 до 1, щоб вирішити, чи буде мутація
    if random.random() < mutation_prob:
        # Змінюємо up_threshold на невелике випадкове значення (додаємо шум)
        mutant["up_threshold"] += random.uniform(-5, 5)
    # Аналогічно для down_threshold
    if random.random() < mutation_prob:
        mutant["down_threshold"] += random.uniform(-5, 5)
```

```
    # А також для initial_instances (зміна на -1, 0 або +1)
    if random.random() < mutation_prob:
        mutant["initial_instances"] += random.choice([-1, 0, 1])
    # Обмежуємо up_threshold у допустимому діапазоні 40–95
    mutant["up_threshold"] = max(40, min(95, mutant["up_threshold"]))
    # Обмежуємо down_threshold у діапазоні 0–80
    mutant["down_threshold"] = max(0, min(80, mutant["down_threshold"]))
    # Гарантуємо, що down_threshold строго менший, ніж up_threshold
    if mutant["down_threshold"] >= mutant["up_threshold"]:
        mutant["down_threshold"] = mutant["up_threshold"] - 5
    # Обмежуємо initial_instances у діапазоні 1–10
    mutant["initial_instances"] = max(1, min(10, mutant["initial_instances"]))
    # Повертаємо мутовану стратегію
    return mutant
# ------------------------ Основний цикл генетичного алгоритму ------------------------
def run_ga(metrics_file="metrics.csv"):
    """
    Запускає повний цикл генетичного алгоритму:
    - створює початкову популяцію
    - виконує оцінювання, відбір, кросовер, мутацію
    - повертає найкращу знайдену стратегію
    :param metrics_file: файл з метриками для симулятора
    """
    # Створюємо початкову популяцію випадкових стратегій
    population = [create_random_individual() for _ in range(POPULATION_SIZE)]
    # Ініціалізуємо змінні для збереження найкращої стратегії та її fitness
    best_individual = None
    best_fitness = None
    # Проходимо по поколіннях
    for generation in range(NUM_GENERATIONS):
```

```
    print(f"\n=== Покоління {generation + 1}/{NUM_GENERATIONS} ===")
    # Обчислюємо fitness (сумарну винагороду) для кожної особини
    fitnesses = []
    for idx, individual in enumerate(population):
        fit = evaluate_individual(individual, metrics_file=metrics_file)
        fitnesses.append(fit)
        print(f"  Особина {idx + 1}: fitness = {fit:.2f}")
    # Знаходимо найкращу особину в поточному поколінні
    gen_best_index = max(range(len(population)), key=lambda i: fitnesses[i])
    gen_best_individual = population[gen_best_index]
    gen_best_fitness = fitnesses[gen_best_index]
    # Якщо це перша ітерація або знайдено кращий результат за попередні покоління
    if best_individual is None or gen_best_fitness > best_fitness:
        best_individual = deepcopy(gen_best_individual)
        best_fitness = gen_best_fitness
    # Виводимо інформацію про найкращу стратегію в поколінні
    print(f"  Найкраща особина покоління: {gen_best_individual}, fitness = {gen_best_fitness:.2f}")
    print(f"  Глобально найкраща особина: {best_individual}, fitness = {best_fitness:.2f}")
    # ---------- Створення нового покоління ----------
    # Сортуємо популяцію за fitness (за спаданням)
    sorted_indices = sorted(range(len(population)), key=lambda i: fitnesses[i], reverse=True)
    # Формуємо список нової популяції, починаючи з ELITE_SIZE найкращих особин (еліт)
    new_population = [deepcopy(population[i]) for i in sorted_indices[:ELITE_SIZE]]
```

```
        # Поки нова популяція не досягла потрібного розміру, генеруємо нових нащадків
        while len(new_population) < POPULATION_SIZE:
            # Вибираємо двох батьків за допомогою турнірного відбору
            parent1 = tournament_selection(population, fitnesses)
            parent2 = tournament_selection(population, fitnesses)
            # Виконуємо кросовер, отримуємо дитину
            child = crossover(parent1, parent2)
            # Виконуємо мутацію над дитиною
            child = mutate(child)
            # Додаємо нову стратегію до нової популяції
            new_population.append(child)
        # Оновлюємо популяцію на нове покоління
        population = new_population
    # Після завершення усіх поколінь повертаємо найкращу знайдену стратегію та її fitness
    return best_individual, best_fitness
if __name__ == "__main__":
    """
    Точка входу в програму.
    Запускається, коли файл виконують безпосередньо (python ga_optimizer.py).
    """
    # Запускаємо генетичний алгоритм та отримуємо найкращу стратегію
    best_strategy, best_fit = run_ga(metrics_file="metrics.csv")
    # Виводимо фінальний результат на екран
    print("\n=== Результат роботи генетичного алгоритму ===")
    print(f"Найкраща стратегія: {best_strategy}")
    print(f"Її сумарний fitness: {best_fit:.2f}")
```

ДОДАТОК Д

```
import random  # Модуль random потрібен для вибору випадкових дій (epsilon-greedy) та інших випадкових операцій
from collections import defaultdict  # defaultdict спростить роботу з Q-таблицею
from cloud_simulator import CloudEnvironment  # Імпортуємо наше середовище, яке ми вже створили раніше
class QLearningAgent:
    """
    Клас QLearningAgent реалізує простий табличний алгоритм Q-learning.
    Агент вчиться на основі винагороди, яку він отримує від середовища CloudEnvironment.
    """
    def __init__(self,
                 alpha=0.1,
                 gamma=0.9,
                 epsilon=1.0,
                 epsilon_min=0.05,
                 epsilon_decay=0.99):
        """
        Ініціалізація параметрів агента.
        :param alpha: коефіцієнт навчання (learning rate)
        :param gamma: коефіцієнт дисконтування майбутньої винагороди
        :param epsilon: початкове значення epsilon для epsilon-greedy політики
        :param epsilon_min: мінімально допустиме значення epsilon
        :param epsilon_decay: множник для поступового зменшення epsilon після кожного епізоду
        """
        # Коефіцієнт навчання. Визначає, наскільки сильно ми оновлюємо Q-значення
        self.alpha = alpha
```

```
        # Фактор дисконту. Визначає, наскільки важлива майбутня винагорода порівняно з поточною
        self.gamma = gamma
        # Поточне значення epsilon (ступінь випадковості в діях)
        self.epsilon = epsilon
        # Мінімальне значення epsilon, нижче якого ми не зменшуємо
        self.epsilon_min = epsilon_min
        # Множник, на який ми множимо epsilon після кожного епізоду (епізод = один прогін по метриках)
        self.epsilon_decay = epsilon_decay
        # Q-таблиця. Для кожного стану зберігається список значень Q для 3 дій (0, 1, 2)
        # Використовуємо defaultdict, щоб при першому зверненні до нового стану автоматично створювався [0.0, 0.0, 0.0]
        self.q_table = defaultdict(lambda: [0.0, 0.0, 0.0])
    def discretize_state(self, state):
        """
        Дискретизуємо стан середовища.
        :param state: словник зі станом (cpu, ram, instances, latency, cost)
        :return: кортеж (cpu_bin, instances), який будемо використовувати як ключ для Q-таблиці
        """
        # Беремо CPU-значення зі стану
        cpu = state["cpu"]
        # Ділимо CPU на "кошики" по 10%. Наприклад, cpu=37 -> 3 (тобто інтервал 30-40)
        cpu_bin = int(cpu // 10)
        # Беремо кількість інстансів із стану
        instances = state["instances"]
        # Формуємо дискретизований стан як кортеж
```

```
        return (cpu_bin, instances
    def select_action(self, state_key):
        """
        Обирає дію на основі epsilon-greedy політики.
        :param state_key: дискретизований стан (cpu_bin, instances)
        :return: action (0, 1 або 2)
        """
        # Генеруємо випадкове число від 0 до 1, щоб вирішити, чи діяти випадково
        if random.random() < self.epsilon:
            # Випадкова дія: 0 = нічого не робити, 1 = масштабувати вгору, 2 = масштабувати вниз
            return random.choice([0, 1, 2])

        # Якщо не випадкова дія, обираємо дію з максимальним Q-значенням для поточного стану
        q_values = self.q_table[state_key]  # Отримуємо список [Q(s,0), Q(s,1), Q(s,2)]
        max_q = max(q_values)  # Знаходимо максимальне Q-значення у поточному стані
        # Знаходимо всі індекси дій, які мають це максимальне значення (на випадок кількох однаково найкращих)
        best_actions = [action for action, q in enumerate(q_values) if q == max_q]
        # Випадково обираємо одну з найкращих дій (якщо їх там декілька)
        return random.choice(best_actions)
    def update_q(self, state_key, action, reward, next_state_key, done):
        """
        Оновлює Q-значення за класичною формулою Q-learning.
        :param state_key: поточний стан (дискретизований)
        :param action: обрана дія (0, 1 або 2)
        :param reward: отримана винагорода
        :param next_state_key: новий стан після виконання дії (дискретизований)
```

```
        :param done: True, якщо епізод завершився; False інакше
        """
        # Поточне Q-значення для пари (стан, дія)
        current_q = self.q_table[state_key][action
        # Якщо епізод завершився, майбутнє значення Q вважаємо 0 (немає наступного стану)
        if done:
            max_next_q = 0.0
        else:
            # Інакше беремо максимальне Q-значення серед усіх дій у наступному стані
            max_next_q = max(self.q_table[next_state_key])
        # Класична формула оновлення Q:
        # Q(s,a) = Q(s,a) + alpha * (reward + gamma * max_a' Q(s',a') - Q(s,a))
        new_q = current_q + self.alpha * (reward + self.gamma * max_next_q - current_q)
        # Записуємо оновлене значення Q назад у таблицю
        self.q_table[state_key][action] = new_q
    def decay_epsilon(self):
        """
        Поступово зменшує epsilon після кожного епізоду,
        щоб агент з часом менше діяв випадково і більше покладався на вивчену політику.
        """
        # Множимо epsilon на коефіцієнт зменшення
        self.epsilon *= self.epsilon_decay
        # Обмежуємо epsilon знизу значенням epsilon_min
        if self.epsilon < self.epsilon_min:
            self.epsilon = self.epsilon_min
def train_q_agent(
        metrics_file="metrics.csv",
        num_episodes=50
```

```
):
    """
    Функція для навчання Q-learning агента на основі CloudEnvironment.
    :param metrics_file: файл з метриками, які використовує середовище
    :param num_episodes: кількість епізодів навчання (скільки разів пройдемо по метриках)
    :return: навченого агента
    """
    # Створюємо екземпляр агента QLearningAgent з типовими параметрами
    agent = QLearningAgent()
    # Проходимо по всіх епізодах навчання
    for episode in range(num_episodes):
        print(f"\n=== Епізод {episode + 1}/{num_episodes} ===")
        # На кожен епізод створюємо нове середовище, щоб епізод починався "з нуля"
        env = CloudEnvironment(metrics_file=metrics_file)
        # Скидаємо середовище і отримуємо початковий стан
        state = env.reset()
        # Дискретизуємо стан для роботи з Q-таблицею
        state_key = agent.discretize_state(state)
        # Ініціалізуємо накопичувач сумарної винагороди за епізод
        total_reward = 0.0
        # Ознака завершення епізоду
        done = False

        # Поки епізод не завершився, виконуємо кроки
        while not done:
            # Обираємо дію на основі поточного стану і політики epsilon-greedy
            action = agent.select_action(state_key)
            # Виконуємо один крок у середовищі: отримуємо новий стан, винагороду та ознаку завершення
```

```
            next_state, reward, done = env.step_forward(action)
            # Дискретизуємо новий стан для роботи з Q-таблицею
            next_state_key = agent.discretize_state(next_state)
            # Оновлюємо Q-значення для поточної пари (стан, дія)
            agent.update_q(state_key, action, reward, next_state_key, done)
            # Додаємо винагороду поточного кроку до сумарної винагороди за епізод
            total_reward += rewar
            # Переходимо до наступного стану
            state_key = next_state_key
        # Після завершення епізоду зменшуємо epsilon (щоб з часом агент ставав менш випадковим)
        agent.decay_epsilon()
        # Виводимо сумарну винагороду за епізод та поточне значення epsilon
        print(f"Сумарна винагорода за епізод: {total_reward:.2f}")
        print(f"Поточне epsilon: {agent.epsilon:.3f}")
    # Після завершення усіх епізодів повертаємо навченого агента
    return agent
def evaluate_agent(agent, metrics_file="metrics.csv"):
    """
    Оцінює вже навченого агента без дослідження (epsilon = 0).
    :param agent: навченний QLearningAgent
    :param metrics_file: файл з метриками для середовища
    :return: сумарна винагорода та список кроків (для аналізу)
    """
    # Тимчасово зберігаємо поточне epsilon, щоб відновити його пізніше
    old_epsilon = agent.epsilon
    # Встановлюємо epsilon = 0, щоб агент діяв тільки жадібно (greedy) за найкращою політикою
    agent.epsilon = 0.0
    # Створюємо нове середовище
```

```
env = CloudEnvironment(metrics_file=metrics_file)
# Скидаємо середовище
state = env.reset()
# Дискретизуємо стан
state_key = agent.discretize_state(state)
# Ініціалізуємо сумарну винагороду
total_reward = 0.0

# Список для збереження історії (стан, дія, винагорода)
trajectory = []
# Ознака завершення епізоду
done = False
# Запускаємо епізод
while not done:
    # Обираємо найкращу дію (оскільки epsilon = 0, select_action діє жадібно)
    action = agent.select_action(state_key)
    # Виконуємо крок у середовищі
    next_state, reward, done = env.step_forward(action)
    # Дискретизуємо наступний стан
    next_state_key = agent.discretize_state(next_state)
    # Додаємо винагороду до сумарної
    total_reward += reward
    # Зберігаємо інформацію про цей крок для аналізу
    trajectory.append({
        "state": state,
        "action": action,
        "reward": reward,
        "next_state": next_state
    })
    # Переходимо до наступного стану
```

```
            state = next_state
            state_key = next_state_key
        # Відновлюємо старе значення epsilon
        agent.epsilon = old_epsilon
        # Повертаємо сумарну винагороду та усю траєкторію
        return total_reward, trajectory
if __name__ == "__main__":
    """
    Точка входу в програму.
    Тут ми запускаємо навчання агента, а потім його оцінку.
    """
    # Запускаємо навчання агента на основі метрик із файлу metrics.csv
    trained_agent = train_q_agent(metrics_file="metrics.csv", num_episodes=50)
    # Оцінюємо вже навченого агента без випадкових дій
    eval_reward, eval_trajectory = evaluate_agent(trained_agent,
metrics_file="metrics.csv")
    # Виводимо сумарну винагороду при оцінці
    print("\n=== Оцінка навченого агента ===")
    print(f"Сумарна винагорода при оцінці: {eval_reward:.2f}")
    print(f"Кількість кроків у епізоді: {len(eval_trajectory)}")
```

ДОДАТОК Е

```
from ga_optimizer import run_ga  # Імпортуємо функцію запуску GA, щоб отримати найкращу стратегію

from rl_agent import QLearningAgent, evaluate_agent  # Імпортуємо QLearningAgent та функцію оцінки агента

from cloud_simulator import CloudEnvironment  # Імпортуємо середовище, з яким буде працювати агент

import math  # Модуль math може стати в пригоді для обчислень (на випадок розширення)

def initialize_q_from_ga(agent, best_strategy, max_instances=10):
    """
    Ініціалізує Q-таблицю агента на основі найкращої стратегії,
    знайденої генетичним алгоритмом.
    :param agent: екземпляр QLearningAgent
    :param best_strategy: словник з полями up_threshold, down_threshold, initial_instances
    :param max_instances: максимальна кількість інстансів для генерації станів
    """
    # Витягуємо пороги масштабування з найкращої стратегії GA
    up_threshold = best_strategy["up_threshold"]      # Поріг CPU для масштабування вгору
    down_threshold = best_strategy["down_threshold"]  # Поріг CPU для масштабування вниз
```

```
    # Проходимо всі можливі кошики CPU (0–10, що відповідає 0–100% з кроком 10)
    for cpu_bin in range(0, 11):
        # Обчислюємо "середнє" значення CPU для цього кошика (наприклад, bin=3 -> 35%)
        cpu_value = cpu_bin * 10 + 5
        # Проходимо всі можливі значення кількості інстансів від 1 до max_instances
        for instances in range(1, max_instances + 1):
            # Формуємо дискретизований стан у форматі, з яким працює агент
            state_key = (cpu_bin, instances)
            # Створюємо базовий список Q-значень для дій [нічого, вгору, вниз]
            q_values = [0.0, 0.0, 0.0]
            # Якщо CPU вище порога up_threshold, підсилюємо дію "масштабувати вгору" (action=1)
            if cpu_value > up_threshold:
                q_values[1] = 1.0    # Перевага дії "масштабувати вгору"
                q_values[2] = -1.0   # Зменшуємо привабливість зменшення інстансів
            # Якщо CPU нижче порога down_threshold, підсилюємо дію "масштабувати вниз" (action=2)
            elif cpu_value < down_threshold:
                q_values[2] = 1.0    # Перевага дії "масштабувати вниз"
                q_values[1] = -1.0   # Зменшуємо доцільність збільшення інстансів
            else:
                # Якщо CPU між порогами, перевага "нічого не робити"
```

```
            q_values[0] = 0.5    # Трохи позитивне значення для дії "нічого не робити"
            # Решта значень залишаються 0.0
        # Записуємо початкові Q-значення для даного стану в Q-таблицю агента
        agent.q_table[state_key] = q_values
def train_hybrid_agent(metrics_file="metrics.csv", num_episodes=30):
    """
    Навчає гібридного агента GA+RL:
    1) Спочатку запускається GA для знаходження найкращої стратегії.
    2) На основі цієї стратегії ініціалізується Q-таблиця RL-агента.
    3) Далі RL-агент продовжує навчання у середовищі CloudEnvironment.
    :param metrics_file: файл із метриками, які використовує середовище
    :param num_episodes: кількість епізодів навчання RL-частини
    :return: навченого гібридного агента та найкращу стратегію GA
    """
    # КРОК 1: Запускаємо GA для пошуку найкращої стратегії
    print("=== ЕТАП 1: Запуск генетичного алгоритму для пошуку стартової стратегії ===")
    best_strategy, best_fitness = run_ga(metrics_file=metrics_file)
    # Виводимо короткий підсумок роботи GA
    print("\nНайкраща стратегія з GA:")
    print(f"  up_threshold      = {best_strategy['up_threshold']:.2f}")
    print(f"  down_threshold    = {best_strategy['down_threshold']:.2f}")
    print(f"  initial_instances = {best_strategy['initial_instances']}")
```

```
    print(f"  fitness          = {best_fitness:.2f}")
    # КРОК 2: Створюємо RL-агента
    print("\n=== ЕТАП 2: Створення RL-агента та ініціалізація з GA-стратегії ===")
    agent = QLearningAgent()  # Створюємо Q-learning агента з параметрами за замовчуванням
    # Ініціалізуємо Q-таблицю агента на основі інформації з GA
    initialize_q_from_ga(agent, best_strategy, max_instances=10)
    # Додатково можемо встановити epsilon меншим, ніж 1.0, бо вже маємо непогану стартову стратегію
    agent.epsilon = 0.5  # Менше випадкових дій, більше спираємося на GA-політику
    # КРОК 3: Навчання RL-агента в середовищі
    print("\n=== ЕТАП 3: Навчання RL-агента поверх стартової стратегії GA ===")
    # Запускаємо цикл по епізодах навчання
    for episode in range(num_episodes):
        print(f"\n--- Гібридний епізод {episode + 1}/{num_episodes} ---")
        # Для кожного епізоду створюємо новий екземпляр середовища
        env = CloudEnvironment(metrics_file=metrics_file)
        # Скидаємо середовище та отримуємо початковий стан
        state = env.reset()
        # Встановлюємо початкову кількість інстансів згідно з GA-стратегією
        env.instances = best_strategy["initial_instances"]
        # Дискретизуємо стан для роботи з Q-таблицею
        state_key = agent.discretize_state(state)
```

```
        # Накопичувач сумарної винагороди за епізод
        total_reward = 0.0
        # Ознака завершення епізоду
        done = False
        # Поки епізод триває, виконуємо кроки взаємодії з середовищем
        while not done:
            # Обираємо дію з урахуванням поточного epsilon (гібрид випадковості та політики)
            action = agent.select_action(state_key)
            # Виконуємо крок у середовищі та отримуємо новий стан, винагороду та ознаку завершення
            next_state, reward, done = env.step_forward(action)
            # Дискретизуємо новий стан
            next_state_key = agent.discretize_state(next_state)
            # Оновлюємо Q-значення на основі спостереження (s, a, r, s')
            agent.update_q(state_key, action, reward, next_state_key, done)
            # Додаємо винагороду до сумарної
            total_reward += reward
            # Переходимо до наступного стану
            state_key = next_state_key
        # Після завершення епізоду зменшуємо epsilon (агент стає менш випадковим)
        agent.decay_epsilon()
```

```
        # Виводимо інформацію про епізод
        print(f"Сумарна винагорода (гібрид GA+RL) за епізод: {total_reward:.2f}")
        print(f"Поточне epsilon: {agent.epsilon:.3f}")
    # Повертаємо навченого агента та найкращу стратегію GA
    return agent, best_strategy
if __name__ == "__main__":
    """
    Точка входу в програму для гібридного підходу GA+RL.
    Тут ми:
      1) навчаємо гібридного агента,
      2) оцінюємо його якість,
      3) виводимо підсумкову інформацію.
    """
    # Вказуємо файл із метриками, зібраними раніше
    metrics_path = "metrics.csv"
    # Запускаємо навчання гібридного агента GA+RL
    hybrid_agent, ga_best_strategy = train_hybrid_agent(metrics_file=metrics_path, num_episodes=30)
    # Оцінюємо навченого гібридного агента (без випадковості)
    eval_reward, eval_trajectory = evaluate_agent(hybrid_agent, metrics_file=metrics_path)
    # Виводимо підсумки
    print("\n=== ПІДСУМКИ ГІБРИДНОГО ПІДХОДУ GA+RL ===")
```

```
print("GA-стратегія, з якої стартував RL:")
print(f"  up_threshold      = {ga_best_strategy['up_threshold']:.2f}")
print(f"  down_threshold    = {ga_best_strategy['down_threshold']:.2f}")
print(f"  initial_instances = {ga_best_strategy['initial_instances']}")
print(f"\nСумарна винагорода гібридного агента при оцінці: {eval_reward:.2f}")
print(f"Кількість кроків в епізоді: {len(eval_trajectory)}")
```

ДОДАТОК Є

```
import pandas as pd  # Для роботи з табличними даними та збереження результатів у CSV
import matplotlib.pyplot as plt  # Для побудови графіків результатів
from cloud_simulator import CloudEnvironment  # Середовище, яке ми вже реалізували
from ga_optimizer import run_ga  # Функція для запуску генетичного алгоритму
from rl_agent import QLearningAgent, train_q_agent, evaluate_agent  # RL-агент та функції навчання/оцінки
from hybrid_ga_rl import train_hybrid_agent  # Функція навчання гібридного агента GA+RL
def run_policy(policy_name, policy_fn, metrics_file="metrics.csv", initial_instances=1):
    """
    Універсальна функція для прогону політики (статичної чи евристичної) через середовище.
    :param policy_name: назва сценарію (для логів)
    :param policy_fn: функція, яка за станом повертає дію (0, 1 або 2)
    :param metrics_file: шлях до файлу з метриками
    :param initial_instances: початкова кількість інстансів у середовищі
    :return: словник зі зведеними метриками
    """
    # Створюємо середовище на основі реальних метрик
    env = CloudEnvironment(metrics_file=metrics_file)
    # Скидаємо середовище до початкового стану
    state = env.reset()
    # Встановлюємо початкову кількість інстансів (наприклад, 1, 2, 3...)
    env.instances = initial_instances
    # Ініціалізуємо накопичувачі для різних метрик
    total_reward = 0.0            # Сумарна винагорода за весь епізод
```

```
    total_latency = 0.0          # Сума затримок за всі кроки
    total_cost = 0.0             # Сума вартості за всі кроки
    total_cpu = 0.0              # Сума завантаження CPU за всі кроки
    total_instances = 0.0        # Сума кількості інстансів за всі кроки
    sla_violations = 0           # Лічильник порушень SLA за латентністю
    steps = 0                    # Кількість кроків симуляції
    # Основний цикл симуляції до завершення епізоду
    done = False
    while not done:
        # Вибираємо дію за допомогою переданої політики policy_fn
        action = policy_fn(state)
        # Виконуємо крок у середовищі: отримуємо наступний стан, винагороду та статус завершення
        next_state, reward, done = env.step_forward(action)
        # Оновлюємо лічильники
        total_reward += reward
        total_latency += next_state["latency"]
        total_cost += next_state["cost"]
        total_cpu += next_state["cpu"]
        total_instances += next_state["instances"]
        # Перевіряємо порушення SLA: наприклад, якщо латентність > 200 мс
        if next_state["latency"] > 200:
            sla_violations += 1
        # Збільшуємо кількість кроків
        steps += 1
        # Переходимо до наступного стану
        state = next_state
    # Обчислюємо середні значення по кроках
    avg_latency = total_latency / steps if steps > 0 else 0.0
    avg_cost = total_cost / steps if steps > 0 else 0.0
```

```
    avg_cpu = total_cpu / steps if steps > 0 else 0.0
    avg_instances = total_instances / steps if steps > 0 else 0.0
    # Формуємо словник з результатами
    return {
        "scenario": policy_name,
        "total_reward": total_reward,
        "avg_latency": avg_latency,
        "avg_cost": avg_cost,
        "avg_cpu": avg_cpu,
        "avg_instances": avg_instances,
        "sla_violations": sla_violations,
        "steps": steps
    }
def static_policy_factory():
    """
    Створює політику для сценарію A (статичне масштабування).
    У цьому сценарії інстанси не змінюються, тобто завжди дія 0.
    :return: функція, яка ігнорує стан і завжди повертає 0
    """
    def static_policy(state):
        # Ігноруємо параметри стану і нічого не змінюємо
        return 0  # 0 = нічого не робити
    # Повертаємо внутрішню функцію як політику
    return static_policy
def autoscaling_policy_factory(up_threshold=70.0, down_threshold=30.0):
    """
    Створює політику для сценарію B (евристичний autoscaling).
    :param up_threshold: поріг CPU для масштабування вгору
    :param down_threshold: поріг CPU для масштабування вниз
    :return: функція-політика
```

```
    """
    def autoscaling_policy(state):
        # Зчитуємо поточне значення CPU зі стану
        cpu = state["cpu"]
        # Якщо CPU перевищує верхній поріг, додаємо інстанс
        if cpu > up_threshold:
            return 1  # 1 = масштабування вгору
        # Якщо CPU нижче нижнього порогу, зменшуємо інстанси
        if cpu < down_threshold:
            return 2  # 2 = масштабування вниз
        # В іншому випадку нічого не робимо
        return 0
    # Повертаємо політику з "зашитими" порогами
    return autoscaling_policy
def ga_policy_factory(best_strategy):
    """
    Створює політику для сценарію C на основі найкращої стратегії GA.
    :param best_strategy: словник з полями up_threshold та down_threshold
    :return: функція-політика
    """
    up_threshold = best_strategy["up_threshold"]          # Зчитуємо поріг для масштабування вгору
    down_threshold = best_strategy["down_threshold"]    # Зчитуємо поріг для масштабування вниз
    def ga_policy(state):
        # Беремо поточне значення CPU зі стану
        cpu = state["cpu"]
        # Якщо перевищено поріг вверх — масштабування вгору
        if cpu > up_threshold:
            return 1
```

```
        # Якщо нижче порога вниз — масштабування вниз
        if cpu < down_threshold:
            return 2
        # Інакше — нічого не робити
        return 0
    # Повертаємо політику
    return ga_policy
def rl_policy_factory(agent):
    """
    Створює політику для сценарію D (чистий RL).
    :param agent: навченний QLearningAgent
    :return: функція-політика
    """
    def rl_policy(state):
        # Дискретизуємо стан за допомогою методу агента
        state_key = agent.discretize_state(state)
        # Зберігаємо старе epsilon, щоб не втручатися в політику агента глобально
        old_epsilon = agent.epsilon
        # Встановлюємо epsilon = 0, щоб діяти жадібно (без випадковості)
        agent.epsilon = 0.0
        # Обираємо дію за допомогою політики select_action
        action = agent.select_action(state_key)
        # Повертаємо epsilon до старого значення
        agent.epsilon = old_epsilon
        # Повертаємо обрану дію
        return action
    # Повертаємо політику, пов'язану з агентом
    return rl_policy
```

```
def run_all_scenarios(metrics_file="metrics.csv"):
    """
    Запускає всі сценарії A–E та повертає результати в вигляді DataFrame.
    :param metrics_file: файл з метриками
    :return: pandas DataFrame з результатами усіх сценаріїв
    """
    results = []  # Список для збирання результатів усіх сценаріїв
    # ---------- Сценарій A: Статичне масштабування ----------
    print("=== Сценарій A: Статичне масштабування ===")
    static_policy = static_policy_factory()
    res_static = run_policy("A_Static", static_policy, metrics_file=metrics_file,
initial_instances=2)
    print(res_static)
    results.append(res_static)
    # ---------- Сценарій B: Евристичний autoscaling ----------
    print("\n=== Сценарій B: Евристичний autoscaling ===")
    auto_policy = autoscaling_policy_factory(up_threshold=70.0, down_threshold=30.0)
    res_auto = run_policy("B_Autoscaling", auto_policy, metrics_file=metrics_file,
initial_instances=2)
    print(res_auto)
    results.append(res_auto)
    # ---------- Сценарій C: Тільки GA ----------
    print("\n=== Сценарій C: Тільки GA (найкраща стратегія) ===")
    best_individual, best_fitness = run_ga(metrics_file=metrics_file)
    print(f"Найкраща стратегія GA: {best_individual}, fitness = {best_fitness:.2f}")
    ga_policy = ga_policy_factory(best_individual)
    res_ga = run_policy("C_GA", ga_policy, metrics_file=metrics_file,
                        initial_instances=best_individual["initial_instances"])
    print(res_ga)
    results.append(res_ga)
```

```
    # ---------- Сценарій D: Тільки RL ----------
    print("\n=== Сценарій D: Тільки RL (Q-learning) ===")
    # Навчаємо RL-агента
    rl_agent = train_q_agent(metrics_file=metrics_file, num_episodes=30)
    # Оцінюємо агента та отримуємо траєкторію
    eval_reward_rl, trajectory_rl = evaluate_agent(rl_agent, metrics_file=metrics_file)
    print(f"Сумарна винагорода RL при оцінці: {eval_reward_rl:.2f}")
    # Додатково прогонимо RL як політику через run_policy, щоб отримати середні метрики
    rl_policy = rl_policy_factory(rl_agent)
    res_rl = run_policy("D_RL", rl_policy, metrics_file=metrics_file, initial_instances=1)
    print(res_rl)
    results.append(res_rl)
    # ---------- Сценарій E: Гібрид GA+RL ----------
    print("\n=== Сценарій E: Гібрид GA+RL ===")
    hybrid_agent, ga_best_strategy = train_hybrid_agent(metrics_file=metrics_file, num_episodes=30)
    # Оцінюємо гібридного агента
    eval_reward_hybrid, trajectory_hybrid = evaluate_agent(hybrid_agent, metrics_file=metrics_file)
    print(f"Сумарна винагорода гібридного агента при оцінці: {eval_reward_hybrid:.2f}")
    # Створюємо політику поверх гібридного агента
    hybrid_policy = rl_policy_factory(hybrid_agent)
    res_hybrid = run_policy("E_GA+RL", hybrid_policy, metrics_file=metrics_file,
                            initial_instances=ga_best_strategy["initial_instances"])
    print(res_hybrid)
    results.append(res_hybrid)
    # Перетворюємо список словників на pandas DataFrame
    df_results = pd.DataFrame(results)
```

```
    # Повертаємо DataFrame з результатами
    return df_results
def save_and_plot_results(df_results, output_csv="scenarios_results.csv"):
    """
    Зберігає результати в CSV та будує простий графік для аналізу.
    :param df_results: DataFrame з результатами сценаріїв
    :param output_csv: ім'я вихідного CSV-файлу
    """
    # Зберігаємо результати у CSV-файл для подальшого використання в магістерській
    df_results.to_csv(output_csv, index=False, encoding="utf-8")
    print(f"\nРезультати всіх сценаріїв збережено у файл: {output_csv}")
    # Будуємо графік сумарної винагороди для кожного сценарію
    plt.figure()  # Створюємо нову фігуру для графіка
    # По осі X — назви сценаріїв, по осі Y — сумарна винагорода
    plt.bar(df_results["scenario"], df_results["total_reward"])
    # Додаємо підписи до осей та заголовок
    plt.xlabel("Сценарій")
    plt.ylabel("Сумарна винагорода")
    plt.title("Порівняння сумарної винагороди для різних сценаріїв управління ресурсами")
    # Повертаємо відображення по X під кутом, щоб підписи не накладалися
    plt.xticks(rotation=45)
    # Показуємо графік на екрані (тут ти зможеш зробити скріншот)
    plt.tight_layout()
    plt.show()
if __name__ == "__main__":
    """
    Точка входу в програму.
    Тут:
```

```
  1) запускаються усі сценарії A–E,
  2) збираються результати в таблицю,
  3) зберігається CSV,
  4) будується графік для подальшого аналізу.
"""
# Вказуємо файл з метриками, зібраними раніше
metrics_path = "metrics.csv"
# Запускаємо всі сценарії та отримуємо результати у вигляді DataFrame
df = run_all_scenarios(metrics_file=metrics_path)
# Виводимо результати у консоль для візуальної перевірки
print("\n=== Зведена таблиця результатів сценаріїв ===")
print(df)
# Зберігаємо результати та будуємо графік
save_and_plot_results(df, output_csv="scenarios_results.csv")
```

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра інформаційних систем та технологій

МЕТОДИ УПРАВЛІННЯ РЕСУРСАМИ ІНФОРМАЦІЙНИХ СИСТЕМ У ХМАРНИХ СЕРЕДОВИЩАХ

Виконав: студент групи САДМ-61
Носов Єгор Дмитрович

Керівник: д.т.н., професор,
Шушура Олексій Миколайович

Київ – 2026

АКТУАЛЬНІСТЬ

2

Сучасні хмарні інформаційні системи широко використовуються в електронній комерції, потокових сервісах та корпоративних застосунках, для яких характерні різкі та нерівномірні зміни навантаження.

Під час пікових періодів це може призводити до зростання затримок, погіршення якості обслуговування та порушення SLA, а у періоди низької активності - до перевитрат обчислювальних ресурсів.

Традиційні підходи до масштабування, зокрема статичні та порогові алгоритми, не враховують комплексний стан системи та не здатні адаптуватися до непередбачуваних сценаріїв навантаження.

У зв'язку з цим актуальним є застосування інтелектуальних методів управління ресурсами, зокрема генетичних алгоритмів, підкріплювального навчання та їхніх гібридних поєднань, які дозволяють одночасно зменшити витрати та забезпечити стабільну продуктивність хмарних сервісів.

МЕТА РОБОТИ ТА ЗАВДАННЯ

3

Мета роботи: автоматизація процесу управління ресурсами у хмарних середовищах.

Завдання роботи:

1. Розглянути сучасні хмарні платформи та технології, їхню структуру та особливості;
2. Виконати математичне моделювання процесів управління ресурсами.
3. Розглянути архітектуру системи керування ресурсами.
4. Розробити програмне забезпечення для дослідження алгоритму управління ресурсами;
5. Проаналізувати розроблений алгоритм управління ресурсами та його результати.

ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

4

Об'єкт дослідження: процес управління ресурсами в хмарних системах.

Предмет дослідження: моделі, методи та інформаційні технології управління ресурсами в хмарних системах.

СУЧАСНІ ХМАРНІ ПЛАТФОРМИ 5

РОЗРОБКА МАТЕМАТИЧНОЇ МОДЕЛІ 6

Метою моделі є формалізація взаємозв'язку між навантаженням, продуктивністю, вартістю та іншими параметрами, забезпечуючи адаптивне прийняття рішень в динамічних умовах.

Для системи з n віртуальних машин та m ресурсних вузлів:

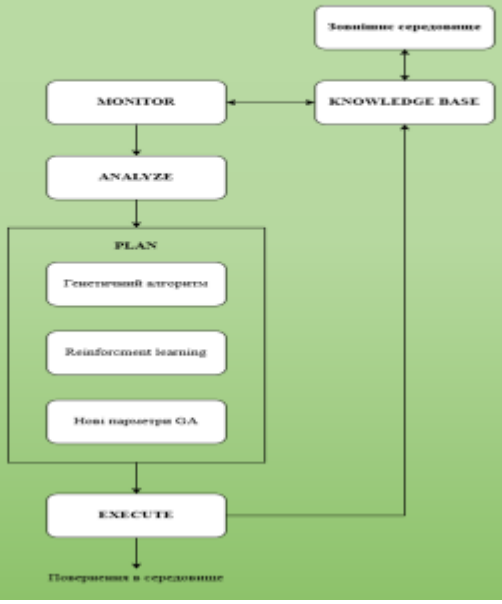
$$\max Q = \sum_{i=1}^{n}\sum_{j=1}^{m} w_1 P_{ij} + w_2 A_{ij} + w_3 S_{ij} - w_4 C_{ij} - w_5 E_{ij} \quad ,$$

За умов:

$$\sum_{i=1}^{n} CPU_{ij} \le CPU_j^{max},\ \sum_{i=1}^{n} RAM_{ij} \le RAM_j^{max}, x_{ij} \in \{0,1\}$$

АРХІТЕКТУРА СИСТЕМИ КЕРУВАННЯ РЕСУРСАМИ

7



МЕТРИКИ ДЛЯ ОЦІНКИ РЕЗУЛЬТАТІВ

8

Категорія	Метрика	Опис та призначення
Продуктивність (Performance)	Середній час відповіді	Вимірює затримку обробки запитів системою.
	Пікова затримка (95th Percentile)	Критично важлива для SLA; показує найгірші ситуації.
	Пропускна здатність	Кількість оброблених запитів/сек.
Ефективність використання ресурсів (Resource Efficiency)	Завантаженість CPU	Ступінь використання обчислювальних ресурсів.
	Завантаженість RAM	Визначає, наскільки ефективно використовується пам'ять.
	Кількість активних інстансів	Відображає рівень масштабування.
	Частота масштабувань	Показує стабільність роботи алгоритму.
Економічні показники (Cost Efficiency)	Вартість використання	Загальна вартість оренди хмарних ресурсів.
	Вартість на 1 запит	Дає змогу оцінити економічну ефективність.
Якість обслуговування (QoS/SLA)	Порушення SLA	Кількість випадків, коли час відповіді перевищив поріг.
	Успішність обробки	Частка запитів, виконаних без помилок.
Адаптивність алгоритму (Adaptivity)	Швидкість реакції	Час, за який алгоритм змінює конфігурацію після зміни навантаження.

ІНСТРУМЕНТИ ДЛЯ РЕАЛІЗАЦІЇ

9

python™

Matplotlib

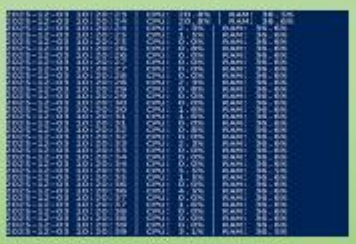
NumPy

pandas

ЗБІР МЕТРИК ТА СИМУЛЯТОР ХМАРНОГО СЕРЕДОВИЩА

10

Для моделювання навантаження були використані реальні вимірювання CPU та RAM, отримані зі середовища Windows 10 під час виконання типових задач користувача.
Дані збиралися протягом 90 секунд з інтервалом 0.5 секунди



Для відтворення поведінки хмарної інфраструктури створено симулятор, який використовує реальні метрики, що дозволяє забезпечити правдоподібність моделі.

11

СЦЕНАРІЙ «А» ТА СЦЕНАРІЙ «Б»

Статичний сценарій працює з фіксованою кількістю інстансів.

Алгоритм роботи:

1. На кожному кроці симуляції отримуються реальні метрики CPU.
2. Кількість інстансів залишається константою (2 інстанси).
3. Обчислюється latency, cost та reward.
4. Всі результати накопичуються до завершення епізоду;

Результати сценарію «А»

Показник	Значення
Total reward	–189.15
Avg latency	55.23 мс
Avg cost	0.05
Avg instances	1.00

Евристичний сценарій працює за правилом:

- якщо CPU > 70% → збільшити інстанси;
- якщо CPU < 30% → зменшити інстанси;
- інакше → залишити кількість незмінною.

Алгоритм роботи:

1. На кожному кроці аналізується поточний CPU.
2. Вибирається дія (вгору / вниз / нічого).
3. Кількість інстансів змінюється адаптивно.
4. Обчислюється latency та cost.
5. Накопичується reward.

Результати сценарію «Б»

Показник	Значення
Total reward	–189.15
Avg latency	55.23 мс
Avg cost	0.05
Avg instances	1.00

12

СЦЕНАРІЙ «В» ТА СЦЕНАРІЙ «Г»

Генетичний алгоритм дозволяє знайти оптимальні пороги масштабування та початкову кількість інстансів на основі: кросинговеру, мутацій, селекції найкращих особин, оцінки fitness-функції через симулятор.

Алгоритм роботи:

1. Кодування рішення, де хромосома – це варіант розподілу VM по вузлах;
2. Формується початкова популяція – генерується кілька випадкових конфігурацій розміщення;
3. Для кожної конфігурації обчислюється функція пристосованості:
$$Q = w_1P + w_2A + w_3S - w_4C - w_5E$$
4. Селекція – краще рішення з більшою вірогідністю перейдуть далі;
5. Кросовер і мутація
6. Результат – генетичний алгоритм знаходить глобально оптимальну конфігурацію;

RL-агент навчається через взаємодію зі середовищем: на кожному кроці він: отримує стан, виконує дію, отримує винагороду, оновлює Q-таблицю.

Алгоритм роботи:

1. Агент зчитує поточні метрики, і відповідно до них збільшує, зменшує чи залишає конфігурацію без змін;
2. Агент оцінює наслідки свого рішення: зміни продуктивності, витрат та чи дотримнао SLA;
3. Агент коригує свою поведінку;
4. Процес «спостереження → дія → оцінка → навчання» повторюється багато разів
5. Формується стабільна політика управління
6. Результат - застосовує набуту політику для реагування на зміни навантаження в реальному часі.;

СЦЕНАРІЙ Д

13

Гібридний підхід поєднує сильні сторони GA та Q-learning:

- GA формує «добру стартову політику» через оптимальні пороги;
- RL навчається адаптувати поведінку під динамічні умови.

Алгоритм роботи:

1. Генетичний алгоритм знаходить оптимальну стартову конфігурацію;
2. Генетичний алгоритм формує початкову політику;
3. RL-агент використовує вже сформовану політику;
4. RL-агент коригує політику в реальному часі;
5. Результати роботи RL-агента можуть періодично корегувати генетичний алгоритм або оновлювати вагові коефіцієнти.

```
Найкраща стратегія з GA:
  up_threshold      = 76.14
  down_threshold    = 34.07
  initial_instances = 1
  fitness           = -1074.94

=== ЕТАП 2: Створення RL-агента та ініціалізація з GA-стратегії ===
```

Результат формування стартової політики

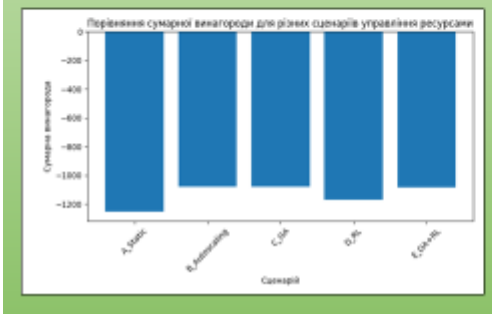
```
=== ПІДСУМКИ ГІБРИДНОГО ПІДХОДУ GA+RL ===
GA-стратегія, з якої стартував RL:
  up_threshold      = 76.14
  down_threshold    = 34.07
  initial_instances = 1

Сумарна винагорода гібридного агента при оцінці: -1083.23
Кількість кроків в епізоді: 179
```

ПОРІВНЯННЯ ТА АНАЛІЗ

14

Сценарій	Загальна винагорода	avg_latency	avg_cost	avg_cpu	avg_instances	sla_violations	Кроків
Сценарій А	-1255.17	50.12	0.09	1.11	2.0	0	179
Сценарій Б	-1081.45	50.41	0.04	1.11	1.0	0	179
Сценарій В	-1081.14	50.39	0.04	1.11	1.0	0	179
Сценарій Г	-1169.38	50.41	0.07	1.11	1.49	0	179
Сценарій Д	-1085.80	50.54	0.05	1.11	1.01	0	179



Сценарій	Переваги	Недоліки
А	простота	немає адаптації
Б	низька вартість	низька адаптивність
В	найкращий загальний результат	не реагує на динаміку
Г	найнижча латентність	найдорожчий
Д	адаптивність + швидкість реакцій	потребує довшого навчання

ВИСНОВКИ

- Розглянуто основні сучасні хмарні технології, їх структуру та осоливості.
- Виконано математичне модулювання процесів управління ресурсами.
- Розглянуто архітектуру системи керування ресурсами.
- Розроблено та реалізовано гібридний алгоритм GA+RL, який поєднує сильні сторони еволюційних методів та підкріплювального навчання.
- Отримані результати доводять доцільність використання гібридних інтелектуальних методів для задач управління ресурсами в хмарних системах, особливо в умовах динамічної зміни навантаження.

АПРОБАЦІЯ

1. Носов Є.Д., Шушура О.М. Методи управління ресурсами інформаційних систем у хмарних середовищах. Матеріали III Всеукраїнської науково-технічної конференції «Технологічні горизонти: дослідження та застосування інформаційних технологій для технологічного прогресу України і світу», 18 листопада 2025 року.

2. Носов Є.Д., Шушура О. М., Соломаха С.А., Інтелектуальні методи управління ресурсами інформаційних систем у хмарних технологіях. Матеріали VIII Всеукраїнської науково-технічної конференції «Комп'ютерні технології: інновації, проблеми, рішення», м. Житомир, 02-03 грудня 2025 року

Дякую за увагу!