МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
КРИВОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ СИСТЕМ ТА МЕРЕЖ

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

за спеціальністю 123
«КОМП'ЮТЕРНА ІНЖЕНЕРІЯ»

Тема наукової роботи:

«МЕТОДИ ТА ЗАСОБИ АВТОМАСШТАБУВАННЯ ВЕБСЕРВІСІВ НА ОСНОВІ ВИКОРИСТАННЯ МАШИННОГО НАВАЧАННЯ»

Виконав _____ М.П. Косей

Керівник роботи _____ Д. І. Кузнєцов

Нормоконтроль _____ Д. І. Кузнєцов

Завідувач кафедри _____ А. І. Купін

Кривий Ріг
2023

КРИВОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ СИСТЕМ ТА МЕРЕЖ

Ступінь вищої освіти магістр
Спеціальність 123 «Комп'ютерна інженерія»

ЗАТВЕРДЖУЮ
Завідувач кафедри, голова циклової комісії

_____ А. І. Купін

"____" _____ 20__ року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Косей Максим Петрович
(прізвище, ім'я, по батькові)

1. Тема роботи МЕТОДИ ТА ЗАСОБИ АВТОМАСШТАБУВАННЯ ВЕБ-СЕРВІСІВ НА ОСНОВІ ВИКОРИСТАННЯ МАШИННОГО НАВЧАННЯ

Керівник роботи Кузнєцов Д. І., кандидат технічних наук, доцент,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом вищого навчального закладу від "___"_____ 20__року №__

2. Строк подання студентом роботи _____

3. Вихідні дані до роботи _____

_____

_____

_____

_____

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)_____

_____

_____

_____

_____

_____

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)_____

_____

_____

_____

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання_____

КАЛЕНДАРНИЙ ПЛАН

/п	Назва етапів магістерської роботи	Строк виконання етапів роботи	При мітка

Студент _____ _____
(підпис) (прізвище та ініціали)

Керівник роботи _____ _____
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Пояснювальна записка: ? сторінок, ? рисунків, ? таблиць, ? додатків, ? - використаних джерела.

Мета даного курсового проекту – ?????.

Даний проект складається з 2 розділів.

У першому розділі виконується ????.

Другий розділ ??????.

Explanatory note: ?? pages, ?? figures, ?? tables, ?? appendices, 4 - used sources.

The purpose of this course project is ?????.

This project consists of 2 sections.

In the first section, ????.

The second section is ????.

.

					КНУ.РМ.123.23.7.3		
Змн.	Арк.	№ документа	Підпис	Дата			
Розробив		Косей			ЗМІСТ	Літера	Аркуш	Аркушів
Перевірив		Кузнєцов						
						КІ-22м		
Н.контроль								
Затвердив								

З М І С Т

ПЕРЕЛІК ТЕРМИНІВ ТА СКОРОЧЕНЬ

Машине Навчання (ML, англ. Machine Learning) - підгалузь штучного інтелекту (AI), яка зосереджується на розробці алгоритмів та моделей, які дозволяють комп'ютерам покращити свою продуктивність у вирішенні конкретних завдань за допомогою навчання на даних, не будучи явно зарограмованими.

Мікросервісна Архітектура (MSA, англ. Microservices Architecture) - це архітектурний стиль проєктування та розробки комп’ютерних систем, який структурує програмне забезпечення як набір невеликих, незалежних служб (мікросервісів), які співпрацюють разом для виконання більшої задачі чи функції. Кожен мікросервіс є окремим застосунком, який може бути розгорнутий, масштабований та керований незалежно від інших мікросервісів у системі**.**

Інтерфейс Програмування Застосунків (API, англ. Application Programming Interface) - це набір правил, який дозволяє програмам спілкуватися та обмінюватися даними між собою.

DevOps (акронім від англ. Development і Operations) - це культура та набір практик у розробці програмного забезпечення, спрямованих на покращення співпраці між розробниками та операторами, автоматизацію процесів та прискорення впровадження програмного забезпечення.

CI/CD (акронім від англ. Continuous Delivery і Continuous Deployment) - це методологія автоматизованої розробки програмного забезпечення, яка об'єднує постійну інтеграцію нового коду розробників з автоматичним випуском готових версій програми, забезпечуючи швидке та безпечне впровадження змін в програмі. Це допомагає покращити якість програми, знизити ризики та забезпечити ефективний розвиток програмного продукту.

SAGA – (акронім від англ. «Segregated Access of Global Atomicity») патерн проєктування в мікросервісній архітектурі , який використовується для керування транзакціями та забезпечення консистентності даних в розподілених сервісах.

ВСТУП

Вебсервіси є важливою частиною сучасного інтернету. Вони дозволяють веб-додаткам взаємодіяти один з одним, незалежно від того, на яких платформах або мовах програмування вони написані. Це робить вебсервіси цінним інструментом для розробників, які хочуть створювати гнучкі та масштабовані вебдодатки.

Вебсервіси використовуються в широкому спектрі додатків, включаючи системи електронної комерції, банки, мобільні оператори та веб-магазини.

Наприклад, системи електронної комерції часто використовують вебсервіси для обробки платежів, доставки товарів та надання клієнтської підтримки, банки використовують вебсервіси для надання фінансових послуг, таких як банківські операції, кредити та депозити, мобільні оператори використовують вебсервіси для управління своїми мережами та надання послуг своїм клієнтам, вебмагазини використовують вебсервіси для каталогізації товарів, обробки замовлень та доставки товарів.

Важливість вебсервісів зростає з кожним роком, тому стає важливим забезпечити масштабованість вебсервісів, тобто здатність системи збільшувати обсяг обробки даних із збільшенням кількості користувачів.

Актуальність роботи. Вебсервіси стають все більш популярними, що призводить до зростання навантаження на них. Для забезпечення високої якості обслуговування вебсервіси повинні бути здатні масштабуватися в залежності від навантаження.

Існують два види масштабування вебсервісів - вертикальне та горизонтальне.

Вертикальне масштабування передбачає збільшення потужності сервера шляхом оновлення апаратної складової, такої як процесор, пам'ять або жорсткий диск.

Горизонтальне масштабування передбачає додавання нових серверів до системи, що дозволяє збільшити обсяг обробки даних в системі.

Вибір методу масштабування системи залежить від конкретних потреб бізнесу та ресурсів, які доступні для розгортання системи. Однак, у більшості випадків, горизонтальне масштабування є більш ефективним, оскільки воно дозволяє збільшувати обсяг обробки даних без збільшення витрат на оновлення апаратної складової сервера. Більше того, горизонтальне масштабування дозволяє забезпечити вищу доступність та стійкість системи, оскільки при відмові одного сервера, інші сервери в системі можуть продовжувати працювати.

Ще одним методом масштабування є використання контейнерів. Контейнери дозволяють виконувати додатки в ізольованому середовищі, що забезпечує масштабованість та стабільність системи.

Більше того, використання контейнерів дозволяє створювати нові контейнери за необхідності та знижує час на їх розгортання.

Ручне масштабування вебсервісів є трудомістким та дорогим, тому застосування штучного інтелекту (ШІ) є перспективним підходом до автоматизації цього процесу.

Застосування ШІ для автоматизації масштабування вебсервісів є активною областю досліджень. Вчені та інженери розробляють нові методи та алгоритми ШІ, які можуть допомогти покращити ефективність і точність автоматизованого масштабування.

Ось кілька конкретних прикладів застосування ШІ для автоматизації масштабування вебсервісів:

Amazon Web Services (AWS) використовує ШІ для своєї системи Auto Scaling, яка автоматично розгортає або вимагає сервери в залежності від навантаження.

Google Cloud Platform (GCP) використовує ШІ для своєї системи Cloud Autoscaling, яка має схожий функціонал, як Auto Scaling від AWS.

Microsoft Azure використовує ШІ для своєї системи Azure Autoscale, яка також має схожий функціонал.

Автомасштабування на основі машинного навчання (**ML**) може бути використано не тільки для вебсервісів, але й для інших типів додатків, таких як мобільні додатки, вбудовані системи та інші. Наприклад, автомасштабування можна використовувати для мобільних додатків, які отримують велику кількість запитів на сервер, наприклад, додатки для онлайн-торгівлі або соціальні мережі.

Автомасштабування вебсервісів на основі машинного навчання дозволяють автоматично налаштовувати ресурси серверів в залежності від навантаження на вебсервіс.

Для цього використовуються алгоритми машинного навчання, які аналізують дані моніторингу сервісу та прогнозують навантаження на майбутнє. Наприклад, можуть використовуватись алгоритми класифікації, регресії, кластеризації або нейронні мережі.

Переваги використання методів та засобів автомасштабування на основі машинного навчання та контейнерів включають:

- висока доступність та стійкість системи завдяки горизонтальному масштабуванню та використанню мікросервісної архітектури;
- ефективне використання ресурсів завдяки автоматичному масштабуванню ресурсів на основі актуального навантаження;

- гнучкість та масштабованість системи завдяки можливості додавати нові сервери та контейнери за необхідності та масштабувати окремі сервіси в залежності від навантаження;
- висока продуктивність та уникнення непередбачуваних відмов завдяки використанню машинного навчання для прогнозування навантаження на майбутнє та автоматичного масштабування ресурсів сервера на основі актуального навантаження.

Хоча методи та засоби автомасштабування на основі машинного навчання та контейнерів мають багато переваг, вони також мають деякі недоліки. Одним з головних недоліків є складність впровадження та налаштування цих засобів. Крім того, використання машинного навчання може вимагати значних обчислювальних ресурсів, що може призвести до збільшення витрат на обладнання та утримання системи. Нарешті, автомасштабування може бути складним при використанні сторонніх сервісів, таких як бази даних або інші засоби, які можуть бути несумісними з автомасштабуванням.

Додатково, варто зазначити, що при використанні методів та засобів автомасштабування вебсервісів, необхідно враховувати питання безпеки. Наприклад, при використанні контейнерів необхідно забезпечити безпеку контейнерів, оскільки вони можуть містити конфіденційні дані. Крім того, варто забезпечити безпеку мережі, на якій працює вебсервіс, оскільки недостатня безпека може призвести до злому системи та втрати конфіденційної інформації.

Крім того, необхідно враховувати той факт, що безпека є постійним процесом та вимагає постійного моніторингу та оновлення методів та засобів захисту. Тому, при виборі методів та засобів автомасштабування вебсервісів, необхідно враховувати питання безпеки та забезпечити постійний моніторинг та оновлення системи з метою забезпечення найвищого рівня безпеки вебсервісу.

При виборі методів та засобів автомасштабування необхідно також враховувати питання економічної доцільності. Використання складних методів та засобів може призвести до значного збільшення витрат на розгортання та підтримку системи. Тому, при виборі методів та засобів автомасштабування необхідно зважати на їхню ефективність та вартість, а також на те, які методи та засоби найбільш оптимальні для конкретної системи.

Мета і завдання дослідження. Метою дослідження цієї роботи є розробка ефективних методів та алгоритмів автомасштабування вебсервісів на основі машинного навчання для забезпечення стабільної роботи сервісів при зміні навантаження.

Завдання включає аналіз існуючих методів автомасштабування, розробку нових алгоритмів, їхню реалізацію та тестування на реальних вебсервісах.

Об’єктом дослідження є веб-сервіси, які використовуються для надання онлайн-послуг, а **предметом дослідження** є методи та засоби автомасштабування цих веб-сервісів з використанням машинного навчання.

Методи дослідженнь - аналіз літературних джерел, експериментальні методи для розробки та валідації алгоритмів автомасштабування, системний аналіз, аналітичні методи, а також методи машинного навчання для створення прогностичних моделей та систем автоматичного управління масштабуванням.

Наукова новизна одержаних результатів полягає в розробці нових алгоритмів автомасштабування вебсервісів на основі машинного навчання, які забезпечують оптимальне використання ресурсів та покращують якість обслуговування користувачів під час зміни навантаження.

Практичне значення отриманих результатів. Результати досліджень дозволять використовувати розобленні алгорітми для ефективного автомасштабування вебсервісів на основі машинного навчання для забезпечення стабільної роботи сервісів при зміні навантаження.

Апробація роботи. Апробація досліджень відбувалась на КІСМ-2023 - XVI Всеукраїнської науково-практичної WEB конференції аспірантів, студентів та молодих вчених «Комп'ютерні інтелектуальні системи та мережі» (21-23 березня 2023 року, м. Кривий Ріг).

1. РОЗДІЛ І. ОГЛЯД АРХІТЕКТУРИ ВЕБСЕРВІСІВ ТА МАШИНОГО НАВЧАННЯ

1.1 ІСТОРІЯ РОЗВИТКУ ВЕБСЕРВИСІВ

Розвиток веб сервісів має довгу і цікаву історію, яка починається з появи Інтернету і його популяризації серед широких мас.

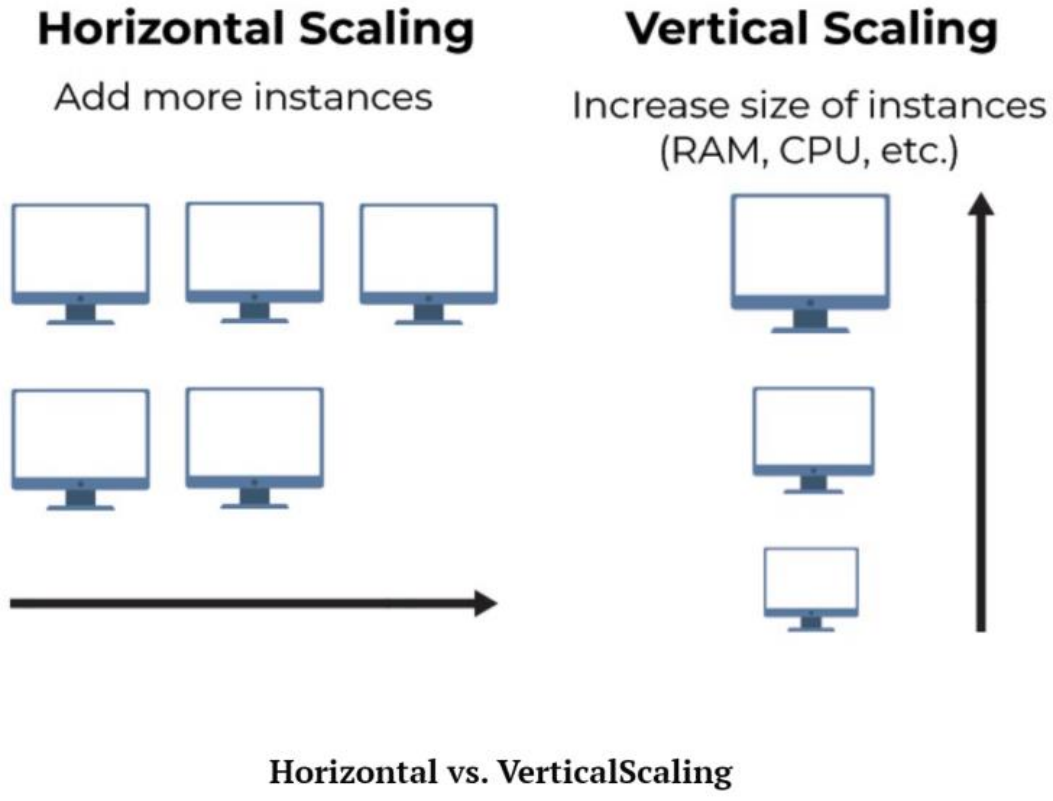
Ось основні етапи розвитку вебсервісів:

- первинні вебсервіси (1990-ті роки): Початок 1990-х років був періодом, коли Інтернет тільки почав набирати популярність. Вебсервіси, такі як електронна пошта, прості веб-сторінки і навігаційні сайти, були поширеними. У цей період розвивалися основні протоколи Інтернету, такі як HTTP (протокол передачі гіпертексту) і HTML (мова розмітки гіпертексту), які визначали стандарти взаємодії між клієнтами і серверами.
- розширення функціональності (2000-ті роки): Протягом 2000-х років вебсервіси почали набувати більшої функціональності і складності. З'явилися соціальні мережі, електронні магазини, блоги, форуми і інші типи веб-додатків. Для розробки таких додатків використовувалися серверні мови програмування, такі як PHP, Java, Ruby, Python і т.д. Також розповсюджувались реляційні бази даних для зберігання і керування інформацією.
- поява хмарними платформ (2010-ті роки): У 2010-х роках з'явилися хмарні платформи для розробки веб-сервісів, такі як Amazon Web Services (AWS), Microsoft Azure і Google Cloud Platform. Ці платформи надають інфраструктуру та інструменти для розгортання, керування і масштабування веб-додатків. З'явилися також сервіси веб-аналітики, оптимізації продуктивності, хмарні сховища даних, мобільні додатки тощо.
- розповсюдження мікросервісної архітектури (з 2010-х років): Одним із сучасних трендів є використання мікросервісної архітектури для розробки вебсервісів. Замість створення монолітних додатків, розробники розбивають функціональність на невеликі незалежні компоненти, які можуть бути розгорнуті, масштабовані і керовані окремо. Це дозволяє більшу гнучкість, швидку розробку і розгортання, а також полегшує інтеграцію з іншими сервісами.
- розширення штучного інтелекту та Інтернету речей (з 2010-х років): Останнім часом вебсервіси почали використовувати штучний інтелект для автоматизації рутинних задач, аналізу даних і покращення користувацького досвіду. Також розробляються веб-сервіси для підключення до Інтернету речей, дозволяючи керувати фізичними пристроями через мережу.

1.2 ВИДИ МАСШТАБУВАННЯ

Існують два основних види масштабування, які використовуються для забезпечення зростання обсягу ресурсів і продуктивності системи. (див. Рисунок 1.1.):

- вертикальне масштабування (**Vertical Scaling**): Цей тип масштабування включає збільшення потужності апаратного забезпечення, такого як процесори, пам'ять і диск. При вертикальному масштабуванні один окремий сервер може обробляти більше завдань або обробляти більш об'ємні дані. Наприклад, збільшення обсягу оперативної пам'яті або перехід на більш потужний процесор.
- горизонтальне масштабування (**Horizontal Scaling**): Цей тип масштабування полягає в додаванні додаткових серверів або вузлів до системи. Він забезпечує розподілення навантаження між багатьма фізичними або віртуальними серверами, щоб забезпечити більшу оброблювальну потужність та доступність. Горизонтальне масштабування часто використовується в хмарних середовищах та системах розподіленого обчислення.



Horizontal vs. VerticalScaling

Рисунок 1.1 - Основні види масштабування вебсервисів

1.3 АРХІТЕКТУРА ВЕБСЕРВИСІВ

Архітектура вебсервисів може бути реалізована за допомогою двох основних підходів: монолітна архітектура та мікросервісна архітектура (див. Рисунок 1.2).

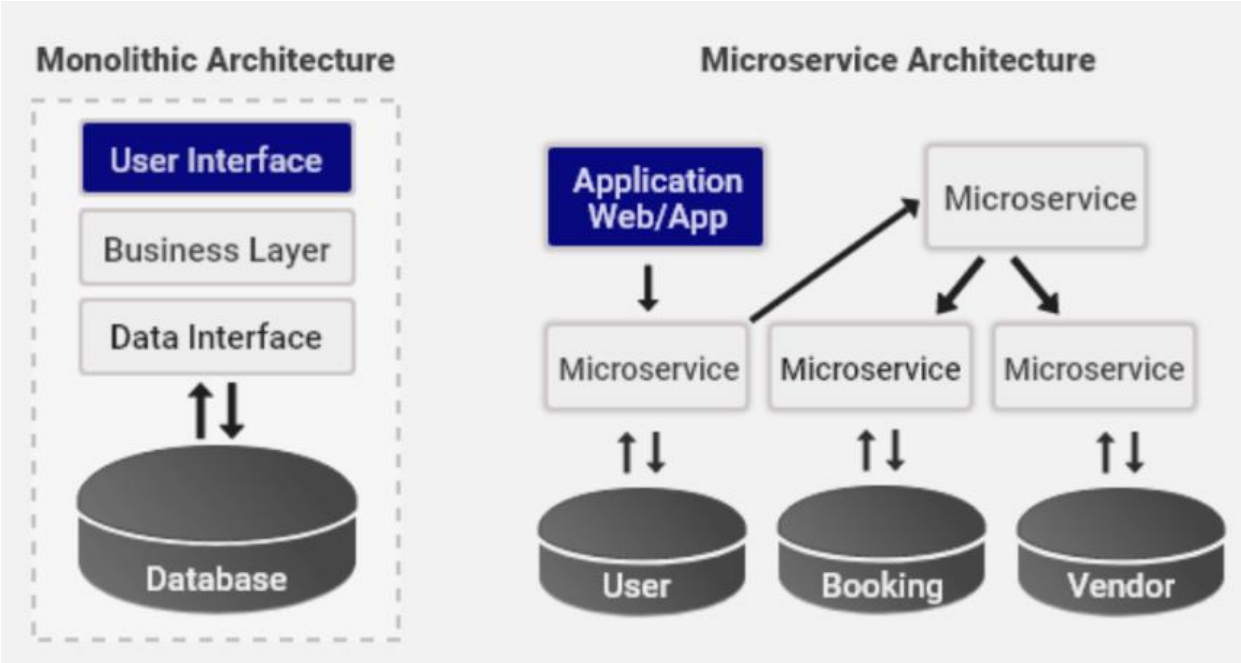


Рисунок 1.2 - Монолітна та мікросервісна архітектура

Монолітна архітектура - це традиційний підхід до розробки веб-додатків, при якому весь функціонал додатка розташований в одному програмному модулі, зазвичай монолітному додатку або монолітному серверу. У монолітній архітектурі весь код, база даних та логіка розташовані в єдиному додатку, що спрощує розробку та розгортання.

Переваги монолітної архітектури включають простоту розробки та тестування, відсутність проблем з взаємодією між компонентами, а також зменшення витрат на інфраструктуру. Однак, монолітні додатки можуть стати важкими для масштабування та розвитку у великих проектах, а також вони можуть бути менш гнучкими у впровадженні нових функцій.

Мікросервісна архітектура (див. Рисунок 1.3) - це підхід, при якому великий веб-додаток розбивається на невеликі незалежні сервіси, які працюють разом із використанням легковагих комунікаційних механізмів, таких як **API**. Кожен сервіс відповідає за обмежений функціонал та має свою власну базу даних.

Мікросервісна архітектура дозволяє забезпечити більшу модульність, масштабованість та гнучкість у розробці веб-додатків. Кожен сервіс може бути незалежно розвиватися, масштабуватися та підтримуватися. Крім того, мікросервіси можуть використовувати різні технології та мови програмування, що дає розробникам більшу свободу вибору технологій.

Однак, мікросервісна архітектура також має свої виклики, включаючи складність взаємодії між сервісами, управління конфігурацією та моніторингом, а також більшу складність у впровадженні та управлінні багатьма сервісами.

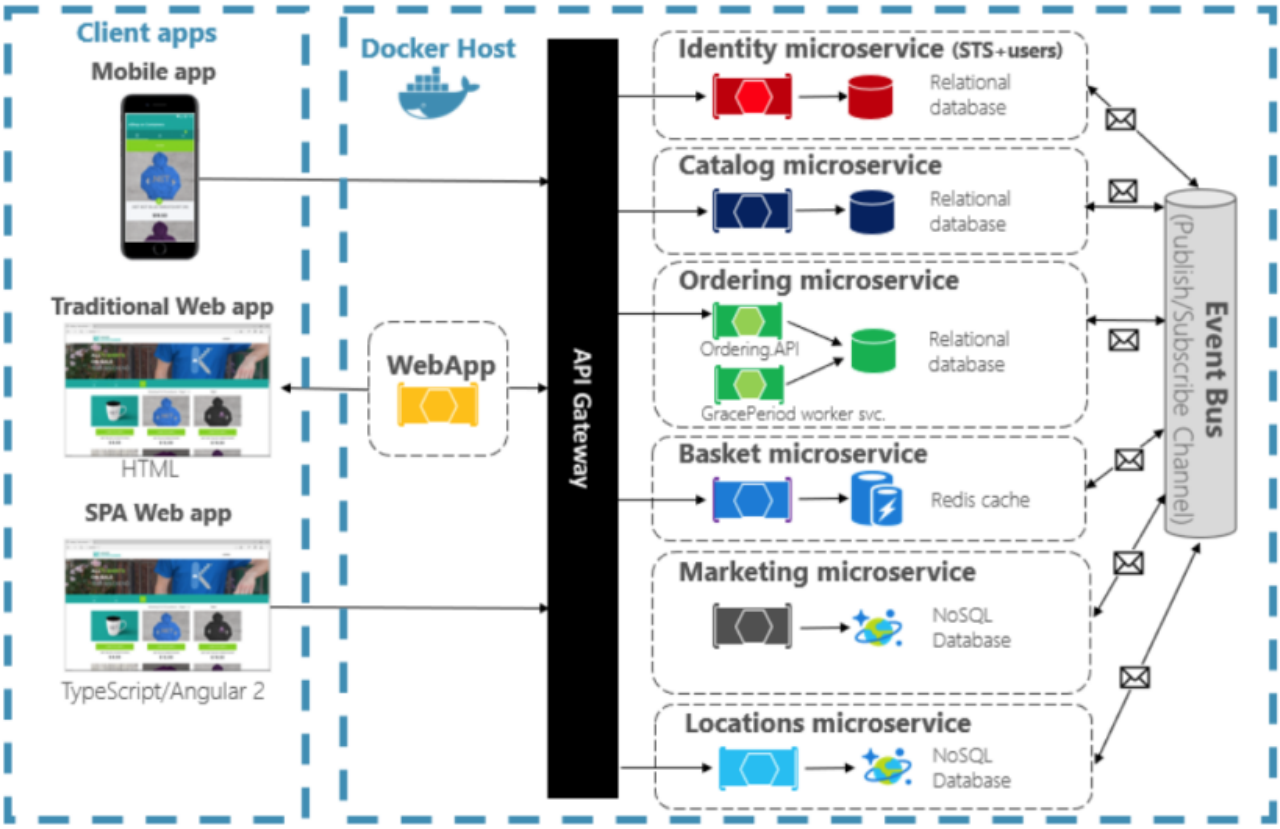


Рисунок 1.3 - Приклад мікросервісної архітектури

Розглянемо більш подрібніше переваги та недоліки MSA в порівнянні з монолітною архітектурою (див. таблицю 1.1).

Мікросервісна архітектура (див. Рисунок 1.3). краще підходить для масштабування, ніж монолітна архітектура, з таких причин:

- **Децентралізація**. Микросервіси розподілені на кілька серверів, що робить їх більш масштабованими, ніж монолітні додатки, які працюють на одному сервері. Це означає, що можна легко додавати або видаляти сервери в міру необхідності, щоб підтримувати потрібну продуктивність.
- **Ізоляція**. Кожний мікросервіс є ізольованим від інших, що означає, що не потрібно масштабувати весь додаток, якщо є навантаження лише на один мікросервіс. Це також означає, що можна масштабувати мікросервіси незалежно один від одного, що може бути корисним для оптимізації витрат.
- **Архітектура шарів**. Микросервіси часто будуються за архітектурою шарів, що дозволяє масштабувати додаток за допомогою різних технологій для кожного рівня.

Наприклад, рівень зберігання можна масштабувати горизонтально, а рівень обробки - вертикально.

Таблиця 1.1 – Порівняння **MSA** з монолітною архітектурою

Характеристика	**MSA - архітектура**	**Монолітна архітектура**
Структура	Високий ступінь автономності. Функції системи розділені на незалежні, слабко зв'язані частини меншого обсягу коду.	Відсутність автономності. Функції системи тісно пов'язані в одному великому блоку коду.
Портативність	Дуже висока	Дуже обмежена портативність
Повторне використання	Високо придатний для повторного використання	Дуже обмежена здатність повторно використовувати код
Модульність і масштабованість	Високо модульний та масштабований	Обмежена модульність та важко масштабувати
Початковий час виходу на ринок	Початковий час виходу на ринок залежить від готовності окремих сервісів. Чим більше повторно використовується код, тим коротший час. Якщо мікросервіси системи розробляються з нуля, час зазвичай довший ніж для монолітної архітектури.	Тривалий час виходу на ринок, особливо у великих системах. Коротший час виходу на ринок у малих та простих системах.
Цикл випуску нових версій та оновлень	Дуже короткий цикл випуску новий версій, швидко впровадження змін та оновлень.	Довгий і зазвичай дуже трудомісткий цикл випуску нових версій та оновлень
Початкові витрати	Зазвичай високі. Залежить від розміру системи. Початкові витрати компенсуються економією операційних витрат	Зазвичай низькі. Стають більшими у великих корпоративних системах
Операційні витрати	Низькі. Легше підтримувати та експлуатувати	Високі. Складно підтримувати та експлуатувати.
Складність	Висока	Низька
Контроль над API	Високий	Низький
Структурна цілісність даних	Децентралізовані бази даних, тому цілісність даних важче підтримувати	Централізована база даних, тому простіше підтримувати цілісність даних у всій системі

Продовження таблиці 1.1 – Порівняння **MSA** з монолітною архітектурою

Характеристика	MSA - архітектура	Монолітна архітектура
Продуктивність	Зазвичай нижча	Зазвичай вища
Безпека	Більше проблем із безпекою	Менше проблем із безпекою
Впровадження в організації розробки програмного забеспечення	Важко впровадити залежно від організаційної структури. Вимагає впровадження гнучкої розробки та **DevOps (CI/CD** та ін.**)**. Може знадобитися організаційна трансформація яка може зайняти багато часу для її досягнення	Легко впровадити. Мінімальна організаційна трансформація потрібна, якщо потрібна взагалі.
Відмовостійкість	Зазвичай вище	Зазвичай нижче

1.4 МІКРОСЕРВІСНА АРХІТЕКТУРА (MSA)

MSA - спосіб побудови складної системи з набору менших додатків, кожен з яких призначений для конкретної обмеженої функції.

Ці малі додатки (або сервіси, або мікросервіси) розробляються незалежно один від одного і можуть функціонувати незалежно один від одного. Кожен мікросервіс має **API**-інтерфейс для спілкування з іншими мікросервісами у системі.

Спосіб організації цих окремих мікросервісів разом визначає функціональні можливості великої системи.

Щоб розуміти цінність мікросервісів та виклики, які виникають при розробці **MSA**, важливо зрозуміти, як мікросервіси взаємодіють та спілкуються один з одним.

Ця взаємодія може бути лінійною або нелінійною.

При лінійній взаємодії (див. Рисунок 1.4) мікросервіси передають дані один одному послідовно, обробляючи їх у системі. Вхідні дані завжди передаються першому мікросервісу, і вихідні дані завжди формуються останнім мікросервісом у системі.

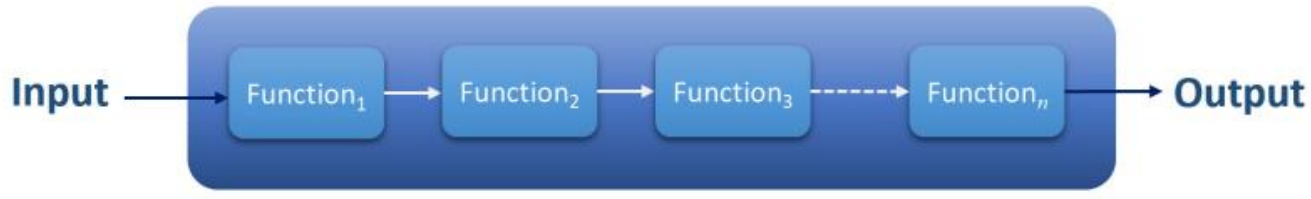


Рисунок 1.4 - Лінійна взаємодія мікросервісів

Практично в більшості існуючих систем взаємодія нелінійна (див. Рисунок 1.5).

У нелінійній мікросервісній взаємодії дані розподіляються між різними функціями у системі. Вхідні дані можуть передаватися в будь-яку функцію системи, і вихідні дані можуть бути згенеровані будь-якою функцією у системі.

Розглянемо нелінійну взаємодію на практичному прикладі в типовій системі електронної комерції (див. Рисунок 1.6).

Кожна функція в процесі "Оформлення замовлення" (**Placing an Order**) представляє собою мікросервіс. Після того, як клієнт розміщує замовлення, спрацьовує виклик **API** до мікросервісу "Додати/Оновити інформацію про клієнта" (**Add/Update Customer Information**), щоб зберегти або оновити інформацію про клієнта. Цей мікросервіс виключно відповідає за управління інформацією про клієнта на основі даних, які він отримує при **API** виклику.

Також одночасно виконується інший **API** виклик до частини процесу "Перевірка платежу" (**Verify Payment**).

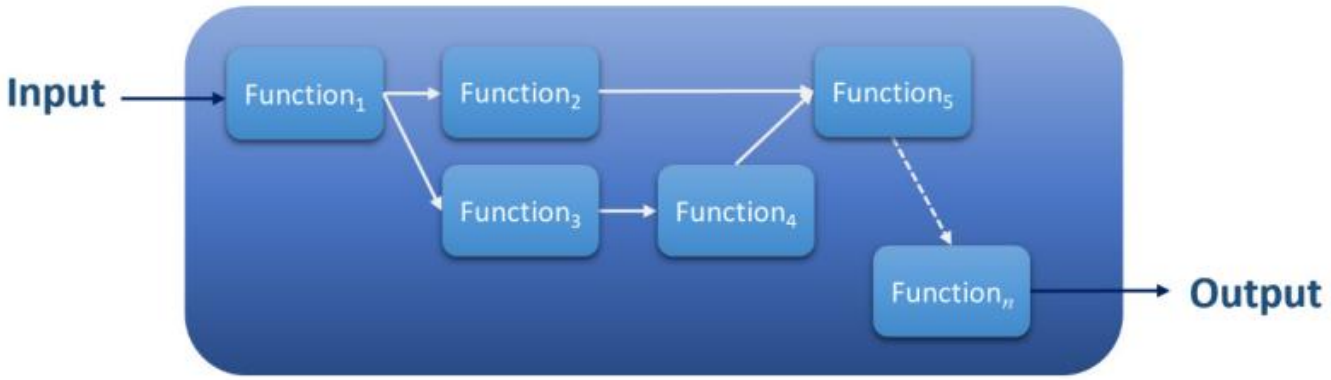


Рисунок 1.5 - Нелінійна взаємодія мікросервісів

Цей виклик буде спрямований до мікросервісу "Обробка платежу через PayPal" (**Process PayPal Payment**) або "Обробка платежу кредитною карткою" (**Process Credit Card Payment**) в залежності від типу платежу, вказаного в **API** виклику. Тут важливо відзначити, як процес перевірки платежу розбито на два різних мікросервіси, кожен з яких спеціально розроблений для конкретної функції оплати.

Це забезпечує гнучкість і переносимість цих мікросервісів до інших частин системи або до іншої системи за необхідності.

Після обробки оплати, інші мікросервіси у системі отримують **API** виклики для виконання замовлення.

Цей приклад показує, наскільки модульним та гнучким може бути проєктування системи **MSA**.

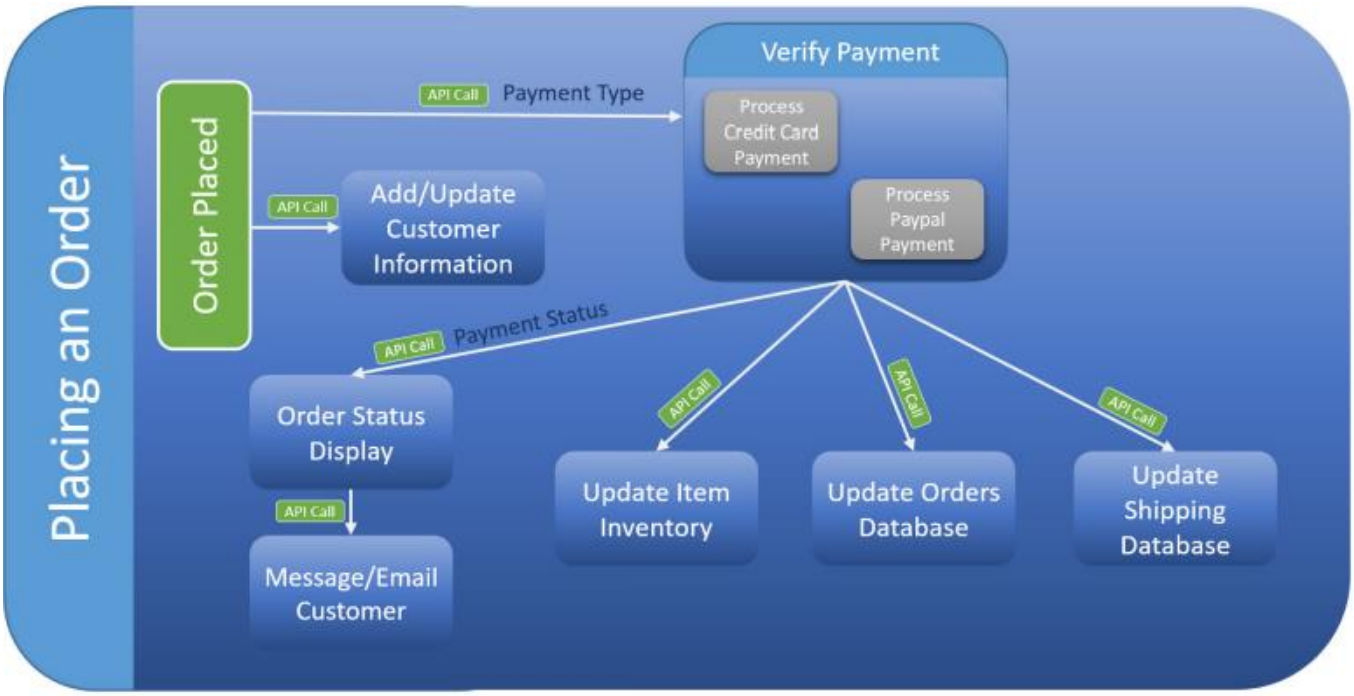


Рисунок 1.6 - Приклад нелінійної взаємодії мікросервісів

1.5 ШТУЧНИЙ ІНТЕЛЕКТ(AI), МАШИННЕ НАВЧАННЯ(ML) ТА ГЛИБОКЕ НАВЧАННЯ(DL)

Незважаючи на нещодавній підйом популярності до штучного інтелекту (**Artificail Inteligance - AI**) та машинного навчання (**Machine Learning - ML**), галузь штучного інтелекту існує з 60-х років XX століття. З появою різних підгалузей **AI** важливо вміти відрізняти їх одна від одної і розуміти, що саме вони означають і що включають в себе.

По-перше, **AI** - це загальна галузь, яка охоплює всі підгалузі, які ми бачимо сьогодні, такі як **ML**, глибоке навчання(**Deep Learning - DL**) (див. рисунок 1.7) та інші.

Люба система, яка сприймає чи отримує інформацію з оточення та виконує дії для максимізації винагороди чи досягнення своєї мети, вважається **AI** системою.

Це дуже поширене сьогодні у робототехніці. Більшість наших машин розроблені так, щоб вони могли збирати дані за допомогою своїх сенсорів, таких як камери, сонари чи гіроскопи і використовувати зібрані дані для ефективного виконання певного завдання.

Цей концепт дуже схожий на те, як поводять себе люди. Люди використовують свої органи почуттів для збору інформації з навколишнього середовища і базуючись на отриманій інформації, виконують певні дії.

AI - це розгалужена галузь, але її можна розбити на різні підгалузі, одну з яких ми знаємо сьогодні як **ML**. Те, що робить **ML** унікальним, полягає в тому, що ця

галузь працює над створенням систем або машин, які можуть навчатися та вдосконалювати свої моделі без явного програмування.

ML робить це, збираючи дані, відомі як тренувальні дані, і намагаючись знайти у цих даних закономірності та патерни для точних передбачень, не будучи явно запрограмованим для цього. У **ML** використовуються різні методи для вивчення даних, і ці методи обираються залежно від проблем, з якими приходиться мати справу.

Підходи, які використовуються у **ML** традиційно поділяють на три великі категорії:

1) навчання з учителем (**Supervised Learning - SL**);
2) навчання без учителя (**Unsupervised Learning - UL**);
3) навчання з підсиленням (**Reinforcement Learning - RL**).

SL допомагає зрозуміти взаємозв'язок між вхідними та вихідними даними. Один із типових прикладів **SL** - це передбачення ціни на будинок у певному місті.

Збираються дані про існуючі будинки, а саме їх характеристики та поточні ціни (навчальний набір), і далі вивчаються закономірності між характеристиками цих будинків та їхніми цінами.

Після цього можна взяти будинок, який не входить до навчального набору, і використати побудовану модель для передбачення його ціни на основі його характеристик.

UL включає в себе вивчення структури даних за допомогою методів групування або кластеризації. Цей метод часто використовується в маркетингових цілях.

Наприклад, магазин хоче розділити своїх клієнтів на різні групи, щоб ефективно пристосовувати свої продукти до різних демографічних груп.

Він може отримати історію покупок своїх клієнтів, вивчити ці дані для визначення закономірностей покупок і рекомендувати певні товари або послуги, що їх можуть зацікавити, тим самим максимізуючи свій прибуток.

Перш ніж розглядати **DL**, що є підгалуззю **ML**, важливо розібратися, що таке штучні нейронні мережі (**Artificial Neural Networks - ANN**).

Взявши за приклад нейрони у мозку, **ANN** є моделями, що складаються з мережі пов'язаних між собою вузлів, також відомих як штучні нейрони. Вони містять набір входів(**Input**), приховані шари(**Hidden Layer**), що з'єднують нейрони, а також вихідний вузол(**Output**) (див. рисунок 1.8).

Кожен нейрон має вхід та вихід, які можуть бути передані по всій мережі. Для розрахунку виходу нейрона береться зважена сума всіх входів, множиться на вагу нейрона та, як правило, додається параметр зсуву.

Цей процес триває до досягнення останнього шару, який є вихідним нейроном. Застосовується нелінійна функція активації, така як сигмоїдна функція, для отримання кінцевого передбачення.

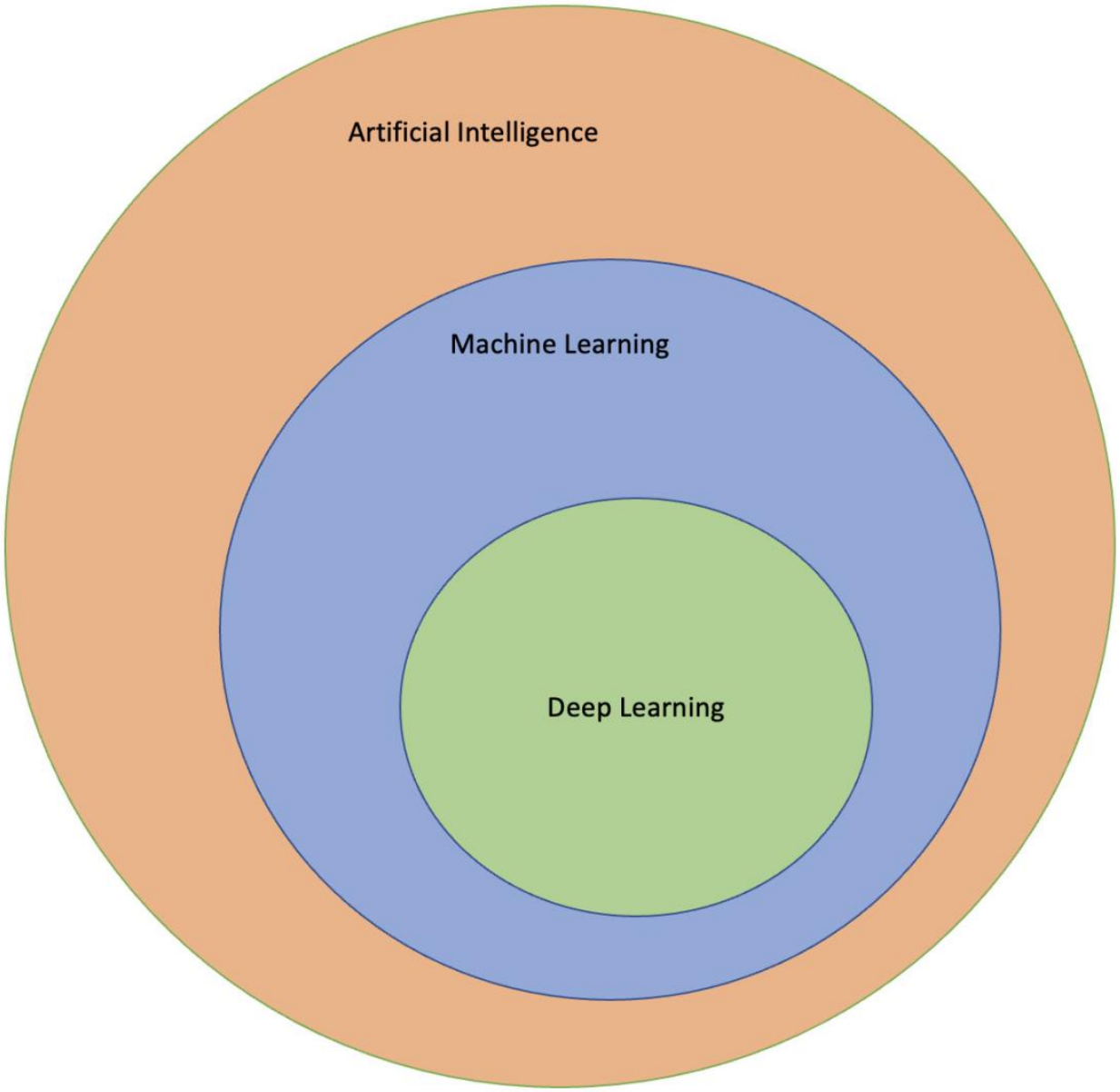


Рисунок 1.7 - Взаємозв'язок між **AI, ML, DL**

Отримане передбачене значення вводиться в функцію витрат. Ця функція показує, наскільки добре вчиться наша мережа.

Значення функції витрат використовується для зворотного розповсюдження помилок через всі шари назад до першого шару, коригуючи ваги нейронів. Завдяки цьому можна створювати потужні моделі, які можуть виконувати завдання, такі як розпізнавання почерку, ігровий **AI** та ін.

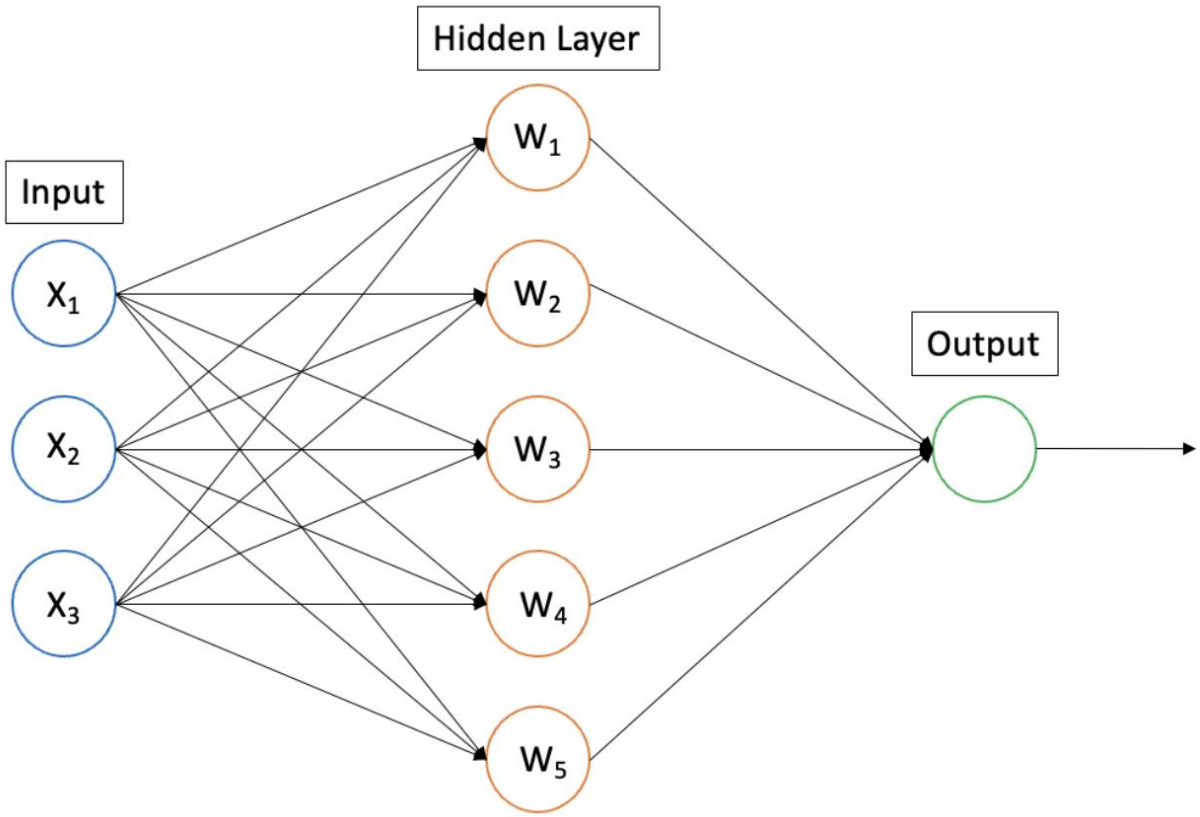


Рисунок 1.8 - Artificial Neural Networks - ANN

В деяких випадках **ANN** можуть бути дуже потужними, але існують серйозні недоліки, які обмежують їхнє застосування:

- **Чорна скринька**: **ANN** можуть бути важкими для розуміння, що ускладнює розуміння їхньої роботи та причин, чому вони роблять певні прогнози. Це може ускладнити відлагодження **ANN** та довіру їх результатам.
- **Обчислювальна вартість**: Навчання **ANN** може бути обчислювально дорогим, особливо для великих і складних мереж. Це може вимагати спеціалізованого обладнання, такого як GPU, і може займати багато часу на навчання.
- **Перенавчання**: **ANN** схильні до перенавчання, що означає, що вони можуть надто добре вивчити навчальні дані та не зможуть узагальнити нові дані. Це може призвести до поганої продуктивності на реальних прикладах.

Саме тут на сцену виходить **DL**.

DL можна класифікувати за такими ключовими ознаками:

- **Ієрархічна композиція шарів**: Замість того, щоб мати в мережі лише повністю з'єднані шари, ми можемо створювати та об'єднувати кілька різних шарів, що складаються з нелінійних і лінійних перетворень. Ці різні шари відіграють роль у видобутку ключових ознак у даних, які інакше було б важко знайти в штучній нейронній мережі **ANN**.

- **Наскрізне навчання**: Мережа починається з методу, який називається видобуванням ознак. Вона аналізує дані та знаходить спосіб групувати надлишкову інформацію та визначати важливі ознаки даних. Потім мережа використовує ці ознаки для навчання та прогнозування або класифікації за допомогою повністю з'єднаних шарів.

Розподілене представлення нейронів: За допомогою видобування ознак мережа може групувати нейрони для кодування більшої ознаки даних. На відміну від **ANN**, жоден окремий нейрон не кодує все. Це дозволяє моделі зменшити кількість параметрів, на яких вона повинна навчитися, зберігаючи при цьому ключові елементи в даних.

DL широко використовується в комп'ютерному зорі.

Завдяки досягненням у технології захоплення фотографій та відео, **ANN** стало дуже важко навчатися та розпізнавати з високою точністю зображення. Починаючи з того, що при використанні зображення для навчання моделі потрібно розглядати кожний піксель як вхідний параметр моделі.

Так, для зображення роздільною здатністю 256x256 є понад 65 000 вхідних параметрів. Залежно від кількості нейронів у повністю з'єднаному шарі, кількість параметрів може сягати мільйонів. Із такою великою кількістю параметрів виникає ймовірність перенавчання і навчання може займати дуже тривалий час.

За допомогою **DL** можна створити групу шарів, які називаються згорткові нейронні мережі (**Convolutional Neural Networks - CNN**). Ці шари відповідають за зменшення кількості параметрів, які потрібно вивчити моделі, при цьому зберігаючи ключові особливості наших даних. За допомогою цих додаткових елементів ми можемо вивчити, як виділяти певні особливості і використовувати їх для навчання нашої моделі з високою ефективністю та точністю (див. рисунок 1.9).

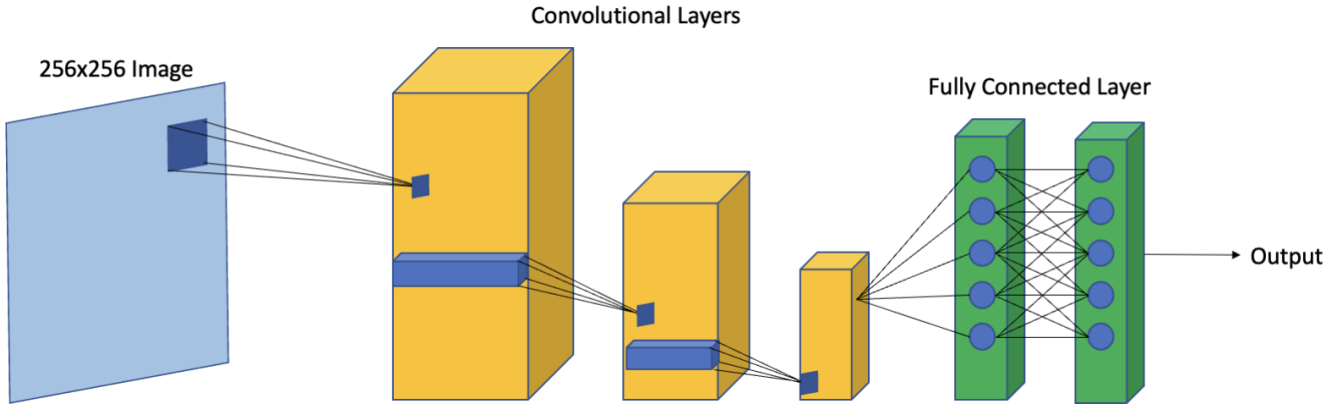


Рисунок 1.9 -. Convolutional Neural Networks – CNN

1.6 ВИСНОВОК ДО РОЗДІЛУ І

У розділі І досліджено основні аспекти розвитку вебсервісів, їхньої структури та вплив **ML** на цю сферу. Зокрема, розділ розглядає історію розвитку вебсервісів, варіанти масштабування, основні концепції архітектури та важливість **MSA**.

Також висвітлено загальні принципи штучного інтелекту, машинного та глибокого навчання та їхній вплив на функціональність вебсервісів. Цей огляд допомагає зрозуміти, які технологічні інновації використовуються для вдосконалення роботи вебсервісів та як машинне навчання змінює їхні можливості.

Описаний розділ надає загальне уявлення про основні принципи та тенденції розвитку вебсервісів та вплив машинного навчання на цю галузь. Це є важливою основою для подальшого дослідження та впровадження інновацій у сфері вебсервісів та їхнього зв'язку з машинним навчанням.

2. РОЗДІЛ ІІ. ВИБІР ТЕХНОЛОГІЙ ТА МОДЕЛЕЙ ML ДЛЯ ЗАСТОСУВАННЯ В MSA

2.1 ПАТЕРНИ ПРОЄКТУВАННЯ В MSA

MSA надає багато переваг у вигляді гнучкості, масштабованості та незалежності сервісів, але також вводить складнощі пов'язані з управлінням цими розподіленими системами.

Саме тому розробники стикаються з викликами і проблемами, такими як: відсутність централізованого контролю, складність у відстеженні та відлагодження помилок (debugging), а також вирішення проблем зі збереженням консистентності даних між мікросервісами.

Використання патернів проєктування у **MSA** стає надзвичайно важливим, оскільки вони пропонують готові та перевірені рішення для таких проблем.

Патерни допомагають знаходити оптимальні шляхи управління консистентністю даних, реалізацією зв'язків між сервісами, моніторингом та масштабуванням.

Вони сприяють створенню стабільних, ефективних мікросервісних застосунків, дозволяючи розробникам зосередитися на функціональності застосунків, замість того, щоб вирішувати складні технічні завдання з нуля.

2.2 ПАТЕРН SAGA

У **MSA**, де компоненти системи розділені на окремі сервіси можуть виникати ситуації, коли один сервіс успішно виконав зміни в базі даних, тоді як інший сервіс, що залежить від цих змін, ще не оновив свої дані. Це може призвести до неконсистентного стану даних в системі.

Крім того, можуть виникати проблеми під час виконання розподілених транзакцій (див. рисунок 2.1), коли частина транзакції виконана успішно, а інша - ні. Це може бути викликано неполадками в мережі, помилками у програмному забезпеченні або іншими непередбаченими обставинами.

Для вирішення проблем з розподіленими транзакціями використовують патерн проєктування **SAGA ("Segregated Access of Global Atomicity").**

Цей патерн базується на ідеї розбиття великої транзакції на менші підтранзакції, які виконуються окремо в кожному мікросервісі (див. рисунок 2.2).

Якщо одна з підтранзакцій виявляється невдалою, то застосовується компенсуюча транзакція, щоб відмінити або зкоригувати зміни, здійснені попередніми транзакціями.

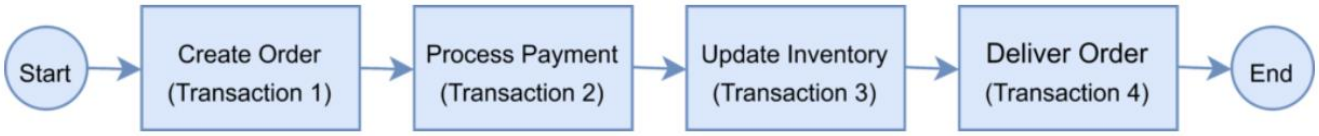


Рисунок 2.1 - Приклад розподіленої транзакції
де межі транзакції перетинаються кількома сервісами та базами даних

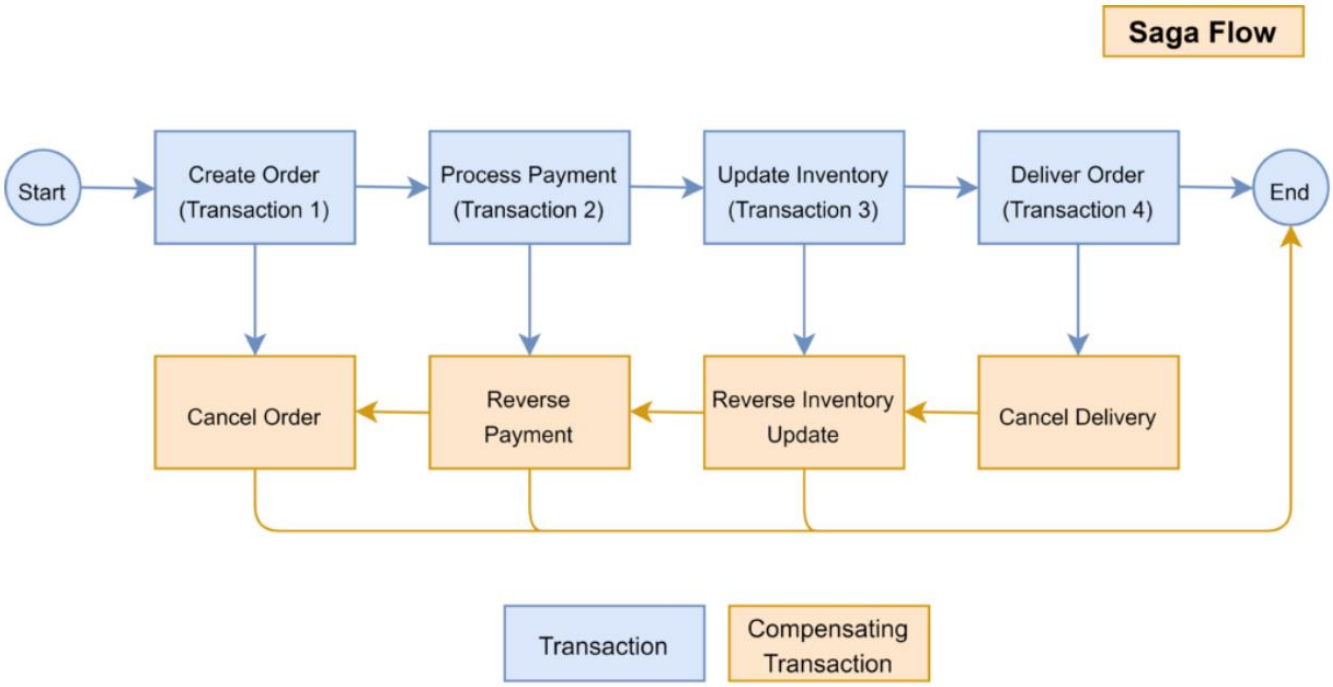


Рисунок 2.2 - Приклад розподіленої транзакції
з використанням патерну **SAGA.** Потік **SAGA (Saga Flow)**

Координатор виконання **SAGA (Saga Execution Coordinator - SEC)** є центральним компонентом для виконання потоку **SAGA** (див. рисунок 2.3).

Він містить журнал **SAGA (SAGA Log)**, який фіксує послідовність подій розподіленої транзакції.

Існують два різновиди впровадження патерну **SAGA: Choreography** та **Orchestration**.

У патерні **SAGA Choreography** кожен мікросервіс, що бере участь у транзакції, публікує подію, яку обробляє наступний мікросервіс.

Це взаємодія сервісів, де кожен сервіс сприймає події від інших та реагує на них, керуючи ходом транзакції.

Для використання цього патерну необхідно вирішити, чи буде мікросервіс частиною **SAGA**. Відповідно, мікросервіс повинен використовувати відповідний фреймворк для реалізації **SAGA**. У цьому патерні координатор виконання **SAGA** може бути вбудований у мікросервіс або виступати як самостійний компонент.

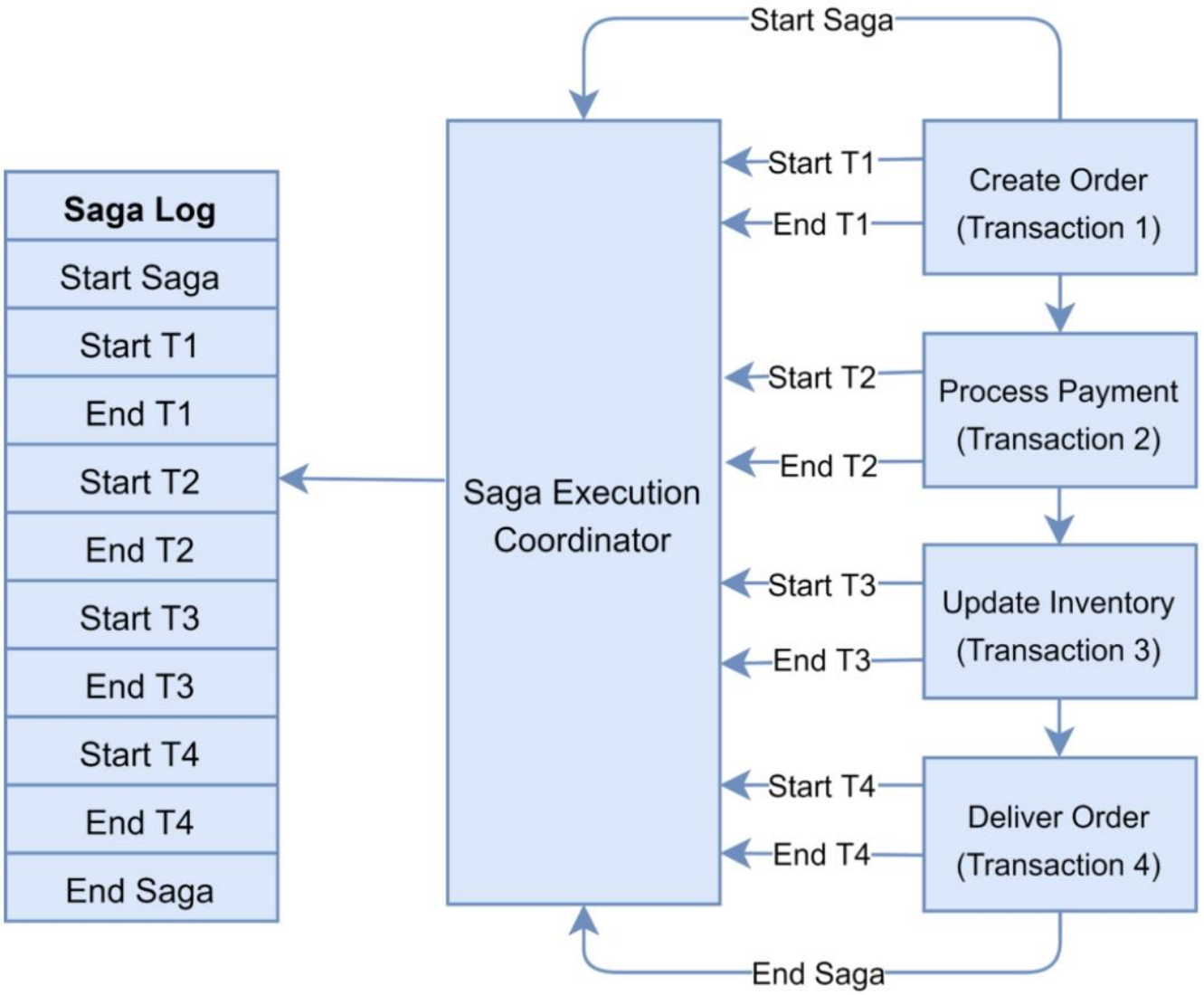


Рисунок 2.3 - Компонент **Saga Execution Coordinator.**

У **SAGA Choreography** потік вважається успішним, якщо всі мікросервіси успішно завершують свої локальні транзакції і жоден з них не повідомляє про невдачу (див. рисунок 2.4).

У разі невдачі мікросервіс повідомляє про неї координатор виконання **SAGA** (**SEC**), через який викликається відповідні компенсуючі транзакції (див. рисунок 2.5).

Якщо виклик компенсуючої транзакції невдалий, то вона повторюється, поки не буде успішно виконана, при чому компенсуюча транзакція повинна бути ідемпотентною та мати можливість до повторного виклику.

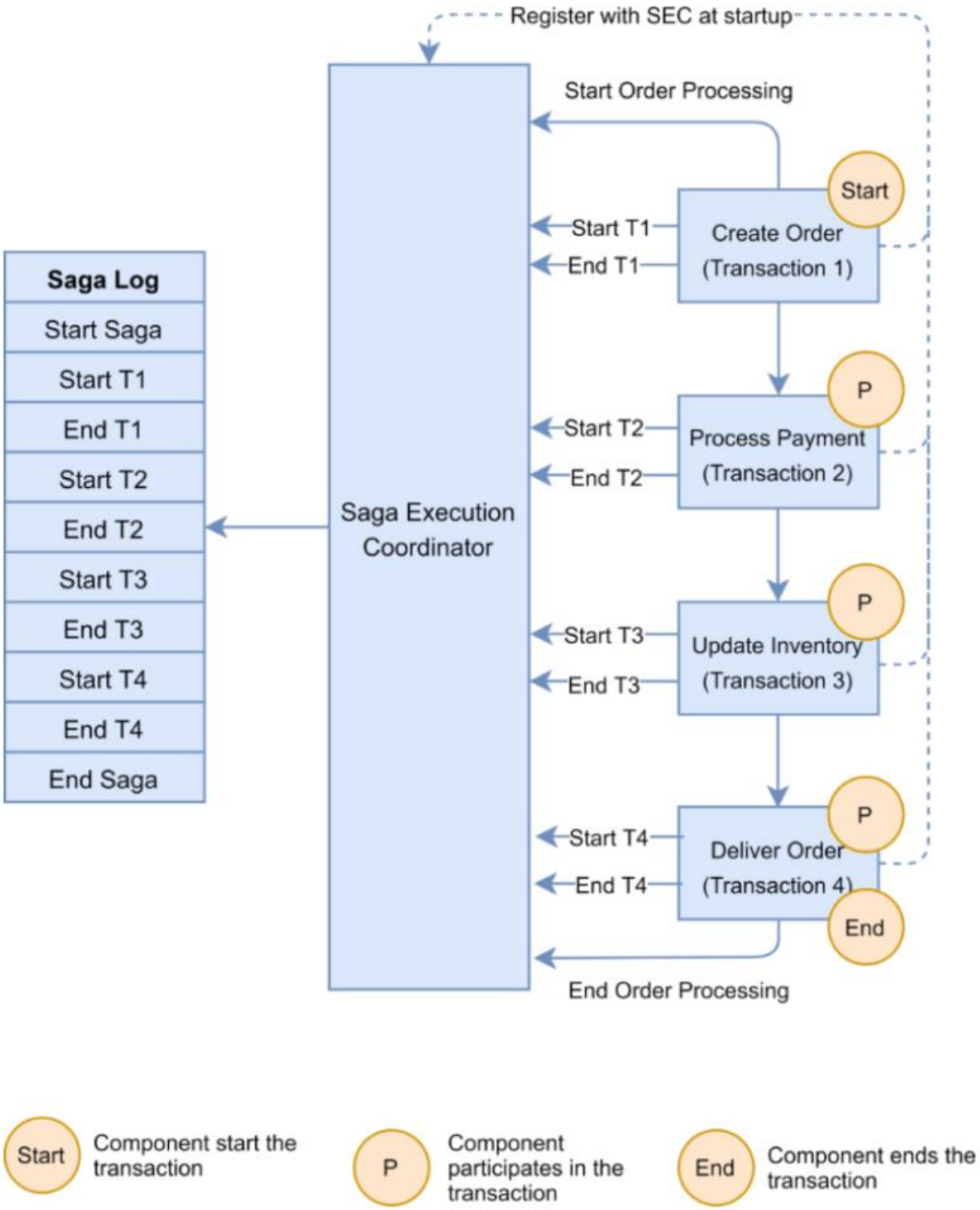


Рисунок 2.4 - SAGA Choreography діаграма ілюструє успішний хід виконання транзакції**.**

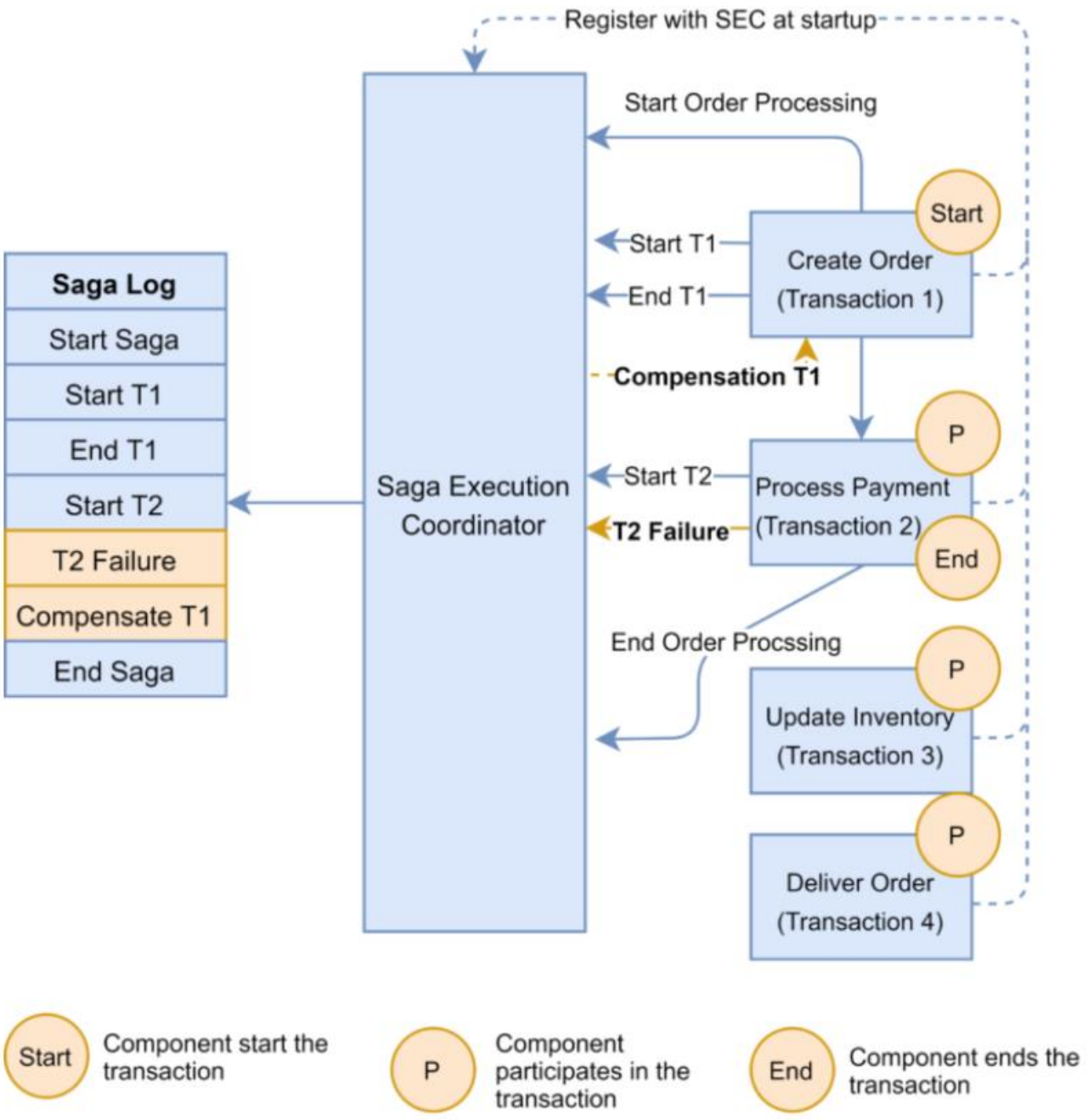


Рисунок 2.5 - SAGA Choreography діаграма ілюструє невдалий хід виконання транзакції та запуск компенсуючої транзакції**.**

Перелік фреймворків які реалізують **SAGA Choreography** в **MSA**:

- **Axon Saga** - невеликий фреймворк, який широко використовується з мікросервісами на основі **Spring Boot**;
- **Eclipse MicroProfile LRA** - реалізація розподілених транзакцій **SAGA** для HTTP-транспорту, заснована на принципах **REST**;

- **Eventuate Tram Saga** - фреймворк **SAGA Choreography** для мікросервісів, побудованих на **Spring Boot** і **Micronaut**;
- **Seata** - відкритий розподілений фреймворк для транзакцій з високою продуктивністю та простими службами розподілених транзакцій.

У патерні **SAGA Orchestration** один єдиний диригент (оркестратор) відповідає за управління загальним статусом розподіленої транзакції (див. рисунок 1.12).

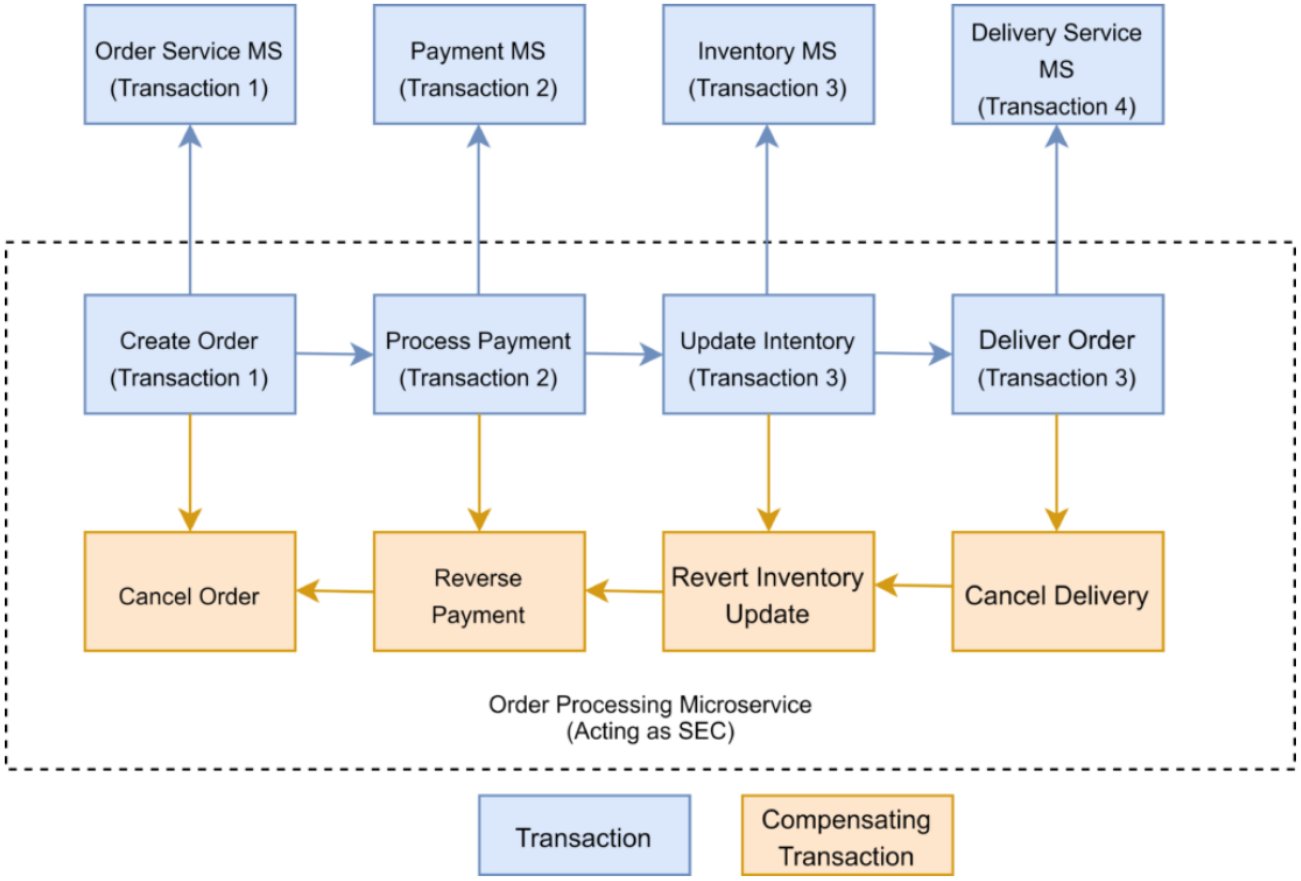


Рисунок 2.6 - SAGA Orchestration діаграма ілюструє хід виконання розподіленої транзакції**.**

Перелік фреймворків які реалізують **SAGA Orchestration** в **MSA**:

- **Camunda** - Java-фреймворк, який підтримує стандарт Business Process Model and Notation (BPMN) для автоматизації робочих процесів.
- **Apache Camel** - надає реалізацію паттерну **SAGA Orchestration** в рамках Enterprise Integration Patterns (EIP).

Патерн **SAGA Orchestration**, незважаючи на централізований контроль, може вирішувати складні сценарії транзакцій, проте він може бути повільнішим через централізований підхід.

Патерн **SAGA Choreography** дозволяє мікросервісам взаємодіяти безпосередньо один з одним, що може бути швидше та гнучкіше, але вимагає правильної синхронізації для уникнення неконсистентності даних у складних транзакційних сценаріях.

2.3 ПАТЕРНИ CRUD та CQRS

У традиційних системах, особливо в монолітних додатках, дуже поширене використання спільної реляційної бази даних, розгорнутої на серверній частині та доступної з клієнтського додатку. Доступ до цієї централізованої бази даних здійснюється за допомогою операцій **Create-Read-Update-Delete (CRUD)** (див. рисунок 2.7).

Проте в сучасних складних **MSA** додатках особливо під час масштабування додадку, ця традиційна реалізація створює проблему. Під час обробки кількох **CRUD**-запитів до бази даних створюються з'єднання таблиць, що призводить до високої ймовірності блокування бази даних. Блокування таблиць викликає затримки та конкуренцію за ресурси, що значно впливає на загальну продуктивність системи.

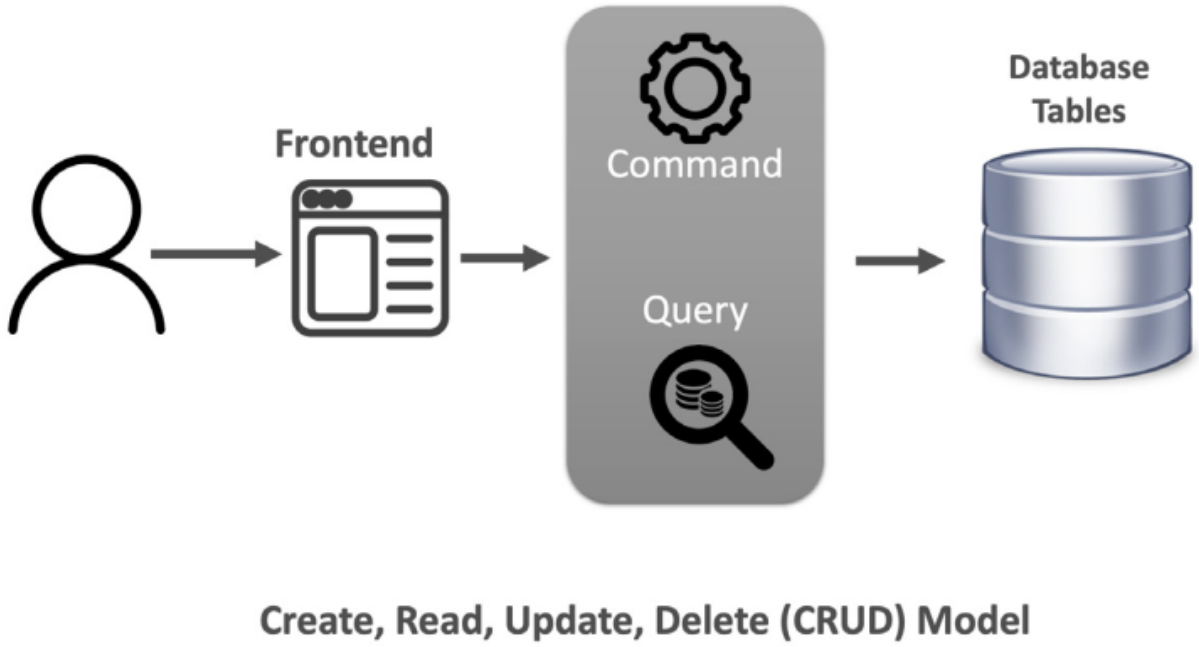


Рисунок 2.7 - CRUD - патерн**.**

Складні запити містять велику кількість з'єднань таблиць і можуть блокувати таблиці, перешкоджаючи будь-яким операціям запису або оновлення до завершення запиту та розблокування таблиць базою даних.

Операції читання бази даних зазвичай виконуються в кілька разів частіше, ніж операції запису, а в системах з інтенсивним виконанням транзакцій ця проблема може посилюватися.

Паттерн **CQRS** (**Command Query Responsibility Segregation**) розділяє один об'єкт на два об'єкти. Отже, замість того, щоб виконувати обидві команди (**Command**) та запити (**Query**) для одного об'єкта, ми розділяємо цей об'єкт на два об'єкти – один для команди, а другий для запиту. Команда – це операція, яка змінює стан об'єкта, а запит не змінює стан системи - просто повертає результат. (див. рисунок 2.8).

Об'єктом тут є база даних (**Database**), і цей розділ бази даних може бути фізичним або логічним.

Проте найкращою практикою є дві фізичні бази даних для CQRS, але все одно можливий варіант використання однієї фізичної базу даних і для команд і для запитів. Наприклад, можна розділити базу даних на два логічні представлення – одне для команд, а інше для запитів.

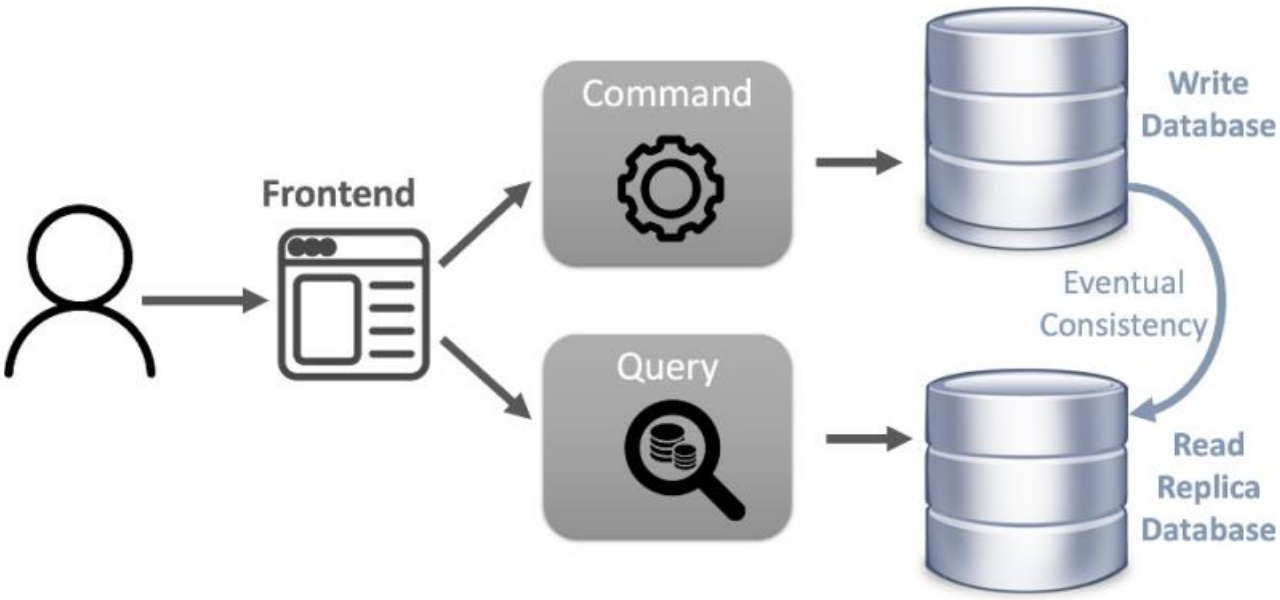


Рисунок 2.8 - CQRS - патерн**.**

Під час використання двох фізичних баз даних у **CQRS** створюється репліка з основної бази даних. Репліку, звичайно, потрібно буде синхронізувати з основною базою даних для узгодженості даних. Синхронізацію можна виконати за допомогою впровадження подійно-орієнтованої архітектури (англ. **Event-driven architecture - EDA**), де брокер повідомлень обробляє всі системні події. Репліка підписується на брокер повідомлень, і кожного разу, коли основна база даних публікує подію в брокері повідомлень, база даних репліки синхронізує цю конкретну зміну.

Між точним часом, коли основна база даних була фактично змінена, і моментом, коли ця зміна відображається в репліці, буде затримка; дві бази даних не будуть на 100% узгодженими протягом цього періоду, але вони будуть узгодженими з плином часу після синхронізації змін. У **CQRS** ця синхронізація називається синхронізацією з остаточною узгодженістю.

При застосуванні **CQRS** у **MSA** значно скорочується затримка операцій з базою даних, а отже, значно підвищується продуктивність зв'язку між окремими службами, що призводить до загального підвищення продуктивності системи.

Тип бази даних, що використовується для **CQRS**, може змінюватися залежно від бізнес вимог до конкретної служби в **MSA**.

Це цілком може бути реляційна база даних (**RDB**), документо-орієнтована база даних, графова база даних, або інший тип **NoSQL** баз даних.

2.4 ПАТЕРН API GATEWAY

Як було раніше розглянуто у підрозділі 1.4 мікросервіси можуть спілкуватися напряму один з одним без потреби у централізованому менеджері (див. рисунок 2.9).

Проте, оскільки система **MSA** розвивається і стає більш зрілою та кількість мікросервісів поступово збільшується, пряме спілкування між мікросервісами може призвести до великих накладних витрат, особливо для викликів, які потребують кількох повторних переходів між споживачем **API** та постачальником **API**.

Згідно з принципом автономності мікросервісів, кожен мікросервіс може використовувати свій технологічний стек і може спілкуватися з використанням іншого контракту **API**, ніж інші мікросервіси в одній системі **MSA**. Наприклад, один мікросервіс може розуміти лише **RESTful API** з **JSON**-структурою даних, тоді як інші можуть спілкуватися лише за допомогою **Thrift API** або **Avro API**.

Крім того, місцезнаходження (IP-адреса та порт для прослуховування) активних інстанцій мікросервісів може динамічно змінюватися у системі на основі мікросервісів. Тому системі потрібен механізм для ідентифікації місця, на яке споживач **API** може направляти свої виклики.

Також можуть виникнути ситуації, коли потрібно пов'язати систему **MSA** з застарілими системами, такими як мейнфрейм AS400 тощо.

Все це вимагає додаткового вбудованого коду в кожен мікросервіс який допоможе мікросервісам зрозуміти застарілі API, виявити мережеве розташування (**Service Discovery**) інших мікросервісів у системі та зрозуміти потреби кожного мікросервісу загалом.

Але такий підхід порушує принцип автономності мікросервісів та призводить до ускладнення системи, коли з кожним новим мікросервісом, який додається до системи, зростає кількість можливих взаємодій між мікросервісами.

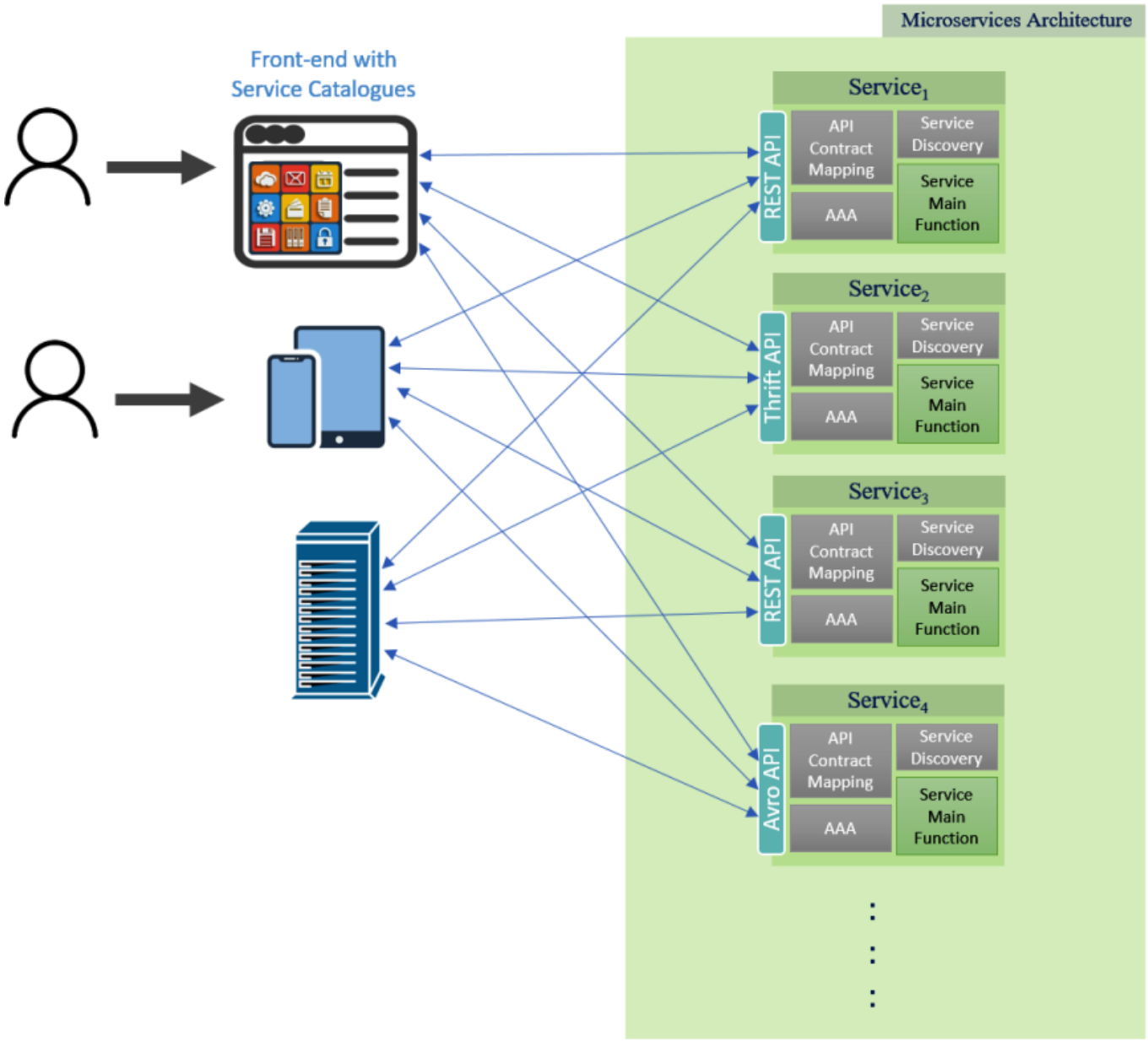


Рисунок 2.9 - MSA система з прямою взаємодією між сервісами.

Ефективнішим підходом до вирішення цих проблем є використання патерну **API GATEWAY** (див. рисунок 2.10).

API GATEWAY через який всі мікросервіси спілкуються один з одним, отримує виклики **API** від споживачів, а потім перетворює отримані дані в структуру даних і протокол, які постачальники API можуть зрозуміти та обробити.

Використовуючи **API GATEWAY** значно скорочується пряме одностороннє спілкування між службами.

Крім того мікросервіси звільняються від коду для виконання задач перетворення контрактів **API(API Contract Mapping)**, знаходження інших сервисів (**Service Discovery**) та задач пов'язаних з контролем та наданням доступу до ресурсів

системи (**Authentication-Authorization-Accounting - AAA**). Натомість ми перекладаємо відповідальність за виконання цих задач на **API GATEWAY**, що ще більше знижує накладні витрати на код і робить мікросервіси максимально легкими та незалежними.

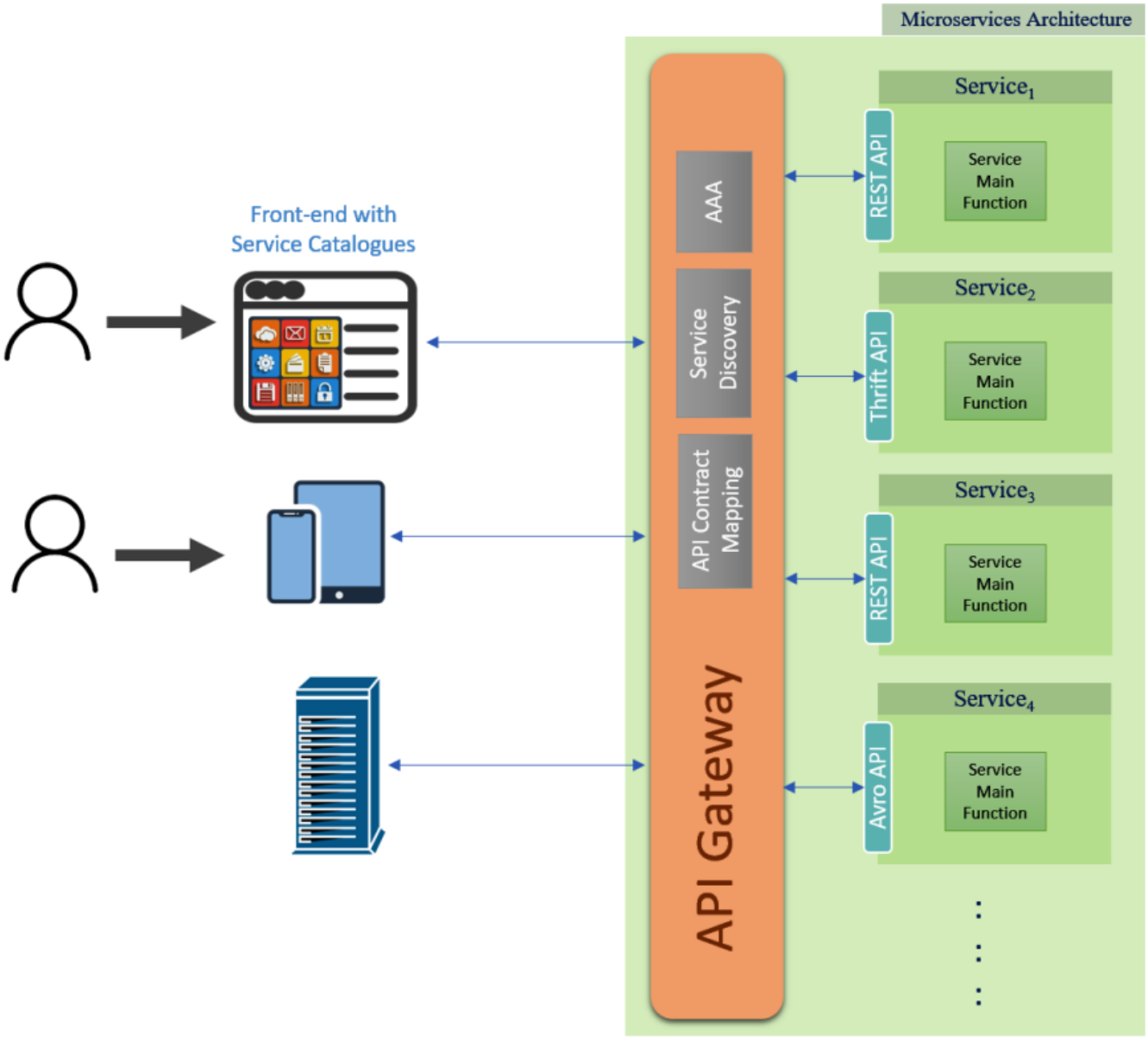


Рисунок 2.10 - MSA система з **API GATEWAY**.

Доступність системи **MSA** залежить від доступності **API GATEWAY**. Тому необхідно розробляти, розгортати **API GATEWAY** як високопродуктивну та високо доступну критично важливу службу.

API GATEWAY може бути розгорнутий як автономна служба, яка є частиною системи **MSA**.

Основні функції **API GATEWAY** наведено нижче:

- мінімізувати кількість викликів **API** між мікросервісами, що робить міжсервісну комунікацію MSA набагато ефективнішою;
- мінімізувати залежності **API** та критичні зміни коду. У системі **MSA** без **API GATEWAY**, якщо з будь-якої причини один із постачальників **API** змінює свій **API**, скоріш за все, відбудеться критичні зміна коду. Це означає, що нам потрібно буде внести зміни до кожного мікросервісу, який спілкується з постачальником API. Використовуючи **API GATEWAY**, зміна в API постачальника буде обмежена лише API-шлюзом, щоб відповідати контракту API між постачальником і споживачами.
- конвертація контрактів **API**(**API Contract Mapping**), що звільняє мікросервіси від вбудовування коду конвертації в їх основні функції;
- запускати механізм виявлення інших сервисів (**Service Discovery**) і звільняти клієнтів від реалізації та запуску цієї функції.
- виступати в якості точки входу для зовнішніх викликів клієнтів до системи **MSA**;
- балансування навантаження викликів **API** між різними екземплярами високодоступних мікросервісів та зняття навантаження з мікросервісів у випадках великого трафіку;
- забезпечення кращої безпеки за допомогою обмеження раптового зростання викликів API під час атак **Distributed Denial of Service (DDoS)** та інших типів кібератак;
- аутентифікація та авторизація користувачів для доступу до різних компонентів системи **MSA**;
- надання комплексної аналітики для забезпечення глибокого аналізу метрик та логів системи, що може сприяти подальшому вдосконаленню дизайну та продуктивності системи.

Незважаючи на всі переваги та функції **API GATEWAY**, в системі **MSA** існують певні недоліки:

- самий очевидний недолік – складність, чим більше протоколів та структур даних **API** ми маємо в системі, тим складніше стає **API GATEWAY**;
- надійна робота системи **MSA** залежить від продуктивності та доступності **API GATEWAY**, що може призвести до небажаного вузького місця в надійності та продуктивності системи;
- введення додаткового посередницького компонента, такого як **API GATEWAY** в між мікросервісній комунікації може збільшує час відповіді мікросервисів.
-

2.5 ПАТЕРН CIRCUIT BREAKER

Ще однією важливою проблемою у системах **MSA** є стабільність та надійність виконання робочих процесів. Патерни типу **SAGA**, які розглядалися у підрозділі 2.2,

використовуються для забезпечення того, щоб всі транзакції у конкретному робочому процесі виконувались або всі успішно, або жодна з них. Але цього недостатньо для надійного функціонування системи **MSA.**

Розглянемо сценарій (див. рисунок 2.11), де викликаний мікросервіс занадто повільно реагує на виклики **API**. Запити успішно виконуються, але відповіді від мікросервісу завершуються таймаутом. Користувач мікросервісу може припустити відмову виконання, і відповідно повторити операцію, що може бути дуже проблематично.

Рисунок 2.11 - Приклад взаємодії між мікросервісами з занадто повільним часом відповіді

У системах **MSA** при створенні мікросервісу відділяються обмежені ресурси та потоки, щоб уникнути того, що один певний мікросервіс захопить усі ресурси системи.

Розглянемо інший сценарій (див. рисунок 2.12), де мікросервіс **Inventory** є частиною робочого процесу і не обробляє та не відповідає на виклики **API** з будь-якої причини. У цьому випадку обидва мікросервіси **Order** і **Payment** будуть продовжувати чекати підтвердження від мікросервісу **Inventory**, перш ніж розпочати звільнення своїх ресурсів.

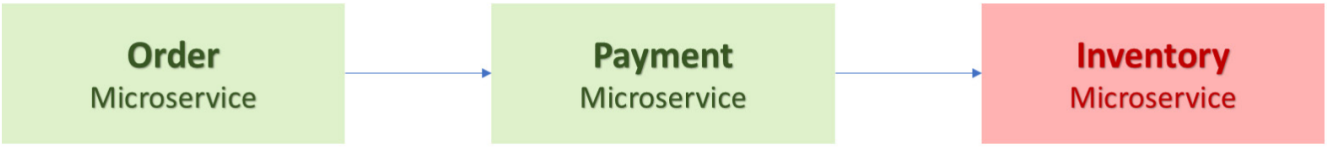


Рисунок 2.12. Приклад взаємодії між мікросервісами коли один з мікросервісів не відповідає на виклики

Такі сценарії в системах **MSA** можуть викликати доміно-ефект, що веде до каскадної відмови кількох мікросервісів, що в свою чергу, може спричинити загальну відмову системи.

Для запобігання каскадній відмові системи використовується патерн **Circuit Breaker** (див. рисунок 2.13).

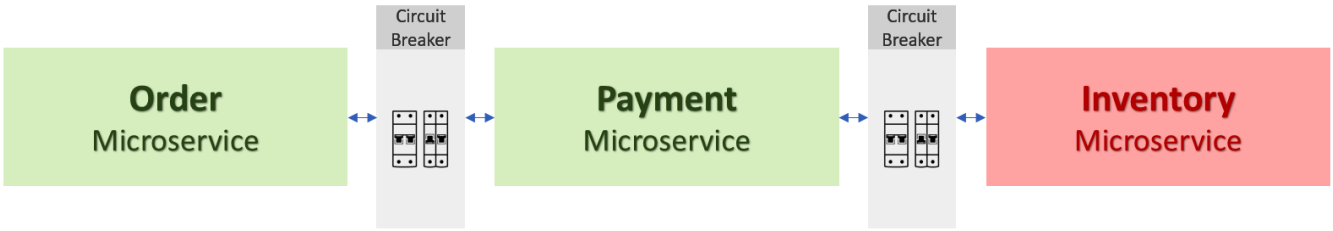


Рисунок 2.13 - Приклад взаємодії між мікросервісами коли один з мікросервісів не відповідає на виклики з використанням **Circuit Breaker**

Circuit Breaker слідкує за продуктивністю мікросервісу за допомогою реальних метрик трафіку. Він аналізує параметри, такі як час відповіді та частка успішних відповідей, і визначає стан мікросервісу в реальному часі.

Якщо мікросервіс перестає відповідати на виклики, **Circuit Breaker** переходить у відкритий стан (**open state**) та негайно відправляє помилку споживачам мікросервісу.

Коли **Circuit Breaker** вважає, що з мікросервісом починають виникати проблеми, він все ще моніторить та оцінює його стан, але він переходить в напіввідкритий стан (**half-open state**), дозволяючи проходити лише обмеженій кількості запитів до контрольованого мікросервісу.

Якщо вимикач потім виявляє нормальну реакцію та поведінку від мікросервісу, він повертається в закритий стан (**closed state**) і всі запити знову направляються на обробку до мікросервісу.

Circuit Breaker не запобігає відмові мікросервісу, який моніториться, а лише запобігає каскадній відмові.

Circuit Breaker не потрібен для кожного окремого мікросервісу в системі **MSA,** а лише на тих мікросервісах, які можуть спричинити каскадну відмову. Тобто архітектори системи повинні визначити, які мікросервіси потребують захисту.

Circuit Breaker може бути реалізований, як окремий мікросервіс або бути частиною **API GATEWAY**. Те, як вимикач буде реалізований, залежить від бізнес вимог до системи, а також від того, які патерни використовуються в архітектурі.

2.6 БІБЛІОТЕКИ ДЛЯ СТВОРЕННЯ МОДЕЛЕЙ ML В PYTHON

Існує багато мов програмування, які використовуються для створення моделей **ML:** MATLAB, R і Python.

Серед них Python став найпопулярнішою мовою програмування у галузі машинного навчання завдяки своїй універсальності та великій кількості бібліотек, які полегшують створення моделей машинного навчання.

NumPy є ключовим бібліотекою при розробці моделей машинного навчання на Python, тому що при створені моделей є багато роботи з великими багатовимірними матрицями. Оскільки основна частина роботи вимагає перетворень, розбиття та виконання складних математичних операцій над цими матрицями, **NumPy** надає швидкі та ефективні інструменти для цього.

Matplotlib є важливою бібліотекою для візуалізації результатів та даних моделі. Вона надає простий спосіб побудови графіків, починаючи від простих лінійних графіків до більш складних, таких як контурні графіки та тривимірні графіки. Популярність цієї бібліотеки пояснюється її легкістю взаємодії з **NumPy**.

Бібліотека **Pandas** стала популярною в спільноті Python завдяки зручності та універсальності при роботі з даними, збереженими у CSV-файлах, що стало останнім трендом. Ця бібліотека використовується для аналізу даних, зберігаючи їх у табличному форматі. Користувачам надаються прості функції для попередньої обробки та маніпулювання даними, що дозволяє їм адаптувати дані до своїх потреб. Крім того, **Pandas** є корисною при роботі з часовими рядами, що є важливим аспектом при побудові прогнозуючих моделей.

Бібліотеки **TensorFlow** та **Keras** є основою для побудови моделей глибокого навчання. Хоча обидві можуть використовуватися окремо, **Keras** використовується як інтерфейс для фреймворку **TensorFlow**, що дозволяє користувачам легко створювати потужні моделі глибокого навчання.

TensorFlow, створений компанією Google, діє як бекенд при створенні моделей машинного навчання. Він працює, створюючи статичні графи потоку даних, які вказують, як дані рухаються через глибоке навчання. Граф містить вузли та ребра, де вузли представляють математичні операції. Він передає ці дані за допомогою багатовимірних масивів, відомих як тензори.

Keras, який пізніше був інтегрований з **TensorFlow**, може бути розглянутий як фронтенд для проектування моделей глибокого навчання. Він був реалізований з урахуванням зручності користувача, дозволяючи їм зосередитися на проектуванні своїх моделей нейронних мереж, не вдаючись у складні деталі бекенду. Це схоже на об'єктно-орієнтоване програмування, оскільки воно реплікує стиль створення об'єктів. Користувачі можуть вільно додавати різні типи шарів, активаційні функції та інше. Вони навіть можуть використовувати готові нейронні мережі для легкого навчання та тестування.

PyTorch - це інший фреймворк для машинного навчання, створений компанією Meta, яка раніше була відома як Facebook. Подібно до **Keras/TensorFlow**, він дозволяє користувачам створювати моделі машинного навчання. Цей фреймворк добре підходить для обробки природної мови (**Natural Language Processing - NLP**) та задач зору, але його можна налаштувати для більшості застосувань. Те, що робить **PyTorch** унікальним - це його динамічний обчислювальний граф. У ньому є модуль під назвою Autograd, який дозволяє виконувати автоматичну диференціацію

динамічно, на відміну від **TensorFlow**, де це є статичним. Крім того, **PyTorch** більш відповідає мові програмування Python, що робить його легшим для розуміння та використання корисних можливостей Python, таких як паралельне програмування.

SciPy - це бібліотека, призначена для наукових обчисленнь. Вона містить багато вбудованих функцій та методів для лінійної алгебри, оптимізації та інтеграції, які часто використовуються в машинному навчанні. Ця бібліотека корисна при розрахунках певних статистичних показників та трансформацій під час побудови моделі машинного навчання.

scikit-learn - це бібліотека машинного навчання, яка є розширенням **SciPy** і побудована з використанням **NumPy** та **Matplotlib**. Вона містить багато вбудованих моделей машинного навчання, таких як випадкові ліси (**Random Forest**), кластеризацію методом k-середніх(**k-means clustering**) та метод опорних векторів(**Support Vector Machine - SVM**).

2.7 РЕГРЕСІЙНІ МОДЕЛІ

Регресійні моделі - це методи моделювання, які використовуються для визначення відносин між незалежними та залежними змінними.

Зазвичай результатом роботи регресійної моделі є неперервне значення, також відоме як кількісна змінна. Типові приклади цього - передбачення ціни на будинок на основі його характеристик або передбачення продажів певного товару в новому магазині на основі інформації про попередні продажі.

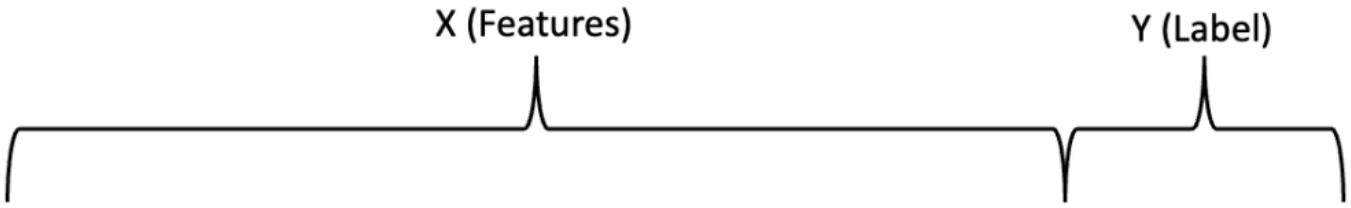
Перед створенням регресійної моделі необхідно зрозуміти дані та їх структуру. Більшість регресійних моделей використовують **SL**.

Для регресійних моделей навчальні дані зазвичай складаються з набору ознак (**Features**) і значення залежної змінної, відомої як мітка (**Label**). Зазвичай ознаки позначаються як X, а мітки як Y. (див. рисунок 2.14)

Зазвичай тренувальні дані розбиваються на дві підгрупи: навчальний(**training**) і тестовий(**testing**) набори. Навчальний набір зазвичай складається з 70-80% від загальної кількості даних, а тестовий набір містить решту. Це дозволяє моделі вчитися на навчальному наборі та перевіряти свої результати на тестовому наборі для оцінки її точності і якості. З отриманих результатів ми можемо зробити висновок про те, як наша модель працює на наборі даних.

Для того, щоб лінійна регресійна модель ефективно працювала, наші дані повинні мати лінійну структуру. Модель використовує цю формулу для навчання та вивчення даних:

$$y_i = \beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n \quad (2.1)$$



Number of Bedrooms	Year Built	Size (Sq Ft.)	Has Garage	Price
5	2019	14,560	Y	$305,000
4	2017	12,487	Y	$275,600
2	2010	9,822	N	$175,000
3	2015	10,110	Y	$235,000

Рисунок 2.14 - Приклад навчальних даних для регресійної моделі

У цьому рівнянні y_i представляє вихід моделі, або те, що зазвичай називають прогнозом(prediction). Прогноз обчислюється шляхом сумування зсуву β_0 та суми добутків $\sum_{i=1}^{n} \beta_i x_i$. При роботі з даними зазвичай використовується їх матричне представлення, що робить їх зрозумілими та легкими у використанні з мовою програмування Python:

$$y = X\beta \quad (2.2)$$

$$y = \begin{bmatrix} y_1 \\ y_2 \\ \cdot \\ \cdot \\ \cdot \\ y_n \end{bmatrix} \quad (2.3)$$

$$X = \begin{bmatrix} x_{11} & \cdots & x_{1c} \\ \vdots & \ddots & \vdots \\ x_{r1} & \cdots & x_{rc} \end{bmatrix} \quad (2.4)$$

$$\beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ . \\ . \\ . \\ \beta_n \end{bmatrix} \quad (2.5)$$

Кількість ознак визначає характер задачі, яка вирішується. Якщо в даних є лише одна ознака (див. рисунок 2.15), то це є проста лінійна регресійна модель.

Вона може вирішувати прості завдання, але для більш складних даних може бути важко встановити взаємозв'язки. Тому можна створити модель багатофакторної лінійної регресії, додавши більше ознак x_i. Це дозволить моделі бути більш надійною та виявляти глибші залежності у даних.

Після тренування моделі потрібно оцінити її та з'ясувати, як вона справляється з тестовими даними.

У випадку лінійної регресії є дві загальні метрики, які використовуються для оцінки моделі - це квадратний корінь з середньоквадратичної помилки (**Root Mean Square Error - RMSE**) та $\mathbf{R^2}$.

RMSE представляє собою стандартне відхилення залишкових помилок у прогнозах(**Residual**). Залишок вказує на відстань між фактичними точками даних та лінією регресії. Чим більше середнє відхилення всіх точок від лінії, тим вища помилка. Це свідчить про слабку модель, оскільки вона не може знайти кореляцію між точками даних (див. рисунок 2.16).

Ця метрика може бути розрахована за допомогою наступної формули, де y_i - це фактичне значення, $\widehat{y_i}$ - - це прогнозоване значення, а n - кількість точок даних:

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N}(y_i - \widehat{y_i})^2}{N}} \quad (2.6)$$

$\mathbf{R^2}$, також відомий як коефіцієнт детермінації, вимірює частку варіації у залежних змінних (Y), яку можна пояснити незалежними змінними (X).

По суті, він показує, наскільки добре дані підходять під модель.

На відміну від **RMSE**, який може бути довільним числом, $\mathbf{R^2}$ виражається числом в діапазоні від 0 до 1, що можна легше зрозуміти. Чим $\mathbf{R^2}$ ближче до 1, тим краще кореляція даних.

Хоча це корисний показник, значення близькі до 1 відсоток не завжди є показником сильної моделі. Якість значення $\mathbf{R^2}$ залежить від конкретного застосування та розуміння користувачем даних.

$\mathbf{R^2}$ можна розрахувати за допомогою цієї формули, де де y_i - це фактичне значення, $\widehat{y_i}$ - - це прогнозоване значення, $\bar{y}$ – середнє значення, а n - кількість точок даних:

$$R^2 = \sqrt{\frac{\sum_{i=1}^{n}(\hat{y}_i - \bar{y})^2}{(\sum_{i=1}^{n}(y_i - \bar{y})^2)}} \quad (2.7)$$

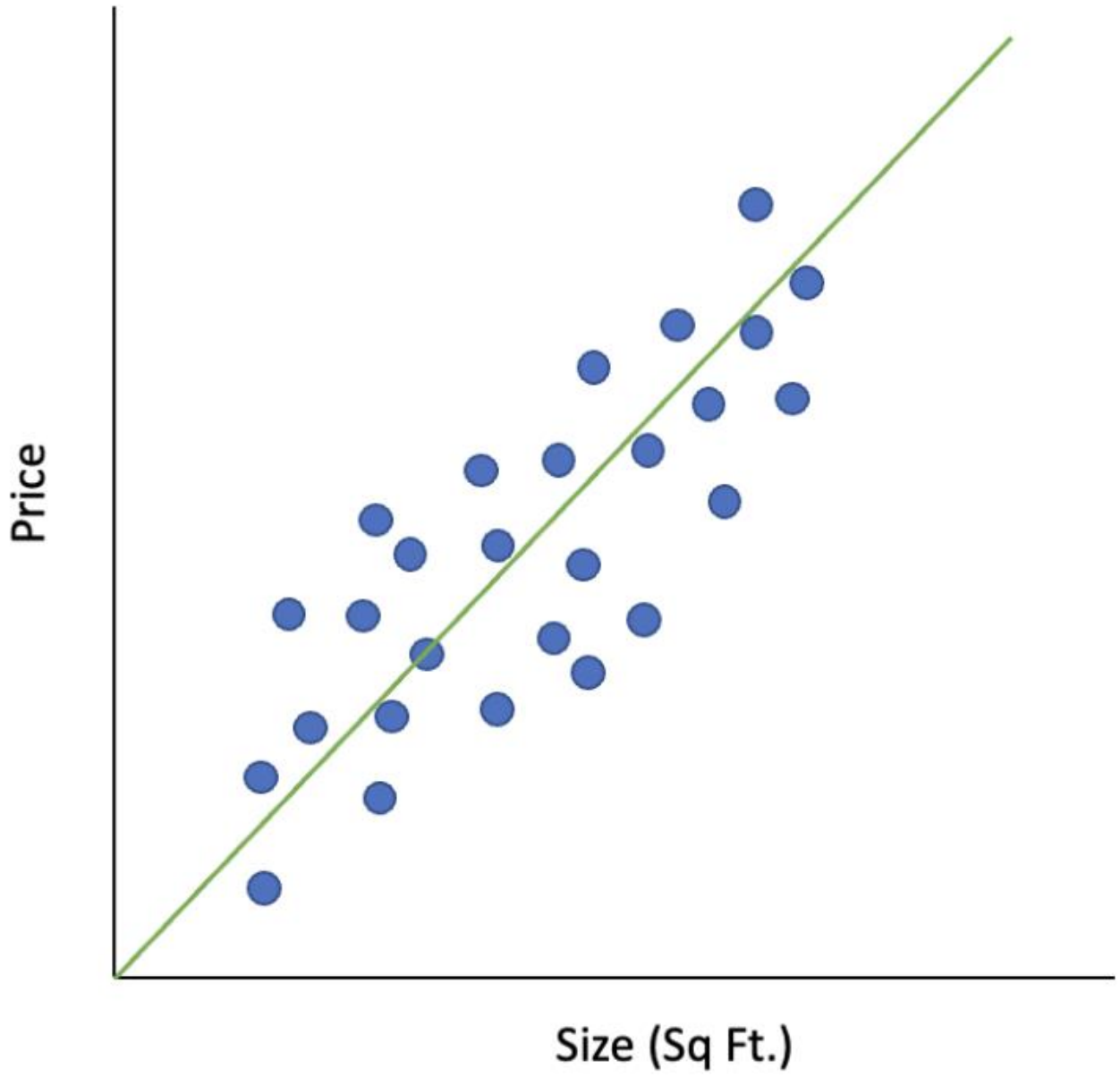


Рисунок 2.15 - Приклад простої лінійної регресії

Багато інших метрик можуть оцінювати ефективність регресійної моделі, але цих двох достатньо для отримання уявлення про те, як вона працює. Під час створення та оцінювання моделі важливо візуалізувати дані та модель, оскільки це може виявити ключові моменти. Графіки можуть допомогти визначити, чи модель перенавчена (**overfitting**) або недонавчена (**underfitting**).

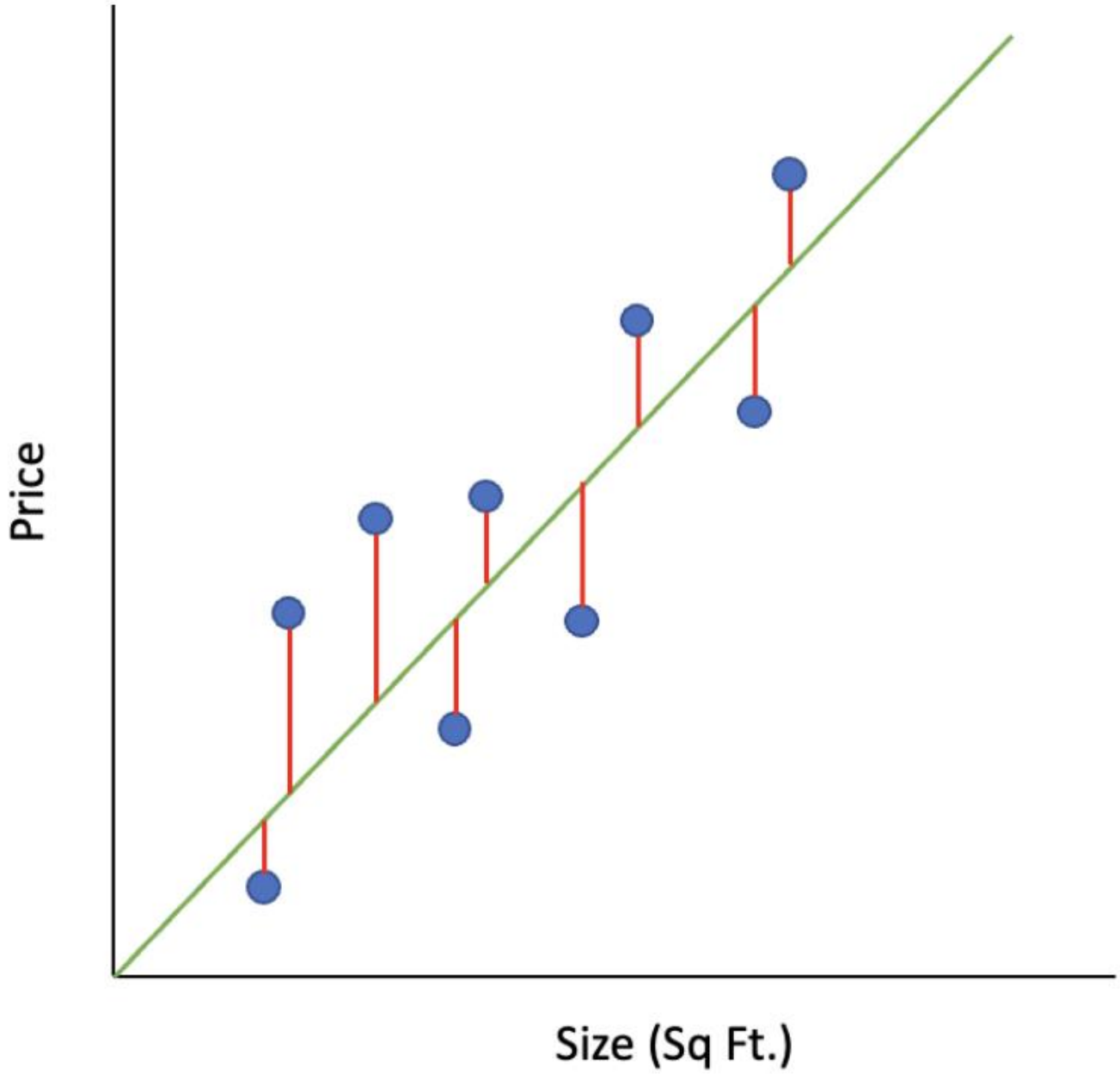


Рисунок 2.16 - Розрахунок залишків лінійної регресійної моделі

Перенавчання відбувається, коли ваша модель занадто підходить для тренувальних даних. **RMSE** буде дуже низьким, і точність тренування майже 100%. Хоча це може здатися привабливим, це свідчить про погану модель. Це може бути спричинено однією з двох речей: недостатньо даних або занадто багато параметрів. У результаті, коли ви тестуєте модель на нових даних, які вона раніше не бачила, вона буде дуже погано працювати через нездатність узагальнювати дані (див. рисунок 2.17).

Для вирішення проблеми перенавчання можна збільшити обсяг тренувальних даних або скоротити складність моделі.

Також корисно випадковим чином перемішати дані перед розбиттям їх на тренувальний та тестовий набори.

Ще однією важливою технікою є регуляризація. Існує багато різних методів регуляризації (L1 або L2 регуляризація) в залежності від моделі, проте всі вони працюють на схожих принципах, додаючи зміщення або шум до моделі для уникнення перенавчання. У рівнянні регресії (1.1), можна додати ще один член ϵ , щоб показати застосування регуляризації до моделі:

$$y_i = \beta_0 + \beta_1 x_1 + \ldots + \beta_n x_n + \epsilon \quad (2.8)$$

З іншого боку, недонавчання виникає (див. рисунок 2.18), коли модель не може знайти значущу кореляцію в межах даних. Це не так часто трапляється, , оскільки в більшості даних легко виявити патерни. Якщо це трапляється, можливо, дані мають занадто багато шуму і є серйозно некорельованими, або модель занадто проста і не має достатньо параметрів, або ж модель не є ефективною для конкретного застосування.

Отже, мета полягає в знаходженні найкращої моделі (див. рисунок 2.19), між перенавчанням та недонавчанням. Це вимагає часу та експериментів для знаходження моделі, яка відповідає потребам, але використання ключових показників та метрик, обговорених в цьому підрозділі, може допомогти рухатися в правильному напрямку.

Збір та обробка даних для моделей машинного навчання є важливими кроками, які можуть впливати на точність та ефективність моделей. Ось деякі ключові моменти, які слід враховувати під час виконання цих кроків:

- **Нормалізація даних** допомагає запобігти тому, щоб ознаки з дуже високими або низькими значеннями домінували над моделлю. Це можна зробити, масштабувавши всі ознаки до загального діапазону, наприклад, від 0 до 1.
- **Очищення даних** допомагає усунути будь-які помилки або невідповідності в даних, які можуть негативно вплинути на модель. Це може включати видалення відсутніх значень, відхилень та дублікатів записів.
- **Розуміння даних** допомагає ідентифікувати важливі ознаки, відхилення та зв'язки між змінними. Це можна зробити за допомогою дослідницького аналізу даних (**Exploratory Data Analysis - EDA**), який включає аналіз та візуалізацію даних.

Зменшення розмірності даних може допомогти покращити продуктивність моделі, зменшивши кількість ознак без втрати надто багато інформації

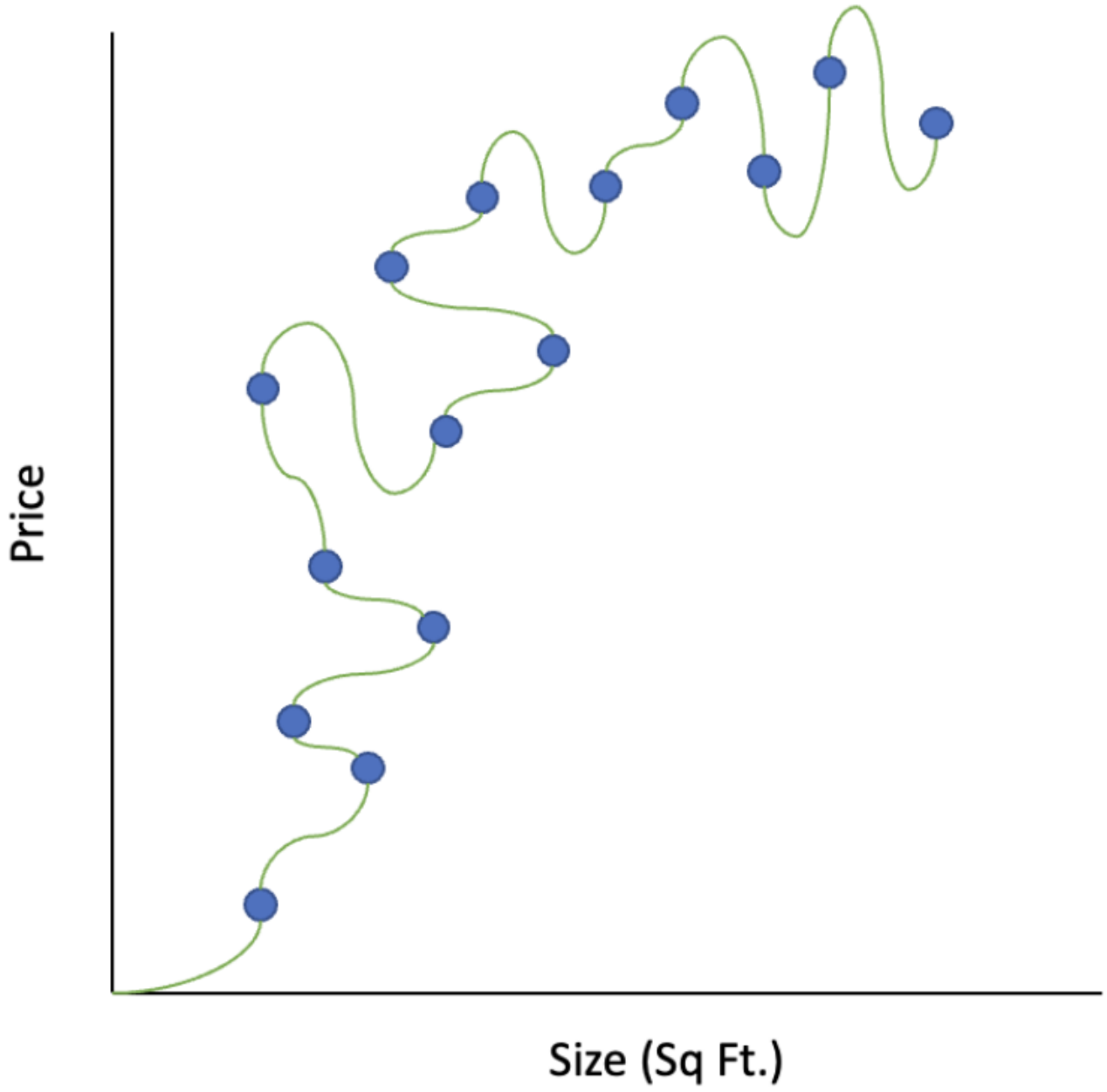


Рисунок 2.17 - Приклад перенавченої лінійної регресії

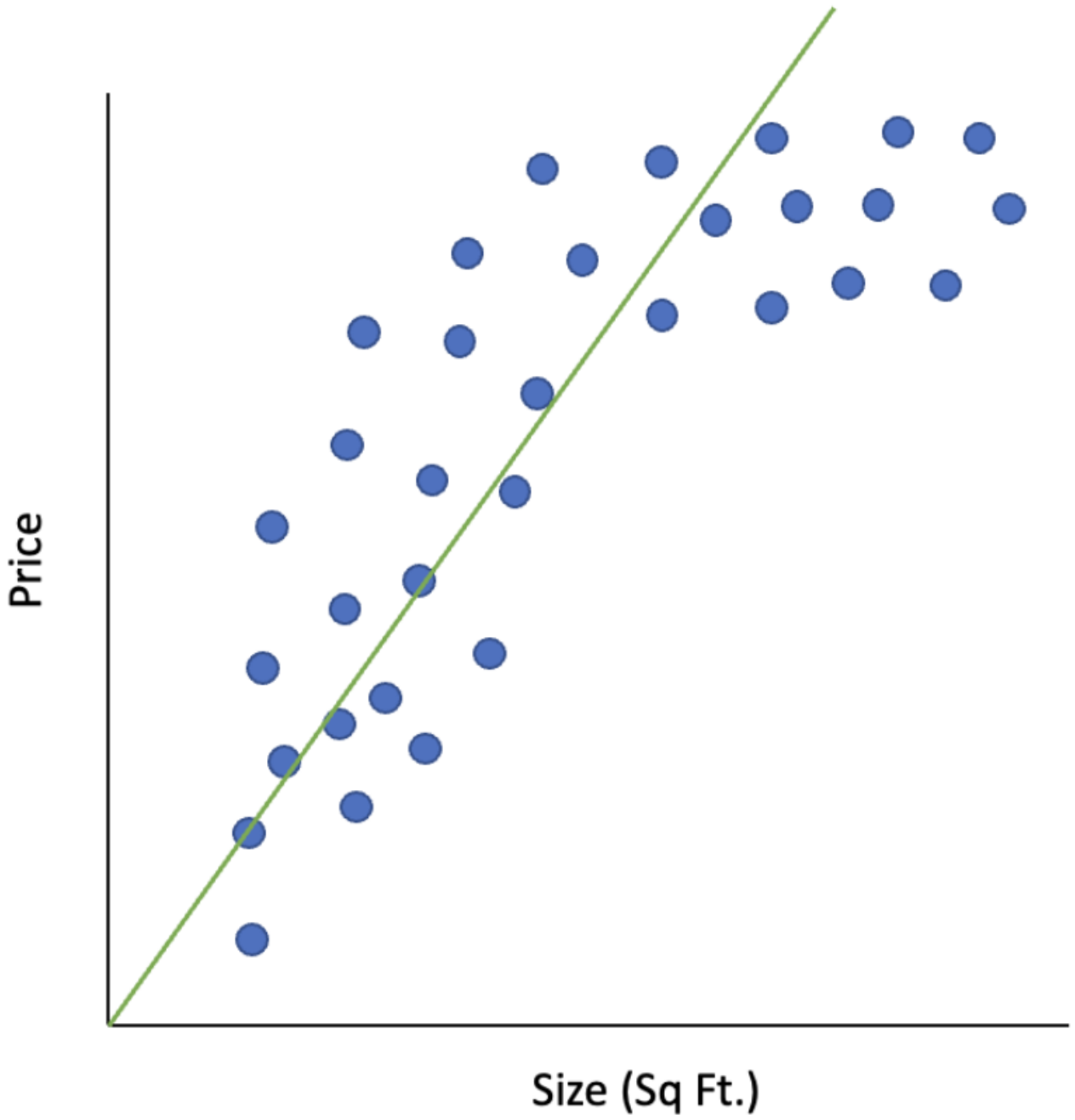


Рисунок 2.18 - Приклад недовченої лінійної регресії

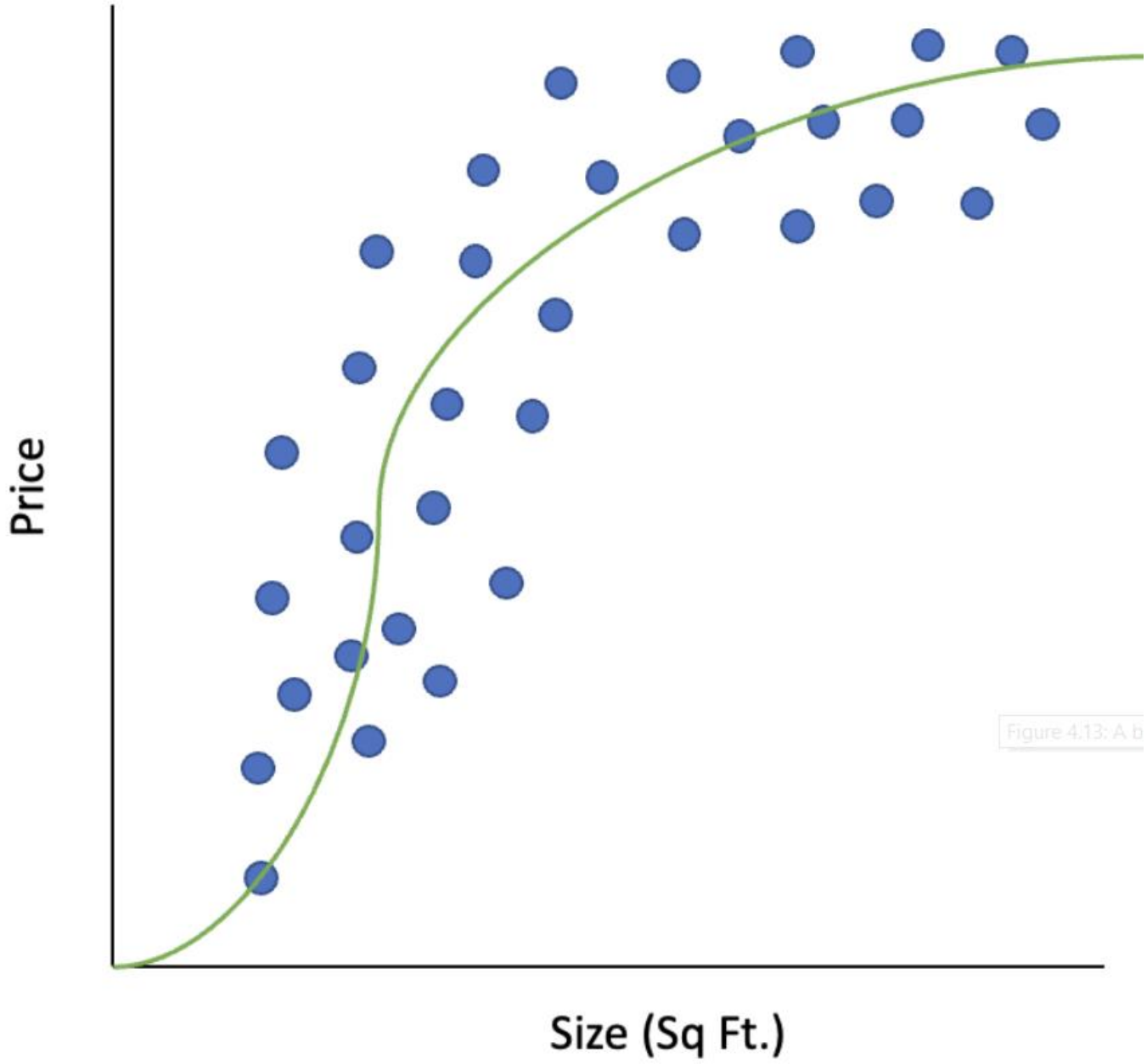


Рисунок 2.19 - Приклад лінійної регресії,
яка найкращим чином описує дані

- .

2.8 БАГАТОКЛАСОВІ МОДЕЛІ КЛАСИФІКАЦІЇ

Багатокласова класифікація - це задача **ML**, яка полягає в тому, щоб класифікувати об'єкти на кілька класів. На відміну від двокласової класифікації, де об'єкт може належати лише до одного з двох класу, у багатокласовій класифікації об'єкт може належати до декількох класів.

Багатокласові моделі класифікації вважаються універсальними, оскільки вони можуть застосовуватися як у **SL**, так і в **UL**, тоді як регресійні моделі в основному

використовуються **SL**. Існує декілько регресійних моделей (наприклад, логістична регресія та машини опорних векторів), які також вважаються моделями класифікації, оскільки вони використовують поріг для розділення виходу неперервних значень на різні категорії.

UL є поширеним застосуванням, яке широко використовується. Хоча **SL** зазвичай працює краще і надає значущі результати, оскільки ми знаємо очікуваний вихід, більшість даних, які ми збираємо, не має розмітки. Потрібно багато часу та коштів, щоб експерти переглянули дані та розмітили їх. **UL** допомагає зменшити витрати та час, дозволяючи моделі спробувати визначити мітки для даних та вилучити з них значущу інформацію. Іноді вони навіть можуть працювати краще, ніж люди.

Кількість категорій на виході класифікаційної моделі визначає її тип. (див. рисунок 2.20). Якщо модель має лише два виходи (наприклад, розділення листів електронної пошти на: "спам" та "не спам"), то це є бінарним класифікатором. У випадку, коли в моделі більше ніж два виходи, вона вважається багатокласовим класифікатором.

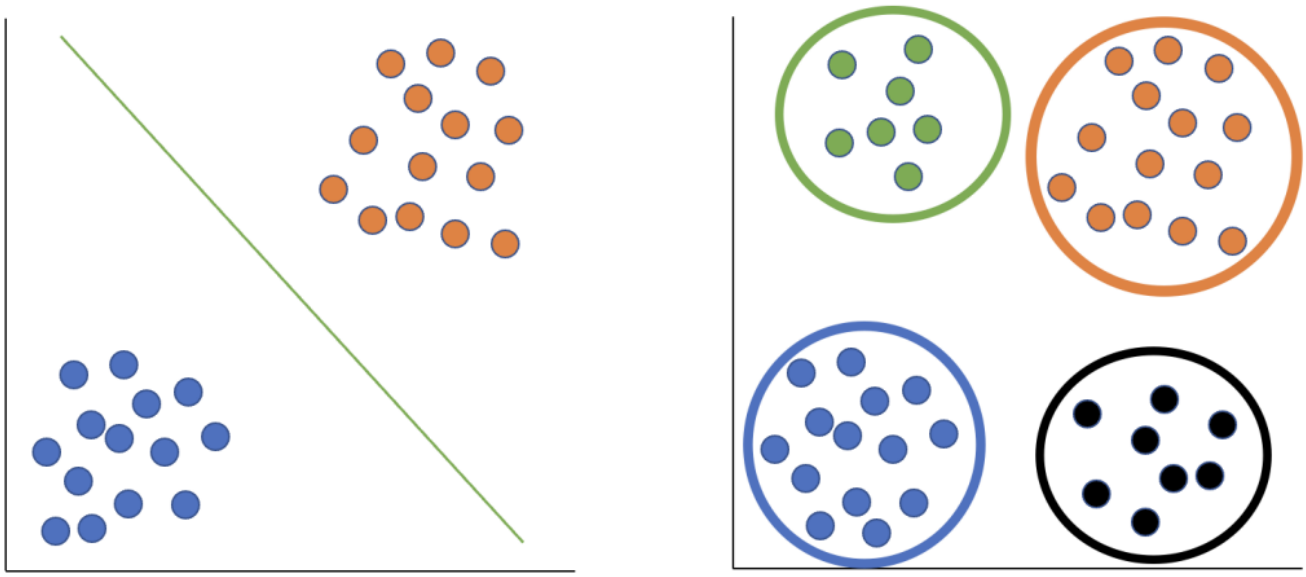


Рисунок 2.20 - Бінарні та багатокласові класифікатори

Класифікатори можна класифікувати за типом навчального алгоритму, який вони використовують. Існують два типи навчальних алгоритмів: ледачі(**lazy learning**) та старанні (**eager learning**).

Ледачі навчальні алгоритми фактично зберігають навчальні дані та чекають, доки вони отримають нові тестові дані. Як тільки вони отримають тестові дані, модель класифікує нові дані на основі вже існуючих даних. Ці тип алгоритмів

потребує менше часу під час навчання, оскільки є можливість постійно додавати нові дані, не перенавчаючи всю модель, але витрачають більше часу під час класифікації, оскільки їм потрібно пройти всі точки даних.

Основні алгоритми цього типу:

1) Алгоритм K-найближчих сусідів (**K-Nearest Neighbor - KNN**).
2) Алгоритм опорних векторів (**Support Vector Machines - SVM**).
3) Алгоритм машинного навчання на основі наївного Байеса (**Naive Bayes Classifier**).

Старанні навчальні алгоритми працюють навпаки. Щоразу, коли до моделі додаються нові дані, вона повинна повторно навчатися. Хоча це займає більше часу порівняно з ледачими навчальними алгоритмами, запит до моделі набагато швидший, оскільки їм не потрібно проходити через всі точки даних.

Основні алгоритми цього типу:

1) Дерево ухвалення рішень (**Decision tree**).
2) Штучні нейронні мережі (**ANN**).

2.9 МОДЕЛІ ДЛЯ АНАЛІЗУ ЧАСОВИХ РЯДІВ

У зв'язку з невизначеністю майбутнього, можливість прогнозування конкретних тенденцій і закономірностей стала актуальною темою в сучасній житі, тому деякі вчені та дослідники розробили моделі, які враховують час при здійсненні прогнозів, таких як передбачення цін на бензин, фондового ринку та обсягів продажів.

Перед тим, як переходити до моделей, необхідно спочатку осмислити різні концепції в аналізі часових рядів.

Перший етап у вирішенні завдань, пов'язаних з часовими рядами, полягає в обробці даних.

Зазвичай часові ряди складаються з чотирьох компонентів: (див. рисунок 2.22)

1) Тренд(**Trend**) - дані слідують за зростаючою або спадаючою безперервною шкалою часу і не виявляють періодичних змін.
2) Сезонність(**Seasonality**) - дані змінюються в зазначені періоди часу.
3) Циклічність(**Cyclical**) - дані змінюються, але без фіксованої періодичності.
4) Нерегулярність(**Irregular**) - дані змінюються випадково без чітких закономірностей.

Часові ряди в аналізі часових рядів класифікуються за стаціонарністю на два класа:

1) Стаціонарні – коли певні атрибути даних, такі як середнє значення, дисперсія та коваріація, не змінюються з часом.
2) Нестаціонарні – навпаки атрибути даних змінюються з часом.

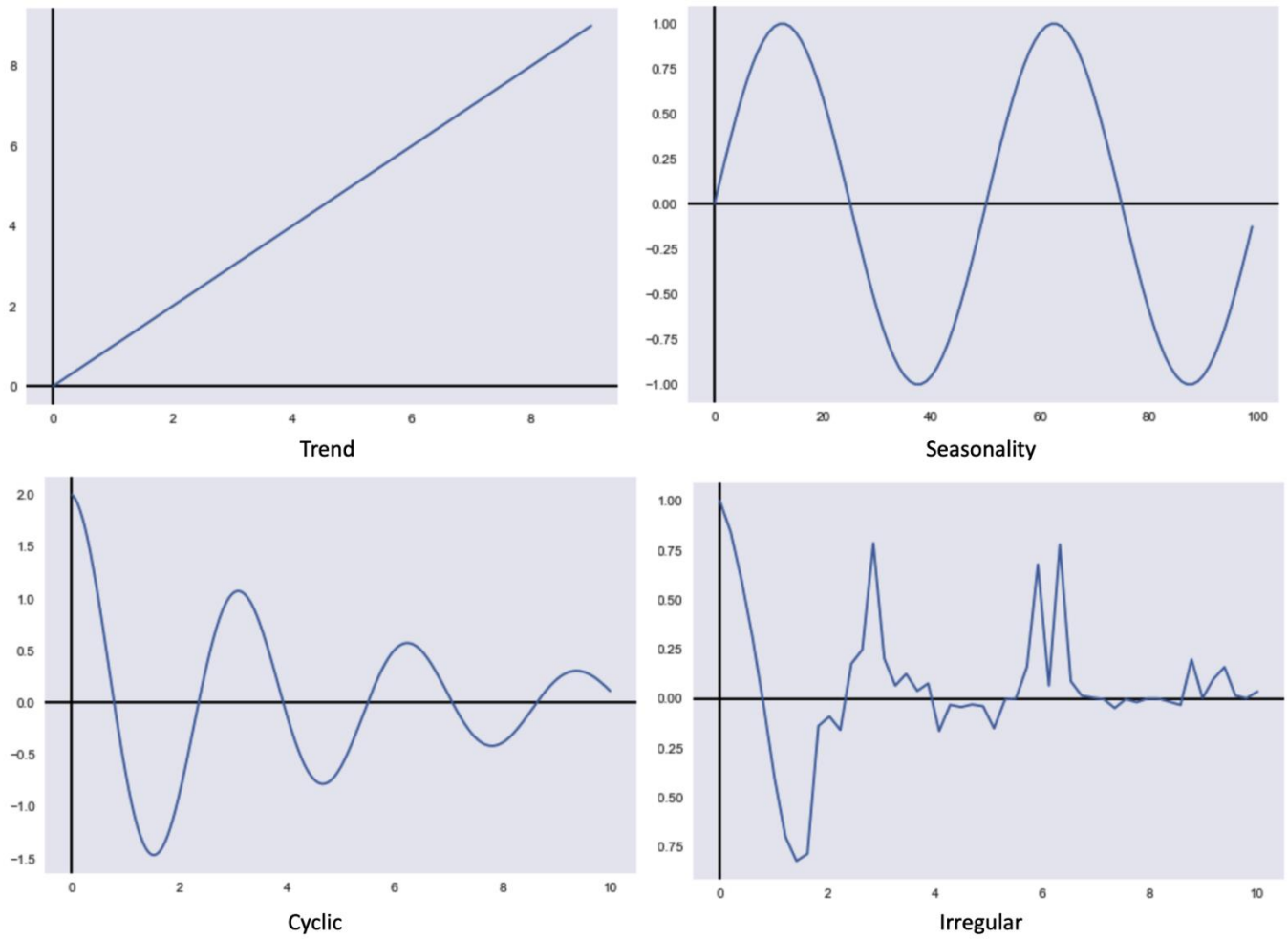


Рисунок 2.21 - Компоненти часових рядів

Якщо мати справу з нестаціонарними рядами при створення моделей **ML** це може призводити до ненадійних результатів, тому потрібно використовувати певні техніки для перетворення ряду до стаціонарного виду: диференціювання та трансформація.

Диференціювання - це математичний метод, який використовується для нормалізації середнього значення та видалення дисперсії (див. рисунок 1.30) і розраховується за допомогою такої формули:

$$\hat{y}_t = y_t - y_{t-1} \quad (2.9)$$

Трансформація - це використання математичних методів для усунення зміни дисперсії, найбільш часто використовуються: логарифмічне перетворення, взяття квадратного кореня, піднесення до степеню.

Після приведення ряду до стаціонарного виду можливо застосування моделей для аналізу і передбачення часових рядів.

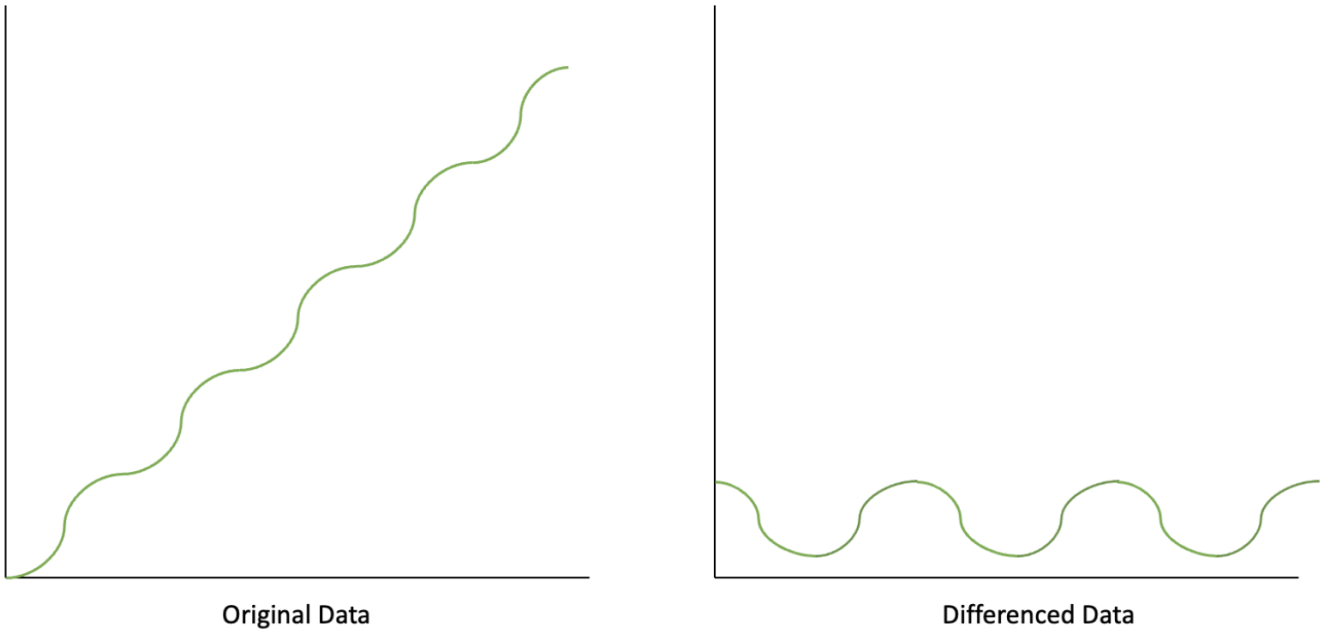


Рисунок 2.22. - Диференціювання компонентів часового ряду

Найпопулярнішою моделлю для аналізу часових рядів є модель авторегресії з інтегрованим ковзним середнім **ARIMA** (**Auto-Regressive Integrated Moving Average**).

Ця модель складається з трьох підкомпонентів:

- Авторегресія (**AR**) - регресійна модель, що використовує залежності між поточним та минулим часом для прогнозування;
- Інтегрованість (**I**) - процес диференціювання для забезпечення стаціонарності даних;
- Ковзне середнє (**MA**) - моделює взаємозв'язок між очікуваними даними та залишковою помилкою, розраховуючи рухоме середнє для запізнених спостережень.

Нейромережі також можуть бути використані для моделювання та прогнозування часових рядів. Основна перевага нейромереж у порівнянні з традиційними методами полягає у їх здатності автоматично виявляти складні залежності в даних та адаптуватись до змін у часовому ряді.

Нейромережі типу **TLRN** (**Time-Lagged Recurrent Network**) та модель **NNARX (Nonlinear Nonparametric AutoRegressive model with eXogenous inputs)** (див. рисунок 2.23) є підтипом рекурентних нейронних мереж (**RNN**), які зазвичай використовуються для моделювання часових рядів, зокрема використовуються для прогнозування наступного елемента часового ряду на основі попередніх елементів. **TLRN** дозволяє враховувати затримки в часі у вхідних даних та зберігати попередні стани мережі для кращого прогнозування майбутніх значень.

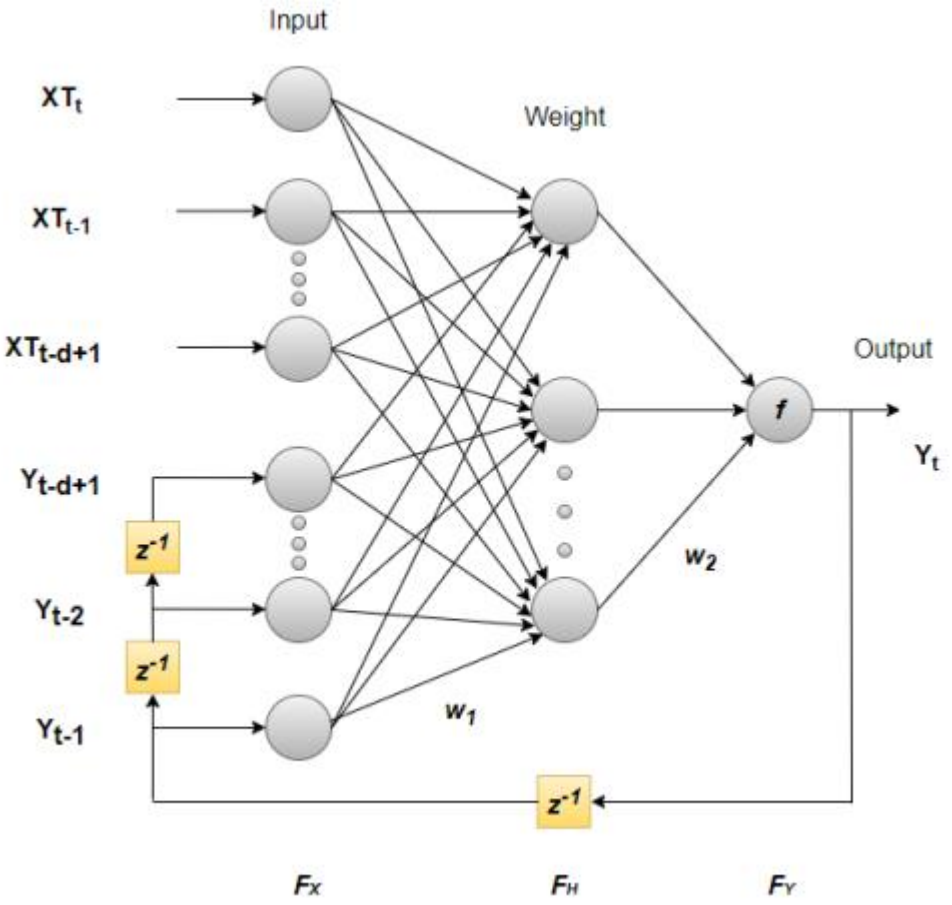


Рисунок 2.23 - Архітектура моделі мережі NARX.

На відміну від штучної нейронної мережі з прямим поширенням, де дані обробляються в одному напрямку від вхідного до вихідного шару без зворотного зв'язку, **RNN** зберігає інформацію про попередні стани даних за допомогою зворотних зв'язків, що дозволяє моделі зберігати попередні стани даних та використовувати їх для подальшої обробки даних.

Пізніше були запропоновані рекурентні нейромережі типу **LSTM** (**Long Short-Term Memory**) та **GRU** (**Gated Recurrent Unit**).

Архітектура **LSTM** була спеціально розроблена для вирішення проблеми зникнення градієнту (**vanishing gradient problem**), яка може виникати під час алгоритму зворотного поширення помилок в традиційних рекурентних нейронних мережах.

Проблема зникнення градієнту виникає, коли сигнал градієнту, який використовується для оновлення вагів під час зворотного поширення помилок (**backpropagation**), стає дуже малим, що призводить до повільного навчання нейромережі і навіть до відсутності навчання.

Ця проблемма вирішується шляхом введення механізму гейтів, що дозволяє мережі вибірково зберігати або забувати інформацію протягом часу.

Архітектура **LTSM Cell** включає три гейти: вхідний, вихідний та забування (див. рисунок 2.24).

Гейтингові блоки можуть вчитися відкриватися або закриватися на основі вхідних даних та попереднього стану пам'яті, дозволяючи мережі вибірково зберігати або відкидати інформацію протягом часу.

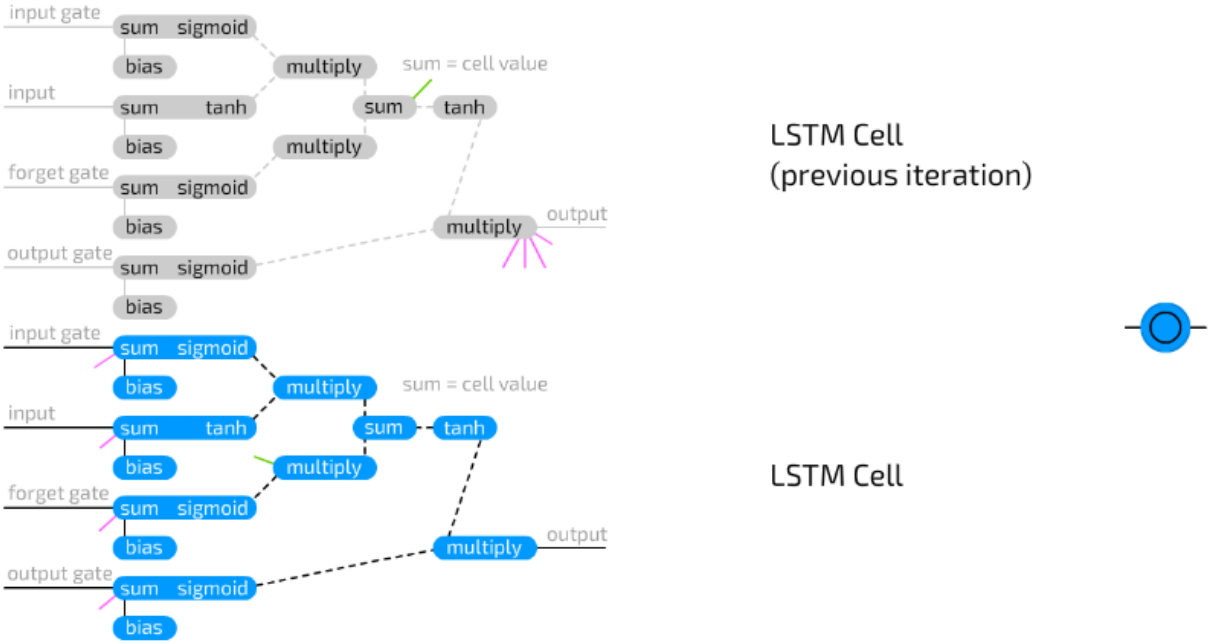


Рисунок 2.24 - Архітектура нейрона - **LTSM Cell**

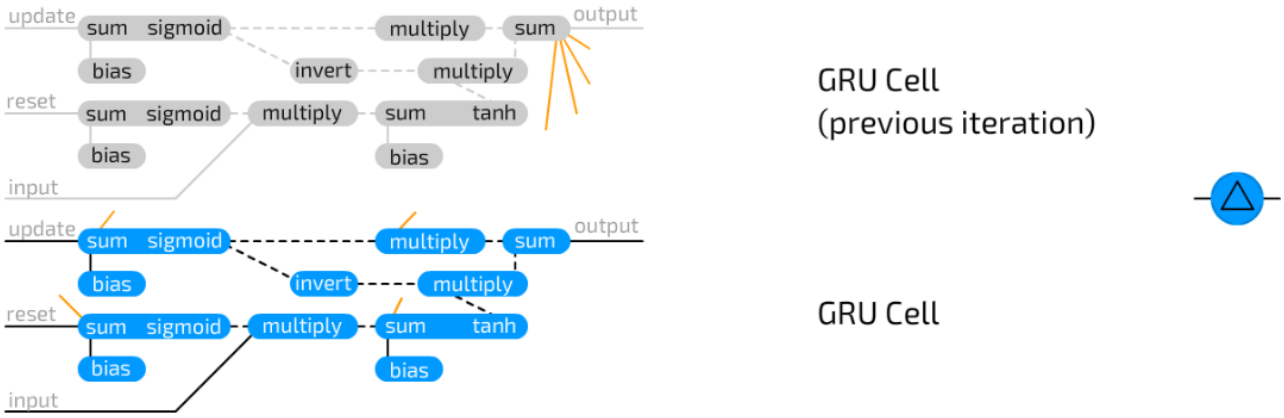


Рисунок 2.25 - Архітектура **GRU Cell**

GRU- є спрощеною версією **LSTM**, з меншою кількістю гейтів: оновлення та скидання, що зазвичай забезпечує їхню більш швидку роботу (див. рисунок 2.25).

2.10 ІНТЕГРАЦІЯ ML В MSA

У попередніх підрозділах 1.1 - 2.9 проаналізовано основні концепції та підходи, які застосовуються в **MSA** та **ML**.

Настав час для синтезу та інтеграції **ML** в систему **MSA**.

Простір для інтеграції машинного навчання в системи MSA є обширним і може бути використаним для багатьох різних сценаріїв, але можна виділити чотири основних сфери інтеграції

Перша сфера - прогнозування навантаження на систему, яке дозволяє визначати, коли мікросервіси зазнають навантаження вище за звичайне, і вжити заходів, для запобігання відказам системи (див. малюнок 2.26).

Прогнозування навантаження на систему є поширеною проблемою при роботі з веб сервісами. **MSA** має перевагу перед монолітними системами, тому що ресурси виділяються окремо для кожного мікросервісу, що спрощує обслуговування та масштабованість.

Проте, як вже зазначалося у підрозділі 2.5, існують ситуації в **MSA**, коли мікросервіс зазнає великого навантаження і, відповідно, це призводить до каскадного ефекту, коли відкази розповсюджуються на інші мікросервіси.

За допомогою **ML** можна навчити модель, використовуючи різноманітні характеристики, такі як час відгуку(**response time**) критичних мікросервісів, та використовувати її для виявлення закономірностей роботи системи **MSA**.

Аналогічно з **Circuit Breaker**, ця модель може оперативно визначати, чи має мікросервіс велике навантаження і вирішувати цю проблему до того, як вона стане критичною і почне негативно впливати на інші мікросервіси.

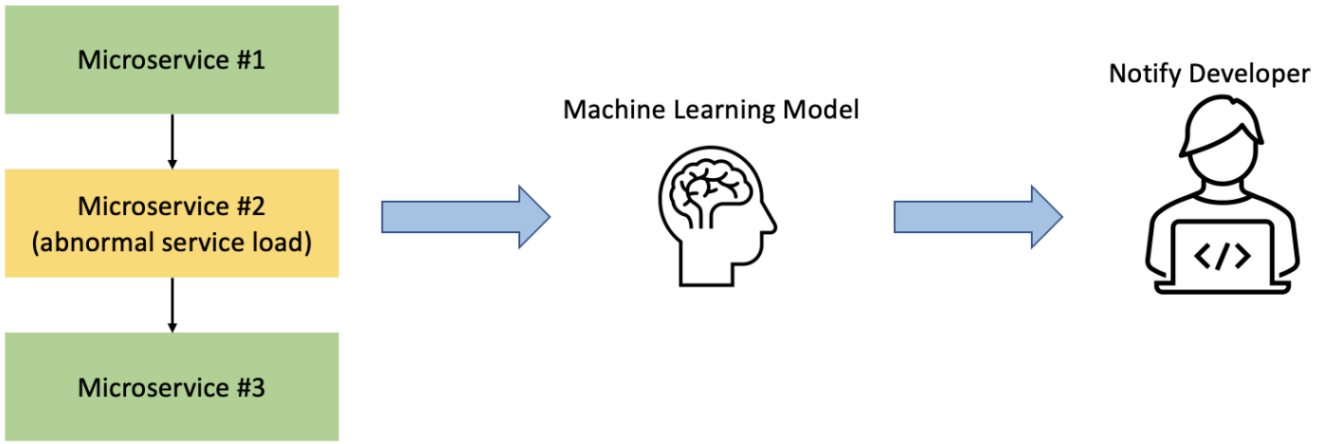


Рисунок 2.26 - Модель прогнозування навантаження на систему

Друга сфера - прогнозування деградації продуктивності системи. Це схоже на прогнозування навантаження на систему, але воно має більш конкретну мету - виявляти проблеми, які можуть призвести до зниження продуктивності або надійності системи. (див. малюнок 2.27).

Так само, як і модель прогнозування навантаження на систему, ми можемо побудувати модель для виявлення аномалій в **MSA**, які можуть призвести до деградації продуктивності системи. Замість того, щоб фокусуватися лише на

навантаженні для конкретного мікросервісу, можливо вивчити всю систему **MSA** і виявити різні закономірності того, як вона працює взагалі.

Системи **MSA** можуть відчувати різні навантаження і помилки протягом певних часів і періодів. Наприклад, система **MSA** стикатися зі сплесками запитів у певні періоди, такі як свята та сезонні події, коли кількість користувачів може різко зрости. Дозволяючи моделі вивчати та розуміти систему MSA і те, як вона працює з часом, можна підготувати модель до кращого виявлення аномалій та запобігання хибних спрацювань.

Крім того, замість того, щоб відстежувати окремі мікросервіси, потрібно оцінювати кластери мікросервісів та те, як вони взаємодіють з усією системою **MSA**. Таким чином, є можливість виявляти певні вузькі місця та помилки, які можуть виникати в масштабі цілої системи.

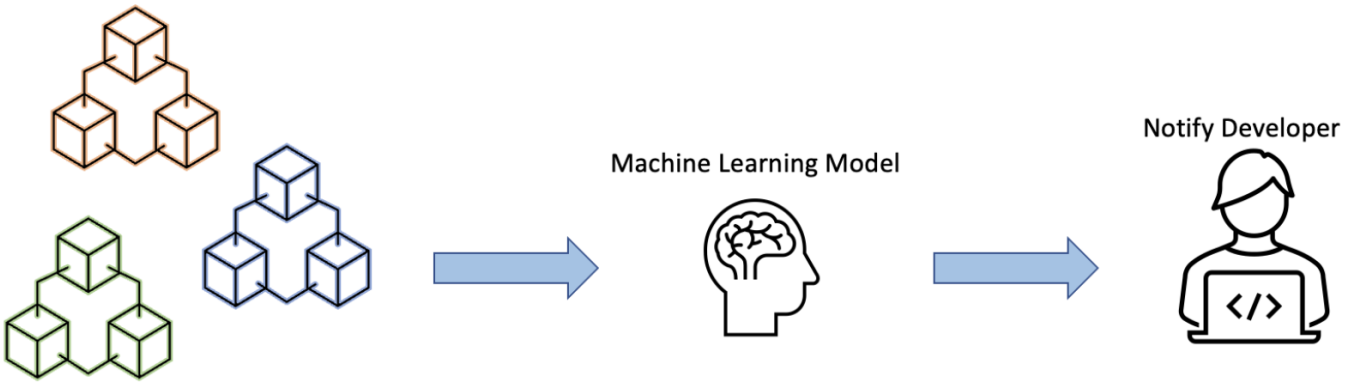


Рисунок 2.27 - Модель прогнозування деградації продуктивності системи

Третя сфера застосування **ML -** безпека системи: В епоху кібербезпеки важливо мати можливість захистити свою систему **MSA** від цілеспрямованих атак. Вивчаючи поведінку вашої системи MSA, модель може прогнозувати та виявляти атаки, які можуть загрожувати безпеці системи (див. малюнок 2.28).

Машинне навчання успішно використовується в галузі кібербезпеки. З розвитком більш вдосконалених хакерських атак стає актульною проблема захисту системи **MSA**.

Машинне навчання спрощує створення надійних моделей, які можуть аналізувати та прогнозувати атаки, ще до того, як вони зможуть вплинути на роботу системи.

Наприкладі, атаки на відмову в обслуговуванні (**Denial of Service - DoS**), яка спрямована на перешкоджання користувачам у доступі до певних послуг. Ці атаки стають все більш вдосконаленими з розвитком технологій. За допомогою машинного навчання можливо модель розпізнавати атаки типу **DoS** в контексті системи **MSA**. Таким чином, вона може визначати, чи піддається наша система таким MSA атакам

та повідомляти команду забезпечення безпеки або впроваджувати контрзаходи для боротьби із певними атаками та збереження цілісності системи.

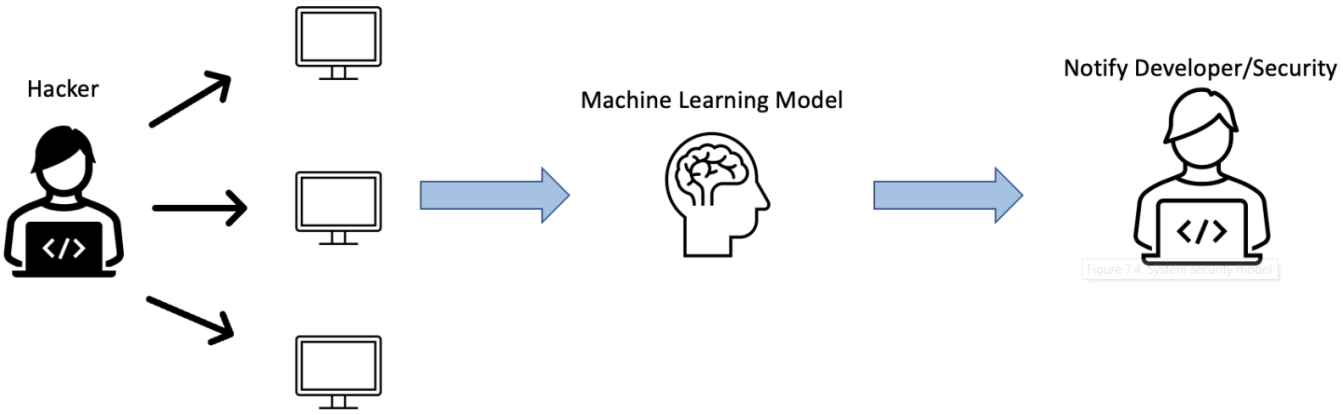


Рисунок 2.28 - Модель захисту системи

Четверта сфера застосування **ML** - планування ресурсів системи.

При зростанні та розвитку системи важливо належним чином розподіляти ресурси та адаптуватися до потреб системи.

За допомогою машинного навчання можливо вивчити, які сервіси вимагають більше ресурсів і скільки потрібно ресурсів для ефективного масштабуваня системи (див. малюнок 2.29)**.**

Частиною процесу самовідновлення системи (**self-healing**) є виділення ресурсів для певних мікросервісів у випадку зростання та розширення системи **MSA**.

З часом при збільшенні кількості користувачів, і внаслідок цього збільшення кількості запитів та нагрузки на систему, можлива некоректна ідентифікація проблем та помилкове планування та виділення ресурсів, але завдяки застосуванню продвинутої моделі **ML**, яка може відстежувати поступовий ріст системи **MSA** та визначати, коли певним сервісам потрібні додаткові ресурси, дозволить уникнути таких помилок та значно покращити надійність системи, оскільки вона здатна коректно і ефективно виділяти ресурси.

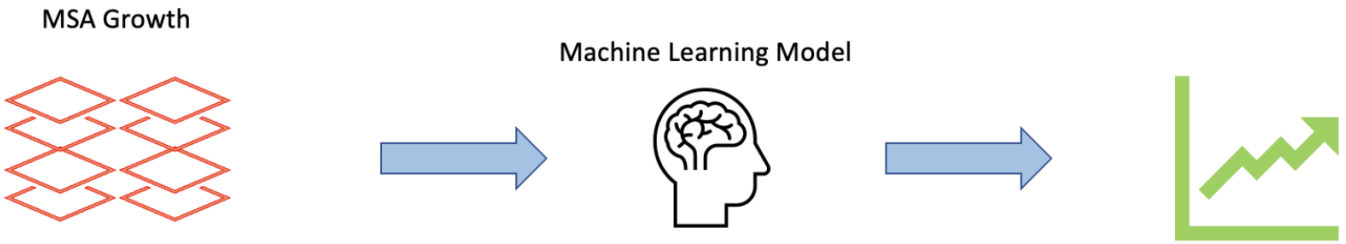


Рисунок 2.29 - . Модель планування ресурсів системи

2.11 ВИСНОВОК ДО РОЗДІЛУ II

У даному розділі детально розглянуто ключові аспекти машинного навчання та їхнє впровадження в мікросервісну архітектуру.

В підрозділах 2.1 – 2.5 детально розглянуті різноманітні патерни проектування в контексті мікросервісної архітектури, такі як патерни **Saga**, **CQRS**, **API Gateway** та **Circuit Breaker**, з описом їх переваг та недоліків.

В підрозділах 2.6 – 2.9 проведено аналіз різних моделей **ML**, від регресійних до багатокласових моделей класифікації та моделей для аналізу часових рядів, а також описали сучасні бібліотеки для створення моделей машинного навчання з використанням мови програмування Python.

Закінчивши аналіз елементів і підходів **MSA** і **ML**, в підрозділі 2.10 зроблений висновок, що впровадження **ML** в **MSA** є важливим етапом для підвищення ефективності та можливостей розвитку складних та великих систем. Такий підхід дозволяє автоматизувати процеси, покращувати якість прийняття рішень та реагувати на зміни в реальному часі.

В цілому, розділ II дає обґрунтоване розуміння взаємодії машинного навчання та мікросервісів, а також забезпечив практичні вказівки та приклади для успішної інтеграції цих технологій у сучасні системи.

Матеріал викладений в розділі II про взаємодію машинного навчання та мікросервісної архітектури, становить цінну основу для третього та четвертого практичного розділу в якому розробляється і досліджується інтелектуальна система масштабування вебсервисів на базі **MSA** з використанням **ML**.

3. РОЗДІЛ ІІІ. ЗАСТОСУВАННЯ ML В СИСТЕМІ MSA

3.1 АЛГОРИТМИ ІНТЕЛЕКТУАЛЬНОЇ СИСТЕМИ МАСШТАБУВАННЯ ВЕБСЕРВИСІВ НА БАЗІ MSA

Зазвичай для автомасштабування вебсервисів використовують стратегію порогових правил (**Threshold Rules Policy**), яка полягає в встановленні певних порогів або меж, які, якщо перевищуються або досягаються, спричиняють автоматичне масштабування ресурсів, щоб забезпечити оптимальну продуктивність та надійність системи.

Використання методів **ML** може значно покращити стратегії автомасштабування вебсервісів, особливо для великих та складних систем, оскільки такі системи часто мають велику кількість параметрів та навантаження, що змінюються в дуже динамічний спосіб.

В таких системах виникають закономірності, які є результатом повторюваності або схожості в даних, взаємодії між компонентами системи або способах, якими система обробляє дані.

Ці закономірності можуть бути виявлені за допомогою різних алгоритмів машинного навчання, що може суттєво покращити ефективність та масштабованість системи, а також забезпечити більш оптимальну роботу системи в цілому.

Як було зазначено у підрозділі 2.11, існує багато областей у системі **MSA**, де можливо використання штучного інтелекту.

Увага буде зосереджена на двох основних потенційних напрямках покращення (див. рисунок 3.1), що реалізується окремими додатковими сервісами штучного інтелекту. Перший полягає в підвищенні швидкості реакції системи у разі відмови мікросервісу або зниження продуктивності. Друга область покращення - це введення проактивної ролі вимикача ланцюга (**Circuit Breaker**).

Перший мікросервіс штучного інтелекту називається "Служба контролю рівня продуктивності" (**Performance Baseline Watchdog - PBW**). **PBW** - це мікросервіс **ML**, який визначає, чи відповідає продуктивність кожного мікросервісу в системі очікуванням. Якщо продуктивність мікросервісу падає нижче очікуваного рівня на певну величину, **PBW** надсилає попередження до систем підтримки операцій або управління мережею. Якщо продуктивність падає ще нижче, **PBW** надсилає попередження до системи підтримки операцій (**Operation Support System - OSS**) або системи управління мережею (**Network Management System - NMS**) та може вжити заходів для автоматичного виправлення проблеми.

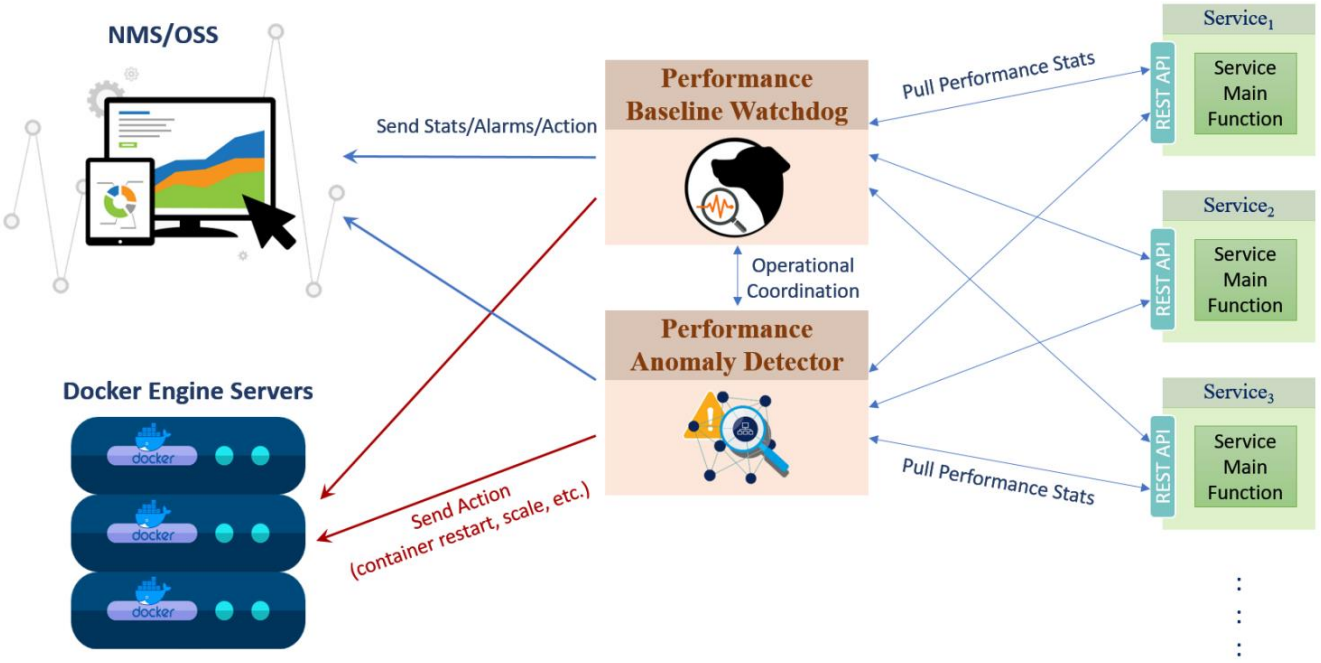


Рисунок 3.1 – PBW та **PAD** - мікросервіси штучного інтелекту для підвищення надійності та керованості системи **MSA**.

Таблиця 3.1 - Проблеми системи **MSA** та дії **PBW**.

Проблема	**Дія, що активується PBW**
Повільна відповідь або тайм-аути	Масштабування мікросервісу вертикально або горизонтально або перезавантаження мікросервісу
HTTP-помилки відповіді	Перевірка стану Apache, Flask, JVM, тома Docker, SQL-служби тощо. Перезавантаження служби за потреби
Мікросервіс не відповідає (вимкнений)	Перезавантаження контейнера мікросервісу

Другий мікросервіс штучного інтелекту - "Служба виявлення аномалій продуктивності" (**Performance Anomaly Detector - PAD**). **PAD** - це сервіс машинного навчання, який охоплює всю систему **MSA**. Він аналізує патерни продуктивності **MSA** і намагається виявити будь-яку незвичайну поведінку. **PAD** знаходить проблемні патерни у поведінці мікросервисів, автоматично виявляє проблеми перед її виникненням та активно діє для усунення неполадок.

Алгоритм **PBW** розраховує очікувані показники продуктивності на основі зібраної статистики продуктивності. Зібрана статистика продуктивності включає статистику часу відгуку **API**, помилки або рівень помилок окремих мікросервісів, коди відповіді **API** та навантаження, прикладене до самого мікросервісу.

Попередньо визначені дії запускаються залежно від того, наскільки мікросервіс відхиляється від розрахованого показника продуктивності.

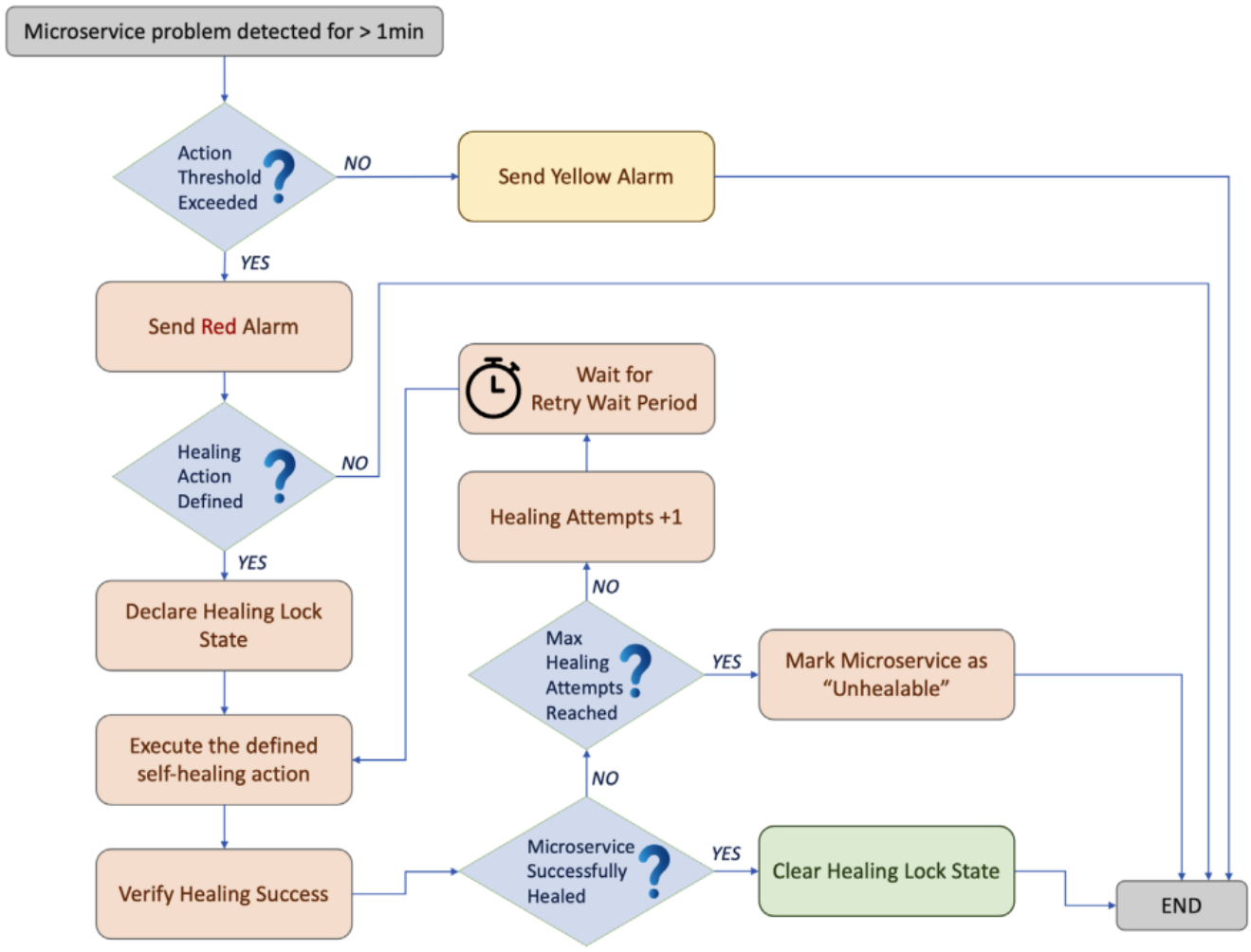


Рисунок 3.2 – Алгоритм самовідновлення мікросервісів (self-healing).

Таблиця 3.2 – Пояснення термінів до рисунку 3.2 .

Термін	Опис
Лікувальна дія (**Healing Action**):	Дія, яка виконується для виправлення неполадки.
Блокування мікросервісу для самовідновлення (**Healing Lock State**)	Стан мікросервісу, в якому тільки алгоритм самовідновлення може процювати з проблемним мікросервісом.
Період очікування перед повторною спробою (**Retry Wait Period**)	Час, який потрібно чекати, коли лікувальна дія не вдається, перш ніж спробувати знову. За замовчуванням період очікування перед повторною спробою становить 2 хвилини.
Стан невиліковності (**Unhealable State**)	Стан, в якому мікросервіс маркується невиправним після його невдалої спроби виправити його.
Максимальна кількість спроб відновлення (**Maximum Healing Attempts**)	Максимальна кількість спроб, яка здійснюється для виправлення мікросервісу, перш ніж промаркувати мікросервіс невиправним.

Залежно від конфігурації **PBW**, чим більше відхилення, тим більша ймовірність, що буде ініційовано проактивну дію, щоб спробувати самовідновитися. Однак у разі незначного відхилення не повинно спрацьовувати жодних дій з самовідновлення - достатньо попередження системи, яке інформує системного адміністратора.

У таблиці 3.1 наведено деякі з можливих системних проблем, з якими можна зіткнутися під час роботи системи та дії, які служба **PBW** вживатиме для спроби виправити проблему, а на рисунку 3.2 і таблиці 3.2 алгоритм самовідновлення мікросервісів (self-healing).

В **PBW** використовується для навчання і передбачення модель лінійної регресії, а в **PAD –** однокласова машина опорних векторів (**One-Class SVM**).

На відміну від традиційних машин опорних векторів, які використовуються для завдань класифікації, де дані мають мітки, **One-Class SVM** призначена для ситуацій, коли доступний лише один клас даних (дані без міток). Його основна мета - визначати і класифікувати нормальні точки даних від викидів або аномалій.

3.2 ВИСНОВОК ДО РОЗДІЛУ III

Впровадження методів **ML** у автомасштабування вебсервісів може забезпечити значні переваги та покращення ефективності системи **MSA**.

Використання **ML** особливо корисне для великих та складних систем, оскільки дозволяє виявляти закономірності у даних та взаємодії компонентів системи.

Запропоновані алгоритми **PBW** та **PAD** дають такі вигоди при використанні :

1. Покращена надійність системи: Ці алгоритми дозволяють системі реагувати на відхилення в продуктивності мікросервісів та виявляти аномалії, навіть до їх виникнення. Це дозволяє операторам системи приймати заходи для усунення проблем швидше та ефективніше, підвищуючи загальну надійність системи.
2. Оптимізація продуктивності: Швидке виявлення проблем та автоматичне виправлення дозволяє уникнути втрати продуктивності. Вчасне реагування на відхилення допомагає підтримувати стабільність та оптимальний рівень роботи системи, що в свою чергу сприяє покращенню її продуктивності.
3. Попереднє виявлення проблем: **PAD** допомагає виявляти аномальні патерни або незвичайну поведінку, ще до того, як вони можуть спричинити серйозні проблеми. Це дозволяє системі попередити відмови або зниження продуктивності, надаючи можливість операторам підготуватися до можливих проблем та запобігти їх поширенню.

4. РОЗДІЛ IV. ІНТЕЛЕКТУАЛЬНА СИСТЕМА МАСШТАБУВАННЯ ВЕБСЕРВИСІВ НА БАЗІ MSA З ВИКОРИСТАННЯМ ML

4.1 АПАРАТНА КОНФІГУРАЦІЯ СЕРВЕРА ДЛЯ ДОСЛІДЖЕННЯ, ТЕСТУВАННЯ, ТАРОЗГОРТАННЯ СИСТЕМИ

Апаратна конфігурація сервера є дуже важливою для успішного дослідження, тестування та розгортання системи.

Правильна конфігурація сервера може забезпечити достатню продуктивність та надійність.

Для проведення практичних досліджень зібраний персональний комп'ютер відповідно до апаратних характеристик, вказаних у таблиці 4.1.

Таблиця 4.1 – Апаратна конфігурація сервера

Найменування	Модель та кількість	Характеристики
Процесор	Intel® Xeon® E5-2670 v3 – 1 шт.	Cache - 30 MB Intel® Smart Cache / Frequency - 2.30 GHz / Total Cores – 12; Total Threads - 24; Max Turbo Frequency - 3.10 GHz
Материнська плата	Machinist X99-K9 – 1 шт.	Chipset - Intel C612 (Wellsburg-G)
Оперативна пам'ять	Micron 36ASF2G72PZ-2G1A2 - 2 шт.	Тип - DDR4 SDRAM/ Загальний обсяг – 32GB
Диск	SSDPR-CX400-512-G2 – 1 шт.	Тип – SATA3 SSD TLC/ Загальний обсяг – 512 GB
Мережевий адартер	RTL8168/8111 PCI-E Gigabit Ethernet	Максимальна швидкість зєднання - 1000 Mbps.
Блок живлення	FSP 500W (H3-500)	Тип – ATX / Потужність – 500 Вт

4.2 ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ДЛЯ РОЗРОБКИ, ДОСЛІДЖЕННЯ, ТЕСТУВАННЯ ТА РОЗГОРТАННЯ СИСТЕМИ

Для розробки інтелектуальної системи масштабування вебсервисів на базі **MSA** з використанням **ML** обрано програмне забезпечення (**ПЗ**) **PyCharm.**

PyCharm — це популярне інтегроване середовище розробки (**IDE - Integrated development environment**) для розробки на мові програмування Python компанії JetBrains, яке надає широкий спектр функцій, включаючи підсвічування синтаксису, автодоповнення коду та відлагодження.

PyCharm доступне в двох виданнях: Community та Professional. Видання Community є безкоштовним та відкритим, а видання Professional є комерційним продуктом, який включає додаткові функції, такі як рефакторинг коду та підтримка наукових обчислень. JetBrains надає студентам та викладачам безкоштовні ліцензіїі для PyCharm Professional та для інших своїх продуктів.

Для проєктування і управлінням базами даних обрано ПЗ **DBeaver Community**.

DBeaver Community - це безкоштовне ПЗ з відкритим кодом (Open Sourse) для управління базами даних, яке підтримує широкий спектр баз даних, включаючи MySQL, PostgreSQL, Oracle, Microsoft SQL Server та інші.

DBeaver Community має широкий спектр функцій для проектування баз даних, включаючи:

- створення та редагування моделей даних;
- візуалізація структури баз даних;
- створення та редагування таблиць, записів та запитів;
- виконання запитів до баз даних;
- менеджер об'єктів бази даних;
- менеджер транзакцій.

В якості операційної системи для розгортання обрано **Ubuntu Server.**

Ubuntu Server – це операційна серверна система на базі дистрибутива Debian GNU/Linux. Вона заснована на ядрі Linux, яке забезпечує високу продуктивність, безпеку та широку підтримку для різних типів апаратного забезпечення.

Ubuntu Server має ряд переваг, включаючи:

- широкий вибір установлених програм, включаючи веб-сервери, сервери баз даних, файлові сервери та інші серверні додатки.
- довготривала підтримка версій, яка становить 5 років для звичайних випусків та 10 років для випусків LTS;
- зручний інтерфейс користувача, який полегшує управління та налаштування сервера.
- **Ubuntu Server** успадковує деякі основні риси Debian, такі як стабільність, надійність та велика кількість пакунків програмного забезпечення, але Ubuntu відрізняється від Debian спрощеним інтерфейсом та орієнтацією на кінцевого користувача, що робить **Ubuntu Server** більш привабливішою для широкого кола користувачів.

В якості системи управління базами (СУБД) даних обрано MySQL.

MySQL - це відкрита СУБД, яка використовується для зберігання та керування даними. Вона є однією з найпопулярніших СУБД у світі і її використовують, як маленькі, так і великі установи та компанії.

Для розгортання системи **MSA** використовується ПЗ **Docker.**

Docker - це платформа програмного забезпечення, яка дозволяє розробникам, системним адміністраторам та інженерам розробляти, розгортати та керувати програмним забезпеченням у контейнерах. Контейнери - це невеликі, ізольовані середовища виконання, які можна використовувати для запуску програмного забезпечення.

Docker працює, створюючи образ контейнера, який є набором інструкцій, необхідних для створення контейнера. Образ контейнера містить операційну систему, бібліотеки та програми, необхідні для запуску програми.

Контейнери схожі на віртуальні машини (VM), але мають кілька принципових відмінностей. Як показано на рисунку 4.1, контейнери використовують одне і те ж ядро операційної системи хоста, але ізолюють і обмежують виділені ресурси, що дає щось схоже на VM, але набагато легше з точки зору використання ресурсів.

У таблиці 4.2 надано порівняння характеристик контейнерів та віртуальних машин.

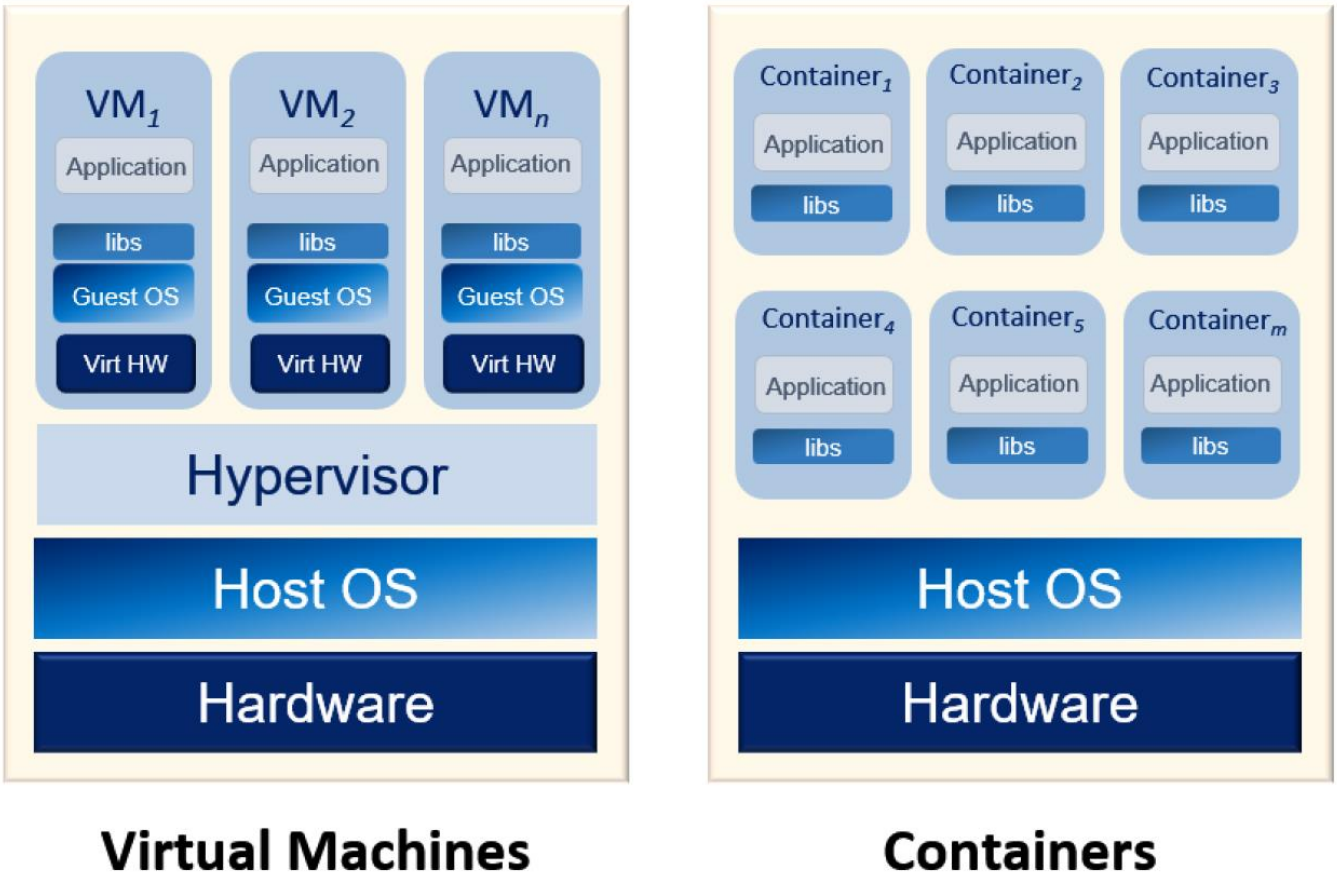


Рисунок 4.1 - Порівняння контейнерів з віртуальними машинами.

Таблиця 4.2 – Порівняння контейнерів з віртуальними машинами.

Характеристика	Контейнери	Віртуальні машини
Використання ресурсів та накладні витрати	Легкі, більш ефективне використання ресурсів.	Високий рівень накладних витрат та велика інтенсивність використання ресурсів
Розміри контейнера та додатку	Середні розміри від 5 до 20 МБ	Розміри вимірюються в сотнях МБ або ГБ

Продовження таблиці 4.2 - Порівняння характеристик контейнерів з віртуальними машинами.

Характеристика	Контейнери	Віртуальні машини
Продуктивність	Висока продуктивність	Нижча продуктивність
Розміри контейнера та додатку	Середні розміри від 5 до 20 МБ	Розміри вимірюються в сотнях МБ або ГБ
Масштабованість	Легка масштабованість/висока горизонтальна масштабованість	Масштабування важче та вимагає більше ресурсів
Час завантаження	Дуже короткий час запуску (1-3 секунди)	Час запуску - хвилини
Переносимість	Системонезалежна та високо переносима	Обмежена переносимість
Придатність для DevOps та CI/CD	Сприяє більш гнучкому DevOps та плавнішому CI/CD	Може уповільнити операції CI/CD
Доступ до обладнання хоста	Додатки мають доступ до обладнання безпосередньо	Немає прямого доступу до обладнання
Безпека	Менше безпеки; використовує те саме ядро операційної системи	Більш безпечно; кожна ВМ має своє власне ядро операційної системи

4.3 ПІДГОТОВЧІ КРОКИ ДЛЯ РОЗГОРТАННЯ СИСТЕМИ

Після вибору апаратного забезпечення (див. Таблиця 4.1) встановлюється операційна система **Ubuntu Server**.

Під час встановлення операційної системи **Ubuntu Server** на підготовлене апаратне забезпечення необхідно виконати такі кроки:

4. Завантаження образу ISO **Ubuntu Server** з офіційного веб-сайту - https://ubuntu.com/download/server.
5. Створення завантажувального носія USB або DVD за допомогою програм, таких як Rufus, Etcher, UNetbootin або dd(Command Line Linux).
6. Зміна налаштувань BIOS/UEFI для вибору пристрою завантаження для USB або DVD.
7. Перезавантажтення комп'ютера з завантажувального носія та запуск встановлення.
8. Вибір мови та налаштувань інших основні налаштування системи.
9. Налаштування мереж та введення інформації для підключення до мережі, якщо це необхідно.
10. Вибір диска або розділу диска для встановлення Ubuntu Server.
11. Створення облікового запису користувача та ведення даних користувача (ім'я, пароль тощо).
12. Встановлення додаткових компоненті: SSH – server для безпечного віддаленого доступу до сервера через зашифроване з'єднання.
13. Завершення встановлення та перезавантаження.

14. Вхід до системи через SSH за допомогою SSH клієнта, таких як PuTTY (для Windows), OpenSSH (вбудований в багато Unix-подібних систем, включаючи Linux і macOS), або будь-який інший SSH клієнт (див. рисунок 4.2).

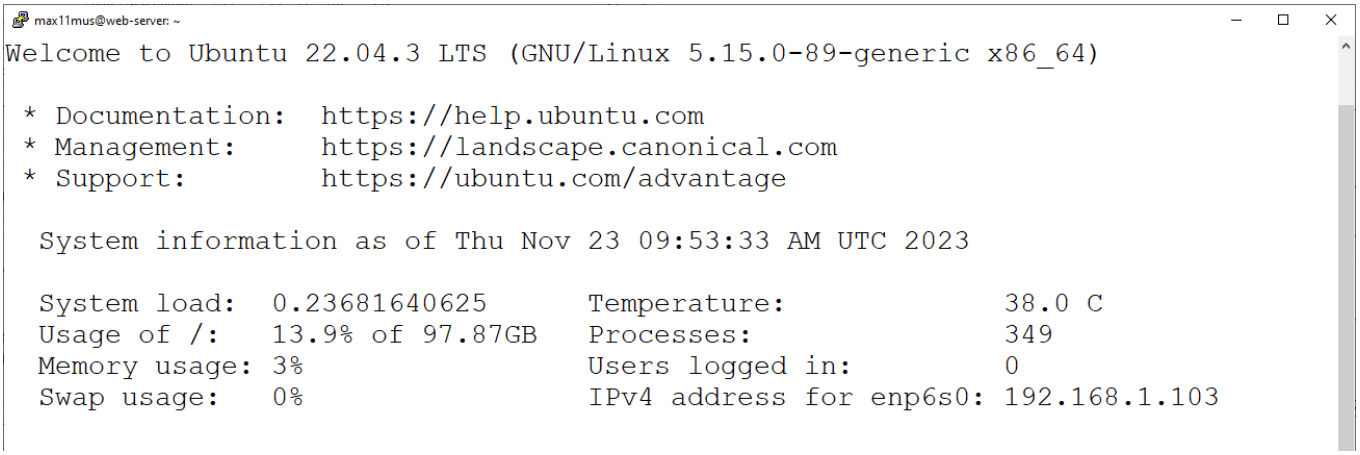


Рисунок 4.2 - Підключення до серверу за допомогою SSH клієнта PuTTY.

Після встановлення операційної системи **Ubuntu Server** встановлюється та налаштовується **Docker** зідно інструкції розташованої на https://docs.docker.com/engine/install/ubuntu/.

Bash - скрипт для встановлення **Docker** надається в додатку А.

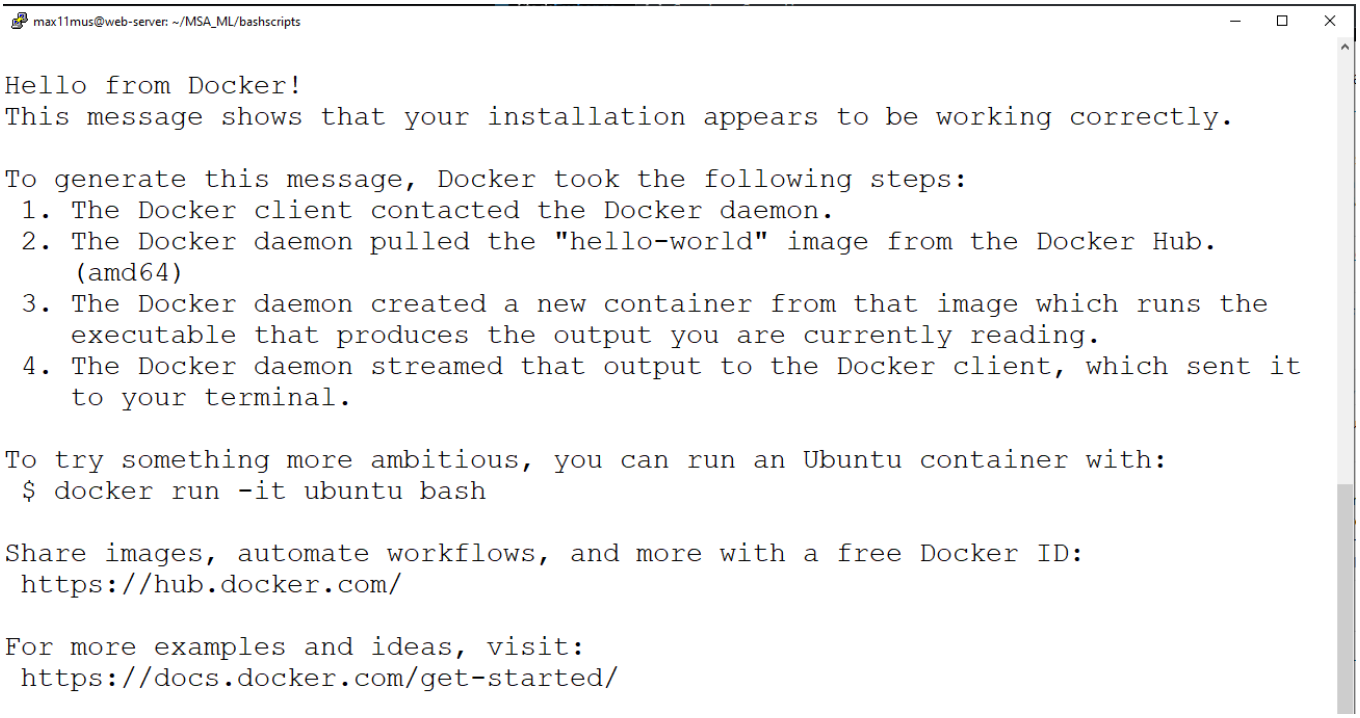


Рисунок 4.3 – Результат виконання Bash - скрипт для встановлення Docker.

4.4 АРХІТЕКТУРА СИСТЕМИ MSA

Розглянемо архітектуру системи MSA на прикладі додатку для вебмагазину.

Додаток для вебмагазину - це спрощена гіпотетична система замовлення товарів, спеціально розроблена для демонстрації процесу та кроків, необхідних для впровадження **ML** в систему **MSA**, щоб показати застосування концепцій та методології.

У додатку користувач може розмістити замовлення з існуючого каталогу продуктів та відстежувати статус доставки замовлення. Для спрощення, всі продажі є остаточними, і товари не підлягають поверненню.

Система зможе провести оплату замовлення, призначити кур'єра для доставки замовлення та відстежувати всі оновлення стосовно замовлення та доставки.

Спочатку необхідно зрозуміти, які поточні функції реалізовані в системі.

В таблиці 4.3 перераховані функції системи тай їх опис.

Таблиця 4.3 – Опис функцій системи.

Функція	Опис
place_order()	Функція призначена для створення нового запису з усією інформацією про замовлення в системі. Вона зберігає всю необхідну інформацію про замовлення і позначає його статус як "очікується", що означає - замовлення створено, але ще не завершено, і очікує подальшого оброблення та завершення процесу оформлення. Ця функція є початковим етапом процесу оформлення замовлення в системі.
check_inventory()	Функція перевірки наявності товару в оформленому замовленні.
verify_cc_payment()	Функція перевірки оплати загальної суми замовлення кредитною картою. Повертає код помилки, якщо оплата не була проведена.
verify_paypal_payment()	Функція перевірки оплати загальної суми замовлення через PayPal. Повертає код помилки, якщо оплата не була проведена.
update_inventory()	Функція, яка після підтвердження замовлення та успішної оплати, оновлює запаси товарів.
create_order()	Функція, яка після успішної обробки замовлення, змінює статус замовлення та розпочинає процес підготовки замовлення.
create_shipping_request()	Функція, яка формує запит на доставку замовлення та повідомляє кур'єру про наявне замовлення для доставки.
order_status_update()	Функція для оновлення статусу замовлення: підготовка, доставка, отримання і т.д.
shipment_status_update()	Функція для оновлення статусу доставки :очікування відправлення, у дорозі, отримано і т.д.
web_msg_notification()	Функція для повідомлення користувача про будь-які зміни або оновлення в оформленому замовленні, через вебповідомлення.
email_notification()	Функція для повідомлення користувача про будь-які зміни або оновлення в оформленому замовленні, через електрону пошту.

Продовження таблиці 4.3 – Опис функцій системи.

Функція	Опис
sms_notification()	Функція для повідомлення користувача про будь-які зміни або оновлення в оформленому замовленні, через СМС-повідомлення.
register_customer()	Функція для створення акаунта клієнта із зазначенням повного імені, адреси, номера телефону та інших деталей.

На малюнку 4.4 показано робочий процес розміщення замовлень та послідовність виклику функцій.

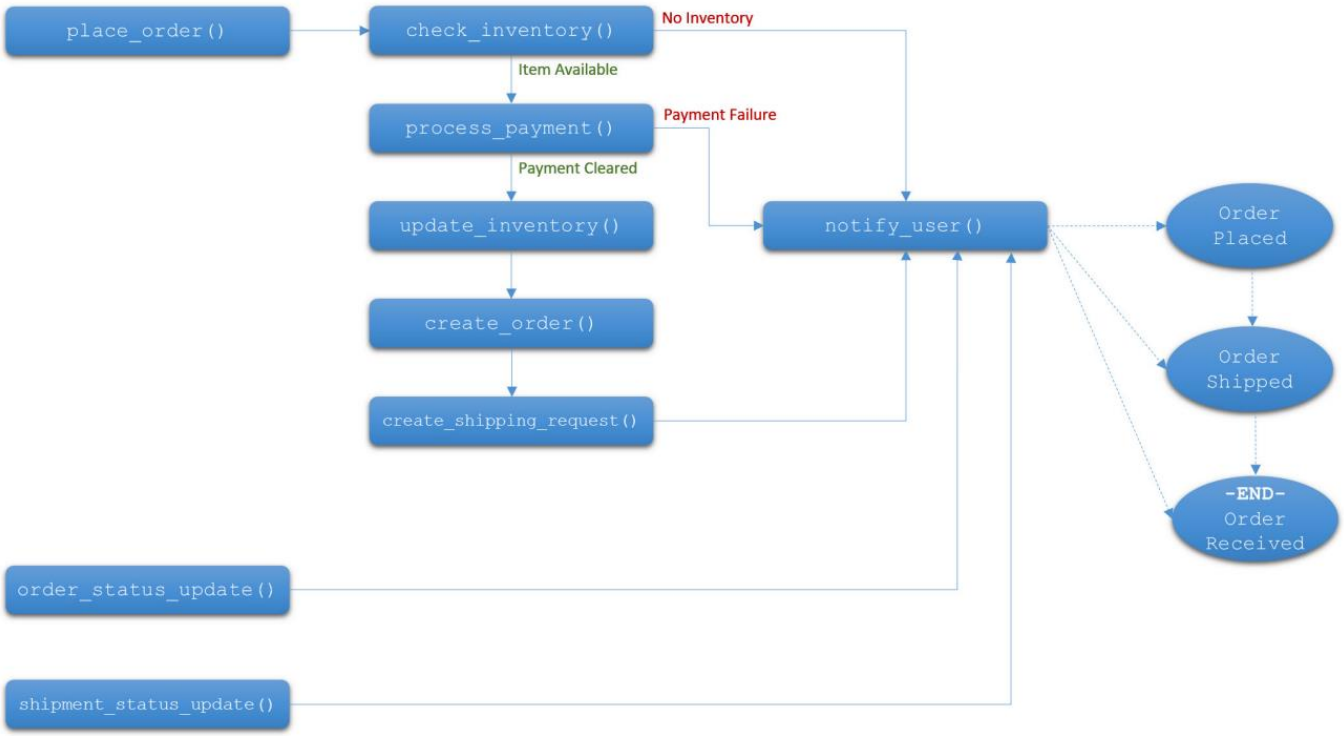


Рисунок 4.4 – Робочий процес розміщення замовлень та послідовність виклику функцій.

На рисунку 4.5 - діаграма, яка показує, структуру бази даних і до яких частин бази даних мають доступ функції із таблиці 4.3.

Для побудови додатку використовується **Flask**, легкого та гнучкого вебфреймворку для розробки веб-додатків на мові програмування **Python**, який написан на чистому **Python**, без використання сторонніх бібліотек. Це робить його легким і швидким у використанні.

Додаток складається з таких мікросервісів:

1. Інтерфейс фронтенду для панелі керування веб-додатком (dashboard_ms.py – Додаток Б);
2. Управління клієнтами (customer_management_ms.py – Додаток В);
3. Управління продуктами (product_management_ms.py – Додаток Г);
4. Управління інвентарем (inventory_management_ms.py – Додаток Ґ);

5. Управління платежами (payment_management_ms.py - Додаток Д).

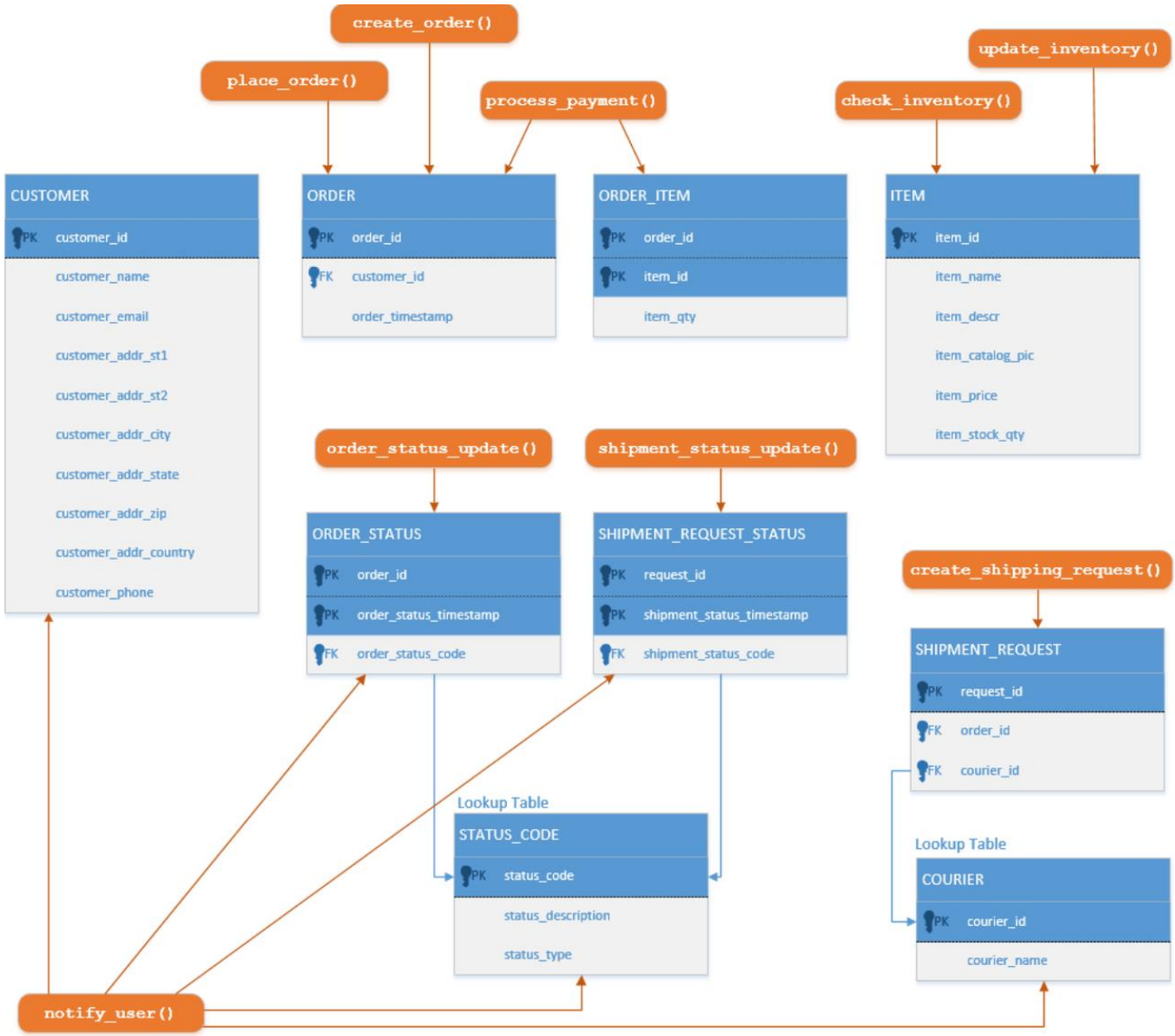


Рисунок 4.5 – Робочий процес розміщення замовлень
та послідовність виклику функцій.

4.5 РОЗГОРТАННЯ СИСТЕМИ MSA

Для розгортання системи **MSA** використовується Docker

Для запуску кожного мікросервісу створено Dockerfile - текстовий файл, який містить інструкції для побудови Docker-образу та використовується для автоматизації процесу створення образів контейнерів:

1. Dockerfile для dashboard_ms.py (Додаток Е);

2. Dockerfile для customer_management_ms.py (Додаток Є);
3. Dockerfile для product_management_ms.py (Додаток Ж);
4. Dockerfile для inventory_management_ms.py (Додаток З);
5. Dockerfile для payment_management_ms.py (Додаток И).

Для розгортання всієї системи використовується інструмент Docker Compose, який дозволяє вам конфігурувати та запускати багато контейнерні додатки.

Файл docker-compose.yml (Додаток І) містить інформацію про всі складові додатку, такі як контейнери, мережі, томи для зберігання даних тощо. Цей файл дозволяє вам визначити параметри для кожного контейнера, такі як образ, з якого створюється контейнер, порти, які потрібно відкрити, змінні середовища, команди для виконання при створенні контейнера і багато іншого.

Команда ‘docker-compose up -d –build’ використовується для одночасної побудови образів та запуску контейнерів, описаних у файлі docker-compose.yml, в фоновому режимі (див. рисунок 4.6).

Рисунок 4.6 – Запуск системи за допомогою команди ‘docker-compose up -d –build’.

На рисунку 4.7 відображена інформація про активні контейнери, яка отримана за допомогою інструмента для управління контейнерами Portainer.

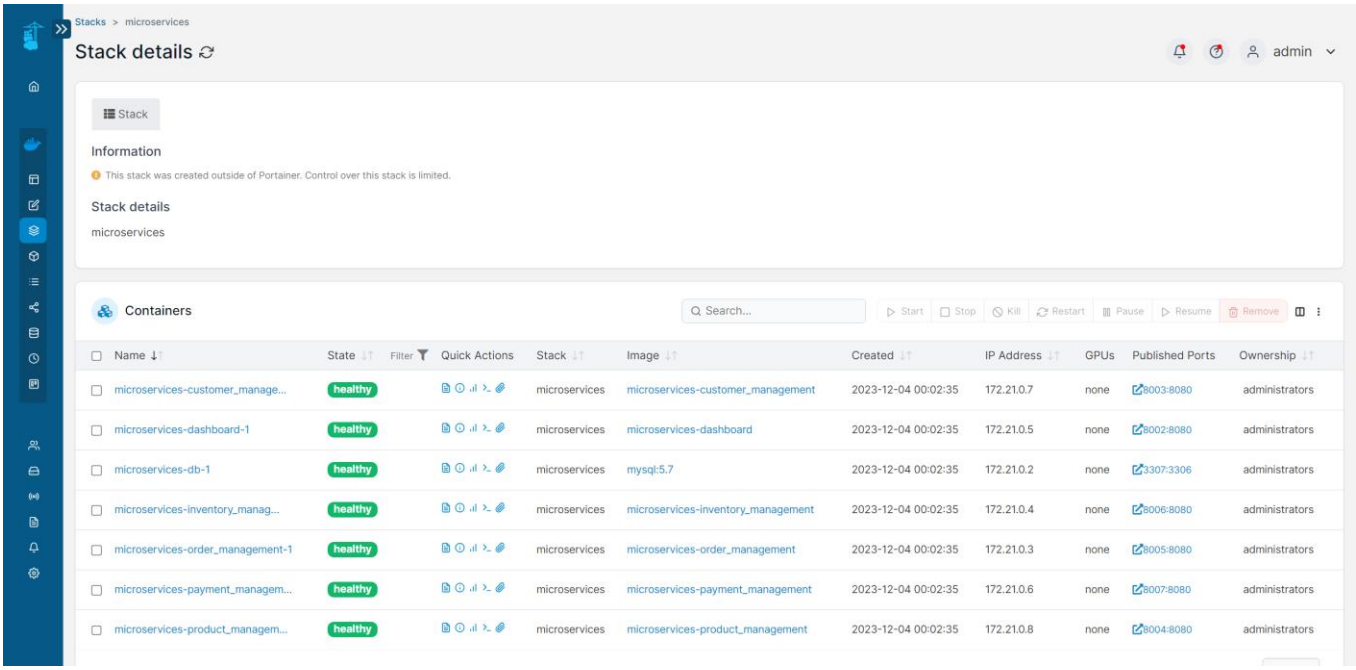


Рисунок 4.7 – Інформація про активні контейнери, яка отримана за допомогою інструмента для управління контейнерами Portainer.

4.6 РЕАЛІЗАЦІЯ АЛГОРИТМІВ ML В СИСТЕМІ MSA

В цьому підрозділі зроблена реалізація алгоритмів запропонованих у розділі III.

Для збору навчальних даних для моделей машинного навчання створено декілька інструментів (Додаток Ї, Й) для збору навчальних та тестових даних, імітації навантаження системи та мікросервісів, а також для вимірювання продуктивності мікросервісів.

Генератор трафіку **API** (simulate_api_rqsts.py - Додаток Ї) допомагає імітувати навантаження запитів **API** для одного або кількох мікросервісів системи. Генератор створює багатопотокові запити **API** до кількох цільових мікросервісів. HTTP-запити API надсилаються до кожного мікросервісу паралельно. Навантаження API вимірюється кількістю запитів на хвилину, а запити **API** можуть бути рівномірно або випадково розподілені.

У режимі рівномірного розподілу запитів, час між кожним викликом API завжди однаковий. Наприклад, якщо налаштовано рівномірно розподілене навантаження в 600 запитів в хвилину, генератор надсилатиме виклик **API** кожні 100 мс. Цей режим корисний для аналізу реакції конкретного мікросервісу на навантаження **API**.

У режимі випадкового розподілу кожен виклик **API** надсилається через випадковий період, з моменту, коли був надісланий попередній виклик. Однак

завжди залишається в межах від 5 мс до 195 мс відносно. Цей режим краще відображає навантаження запитів **API** в реальному часі.

Монітор продуктивності мікросервісів (ms_perfmon.py - Додаток Й) є ще одним багатопотоковим інструментом і використовується для збору та створення навчальних даних для моделей штучного інтелекту.

Монітор продуктивності надсилає паралельні виклики API до кожного мікросервісу в системі, а потім реєструє адресу виклику API, дату та час його надсилання, час відгуку мікросервісу, код HTTP-відповіді, а також через Docker API різні статистичні дані та інформацію про контейнери. На рисунку 4.8 показано приклад журналу зібраних даних у форматі CSV.

На початковому етапі роботи інтелектуальна система масштабування буде працювати в ідеальному середовищі симуляції (без помилок або затримок) для збору навчальних даних, які необхідні для розробки моделей штучного інтелекту. Після отримання достатньої кількості даних система буде готова обробляти реальний трафік (Див. рисунок 4.9).

Основні кроки ініціалізації системи:

1. Запуск системи без сервісів штучного інтелекту для збору навчальних даних.
2. Очищення зібраних даних від аномалій, якщо такі є (Додаток М - training_data_cleanup.py).
3. Навчання алгоритмів штучного інтелекту на зібраних навчальних даних, та побудова моделей (Додаток К – PBW_learn.py, Додаток Л - PAD_learn.py).
4. Повторна ініціалізація системи з усіма сервісами штучного інтелекту.
5. Початок роботи в реальних умовах через введення помилок, затримок у відповідях на дані та імітацію відмов сервісів і т.д.

Інструмент “ms_perfmon.py” (Додаток Й) створює окремі файли статистики для кожного мікросервісу у директорії “perfmon_stats”. Важливо залишати цей інструмент запущеним і моніторити статистику продуктивності системи при мінімальному навантаженні. Дані про продуктивність збираються кожні 10 секунд.

Інструмент “training_data_cleanup.py” автоматично очищує файли даних про продуктивність у директорії “perfmon_stats” та створює директорію “scrubbed_stats” з очищеними даними для кожного мікросервісу. Ці очищені файли будуть використані для подальшого навчання моделей штучного інтелекту(Додаток К – PBW_learn.py, Додаток Л - PAD_learn.py).

Для того щоб сервіси штучного інтелекту (**PBW** і **PAD**) могли керувати (запускати, зупиняти та перезапускати) контейнерами **Docker** спочатку потрібно виконати такі кроки:

1. Внести зміни в конфігурацію **Docker** “/lib/systemd/system/docker.service” в строку “ExecStart …” (див. рисунок 4.10).

2. Перезапустит **Docker** та перевірити доступність API(див. рисунок 4.11).

```
max11mus@web-server: ~/MSA_ML/microservices
url,request_datetime,response_time,status_code,container_name,cpu_usage,memory_u
sage,network_io
http://inventory_ms:8080,2023-12-05 15:02:00,0.594,200,container1,22%,264MB,6MB
http://inventory_ms:8080,2023-12-05 15:03:00,0.652,200,container1,27%,312MB,7MB
http://inventory_ms:8080,2023-12-05 15:04:00,0.710,200,container1,32%,360MB,8MB
http://inventory_ms:8080,2023-12-05 15:05:00,0.768,200,container1,37%,408MB,9MB
http://inventory_ms:8080,2023-12-05 15:06:00,0.826,200,container1,42%,456MB,10MB

http://inventory_ms:8080,2023-12-05 15:02:30,0.465,404,container2,16%,136MB,4MB
http://inventory_ms:8080,2023-12-05 15:03:30,0.523,200,container2,19%,154MB,5MB
http://inventory_ms:8080,2023-12-05 15:04:30,0.581,200,container2,22%,172MB,6MB
http://inventory_ms:8080,2023-12-05 15:05:30,0.639,200,container2,25%,190MB,7MB
http://inventory_ms:8080,2023-12-05 15:06:30,0.697,200,container2,28%,208MB,8MB

http://inventory_ms:8080,2023-12-05 15:03:00,0.774,200,container3,33%,528MB,11MB
http://inventory_ms:8080,2023-12-05 15:04:00,0.832,200,container3,38%,576MB,12MB
http://inventory_ms:8080,2023-12-05 15:05:00,0.890,200,container3,43%,624MB,13MB
http://inventory_ms:8080,2023-12-05 15:06:00,0.948,200,container3,48%,672MB,14MB
```

Рисунок 4.8 – Приклад журналу зібраних даних у форматі CSV монітором продуктивності мікросервісів.

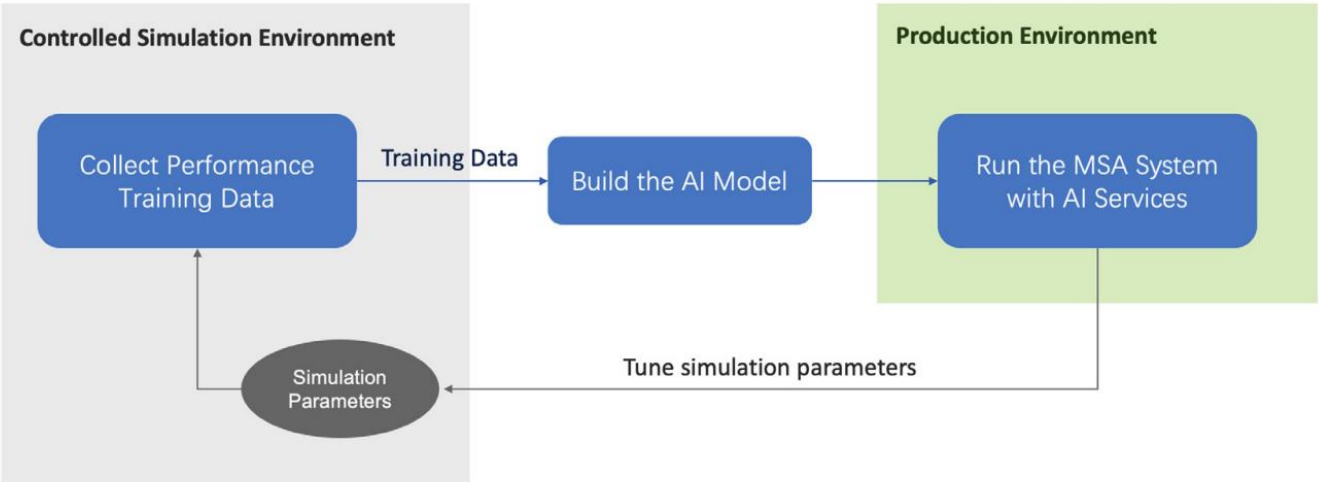


Рисунок 4.9 – Схема ініціалізації та запуску системи MSA з використанням штучного інтелекту.

Після того, було зібрано достатньо навчальних даних, можна ініціалізувати служби штучного інтелекту. Це можна зробити, використовуючи команду: “ nohup docker compose -f msa-ai.yaml up --build &”. (див. рисунок 4.12, Додаток Н)

Під час навчання **PBW** зміг побудувати модель та обчислити очікуваний час відгуку кожного мікросервісу в системі **MSA**. Як видно з наступного зразка логу, за нормального навантаження системи середній час відгуку мікросервісу inventory_ms становить близько 20 мс (див. рисунок 4.13).

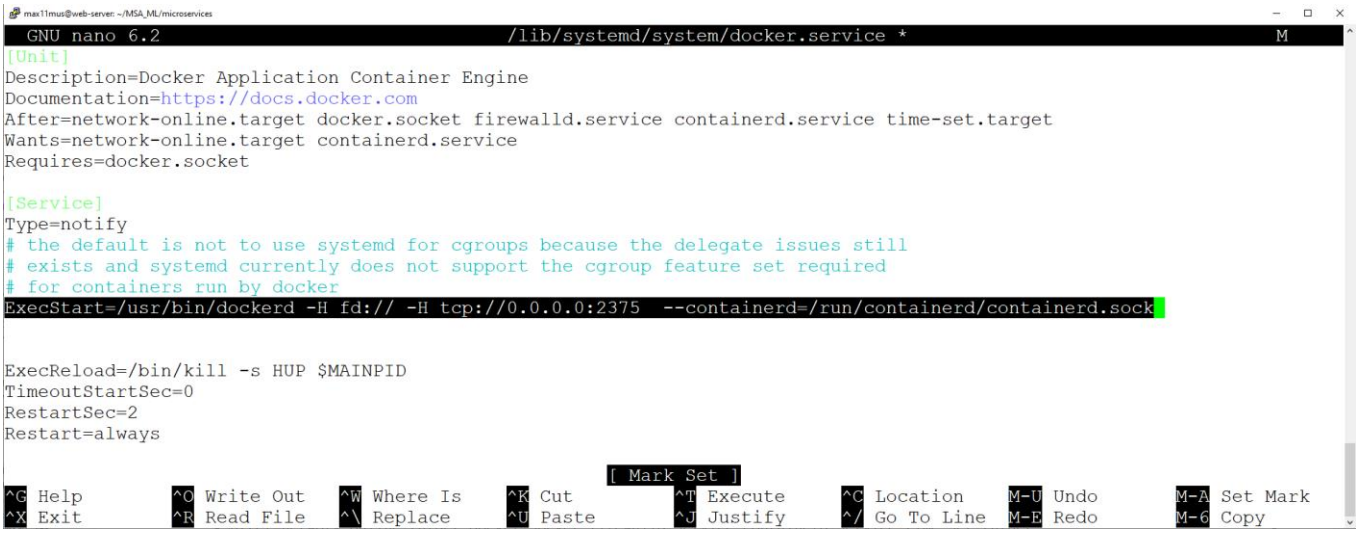


Рисунок 4.10 – Увімкнення з’єднання з **Docker** через tcp-порт 2375

Рисунок 4.11 – Перезапуск та перевірка з’єднання через tcp-порт 2375.

```
max11mus@web-server:~/MSA_ML/microservices$ nohup docker compose -f msa-ai.yaml up --build &
[1] 959967
max11mus@web-server:~/MSA_ML/microservices$ nohup: ignoring input and appending output to 'nohup.out'

max11mus@web-server:~/MSA_ML/microservices$ docker ps
CONTAINER ID   IMAGE                                COMMAND                  CREATED              STATUS
      PORTS                                                NAMES
77ec1f3f6f4f   microservices-pwd                     "/bin/sh -c 'python …"   About a minute ago   Up 33 seconds
      8080/tcp                                             microservices-pwd-1
cb8c43050744   microservices-pad                     "/bin/sh -c 'python …"   About a minute ago   Up 33 seconds
      8080/tcp                                             microservices-pad-1
9b07838a76e4   microservices-order_management        "/bin/sh -c 'python …"   About a minute ago   Up About a minute (healt
hy)   0.0.0.0:8005->8080/tcp, :::8005->8080/tcp            microservices-order_management-1
9f3400dc8c00   microservices-customer_management     "/bin/sh -c 'python …"   About a minute ago   Up About a minute (healt
hy)   0.0.0.0:8003->8080/tcp, :::8003->8080/tcp            microservices-customer_management-1
c033b5c5db06   microservices-product_management      "/bin/sh -c 'python …"   About a minute ago   Up About a minute (healt
hy)   0.0.0.0:8004->8080/tcp, :::8004->8080/tcp            microservices-product_management-1
66be8da21457   microservices-payment_management      "/bin/sh -c 'python …"   About a minute ago   Up About a minute (healt
hy)   0.0.0.0:8007->8080/tcp, :::8007->8080/tcp            microservices-payment_management-1
1b00b0ecceab   microservices-dashboard               "/bin/sh -c 'python …"   About a minute ago   Up About a minute (healt
hy)   0.0.0.0:8002->8080/tcp, :::8002->8080/tcp            microservices-dashboard-1
51e550e66a3d   microservices-inventory_management    "/bin/sh -c 'python …"   About a minute ago   Up About a minute (healt
hy)   0.0.0.0:8006->8080/tcp, :::8006->8080/tcp            microservices-inventory_management-1
f3c51f32df4e   mysql:5.7                            "docker-entrypoint.s…"   About a minute ago   Up About a minute (healt
hy)   33060/tcp, 0.0.0.0:3307->3306/tcp, :::3307->3306/tcp  microservices-db-1
b235bcb96fb6   portainer/portainer                  "/portainer"             9 days ago           Up 2 hours
      8000/tcp, 9443/tcp, 0.0.0.0:9000->9000/tcp, :::9000->9000/tcp   portainer
max11mus@web-server:~/MSA_ML/microservices$
```

Рисунок 4.12 – Запуск системи **MSA** з усіма сервісами штучного інтелекту.

Поріг попередження для **PBW** сконфігуровано на рівні 250 мс, а поріг дії – 750 мс. Зараз відбувається введення навантаження API до мікрослужби Інвентаризації за допомогою симуляції запитів та затримок відповіді.

Наведено фрагмент журналу дій **PBW** для перевірки роботи алгоритма (див. рисунок 4.14).

```
max11mus@web-server: ~/MSA_ML/microservices
url,request_datetime,response_time,status_code,container_name,cpu_usage,memory_u
sage,network_io
http://inventory_ms:8080,2023-12-05 15:02:00,0.594,200,container1,22%,264MB,6MB
http://inventory_ms:8080,2023-12-05 15:03:00,0.652,200,container1,27%,312MB,7MB
http://inventory_ms:8080,2023-12-05 15:04:00,0.710,200,container1,32%,360MB,8MB
http://inventory_ms:8080,2023-12-05 15:05:00,0.768,200,container1,37%,408MB,9MB
http://inventory_ms:8080,2023-12-05 15:06:00,0.826,200,container1,42%,456MB,10MB

http://inventory_ms:8080,2023-12-05 15:02:30,0.465,404,container2,16%,136MB,4MB
http://inventory_ms:8080,2023-12-05 15:03:30,0.523,200,container2,19%,154MB,5MB
http://inventory_ms:8080,2023-12-05 15:04:30,0.581,200,container2,22%,172MB,6MB
http://inventory_ms:8080,2023-12-05 15:05:30,0.639,200,container2,25%,190MB,7MB
http://inventory_ms:8080,2023-12-05 15:06:30,0.697,200,container2,28%,208MB,8MB

http://inventory_ms:8080,2023-12-05 15:03:00,0.774,200,container3,33%,528MB,11MB
http://inventory_ms:8080,2023-12-05 15:04:00,0.832,200,container3,38%,576MB,12MB
http://inventory_ms:8080,2023-12-05 15:05:00,0.890,200,container3,43%,624MB,13MB
http://inventory_ms:8080,2023-12-05 15:06:00,0.948,200,container3,48%,672MB,14MB
```

Рисунок 4.13 – Журнал зібраних даних у форматі CSV для мікросервісу inventory_ms.

max11mus@web-server: ~/MSA_ML/microservices

```
2023-11-23 18:24:00.518005: Alarming high response time
(0.6386377334594727) detected in inventory_ms. No alarm
triggered yet.
2023-11-23 18:24:11.469172: Alarming high response time
(0.7164063453674316) detected in inventory_ms. No alarm
triggered yet.
2023-11-23 18:24:22.203452: Alarming high response time
(0.7233438491821289) detected in inventory_ms. No alarm
triggered yet.
2023-11-23 18:24:32.942619: Alarming high response time
(0.7101089954376221) detected in inventory_ms. No alarm
triggered yet.
2023-11-23 18:24:43.668907: Alarming high response time
(0.6982685089111328) detected in inventory_ms. No alarm
triggered yet.
2023-11-23 18:24:54.777383: Actionable high response time
(0.8207950115203857) detected in inventory_ms. No action
triggered yet.
2023-11-23 18:29:31.852330: Actionable high response time
(1.326528787612915) detected in inventory_ms. No action
triggered yet. Yellow alarm triggered and sent to NMS/OSS
system.
2023-11-23 18:29:43.196200: Actionable high response time
(1.4279899597167969) detected in inventory_ms. No action
triggered yet. Yellow alarm triggered and sent to NMS/OSS
system.
2023-11-23 18:30:05.310226: Actionable high response time
(1.0108487606048584) detected in inventory_ms. No action
triggered yet. Yellow alarm triggered and sent to NMS/OSS
system.
2023-11-23 18:30:16.334608: Actionable high response time
(1.1380960941314697) detected in inventory_ms. Red Alarm
triggered and sent to NMS/OSS system.
2023-11-23 18:30:16.334608: Self-healing lock state declared
for inventory_ms container.
2023-11-23 18:30:16.334608: Self-healing action triggered.
Restarting inventory_ms container (inventory_management_
container).
2023-11-23 18:30:21.359377: Verifying inventory_ms
operations...
2023-11-23 18:30:22.945823: inventory_ms was successfully
restarted
2023-11-23 18:30:23.089051: Self-healing lock state cleared for
inventory ms container.
```

Рисунок 4.14 – Фрагмент журналу дій **PBW**.

Найкращий спосіб продемонструвати роботу **PAD** - це симулювати каскадне відмовлення і переглянути, як **PAD** може повернути систему **MSA** до нормальної роботи. Патерн **CIRCUIT BREAKER**, про який йшлося у підрозділі 2.5, допомагає запобігти каскадній відмові, але він все ще не може вирішити проблем відказу мікросервісу у традиційній реалізації патерну до тих пір, поки вручну не буде виправлено проблему у самому мікросервісі.

Наведено фрагмент журналу дій **PAD** для перевірки роботи алгоритма (див. рисунок 4.15).

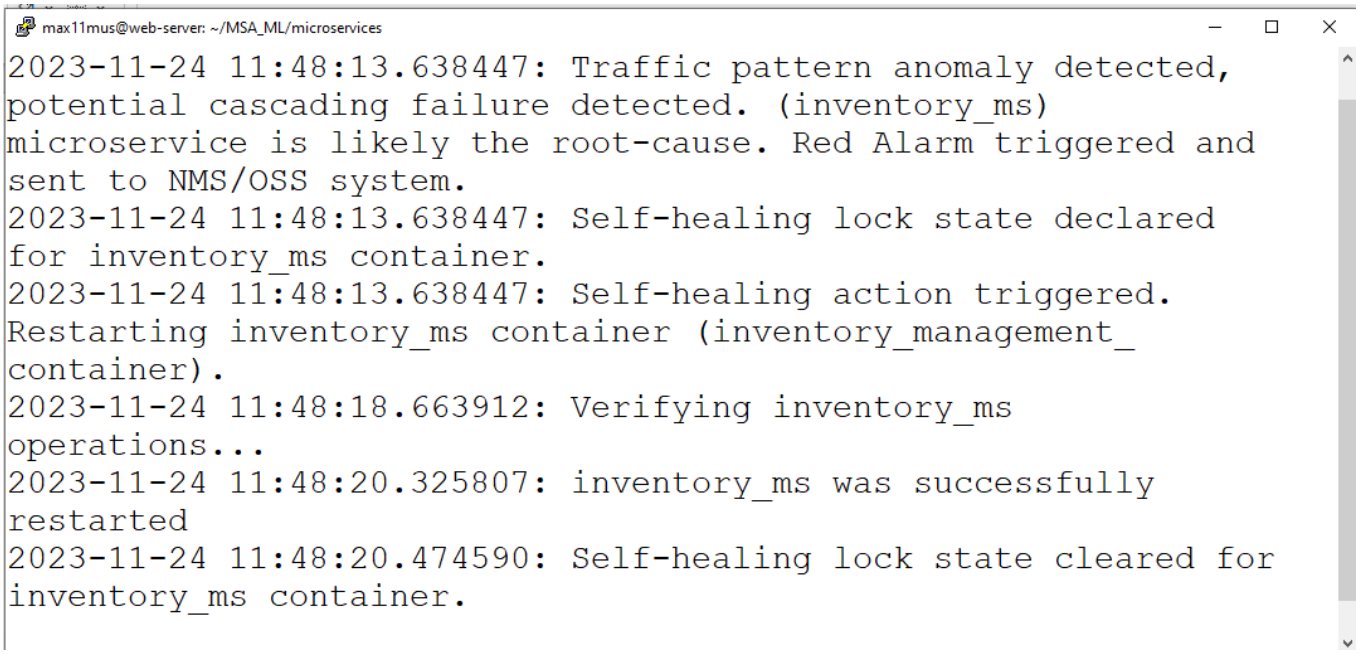


Рисунок 4.15 – Фрагмент журналу дій **PAD**.

max11mus@web-server: ~/MSA_ML/microservices

```
2023-11-23 18:24:00.518005: Alarming high response time
(0.6386377334594727) detected in inventory_ms. No alarm
triggered yet.
2023-11-23 18:24:11.469172: Alarming high response time
(0.7164063453674316) detected in inventory_ms. No alarm
triggered yet.
2023-11-23 18:24:22.203452: Alarming high response time
(0.7233438491821289) detected in inventory_ms. No alarm
triggered yet.
2023-11-23 18:24:32.942619: Alarming high response time
(0.7101089954376221) detected in inventory_ms. No alarm
triggered yet.
2023-11-23 18:24:43.668907: Alarming high response time
(0.6982685089111328) detected in inventory_ms. No alarm
triggered yet.
2023-11-23 18:24:54.777383: Actionable high response time
(0.8207950115203857) detected in inventory_ms. No action
triggered yet.
2023-11-23 18:29:31.852330: Actionable high response time
(1.326528787612915) detected in inventory_ms. No action
triggered yet. Yellow alarm triggered and sent to NMS/OSS
system.
2023-11-23 18:29:43.196200: Actionable high response time
(1.4279899597167969) detected in inventory_ms. No action
triggered yet. Yellow alarm triggered and sent to NMS/OSS
system.
2023-11-23 18:30:05.310226: Actionable high response time
(1.0108487606048584) detected in inventory_ms. No action
triggered yet. Yellow alarm triggered and sent to NMS/OSS
system.
2023-11-23 18:30:16.334608: Actionable high response time
(1.1380960941314697) detected in inventory_ms. Red Alarm
triggered and sent to NMS/OSS system.
2023-11-23 18:30:16.334608: Self-healing lock state declared
for inventory_ms container.
2023-11-23 18:30:16.334608: Self-healing action triggered.
Restarting inventory_ms container (inventory_management_
container).
2023-11-23 18:30:21.359377: Verifying inventory_ms
operations...
2023-11-23 18:30:22.945823: inventory_ms was successfully
restarted
2023-11-23 18:30:23.089051: Self-healing lock state cleared for
inventory ms container.
```

Рисунок 4.14 – Фрагмент журналу дій **PBW**.

4.7 ВИСНОВОК ДО РОЗДІЛУ IV

У розділі IV детально розглянуті основні аспекти створення та впровадження інтелектуальної системи **MSA** та використання технологій **ML**. Розділ містить інформацію про конфігурацію сервера для тестування та розгортання системи, програмне забезпечення для розробки і тестування.

Докладно описан процес розгортання системи та впровадження алгоритмів машинного навчання.

Аналіз цього розділу свідчить про те, що використання мікросервісів у поєднанні з машинним навчанням дає можливість створювати гнучкі, масштабовані та інтелектуальні системи, спроможні ефективно обробляти великі обсяги даних та вирішувати складні завдання.

Загальний висновок розділу IV підкреслює значення використання передових технологій для створення інноваційних систем, які можуть адаптуватися до змінних умов ринку. Реалізація такої інтелектуальної системи може стати ключовим фактором для підвищення продуктивності та конкурентоспроможності вебсервісів у сучасній промисловості.

Відповідно до викладених результатів, розділ IV надає комплексне розуміння впливу та можливостей використання машинного навчання у контексті масштабованих вебсервісів на основі мікросервісної архітектури. Це дозволяє глибше розуміти важливість інтеграції технологій штучного інтелекту в сучасні інформаційні системи та розширює можливості їхнього функціонування в майбутньому.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

Додаток А.
Bash - скрипт для встановлення Docker

```
# Скрипт для встановлення Docker
#!/bin/bash

# Add Docker's official GPG key:
# Додаємо офіційний GPG ключ Docker:
sudo apt-get update
sudo apt-get install ca-certificates curl gnupg
sudo install -m 0755 -d /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg |
sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
sudo chmod a+r /etc/apt/keyrings/docker.gpg

# Add the repository to Apt sources:
# Додаємо репозиторій до джерел Apt:
echo \
  "deb [arch="$(dpkg --print-architecture)" signed-
by=/etc/apt/keyrings/docker.gpg] https://down-
load.docker.com/linux/ubuntu \
  "$(. /etc/os-release && echo "$VERSION_CODENAME")" stable"
| \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update

# Install Docker:
# Встановлюємо Docker:
 sudo apt-get install docker-ce docker-ce-cli containerd.io
docker-buildx-plugin docker-compose-plugin

# Install Docker Compose:
# Встановлюємо Docker Compose:
 sudo apt-get install docker-compose-plugin

#Verify that the Docker Engine
#Перевіряємо, що Docker Engine встановлено коректно:
sudo docker run hello-world
```

Додаток Б.
dashboard_ms.py - Інтерфейс фронтенду для панелі керування вебдодатком

```
# DashBoard MS

from flask import Flask, render_template, request, redirect, jsonify
import requests
import socket

# create a Flask instance
app = Flask(__name__)
app.config['JSON_SORT_KEYS'] = False

# Product Management API endpoints
PRODUCTS_API = "http://localhost:8004/products"
PRODUCT_API = "http://localhost:8004/products/{}"

# Customer Management API endpoints
CUSTOMERS_API = "http://localhost:8003/customers"
CUSTOMER_API = "http://localhost:8003/customers/{}"

# Order Management API endpoints
ORDERS_API = "http://localhost:8005/orders"
ORDER_API = "http://localhost:8005/orders/{}"

# Payment Management API endpoints
PAYMENTS_API = "http://localhost:8009/payments"
PAYMENT_API = "http://localhost:8009/payments/{}"

# Inventory Management API endpoints
INVENTORY_API = "http://localhost:8006/inventory"
INVENTORY_ITEM_API = "http://localhost:8006/inventory/{}"


##############################
###### The root / URL Route points to the home page
@app.route('/')
def home():
    try:
        host_name = socket.gethostname()
        host_ip = socket.gethostbyname(host_name)
        return render_template('index.html', hostname=host_name, ip=host_ip)
    except:
        return render_template('error.html')


##############################
###### Product list page
@app.route('/products')
def products():
    response = requests.get(PRODUCTS_API)
    products = response.json()['products']
    return render_template('products.html', products=products)


##############################
###### Product detail page
```

Додаток Б (продовження)

```
@app.route('/products/<int:item_id>')
def product_detail(item_id):
    response = requests.get(PRODUCT_API.format(item_id))
    product = response.json()
    return render_template('product_detail.html', product=product)


##############################
###### Cart page
@app.route('/cart')
def cart():
    cart = request.cookies.get('cart')
    if cart:
        cart = [int(item) for item in cart.split(',')]
        products = []
        total_price = 0
        for item in cart:
            response = requests.get(PRODUCT_API.format(item))
            product = response.json()
            products.append(product)
            total_price += product['item_price']
        return render_template('cart.html', products=products, total_price=to-
tal_price)
    else:
        return render_template('cart.html')


##############################
###### Add product to cart
@app.route('/cart/add/<int:item_id>')
def add_to_cart(item_id):
    cart = request.cookies.get('cart')
    if cart:
        cart = cart + ',' + str(item_id)
    else:
        cart = str(item_id)
    response = redirect('/products')
    response.set_cookie('cart', cart)
    return response


##############################
###### Remove product from cart
@app.route('/cart/remove/<int:item_id>')
def remove_from_cart(item_id):
    cart = request.cookies.get('cart')
    if cart:
        cart_list = [int(item) for item in cart.split(',')]
        if id in cart_list:
            cart_list.remove(item_id)
        cart = ','.join(str(item) for item in cart_list)
        response = redirect('/cart')
        response.set_cookie('cart', cart)
        return response
    else:
        return redirect('/cart')
```

Додаток Б (продовження)

```
############################
###### Checkout page
@app.route('/checkout', methods=['GET', 'POST'])
def checkout():
    if request.method == 'POST':
        # Get form data
        customer_name = request.form['customer_name']
        customer_email = request.form['customer_email']
        customer_addr_st1 = request.form['customer_addr_st1']
        customer_addr_st2 = request.form['customer_addr_st2']
        customer_addr_city = request.form['customer_addr_city']
        customer_addr_state = request.form['customer_addr_state']
        customer_addr_zip = request.form['customer_addr_zip']
        customer_addr_country = request.form['customer_addr_country']

        # Create customer
        customer_data = {'customer_name': customer_name, 'customer_email': cus-
tomer_email,
                         'customer_addr_st1': customer_addr_st1, 'cus-
tomer_addr_st2': customer_addr_st2,
                         'customer_addr_city': customer_addr_city, 'cus-
tomer_addr_state': customer_addr_state,
                         'customer_addr_zip': customer_addr_zip, 'cus-
tomer_addr_country': customer_addr_country,
                         'customer_phone': customer_phone}
        response = requests.post(CUSTOMERS_API, json=customer_data)
        customer = response.json()

        # Create order
        cart = request.cookies.get('cart')
        cart_list = [int(item) for item in cart.split(',')]
        order_data = {'customer_id': customer['customer_id'], 'status': 'pro-
cessing', 'total_price': 0}
        response = requests.post(ORDERS_API, json=order_data)
        order = response.json()

        # Add products to order
        total_price = 0
        for item in cart_list:
            response = requests.get(PRODUCT_API.format(item))
            product = response.json()
            order_item_data = {'order_id': order['order_id'], 'item_id': prod-
uct['item_id'], 'quantity': 1,
                               'price': product['item_price']}
            response = requests.post(ORDER_API.format(order['order_id']), json=or-
der_item_data)

            # Update inventory
            response = requests.get(INVENTORY_ITEM_API.format(product['item_id']))
            inventory_item = response.json()
            inventory_item['quantity'] -= 1
            response = requests.put(INVENTORY_ITEM_API.format(product['item_id']),
json=inventory_item)

            total_price += product['item_price']
```

Додаток Б (продовження)

```
        # Update order total price
        order['total_price'] = total_price
        response = requests.put(ORDER_API.format(order['order_id']), json=order)

        # Process payment
        payment_data = {'customer_id': customer['customer_id'], 'order_id': or-
der['order_id'], 'amount': total_price}
        response = requests.post(PAYMENTS_API, json=payment_data)

        # Clear cart
        response = redirect('/')
        response.delete_cookie('cart')
        return response

    else:
        cart = request.cookies.get('cart')
        if cart:
            cart_list = [int(item) for item in cart.split(',')]
            products = []
            total_price = 0
            for item in cart_list:
                response = requests.get(PRODUCT_API.format(item))
                product = response.json()
                products.append(product)
                total_price += product['item_price']
            return render_template('checkout.html', products=products, to-
tal_price=total_price)
        else:
            return redirect('/cart')

##############################
###### Health Check Endpoint
@app.route('/healthcheck')
def health_check():
    # You can perform various health checks here
    # For simplicity, return a basic message indicating the service is healthy
    return jsonify({'status': 'ok'})


if __name__ == "__main__":
        # for debugging locally
        # app.run(debug=True, host='0.0.0.0',port=8080)

        # for production
        app.run(host='0.0.0.0', port=8080)
```

Додаток В.
customer_management_ms.py - Управління клієнтами

```
#Customer Management MS

from flask import Flask, request, jsonify
import mysql.connector

# create a Flask instance
app = Flask(__name__)
app.config['JSON_SORT_KEYS'] = False

# Connect to MySQL database
mydb = mysql.connector.connect(
  host="db",
  user="root",
  password="root",
  database="ml_msa"
)

mycursor = mydb.cursor()

service_info =   """
<!DOCTYPE html>
<head>
   <title>CUSTOMER MANAGEMENT Microservice</title>
</head>
<body>
   <h3>Customer Management Microservice Part of MSA System</h3>
</body>
                """


# The root / URL Route
# Returns an HTML with service info when service root is requested
@app.route('/', methods=['GET'])
def about():
    return service_info


# The API '/customer' URL Route with Customer Management API functions
#########################################################################

# API to add a customer
@app.route('/customers', methods=['POST'])
def add_customer():
    data = request.get_json()
    sql_stmt = "INSERT INTO customer (customer_name, customer_email, customer_phone,
customer_addr_st1, customer_addr_st2, customer_addr_city, customer_addr_state, cus-
tomer_addr_zip, customer_addr_country) VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s)"
    sql_vals = (data['customer_name'], data['customer_email'], data['cus-
tomer_phone'], data['customer_addr_st1'], data['customer_addr_st2'], data['cus-
tomer_addr_city'], data['customer_addr_state'], data['customer_addr_zip'],
data['customer_addr_country'])
    mycursor.execute(sql_stmt, sql_vals)
    mydb.commit()
    return jsonify({"success": True, 'msg': 'Customer added successfully'})
```

Додаток В (продовження)

```
# API to get all customers in the db
@app.route('/customers', methods=['GET'])
def get_customers():
    mycursor.execute("SELECT * FROM customer")
    customers = mycursor.fetchall()
    customer_list = []
    for customer in customers:
        customer_dict = {'customer_id': customer[0], 'customer_name': customer[1],
'customer_email': customer[2], 'customer_phone': customer[3], 'customer_addr_st1':
customer[4], 'customer_addr_st2': customer[5], 'customer_addr_city': customer[6],
'customer_addr_state': customer[7], 'customer_addr_zip': customer[8], 'cus-
tomer_addr_country': customer[9]}
        customer_list.append(customer_dict)
    return jsonify({"success": True, 'result': customer_list, 'msg': 'Customer list
returned successfully'})

# API to get a specific customer info using customer_id
@app.route('/customers/<int:customer_id>', methods=['GET'])
def get_customer(customer_id):
    mycursor.execute("SELECT * FROM customer WHERE customer_id = %s", (cus-
tomer_id,))
    customer = mycursor.fetchone()
    if customer:
        customer_dict = {'customer_id': customer[0], 'customer_name': customer[1],
'customer_email': customer[2], 'customer_phone': customer[3], 'customer_addr_st1':
customer[4], 'customer_addr_st2': customer[5], 'customer_addr_city': customer[6],
'customer_addr_state': customer[7], 'customer_addr_zip': customer[8], 'cus-
tomer_addr_country': customer[9]}
        return jsonify(customer_dict)
        return jsonify({"success": True, 'result': customer_dict, 'msg': 'Customer
info returned successfully'})
    else:
        return jsonify({"success": False, 'msg': 'Customer not found'})

# API to update customer info
@app.route('/customers/<int:customer_id>', methods=['PUT'])
def update_customer(customer_id):
    data = request.get_json()
    sql_stmt = "UPDATE customer SET customer_name = %s, customer_email = %s, cus-
tomer_phone = %s, customer_addr_st1 = %s, customer_addr_st2 = %s, customer_addr_city
= %s, customer_addr_state = %s, customer_addr_zip = %s, customer_addr_country = %s
WHERE customer_id = %s"
    sql_vals = (data['customer_name'], data['customer_email'], data['cus-
tomer_phone'], data['customer_addr_st1'], data['customer_addr_st2'], data['cus-
tomer_addr_city'], data['customer_addr_state'], data['customer_addr_zip'],
data['customer_addr_country'], id)
    mycursor.execute(sql_stmt, sql_vals)
    mydb.commit()
    return jsonify({"success": True, 'msg': 'Customer info updated successfully'})

# API to delete a customer
@app.route('/customers/<int:customer_id>', methods=['DELETE'])
def delete_customer(customer_id):
    mycursor.execute("DELETE FROM customer WHERE customer_id = %s", (customer_id,))
    mydb.commit()
    return jsonify({"success": True, 'msg': 'Customer successfully deleted'})
```

Додаток В (продовження)

```
############################
###### Health Check Endpoint
@app.route('/healthcheck')
def health_check():
    # You can perform various health checks here
    # For simplicity, return a basic message indicating the service is healthy
    return jsonify({'status': 'ok'})



if __name__ == "__main__":
        # for debugging locally
        # app.run(debug=True, host='0.0.0.0',port=8080)

        # for production
        app.run(host='0.0.0.0', port=8080)
```

Додаток Г.
product_management_ms.py – Управління продуктами

```
#Product Management MS

from flask import Flask, request, jsonify
import mysql.connector

# create a Flask instance
app = Flask(__name__)
app.config['JSON_SORT_KEYS'] = False

# Connect to MySQL database
mydb = mysql.connector.connect(
  host="db",
  user="root",
  password="root",
  database="ml_msa"
)
mycursor = mydb.cursor()

with app.app_context():
    service_name = "product_management"
    service_descr = "Product Management"
    service_json = jsonify({"service_name":service_name, "service_descr":service_de-
scr})

service_info =   """
<!DOCTYPE html>
<head>
   <title>PRODUCT MANAGEMENT Microservice</title>
</head>
<body>
   <h3>Product Management Microservice Part of MSA System</h3>
</body>
                """

# The root / URL Route
# Returns an HTML with service info when service root is requested
@app.route('/', methods=['GET'])
def about():
    return service_info

# API to add a new product
@app.route('/products', methods=['POST'])
def add_product():
    data = request.get_json()
    # Insert product into products table
    sql_stmt = "INSERT INTO item (item_name, item_price, item_descr, item_cata-
log_pic, item_stock_qty) VALUES (%s, %s, %s, %s, %s)"
    sql_vals = (data['item_name'], data['item_price'], data['item_descr'],
data['item_catalog_pic'], data['item_stock_qty'])
    mycursor.execute(sql_stmt, sql_vals)
    mydb.commit()
    return jsonify({"success": True, 'msg': 'Product added successfully'})

# API to get all products
@app.route('/products', methods=['GET'])
def get_products():
```

Додаток Г (продовження)

```
    mycursor.execute("SELECT * FROM item")
    products = mycursor.fetchall()
    product_list = []
    for product in products:
        product_dict = {'item_id': product[0], 'item_name': product[1],
'item_price': float(product[2]), 'item_descr': product[3], 'item_catalog_pic': prod-
uct[4], 'item_stock_qty': product[5]}
        product_list.append(product_dict)
    return jsonify({"success": True, 'result': product_list, 'msg': 'Product list
successfully returned'})

# API to get a single product using product_id
@app.route('/products/<int:item_id>', methods=['GET'])
def get_product(item_id):
    mycursor.execute("SELECT * FROM item WHERE item_id = %s", (item_id,))
    product = mycursor.fetchone()
    if product:
        product_dict = {'item_id': product[0], 'item_name': product[1],
'item_price': float(product[2]), 'item_descr': product[3], 'item_catalog_pic': prod-
uct[4], 'item_stock_qty': product[5]}
        return jsonify({"success": True, 'result': product_dict, 'msg': 'Product
info successfully returned'})
    else:
        return jsonify({"success": False, 'msg': 'Product not found'})

# API to update a product info
@app.route('/products/<int:item_id>', methods=['PUT'])
def update_product(item_id):
    data = request.get_json()
    # Update product in products table
    sql_stmt = "UPDATE item SET item_name = %s, item_price = %s, item_descr = %s,
item_catalog_pic = %s, item_stock_qty = %s WHERE item_id = %s"
    sql_vals = (data['item_name'], data['item_price'], data['item_descr'],
data['item_catalog_pic'], data['item_stock_qty'], item_id)
    mycursor.execute(sql_stmt, sql_vals)
    mydb.commit()
    return jsonify({"success": True, 'msg': 'Product updated successfully'})

# API to delete a product
@app.route('/products/<int:item_id>', methods=['DELETE'])
def delete_product(item_id):
    # Delete product from products table
    sql_stmt = "DELETE FROM item WHERE item_id = %s"
    sql_vals = (item_id,)
    mycursor.execute(sql_stmt, sql_vals)
    mydb.commit()
    return jsonify({"success": True, 'msg': 'Product deleted successfully'})

##############################
###### Health Check Endpoint
@app.route('/healthcheck')
def health_check():
    # You can perform various health checks here
    # For simplicity, return a basic message indicating the service is healthy
    return jsonify({'status': 'ok'})
```

Додаток Г (продовження)

```
if __name__ == "__main__":
        # for debugging locally
        # app.run(debug=True, host='0.0.0.0',port=8080)

        # for production
        app.run(host='0.0.0.0', port=8080)
```

Додаток Ґ.
inventory_management_ms.py - Управління інвентарем

```
#Inventory MS

from flask import Flask, request, jsonify
import os, time, random, csv
import mysql.connector

# create a Flask instance
app = Flask(__name__)
app.config['JSON_SORT_KEYS'] = False

# Connect to MySQL database
mydb = mysql.connector.connect(
  host="db",
  user="root",
  password="root",
  database="ml_msa"
)

mycursor = mydb.cursor()

with app.app_context():
    service_name = "inventory_management"
    service_descr = "Inventory Management"
    service_json = jsonify({"service_name":service_name, "service_descr":service_de-
scr})

delay_cfg_filename = "ms_delays.cfg"

service_info =   """
<!DOCTYPE html>
<head>
   <title>INVENTORY MANAGEMENT Microservice</title>
</head>
<body>
   <h3>Inventory Management Microservice Part of MSA System</h3>
</body>
                """

# curl http://localhost:8080/api?func=about
api_funcs = [
    "about",
    "service_info",
    "check_inventory",
    "update_inventory",
    "delay_response"
]

# The root / URL Route
# Returns an HTML with service info when service root is requested
@app.route('/', methods=['GET'])

def simulate_delay():
    global delay_cfg_filename

    delay_min_sec = 0
    delay_max_sec = 0
```

Додаток Ґ (продовження)

```
    file_exists = os.path.isfile(delay_cfg_filename)
    if file_exists:
        #print("Found the delay cfg file, assigning delays...")
        with open(delay_cfg_filename, 'r') as f:
            delay_cfg = csv.reader(f)
            # The file should have only one row "delay_min_sec,delay_max_sec"
            for row in delay_cfg:
                delay_min_sec = float(row[0])
                delay_max_sec = float(row[1])
        #print("Delay min & max values = %s and %s" %(delay_min_sec, delay_max_sec))
    else:
        #print("Did NOT find the file delay cfg file, initiating...")
        f = open(delay_cfg_filename, "w")
        f.write("0,0")

    f.close()

    #Simulate a delay if received an API to do so
    if delay_max_sec > 0:
        delay_seconds = round(delay_min_sec + random.random()*(delay_max_sec-de-
lay_min_sec), 3)
        #print("Adding a delay %s ..." %delay_seconds)
        time.sleep(delay_seconds)

    return ""


def about():
    simulate_delay()
    return service_info

# The API '/api' URL Route with API functions
##############################################
# Returns an error message if wrong arguments are passed.
@app.route('/api', methods=['GET'])
def abc_msa():
    global delay_cfg_filename

    simulate_delay()

    # A def to return a json with an error message if API usage error is detected
    def usage_error(msg_details):
        error_message =  f"{request.args[0]} usage error"
        return jsonify({'success':False, 'msg': f"{error_message}: {msg_details}"})

    # Copy all API args into a new dict
    request_args = request.args

    def service_info():
        # returns a JSON with service information
        return service_json

    # Check if a valid API function has been called
    func_call = request_args['func']
    if not ( func_call in (api_funcs) ):
        #usage_error("Invalid API function called")
        return jsonify({'success': False, 'msg': f"{request.args[0]}: Invalid API
```

Додаток Ґ (продовження)

```
function called"})


##############################################
##### SIMULATE A RESPONSE DELAY API CALL #####
##############################################
    if func_call == "delay_response":
        delay_min_sec = round( int(request.args['delay_min']) / 1000, 3)
        delay_max_sec = round( int(request.args['delay_max']) / 1000, 3)

        f = open(delay_cfg_filename, "w")
        f.write("%s,%s" %(delay_min_sec, delay_max_sec) )
        f.close()

        return jsonify({'msg':"delay response assigned successfully"})


############################
## service_info API CALL ##
############################
    if func_call == "service_info":
        return service_info()


############################
##### ABOUT API CALL  #####
############################
    if func_call == "about":
        return jsonify({"success": True})


############################
##### check_inventory ####
############################
    if func_call == "check_inventory":
        item_id = request.args['item_id']
        mycursor.execute("SELECT * FROM item WHERE item_id = %s", (item_id,))
        product = mycursor.fetchone()
        if product:
            product_dict = {'item_id': product[0], 'item_name': product[1],
'item_price': float(product[2]), 'item_descr': product[3], 'item_catalog_pic': prod-
uct[4], 'item_stock_qty': product[5]}
            return jsonify({"success": True, 'result': product_dict, 'msg': 'Product
info successfully returned'})
        else:
            return jsonify({"success": False, 'msg': 'Product not found'})



############################
##### update_inventory ####
############################
    if func_call == "update_inventory":
        data = request.get_json()
        # Update product in products table
        sql_stmt = "UPDATE item SET item_name = %s, item_price = %s, item_descr =
%s, item_catalog_pic = %s, item_stock_qty = %s WHERE item_id = %s"
```

Додаток Ґ (продовження)

```
        sql_vals = (data['item_name'], data['item_price'], data['item_descr'],
data['item_catalog_pic'], data['item_stock_qty'], item_id)
        mycursor.execute(sql_stmt, sql_vals)
        mydb.commit()
        return jsonify({"success": True, 'msg': 'Product updated successfully'})


##############################
###### Health Check Endpoint
@app.route('/healthcheck')
def health_check():
    # You can perform various health checks here
    # For simplicity, return a basic message indicating the service is healthy
    return jsonify({'status': 'ok'})



if __name__ == "__main__":
        # for debugging locally
        # app.run(debug=True, host='0.0.0.0',port=8080)

        # for production
        app.run(host='0.0.0.0', port=8080)
```

Додаток Д.
payment_management_ms.py - Управління платежами

```
#Payment Management MS

from flask import Flask, request, jsonify
import mysql.connector

# create a Flask instance
app = Flask(__name__)
app.config['JSON_SORT_KEYS'] = False

# Connect to MySQL database
mydb = mysql.connector.connect(
  host="db",
  user="root",
  password="root",
  database="ml_msa"
)

mycursor = mydb.cursor()

with app.app_context():
    service_name = "payment_management"
    service_descr = "Payment Management"
    service_json = jsonify({"service_name":service_name, "service_descr":service_de-
scr})

service_info =   """
<!DOCTYPE html>
<head>
   <title>PAYMENT MANAGEMENT Microservice</title>
</head>
<body>
   <h3>Payment Management Microservice Part of MSA System</h3>
</body>
                """

# The root / URL Route
# Returns an HTML with service info when service root is requested
@app.route('/', methods=['GET'])
def about():
    return service_info

# The API '/payments' URL Route with Payment Management API functions
######################################################################
# API endpoint to add a new payment for an order
@app.route('/payments', methods=['POST'])
def add_payment():
    data = request.get_json()
    order_id = data['order_id']
    amount = data['amount']
    payment_date = data.get('payment_date')
    sql_stmt = "INSERT INTO payments (order_id, payment_date, amount) VALUES (%s,
%s, %s)"
    sql_vals = (order_id, payment_date, amount)
    mycursor.execute(sql_stmt, sql_vals)
    mydb.commit()
    sql_stmt = "UPDATE orders SET total_price = total_price + %s WHERE id = %s"
```

Додаток Д (продовження)

```
    sql_vals = (amount, order_id)
    mycursor.execute(sql_stmt, sql_vals)
    mydb.commit()
    return jsonify({'msg': 'Payment added successfully'})

# API to get all processed payments
@app.route('/payments', methods=['GET'])
def get_payments():
    mycursor.execute("SELECT * FROM payment")
    payments = mycursor.fetchall()
    payment_list = []
    for payment in payments:
        payment_dict = {'payment_id': payment[0], 'order_id': payment[1], 'pay-
ment_date': payment[2], 'amount': float(payment[3])}
        payment_list.append(payment_dict)
    return jsonify({"success": True, 'result': payment_list, 'msg': 'Payment list
successfully returned'})

# API to get a single processed payment
@app.route('/payments/<int:payment_id>', methods=['GET'])
def get_payment(payment_id):
    mycursor.execute("SELECT * FROM payment WHERE payment_id = %s", (payment_id,))
    payment = mycursor.fetchone()
    if payment:
        payment_dict = {'payment_id': payment[0], 'order_id': payment[1], 'pay-
ment_date': payment[2], 'amount': float(payment[3])}
        return jsonify({"success": True, 'result':payment_dict, 'msg': 'Payment info
successfully returned'})
    else:
        return jsonify({"success": False, 'msg': 'Payment not found'})

# API to update the payment
@app.route('/payments/<int:payment_id>', methods=['PUT'])
def update_payment(payment_id):
    data = request.get_json()
    # Update payment in payments table
    sql_stmt = "UPDATE payment SET payment_date = %s, amount = %s WHERE payment_id =
%s"
    sql_vals = (data['payment_date'], data['amount'], payment_id)
    mycursor.execute(sql_stmt, sql_vals)
    mydb.commit()
    return jsonify({"success": True, 'msg': 'Payment updated'})

##############################
###### Health Check Endpoint
@app.route('/healthcheck')
def health_check():
    # You can perform various health checks here
    # For simplicity, return a basic message indicating the service is healthy
    return jsonify({'status': 'ok'})


if __name__ == "__main__":
        # for debugging locally
        # app.run(debug=True, host='0.0.0.0',port=8080)
```

Додаток Д (продовження)

```
# for production
app.run(host='0.0.0.0', port=8080)
```

Додаток Е.
Dockerfile для dashboard_ms.py

```
# Use the Python based based on the popular Alpine Linux project
#Alpine Linux is much smaller than most distribution base images (~5MB), and thus leads to much slimmer images in general.
FROM python:3-alpine

# Expose port 8080 (this doesn't actually publish the port, it's just for documentation)
EXPOSE 8080

# Set the working directory inside the container to /app
WORKDIR /app

# Copy the requirements.txt file from the local machine to the /app directory in the container
COPY requirements.txt /app

# Install the Python dependencies specified in requirements.txt
RUN pip install -r requirements.txt

# Copy the customer_management_ms.py file from the local machine to the /app directory in the container
COPY dashboard_ms.py /app
COPY ./templates /app/templates

# Set the default command to execute the Python script when the container starts
CMD python dashboard_ms.py
```

Додаток Є.
Dockerfile для customer_management_ms.py

```
# Use the Python based  based on the popular Alpine Linux project
#Alpine Linux is much smaller than most distribution base images (~5MB), and thus leads to much slimmer images in general.
FROM python:3-alpine

# Expose port 8080 (this doesn't actually publish the port, it's just for documentation)
EXPOSE 8080

# Set the working directory inside the container to /app
WORKDIR /app

# Copy the requirements.txt file from the local machine to the /app directory in the container
COPY requirements.txt /app

# Install the Python dependencies specified in requirements.txt
RUN pip install -r requirements.txt

# Copy the customer_management_ms.py file from the local machine to the /app directory in the container
COPY customer_management_ms.py /app

# Set the default command to execute the Python script when the container starts
CMD python customer_management_ms.py
```

Додаток Ж.
Dockerfile для product_management_ms.py

```
# Use the Python based  based on the popular Alpine Linux project
#Alpine Linux is much smaller than most distribution base images (~5MB), and thus leads to much slimmer images in general.
FROM python:3-alpine

# Expose port 8080 (this doesn't actually publish the port, it's just for documentation)
EXPOSE 8080

# Set the working directory inside the container to /app
WORKDIR /app

# Copy the requirements.txt file from the local machine to the /app directory in the container
COPY requirements.txt /app

# Install the Python dependencies specified in requirements.txt
RUN pip install -r requirements.txt

# Copy the customer_management_ms.py file from the local machine to the /app directory in the container
COPY product_management_ms.py /app

# Set the default command to execute the Python script when the container starts
CMD python product_management_ms.py
```

Додаток 3.
Dockerfile для inventory_management_ms.py

```
# Use the Python based  based on the popular Alpine Linux project
#Alpine Linux is much smaller than most distribution base images (~5MB), and thus leads to much slimmer images in general.
FROM python:3-alpine

# Expose port 8080 (this doesn't actually publish the port, it's just for documentation)
EXPOSE 8080

# Set the working directory inside the container to /app
WORKDIR /app

# Copy the requirements.txt file from the local machine to the /app directory in the container
COPY requirements.txt /app

# Install the Python dependencies specified in requirements.txt
RUN pip install -r requirements.txt

# Copy the customer_management_ms.py file from the local machine to the /app directory in the container
COPY inventory_management_ms.py /app

# Set the default command to execute the Python script when the container starts
CMD python inventory_management_ms.py
```

Додаток И.
Dockerfile для payment_management_ms.py

```
# Use the Python based  based on the popular Alpine Linux project
#Alpine Linux is much smaller than most distribution base images (~5MB), and thus leads to much slimmer images in general.
FROM python:3-alpine

# Expose port 8080 (this doesn't actually publish the port, it's just for documentation)
EXPOSE 8080

# Set the working directory inside the container to /app
WORKDIR /app

# Copy the requirements.txt file from the local machine to the /app directory in the container
COPY requirements.txt /app

# Install the Python dependencies specified in requirements.txt
RUN pip install -r requirements.txt

# Copy the customer_management_ms.py file from the local machine to the /app directory in the container
COPY payment_management_ms.py /app

# Set the default command to execute the Python script when the container starts
CMD python payment_management_ms.py
```

Додаток І.
docker-compose.yaml

```
version : "2"

services:
  db:
    image: mysql:5.7
    command: --default-authentication-plugin=mysql_native_password
    environment:
      MYSQL_ROOT_PASSWORD: root
    ports:
      - "3307:3306"
    healthcheck:
      test: [ "CMD", "mysqladmin", "--password=root", "-h", "localhost", "ping" ]
    volumes:
      - "./db/schema.sql:/docker-entrypoint-initdb.d/1.sql"
      - "./db/data.sql:/docker-entrypoint-initdb.d/2.sql"

  dashboard:
    build: ./dashboard
    links:
      - db
    ports:
       - "8002:8080"
    depends_on:
      db:
        condition: service_healthy
    healthcheck:
      test: [ "CMD-SHELL", "wget --quiet --spider http://localhost:8080/healthcheck
|| exit 1" ]
      interval: 30s
      retries: 3
      timeout: 10s

  customer_management:
    build: ./customer_management
    links:
      - db
    ports:
      - "8003:8080"
    depends_on:
      db:
        condition: service_healthy
    healthcheck:
      test: [ "CMD-SHELL", "wget --quiet --spider http://localhost:8080/healthcheck
|| exit 1" ]
      interval: 30s
      retries: 3
      timeout: 10s

  inventory_management:
    build: ./inventory
    links:
      - db
    ports:
      - "8006:8080"
    depends_on:
      db:
```

Додаток І (продовження)

```
        condition: service_healthy
    healthcheck:
      test: [ "CMD-SHELL", "wget --quiet --spider http://localhost:8080/healthcheck
|| exit 1" ]
      interval: 30s
      retries: 3
      timeout: 10s

  order_management:
    build: ./order
    links:
      - db
    ports:
      - "8005:8080"
    depends_on:
      db:
        condition: service_healthy
    healthcheck:
      test: [ "CMD-SHELL", "wget --quiet --spider http://localhost:8080/healthcheck
|| exit 1" ]
      interval: 30s
      retries: 3
      timeout: 10s

  payment_management:
    build: ./payment
    links:
      - db
    ports:
      - "8007:8080"
    depends_on:
      db:
        condition: service_healthy
    healthcheck:
      test: [ "CMD-SHELL", "wget --quiet --spider http://localhost:8080/healthcheck
|| exit 1" ]
      interval: 30s
      retries: 3
      timeout: 10s

  product_management:
    build: ./product_management
    links:
      - db
    ports:
      - "8004:8080"
    depends_on:
      db:
        condition: service_healthy
    healthcheck:
      test: [ "CMD-SHELL", "wget --quiet --spider http://localhost:8080/healthcheck
|| exit 1" ]
      interval: 30s
      retries: 3
      timeout: 10s
```

Додаток Ї.
simulate_api_rqsts.py

```
import sys, os, asyncio, aiohttp, datetime, time

build_no = "1"
usage_help = """
        Usage:
            simulate_api_rqsts --version | --help | <rate> <url list file> [-r] [-v]

        Where:
            --version: Shows the current release/build version

            --help: shows this message

            rate: API call simulate requests rate measured in requests per minute

            url list file: Is the text file with a list of all target URLs where the
API calls will be sent to
                 Have each target URL in a separate line

            -r: Specify a randomly paced rate. If omitted, a uniformly paced rate
will be used.

                The uniformly paced requests are paced out so that the time between
each API call is always the same,
                so if we are configuring a uniformly paced load of 600 API re-
quests/min, simulate_api_rqsts
                will send 1 API call every T = 100 ms.

                In the randomly-paced case, each API call is sent after a random pe-
riod, TR, from the time where
                the previous call was sent, but so that TR can never be larger or
smaller than 95% of T. So if we are
                configuring a randomly-paced load of 600 API requests/min, TR, in
that case, will be equal to a value
                greater than 5 ms and smaller than 195 ms.
                simulate_api_rqsts will send 1 API call every:
                (1-95%)T <= TR <= (1+95%)T (i.e., for 600 requests/min: 5 ms <= TR
<= 195 ms)
                The sum of all TRs, however, will still be approximately equal to
the configured requests/min. In our
                example here, the load is 600 API requests/min.
                Uniformly paced requests are better when you are manually analyzing
how a particular microservice
                responds to the API load, while randomly paced requests are a better
representation of a real-time
                production API request load.

            -v: Verbose mode
"""

#Initialize Configuration Params
REQ_TIMEOUT = 2
variable_reqs = '-r' in sys.argv
verbose = '-v' in sys.argv
reqs_rate = 0

async def get(url, session):
```

Додаток Ї (продовження)

```
    try:
        async with session.get(url=url, timeout=REQ_TIMEOUT) as response:
            resp = await response.read()

    except Exception as e:
        pass

async def main(urls):
    async with aiohttp.ClientSession() as session:
        ret = await asyncio.gather(*[get(url, session) for url in urls])



# Function to get a list of random delays for every minute
def get_1min_delays(reqs_rate):
    if not reqs_rate:
        return []

    t = 60 / reqs_rate
    reqs_rate_min = t * 0.05
    reqs_rate_max = t * 1.95

    # calculate the random Tr list size based on worse case scenario of, all Tr List
element are equal to reqs_rate_min. There will be a break anyway to avoid too many
items in the list
    tr_list = []
    tr_list_size = int(round(60 /reqs_rate_min ,0))
    total_Tr = 0

    for i in range(1, tr_list_size):
        Tr = round(random.uniform(reqs_rate_min, reqs_rate_max), 2)
        tr_list.append(Tr)
        total_Tr = total_Tr + Tr
        if total_Tr >= 60:
            break;

    return tr_list



show_version_output = "\nsimulate_api_rqsts ver. %s\n" % build_no

if len(sys.argv) > 1:
    if '--help' in sys.argv:
        print(show_version_output)
        print(usage_help)
        sys.exit(0)

    if '--version' in sys.argv:
        print(show_version_output)
        sys.exit(0)

    reqs_rate = sys.argv[1]
    if reqs_rate.isnumeric():
        reqs_rate = int(reqs_rate)
        if reqs_rate<=0:
            print("ERROR: API Calls Rate needs to be a number > 0")
```

Додаток Ї (продовження)

```
            sys.exit(-1)

        url_list = []
        if len(sys.argv) > 2:
            url_list_file = open(sys.argv[2], "r")
            for url in url_list_file:
                url_list.append(url)

    else:
        print("ERROR: API Calls Rate needs to be an integer")
        sys.exit(-1)


else:
    print("ERROR: Missing parameters")
    print(show_version_output)
    print(usage_help)
    sys.exit(-1)


# Assume the default of a uniformly paced requests
# Calculate T, the frequency of the API calls
sleep_time = 60 / reqs_rate

tr_list_len = 0
if variable_reqs:
    tr_list = get_1min_delays(reqs_rate)
    tr_list_len = len(tr_list)


if verbose:
    print("Simulating API calls to the URL List:")
    print(url_list)

n = 1
i = 0
while True:
    if verbose:
        print("HTTP Req... #"+ str(n))
        n = n+1

    asyncio.run(main(url_list))

    if variable_reqs and tr_list_len>0:
        time.sleep(tr_list[i])
        i = i+1
        # If all random generated delays are processed, regenerate a new 1-min list
of delays
        if i>=tr_list_len:
            i=0
            tr_list = get_1min_delays(reqs_rate)
    else:
        time.sleep(sleep_time)
```

Додаток Й.
ms_perfmon.py

```
import sys, os, asyncio, aiohttp, datetime, time

build_no = "1"
usage_help = """
        Usage:
            ms_perfmon.py --version | --help | <pull interval> <url list file> [-v]

        Where:
            --version: Shows the current release/build version

            --help: shows this message

            pull interval: Time in seconds between each service's stats pull request

            url list file: Is the text file with a list of all target URLs where the
API calls will be sent to
                 Have each target URL in a separate line

            -v: Verbose mode
"""

show_version_output = "\nms_perfmon ver. %s\n" % build_no

if len(sys.argv) <= 1:
    print("ERROR: Missing parameters")
    print(show_version_output)
    print(usage_help)
    sys.exit(-1)


#Initialize Params
REQ_TIMEOUT = 3
verbose = '-v' in sys.argv

if '--help' in sys.argv:
    print(show_version_output)
    print(usage_help)
    sys.exit(0)

if '--version' in sys.argv:
    print(show_version_output)
    sys.exit(0)

pull_interval = sys.argv[1]
if pull_interval.isnumeric():
    pull_interval = int(pull_interval)
    if pull_interval<=0:
        print("ERROR: Stats pull interval needs to be a number > 0s")
        sys.exit(-1)

    url_list = []
    if len(sys.argv) > 2:
        url_list_file = open(sys.argv[2], "r")
        for url in url_list_file:
            url_list.append(url)
```

Додаток Й (продовження)

```
else:
    print("ERROR: API Calls Rate needs to be an integer")
    sys.exit(-1)


def create_url_filename(url):
    filename = url.replace("http://", "")
    filename = url.replace("https://", "")
    filename = filename.replace(":", "_")
    filename = filename.replace("/", "_")
    filename = filename.replace("\n", "")
    return filename

async def get(url, session):
    try:
        request_datetime = str(datetime.datetime.now())
        start_time = time.time()
        async with session.get(url=url, timeout=REQ_TIMEOUT) as response:
            resp = await response.read()
            end_time = time.time()
            response_status_code = response.status
            response_total_seconds = str(end_time-start_time)

            filename = "perfmon_stats/" + create_url_filename(url) + ".csv"
            file_exists = os.path.isfile(filename)
            if file_exists:
                f = open(filename, "a")
            else:
                f = open(filename, "w")
                f.write("url,request_datetime,response_time,status_code\n")

            f.write("%s,%s,%s,%s\n" %(url, request_datetime, response_total_seconds,
response_status_code) )
            f.close()

    except Exception as e:
        filename = "perfmon_stats/" + create_url_filename(url) + ".csv"
        file_exists = os.path.isfile(filename)
        if file_exists:
            f = open(filename, "a")
        else:
            f = open(filename, "w")
            f.write("url,request_datetime,response_time\n")

        f.write("%s,%s,%s,%s\n" %(url, request_datetime, 0, "ClientConnectorError")
)
        f.close()


async def main(urls):
    async with aiohttp.ClientSession() as session:
        ret = await asyncio.gather(*[get(url, session) for url in urls])

if verbose:
    print("Pulling stats for the following URL List:")
    print(url_list)
```

Додаток Й (продовження)

```
n = 1
while True:
    if verbose:
        print("Stats pull request... #"+ str(n))
        n = n+1

    asyncio.run(main(url_list))
    time.sleep(pull_interval)
```

Додаток К.
PBW_learn.py

```
import numpy as np
import pandas as pd
from sklearn.linear_model import LinearRegression

# Get a list of all files in the scrubbed_stats directory
files = os.listdir('scrubbed_stats')

# Initialize an empty list to store the trained models
trained_models = []

# Process each file
for file in files:
    # Read the CSV file into a pandas DataFrame
    df = pd.read_csv('scrubbed_stats/' + file)

    # Set the target column (response time)
    target_column = "response_time"

    # Create a Linear Regression model
    model = LinearRegression()

    # Extract features (remaining columns)
    features = df.drop(columns=[target_column])

    # Split the data into training and test sets
    X_train, X_test, y_train, y_test = train_test_split(features, df[target_column],
test_size=0.2)

    # Train the model
    model.fit(X_train, y_train)

    # Evaluate the model on the test set
    model_score = model.score(X_test, y_test)
    print(f"Model score for {file}: {model_score}")

    # Save the model
    model_file = file.replace(".csv", ".pkl")
    with open(f"models/{model_file}", "wb") as f:
        pickle.dump(model, f)

    # Append the trained model to the list
    trained_models.append(model)
```

Додаток Л.
PAD_learn.py

```
import numpy as np
import pandas as pd
from sklearn.svm import OneClassSVM

# Get a list of all files in the scrubbed_stats directory
files = os.listdir('scrubbed_stats')

# Initialize an empty list to store the trained models
anomaly_models = []

# Process each file
for file in files:
    # Read the CSV file into a pandas DataFrame
    df = pd.read_csv('scrubbed_stats/' + file)

    # Set the target column (response time)
    target_column = "response_time"

    # Create a OneClassSVM model
    model = OneClassSVM(gamma='scale', nu=0.1)

    # Extract features (response time)
    features = np.array(df[target_column]).reshape(-1, 1)

    # Train the model
    model.fit(features)

    # Evaluate the model on the data
    y_pred = model.predict(features)

    # Identify anomalies
    anomaly_indices = np.where(y_pred == -1)[0]
    anomalies = df.iloc[anomaly_indices]

    # Append the trained model and anomalies to the lists
    anomaly_models.append(model)

    df = df.assign(Anomaly=y_pred == -1)
    df.to_csv(f"predicted_{file}_rt_anomalies.csv", mode='w', index=False)
```

Додаток М.
training_data_cleanup.py

```
import os
import pandas as pd

def clean_data():
    # Get a list of all CSV files in the perfmon_stats directory
    csv_files = os.listdir('perfmon_stats')

    # Create a directory for the scrubbed data
    if not os.path.exists('scrubbed_stats'):
        os.mkdir('scrubbed_stats')

    # Clean each CSV file
    for csv_file in csv_files:
        # Get the microservice name from the filename
        microservice_name = csv_file.split('.')[0]

        # Read the CSV file into a pandas DataFrame
        df = pd.read_csv('perfmon_stats/' + csv_file)

        # Drop any rows with missing values
        df.dropna(inplace=True)

        # Convert the response_time column to numeric format
        df['response_time'] = pd.to_numeric(df['response_time'], errors='coerce')

        # Drop any rows with invalid response_time values
        df.dropna(subset=['response_time'], inplace=True)

        # Save the cleaned DataFrame to a new CSV file
        df.to_csv('scrubbed_stats/' + microservice_name + '.csv', index=False)

if __name__ == '__main__':
    clean_data()
```

Додаток Н.
msa-ai.yaml

```yaml
version : "2"

services:
  db:
    image: mysql:5.7
    command: --default-authentication-plugin=mysql_native_password
    environment:
      MYSQL_ROOT_PASSWORD: root
    ports:
      - "3307:3306"
    healthcheck:
      test: [ "CMD", "mysqladmin", "--password=root", "-h", "localhost", "ping" ]
    volumes:
      - "./db/schema.sql:/docker-entrypoint-initdb.d/1.sql"
      - "./db/data.sql:/docker-entrypoint-initdb.d/2.sql"

  dashboard:
    build: ./dashboard
    links:
      - db
    ports:
      - "8002:8080"
    depends_on:
      db:
        condition: service_healthy
    healthcheck:
      test: [ "CMD-SHELL", "wget --quiet --spider http://localhost:8080/healthcheck || exit 1" ]
      interval: 30s
      retries: 3
      timeout: 10s

  customer_management:
    build: ./customer_management
    links:
      - db
    ports:
      - "8003:8080"
    depends_on:
      db:
        condition: service_healthy
    healthcheck:
      test: [ "CMD-SHELL", "wget --quiet --spider http://localhost:8080/healthcheck || exit 1" ]
      interval: 30s
      retries: 3
      timeout: 10s

  inventory_management:
    build: ./inventory
    links:
      - db
    ports:
      - "8006:8080"
```

Додаток Н (продовження)

```
  depends_on:
    db:
      condition: service_healthy
  healthcheck:
    test: [ "CMD-SHELL", "wget --quiet --spider http://localhost:8080/healthcheck || exit 1" ]
    interval: 30s
    retries: 3
    timeout: 10s

order_management:
  build: ./order
  links:
    - db
  ports:
    - "8005:8080"
  depends_on:
    db:
      condition: service_healthy
  healthcheck:
    test: [ "CMD-SHELL", "wget --quiet --spider http://localhost:8080/healthcheck || exit 1" ]
    interval: 30s
    retries: 3
    timeout: 10s

payment_management:
  build: ./payment
  links:
    - db
  ports:
    - "8007:8080"
  depends_on:
    db:
      condition: service_healthy
  healthcheck:
    test: [ "CMD-SHELL", "wget --quiet --spider http://localhost:8080/healthcheck || exit 1" ]
    interval: 30s
    retries: 3
    timeout: 10s

product_management:
  build: ./product_management
  links:
    - db
  ports:
    - "8004:8080"
  depends_on:
    db:
      condition: service_healthy
  healthcheck:
    test: [ "CMD-SHELL", "wget --quiet --spider http://localhost:8080/healthcheck || exit 1" ]
    interval: 30s
    retries: 3
    timeout: 10s
```

Додаток Н (продовження)

```
pwd:
  build: ./pwd
  depends_on:
    dashboard:
      condition: service_healthy
    product_management:
      condition: service_healthy
    inventory_management:
      condition: service_healthy
    customer_management:
      condition: service_healthy
    order_management:
      condition: service_healthy
    payment_management:
      condition: service_healthy

pad:
  build: ./pad
  depends_on:
    dashboard:
      condition: service_healthy
    product_management:
      condition: service_healthy
    inventory_management:
      condition: service_healthy
    customer_management:
      condition: service_healthy
    order_management:
      condition: service_healthy
    payment_management:
      condition: service_healthy
```