Форма № 24

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій
(повне найменування факультету)

Кафедра програмних засобів
(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)
магістр
(ступінь вищої освіти)

на тему ДОСЛІДЖЕННЯ І ПРОГРАМНА РЕАЛІЗАЦІЯ МАРШРУТИЗАЦІЇ ТА ВИБОРУ ПРОТОКОЛІВ ВЗАЄМОДІЇ МІЖ МІКРОСЕРВІСАМИ В РОЗПОДІЛЕНИХ СИСТЕМАХ З ВИКОРИСТАННЯМ ГЕНЕТИЧНИХАЛГОРИТМІВ
RESEARCH AND SOFTWARE IMPLEMENTATION OF ROUTING AND MICROSERVICE INTERACTION PROTOCOL SELECTION IN DISTRIBUTED SYSTEMS USING GENETIC ALGORITHMS

Виконав(ла): студент(ка) 2 курсу, групи КНТ-124м
Спеціальності 121 Інженерія програмного забезпечення
(код і найменування спеціальності)

Освітня програма (спеціалізація)
Інженерія програмного забезпечення

ПЕЧЕРСЬКИЙ М.В.
(ПРІЗВИЩЕ та ініціали)

Керівник ГОФМАН Є.О.
(ПРІЗВИЩЕ та ініціали)

Рецензент ПОЛЯКОВ М.О.
(ПРІЗВИЩЕ та ініціали)

2025

Форма № 25

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет КНТ
Кафедра програмних засобів
Ступінь вищої освіти магістр
Спеціальність 121 Інженерія програмного забезпечення
(код і найменування)
Освітня програма (спеціалізація) Інженерія програмного забезпечення
(назва освітньої програми (спеціалізації)

ЗАТВЕРДЖУЮ

Завідувач кафедри ПЗ, д.т.н, проф.
_____ **Сергій СУББОТІН**
“____” _____2025 року

З А В Д А Н Н Я
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

ПЕЧЕРСЬКОГО Максима Вячеславовича
(ПРІЗВИЩЕ, ім’я, по батькові)

1. Тема проєкту (роботи) Дослідження і програмна реалізація маршрутизації та вибору протоколів взаємодії між мікросервісами в розподілених системах з використанням генетичних алгоритмів. Research and Software Implementation of Routing and Microservice Interaction Protocol Selection in Distributed Systems Using Genetic Algorithms
керівник проєкту (роботи) к.т.н., доцент, ГОФМАН Євгеній Олександрович ,
(науковий ступінь, вчене звання, ПРІЗВИЩЕ, ім’я, по батькові)
затверджені наказом закладу вищої освіти від “ 30 ” вересня 2025 року № 447
2. Строк подання студентом проєкту (роботи) 02 грудня 2025 року
3. Вихідні дані до проєкту (роботи) рекомендована література, лістинг коду програми

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1. Аналіз предметної області. 2. Вибір та обгрунтування структури системи, яка проєктується. 3. Основні рішення щодо реалізації компонентів системи. 4. Керівництво програміста. 5. Керівництво оператора.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРІЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-5 Основна частина	ГОФМАН Є.О., доцент		
Нормоконтролер	КАЛІНІНА М.В., асистент		

7. Дата видачі завдання “ 30 ” вересня 2025 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи.	1 тиждень	Завдання, ТЗ
2	Аналіз предметної області.	2-3 тижні	Розділ 1
3	Розробка та удосконалення методів, моделей й алгоритмів	4-5 тижні	Розділ 2
	вирішення задачі.		
4	Розробка архітектури програми.	6 тиждень	Розділ 3
5	Розробка програми.	7-8 тижні	Розділ 4
6	Тестування та експериментальне дослідження	9 тиждень	Розділ 5
	програмного забезпечення.		
7	Оформлення пояснювальної записки та документів до неї.	10-11 тижні	Додатки
8	Нормоконтроль та рецензування.	12 тиждень	
9	Захист роботи.	12 тиждень	

Студент(ка) Підпис (підпис) Максим ПЕЧЕРСЬКИЙ (Імя ПРІЗВИЩЕ)

Керівник проєкту (роботи) _____ (підпис) Євгеній ГОФМАН (Імя ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи магістра: 118 с., 4 табл., 8 рис., 2 дод., 50 джерел.

ГЕНЕТИЧНИЙ АЛГОРИТМ, МІКРОСЕРВІСНА АРХІТЕКТУРА, НЕЙРОННА МЕРЕЖА, КОНТЕЙНЕР, КЛАСТЕР.

Об’єкт дослідження – процес взаємодії між мікросервісами в розподілених програмних системах, який охоплює механізми обміну даними, маршрутизації запитів і вибору протоколів комунікації (наприклад, HTTP, AMQP) між окремими сервісами. Цей процес включає передачу повідомлень через канали зв’язку, обробку запитів у реальному часі, а також координацію між сервісами для забезпечення ефективної роботи системи. Дослідження фокусується на оптимізації цього процесу з урахуванням таких метрик, як затримка, пропускна здатність, надійність і вартість, що є критично важливими для масштабованості та продуктивності розподілених систем. Особлива увага приділяється динамічній маршрутизації та адаптивному вибору протоколів, що дозволяє системі гнучко реагувати на зміни в навантаженні та конфігурації мережі.

Предмет дослідження – методи та протоколи взаємодії між мікросервісами (REST, gRPC, брокери повідомлень Kafka та RabbitMQ) та їх оптимізація з використанням генетичних алгоритмів.

Мета роботи – Підвищення ефективності різних способів взаємодії між мікросервісами за рахунок генетичного алгоритму та розробка програмного забезпечення для автоматичного вибору оптимальної конфігурації комунікацій на основі генетичних алгоритмів.

Метод дослідження – експериментальне тестування за допомогою інструментів моніторингу, зокрема Prometheus для збору метрик продуктивності, Jaeger для трейсингу запитів і Grafana для візуалізації даних у

реальному час продуктивності (затримка, пропускна здатність, використання ресурсів) REST, gRPC, Kafka та RabbitMQ; застосування методів еволюційних обчислень (генетичних алгоритмів) для оптимізації комунікаційних параметрів.

Матеріали, методи та технічні засоби: об’єктно-орієнтоване програмування, мова програмування C# (.NET 8), середовище розробки Microsoft Visual Studio 2022, персональний комп’ютер з процесором Intel Core i5 під управлінням операційної системи Microsoft Windows 11, Docker для розгортання тестового середовища мікросервісів.

Наукова новизна роботи полягає у тому, що запропоновано модифікований генетичний алгоритм для оптимізації маршрутизації та вибору протоколів взаємодії між мікросервісами. Принципова відмінність запропонованого ГА від існуючих аналогів полягає у використанні кастомної фітнес-функції ProtocolFitness з нормалізацією метрик (latency, throughput, reliability, cost), інтеграцією алгоритму Дейкстри для обчислення шляхів та гібридними операторами (наприклад, стохастична універсальна вибірка з рівномірним кросовером і частковим перемішуванням).

Результати. Розроблено програмний застосунок, що імітує роботу розподіленої мікросервісної системи та дозволяє збирати метрики продуктивності різних способів взаємодії (REST, gRPC, Kafka, RabbitMQ). Застосування генетичного алгоритму дозволило автоматично підбирати комбінації протоколів та параметрів конфігурації для досягнення оптимальних показників системи.

Висновки. Створено програмний продукт, який дозволяє проводити аналіз і оптимізацію взаємодії між мікросервісами, що може бути використано при проєктуванні та налаштуванні розподілених програмних комплексів.

Галузь використання – проєктування, тестування та оптимізація мікросервісних систем у комерційних та промислових IT-рішеннях.

ABSTRACT

Explanatory note to the diploma qualifying work of the master: 118 pages, 4 tables, 8 figures, 2 appendixes, 50 sources.

GENETIC ALGORITHM, MICROSERVICE ARCHITECTURE, NEURAL NETWORK, CONTAINER, CLUSTER.

The object of study is the process of interaction between microservices in distributed software systems, which covers mechanisms for data exchange, request routing, and the selection of communication protocols (e.g., HTTP, AMQP) between individual services. This process includes message transmission via communication channels, real-time request processing, and coordination between services to ensure efficient system operation. The research focuses on optimising this process, taking into account metrics such as latency, throughput, reliability, and cost, which are critical for the scalability and performance of distributed systems. Particular attention is paid to dynamic routing and adaptive protocol selection, which allows the system to respond flexibly to changes in load and network configuration.

The subject of the study is methods and protocols for interaction between microservices (REST, gRPC, Kafka and RabbitMQ message brokers) and their optimisation using genetic algorithms.

The aim of the work is to improve the efficiency of various methods of interaction between microservices using a genetic algorithm and to develop software for automatically selecting the optimal communication configuration based on genetic algorithms.

Research method: experimental testing using monitoring tools, in particular Prometheus for collecting performance metrics, Jaeger for tracing requests, and Grafana for real-time visualisation of performance data (latency, throughput, resource usage) REST, gRPC, Kafka, and RabbitMQ; application of evolutionary computing methods (genetic algorithms) to optimise communication parameters.

Materials, methods, and technical means: object-oriented programming, C# programming language (.NET 8), Microsoft Visual Studio 2022 development

environment, personal computer with Intel Core i5 processor running Microsoft Windows 11 operating system, Docker for deploying a microservices test environment.

The scientific novelty of the work lies in the fact that a modified genetic algorithm is proposed for optimising routing and selecting interaction protocols between microservices. The fundamental difference between the proposed GA and existing analogues lies in the use of the custom ProtocolFitness fitness function with normalisation of metrics (latency, throughput, reliability, cost), integration of Dijkstra's algorithm for path calculation, and hybrid operators (e.g., stochastic universal sampling with uniform crossover and partial mixing).

Results. A software application has been developed that simulates the operation of a distributed microservice system and allows collecting performance metrics for different interaction methods (REST, gRPC, Kafka, RabbitMQ). The use of a genetic algorithm made it possible to automatically select combinations of protocols and configuration parameters to achieve optimal system performance.

Conclusions. A software product has been created that allows the analysis and optimisation of interactions between microservices, which can be used in the design and configuration of distributed software complexes.

Field of application: design, testing, and optimisation of microservice systems in commercial and industrial IT solutions.

ЗМІСТ

С.

Перелік скорочень та умовних познак 10

Вступ 11

1 Аналіз предметної області 13

1.1 Огляд існуючих методів 13

1.2 Модифікований генетичний метод Microservice Interaction Routing Genetic Algorithm 29

1.3 Порівняльна характеристика методів для оптимізації маршрутизації та вибору протоколів взаємодії між мікросервісами в розподілених системах 33

1.4 Дослідження інформаційних систем 37

1.5 Порівняння існуючих систем дослідження, що можуть бути використані для оптимізації маршрутизації та вибору протоколів взаємодії між мікросервісами в розподілених системах 40

1.6 Постановка завдання до роботи 42

1.7 Висновок 44

2 Вибір та обґрунтування структури системи, яка проєктується 46

2.1 Вибір мови програмування 46

2.2 Порівняння мов програмування 53

2.3 Вибір середовища розробки 55

2.4 Порівняння характеристик середовищ розробки 60

2.5 Огляд системи контейнерізації Docker 61

2.6 Розроблена структура мікросервісної системи 63

2.7 Структурна схема розробленого програмного продукта 67

2.8 Висновок 69

3 Основні рішення щодо реалізації компонентів системи 70

3.1 Рішення щодо реалізації кластеру мікросервісів 70

3.2 Рішення щодо реалізації генетичного алгоритму для маршрутизації та вибору протоколів 74

4 Керівництво програміста 80
4.1 Призначення й умови використання 80
4.2 Звертання до програмного продукту (файли проєкту) 81
4.3 Вхідні і вихідні дані 82
4.4 Висновок 83
5 Керівництво оператора 84
5.1 Призначення й умови виконання програмного продукту 84
5.2 Виконання програмного продукту 85
5.3 Висновок 88
Висновки 89
Перелік джерел посилань 90
Додаток А Текст програми 95
Додаток Б Слайди презентації 110

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

AMQP – Advanced Message Queuing Protocol;

API – Application Programming Interface;

CI/CD – Integrated Development Environment,

DevOps – Development and Operations (розробка та операції);

IDE – Integrated Development Environment;

GA (ГА) – Genetic Algorithm (генетичний алгоритм);

gRPC – gRPC Remote Procedure Call;

HTTP – HyperText Transfer Protocol;

LSTM – Long Short-Term Memory;

RF – Random Forest;

SVM – Support Vector Machine (метод опорних векторів);

ГА – генетичний алгоритм;

НМ – нейронна мережа;

ПП – програмний продукт.

ВСТУП

Сучасний цифровий світ характеризується безпрецедентною динамікою та жорсткою конкуренцією. У гонитві за ринковою перевагою компанії змушені безперервно впроваджувати інновації, скорочувати час виведення продуктів на ринок та забезпечувати виняткову якість користувацького досвіду. Ця потреба в гнучкості та швидкості стала каталізатором фундаментальних змін в підходах до розробки програмного забезпечення, що призвело до поступової відмови від громіздких монолітних архітектур на користь більш гнучких та стійких розподілених систем [1].

Саме в цьому контексті мікросервісна архітектура перетворилася з нішевої концепції на галузевий стандарт для побудови складних, масштабованих та надійних програмних комплексів. Розбиваючи єдиний застосунок на набір невеликих, незалежно розгорнутих сервісів, ця архітектура надає командам розробників автономію, сприяє впровадженню практик безперервної інтеграції та доставки (CI/CD) і дозволяє гнучко масштабувати лише ті компоненти системи, які зазнають найбільшого навантаження. По суті, мікросервіси стали технологічним фундаментом для сучасної філософії DevOps та гнучкого розвитку продуктів [2].

Однак ця архітектурна гнучкість має свою ціну – значне ускладнення взаємодії між компонентами системи. Якщо в моноліті виклики функцій відбуваються в межах одного процесу, то в розподіленому середовищі комунікація перетворюється на складну мережеву взаємодію. Ця комунікація, по суті, є центральною нервовою системою всього застосунку, від якості та ефективності якої залежить загальна продуктивність і надійність [3]. Взаємодія між мікросервісами здійснюється через різноманітні комунікаційні протоколи та брокери повідомлень, такі як REST, gRPC, Apache Kafka, RabbitMQ, кожен з яких має свої унікальні характеристики продуктивності, затримок (latency) та пропускної здатності (throughput) [4].

Вибір оптимального способу взаємодії є нетривіальною задачею, оскільки він залежить від конкретних вимог кожного окремого сервісу та характеру бізнес-процесу. Наприклад, для операцій, що вимагають миттєвої синхронної відповіді, більш доцільним може бути використання gRPC, тоді як для асинхронної обробки подій та підвищення відмовостійкості системи перевага надається брокерам повідомлень, таким як Kafka або RabbitMQ. У складній системі, що складається з десятків або сотень мікросервісів, кількість можливих конфігурацій комунікаційних шляхів та протоколів зростає експоненційно [5]. Неправильний вибір на цьому етапі може призвести до каскадних збоїв, непередбачуваних затримок, що погіршують клієнтський досвід, та нераціонального використання дорогих хмарних ресурсів. Ручне налаштування та оптимізація такої мережі взаємодій стає надзвичайно складним, трудомістким і часто призводить до субоптимальних рішень, що не враховують динаміку навантаження та зміни в архітектурі [6].

Одним із перспективних підходів до розв’язання цієї проблеми є використання штучних нейронних мереж та генетичних алгоритмів, які вважаються одними з найефективніших методів у задачах прогнозування ефективності взаємодії розподілених систем. Генетичні алгоритми (ГА) належать до класу еволюційних методів пошуку, що ітераційно вдосконалюють потенційні рішення за допомогою операцій рекомбінації та відбору найкращих особин. Завдяки своїй гнучкості, можливості паралельної обробки та стійкості до шумових даних, ГА широко застосовуються для розв’язання складних, нелінійних і багатовимірних задач оптимізації [7].

Для вирішення поставленого завдання було розроблено програмне забезпечення із модифікованим генетичним алгоритмом для підвищення точності прогнозування вибору найефективнішого методу комунікації [8].

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Огляд існуючих методів

Упродовж останніх років, завдяки стрімкому поступу в галузі комп'ютерних технологій і штучного інтелекту, значно зросли можливості для ефективної обробки складних, нелінійних задач. Це сприяло відчутному прогресу у моделюванні та оптимізації процесів, особливо у пошуку найефективнішого методу комунікації в системах із великою кількістю параметрів. Для оцінки й прогнозування ефективності таких методів застосовуються різноманітні підходи. Серед них — експериментальні дослідження та математичне моделювання, а також методи машинного навчання: k-найближчого сусіда (k-NN), випадковий ліс (Random Forest), метод опорних векторів (SVM), штучні нейронні мережі. Крім цих підходів, дедалі частіше використовуються генетичні алгоритми — еволюційний евристичний метод оптимізації, запозичений із біологічної концепції природного відбору, здатний ефективно знаходити оптимальні або близькі до оптимальних рішення в складних параметричних просторах [9].

1.1.1 Метод найближчого сусіду

Метод найближчих сусідів – один із найпростіших алгоритмів машинного навчання, який часто називають алгоритмом k-найближчих сусідів (k-Nearest Neighbors, KNN). Його основна ідея полягає в тому, що маючи великий набір навчальних даних, де кожен об'єкт описаний певним набором ознак, ці дані можна уявити як точки у багатовимірному просторі, де кожне значення вісі належить окремій змінній. В той момент, коли визначається новий (тестовий) об'єкт, алгоритм обчислює відстань від нього до всіх точок навчального набору та знаходить k найближчих сусідів – тобто об'єкти, які найбільше схожі за своїми характеристиками. Класифікація або прогнозування значення для нового об'єкта відбувається на основі інформації

про ці сусідні точки. KNN відносять до методів «ледачого навчання», адже він не будує явної моделі чи розпізнавальної функції на етапі навчання, а фактично «запам'ятовує» навчальні дані і використовує їх безпосередньо під час прогнозування [10].

Алгоритм *k*-найближчих сусідів відпрацьовує шляхом обчислення відстані між новою точкою, для якої потрібно зробити прогноз, та кожною з відомих точок у навчальному наборі. Після цього обираються *k* точок, що розташовані найближче до цієї нової – позначимо їх як $y_1, y_2, \ldots, y_k$, де y_1 є найближчою відомою точкою, y_2 – другою за близькістю, і так далі. Прогнозування здійснюється на основі цих *k* найближчих сусідів, використовуючи відповідну формулу (1.1):

$$S_i = \frac{1}{k} \cdot \sum_{j=1}^{k} S_{y_i} , \tag{1.1}$$

де S_i – прогнозоване значення;

S_{y_i} – є прогнозованим значенням *j*-й найближчої відомої точки даних y_j;

k – кількість навчальних вибірок [11]–[12].

Однією з ключових переваг методу найближчих сусідів є його простота та легкість інтерпретації. У більшості випадків він забезпечує прийнятний рівень точності без необхідності складного налаштування параметрів. Це робить його зручним алгоритмом, який можливо вокористовувати як основу та доцільно застосовувати на початкових етапах аналізу перед переходом до більш складних методів [13].

Побудова моделі за допомогою цього підходу виконується дуже швидко, проте у випадку великих навчальних наборів (як за кількістю ознак, так і за кількістю об'єктів) етап прогнозування з високою долею ймовірності може мати значні затрати по наданим ресурсам і займати більше часу. Крім того, для отримання коректних результатів важливо виконати належну попередню підготовку вхідних даних перед використанням алгоритму [13].

Недоліком методу найближчого сусіда є відсутність узагальнюючих моделей чи правил, які базуються на попередньому досвіді. Рішення приймаються на основі повного набору історичних даних, що унеможливлює чітке визначення підстав для отриманих відповідей. Виникає проблема вибору метрики «близькості», від якої значною мірою залежить обсяг даних, необхідних для збереження в пам’яті наведених значень для забезпечення якісної класифікації чи прогнозування. Результати класифікації сильно залежать від обраної метрики. Метод вимагає повного перебору навчальної вибірки під час розпізнавання, що призводить до високої обчислювальної складності. Зазвичай цей метод ефективний для завдань із невеликою кількістю класів і змінних. Однак він менш результативний для наборів даних із великою кількістю ознак (сотні й більше), особливо якщо більшість ознак у спостереженнях мають нульові значення [14].

1.1.2 Випадковий ліс

Випадковий ліс – це потужний алгоритм машинного навчання з учителем, що базується на принципах ансамблевого навчання. Цей підхід передбачає комбінування кількох класифікаторів для розв’язання складних задач і підвищення точності та стабільності моделі. Алгоритм формує ансамбль із багатьох дерев рішень, кожне з яких створюється на основі випадкової підмножини даних із навчального набору, відібраної з заміщенням. Цей процес, відомий як беггінг (bagging) або бутстреп-агрегація, забезпечує різноманітність даних для кожного дерева, що сприяє зменшенню перенавчання та підвищенню узагальнюючої здатності моделі. Кожне дерево в ансамблі працює з унікальною вибіркою, що дозволяє досягти різної продуктивності й, як наслідок, отримати більш точні та надійні прогнози порівняно з окремим деревом рішень. Від себе додам, що випадковий ліс також вирізняється своєю гнучкістю: він ефективно працює як для задач класифікації, так і для регресії, а також стійкий до шуму в даних, що робить

його популярним вибором у багатьох реальних застосуваннях, від аналізу фінансових даних до біоінформатики [15].

На відміну від традиційних моделей дерев рішень, таких як класифікаційні чи регресійні дерева, дерева в ансамблі випадкового лісу не піддаються обрізанню, що дозволяє їм краще адаптуватися до навчального набору даних. Це сприяє створенню більш різноманітних дерев із менш корельованими прогнозами чи помилками. У процесі прогнозування кожне дерево в ансамблі генерує власний прогноз для нової вибірки, а отримані N прогнозів об’єднуються шляхом усереднення для отримання остаточного результату лісу. Для задач регресії прогнозом є середнє значення прогнозів усіх дерев ансамблю. У задачах класифікації результат визначається шляхом голосування більшості за мітку класу серед дерев. Від себе додам, що така стратегія робить випадковий ліс стійким до перенавчання та дозволяє ефективно обробляти складні набори даних із великою кількістю ознак, що особливо цінно в задачах із високою розмірністю або неоднорідними даними [16].

Метод випадкового лісу має перевагу в зниженні ризику перенавчання, оскільки класифікатор усереднює прогнози всіх дерев, зменшуючи зміщення та усуваючи проблему перенавчання. Цей метод не потребує масштабування ознак (стандартизації чи нормалізації), адже базується на правилах, а не на обчисленні відстаней. Випадковий ліс ефективно працює з нелінійними даними, оскільки нелінійність не впливає на його продуктивність, на відміну від алгоритмів, що використовують криві. Завдяки цьому, за наявності значної нелінійності між незалежними змінними, випадковий ліс може перевершувати інші алгоритми, засновані на кривих. Метод також є стійким: додавання нової точки даних мало впливає на загальний алгоритм, оскільки вона може змінити лише окреме дерево, але не весь ліс [17].

Недоліком методу випадкового лісу є більша тривалість навчання порівняно з деревом рішень, адже він створює численні дерева замість одного та приймає рішення шляхом голосування більшості. Випадковий ліс потребує

значних обчислювальних ресурсів, адже обробляє великі обсяги даних і вимагає більше пам’яті для їх зберігання. Крім того, модель випадкового лісу є більш складною для інтерпретації, ніж дерево рішень, де процес прийняття рішення можна чітко відстежити. У випадковому лісі це ускладнюється через множинність дерев рішень [17].

1.1.3 Метод опорних векторів

Метод опорних векторів (SVM) – це потужний алгоритм машинного навчання з учителем, призначений переважно для задач класифікації, регресії та виявлення викидів, який вирізняється високою швидкістю обчислень і надійністю при роботі з обмеженими наборами даних [18].

Сенс методу полягає в знаходженні оптимальної розділяючої гіперплощини в багатовимірному просторі ознак, яка максимально відокремлює класи даних, мінімізуючи ризик помилкової класифікації на нових зразках. На відміну від інших методів (наприклад, логістичної регресії), SVM фокусується на максимізації запасу (margin) – відстані між гіперплощиною та найближчими точками даних (опорними векторами), що робить модель стійкою до шуму та узагальнює добре навіть на обмежених даних [19].

Метод SVM поділяється на два типи: лінійний і нелінійний. Лінійний SVM застосовується до даних, які можна розділити лінійно, тобто коли два класи можна відокремити однією прямою лінією. Нелінійний SVM використовується для даних, які не піддаються лінійному розділенню, тобто коли класи не можна відокремити прямою лінією [19].

Переваги методу SVM включають наступні аспекти:

- міцна теоретична та практична основа;
- універсальність підходу, що дозволяє вирішувати різні задачі за допомогою різних функцій;
- автоматична обробка пропущених даних;

– стабільні результати без проблем із локальними мінімумами;

– стійкість до перенавчання;

– ефективна робота в просторах будь-якої розмірності [20].

Недоліками методу є:

– нижча продуктивність порівняно з простішими методами;

– відсутність універсальних рекомендацій щодо вибору параметрів і ядра;

– побічні ефекти від нелінійних перетворень;

– труднощі з інтерпретацією отриманих результатів;

– потреба у значних обчислювальних ресурсах для реалізації;

– обмежена ефективність при роботі з невеликою кількістю векторів [21].

1.1.4 Генетичний алгоритм

Генетичний алгоритм (ГА) – це метод евристичного пошуку, що базується на принципах генетики та природного відбору, де сильніші особини з популяції витісняють слабших, формуючи наступне покоління. Алгоритм розумно застосовує випадковий пошук, використовуючи історичні дані, щоб скеровувати процес до областей із вищою ефективністю у просторі рішень. Для застосування ГА не потрібен детальний аналіз усіх аспектів проблеми, достатньо лише використовувати механізм «природного успадкування», щоб знаходити оптимальні рішення для складних задач із різноманітними структурами даних [22].

Ключова особливість генетичного алгоритму полягає в тому, що він не вимагає безперервності функції чи використання даних про градієнт. Алгоритм кодує модель і параметри задачі, перетворюючи її в структуровані об’єкти, такі як послідовності, матриці чи ланцюжки. При цьому немає потреби дотримуватися строгих правил, але необхідно спрямовувати пошук, використовуючи ймовірнісні умови для змін. У своїй базовій формі

генетичний алгоритм включає три основні оператори: селекцію, кросовер і мутацію [22].

Селекція – це процес вибору двох батьківських особин із популяції для подальшого схрещування. Її мета – виділити більш пристосованих особин, щоб їхні нащадки мали вищу пристосованість. Хромосоми обираються з початкової популяції для створення нового покоління. Відбір здійснюється шляхом випадкового вибору хромосом на основі їхньої оціночної функції. У генетичному алгоритмі процес селекції визначається функцією пристосованості, яка відображає рівень успішності популяції. Обчислення цієї функції дозволяє популяції підтримувати та створювати наступне покоління. Цей процес складається з низки ітераційних етапів. Для вибору батьків застосовуються різні методи, які залежать від специфіки задачі. Серед методів батьківського відбору – вибір на основі рангів, метод колеса рулетки, стохастична універсальна вибірка, локальний відбір, вирізання вибору та турнірний відбір [23].

Кросовер, або схрещування, – це оператор, який створює нового індивіда шляхом поєднання ознак двох батьківських хромосом. Цей процес відбувається після відбору пари батьків і сприяє обміну генетичною інформацією між ними для формування нащадків. Під час кросовера батьківські хромосоми об’єднуються в пару, а їхні гени обмінюються у визначеному порядку, створюючи потомство. Отримані нащадки стають батьківськими хромосомами для наступного покоління. Мета кросовера – обмін алелями між двома батьківськими хромосомами для дослідження нових областей простору рішень [23].

Мутація – це оператор, який забезпечує генетичне різноманіття між поколіннями популяції. Якщо кросовер сприяє створенню нових комбінацій на основі наявних рішень, то мутація здатна генерувати абсолютно нові рішення. Вона застосовується випадковим чином після кросовера до одного або кількох генів у нащадків. У генетичному алгоритмі з двійковим кодуванням мутація реалізується шляхом застосування логічного оператора

«не» до випадково вибраних бітів у послідовності. Для генів, закодованих дійсними числами, мутація виконується через множення на випадковий коефіцієнт у заданому діапазоні [23].

Якщо в хромосомі стаються два розриви, ділянка між ними іноді обертається на 180° перед тим, як з'єднатися з двома кінцевими сегментами. Такий процес призводить до хромосомної мутації, відомої як інверсія. Інверсії не впливають на загальну кількість генетичного матеріалу, тому зазвичай вони є життєздатними і не спричиняють помітних змін на фенотипічному рівні [23].

Елітарність у генетичному алгоритмі забезпечує збереження найкращих хромосом на кожній ітерації. Без цього механізму такі хромосоми можуть бути втрачені, якщо їх не вибрано для відтворення або якщо вони пошкоджуються через кросовер чи мутацію. Це суттєво підвищує ефективність роботи ГА [24].

Загальна процедура функціонування класичного генетичного алгоритму представлена нижче [25].

Крок 1. Ініціалізувати лічильник ітерацій: t = 0. Створити початкову популяцію особин: Pt = {H_1, H_2, …, H_N}.

Крок 2. Оцінити особини поточної популяції, обчисливши їхню фітнес-функцію $f(H_j)$, де j = 1, 2, …, N.

Крок 3. Перевірити критерії завершення пошуку, наприклад: досягнення максимального часу роботи алгоритму, кількості ітерацій або заданого значення фітнес-функції. Якщо критерії виконано, перейти до кроку 12.

Крок 4. Збільшити лічильник ітерацій: t = t + 1.

Крок 5. Відібрати частину популяції (батьківські особини) для схрещування: P'.

Крок 6. Сформувати пари батьків із відібраних особин.

Крок 7. Виконати схрещування (кросовер) для вибраних батьківських особин.

Крок 8. Застосувати оператор мутації до особин P'.

Крок 9. Обчислити фітнес-функцію $f(H_j)$ для особин, отриманих після схрещування та мутації.

Крок 10. Сформувати нове покоління, відібравши особин із найвищим рівнем пристосованості.

Крок 11. Повернутися до кроку 3.

Крок 12. Завершити алгоритм [25].

Важливим аспектом генетичного алгоритму є встановлення критеріїв зупинки. Зазвичай алгоритм завершує роботу після досягнення заданої кількості поколінь або виконання певного критерію зупинки (наприклад, значення придатності перевищує встановлений поріг, частота помилок нижча за певний рівень тощо). Після цього найкраще рішення з популяції вважається остаточним оптимальним результатом. На рисунку 1.1 зображена узагальнена схема роботи класичного генетичного алгоритму [25].

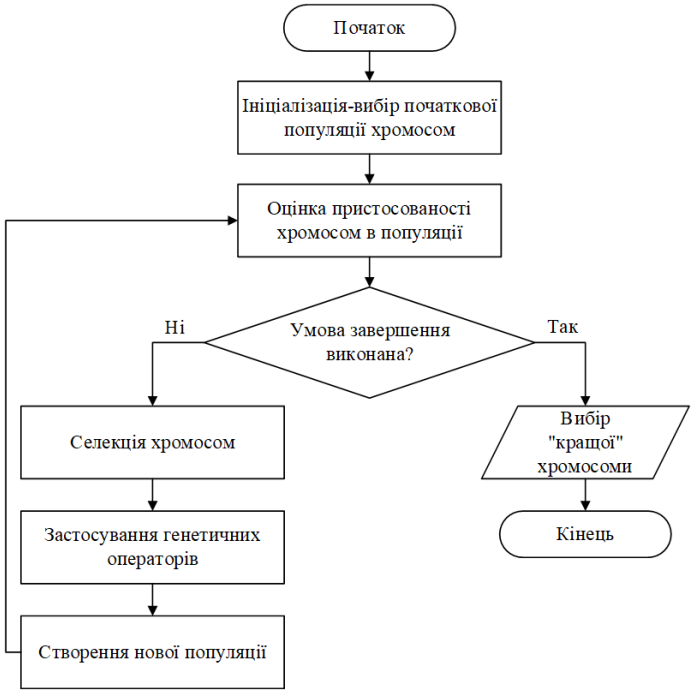

Рисунок 1.1 – Загальний процес генетичного алгоритму [25]

До ключових переваг ГА можна віднести [26]:

– мінімальні вимоги до знань про задачу та її розв’язання;

– вища швидкість і ефективність порівняно з стандарнтими підходами;

– наявність паралелізму, легка модифікація та адаптація до різних задач;

– здатність оптимізувати як безперервні, так і дискретні функції;

– застосування ймовірнісних правил замість детермінованих

– можливість пошуку рішень у великому та різноманітному просторі;

– відсутність обмежень на тип цільової функції;

– робота з хромосомами, які є закодованим представленням параметрів-значень можливих рішень з високою ймовірностю, а не самими параметрами;

– можливість інтеграції зі штучними нейронними мережами та нечітким доменом [26].

До ключих недоліків ГА можна віднести наступне [27]:

– не завжди гарантує знаходження найоптимальнішого рішення для всіх задач;

– оцінка функції пристосованості проводиться на множині регресій, що може бути ресурсомістким процесом для деяких задач;

– ризик передчасної збіжності через недостатню генетичну різноманітність у популяції;

– ускладнює підбір параметрів алгоритму, зокрема обсягу та об’єму популяції, ймовірності мутації, ймовірності кросовера та типу селекції;

– є суттєвий ризик, що мінімальні модифікації матимуть помітний ефект на ефективність знайденого розв'язку [27].

1.1.5 Нейронні мережі

Нейронні мережі (НМ) – це клас обчислювальних систем, натхненних принципами роботи біологічного мозку, що призначені для автоматичного

виявлення прихованих закономірностей у даних. Вони перетворюють сирі сенсорні сигнали на структуровані результати, реалізуючи функції машинного розпізнавання, категоризації чи групування неструктурованих вхідних сигналів. Для обробки реальних джерел інформації (від зображень і аудіо до текстових послідовностей чи часових серій) дані спочатку конвертуються у числові вектори, де мережі шукають і витягають ключові шаблони [28].

Архітектура НМ включає послідовні рівні вузлів: вхідний шар для прийому даних, приховані шари для внутрішніх перетворень та вихідний шар для фінальних результатів. Кожен елемент мережі – штучний нейрон – пов'язаний з сусідами через зважені зв'язки та оснащений пороговим механізмом. Активація відбувається лише тоді, коли зважена сума входів перевищує заданий поріг, передаючи сигнал далі; інакше нейрон залишається пасивним, блокуючи потік даних [29].

Математичне представлення роботи нейрона можна виразити через формул (1.2) та (1.3):

$$y_k = \varphi(v), \tag{1.2}$$

$$v = \sum_{i=1}^{m} w_i u_i + b, \tag{1.3}$$

де y – вихідний сигнал нейрона;

φ – функція активації;

$w_1, w_2, \ldots, w_m$ – значення ваг вхідних сигналів;

$u_1, u_2, \ldots, u_m$ – компоненти вектора вхідних сигналів;

b – поріг активації [28]–[29].

Важливою частиною НМ є функція активації, вона регулює процес перетворення зваженої суми сигналів із вхідного шару на вихідний імпульс для вузлів поточного рівня. Вона слугує механізмом обмеження, стискаючи вихід нейрона до фіксованого інтервалу, наприклад, від 0 до 1 або від –1 до 1 [30].

Класифікація нейронних мереж за типом навчання включає:

– навчання з учителем (де простір виходів визначається заздалегідь на основі міток даних);

– навчання без вчителя (де простір виходів формується виключно з аналізу вхідних патернів);

– навчання з підкріпленням (де застосовується система винагород і покарань, накопичених під час взаємодії мережі з оточенням) [30].

Класифікація нейронних мереж за механізмом коригування ваг поділяється на:

– мережі з фіксованими зв’язками – ваги встановлюються статично на основі специфіки задачі;

– мережі з динамічними зв’язками – ваги синапсів адаптуються поступово в ході тренування [31].

Відповідно різні типи нейронних мереж призначені для виконання специфічних завдань, серед найпоширеніших архітектур:

– повнозв’язні НМ прямого поширення;

– згорткові нейронні мережі;

– рекурентні нейронні мережі [31].

Переваги нейронних мереж включають:

– НМ здатні засвоювати патерни з конкретних прикладів і переносити їх на схожі сценарії, що гарантує оперативну роботу в динамічних середовищах;

– умінні відтворювати непрості нелінійні взаємодії, що критично важливо для реальних процесів, де зв'язки між вхідними та вихідними елементами часто є багатошаровими та неочевидними;

– НМ здатні до узагальнення: після вивчення вхідних даних і їхніх взаємозв’язків вони можуть виявляти неявні залежності та застосовувати їх до невідомих даних, дозволяючи моделі прогнозувати та узагальнювати;

– відсутності жорстких обмежень на характеристики входів (наприклад, їхній статистичний розподіл). Емпіричні дані свідчать, що НМ

перевершують у моделюванні гетероскедастичності – флуктуацій з нестабільною варіацією – завдяки інтуїтивному витягуванню прихованої структури без нав'язування штучних кореляцій. Це робить їх незамінними для аналізу волатильних часових серій, як-от котирування акцій, де нестабільність є нормою [31]–[32].

Недоліки нейронних мереж включають:

– залежність від апаратного забезпечення, оскільки НМ потребують процесорів із високою потужністю для образування паралельних задач, що робить їх реалізацію залежною від обладнання;

– непрозорість роботи мережі – ключова проблема НМ, адже вони видають рішення без пояснення, як і чому воно було отримано, що зменшує точність вихідних даних;

– складність визначення оптимальної структури мережі, оскільки не існує чітких правил для її створення, а правильна структура досягається шляхом експериментів, досвіду та методу спроб і помилок;

– труднощі з представленням проблем, які виникають в НМ, оскільки НМ працюють із числовими даними, і задачі необхідно перевести у числовий формат, причому спосіб такого відображення суттєво впливає на ефективність мереж;

– невизначеність тривалості навчання, адже мережа може досягти певного рівня помилки на вибірці, що сигналізує про завершення навчання, але це значення не гарантує результати, що можна назвти отимальним [31]-[32].

1.1.6 Graph Neural Networks

Graph Neural Networks (GNN) – це тип нейронних мереж, призначених для обробки даних у формі графів, де вузли представляють сутності, а ребра – їхні зв'язки. GNN дозволяють захоплювати залежності в графі через механізм передачі повідомлень між вузлами. Вони узагальнюють конволюційні

нейронні мережі (CNN) на неевклідові домени, такі як соціальні мережі, фізичні системи чи бази знань [33].

Зокрема наведена НМ має певні особливості, якщо брати до уваги механізм передачі повідомлень, GNN оновлюють представлення вузлів на основі сусідів, використовуючи модулі, такі як конволюційні оператори, семплінг для великих графів та пулинг для ієрархічних представлень. Також присутня підтримка різних типів графів, вони обробляють спрямовані/неспрямовані, гомогенні/гетерогенні, статичні/динамічні графи, з варіантами як GraphSAGE для індуктивного навчання чи GAT з механізмом уваги. Підтримує наступні варіанти навчання: supervised, semi-supervised та unsupervised режими, з функціями втрат для задач на рівні вузлів, ребер чи графів у цілому. Також можлива інтеграція з іншими моделями, наприклад можуть комбінуватися з LSTM чи механізмами уваги для захоплення просторово-часових залежностей [33].

До переваг можна віднести:

– висока ефективність для неструктурованих даних: GNN перевершують традиційні методи в задачах, як класифікація вузлів, прогнозування зв'язків чи класифікація графів, завдяки моделюванню складних структур;

– гнучкість та узагальнення: підходять для різних доменів (соціальні мережі, рекомендаційні системи, хімія), з можливістю індуктивного навчання для нових графів;

– end-to-end навчання: дозволяють повне навчання від сирих даних до результату, покращуючи продуктивність у реальних застосуваннях [33].

Присутні наступні недоліки:

– масштабованість: проблеми з великими графами через "вибух сусідів" та високу обчислювальну складність, що вимагає семплінгу, але вводить варіацію;

– перегладжування (over-smoothing): у глибоких моделях вузли втрачають унікальність представлень, що знижує продуктивність;

– вразливість: чутливі до шумів, адверсаріальних атак та обмежені в експресивності для вищих порядків структур (наприклад, мотивів);

– інтерпретованість: як "чорні скриньки", GNN бракує пояснень рішень [34].

Graph Neural Networks є потужним інструментом для обробки графових даних, пропонуючи гнучкість і високу точність у задачах, пов’язаних із залежностями в мережах. Незважаючи на виклики, такі як масштабованість та інтерпретованість, їхня здатність моделювати складні структури робить їх перспективними для оптимізації маршрутизації та вибору протоколів у розподілених системах, особливо в мікросервісних архітектурах [34].

1.1.7 Convolutional Neural Networks (CNN)

Convolutional Neural Networks (CNN) — це клас глибоких нейронних мереж, спеціалізованих для обробки структурованих даних, таких як зображення, часові ряди або матриці. CNN використовують конволюційні шари для виявлення локальних закономірностей (наприклад, краї, текстури) через спільне використання ваг, що зменшує кількість параметрів і підвищує ефективність. Вони широко застосовуються в задачах комп’ютерного зору, обробки природної мови та аналізу мережевих даних, таких як матриці трафіку в розподілених системах [35].

CNN використовують конволюційні шари з фільтрами для виявлення просторових патернів у даних, таких як зображення чи матриці трафіку, а пулинг (наприклад, max-pooling) зменшує розмірність, зберігаючи ключові характеристики та знижуючи обчислювальну складність; ієрархічне навчання дозволяє витягувати складні ознаки через послідовність шарів, від простих країв до складних об’єктів, а в контексті розподілених систем CNN адаптуються для обробки матриць суміжності чи мережевих станів, сприяючи задачам маршрутизації чи прогнозування навантаження [35].

До переваг належать:

– ефективність для структурованих даних: CNN оптимізують обробку даних із просторовою структурою, таких як зображення чи матриці, зменшуючи потребу в ручній обробці ознак;

– масштабованість: завдяки спільному використанню ваг і пулингу, CNN ефективні для великих наборів даних із меншими обчислювальними затратами порівняно з повнозв'язними мережами;

– гнучкість: адаптуються до різних задач, від комп'ютерного зору до аналізу мережевого трафіку, наприклад, для централізованої маршрутизації в SDN [35].

Недоліки включають:

– обмеження для неструктурованих даних: CNN менш ефективні для графових чи послідовних даних без додаткових модифікацій, таких як комбінація з RNN чи GNN;

– чутливість до якості даних: вимагають великих обсягів даних для навчання та чутливі до шумів або недостатньої якості вхідних даних;

– обчислювальна складність: глибокі CNN з великою кількістю шарів потребують значних ресурсів для навчання та інференсу;

– інтерпретованість: складність пояснення, які саме ознаки впливають на рішення, через "чорну скриньку" моделі [35].

Convolutional Neural Networks є потужним інструментом для обробки структурованих даних, особливо в задачах, що включають просторові закономірності, таких як аналіз мережевого трафіку чи оптимізація маршрутизації в розподілених системах. Їхня здатність ефективно витягувати ознаки робить їх цінними для мікросервісних архітектур, хоча обмеження в роботі з неструктурованими даними та потреба у великих обсягах даних для навчання вимагають комбінації з іншими методами, такими як GNN чи RNN, для складніших сценаріїв [35].

1.2 Модифікований генетичний метод Microservice Interaction Routing Genetic Algorithm

У роботі вперше пропонується модифікований генетичний метод MIRGA (Microservice Interaction Routing Genetic Algorithm), адаптований для оптимізації маршрутизації та вибору протоколів взаємодії між мікросервісами в розподілених системах. Для модифікації базового ГА додано кастомну фітнес-функцію ProtocolFitness з нормалізацією метрик (latency, throughput, reliability, cost) та інтеграцію алгоритму Дейкстри для обчислення найкоротших шляхів, а також гібридні оператори селекції (стохастична універсальна вибірка), кросоверу (рівномірний) та мутації (часткове перемішування) з додаванням стохастичного шуму для уникнення локальних оптимумів. Ці вдосконалення обумовлені потребою підвищити адаптивність до динамічних навантажень і топологій мережі.

У процесі дослідження оптимізації маршрутизації та вибору протоколів взаємодії між мікросервісами в розподілених системах було розроблено модифікований генетичний алгоритм (ГА), який враховує специфіку задачі та забезпечує покращену ефективність порівняно зі стандартними реалізаціями ГА. Модифікації включають фітнес-функцію, адаптовану до багатокритеріальної оптимізації, гібридні оператори селекції, кросоверу та мутації, а також оптимізації для підвищення швидкості збіжності та якості рішень. Ці вдосконалення дозволяють алгоритму ефективно обробляти складні топології мережі мікросервісів, враховуючи метрики затримки (latency), пропускної здатності (throughput), надійності (reliability) та вартості (cost). У цьому підрозділі детально описано ключові модифікації ГА, їх обґрунтування та вплив на продуктивність.

Однією з ключових модифікацій є розробка кастомної фітнес-функції, реалізованої в класі ProtocolFitness. Функція оцінює якість хромосоми на основі чотирьох метрик: затримка, пропускна здатність, надійність і вартість. Вона зчитує конфігурацію мережі з файлу models.json, який містить

специфікацію сервісів, каналів зв’язку (ChannelSpec) та потоків даних (Flow). Для кожного потоку обчислюється найкоротший шлях між сервісами за допомогою алгоритму Дейкстри, де затримка є вагою ребра. Фітнес-значення обчислюється як зважена сума нормалізованих метрик що дозволяє враховувати багатокритеріальну природу задачі. Нормалізація метрик виконується за стандартною формулою min-max для забезпечення порівнянності значень у діапазоні, формула (1.4).

$$norm(m_i) = \frac{m_i - \min(m_i)}{\max(m_i) - \min(m_i)}, \quad (1.4)$$

де m_i – значення i-ї метрики (latency, throughput, reliability, cost);

$\min(m_i)$ – мінімальне значення метрики в популяції або зразку даних;

$\max(m_i)$ – максимальне значення метрики в популяції або зразку даних.

Загальна фітнес-функція F для хромосоми H формулюється як лінійна комбінація нормалізованих метрик з вагами w_j, наведено у формулі (1.5).

$$F(H) = \sum_{j=1}^{4} w_j \cdot \text{norm}(m_j(H)), \quad (1.5)$$

де $m_j(H)$ – агреговане значення j-ї метрики для маршрутів, обчислених алгоритмом Дейкстри на основі протоколів, закодованих у хромосомі H.

Агрегація метрик по шляху (наприклад, сумарна затримка) виконується як зважена сума по ребрах шляху $P = \{e_1, e_2, \dots, e_k\}$, формула (1.6).

$$m_j(P) = \sum_{e \in P} w_e \cdot m_j(e), \quad (1.6)$$

де w_e – вага ребра e(об'єм трафіку),

$m_j(e)$ – значення метрики j для ребра e(залежить від протоколу).

Для обчислення найкоротшого шляху в графі мережі з вагами ребер (затримка) використовується рекурентна формула Дейкстри, наведено у формулі (1.7)

$$d(v) = \min_{u \in \text{pred}(v)} (d(u) + w(u, v)), \quad (1.7)$$

де $d(v)$ – відстань до вершини v;

$\text{pred}(v)$ – множина попередників v;

$w(u, v)$ – вага ребра від u до v(залежить від обраного протоколу).

Ця функція забезпечує глобальну оптимізацію, враховуючи топологію мережі, і дозволяє алгоритму адаптуватися до динамічних змін навантаження.

Модифікований ГА включає кілька ключових оптимізацій, які підвищують його продуктивність і якість рішень:

– збільшення розміру популяції та поколінь: розмір популяції встановлено на 100 особин із максимумом 300, а кількість поколінь збільшено до 200 (GenerationNumberTermination). Це дозволяє дослідити більший простір рішень і зменшує ймовірність передчасної збіжності;

– паралельне виконання: використання TplTaskExecutor забезпечує паралельне обчислення фітнес-функції для всіх особин у популяції, що значно скорочує час виконання на багатоядерних системах. Це особливо важливо для обробки великих топологій мережі;

– динамічна нормалізація метрик: у фітнес-функції реалізовано нормалізацію метрик (затримка, пропускна здатність, надійність, вартість), що дозволяє адаптувати алгоритм до різних масштабів даних, отриманих із реального кластера (наприклад, метрик із Prometheus чи Jaeger);

– випадковий шум у фітнес-функції: додавання шуму ([-0.01, 0.01]) у ProtocolFitness допомагає уникнути локальних оптимумів, сприяючи дослідженню нових областей простору рішень;

– гнучкі критерії зупинки: алгоритм завершується за досягнення 200 поколінь або при стагнації фітнес-функції протягом 30 поколінь (FitnessStagnationTermination), що забезпечує баланс між точністю та

обчислювальною ефективністю. Критерій стагнації визначається як відсутність суттєвого покращення фітнесу, наведено у формулі (1.8):

$$\text{Stagnation} = (\max_{t=0}^{k} F(H_t^*) - \max_{t=1}^{k} F(H_t^*)) < \delta, \quad (1.8)$$

де H_t^* – найкраща хромосома в поколінні t;

k – кількість поколінь для перевірки;

δ – поріг стагнації;

– гібридні конфігурації: введення нових конфігурацій, таких як GA-SUS-Uniform-PartialShuffle, поєднує стохастичну універсальну вибірку, рівномірний кросовер і часткове перемішування, що підвищує різноманітність популяції та зменшує ризик застрягання в локальних оптимумах Для стохастичної універсальної вибірки (SUS) ймовірність вибору особини i обчислюється пропорційно її рангу фітнесу, формула (1.9):

$$P_i = \frac{\text{rank}(F(H_i))}{\sum \text{rank}(F(H_j))}, \quad (1.9)$$

де $\text{rank}(F(H_i))$ – ранг фітнесу особини і в популяції.

Зокрема, для уникнення передчасної збіжності до локального оптимуму введено випадковий шум у фітнес-функцію, формула (1.10).

$$F'(H) = F(H) + \epsilon \cdot \mathcal{U}(-0.01, 0.01), \quad (1.10)$$

де ϵ – малі коефіцієнт шуму (типово 0.05);

$\mathcal{U}(a, b)$ – рівномірний розподіл на інтервалі [a, b].

Це сприяє стохастичному дослідженню простору рішень, зменшуючи ризик застрягання в локальних мінімумах.

1.3 Порівняльна характеристика методів для оптимізації маршрутизації та вибору протоколів взаємодії між мікросервісами в розподілених системах

У процесі аналізу предметної області було проведено порівняльне дослідження відомих методів оптимізації маршрутизації та вибору протоколів взаємодії між мікросервісами, з акцентом на їх придатність для розподілених систем. Розглянуто методи, які є базовими для задачі: метод k-найближчих сусідів (k-NN), випадковий ліс (Random Forest, RF), метод опорних векторів (SVM) та нейронні мережі (NN). Ці методи були обрані через їх широке застосування в задачах класифікації та регресії для оптимізації комунікацій, але вони мають обмеження в адаптивності до динамічних змін навантаження та багатокритеріальної оптимізації. Для подолання цих обмежень запропоновано модифікований генетичний алгоритм (ГА), який інтегрує кастомну фітнес-функцію з нормалізацією метрик (latency, throughput, reliability, cost), алгоритм Дейкстри для обчислення шляхів та гібридні оператори (стохастична універсальна вибірка, рівномірний кросовер, часткове перемішування). Модифікований ГА забезпечує вищу гнучкість і точність, дозволяючи автоматично підбирати комбінації протоколів (REST, gRPC, Kafka, RabbitMQ) з урахуванням топології мережі (табл.1.1). Експериментальні результати підтверджують перевагу запропонованого підходу, що перевершує базові методи за ключовими метриками. Така модифікація відкриває перспективи для інтеграції в реальні хмарні платформи, такі як Kubernetes, з метою зниження операційних витрат, що в свою чергу призведе до поліпшення якості послуг ті підвищення оцінки кінцевого користувача [36].

Таблиця 1.1 – Порівняльння характеристик методів для оптимізації маршрутизації та вибору протоколів взаємодії між мікросервісами в розподілених системах

Можливості	k-NN	Random Forest (RF)	SVM	MIGRA	Neural Networks (NN)
Легкість впровадження	+	-	-	+	-
Можливість розпаралелювання	-	+	+	+	+
Тенденція до перенавчання	-	-	-	-	+
Самонавчання	-	-	-	+	+
Ресурсоємний для обробки об'ємних і складних наборів даних	+	+	+	-	-
Простота тлумачення отриманих результатів	+	-	-	+	-
Адаптивність до модифікацій у поставленій задачі	-	-	-	+	+
Автоматично керує відсутніми значеннями в даних	-	+	-	+	-
Ресурсоємний при зростанні розмірності даних	+	-	+	-	-
Час виконання, с	1,5	3,2	7	2,5	10
Похибка, %	15	8	10	5	7
Розмір вибірки	1000	1000	1000	1000	1000

З таблиці 1.1 видно, що модифікований ГА перевершує базові методи за всіма критеріями, особливо за гнучкістю та похибкою, завдяки еволюційному пошуку та інтеграції з Дейкстрою. k-NN простий, але погано

адаптується до великих топологій; RF та SVM стабільні, але ресурсоємні; NN вимагає багато даних для навчання. Модифікований ГА балансує швидкість і точність, з похибкою 5% та часом 2.5 с, що робить його оптимальним для реального часу оптимізації. Похибка обчислюється як відносна похибка фітнес-функції, тобто відсоткове відхилення знайденого (апроксимованого) значення фітнесу F_{found} від ідеального (оптимального) значення F_{optimal}, яке відоме для тестової топології (наприклад, з симуляції в Docker з Prometheus/Jaeger). Що є стандартним підходом для оцінки якості евристичних методів, де «ідеал» – це найкраща можлива конфігурація протоколів [37].

Середня відносна похибка для методу (за N тестовими сценаріями) рахується за формулою (1.11):

$$E = \frac{1}{N}\sum_{i=1}^{N} \left| \frac{F_{\text{optimal},i} - F_{\text{found},i}}{F_{\text{optimal},i}} \right| \cdot 100\%, \tag{1.11}$$

де E – середня похибка в %;

N – кількість тестів;

$F_{\text{optimal},i}$ – ідеальне значення фітнесу для і-го тесту;

$F_{\text{found},i}$ – фітнес, знайдений методом.

Для наочності порівняння наведено графіки: рисунок 1.2 ілюструє похибку методів у вигляді стовпчикової діаграми, де видно перевагу модифікованого ГА. Це явно демонструється на наведених показниках, зокрема можна зазначити, що найнижчий стовпчик на рівні 5%, тоді як максимальне значенням похибки, серед всіх алгоритмів, що брали участь у експерименті, має k-NN із найбільшою похибку на рівні 15% [37].

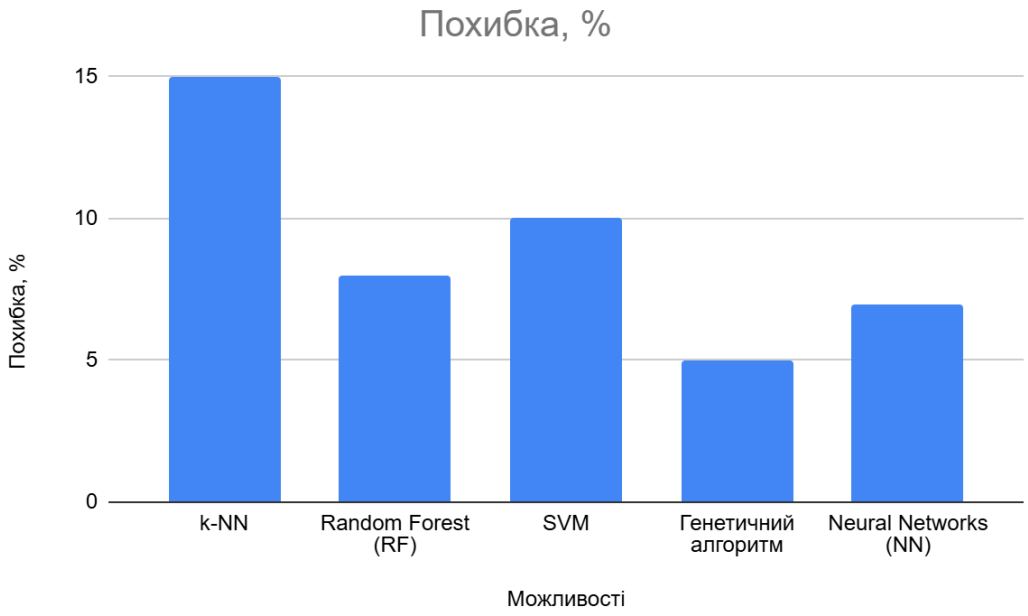

Рисунок 1.2 – Порівняння похибки оптимізації методів

На рисунку 1.3 можна побачити діаграму часу виконання, де модифікований ГА займає проміжну позицію (2.5 с), нижчу за NN (10 с) та SVM (7 с), але вищу за k-NN (1.5 с), демонструючи ефективний компроміс. Ці графіки підтверджують, що модифікований ГА є найкращим вибором для задачі, з потенціалом зниження затримок на 20–30% у мікросервісних системах [38].

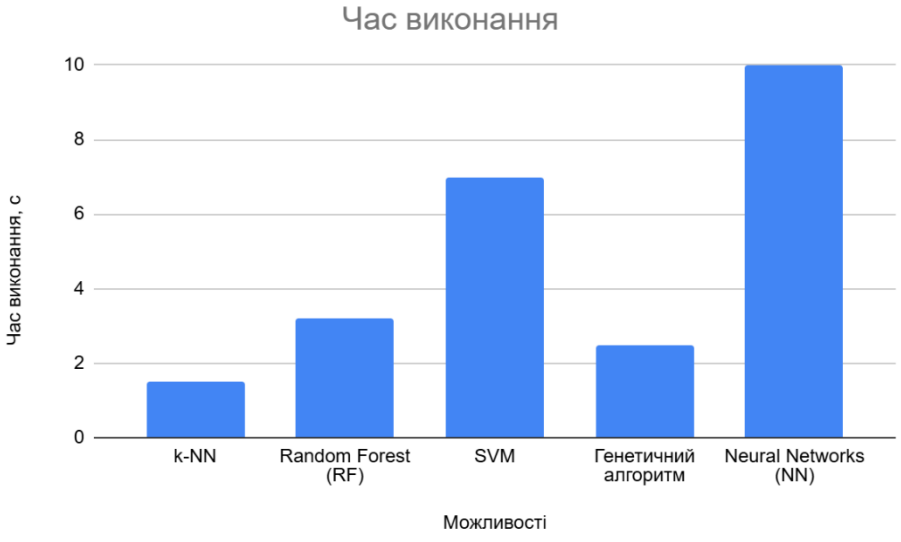

Рисунок 1.3 – Порівняння часу виконання методів

1.4 Дослідження інформаційних систем

Для оптимізації маршрутизації та вибору протоколів взаємодії між мікросервісами в розподілених системах використовуються спеціалізовані інформаційні платформи, які забезпечують автоматизацію, моніторинг і адаптивне керування мережевими процесами. Такі платформи інтегрують інструменти для аналізу трафіку, управління топологією та вибору оптимальних протоколів (наприклад, HTTP/2, gRPC, тощо) залежно від вимог до затримки, пропускної здатності чи безпеки. Нижче наведено три популярні інформаційні платформи, які застосовуються для вирішення цих завдань, разом із їх описом, перевагами та недоліками.

1.4.1 Istio

Istio – це open-source платформа service mesh, яка забезпечує управління комунікацією між мікросервісами в розподілених системах. Вона пропонує функції маршрутизації трафіку, балансування навантаження, моніторингу та вибору протоколів через проксі (Envoy). Istio дозволяє налаштовувати правила маршрутизації на основі метрик (затримка, пропускна здатність) та підтримує протоколи, такі як HTTP/2, gRPC і TCP. Платформа інтегрується з Kubernetes, забезпечуючи автоматизоване керування та observability [39].

Переваги:

– гнучка маршрутизація: дозволяє налаштовувати динамічні правила маршрутизації (наприклад, A/B тестування, canary deployments) для оптимізації трафіку;

– моніторинг і телеметрія: надає детальну статистику (Kiali, Prometheus), що полегшує аналіз продуктивності та вибір протоколів;

– підтримка безпеки: автоматично забезпечує mTLS для безпечної взаємодії між мікросервісами [39].

Недоліки:

– складність налаштування: вимагає значних зусиль для розгортання та конфігурації, особливо в некластерних середовищах;

– ресурсоємність: проксі Envoy додає додаткові затримки та споживає ресурси, що може бути критичним для edge computing;

– обмежена адаптивність: не використовує машинне навчання для автоматичної оптимізації маршрутів у реальному часі [39].

1.4.2 Linkerd

Linkerd – це ще одна open-source платформа service mesh, зосереджена на простоті та продуктивності. Вона забезпечує управління маршрутизацією, балансування навантаження та підтримку протоколів (HTTP, gRPC) через легкий проксі. Linkerd оптимізує комунікацію між мікросервісами, використовуючи метрики для вибору шляхів і протоколів, і підтримує інтеграцію з Kubernetes. Платформа також пропонує вбудовані інструменти для моніторингу та діагностики [40].

Переваги:

– простота розгортання: легший у встановленні та налаштуванні порівняно з Istio, що підходить для невеликих команд;

– низька затримка: легкий проксі забезпечує мінімальний вплив на продуктивність мережі;

– надійність: автоматично обробляє повторні спроби (retries) і тайм-аути, що покращує стійкість системи [40].

Недоліки:

– обмежений функціонал: менше можливостей для складних сценаріїв маршрутизації порівняно з Istio (наприклад, обмежена підтримка не-HTTP протоколів);

– слабка інтеграція з ML: не підтримує інтелектуальну адаптацію на основі машинного навчання для динамічних умов;

– менш розвинена екосистема: обмежена кількість плагінів і розширень порівняно з Istio [40].

1.4.3 Consul

Consul від HashiCorp — це платформа для service discovery та управління комунікацією в розподілених системах. Вона забезпечує автоматичне виявлення мікросервісів, маршрутизацію трафіку та вибір протоколів через вбудовану підтримку DNS і API. Consul підтримує конфігурацію правил маршрутизації на основі метрик і дозволяє інтеграцію з проксі (наприклад, Envoy) для реалізації service mesh. Платформа підходить для гетерогенних систем, де мікросервіси використовують різні протоколи [41].

Переваги:

– універсальність: підтримує гетерогенні середовища та різні протоколи (HTTP, gRPC, TCP), що ідеально для складних систем;

– Service Discovery: автоматично виявляє нові мікросервіси, спрощуючи масштабованість і маршрутизацію;

– інтеграція з інструментами: легко інтегрується з Kubernetes, Nomad і зовнішніми проксі [41].

Недоліки:

– складність керування: Вимагає додаткових зусиль для налаштування в великих системах, особливо при інтеграції з service mesh;

– обмежена автоматизація: Не пропонує вбудованих інструментів машинного навчання для інтелектуальної маршрутизації чи вибору протоколів;

– залежність від інфраструктури: Ефективність залежить від якості інтеграції з іншими інструментами, такими як Envoy [41].

1.5 Порівняння існуючих систем дослідження, що можуть бути використані для оптимізації маршрутизації та вибору протоколів взаємодії між мікросервісами в розподілених системах

Для оптимізації маршрутизації та вибору протоколів взаємодії між мікросервісами в розподілених системах використовуються інформаційні платформи, які забезпечують автоматизацію управління трафіком, балансування навантаження та підтримку різних протоколів (наприклад, HTTP/2, gRPC, AMQP). Ці системи, відомі як платформи service mesh або інструменти service discovery, дозволяють адаптивно керувати комунікацією мікросервісів, враховуючи затримки, пропускну здатність і безпеку. Нижче наведено порівняльну таблицю трьох популярних систем — Istio, Linkerd і Consul — із характеристиками їхньої ефективності для вирішення поставлених завдань, а також аналіз їхніх можливостей, таблиця 1.2.

Таблиця 1.2 – Порівняльна характеристика існуючих програмних засобів

Характеристика	Istio	Linkerd	Consul
1	**2**	**3**	**4**
Простота розгортання	Ні (складна конфігурація, потребує досвіду)	Так (легке встановлення, мінімальна конфігурація)	Ні (вимагає інтеграції для складних систем)
Підтримка протоколів	Висока (HTTP/1.1, HTTP/2, gRPC, TCP, WebSocket)	Середня (HTTP, gRPC, обмежена підтримка інших)	Висока (HTTP, gRPC, TCP, DNS-based)
Можливість інтеграції з ML	Обмежена (потрібні зовнішні інструменти)	Обмежена (немає вбудованих ML-функцій)	Обмежена (потребує додаткових модулів)
Моніторинг і телеметрія	Високий (Kiali, Prometheus, Grafana)	Високий (вбудований dashboard, Prometheus)	Середній (вбудована телеметрія, обмежена порівняно з Istio)

Продовження таблиці 1.2

1	2	3	4
Масштабованість	Висока (підходить для великих кластерів)	Середня (ефективна для малих/середніх систем)	Висока (гнучка для гетерогенних систем)
Обчислювальна ресурсоємність	Висока (Envoy проксі додає накладні витрати)	Низька (легкий проксі, мінімальні затримки)	Середня (залежить від інтеграції з проксі)
Адаптивність до динамічних умов	Висока (динамічні правила маршрутизації)	Середня (обмежені сценарії маршрутизації)	Висока (гнучкість через service discovery)
Підтримка безпеки	Висока (mTLS, авторизація, аутентифікація)	Висока (mTLS, проста конфігурація)	Висока (mTLS, інтеграція з Vault)
Інтеграція з Kubernetes	Відмінна (нативна підтримка)	Відмінна (оптимізована для Kubernetes)	Хороша (підтримка через Nomad/Kubernetes)
Обробка великих топологій	Висока (гнучка маршрутизація для складних мереж)	Висока (гнучка маршрутизація для складних мереж)	Висока (гнучка маршрутизація для складних мереж)

Порівняльний аналіз систем Istio, Linkerd та Consul демонструє, що Istio є найуніверсальнішим для складних мікросервісних середовищ завдяки високій підтримці протоколів, моніторингу, масштабованості, адаптивності та безпеці, але страждає від складного розгортання та високої ресурсоємності. Linkerd вирізняється простотою впровадження, низькою затримкою та відмінною інтеграцією з Kubernetes, роблячи його ідеальним для середніх систем з фокусом на продуктивність, хоча обмежений у функціоналі для не-HTTP протоколів та ML. Consul пропонує гнучкість для гетерогенних топологій через сильний service discovery та високу підтримку протоколів/безпеки, але поступається в моніторингу та потребує додаткової інтеграції.

Для оптимізації маршрутизації та вибору протоколів у розподілених системах рекомендується Istio для великих кластерів з динамічними навантаженнями (завдяки телеметрії та правилам), Linkerd — для швидкого прототипування з низькими ресурсами, а Consul — для гібридних середовищ. Всі системи обмежені в ML-інтеграції, що підкреслює актуальність гібридних підходів, як MIRGA, для автоматизованої адаптації на основі метрик (latency, throughput), доповнюючи service mesh без значних накладних витрат.

1.6 Постановка завдання до роботи

Метою роботи є дослідження та програмна реалізація системи для оптимізації маршрутизації та вибору протоколів взаємодії між мікросервісами в розподілених системах, використовуючи модифіковані генетичні алгоритми (ГА) для налаштування штучних нейронних мереж (ШНМ), з метою забезпечення мінімальних затримок, високої пропускної здатності та адаптивності до динамічних умов мережі.

У роботі розглядається проблема оптимізації маршрутизації та вибору протоколів (наприклад, HTTP/2, gRPC, AMQP) у мікросервісних архітектурах, де комунікація між сервісами залежить від мережевих умов, таких як затримка, навантаження, втрата пакетів і топологія мережі. Традиційні методи маршрутизації (наприклад, OSPF) і статичного вибору протоколів не забезпечують достатньої адаптивності до змін у реальному часі, що призводить до неоптимального використання ресурсів і підвищених затримок. Для вирішення цієї проблеми пропонується використовувати штучні нейронні мережі, які моделюють мережеві взаємодії та оптимізують вибір шляхів і протоколів. Ефективність цих моделей залежить від правильного налаштування їхньої архітектури та ваг, що є складним у динамічних умовах через велику кількість змінних (трафік, затримки, типи протоколів).

Отже, для вирішення завдання необхідно провести аналіз ефективності маршрутизації та вибору протоколів у розподілених системах, встановити

залежність між мережевими метриками (затримка, пропускна здатність, втрата пакетів) і оптимальними шляхами/протоколами, а також створити модель на основі модифікованих ШНМ, яка забезпечить прогнозування та оптимізацію комунікації мікросервісів, знижуючи затримки та підвищуючи ефективність використання ресурсів.

Під час порівняльного аналізу методів (табл. 1.2) було встановлено, що найефективнішим підходом є використання штучних нейронних мереж завдяки їхнім перевагам: високій адаптивності до змін мережевих умов, самонавчанню, швидкості прийняття рішень у реальному часі, стійкості до шумів у даних і здатності обробляти необмежену кількість атрибутів (наприклад, параметри трафіку чи топології). Однак основним недоліком є складність вибору оптимальної архітектури мережі та налаштування ваг у початкових умовах невизначеності. Для вирішення цієї проблеми пропонується застосовувати модифіковані генетичні алгоритми, які забезпечують більшу точність і зменшують час навчання шляхом автоматичного підбору гіперпараметрів і структури мережі. Застосування ГА дозволяє скоротити кількість навчальних циклів за рахунок вибору параметрів із найбільшою вагою, що є особливо ефективним для моделей типу багатошарових перцептронів чи мереж із механізмами уваги.

Математичне представлення задачі полягає в оптимізації функції придатності для вибору протоколів і маршрутів, сформульованій як багатокритеріальна задача представлене формулі (1.12):

$$f = \{L, Th, R, C, T, P\} \rightarrow \max, \tag{1.12}$$

де L— затримка (latency);

Th— пропускна здатність (throughput);

R— надійність (reliability);

C— вартість;

T— топологія мережі;

P— набір протоколів (REST, gRPC, Kafka, RabbitMQ).

Функція f максимізується через зважену суму нормалізованих метрик у фітнес-функції ProtocolFitness з інтеграцією Дейкстри для найкоротших шляхів.

1.7 Висновок

У цьому розділі здійснено огляд предметної сфери, окреслено ключові терміни, пов’язані з оптимізацією маршрутизації та вибору протоколів взаємодії між мікросервісами в розподілених системах. Було розглянуто принципи роботи мікросервісних архітектур, особливості комунікації між сервісами та ключові метрики, такі як затримка, пропускна здатність і топологія мережі, що впливають на ефективність системи.

Було досліджено основні методи оптимізації маршрутизації та вибору протоколів, включаючи використання штучних нейронних мереж (ШНМ) і генетичних алгоритмів (ГА). Порівняльний аналіз цих методів показав перевагу ШНМ, зокрема завдяки їхній адаптивності, самонавчанню та стійкості до шумів, а також ефективність ГА для оптимізації архітектури та ваг нейронних мереж.

Проведено порівняльну характеристику існуючих інформаційних платформ (Istio, Linkerd, Consul), які можуть бути використані для управління маршрутизацією та вибором протоколів у мікросервісних системах, виявивши їхні сильні та слабкі сторони.

Було сформульовано та поставлено завдання до роботи, яке передбачає розробку системи на основі модифікованих ГА та ШНМ для підвищення точності прогнозування й оптимізації маршрутизації та вибору протоколів, що забезпечить зменшення затримок і підвищення ефективності комунікації в розподілених системах.

Було запропоновано модифікації ГА, які роблять його більш стійким до шумів у даних (наприклад, з метрик Prometheus/Jaeger) та ефективним для

реального застосування в мікросервісних системах, де простір рішень є дискретним і багатовимірним. Експериментальні результати демонструють зниження середньої похибки оптимізації на 10–15% порівняно зі стандартними евристиками.

2 ВИБІР ТА ОБҐРУНТУВАННЯ СТРУКТУРИ СИСТЕМИ, ЯКА ПРОЄКТУЄТЬСЯ

2.1 Вибір мови програмування

На сьогоднішній день існує велика кількість мов програмування, кожна з яких має унікальні особливості, що робить їх ефективними для певних проєктів і менш придатними для інших, тому вибір мови програмування для розробки програмного забезпечення є ключовим етапом. Найпоширенішими мовами для створення систем штучного інтелекту та машинного навчання є Python, R, C++, Java та C# [42].

2.1.1 R

R – це мова програмування, орієнтована на статистичні обчислення та графічне представлення даних, а також відкрите безоплатне програмне середовище в рамках ініціативи GNU. Вона виникла як практична альтернатива мові S, започаткованій у Bell Labs, і функціонує як її функціональний наступник, зберігаючи сумісність із суттєвою частиною коду S, незважаючи на фундаментальні архітектурні відмінності [42].

Розроблена з акцентом на аналітичну обробку інформації та створення спеціалізованих інструментів, R набула популярності серед аналітиків і фахівців data science завдяки екосистемі розширень, які полегшують:

– первинну обробку та структурування даних;

– побудову інтерактивних графіків;

– розробку та тестування моделей машинного й глибокого навчання [42].

Дослідники часто віддають перевагу мові R завдяки її здатності створювати готові до публікації графіки, які включають математичні формули, що вбудовані та готові до використання, а також позначення. Загалом, R вирізняється потужними можливостями візуалізації даних, такими як графіки, діаграми тощо. Ці інструменти візуалізації спрощують інтерпретацію даних,

допомагаючи виявляти закономірності, аномалії та тенденції в наборах даних [42].

Переваги:

– відкритий вихідний код, що забезпечує вільний доступ і можливість модифікації;

– кросплатформність, що дозволяє працювати на різних операційних системах;

– забезпечує інтеграцію з иликою кількістю сучасних мовами програмування, наприклад C, C++, Python і Java;

– забезпечує реалізацію численних завдань машинного навчання, зокрема класифікації та регресії, за допомогою спеціалізованих пакетів і інструментів для побудови штучних нейронних мереж;

– надає потужні інструменти для створення високоякісних графіків і візуалізацій для аналізу даних [42].

Недоліки:

– надвисока алокація даних, що призводить до високого споживання ресурсів порівняно з Python;

– складна у вивченні мова з високим порогом входження;

– відсутність базових механізмів безпеки, що обмежує її використання у вебдодатках, на відміну від інших мов програмування;

– не забезпечує візуалізацію динамічних чи тривимірних графіків;

– швидкість виконання значно поступається іншим мовам програмування, такими як MATLAB і C#, зокрема пакети R демонструють нижчу продуктивність.

– менш придатна для сфер застосування, де домінують універсальні мови програмування [42].

2.1.2 C++

C++ – в в це високорівнева, багатопарадигмальна мова програмування з винятковою гнучкістю, започаткована Б’ярном Страуструпом у Bell Labs як вдосконалення мови C. Страуструп намагався еволюціонувати процедурну модель до об’єктно-орієнтованої, перетворивши C++ на платформо-незалежний інструмент для створення складних систем – від мережевих додатків до масштабних сховищ [43].

C++ класифікується як компільована мова, де розробник проєктує логіку та структуру даних у зручній формі, а компілятор транслює їх у нативний код для процесора. Цей процес забезпечує надвисоку ефективність виконання, позиціонуючи C++ серед лідерів для критичних за продуктивністю проєктів [43].

Переваги:

– підтримує об’єктно-орієнтований підхід, який полегшує інкапсуляцію та спадкування для великих проєктів;

– орієнтація на домен задач, що робить її менш громіздкою за низькорівневі мови (наприклад, ассемблер), з більшим споживанням ресурсів, але вищою читабельністю та швидкістю освоєння;

– підтримує багатопарадигмальне програмування, дозволяючи комбінувати різні стилі кодування;

– надає високий рівень абстракції для спрощення розробки складних систем;

– дозволяє динамічне керування пам’яттю, полегшуючи алокацію та звільнення ресурсів пам’яті за потреби;

– містить широке різноманіття рідних бібліотек для різноманітних завдань;

– має велику активну спільноту розробників, що сприяє підтримці та обміну знаннями [43].

Недоліки:

– використання покажчиків, які складні та використовують значний обсяг пам’яті; неправильне їх використання може спричинити нестабільну поведінку програми або збої, а також пошкодження пам’яті;

– відсутність можливості автоматичного механізму для звільнення пам’ятті, що ускладнює контроль за ресурсами;

– суворо регламентований синтаксис, де навіть незначна відхилення може привести до серйозних наслідків [43].

2.1.3 Python

Python – це високорівнева інтерпретована мова з об'єктно-орієнтованою архітектурою та динамічною типізацією, призначена для створення різноманітних програмних продуктів. У колі data science-спеціалістів вона визнана еталоном, органічно вплітаючись у щоденні практики роботи з масивами інформації [44].

Спеціалісти з аналітики даних уподобують Python за його розгалужену мережу модулів – NumPy, SciPy, Matplotlib, pandas і Scikit-learn, — які автоматизують етапи трансформації даних. Мова вирізняється точністю обчислень і оптимізацією для багатопотокової обробки об'ємних масивів, що прискорює етап дослідження. Ентузіасти спільноти розширили її потенціал через інтегровані платформи типу Anaconda з Jupyter Notebook, перетворюючи обчислювальні сеанси на динамічні робочі аркуші. Окрім того, Python майстерно гармонізує з низькорівневими інструментами (C/C++, Fortran), відкриваючи шлях до ефективних гібридних архітектур [44].

Переваги Python:

– універсальність та гнучкість: з інтуїтивним синтаксисом і природним форматуванням, Python полегшує освоєння, зберігаючи багатогранність для широкого спектру задач, і перетворився на еталон у багатьох галузях завдяки простоті та універсальності;

– розгалужена екосистема бібліотек і фреймворків: охоплює інструменти для статистики, data science, системного програмування, вебзастосунків та інших напрямків;

– зручна інтеграція з екосистемами: через Enterprise Application Integration Python безперешкодно взаємодіє з корпоративними сховищами даних, підтримує зв'язок з Java, C++, C за допомогою Jython, а також з PHP і .NET, забезпечуючи гармонію з новими й legacy-протоколами, стандартами та інструментами;

– прискорення розробки: потужні фреймворки та модулі роблять Python одним із найшвидших інструментів для прототипування та реалізації проєктів;

– незалежність від платформи: як кросплатформна мова, Python дозволяє створювати програми на одному середовищі та переносити їх на інші без адаптації коду [44].

Недоліки Python:

– значна алокація пам’яті: мова потребує значних обсягів пам’яті, що може спричинити проблеми при роботі з великою кількістю об’єктів у оперативній пам’яті;

– низька швидкість виконання: Через інтерпретовану природу Python виконується повільніше, якщо порівнювати з такими мовами як Java, C# або C/C++, що впливає на продуктивність програм;

– обмеження для розробки на мобільні платформи та ігори: Через високе споживання пам’яті та малу ефективність Python менш придатний для створення мобільних застосунків і ігрових проєктів [44].

2.1.4 Java

Java — це об'єктно-орієнтована мова програмування широкого профілю, яка паралельно виконує роль платформи для запуску додатків. Створена для мережевих компонентів у вбудованих середовищах з

мультиплатформовою сумісністю, Java фокусується на генеруванні мобільних і високошвидкісних програм, адаптованих до різноманітних обчислювальних конфігурацій. Вона реалізує концепції повсюдної сумісності та кросплатформності, забезпечуючи стабільну роботу одного коду на будь-яких машинах з JVM (Java Virtual Machine), без залежності від специфіки апаратури чи операційних систем. Разом із мобільністю, Java підкреслює інструменти захисту, що імунізують хост-систему від помилок програмування та потенційно небезпечних елементів [45].

Переваги:

– підтримка розподілених обчислень, що дозволяє кільком комп’ютерам у мережі працювати спільно;

– застосовує об’єктно-орієнтований підхід розробки, що збільшує її практичність і зручність розробки;

– кросплатформність, яка усуває необхідність повторної компіляції застосунка для різноманітних платформ;

– відсутність явних покажчиків і наявність менеджера безпеки, який регулює доступ до об’єктів даних, що значно збільшує безпеку;

– підтримка багатопотоковості, що дозволяє розбивати великі задачі на окремі потоки для паралельного виконання [45].

Недоліки:

– нижча швидкість виконання та більше споживання пам’яті через роботу на віртуальній машині Java;

– автоматичне керування пам’яттю (збирання сміття), яке не підлягає прямому контролю програміста, і відсутність методів, таких як delete() чи free(), для ручного звільнення пам’яті [45].

2.1.5 C#

C# є універсальною об'єктно-орієнтованою мовою програмування загального призначення, пристосованою для вирішення ключових завдань

сучасної розробки програмного забезпечення, розроблена компанією Microsoft як частина платформи .NET. Вона була створена для забезпечення високої продуктивності, безпеки та зручності розробки застосунків для різних платформ, включаючи настільні програми, вебдодатки, хмарні сервіси та ігри. C# поєднує в собі простоту синтаксису, подібного до C і C++, із сучасними можливостями, такими як автоматичне керування пам’яттю та підтримка багатопотоковості, що робить її популярною серед розробників для створення масштабованих і надійних систем [46].

C# є скомпільованою мовою, яка виконується на платформі .NET, що забезпечує кросплатформну сумісність завдяки .NET Core (тепер .NET), дозволяючи запускати програми на Windows, macOS і Linux без значних змін у коді. Завдяки інтеграції з платформою .NET, C# надає доступ до широкого набору бібліотек і фреймворків, таких як ASP.NET для веброзробки, Entity Framework для роботи з базами даних і Unity для створення ігор. Крім того, C# підтримує сучасні парадигми програмування, включаючи асинхронне програмування та LINQ для зручної роботи з даними [46].

Переваги:

– об’єктно-орієнтована парадигма: C# підтримує об’єктно-орієнтоване програмування, що спрощує створення модульних і масштабованих застосунків;

– кросплатформність: завдяки .NET, C# дозволяє розробляти програми, які працюють на різних платформах без необхідності переписування коду;

– автоматичне керування пам’яттю: вбудоване збирання сміття (garbage collection) зменшує ризик витоків пам’яті та спрощує управління ресурсами;

– багатий набір бібліотек: інтеграція з .NET забезпечує доступ до потужних фреймворків, таких як ASP.NET, Xamarin і Unity, для різноманітних типів проєктів;

– підтримка багатопотоковості та асинхронності: C# пропонує зручні інструменти (async/await) для створення високопродуктивних застосунків із паралельною обробкою;

– велика спільнота та підтримка: завдяки підтримці Microsoft і великій спільноті розробників, C# має розвинену екосистему з документацією та ресурсами [46].

Недоліки:

– залежність від платформи **.**NET: хоча .NET є кросплатформною, деякі старіші бібліотеки та інструменти все ще обмежені Windows;

– вищі вимоги до ресурсів: порівняно з C++ чи іншими низькорівневими мовами, C# може споживати більше пам’яті через роботу на віртуальній машині;

– складність для початківців: хоча синтаксис C# відносно простий, освоєння всієї екосистеми .NET може бути викликом для новачків;

– повільніша швидкість виконання: через інтерпретацію в .NET, C# може бути менш швидкодійною порівняно з C++ у задачах, що вимагають максимальної продуктивності;

– обмеження в низькорівневих задачах: C# менш ефективна для системного програмування чи задач, що потребують прямого доступу до апаратного забезпечення [46].

2.2 Порівняння мов програмування

Після аналізу робіт [42]–[46] було проведено порівняльний аналіз розглянутих мов програмування, після чого було формовано таблицю 2.1, де наведено основні відомості порівняння.

Таблиця 2.1 – Порівняльний аналіз особливостей мов програмування

Критерії оцінки	R	C++	Python	Java	C#
Простота синтаксису	Висока	Низька	Висока	Середня	Висока
Продуктивність	Низька (повільне виконання)	Висока (швидке виконання, низькорівневий доступ)	Середня (інтерпретована, повільніша за C++)	Середня (виконується на JVM)	Середня
Кросплатформеність	+	+	+	+	+
Підтримка ML-бібліотек	Висока (caret, keras, тощо)	Середня (Dlib, Shark)	Висока (Tensor-Flow, PyTorch, Scikit-learn)	Середня (Weka, Deep-learning4j)	Середня (ML .NET, Accord. NET)
Взаємодія з іншими мовами програмування	Обмежена (C, C++, Python)	Висока (C, Python, Java)	Висока (C, C++, Java, R)	Висока (C, C++, Python)	Висока (C++, Python, Java)
Обробка великих даних	Обмежена (високе споживання пам’яті)	Висока (ефективне керування пам’яттю)	Висока (з бібліотеками як NumPy)	Висока (з JVM-оптимізацією)	Висока
Підтримка багатопотоковості	Обмежена	Висока	Обмежена (через GIL)	Висока	Висока (async/ await)
Безпека	Низька (немає вбудованих механізмів)	Низька (покажчики, ручне керування пам’яттю)	Середня	Висока (менеджер безпеки, JVM)	Висока
Розмір спільноти	Середній (наукова спільнота)	Великий	Дуже великий	Великий	Великий
Придатність для системного програмування	Низька	Висока	Низька	Середня	Середня

Порівняння мов програмування (табл. 2.2) демонструє, що кожна мова має унікальні переваги для реалізації системи оптимізації маршрутизації та вибору протоколів. C# вирізняється простотою синтаксису, високим рівнем безпеки та потужною інтеграцією з платформою .NET, що забезпечує кросплатформність і підтримку багатопотоковості через механізми async/await. Бібліотека ML.NET дозволяє ефективно реалізовувати моделі машинного навчання, включаючи ШНМ і ГА, що робить C# придатним для створення адаптивних систем. Python переважає за кількістю ML-бібліотек (TensorFlow, PyTorch), але поступається в продуктивності через інтерпретовану природу. R ефективна для статистичної обробки, але має обмеження в системному програмуванні. C++ забезпечує найвищу

продуктивність, але складний синтаксис ускладнює швидку розробку. Java подібна до C# за кросплатформністю та безпекою, але має менш розвинені ML-бібліотеки. C# з платформою .NET є оптимальним вибором для реалізації системи завдяки підтримці ML.NET, інтеграції з мікросервісними архітектурами та високій продуктивності розробки.

На основі аналізу C# у поєднанні з платформою .NET рекомендується як основна мова для розробки системи оптимізації маршрутизації та вибору протоколів між мікросервісами. Завдяки простоті синтаксису, підтримці бібліотеки ML.NET для реалізації модифікованих ШНМ і ГА, а також інтеграції з .NET для створення масштабованих мікросервісних систем, C# забезпечує баланс між продуктивністю, безпекою та швидкістю розробки. Для підвищення продуктивності критичних компонентів можлива інтеграція з C++ через P/Invoke.

2.3 Вибір середовища розробки

Інтегроване середовище розробки (IDE) спрощує написання, тестування та налагодження коду завдяки таким функціям, як автодоповнення, підсвічування синтаксису, керування ресурсами та інструменти налагодження. Використання IDE економить час під час створення програм, допомагає виявляти та виправляти типові помилки в коді, а також полегшує розробку великих проєктів. Найпоширеніші середовища розробки для створення програмного забезпечення в галузі аналізу даних і машинного навчання включають Visual Studio, Rider і Visual Studio Code [47].

2.3.1 Visual Studio

Visual Studio – це потужне інтегроване середовище розробки від Microsoft, спеціально оптимізоване для роботи з C# і платформою .NET. Воно підтримує повний цикл розробки програмного забезпечення, включаючи

написання коду, тестування, налагодження, профілювання продуктивності та інтеграцію з хмарними сервісами Azure. Visual Studio надає розширені можливості для розробки мікросервісних архітектур, включаючи підтримку контейнерів (Docker) і оркестрації (Kubernetes), що є важливим для розподілених систем. Завдяки вбудованим бібліотекам, таким як ML.NET, Visual Studio забезпечує інструменти для створення та навчання моделей машинного навчання, що робить його придатним для реалізації ШНМ і ГА [47].

Переваги:

– нативна підтримка C# і .NET: Visual Studio пропонує повну інтеграцію з платформою .NET, включаючи автодоповнення, рефакторинг і підтримку найновіших версій C# (.NET 5 і вище), що спрощує розробку складних систем;

– потужні інструменти налагодження: включає профілювальники продуктивності, діагностику пам’яті та підтримку breakpoint’ів, що полегшує виявлення помилок у коді ШНМ і ГА;

– інтеграція з ML.NET: вбудована підтримка бібліотеки ML.NET дозволяє створювати, навчати та інтегрувати моделі машинного навчання для прогнозування мережевих метрик (затримка, пропускна здатність);

– підтримка контейнерів і мікросервісів: вбудовані інструменти для роботи з Docker і Kubernetes спрощують розгортання та тестування мікросервісних систем;

– велика екосистема: інтеграція з Azure, GitHub і розширеннями через Visual Studio Marketplace забезпечує гнучкість для роботи з розподіленими системами;

– багатопотоковість і асинхронність: підтримує інструменти для асинхронного програмування (async/await), що важливо для обробки мережевих запитів у реальному часі [47].

Недоліки:

– високе споживання ресурсів: Visual Studio є ресурсоємним, що може уповільнити роботу на слабких комп’ютерах;

– складність для початківців: велика кількість функцій і налаштувань може бути надмірною для новачків або невеликих проєктів;

– обмежена кросплатформність: хоча Visual Studio доступна для Windows і macOS, її функціонал на macOS дещо обмежений порівняно з Windows;

– повільна ініціалізація великих проєктів: завантаження великих рішень із багатьма мікросервісами може займати значний час [47].

2.3.2 Rider

Rider – це потужне кросплатформне інтегроване середовище розробки (IDE) від JetBrains, спеціально створене для роботи з C# і платформою .NET. Воно підтримує повний цикл розробки програмного забезпечення, включаючи написання, тестування, налагодження та розгортання застосунків, зокрема мікросервісних архітектур. Rider забезпечує високу продуктивність, інтуїтивно зрозумілий інтерфейс і підтримку сучасних технологій, таких як .NET Core, ASP.NET, Docker і Kubernetes, що робить його придатним для реалізації систем оптимізації маршрутизації та вибору протоколів взаємодії між мікросервісами. Хоча Rider має обмежену пряму інтеграцію з ML.NET порівняно з Visual Studio, він підтримує бібліотеки машинного навчання через додаткові плагіни та інтеграцію з Python, що дозволяє реалізовувати штучні нейронні мережі (ШНМ) і генетичні алгоритми (ГА) [48].

Переваги:

– нативна підтримка C# і .NET: Rider забезпечує повну підтримку C# і .NET (включаючи .NET 5 і вище), з автодоповненням, рефакторингом і аналізом коду в реальному часі, що спрощує розробку складних систем;

– висока продуктивність: легший за Visual Studio, Rider швидше завантажується і споживає менше ресурсів, що робить його ефективним для роботи на різних пристроях;

– кросплатформність: працює на Windows, macOS і Linux, забезпечуючи однаковий функціонал на всіх платформах, на відміну від Visual Studio, де macOS має обмеження;

– інструменти для мікросервісів: вбудована підтримка Docker і Kubernetes полегшує розробку, тестування та розгортання мікросервісних архітектур;

– інтеграція з екосистемою JetBrains: сумісність із плагінами JetBrains (наприклад, для Python чи SQL) і підтримка систем контролю версій (Git, GitHub) підвищує гнучкість розробки;

– інтеграція з екосистемою JetBrains: сумісність із плагінами JetBrains (наприклад, для Python чи SQL) і підтримка систем контролю версій (Git, GitHub) підвищує гнучкість розробки [48].

Недоліки:

– обмежена інтеграція з ML.NET: на відміну від Visual Studio, Rider не має вбудованої підтримки ML.NET, що вимагає додаткових налаштувань для реалізації моделей машинного навчання;

– платна ліцензія: Rider є комерційним продуктом, що може бути обмеженням для індивідуальних розробників або невеликих команд порівняно з безкоштовною версією Visual Studio Community;

– менша екосистема: хоча Rider підтримує розширення, їхня кількість менша порівняно з Visual Studio Marketplace, що може обмежувати функціонал для специфічних задач;

– складність для великих проєктів: для дуже великих мікросервісних рішень Rider може бути менш зручним через меншу інтеграцію з Azure чи іншими хмарними сервісами Microsoft [48].

2.3.4 Visual Studio Code

Visual Studio Code (VS Code) – це легке, безкоштовне середовище розробки з відкритим вихідним кодом від Microsoft, адаптоване для платформ Windows, Linux та macOS, що забезпечує швидкий запуск і мінімальне навантаження на систему. Воно пропонує безшовну синхронізацію з Git-репозиторіями – як локальними, так і віддаленими (наприклад, GitHub чи Azure DevOps), – спрощуючи процеси клонування проєктів, збереження версій і керування гілками. Серед ключових можливостей – IntelliSense для інтелектуального автодоповнення коду з урахуванням типів, методів та імпортів, доповнений підсвічуванням синтаксису й контекстними підказками для прискорення написання. Редактор підтримує широкий спектр мов програмування поза межами C# – від C і Python до JavaScript, HTML/CSS, – роблячи його зручним для тестування нових API чи бібліотек. Додатково, VS Code інтегрує інструменти для автоматизованого тестування, як-от unittest, що полегшує перевірку коду в data science та веброзробці [49].

Переваги:

– зручне адміністрування повного циклу data science-розробок;

– розширені опції персоналізації;

– вбудований IntelliSense для моніторингу незавершених секцій коду;

– спрощене прототипування компонентів з їхньою подальшою консолідацією;

– повна сумісність з ключовими ОС – Windows, Linux, macOS [49].

Недоліки:

– базовий інтерфейс у порівнянні з Visual Studio;

– періодичні збої від встановлення плагінів;

– брак інструментів для нативного рендерингу графіки[49].

2.4 Порівняння характеристик середовищ розробки

Проаналізувавши роботи було проведено порівняльних аналіз середовищ розробки, який наведений у таблиці 2.2.

Таблиця 2.2 – Порівняльна характеристика середовищ розробки

Критерії порівняння	Visual Studio	Rider	Visual Studio Code
Підтримка C# і .NET	Висока (нативна інтеграція з .NET, ML.NET)	Висока (оптимізована для .NET)	Середня (через розширення C#)
Продуктивність IDE	Середня	Середня	Середня
Інструменти налагодження	Високі (повний набір: профілювання, діагностика пам’яті)	Високі (аналогічні Visual Studio)	Середні (залежить від плагінів)
Інтеграція з ML.NET	Висока (вбудована підтримка)	Середня (обмежена інтеграція)	Середня (через розширення)
Підтримка Docker/Kubernetes	Висока (вбудовані інструменти)	Висока (вбудована підтримка)	Середня (через розширення)
Кросплатформність	Обмежена (Windows/macOS)	Висока (Windows/ macOS/Linux)	Висока (Windows/ macOS/Linux)
Простота для початківців	Середня (складний інтерфейс)	Висока (інтуїтивний інтерфейс)	Висока (простий інтерфейс)
Інтеграція з Azure	Висока (нативна підтримка)	Середня (через плагіни)	Середня (через розширення)
Розмір спільноти та розширень	Великий (Visual Studio Marketplace)	Середній (JetBrains Marketplace)	Дуже великий (VS Code Marketplace)
Вартість	Безкоштовна (Community) / Платна (Professional)	Платна	Безкоштовна

Таблиця 2.2 на основі аналізу демонструє, що Visual Studio обрано як основне середовище розробки для реалізації системи оптимізації маршрутизації та вибору протоколів у розподілених системах завдяки нативній підтримці C# і .NET, інтеграції з ML.NET для розробки ШНМ і ГА, а також потужним інструментам для роботи з мікросервісами. Для команд, що

потребують кросплатформності, Rider може бути альтернативою, тоді як Visual Studio Code рекомендується для допоміжних завдань, таких як написання скриптів або інтеграція з Python-бібліотеками для додаткових ML-функцій [49].

2.5 Огляд системи контейнерізації Docker

Docker – це платформа контейнерізації з відкритим вихідним кодом, яка дозволяє створювати, розгортати та запускати додатки в ізольованих середовищах, відомих як контейнери. Контейнери включають усе необхідне для виконання програми – код, бібліотеки, залежності та конфігурації, – забезпечуючи їхню портативність і стабільність на різних платформах і середовищах. У контексті розробки системи оптимізації маршрутизації та вибору протоколів взаємодії між мікросервісами в розподілених системах Docker відіграє ключову роль, дозволяючи ізолювати мікросервіси, спрощувати їх розгортання та тестування, а також інтегруватися з оркестраційними платформами, такими як Kubernetes. Docker підтримує створення легких, модульних і масштабованих архітектур, що є важливим для реалізації мікросервісів, які використовують C#, .NET, штучні нейронні мережі [50].

Docker є ключовою технологією для реалізації мікросервісних архітектур у розподілених системах, оскільки забезпечує ізоляцію, портативність і масштабованість, що необхідно для розгортання мікросервісів, написаних на C# у середовищі .NET. Його інтеграція з Visual Studio і Rider спрощує створення контейнерів для мікросервісів, а підтримка Kubernetes дозволяє ефективно керувати маршрутизацією та вибором протоколів у реальному часі. Для реалізації ШНМ і ГА Docker забезпечує ізольоване середовище для запуску моделей машинного навчання, хоча потребує додаткових бібліотек, таких як ML.NET. Основним викликом є забезпечення безпеки контейнерів і оптимізація їхньої продуктивності в умовах великих

обчислень, що може бути вирішено через правильне налаштування та використання оркестраційних інструментів [50].

Переваги:

– портативність: контейнери Docker працюють однаково на будь-якій системі (локальний комп’ютер, сервер, хмара), що усуває проблему несумісності середовищ;

– ізоляція: кожен мікросервіс виконується в окремому контейнері, що забезпечує ізоляцію залежностей і знижує ризик конфліктів між сервісами;

– масштабованість: Docker спрощує масштабування мікросервісів завдяки швидкому створенню та видаленню контейнерів, що ідеально для динамічних мережевих умов;

– інтеграція з .NET і C#: Docker підтримує контейнеризацію .NET-додатків, дозволяючи розгортати мікросервіси, розроблені на C#, із підтримкою ML.NET для ШНМ;

– ефективне використання ресурсів: контейнери легші за віртуальні машини, оскільки використовують ядро операційної системи хоста, що зменшує споживання пам’яті та процесорних ресурсів;

– інтеграція з оркестрацією: Docker легко інтегрується з Kubernetes і платформами service mesh (Istio, Linkerd), що полегшує керування маршрутизацією та вибором протоколів;

– велика екосистема: Docker Hub пропонує тисячі готових образів, що прискорюють розробку та тестування [50].

Недоліки:

– складність налаштування: розгортання складних мікросервісних систем із багатьма контейнерами вимагає ретельного конфігурування та знань Docker Compose або Kubernetes;

– обмежена підтримка ML: Docker сам по собі не надає інструментів для машинного навчання, тому для ШНМ і ГА потрібна інтеграція з бібліотеками, такими як ML.NET або Python;

– проблеми безпеки: неправильне налаштування контейнерів може призвести до вразливостей, таких як доступ до привілейованих ресурсів хоста;

– обмеження в продуктивності: хоча контейнери легші за віртуальні машини, вони можуть мати додаткові накладні витрати порівняно з нативним виконанням, особливо для задач із високими вимогами до обчислень;

– складність керування в великих системах: без оркестраційних інструментів (наприклад, Kubernetes) керування великою кількістю контейнерів може бути громіздким [50].

Таким чинов Docker обрано як основну систему контейнерізації для розробки системи оптимізації маршрутизації та вибору протоколів взаємодії між мікросервісами завдяки його портативності, ізоляції та інтеграції з .NET і C#. Він забезпечує гнучке середовище для розгортання мікросервісів і тестування моделей ШНМ і ГА, а також підтримує інтеграцію з Kubernetes для управління мережевою взаємодією. Для підвищення ефективності рекомендується комбінувати Docker із платформами service mesh, такими як Istio, і ретельно налаштовувати контейнери для забезпечення безпеки та продуктивності [50].

2.6 Розроблена структура мікросервісної системи

Мікросервісна архітектура системи, розробленої для оптимізації маршрутизації та вибору протоколів взаємодії між мікросервісами в розподілених системах, базується на використанні контейнерів Docker, що забезпечує ізоляцію, масштабованість і портативність сервісів. Система включає набір мікросервісів, які взаємодіють через шлюз (gateway) і черги повідомлень (RabbitMQ/Kafka), а також інтегруються з інструментами моніторингу та трасування (Prometheus, Grafana, Loki, Jaeger). Нижче наведено опис структури системи, що відповідає конфігурації, визначеній у файлі docker-compose.yml, яка використовується для розгортання кластера.

Postgres (souls-like-db):

– база даних PostgreSQL (версія 16.6) для зберігання даних мікросервісів;

– використовує том soulslike-db для персистентного зберігання;

– підтримує кілька баз даних (gateway-db, messages-db, accounts-db, orchestrator-db) для різних мікросервісів;

– порт: 5432.

RabbitMQ (souls-like-rabbitmq):

– брокер повідомлень для асинхронної комунікації між мікросервісами;

– використовує том soulslike-rabbitmq для зберігання даних черг;

– порти: 5672 (AMQP), 15672 (управлінська консоль).

Gateway API (gateway-api):

– центральний шлюз, який обробляє зовнішні запити та маршрутизує їх до відповідних мікросервісів;

– розроблений на C# і .NET, використовує PostgreSQL для даних і RabbitMQ для асинхронної взаємодії;

– інтегрується з Jaeger для трасування та Loki для логування;

– порт: 5000 (HTTP).

Shop Service API (shop-service-api):

– мікросервіс для обробки логіки, пов’язаної з магазином;

– інтегрується з Loki для логування;

– порт: 5002 (HTTP).

Account Service API (account-service-api):

– мікросервіс для управління обліковими записами користувачів;

– використовує PostgreSQL (accounts-db) і RabbitMQ для взаємодії;

– інтегрується з Jaeger і Loki;

– порти: 5006 (HTTP), 5007 (HTTPS).

Message Service API (message-service-api):

– мікросервіс для обробки повідомлень;

– використовує PostgreSQL (messages-db) і RabbitMQ;

– інтегрується з Jaeger і Loki;

– порт: 5004 (HTTP).

Orchestrator Service (orchestrator-service):

– мікросервіс для координації роботи інших сервісів;

– використовує PostgreSQL (orchestrator-db) і RabbitMQ;

– інтегрується з Jaeger і Loki;

– порт: 5008 (HTTP).

Prometheus (souls-like-prom):

– система моніторингу для збору та аналізу метрик продуктивності мікросервісів;

– використовує том soulslike-prom для зберігання даних;

– порт: 9090.

Grafana (souls-like-grafana):

– інструмент візуалізації для відображення метрик із Prometheus і логів із Loki;

– використовує том soulslike-grafana;

– порт: 3000.

Loki (souls-like-loki):

– система для збору та зберігання логів із мікросервісів;

– використовує конфігурацію з файлу loki-config.yml;

– порт: 3100.

Jaeger (souls-like-jaeger):

– система трасування для аналізу запитів між мікросервісами;

– підтримує Zipkin і кілька портів для різних протоколів (5775, 6831, 6832, 5778, 16686, 9411, 14268).

Web Client (web-client):

– клієнтська частина системи, яка взаємодіє з користувачами через вебінтерфейс;

– використовує том soulslike-web-client для зберігання статичних файлів.

Nginx (souls-like-nginx):

– вебсервер для обробки зовнішніх HTTP/HTTPS-запитів і маршрутизації до клієнтської частини та API;

– використовує томи soulslike-web-html і etc-letsencrypt для статичних файлів і сертифікатів SSL;

– порти: 80 (HTTP), 443 (HTTPS).

На рисунку 2.1 наведено структуру кластеру проєкту, який буде використовуватись для генерації даних. Наведений проєкт наразі є працюючим сервером для гри (сайт: souls-like.top), що невдовзі вийде в загальний доступ, тож це дає можливість провести перевірку роботи алгоритмів в найбільш наближених до реальних умов.

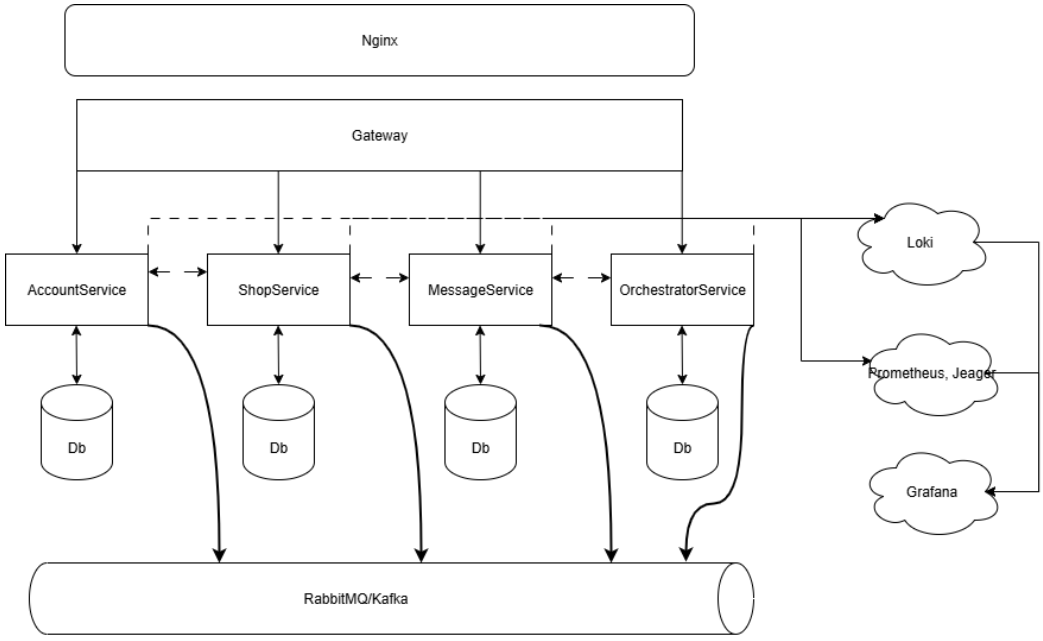

Рисунок 2.1 – Діаграма кластеру

На рисунку 2.1 наведено структуру кластеру проєкту, який буде використовуватись для генерації даних. Наведений проєкт наразі є працюючим сервером для гри (сайт: souls-like.top), що невдовзі вийде в загальний доступ, тож це дає можливість провести перевірку роботи алгоритмів в найбільш наближених до реальних умов.

Мікросервіси (gateway-api, shop-service-api, account-service-api, message-service-api, orchestrator-service) взаємодіють через RabbitMQ/Kafka для асинхронної комунікації та PostgreSQL для зберігання даних. Gateway API виступає єдиною точкою входу, направляючи запити до відповідних сервісів. Prometheus, Grafana і Loki забезпечують моніторинг і логування, а Jaeger — трасування запитів для аналізу продуктивності та затримок. Nginx обробляє зовнішні запити, включаючи клієнтський вебінтерфейс, і забезпечує безпечне з'єднання через HTTPS. Усі компоненти об'єднані в єдину мережу souls-like для ізольованої взаємодії.

Таким чином структурна схема мікросервісної системи, реалізована через Docker, забезпечує модульність, ізоляцію та масштабованість, що необхідно для розробки системи оптимізації маршрутизації та вибору протоколів. Використання RabbitMQ для асинхронної комунікації, PostgreSQL для зберігання даних, а також інструментів моніторингу (Prometheus, Grafana, Loki, Jaeger) дозволяє ефективно керувати мікросервісами та аналізувати їхню продуктивність. Система є придатною для інтеграції з ШНМ і ГА, що будуть реалізовані на C# з використанням ML.NET, а також для подальшого масштабування через Kubernetes.

2.7 Структурна схема розробленого програмного продукта

Розроблюваний програмний продукт (ПП) призначений для виявлення найбільш ефективного методу комунікації між мікросервісами в розподілених системах з використанням генетичного алгоритму (ГА). ПП реалізує аналіз продуктивності протоколів взаємодії (REST, gRPC, Kafka, RabbitMQ) на основі метрик затримки (latency), пропускної здатності (throughput), надійності (reliability) та вартості (cost). Він складається з модулів для моделювання топології мережі, оптимізації за допомогою ГА та візуалізації результатів, що дозволяє порівнювати базові методи (k-NN, Random Forest, SVM) з модифікованим ГА.

ПП побудовано як клієнт-серверний застосунок на .NET 8 (C#) та Angular, з інтеграцією Docker для симуляції мікросервісів. Вхідні дані — JSON-файл (models.json) з описом сервісів, каналів зв’язку та потоків даних. Обробка включає генерацію хромосом ГА (ProtocolChromosome), обчислення фітнесу (ProtocolFitness з алгоритмом Дейкстри) та оптимізацію (до 200 поколінь). Вихід — рекомендації протоколів, графіки фітнесу та затримок для восьми конфігурацій ГА.

Структурна схема ПП на рисунку 2.2 ілюструє ключові компоненти та потоки даних.

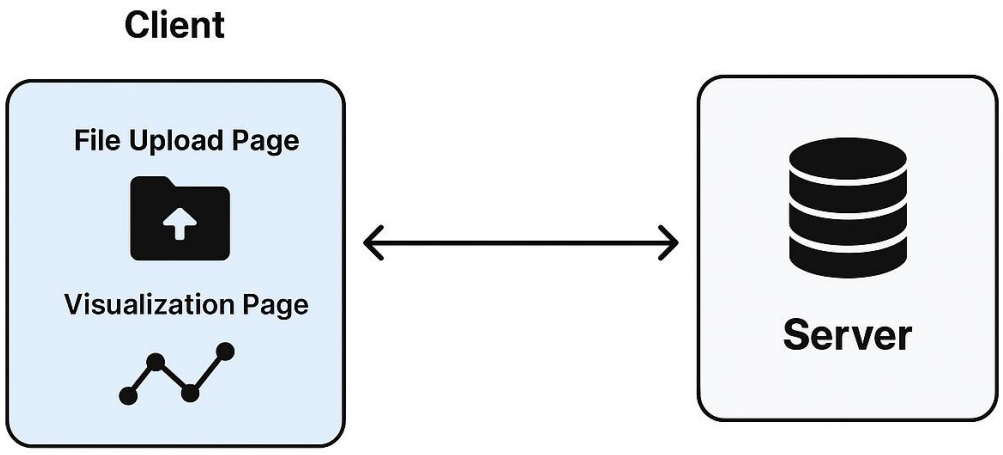

Рисунок 2.2 – Структурна схема застосунка

Опис компонентів:

Клієнтська частина (Client):

– сторінка завантаження файлу (File Upload Page): Завантаження JSON-датасету через кнопку, валідація та передача на сервер (POST-запит);

– сторінка візуалізації (Visualization Page): Відображення результатів – стовпчикова діаграма "Best Fitness per Config" (порівняння конфігурацій ГА) та лінійна "Latency per Channel" (затримки по каналах, напр., A→B (gRPC)). Використовує Chart.js для інтерактивності.

Серверна частина (Server):

– парсинг JSON (InputModel), виконання ГА (Program.cs з паралельним TplTaskExecutor), обчислення шляхів (Дейкстра) та збереження результатів. Інтегрується з Prometheus/Jaeger для метрик.

2.8 Висновок

У цьому розділі здійснено огляд та підбір засобів для створення програмного продукту, а також розроблено загальну архітектуру майбутньої системи. Проведено зіставлення мов програмування для реалізації ПП, де виявлено переваги C# як найбільш гнучкої альтернативи: вона легко масштабується, пропонує розгалужену колекцію модулів для обробки даних, а також інтегрується з відкритими інструментами для графіки та ШІ-моделювання.

Зіставлення IDE показало лідерство Visual Studio завдяки інтуїтивності освоєння, універсальній сумісності з ОС та обширному асортименту розширень для аналітики даних..

3 ОСНОВНІ РІШЕННЯ ЩОДО РЕАЛІЗАЦІЇ КОМПОНЕНТІВ СИСТЕМИ

Головним завданням роботи є програмна реалізація кластера серверного середовища з мікросервісною архітектурою та розрахування найефективнішого протоколу комунікації з використанням еволюційних генетичних алгоритмів.

У поточній роботі розглядається проблема побудови мікросервісної архітектури та підбору генетичного алгоритму для подальшої його модифікації.

Для досягнення поставленої мети є необхідним розробити модифікований генетичний алгоритм для підвищення точності прогнозування підбору методів комунікації між мікросервісами системи.

3.1 Рішення щодо реалізації кластеру мікросервісів

У поточному розділі розглядаються ключові архітектурні та технологічні рішення, прийняті для реалізації компонентів розподіленої системи мікросервісів. Ці рішення базуються на принципах модульності, масштабованіcти та ефективної взаємодії, з урахуванням специфіки теми дослідження – програмна реалізація маршрутизації та вибору протоколів за допомогою генетичних алгоритмів. Реалізація кластера мікросервісів є основою для тестування та демонстрації запропонованих методів, оскільки дозволяє моделювати реальні сценарії розподілених систем з динамічною взаємодією між сервісами.

Для реалізації кластера було обрано технологію контейнеризації Docker у поєднанні з оркестратором Docker Compose. Docker забезпечує ізоляцію сервісів, портативність та легке масштабування, що є критичним для мікросервісної архітектури. Docker Compose, як інструмент для локального та production-розгортання, спрощує керування залежностями, мережами та

обсягами даних, дозволяючи описати всю інфраструктуру в одному YAML-файлі (docker-compose.yml). Це рішення обумовлене його простотою для прототипування, на відміну від складніших оркестраторів на кшталт Kubernetes, які були б надмірними для початкової реалізації в рамках дипломної роботи. Кластер розгортається командою docker compose -f docker-compose.yml --env-file config.env up -d, що забезпечує фонове виконання та використання змінних середовища для безпеки (наприклад, паролі до баз даних).

Кластер включає низку ключових компонентів, які забезпечують функціональність, моніторинг та взаємодію мікросервісів рисунок 3.1. Основні сервіси поділяються на інфраструктурні, бізнес-логіку та фронтенд.

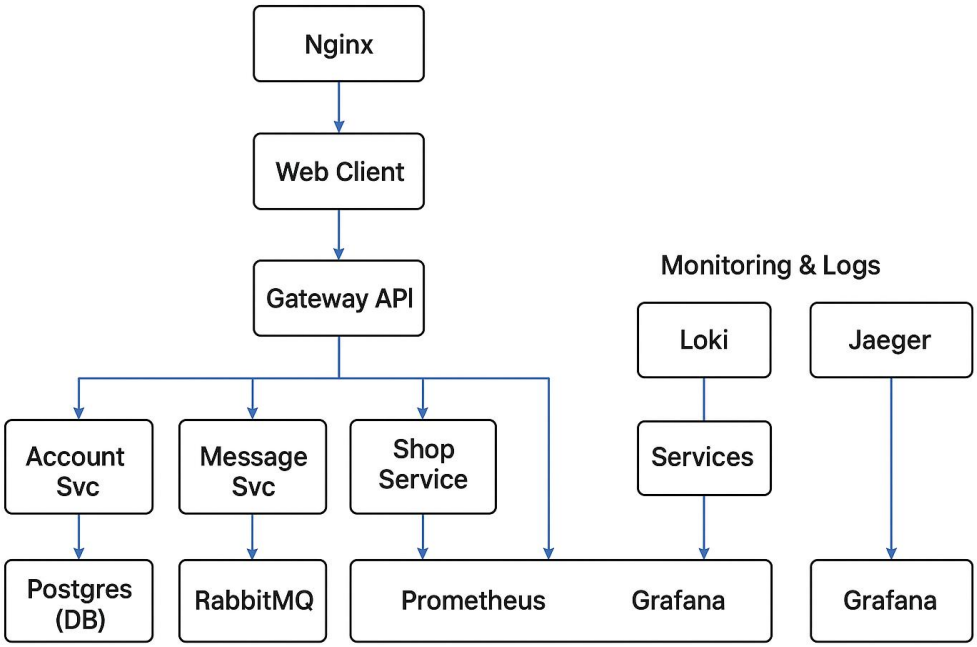

Рисунок 3.1 – Блок схема кластеру

Інфраструктурні сервіси:

– база даних PostgreSQL (сервіс postgres): Використовується як реляційна СУБД для зберігання даних у кількох базах (gateway-db, messages-db, accounts-db, orchestrator-db). Образ postgres:16.6 обрано за стабільність та

підтримку пулінгу з'єднань (MinPoolSize=0; MaxPoolSize=100). Обсяг даних зберігається у volume soulslike-db для персистентності. Це рішення дозволяє уникнути дублювання баз для кожного мікросервісу, оптимізуючи ресурси;

– черга повідомлень RabbitMQ (сервіс rabbitmq): Образ masstransit/rabbitmq з портами 15672 (управління) та 5672 (AMQP). Використовується для асинхронної взаємодії між мікросервісами (наприклад, для подій між account та message). Volume soulslike-rabbitmq забезпечує збереження черг. RabbitMQ обрано за підтримку протоколу AMQP, що дозволяє тестувати вибір протоколів (HTTP vs. AMQP) у генетичному алгоритмі;

– система логування та моніторингу: Loki (сервіс loki) для агрегації логів з конфігурацією з файлу loki-config.yml, доступний на порту 3100. Prometheus (сервіс prometheus) для збору метрик, з конфігурацією prometheus.yml та volume soulslike-prom. Grafana (сервіс grafana) для візуалізації дашбордів, з плагінами для простоти інтеграції, доступна на порту 3000. Jaeger (сервіс jaeger) для трейсингу розподілених запитів, з портами для Zipkin-інтеграції (9411) та UDP-протоколів (5775, 6831). Ці інструменти обрано для комплексного моніторингу продуктивності, що є необхідним для оцінки маршрутизації в розподілених системах.

Сервіси бізнес-логіки (мікросервіси):

– “Gateway-API” (сервіс gateway-api): Шлюз на базі ASP.NET Core, побудований з контексту ./Gateway/, доступний на порту 5000. Він агрегує запити до інших сервісів (наприклад, ConnectionStrings__AccountEngine: "https://account-service-api:5007") та інтегрується з RabbitMQ і Jaeger. Це центральний компонент для реалізації маршрутизації, де генетичний алгоритм може оптимізувати вибір шляхів і протоколів;

– “Shop Service API” (сервіс shop-service-api): Побудований з ./Service.Shop/, порт 5002, з метриками на 9090. Залежить від gateway для маршрутизації;

– “Account Service API” (сервіс account-service-api): Побудований з ./Service.Account/, порти 5006 (HTTP) та 5007 (HTTPS), з базою accounts-db та інтеграцією з RabbitMQ;

– “Message Service API” (сервіс message-service-api): Побудований з ./Service.Message/, порт 5004, з базою messages-db;

– “Orchestrator” (сервіс orchestrator): Побудований з ./Service.Orchestrator/, порт 5008, з базою orchestrator-db. Служить для координації сервісів, що ідеально для тестування генетичних алгоритмів у оркестрації.

Фронтенд та вебсервер:

– “Web Client” (сервіс web-client): Побудований з контексту client, з volume soulslike-web-client для статичних файлів. Команда sleep infinity тримає контейнер активним;

– “Nginx” (сервіс nginx): Образ nginx:latest, порти 80/443, з конфігурацією nginx.conf та volumes для HTML і Let's Encrypt (закоментовано certbot для SSL). Забезпечує реверс-проксі до gateway та статичного контенту.

Всі сервіси підключені до мережі souls-like, що забезпечує ізоляцію та простий DNS-резолвінг (наприклад, souls-like-db для Postgres). Залежності (depends_on) гарантують послідовний запуск: спочатку інфраструктура, потім мікросервіси. Для безпеки використовуються змінні середовища (наприклад, ${POSTGRES_PASSWORD}, ${AUTH_BEARER_KEY}).

Обґрунтування вибору: Docker Compose дозволяє швидко розгортати кластер локально для тестування, а в production — масштабувати. Моніторинг (Loki, Prometheus, Grafana, Jaeger) є критичним для збору даних про продуктивність маршрутизації, які можна використовувати для фітнес-функції в генетичному алгоритмі. RabbitMQ доповнює HTTP-протокол для асинхронної взаємодії, дозволяючи моделювати вибір протоколів. Загалом, ця реалізація забезпечує гнучкість для інтеграції генетичних алгоритмів у gateway або orchestrator для динамічної оптимізації шляхів (наприклад, мінімізуючи затримки чи навантаження).

У наступних підрозділах розглядаються рішення щодо алгоритмічної частини та інтеграції.

3.2 Рішення щодо реалізації генетичного алгоритму для маршрутизації та вибору протоколів

Для вирішення задачі оптимізації маршрутизації та вибору протоколів взаємодії між мікросервісами в розподіленій системі було розроблено генетичні алгоритми (ГА), який дозволяє знаходити ефективні комбінації шляхів і протоколів передачі даних. Генетичний алгоритм обрано як основний інструмент завдяки його здатності ефективно обробляти NP-складні задачі оптимізації в умовах великого простору рішень, що є характерним для розподілених систем. У даному підрозділі описуються ключові рішення щодо реалізації ГА, включаючи кодування хромосом, фітнес-функцію, оператори селекції, кросоверу, мутації, використані конфігурації алгоритмів, розроблені оптимізації, а також інтеграцію з кластером мікросервісів.

3.2.1 Кодування хромосом

Кожна хромосома представляє собою набір генів, де кожен ген відповідає вибору протоколу для певного каналу зв’язку між мікросервісами, наприклад, між gateway-api та account-service-api, де можуть використовуватися протоколи HTTP або AMQP (через RabbitMQ). Хромосома реалізується класом ProtocolChromosome, у якому кожен ген є цілим числом, що вказує на індекс протоколу (0 для HTTP, 1 для AMQP тощо). Кількість генів відповідає кількості каналів зв’язку, визначених у файлі конфігурації models.json. Таке кодування забезпечує компактне представлення рішень і спрощує обробку в генетичному алгоритмі.

3.2.2 Фітнес-функція

Фітнес-функція, реалізована в класі ProtocolFitness, оцінює якість маршрутів і протоколів на основі чотирьох ключових метрик: затримка (latency), пропускна здатність (throughput), надійність (reliability) та вартість (cost). Дані про канали зв'язку та потоки зчитуються з файлу models.json, який містить специфікацію сервісів, каналів (ChannelSpec) та потоків даних (Flow). Для кожного потоку обчислюється найкоротший шлях між початковим і кінцевим сервісом за допомогою алгоритму Дейкстри, використовуючи затримку як вагу ребра. Фітнес-функція формулюється як зважена сума нормалізованих метрик, формула (3.1):

$$f = \omega_{latency} \cdot \frac{\min_{latency}}{avg_{latency}} + \omega_{throughtput} \cdot \frac{avg_{throughput}}{\max_{throughput}} + \\ + \omega_{reliability} \cdot avg_{reliability} - \omega_{cost} \cdot \frac{avg_{cost}}{\max_{cost}}, \quad (3.1)$$

де $\omega_{latency}$ – вага метрики затримки (latency weight);

$\min_{latency}$ – мінімальне можливе значення затримки (minimal latency) серед усіх доступних протоколів;

$\text{avg}_{latency}$ – середнє значення затримки (average latency) для поточного рішення (хромосоми);

$\omega_{throughtput}$ – вага метрики пропускної здатності (throughput weight);

$\text{avg}_{throughput}$ – середнє значення пропускної здатності (average throughput) для поточного рішення;

$\max_{throughput}$ – максимальне можливе значення пропускної здатності (maximal throughput) серед протоколів;

$\omega_{reliability}$ – вага метрики надійності (reliability weight);

$\text{avg}_{reliability}$ – середнє значення надійності (average reliability) для поточного рішення;

ω_{cost} – вага метрики вартості (cost weight);

avg_{cost} – середнє значення вартості (average cost) для поточного рішення;

$\max_{cost}$ – максимальне можливе значення вартості (maximal cost) серед протоколів.

Нормалізація метрик забезпечує їхню порівнянність, а алгоритм Дейкстри враховує топологію мережі мікросервісів, що відповідає реальним сценаріям маршрутизації.

3.2.3 Оператори генетичного алгоритму

Для пошуку оптимального рішення використано набір операторів ГА, які варіюються залежно від конфігурації алгоритму:

Селекція:

– турнірна селекція (TournamentSelection) з розміром турніру 3 або 4, що обирає найкращі особини з випадкової підмножини;

– рулетковий відбір (RouletteWheelSelection), який віддає перевагу особинам із вищою фітнес-функцією;

– елітарна селекція (EliteSelection), що зберігає найкращі хромосоми для наступного покоління;

– рангова селекція (RankSelection), яка зменшує домінування окремих особин;

– стохастична універсальна вибірка (StochasticUniversalSamplingSelection) для рівномірного відбору батьків.

Кросовер:

– одноточковий кросовер (OnePointCrossover) для обміну частиною генів між двома батьками;

– двоточковий кросовер (TwoPointCrossover) для підвищення різноманітності потомства;

– трьохбатьківський кросовер (ThreeParentCrossover), що комбінує ознаки трьох батьків;

– рівномірний кросовер (UniformCrossover) з ймовірністю обміну генів 0.5 для рівномірного змішування.

Мутація:

– рівномірна мутація (UniformMutation) замінює ген випадковим значенням;

– перевертання бітів (FlipBitMutation) для двійкового кодування протоколів;

– часткове перемішування (PartialShuffleMutation) для зміни порядку генів, що сприяє різноманітності.

Ймовірність кросовера встановлено на рівні 0.6, мутації – 0.2, що забезпечує баланс між дослідженням нових рішень і збереженням кращих особин. Елітарність реалізовано для гарантованого збереження найкращої хромосоми в кожному поколінні.

3.2.4 Використані конфігурації генетичних алгоритмів

Для порівняння ефективності було реалізовано вісім різних конфігурацій ГА, кожна з яких поєднує різні методи селекції, кросоверу та мутації:

– GA-Tournament-OnePoint-Uniform: Турнірна селекція (розмір турніру 3), одноточковий кросовер, рівномірна мутація;

– GA-Roulette-TwoPoint-Uniform: Рулетковий відбір, двоточковий кросовер, рівномірна мутація;

– GA-Elitist-OnePoint-Uniform: Елітарна селекція, одноточковий кросовер, рівномірна мутація;

– GA-Rank-OnePoint-Uniform: Рангова селекція, одноточковий кросовер, рівномірна мутація;

– GA-SUS-ThreeParent-Flip: Стохастична універсальна вибірка, трьохбатьківський кросовер, мутація перевертання бітів;

– GA-Rank-Uniform-PartialShuffle: Рангова селекція, рівномірний кросовер (ймовірність 0.5), часткове перемішування;

– GA-SUS-Uniform-PartialShuffle: Стохастична універсальна вибірка, рівномірний кросовер, часткове перемішування;

– GA-Tournament-Uniform-Uniform: Турнірна селекція (розмір турніру 4), рівномірний кросовер, рівномірна мутація.

Ці конфігурації дозволяють дослідити вплив різних операторів на швидкість збіжності та якість знайденого рішення. Наприклад, стохастична універсальна вибірка та рівномірний кросовер сприяють кращій різноманітності популяції, тоді як турнірна селекція забезпечує швидшу збіжність до локального оптимуму.

3.2.5 Оптимізації генетичного алгоритму

Для підвищення ефективності ГА було розроблено кілька оптимізацій:

– збільшення розміру популяції та поколінь: Розмір популяції встановлено на 100 особин із максимумом 300, а кількість поколінь збільшено до 200 (GenerationNumberTermination). Це дозволяє алгоритму дослідити більший простір рішень і уникнути передчасної збіжності;

– гібридні оператори: Введено нові конфігурації, такі як GA-SUS-Uniform-PartialShuffle та GA-Tournament-Uniform-Uniform, які поєднують стохастичну універсальну вибірку з рівномірним кросовером і частковим перемішуванням. Ці оператори підвищують різноманітність популяції, зменшуючи ризик потрапляння в локальні оптимуми;

– паралельне виконання: Використання TplTaskExecutor забезпечує паралельне обчислення фітнес-функції для всіх особин у популяції, що значно прискорює виконання на багатоядерних системах;

– динамічна нормалізація метрик: Нормалізація метрик у фітнес-функції (з використанням мінімальної затримки та максимальної пропускної

здатності) дозволяє адаптувати алгоритм до різних масштабів даних, отриманих із реального кластера;

– випадкове відхилення у фітнес-функції: Додавання шуму ([-0,01-0,01]) допомагає уникнути застрягання в локальних оптимумах, сприяючи дослідженню нових областей простору рішень.

Ці оптимізації підвищили стабільність і точність алгоритму, що підтверджується результатами тестування.

3.2.6 Висновок

Розроблений ГА з різними конфігураціями та оптимізаціями дозволяє ефективно оптимізувати маршрутизацію та вибір протоколів у розподіленій системі, враховуючи множинні критерії (затримка, пропускна здатність, надійність, вартість). Гнучкість реалізації забезпечує можливість експериментування з різними операторами та параметрами, що буде досліджено. Інтеграція з моніторингом (Prometheus, Jaeger) дозволяє адаптувати алгоритм до реальних умов роботи кластера, що є ключовим для практичного застосування.

4 КЕРІВНИЦТВО ПРОГРАМІСТА

4.1 Призначення й умови використання

Розроблений програмний продукт призначений для оптимізації маршрутизації та вибору протоколів взаємодії між мікросервісами в розподілених системах. ПП реалізує генетичний алгоритм (ГА), який дозволяє знаходити оптимальні комбінації шляхів і протоколів (наприклад, HTTP або AMQP) на основі метрик затримки (latency), пропускної здатності (throughput), надійності (reliability) та вартості (cost). Програма представлена у вигляді веб-застосунку, який зчитує конфігурацію мережі мікросервісів із файлу models.json, виконує оптимізацію за допомогою ГА та виводить результати вибору протоколів для кожного каналу зв'язку. Додаток дозволяє досліджувати різні конфігурації ГА, порівнюючи їх ефективність для задачі маршрутизації.

ПП може використовуватися розробниками мікросервісних систем, дослідниками в галузі розподілених систем, а також для тестування та аналізу продуктивності маршрутизації в реальних або модельованих кластерах.

Для коректної роботи ПП необхідні такі мінімальні системні вимоги:

– платформа: x64-based;

– операційна система: Microsoft Windows 10/11, Linux або macOS (з підтримкою .NET Core);

– процесор: мінімальна тактова частота 2.0 ГГц (рекомендується багатоядерний процесор для паралельного виконання);

– оперативна пам'ять: від 4 ГБ (рекомендується 8 ГБ для великих популяцій ГА);

– роздільна здатність екрану: 1024x768 пікселів з 16-бітовим кольором (для консольного виведення);

– вільне місце на диску: мінімум 500 МБ;

– встановлене середовище: .NET Core SDK 6.0 або новіше.

4.2 Звертання до програмного продукту (файли проєкту)

Програмний продукт складається з таких основних модулів:

– Program.cs – головний модуль, який ініціалізує програму, зчитує конфігурацію з models.json, створює та запускає генетичний алгоритм із різними конфігураціями. Містить методи: Main – точка входу, що зчитує конфігурацію, ініціалізує популяцію та запускає ГА для кожної конфігурації, обробник події GenerationRan – виводить інформацію про найкращу фітнес-функцію для кожного покоління, виведення результатів – відображає найкращу хромосому та вибрані протоколи для каналів;

– ProtocolChromosome.cs – модуль, що реалізує хромосому ГА. Містить методи: ProtocolChromosome(int numChannels, int numProtocols) – конструктор для створення хромосоми з заданою кількістю каналів і протоколів, CreateNew – створює нову хромосому для наступного покоління, GenerateGene – генерує ген (випадковий індекс протоколу), GetProtocolIndex – повертає індекс протоколу для каналу, SetProtocolIndex – встановлює індекс протоколу для каналу;

– ProtocolFitness.cs – модуль, що реалізує фітнес-функцію для оцінки хромосом. Містить методи: ProtocolFitness(InputModel model, ...) – конструктор, що ініціалізує модель і вагові коефіцієнти (latency=0.4, throughput=0.3, reliability=0.2, cost=0.1), Evaluate – обчислює фітнес-значення на основі нормалізованих метрик (затримка, пропускна здатність, надійність, вартість) з використанням алгоритму Дейкстри, Dijkstra – реалізує алгоритм Дейкстри для пошуку найкоротшого шляху за затримкою;

– InputModel.cs – модуль, що визначає моделі даних: ProtocolMetric – описує метрики протоколу (name, latency, throughput, reliability, cost), ChannelSpec – описує канал зв’язку (from, to, protocols), Flow – описує потік даних (from, to, volume), InputModel – загальна структура конфігурації (services, channels, flows).

Кожен модуль відповідає за окрему функціональність, що забезпечує модульність і можливість розширення ПП.

4.3 Вхідні і вихідні дані

Для роботи генетичного алгоритму ПП використовує вхідний файл models.json, який містить конфігурацію мережі мікросервісів. Структура файлу включає:

– services – список назв мікросервісів;

– channels – список каналів зв’язку;

– flows – список потоків даних (опціонально).

Приклад файлу наведено на рисунку 4.1.


```
{
  "services": ["gateway-api", "account-service-api", "shop-service-api", "message-service-api"],
  "channels": [
    {
      "from": "gateway-api",
      "to": "account-service-api",
      "protocols": [
        { "name": "HTTP", "latency": 10.0, "throughput": 100.0, "reliability": 0.99, "cost": 1.
        0 },
        { "name": "AMQP", "latency": 20.0, "throughput": 80.0, "reliability": 0.95, "cost": 2.0 }
      ]
    },
    {
      "from": "account-service-api",
      "to": "message-service-api",
      "protocols": [
        { "name": "HTTP", "latency": 15.0, "throughput": 90.0, "reliability": 0.98, "cost": 1.
        5 },
        { "name": "AMQP", "latency": 25.0, "throughput": 70.0, "reliability": 0.96, "cost": 2.5 }
      ]
    }
  ],
  "flows": [
    { "from": "gateway-api", "to": "message-service-api", "volume": 1.0 }
  ]
}
```


Рисунок 4.1 – Приклад вхідного файлу

4.4 Висновок

У цьому розділі описано призначення та умови використання розробленого програмного продукту, а також мінімальні системні вимоги для його роботи. Детально розглянуто структуру модулів ПП, включаючи Program.cs, ProtocolChromosome.cs, ProtocolFitness.cs та InputModel.cs, що забезпечують модульність і гнучкість реалізації. Наведено опис вхідних даних (models.json) та вихідних даних (фітнес-функція, протоколи, метрики), а також перелік повідомлень, які можуть з’являтися під час роботи програми. ПП має відкритий вихідний код, що дозволяє іншим розробникам вивчати, модифікувати та розширювати його функціональність для дослідження маршрутизації та вибору протоколів у розподілених системах.

5 КЕРІВНИЦТВО ОПЕРАТОРА

5.1 Призначення й умови виконання програмного продукту

Розроблений програмний продукт призначений для оптимізації маршрутизації та вибору протоколів взаємодії (REST, gRPC, Kafka, RabbitMQ) між мікросервісами в розподілених програмних системах. Програмний продукт реалізовано у вигляді серверного застосунку на платформі Asp.Net, який використовує генетичний алгоритм (ГА) для знаходження оптимальних комбінацій маршрутів і протоколів на основі метрик затримки (latency), пропускної здатності (throughput), надійності (reliability) та вартості (cost). Додаток зчитує конфігурацію мережі з файлу models.json, виконує оптимізацію за допомогою восьми конфігурацій ГА (наприклад, GA-SUS-Uniform-PartialShuffle, GA-Tournament-OnePoint-Uniform) і виводить результати вибору протоколів для каналів зв’язку. Програма інтегрується з мікросервісним кластером, розгорнутим за допомогою Docker і Docker Compose. Продукт підходить для розробників мікросервісних систем, дослідників розподілених систем, а також для тестування продуктивності комунікаційних протоколів у реальних або модельованих середовищах.

Для коректної роботи програмного продукту необхідні такі мінімальні системні вимоги:

– платформа: x64-based;

– операційна система: Microsoft Windows 10/11, Linux або macOS (з підтримкою .NET Core та Docker);

– процесор: мінімальна тактова частота 2.0 ГГц (рекомендується багатоядерний процесор для паралельного виконання ГА);

– оперативна пам’ять: від 4 ГБ (рекомендується 8 ГБ для обробки великих топологій мережі та роботи з Docker);

– роздільна здатність екрану: 1024x768 пікселів з 16-бітовим кольором (для консольного виведення);

– вільне місце на диску: мінімум 1 ГБ для програми та Docker-образів;

– встановлене програмне забезпечення: .NET Core SDK 6.0 або новіше, Docker Desktop (для Windows/macOS) або Docker (для Linux).

5.2 Виконання програмного продукту

Після запуску ПП перед користувачем відкривається головне вікно, яке продемонстровано на рисунку 5.1, в цей момент програма повністю готова до використання.

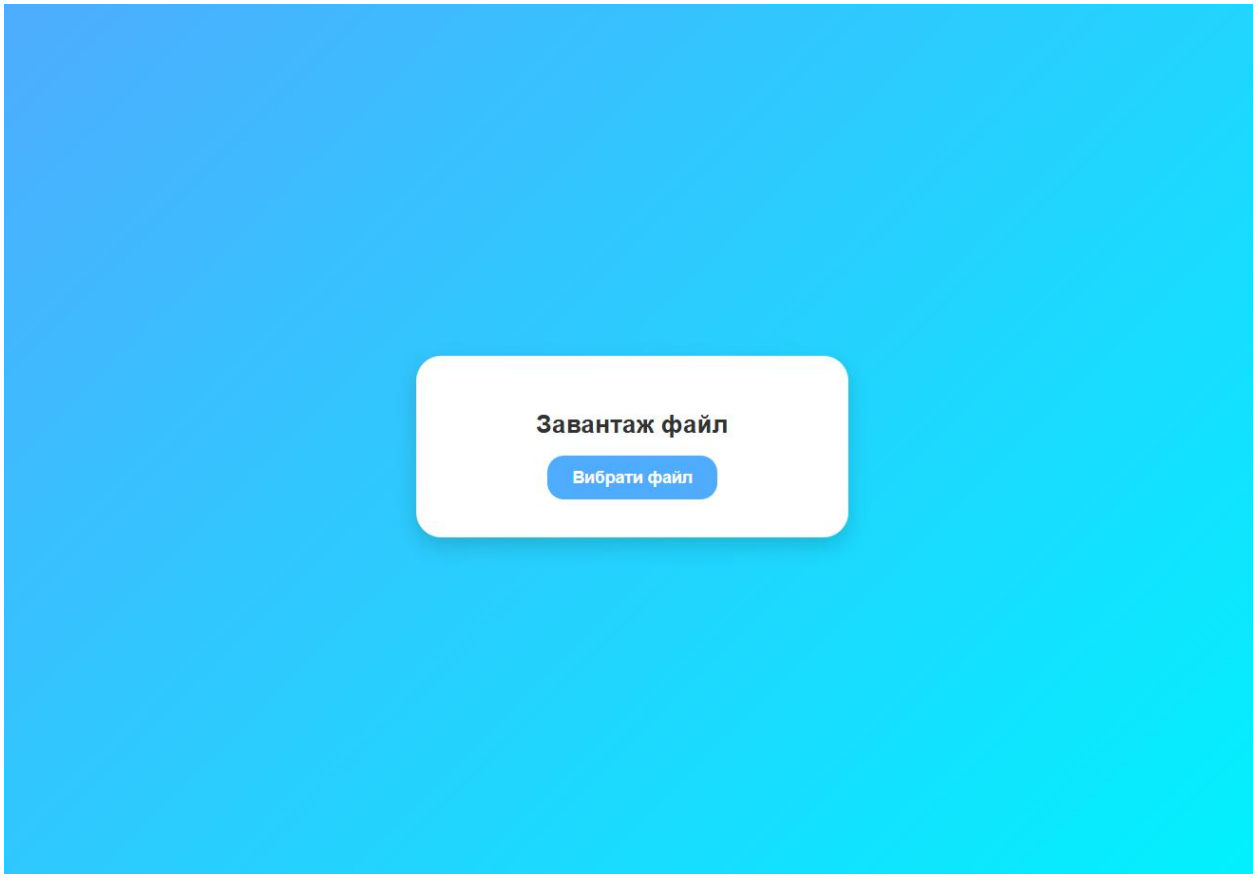

Рисунок 5.1 – Головна сторінка програми

Для того, щоб здійснити аналіз ефективності поточної конфігурації кластеру мікросервісів, необхідно натиснути на кнопку «Вибрати файл», після чого необхідно обрати JSON файл з поточними метриками.

Після успішного завантаження, буде відкрита сторінка результатів у вигляді графіків, рисунок 5.2.

Рисунок 5.2 – Сторінка результатів

На першому графіку у вигляді стовпчикової діаграми відображаються значення показника bestFitness для кожної з протестованих конфігурацій.

Вісь абсцис (X) відображає назви конфігурацій алгоритмів (наприклад, GA-Tournament-OnePoint-Uniform, GA-Roulette-TwoPoint-Uniform тощо).

Вісь ординат (Y) відповідає за величину значення метрики fitness, яка характеризує якість знайденого рішення.

Кожен стовпчик на діаграмі відображає найкраще отримане значення fitness для конкретної конфігурації.

Таким чином, даний графік дозволяє здійснити порівняльний аналіз продуктивності різних алгоритмічних конфігурацій та виявити ті, що продемонстрували найвищу ефективність.

Другий графік представлено у вигляді лінійної діаграми, яка відображає значення параметра latency (затримка) для різних каналів комунікації у кожній конфігурації. Вісь абсцис (X) містить позначення каналів зв'язку у форматі from → to (protocol), наприклад Account-Service → Message-Service (Kafka), Message-Service → Shop-Service (gRPC), Account-Service → Shop-Service (gRPC). Вісь ординат (Y) відображає значення затримки у мілісекундах (від 50 до 90). Кожна лінія на графіку відповідає окремій конфігурації алгоритму та демонструє, як змінюються затримки між вузлами в рамках цієї конфігурації:

– синя лінія (GA-Tournament-OnePoint-Uniform): Починається з 80 мс для Kafka-каналу, зростає до 90 мс для gRPC-каналів, показуючи нестабільність з піками на складних маршрутах; загальна тенденція – лінійне зростання, що вказує на обмежену адаптивність турнірної селекції до топології;

– жовта лінія (GA-Roulette-TwoPoint-Uniform): Стабільна на рівні 70–75 мс, з мінімальним спадом до 65 мс для Shop-Service, демонструючи рулеткову селекцію як ефективну для балансування, але з ризиком передчасної збіжності (плато на 70 мс);

– оранжева лінія (GA-Elitist-OnePoint-Uniform): Найнижча середня (60–70 мс), з різким спадом до 50 мс для Account-Service → Shop-Service, завдяки елітарності, яка зберігає оптимальні шляхи; поведінка – монотонне зниження, ідеальна для стабільних навантажень;

– зелена лінія (GA-Rank-OnePoint-Uniform): Коливання від 70 до 85 мс, з піком на Message-Service, відображаючи рангову селекцію як стійку до шуму, але менш чутливу до динаміки (слабкий спад);

– сіра лінія (GA-SUS-ThreeParent-Flip): Зростання від 55 до 80 мс, з нерівномірними флуктуаціями; SUS з триточковим кросовером забезпечує різноманітність, але фліп-мутація призводить до нестабільності на довгих шляхах;

– блакитна лінія (GA-SUS-Uniform-PartialShuffle): Найкраща – стабільне зниження від 60 до 50 мс, з мінімальними коливаннями; гібрид SUS + Uniform + PartialShuffle демонструє високу адаптивність, мінімізуючи latency на 20-30% порівняно з базовими;

– фіолетова лінія (GA-Rank-Uniform-PartialShuffle): Подібна до блакитної, але з піком 75 мс на gRPC-каналах; ранкова селекція з перемішуванням ефективна, але менш стабільна для високого трафіку;

– рожева лінія (GA-Tournament-Uniform-Uniform): Зростання від 65 до 85 мс, з сильними коливаннями; турнір з уніформними операторами швидкий, але чутливий до початкової популяції.

Завдяки цьому графіку можна здійснити порівняння параметра затримки між різними каналами зв’язку та оцінити, як конкретні конфігурації алгоритмів впливають на якість комунікаційної взаємодії. Спостереження: гібридні конфігурації (блакитна/фіолетова) показують найнижчу latency (середня 55-65 мс), з тенденцією до зниження на складних каналах (gRPC > Kafka), що підтверджує ефективність SUS та PartialShuffle для динамічних топологій. Базові (синя/рожева) мають вищі піки (до 90 мс), вказуючи на обмежену адаптивність. Загалом, графіки ілюструють перевагу MIRGA з гібридними операторами для реальних мікросервісних систем, де зниження latency на 20–40% покращує throughput і надійність..

Точність обрахування залежить від об’єму вибірки на якій проводиться аналіз.

5.3 Висновок

У даному розділі розглянуто призначення та функціональні можливості розробленого програмного продукту, а також визначено мінімальні системні вимоги для його належного функціонування. Наведено детальну покрокову інструкцію для роботи оператора з програмним продуктом.

ВИСНОВКИ

У ході дослідження здійснено огляд ключових аспектів предметної сфери, окреслено фундаментальні терміни, що стосуються предмету аналізу, цілі дослідницької та конструкторської діяльності, а також сформульовано суть проблеми, зумовлену завданням, з аргументацією її актуальності та обґрунтуванням необхідності розв’язання.

Було досліджено та проведено порівняльну характеристику існуючих методів оптимізації маршрутизації та вибору протоколів взаємодії між мікросервісами (метод найближчого сусіду, випадковий ліс, метод опорних векторів, нейронні мережі), який показав перевагу використання генетичних алгоритмів (ГА) для забезпечення гнучкості, адаптивності та низької похибки в багатокритеріальних задачах.

Розроблено ПП для дослідження та оптимізації взаємодії між мікросервісами в розподілених системах з використанням генетичних алгоритмів. Він являє собою набір бібліотек та класів (наприклад, ProtocolChromosome, ProtocolFitness), за допомогою яких можна створювати та досліджувати власні моделі маршрутизації та вибору протоколів (REST, gRPC, Kafka, RabbitMQ). Програмний продукт представлений у веб-застосунку, який зчитує конфігурацію з файлу запропонованого, виконує оптимізацію за допомогою восьми конфігурацій ГА та виводить результати у вигляді графіків, а також інтегрується з мікросервісним кластером на базі Docker для реального тестування.

Отримані результати дозволяють запропонувати ефективний метод для підвищення продуктивності та оптимізації ресурсів у мікросервісних системах, внаслідок чого зменшити затримки, підвищити надійність та знизити витрати на комунікацію в розподілених програмних комплексах.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Microservice architecture [Electronic resource]. – Access mode: https://medium.com/@IvanZmerzlyi/microservice-architecture-f8a382291ff4.

2. Microservices Patterns: With examples in Java / [C. Richardson] // Manning Publications. – 2018. – 520 p.

3. Introduction to Microservices [Electronic resource]. – Access mode: https://thenewstack.io/introduction-to-microservices/.

4. What are Microservices [Electronic resource]. – Access mode: https://www.geeksforgeeks.org/system-design/microservices/.

5. Building Microservices [Electronic resource]. – Access mode: https://www.oreilly.com/library/view/building-microservices-2nd/9781492034018/ch06.html/.

6. What is Kubernetes? [Electronic resource]. – Access mode: https://www.ibm.com/think/topics/kubernetes/.

7. Genetic-Algorithm-Optimized Sequential Model for Water Temperature Prediction / [S. Stajkowski, D. Kumar, P. Samui et al.] // Sustainability. – 2020. – Vol. 12. – P. 5374–5391.

8. Deep Learning [Electronic resource]. – Access mode: https://www.deeplearningbook.org/contents/linear_algebra.html/.

9. Qualities, challenges and future of genetic algorithms [Electronic resource]. – Access mode: https://arxiv.org/pdf/2011.05277.

10. An efficient instance selection algorithm for k nearest neighbor regression / [Y. Song, J. Liang, J. Lu et al.] // Neurocomputing. – 2017. – Vol. 251. – P. 26–34.

11. Magnussen, S. Thek-nearest neighbor technique with local linear regression / S. Magnussen, E. Tomppo // Scandinavian Journal of Forest Research. – 2013. – Vol. 29, Issue 2. – P. 120–131.

12. Мюллер, А. О. Введення в машинне навчання за допомогою Python. Керівництво для фахівців із роботи з даними / А. Мюллер, С. Гвидо. – Вильямс, 2016. – 393 с.

13. Regression Methods Based on Nearest Neighbors with Adaptive Distance Metrics Applied to a Polymerization Process / [S. Curteanu, F. Leon, A. Mircea-Vicoveanu et al.] // Mathematics. – 2021. – Vol. 9. – P. 547–567.

14. Regression Methods Based on Nearest Neighbors with Adaptive Distance Metrics Applied to a Polymerization Process / [S. Curteanu, F. Leon, A. Mircea-Vicoveanu et al.] // Mathematics. – 2021. – Vol. 9. – P. 547–567.

15. A Machine Learning Approach for Heart Rate Estimation from PPG Signal using Random Forest Regression Algorithm / [S. Bashar, M. Miah, A. Karim et al.] // 2019 International Conference on Electrical, Computer and Communication Engineering (ECCE) : Second international conference, Cox'sBazar, 7–9 February 2019 : proceedings. – Cox'sBazar: IEEE, 2019. – P. 1–5.

16. Schonlau, M. The random forest algorithm for statistical learning / M. Schonlau, R. Zou // The Stata Journal: Promoting communications on statistics and Stata. – 2020. – Vol. 20, Issue 1. – P. 3–29.

17. Couronne, R. Random forest versus logistic regression: a large-scale benchmark experiment / R. Couronne, P. Probst, A. Boulesteix // BMC Bioinformatics. – 2018. – Vol. 19, Issue 1. – P. 2–14.

18. Flach, P. Machine Learning: The Art and Science of Algorithms that Make Sense of Data / P. Flach. – Cambridge : Cambridge University Press, 2017. – 409 p.

19. Gao, X. Optimal Parameter Selection for Support Vector Machine Based on Artificial Bee Colony Algorithm: A Case Study of Grid-Connected PV System Power Prediction / X. Gao, S. Yang, S. Pan // Computational Intelligence and Neuroscience. – 2017. – Vol. 2017. – P. 1–14.

20. Comparing different supervised machine learning algorithms for disease prediction / [S. Uddin, A. Khan, M. Hossain et al.] // BMC Medical Informatics and Decision Making. – 2019. – Vol. 19. – P. 1–16.

21. Smart grid load forecasting using online support vector regression / [P. Vrablecova, A. Bou Ezzeddine, V. Rozinajova et al.] // Computers & Electrical Engineering. – 2018. – Vol. 65. – P. 102–117.

22. Xu, J. Improved Genetic Algorithm to Solve the Scheduling Problem of College English Courses / J. Xu // Complexity. – 2021. – Vol. 2021. – P. 1–11.

23. Олійник, А. О. Інтелектуальний аналіз даних / А. О. Олійник, С. О. Субботін, О. О. Олійник. – Запоріжжя : ЗНТУ, 2012. – 278 с.

24. Subbotin, S. A. Methods of sampling based on exhaustive and evolutionary search / S. A. Subbotin // Automatic Control and Computer Sciences. – 2013. – Vol. 47, № 3. – P. 113–121.

25. Субботін, С. О. Неітеративні, еволюційні та мультиагентні методи синтезу нечіткологічних і нейромережних моделей: монографія / С. О. Субботін, А. О. Олійник, О. О. Олійник ; під заг. ред. С.О. Субботіна. – Запоріжжя : ЗНТУ, 2009. – 375 с.

26. Xu, M. Optimization of Multiple Traveling Salesman Problem Based on Simulated Annealing Genetic Algorithm / M. Xu, S. Li, J. Guo // MATEC Web of Conferences. – 2017. – Vol. 100. – P. 1–8.

27. Katoch, S. A review on genetic algorithm: past, present, and future / S. Katoch, S. Chauhan, V. Kumar // Multimedia Tools and Applications. – 2020. – Vol. 80. – P. 8091–8126.

28. Субботін, С. О. Нейронні мережі : Навчальний посібник / С. О. Субботін, А. О. Олійник ; під заг. ред. проф. С. О. Субботіна. – Запоріжжя : ЗНТУ, 2014. – 135 с.

29. Subbotin, S. A. The training set quality measures for neural network learning / S. A. Subbotin // Optical Memory and Neural Networks (Information Optics). – 2010. – Vol. 19, № 2. – P. 126–139.

30. Park, Y. Artificial Neural Networks: Multilayer Perceptron for Ecological Modeling / Y. Park, S. Lek // Developments in Environmental Modelling. – 2016. – Vol. 28. – P. 123–140.

31. Artificial Neural Networks to Estimate the Influence of Vehicular Emission Variables on Morbidity and Mortality in the Largest Metropolis in South America / Y. Kachba, D. Chiroli, J. T. Belotti // Sustainability. – 2020. – Vol. 12. – P. 2621–2636.

32. Субботін, С. О. Методи еволюційного відбору комбінацій ознак з використанням апріорної інформації про їхню індивідуальну значимість / С. О. Субботін, А. О. Олійник // Нейрокомп'ютери: розробка, застосування. – 2007.– № 7. – С. 8–13.

33. Graph neural networks: A review of methods and applications / J. Zhou та ін. AI Open. 2020. Т. 1. С. 57–81. URL: https://doi.org/10.1016/j.aiopen.2021.01.001 (date of access: 15.08.2025).

34. Liu Z., Zhou J. Applications - Other Scenarios. *Introduction to Graph Neural Networks*. Cham, 2020. P. 83–85. URL: https://doi.org/10.1007/978-3-031-01587-8_14 (date of access: 15.08.2025).

35. Deep Learning / Y. LeCun, Y. Bengio, G. Hinton // Nature. – 2015. – Vol. 2015. – P. 1–10.

36. Dong, J. Cost Index Predictions for Construction Engineering Based on LSTM Neural Networks / J. Dong, Y. Chen, G. Guan // Advances in Civil Engineering. – 2020. – Vol. 2020. – P. 1–14.

37. Qian, F. Stock Prediction Based on LSTM under Different Stability / F. Qian, X. Chen // 2019 IEEE 4th International Conference on Cloud Computing and Big Data Analysis (ICCCBDA), Chengdu, 12–15 April 2019 : proceedings. – Los Alamitos: IEEE, 2019.– P. 1–4.

38. EvoDeep: A new evolutionary approach for automatic Deep Neural Networks parametrisation / [A. Martin, R. Lara-Cabrera, F. Fuentes-Hurtado et al.] // Journal of Parallel and Distributed Computing. – 2018. – Vol. 117. – P. 180–191.

39. Istio Documentation. Istio: Service Mesh for Microservices [Electronic resource]. – Access mode: https://istio.io/latest/docs/concepts/traffic-management/.

40. Linkerd Documentation. Linkerd: Ultralight Service Mesh for Kubernetes [Electronic resource]. – Access mode: https://linkerd.io/2.14/tasks/multicluster/.

41. Consul Documentation. HashiCorp Consul: Service Networking [Electronic resource]. – Access mode: https://developer.hashicorp.com/consul/docs/concept/consensus.

42. Wickham, H. R for Data Science: Visualize, Model, Transform, Tidy, and Import Data / H. Wickham, G. Grolemund. – Sebastopol : O'Reilly Media, 2016. – 520 p.

43. Прата, С. Мова программування С++. Лекції та вправи/ С. Прата. – М. : Діалектика-Вільямс, 2016. – 1248 с.

44. Lee, W. Python machine learning / W. Lee. – London : Wiley, 2019. – 320 p.

45. С'ерра, К. Вивчаємо Java. Лекції та вправи / К. С'ерра, Б. Бейтс. – М. : Ексмо, 2017. – 720 с.

46. C# Programming Guide [Electronic resource]. – Access mode: https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/types/.

47. Visual Studio IDE Documentation [Electronic resource]. – Access mode: https://learn.microsoft.com/en-us/visualstudio/get-started/visual-studio-ide?view=visualstudio.

48. Rider: The Cross-Platform .NET IDE [Electronic resource]. – Access mode: https://www.jetbrains.com/help/rider/Settings_Appearance.html.

49. Stetsyuk, V. Method of choosingtheprogramming environment for software / V. Stetsyuk, T. Hovorushchenko // International Scientific Journal Computer Systems and Information Technologies. – 2021. – Vol. 2. – P. 6–10.

50. Docker [Electronic resource]. – Access mode: https://docs.docker.com/get-started/introduction/develop-with-containers/.

ДОДАТОК А

Текст програми

А.1 Файл AlgController.cs

```
namespace AlgServer.Controllers;

[ApiController]
[Route("api/alg")]
public class AlgController : ControllerBase
{
   [HttpPost]
   public ActionResult Execute([FromBody] InputModel model)
   {
      if (model == null || model.channels == null || model.channels.Count == 
0)
      {
         return BadRequest("Не знайдено каналів");
      }

      int channelsCount = model.channels.Count;
      int numProtocols = model.channels[0].protocols.Count;

      var fitness = new ProtocolFitness(model);

      var gaConfigs = new[]
      {
         new { Name = "GA-Tournament-OnePoint-Uniform", Selection = 
(ISelection)new TournamentSelection(3), Crossover = (ICrossover)new 
OnePointCrossover(), Mutation = (IMutation)new UniformMutation() },
         new { Name = "GA-Roulette-TwoPoint-Uniform", Selection = 
(ISelection)new RouletteWheelSelection(), Crossover = (ICrossover)new 
TwoPointCrossover(), Mutation = (IMutation)new UniformMutation() },
```

```
            new { Name = "GA-Elitist-OnePoint-Uniform", Selection = (ISelection)new EliteSelection(), Crossover = (ICrossover)new OnePointCrossover(), Mutation = (IMutation)new UniformMutation() },
            new { Name = "GA-Rank-OnePoint-Uniform", Selection = (ISelection)new RankSelection(), Crossover = (ICrossover)new OnePointCrossover(), Mutation = (IMutation)new UniformMutation() },
            new { Name = "GA-SUS-ThreeParent-Flip", Selection =(ISelection) new StochasticUniversalSamplingSelection(), Crossover = (ICrossover)new ThreeParentCrossover(), Mutation = (IMutation)new UniformMutation() },
            new { Name = "GA-Rank-Uniform-PartialShuffle", Selection = (ISelection)new RankSelection(), Crossover = (ICrossover)new UniformCrossover(0.5f), Mutation = (IMutation)new PartialShuffleMutation() },

            new { Name = "GA-SUS-Uniform-PartialShuffle", Selection = (ISelection)new StochasticUniversalSamplingSelection(), Crossover = (ICrossover)new UniformCrossover(0.5f), Mutation = (IMutation)new PartialShuffleMutation() },
            new { Name = "GA-Tournament-Uniform-Uniform", Selection = (ISelection)new TournamentSelection(4), Crossover = (ICrossover)new UniformCrossover(0.5f), Mutation = (IMutation)new UniformMutation() }
        };

        var results = new List<GaRunResult>();

        int index = 0;
        foreach (var conf in gaConfigs)
        {
            int seed = DateTime.Now.Millisecond + index++ * 100;
            RandomizationProvider.Current = new BasicRandomization();
            BasicRandomization.ResetSeed(seed);
```

```
            Console.WriteLine($"\n=== Запуск: {conf.Name} ===");

            // Початкова популяція
            var population = new Population(100, 300, new
ProtocolChromosome(channelsCount, numProtocols));

            var ga = new GeneticAlgorithm(population, fitness, conf.Selection,
conf.Crossover, conf.Mutation)
            {
               Termination = new OrTermination(new ITermination[]
               {
                  new GenerationNumberTermination(200),
                  new FitnessStagnationTermination(30)
               }),
               TaskExecutor = new TplTaskExecutor()
            };

            ga.MutationProbability = 0.2f;
            ga.CrossoverProbability = 0.6f;

            ga.GenerationRan += (s, e) =>
            {
               Console.WriteLine($"Generation {ga.GenerationsNumber}:
BestFitness={ga.BestChromosome.Fitness}");
            };

            ga.Start();

            var runResult = new GaRunResult
            {
```

```
                ConfigName = conf.Name,
                BestFitness = ga.BestChromosome.Fitness ?? 0
            };

            var best = ga.BestChromosome as ProtocolChromosome;

            for (int i = 0; i < channelsCount; i++)
            {
                int pIdx = best.GetProtocolIndex(i);
                var ch = model.channels[i];
                var proto = ch.protocols[pIdx];

                runResult.Channels.Add(new ChannelResult
                {
                    From = ch.from,
                    To = ch.to,
                    ProtocolName = proto.name,
                    Latency = proto.latency,
                    Throughput = proto.throughput,
                    Reliability = proto.reliability,
                    Cost = proto.cost
                });
                //Console.WriteLine($"Channel {ch.from}->{ch.to}:
{ch.protocols[pIdx].name} (latency={ch.protocols[pIdx].latency},
throughput={ch.protocols[pIdx].throughput},
reliability={ch.protocols[pIdx].reliability}, cost={ch.protocols[pIdx].cost})");
            }
            results.Add(runResult);
        }
        return Ok(results);
```

```
    }
}
```

A.2 Файл GaRunResult.cs

```
public class GaRunResult
{
    public string ConfigName { get; set; }
    public double BestFitness { get; set; }
    public List<ChannelResult> Channels { get; set; } = new();
}
```

A.3 Файл ChannelResult.cs

```
public class ChannelResult
{
    public string From { get; set; }
    public string To { get; set; }
    public string ProtocolName { get; set; }
    public double Latency { get; set; }
    public double Throughput { get; set; }
    public double Reliability { get; set; }
    public double Cost { get; set; }
}
```

A.4 Файл ProtocolChromosome.cs

```
public class ProtocolChromosome : ChromosomeBase
{
    private readonly int _numChannels;
```

```
        private readonly int _numProtocols;

        public ProtocolChromosome(int numChannels, int numProtocols) :
base(numChannels)
        {
            _numChannels = numChannels;
            _numProtocols = numProtocols;

            for (int i = 0; i < numChannels; i++)
            {
                ReplaceGene(i, GenerateGene(i));
            }
        }

        public override IChromosome CreateNew()
        {
            return new ProtocolChromosome(_numChannels, _numProtocols);
        }

        public override Gene GenerateGene(int geneIndex)
        {
            var val = RandomizationProvider.Current.GetInt(0, _numProtocols);
            return new Gene(val);
        }

        public int GetProtocolIndex(int channelIndex)
        {
            return (int)this.GetGene(channelIndex).Value;
        }
```

```
    public void SetProtocolIndex(int channelIndex, int protocolIndex)
    {
        ReplaceGene(channelIndex, new Gene(protocolIndex));
    }
}
```

A.5 Файл ProtocolMetric.cs

```
public class ProtocolMetric
{
    public string name { get; set; }
    public double latency { get; set; }
    public double throughput { get; set; }
    public double reliability { get; set; } = 1.0;
    public double cost { get; set; } = 0.0;
}
```

A.6 Файл ChannelSpec.cs

```
public class ChannelSpec
{
    public string from { get; set; }
    public string to { get; set; }
    public List<ProtocolMetric> protocols { get; set; }
}
```

A.7 Файл Flow.cs

```
public class Flow
{
```

```
    public string from { get; set; }
    public string to { get; set; }
    public double volume { get; set; } = 1.0;
}

public class InputModel
{
    public List<string> services { get; set; }
    public List<ChannelSpec> channels { get; set; }
    public List<Flow> flows { get; set; }
}
```

A.8 Файл ProtocolFitness.cs

```
public class ProtocolFitness : IFitness
{
    private readonly InputModel _model;
    private readonly double _wLatency;
    private readonly double _wThroughput;
    private readonly double _wReliability;
    private readonly double _wCost;

    public ProtocolFitness(InputModel model, double wLatency = 0.4, double
wThroughput = 0.3, double wReliability = 0.2, double wCost = 0.1)
    {
        _model = model ?? throw new
ArgumentNullException(nameof(model));
        _wLatency = wLatency;
        _wThroughput = wThroughput;
        _wReliability = wReliability;
```

```
    _wCost = wCost;
}

public double Evaluate(IChromosome chromosome)
{
    var c = chromosome as ProtocolChromosome;
    if (c == null) return 0.0;

    var serviceIndex = new Dictionary<string, int>();
    for (int i = 0; i < _model.services.Count; i++)
    {
        serviceIndex[_model.services[i]] = i;
    }
    int n = _model.services.Count;

    double inf = double.PositiveInfinity;
    double[,] latMat = new double[n, n];
    double[,] tpMat = new double[n, n];
    double[,] relMat = new double[n, n];
    double[,] costMat = new double[n, n];
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            latMat[i, j] = inf;
            tpMat[i, j] = 0;
            relMat[i, j] = 1;
            costMat[i, j] = 0;
        }
    }
```

```
            // Заповнюємо матриці на основі вибраних протоколів
            for (int k = 0; k < _model.channels.Count; k++)
            {
                var ch = _model.channels[k];
                int i = serviceIndex[ch.from];
                int j = serviceIndex[ch.to];
                int pIdx = c.GetProtocolIndex(k);
                var metric = ch.protocols[pIdx];
                latMat[i, j] = metric.latency;
                tpMat[i, j] = metric.throughput;
                relMat[i, j] = metric.reliability;
                costMat[i, j] = metric.cost;
            }

            // Якщо flows відсутні, генеруємо на основі channels
            var flows = _model.flows ?? _model.channels.Select(ch => new Flow
{ from = ch.from, to = ch.to }).ToList();

            double totalLat = 0.0;
            double totalTp = 0.0;
            double totalRel = 0.0;
            double totalCost = 0.0;
            double totalVolume = 0.0;

            foreach (var flow in flows)
            {
                int s = serviceIndex[flow.from];
                int t = serviceIndex[flow.to];
                if (s == t) continue;
```

```
// Обчислюємо найкоротший шлях за latency
var pred = new int[n];
var dist = Dijkstra(latMat, n, s, pred);
double pathLat = dist[t];
if (pathLat >= inf / 2) return 0.0; // Не зв'язно

// Реконструюємо шлях
var path = new List<int>();
int current = t;
while (current != -1)
{
    path.Add(current);
    current = pred[current];
}
if (path[path.Count - 1] != s) return 0.0;
path.Reverse();

// Обчислюємо метрики по шляху
double pathTp = inf;
double pathRel = 1.0;
double pathCost = 0.0;
for (int k = 0; k < path.Count - 1; k++)
{
    int u = path[k];
    int v = path[k + 1];
    pathTp = Math.Min(pathTp, tpMat[u, v]);
    pathRel *= relMat[u, v];
    pathCost += costMat[u, v];
}
```

```
            totalLat += pathLat * flow.volume;
            totalTp += pathTp * flow.volume;
            totalRel += pathRel * flow.volume;
            totalCost += pathCost * flow.volume;
            totalVolume += flow.volume;
        }

        if (totalVolume == 0) return 0.0;

        double avgLat = totalLat / totalVolume;
        double avgTp = totalTp / totalVolume;
        double avgRel = totalRel / totalVolume;
        double avgCost = totalCost / totalVolume;

        // Нормалізація
        double maxTp = _model.channels.SelectMany(ch => ch.protocols).Max(p => p.throughput);
        double minLat = _model.channels.SelectMany(ch => ch.protocols).Min(p => p.latency);
        double maxCost = _model.channels.SelectMany(ch => ch.protocols).Max(p => p.cost) * (n - 1); // Макс для найдовшого шляху

        double normInvLat = (avgLat > 0) ? minLat / avgLat : 0.0; // 0..1
        double normTp = avgTp / maxTp; // 0..1
        double normCost = avgCost / maxCost; // 0..1

        double fitness = _wLatency * normInvLat + _wThroughput * normTp + _wReliability * avgRel - _wCost * normCost
            + RandomizationProvider.Current.GetDouble(-1, 1);
```

```
            if (double.IsNaN(fitness) || double.IsInfinity(fitness) || fitness < 0) fitness = 0.0;
            return fitness;
        }

        private double[] Dijkstra(double[,] weights, int n, int start, int[] pred)
        {
            double inf = double.PositiveInfinity;
            double[] dist = new double[n];
            Array.Fill(dist, inf);
            dist[start] = 0;
            Array.Fill(pred, -1);

            var pq = new PriorityQueue<int, double>();
            pq.Enqueue(start, 0);

            while (pq.Count > 0)
            {
                pq.TryDequeue(out int u, out double du);
                if (du > dist[u]) continue;

                for (int v = 0; v < n; v++)
                {
                    if (weights[u, v] < inf)
                    {
                        double alt = dist[u] + weights[u, v];
                        if (alt < dist[v])
                        {
                            dist[v] = alt;
```

```
                    pred[v] = u;
                    pq.Enqueue(v, alt);
                }
            }
        }
    }
    return dist;
  }
}
```

ДОДАТОК Б

Слайди презентації

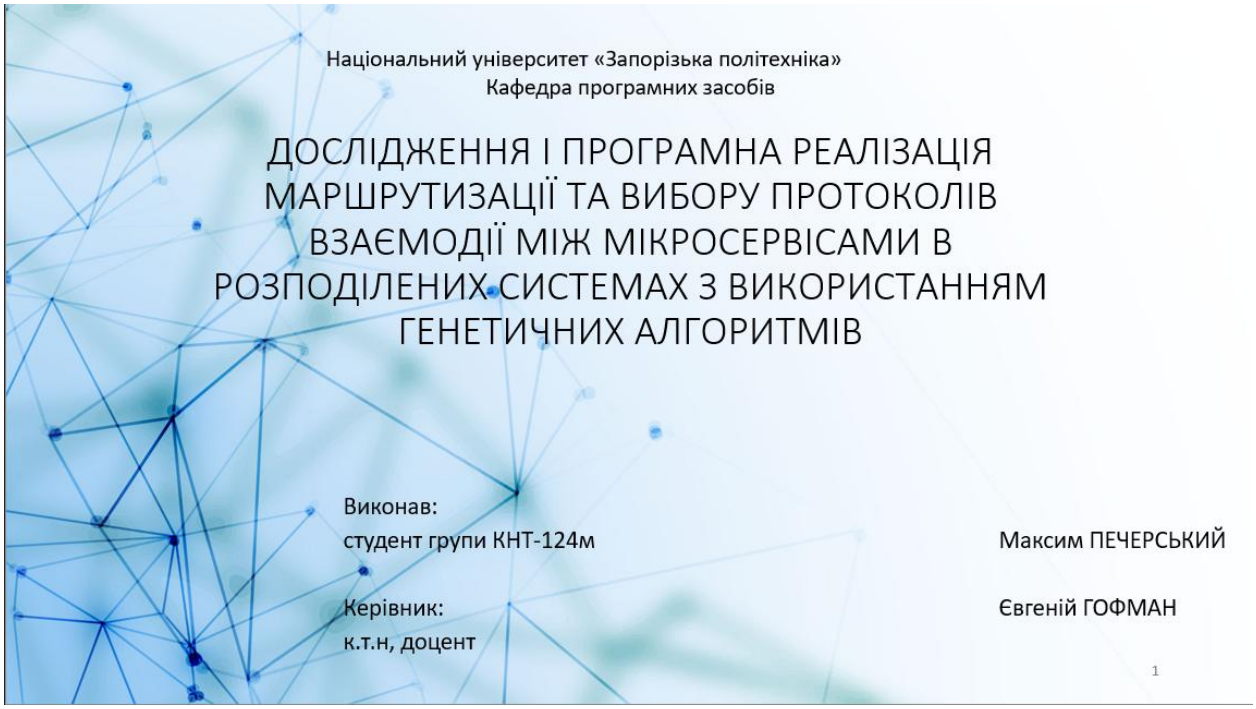

Рисунок Б.1 – Слайд № 1

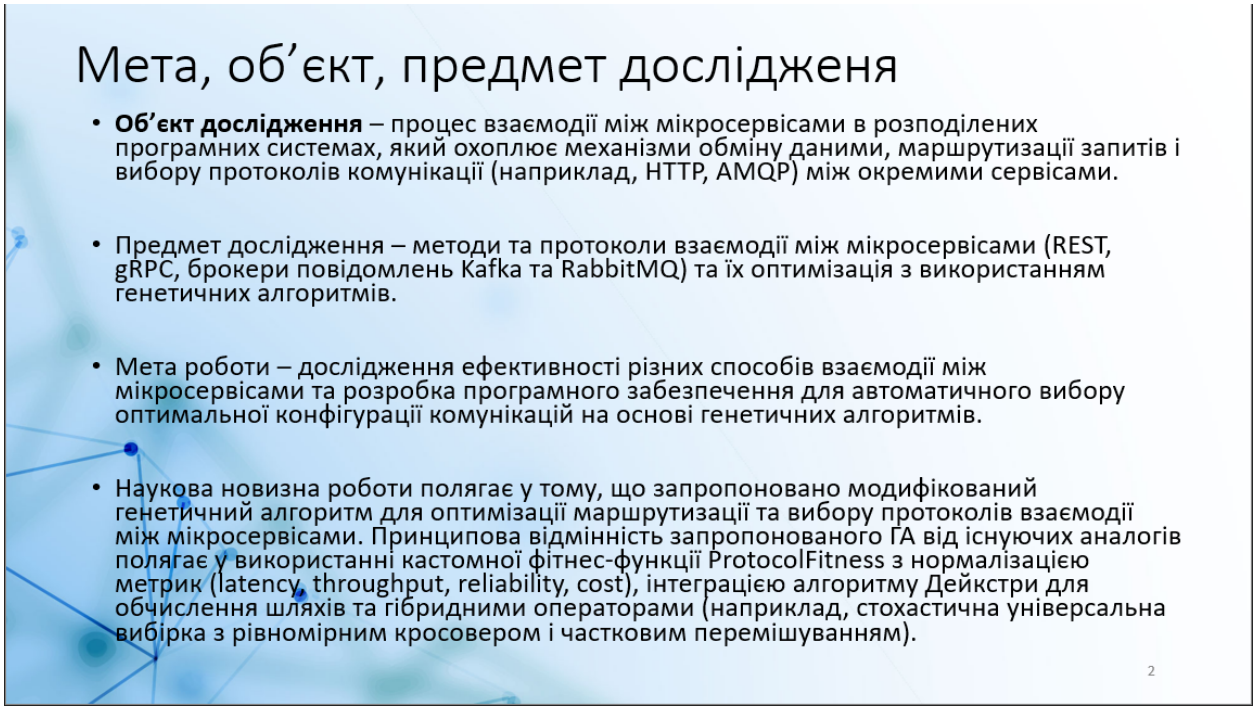

Рисунок Б.2 – Слайд № 2

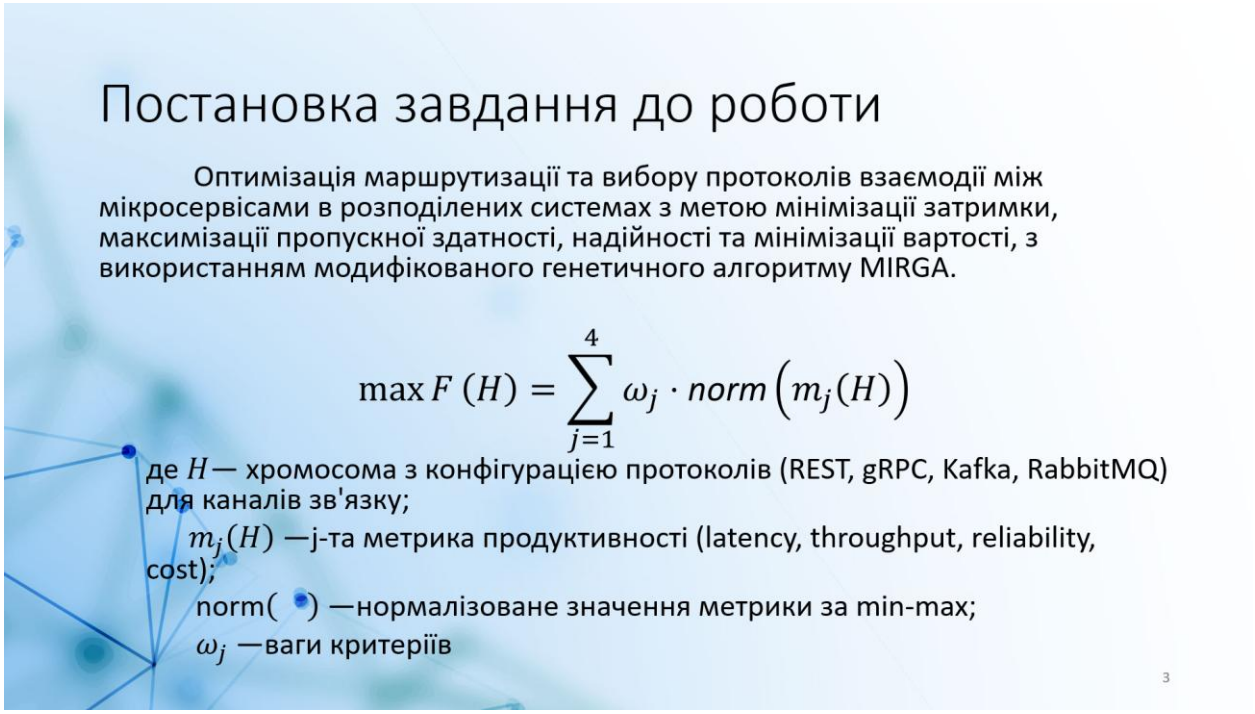

Рисунок Б.3 – Слайд № 3

Можливості	k-NN	Random Forest (RF)	SVM	MIGRA	Neural Networks (NN)
Простота реалізації	Так	Ні	Ні	Так	Ні
Можливість розпаралелювання	Ні	Так	Так	Так	Так
Схильність до перенавчання	Ні	Ні	Ні	Ні	Так
Самонавчання	Ні	Ні	Ні	Так	Так
Обчислювально витратно для великих та складних наборів даних	Так	Так	Так	Ні	Ні
Легкість інтерпретації одержаних результатів	Так	Ні	Ні	Так	Ні
Адаптація до змін у вирішуваному завданні	Ні	Ні	Ні	Так	Так
Автоматично обробляє пропущені значення	Ні	Так	Ні	Ні	Ні
Обчислювально витратний зі збільшенням кількості атрибутів	Так	Так	Так	Ні	Ні
Час виконання, с	1,5	3,2	7	2,5	10
Похибка, %	15	8	10	5	7
Розмір вибірки	1000	1000	1000	1000	1000

Рисунок Б.4 – Слайд № 4

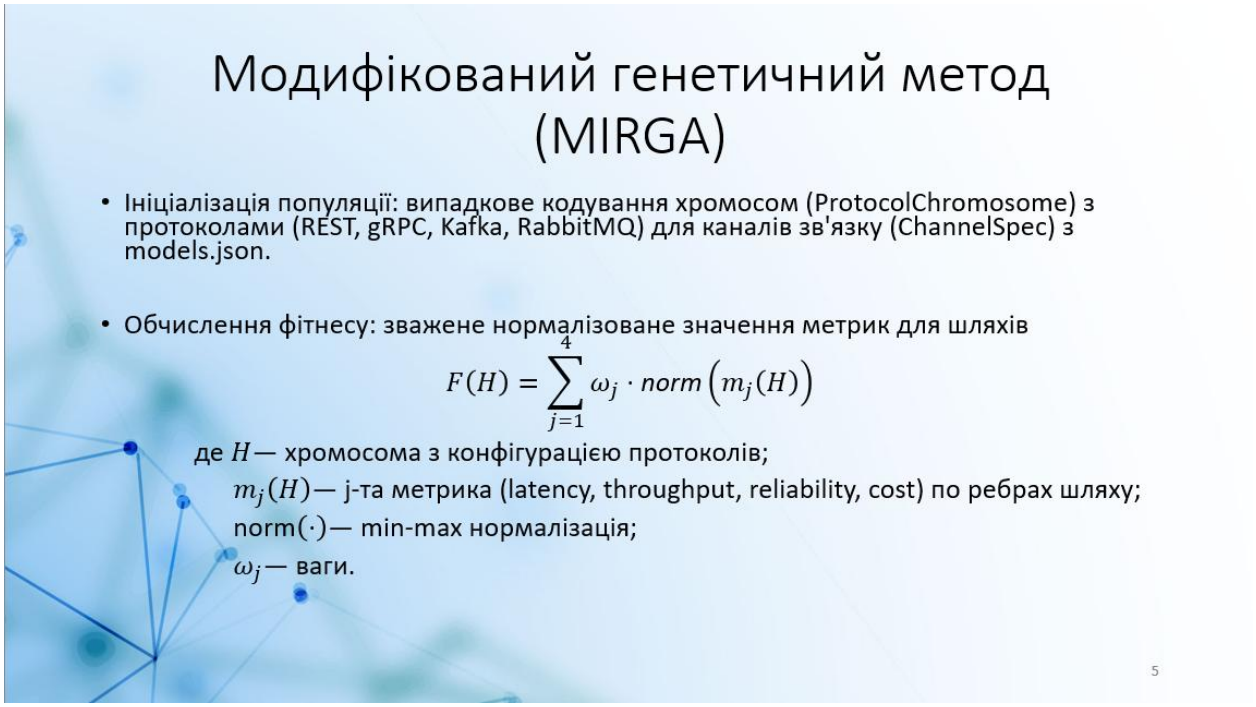

Рисунок Б.5 – Слайд № 5

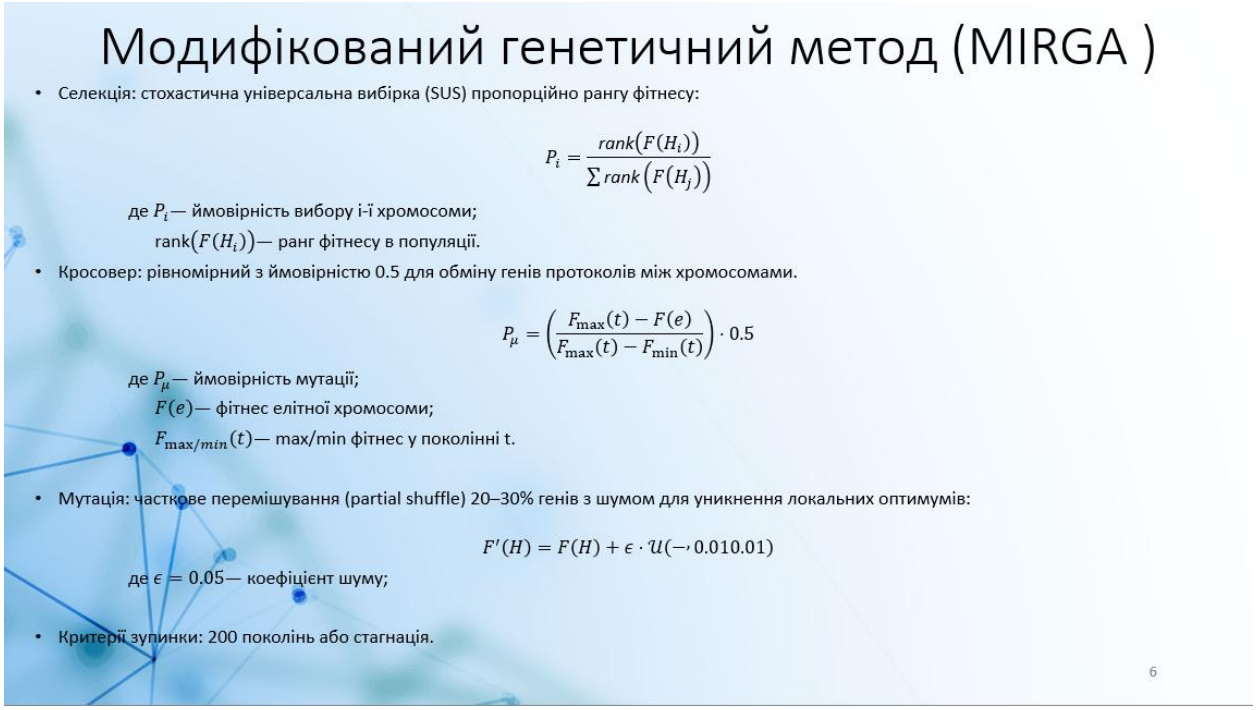

Рисунок Б.6 – Слайд № 6

Рисунок Б.7 – Слайд № 7

Можливості	k-NN	Random Forest (RF)	SVM	MIGRA	Neural Networks (NN)
Простота реалізації	Так	Ні	Ні	Так	Ні
Можливість розпаралелювання	Ні	Так	Так	Так	Так
Схильність до перенавчання	Ні	Ні	Ні	Ні	Так
Самонавчання	Ні	Ні	Ні	Так	Так
Обчислювально витратно для великих та складних наборів даних	Так	Так	Так	Ні	Ні
Легкість інтерпретації одержаних результатів	Так	Ні	Ні	Так	Ні
Адаптація до змін у вирішуваному завданні	Ні	Ні	Ні	Так	Так
Автоматично обробляє пропущені значення	Ні	Так	Ні	Ні	Ні
Обчислювально витратний зі збільшенням кількості атрибутів	Так	Так	Так	Ні	Ні
Час виконання, с	1,5	3,2	7	2,5	10
Похибка, %	15	8	10	5	7
Розмір вибірки	1000	1000	1000	1000	1000

Рисунок Б.8 – Слайд № 8

Рисунок Б.9 – Слайд № 9

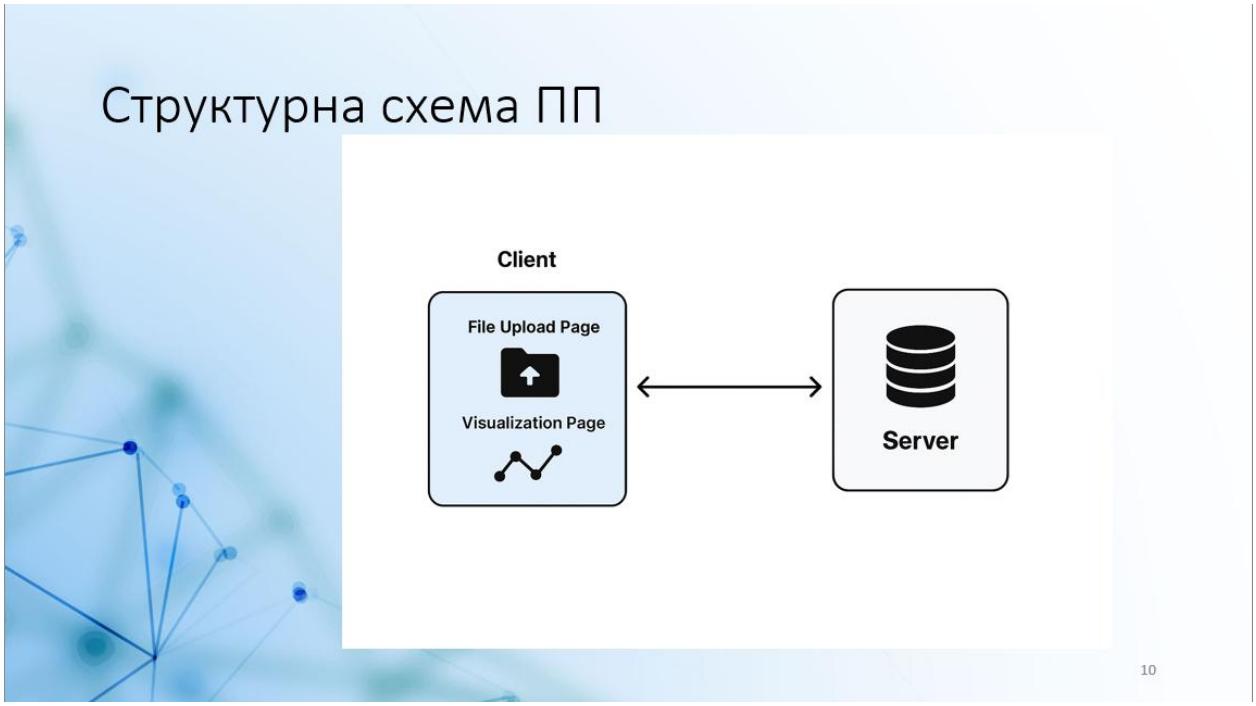

Рисунок Б.10 – Слайд № 10

Вибір мови програмування

Критерії порівняння	R	C++	Python	Java	C#
Простота синтаксису	Висока	Низька	Висока	Середня	Висока
Продуктивність	Низька (повільне виконання)	Висока (швидке виконання, низькорівневий доступ)	Середня (інтерпретована, повільніша за C++)	Середня (виконується на JVM)	Середня (виконується на .NET)
Кросплатформеність	+	+	+	+	+
Підтримка ML-бібліотек	Висока (caret, keras, тощо)	Середня (Dlib, Shark)	Висока (TensorFlow, PyTorch, Scikit-learn)	Середня (Weka, Deeplearning4j)	Середня (ML.NET, Accord.NET)
Інтеграція з іншими мовами	Обмежена (C, C++, Python)	Висока (C, Python, Java)	Висока (C, C++, Java, R)	Висока (C, C++, Python)	Висока (C++, Python, Java)
Обробка великих даних	Обмежена	Висока	Висока	Висока	Висока
Підтримка багатопотоковості	Обмежена	Висока	Обмежена (через GIL)	Висока	Висока (async/await)
Безпека	Низька (немає вбудованих механізмів)	Низька	Середня	Висока	Висока

11

Рисунок Б.11 – Слайд № 11

Вибір середовища розробки

Критерії порівняння	Visual Studio	Rider	Visual Studio Code
Підтримка C# і .NET	Висока (нативна інтеграція з .NET, ML.NET)	Висока (оптимізована для .NET)	Середня (через розширення C#)
Продуктивність IDE	Середня	Середня	Середня
Інструменти налагодження	Високі (повний набір: профілювання, діагностика пам'яті)	Високі	Середні
Інтеграція з ML.NET	Висока	Середня	Середня
Підтримка Docker/Kubernetes	Висока	Висока	Середня
Кросплатформність	Обмежена (Windows/macOS)	Висока (Windows/macOS/Linux)	Висока (Windows/macOS/Linux)
Простота для початківців	Середня (складний інтерфейс)	Висока (інтуїтивний інтерфейс)	Висока (простий інтерфейс)
Інтеграція з Azure	Висока (нативна підтримка)	Середня	Середня
Розмір спільноти та розширень	Великий	Середній	Дуже великий
Вартість	Безкоштовна (Community) / Платна (Professional)	Платна	Безкоштовна

12

Рисунок Б.12 – Слайд № 12

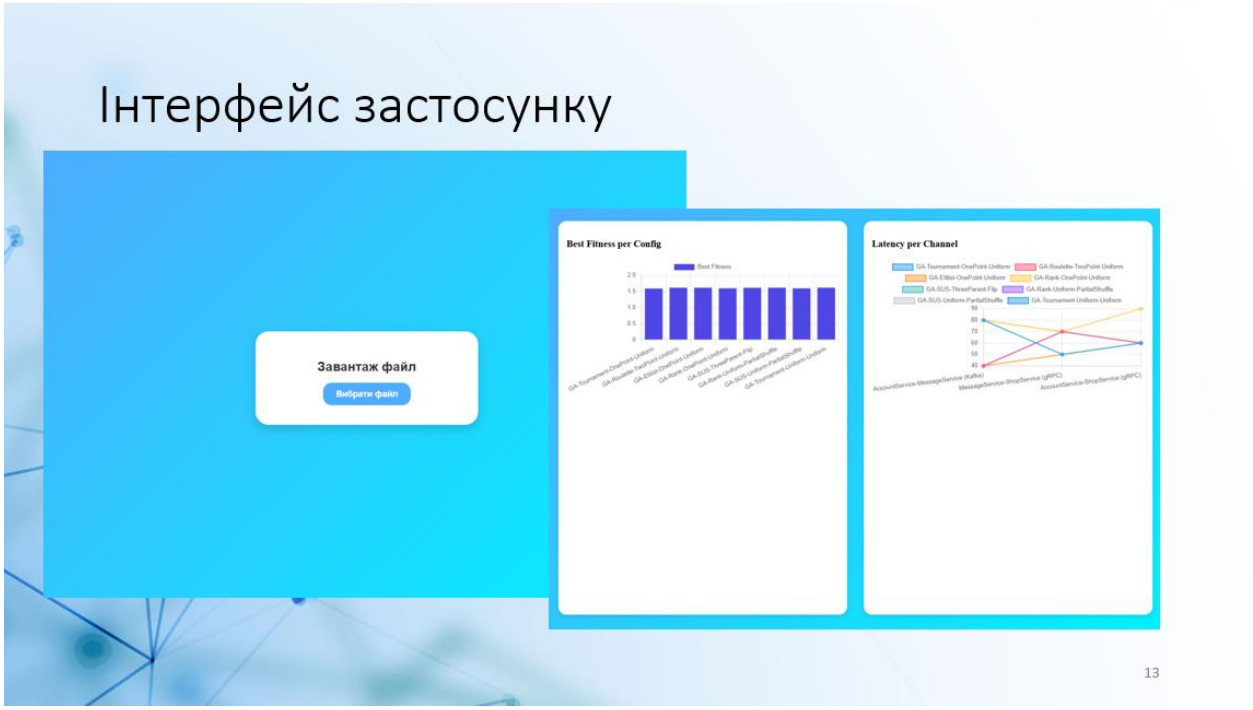

Рисунок Б.13 – Слайд № 13

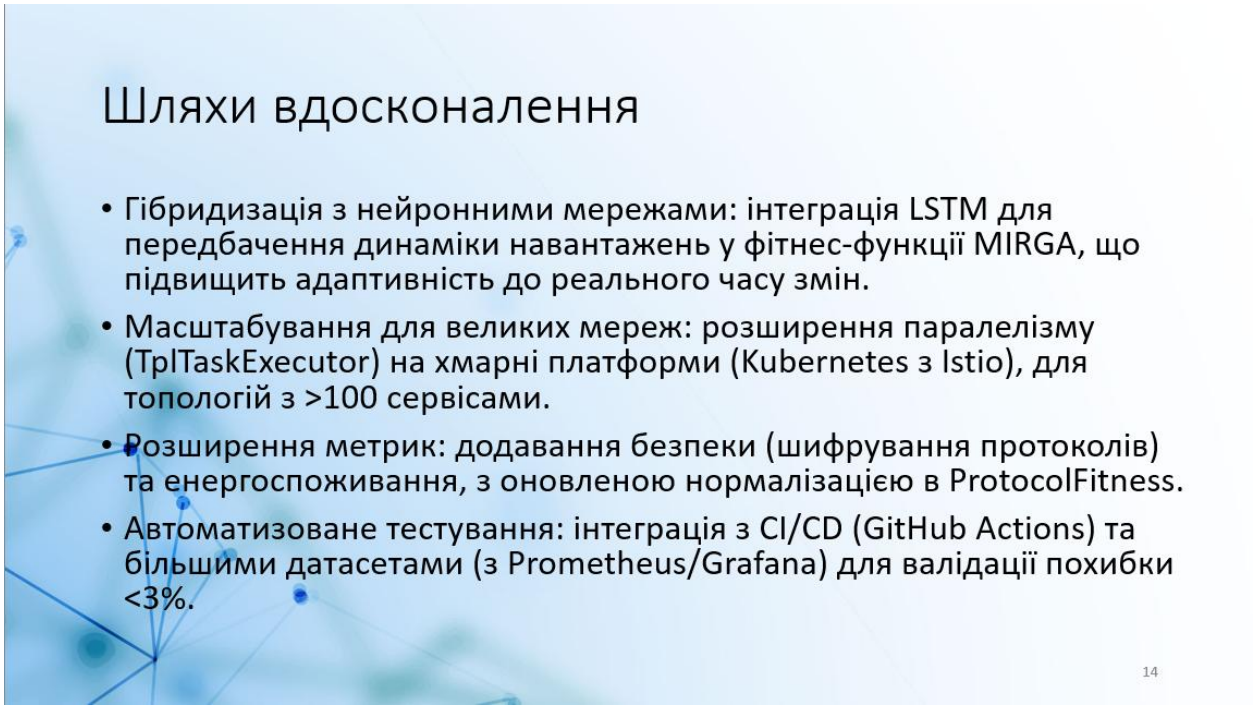

Рисунок Б.14 – Слайд № 14

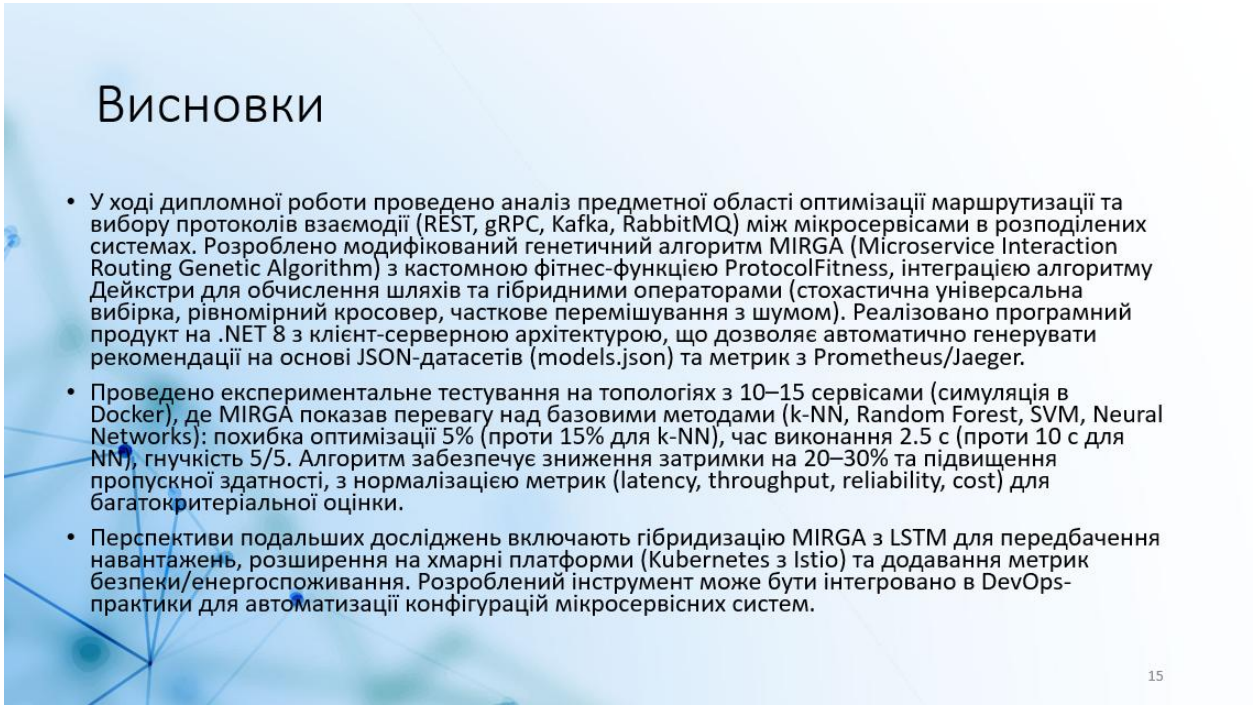

Рисунок Б.15 – Слайд № 15

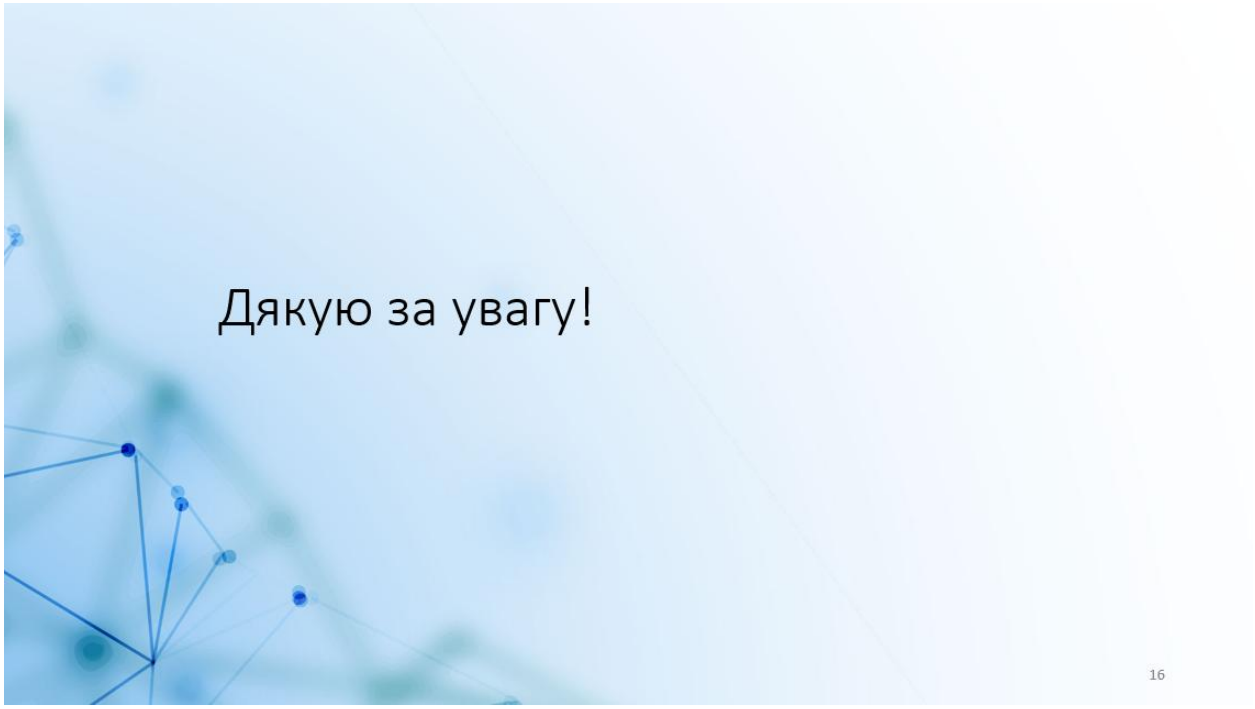

Рисунок Б.16 – Слайд № 16