

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ  
ДЕРЖАВНИЙ ЗАКЛАД  
«ЛУГАНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ  
ІМЕНІ ТАРАСА ШЕВЧЕНКА»

Навчально-науковий інститут математики  
та інформаційних технологій


Кафедра інформаційних технологій та систем

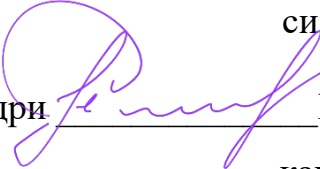
**Лещенко Олексій Володимирович**

**СУЧАСНІ МЕТОДОЛОГІЇ ПРОЄКТУВАННЯ  
ВИСОКОНАВАНТАЖЕНИХ СИСТЕМ ІЗ ВИКОРИСТАННЯМ  
МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ В ХМАРНИХ СЕРЕДОВИЩАХ**

**кваліфікаційна робота  
здобувача вищої освіти другого (магістерського) рівня  
освітньої програми «Мультимедійні системи»  
за спеціальністю 121 „Інженерія програмного забезпечення”**

Особистий підпис  Олексій ЛЕЩЕНКО

Науковий керівник  Світлана ПЕРЕЯСЛАВСЬКА,  
кандидат педагогічних наук, доцент  
кафедри інформаційних технологій та  
систем

Завідувач кафедри  Микола СЕМЕНОВ ,  
кандидат педагогічних наук, доцент  
кафедри інформаційних технологій  
та систем

## АНОТАЦІЯ

**Лещенко О.В.**

**Тема:** Сучасні методології проектування високонавантажених систем із використанням мікросервісної архітектури в хмарних середовищах.

**Спеціальність:** 121 «Інженерія програмного забезпечення».

**Установа:** ДЗ ЛНУ імені Тараса Шевченка, 2026 р.

**Магістерська робота містить:** 62 сторінки, 21 рисуноків, 1 таблицю, 7 додатків, 68 джерел.

**Об'єкт дослідження** – високонавантажені інформаційні системи.

**Предмет дослідження** – архітектурні підходи та інструменти проектування високонавантажених систем з використанням мікросервісної архітектури.

**Мета роботи** – аналіз та розробка ефективних підходів до проектування високонавантажених інформаційних систем із застосуванням мікросервісної архітектури на базі хмарних рішень від Amazon AWS..

**Методи дослідження** – аналіз і синтез архітектурних підходів, порівняльний аналіз монолітної, клієнт–серверної, мікросервісної, serverless та подійно-орієнтованої архітектур, моделювання, проектування програмних систем, експериментальне навантажувальне тестування та тестування відмовостійкості.

**Результати роботи.** Проаналізовано сучасні методології проектування високонавантажених систем та обґрунтовано доцільність використання мікросервісної архітектури в хмарних середовищах. Реалізовано мікросервісну платформу в Amazon Web Services. Результати навантажувального тестування підтвердили ефективність горизонтального масштабування, а експерименти з відмовостійкості засвідчили здатність системи зберігати працездатність у разі відмови окремих сервісів.

**Ключові слова:** високонавантажені системи, мікросервісна архітектура, хмарні обчислення, Amazon Web Services, масштабованість, відмовостійкість.

## ABSTRACT

**Leshchenko O.V.**

**Topic:** Modern methodologies for designing high-load systems using microservice architecture in cloud environments.

**Specialty:** 121 “Software Engineering”.

**Institution:** Taras Shevchenko Luhansk National University, 2026.

**The master’s thesis contains:** 63 pages, 21 figures, 1 table, 7 appendices, 68 references.

**Object of the research:** High-load information systems operating in cloud environments.

**Subject of the research:** Architectural approaches and design tools for building scalable and fault-tolerant systems using microservice architecture within the Amazon Web Services infrastructure.

**Purpose of the work:** analysis and development of effective approaches to designing high-load information systems using microservice architecture based on cloud solutions from Amazon AWS.

**Research methods.** Analysis and synthesis of architectural approaches, comparative analysis of monolithic, client–server, microservice, serverless, and event-driven architectures, modeling, software system design, experimental load testing, and fault tolerance testing.

**Research results.** Modern methodologies for designing high-load systems were analyzed, and the feasibility of using microservice architecture in cloud environments was substantiated. A microservice platform was implemented in Amazon Web Services. The results of load testing confirmed the effectiveness of horizontal scaling, while fault tolerance experiments demonstrated the system’s ability to maintain operability in the event of individual service failures.

**Keywords:** high-load systems, microservice architecture, cloud computing, Amazon Web Services, scalability, fault tolerance.

## ЗМІСТ

ВСТУП.....	8
РОЗДІЛ 1. АНАЛІЗ СУЧАСНИХ МЕТОДОЛОГІЙ ПРОЄКТУВАННЯ ВИСОКОНАВАНТАЖЕНИХ СИСТЕМ .....	10
1.1 Монолітна архітектура (Monolithic Architecture).....	10
1.2 Клієнт-серверна архітектура (Client-Server Architecture) .....	12
1.3 Мікросервісна архітектура (Microservices Architecture) .....	13
1.4 Сервіс-орієнтована архітектура (SOA — Service-Oriented Architecture) .....	15
1.5. Serverless / Function-as-a-Service Architecture (FaaS).....	17
1.6 Подійно-орієнтована архітектура (EDA — Event-Driven Architecture) .....	19
1.7.1. Питання надійності при проектуванні високонавантажених систем .....	24
1.7.2. Питання масштабованості при проектуванні високонавантажених систем .....	25
1.7.3. Питання супроводжуваності при проектуванні високонавантажених систем .....	26
Висновки до розділу 1 .....	29
РОЗДІЛ 2. ПРАКТИЧНА ЧАСТИНА РЕАЛІЗАЦІЇ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ В ХМАРНИХ СЕРЕДОВИЩАХ.....	31
2.1. Загальні відомості та архітектура практичної частини.....	31
2.2. Розгортання інфраструктури.....	33
2.3 Функціонування мікросервісів платформи .....	42
2.3.1 Мікросервіс User_service.....	43
2.3.2 Мікросервіс Payment_service .....	44

2.3.3 Мікросервіс Billing_service .....	44
2.3.4 Мікросервіс Product_service .....	45
2.3.5 Мікросервіс Api_gateway .....	46
2.3.6. Мікросервіс Web_ui .....	47
2.4 Навантажувальне тестування платформи.....	48
2.5. Тестування відмовостійкості платформи .....	51
Висновки до розділу 2 .....	54
ВИСНОВКИ .....	55
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ .....	57
ДОДАТКИ.....	62
Додаток А. Вихідний код додатку.....	63
Додаток Б. Вихідний код додатку .....	73
Додаток В. Вихідний код додатку.....	74
Додаток Г. Вихідний код додатку .....	79
Додаток Д. Вихідний код додатку.....	80
Додаток Е. Вихідний код додатку .....	83
Додаток Є. Вихідний код додатку.....	86

## ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ

**AWS** – Amazon Web Services — хмарна платформа компанії Amazon.

**API** – Application Programming Interface — програмний інтерфейс прикладного програмування.

**CI/CD** – Continuous Integration / Continuous Delivery — безперервна інтеграція та доставка.

**DevOps** – Підхід до розробки, що поєднує розробку та експлуатацію.

**DNS** – Domain Name System — система доменних імен.

**ECR** – Elastic Container Registry — реєстр контейнерних образів AWS.

**ECS** – Elastic Container Service — сервіс оркестрації контейнерів AWS.

**EDA** – Event-Driven Architecture — подійно-орієнтована архітектура.

**ESB** – Enterprise Service Bus — корпоративна сервісна шина.

**FaaS** – Function as a Service — модель виконання функцій у хмарі.

**gRPC** – Google Remote Procedure Call — високопродуктивний RPC-протокол.

**HTTP** – HyperText Transfer Protocol — протокол передачі гіпертексту.

**HTTPS** – HyperText Transfer Protocol Secure — захищена версія HTTP.

**IaC** – Infrastructure as Code — підхід до керування інфраструктурою за допомогою коду.

**IAM** – Identity and Access Management — сервіс керування доступом і ролями.

**JSON** – JavaScript Object Notation — формат обміну даними.

**JWT** – JSON Web Token — формат токенів для автентифікації та авторизації.

**Kafka** – Apache Kafka — платформа потокової обробки подій.

**Kubernetes (K8s)** – Платформа оркестрації контейнерів.

**MVC** – Model-View-Controller — архітектурний шаблон проектування.

**RabbitMQ** – Система брокерів повідомлень.

**REST** – Representational State Transfer — архітектурний стиль побудови веб-сервісів.

**SOA** – Service-Oriented Architecture — сервіс-орієнтована архітектура.

**SPA** – Single Page Application — односторінковий веб-застосунок.

**UI** – User Interface — користувацький інтерфейс.

**VPC** – Virtual Private Cloud — віртуальна приватна хмарна мережа.

**Web UI** – Web User Interface — веб-інтерфейс користувача.

**XML** – eXtensible Markup Language — мова розмітки даних.

## ВСТУП

Високонавантажені системи являють собою складні архітектурні рішення, призначені для забезпечення стабільного функціонування масштабних інформаційних платформ, таких як соціальні мережі, інтернет-магазини та сервіси потокової передачі даних.

Із зростанням кількості користувачів, обсягів даних і частоти запитів значно підвищується навантаження на систему, що вимагає від неї високої стійкості, гнучкості та здатності до масштабування. В умовах інтенсивного росту навантаження традиційні монолітні архітектури нерідко демонструють обмеження щодо масштабованості та відмовостійкості, що обумовлює необхідність переходу до більш сучасних архітектурних рішень, зокрема мікросервісної архітектури, яка дозволяє забезпечити стабільну та ефективну роботу системи.

**Актуальність дослідження.** Актуальність обраної тематики обумовлена зростаючою потребою у створенні надійних і адаптивних до навантаження систем, здатних ефективно функціонувати в умовах високої інтенсивності обробки даних. Впровадження мікросервісного підходу сприяє не лише покращенню масштабованості та відмовостійкості, а й підвищує ефективність управління ресурсами.

В умовах стрімкого зростання вимог до продуктивності та доступності особливого значення набувають хмарні рішення, зокрема платформи на кшталт Amazon AWS, які надають можливість динамічного масштабування відповідно до поточних потреб системи, що є критично важливим для сервісів з непередбачуваним рівнем навантаження.

**Об'єкт дослідження** – високонавантажені інформаційні системи.

**Предмет дослідження** – архітектурні підходи та інструменти проєктування високонавантажених систем з використанням мікросервісної архітектури.



**Мета роботи** – аналіз та розробка ефективних підходів до проектування високонавантажених інформаційних систем із застосуванням мікросервісної архітектури на базі хмарних рішень від Amazon AWS.

Згідно з метою дослідження було визначено задачі:

- проаналізувати сучасні архітектурні підходи при проектуванні високонавантажених систем;
- дослідити принципи проектування висонавантажених систем;
- здійснити практичну реалізацію мікросервісної архітектури високонавантажених систем на хмарній платформі Amazon AWS.

**Методи дослідження** – аналіз і синтез архітектурних підходів, порівняльний аналіз монолітної, клієнт–серверної, мікросервісної, serverless та подійно-орієнтованої архітектур, моделювання, проектування програмних систем, експериментальне навантажувальне тестування та тестування відмовостійкості.

**Результати роботи.** Проаналізовано сучасні методології проектування високонавантажених систем та обґрунтовано доцільність використання мікросервісної архітектури в хмарних середовищах. Реалізовано мікросервісну платформу в Amazon Web Services. Результати навантажувального тестування підтвердили ефективність горизонтального масштабування, а експерименти з відмовостійкості засвідчили здатність системи зберігати працездатність у разі відмови окремих сервісів.

## **РОЗДІЛ 1. АНАЛІЗ СУЧАСНИХ МЕТОДОЛОГІЙ ПРОЄКТУВАННЯ ВИСОКОНАВАНТАЖЕНИХ СИСТЕМ**

У цьому розділі досліджуються найрозповсюджені методології проектування сучасних високонавантажених систем.

### **1.1 Монолітна архітектура (Monolithic Architecture)**

Монолітна архітектура — це єдина логічна виконувана програма, у якій усі компоненти системи тісно пов'язані та функціонують як єдине ціле [7;8]. З точки зору операційної системи, монолітний застосунок запускається як один процес у середовищі серверного застосунку, а під час оновлення або розгортання нової версії застосунок повністю замінює попередню збірку за один крок [9].

Як зазначає Fowler [10], монолітна архітектура добре підходить для невеликих команд розробки, оскільки весь застосунок зберігається у спільному кодовому базисі, що спрощує комунікацію між розробниками та тестування функціоналу. Red Hat [11] також підкреслює, що така архітектура спрощує налагодження і розгортання, адже всі модулі знаходяться в одному пакеті, а внутрішні виклики не створюють мережових затримок, що позитивно впливає на продуктивність системи.

Однак із розвитком систем і збільшенням кількості функціональних модулів монолітна структура починає створювати низку обмежень. За Newman [12], головним недоліком моноліту є складність масштабування: масштабування здійснюється лише цілком для всієї програми, а не для окремих компонентів. Кожна зміна вимагає повної збірки та повторного розгортання всього застосунку, що ускладнює процеси DevOps і безперервної інтеграції.

Крім того, IBM [13] зазначає, що будь-які модифікації в одній частині коду можуть спричинити непередбачувані наслідки для інших модулів, що суттєво збільшує ризик помилок під час супроводу. Зі зростанням коду також

ускладнюється тестування, адже необхідно перевіряти всю систему після кожної зміни. Це робить моноліт менш гнучким у порівнянні з більш сучасними архітектурними підходами.

### Monolithic Architecture

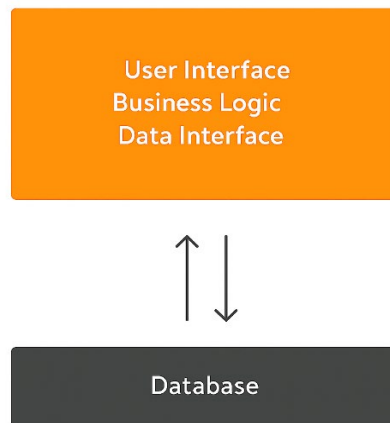


Рисунок 1.1 - Схема монолітної архітектури

#### **Переваги монолітної архітектури:**

- Простота початкової розробки та розгортання.
- Відсутність мережових затримок між модулями.
- Єдине середовище виконання спрощує налагодження.
- Легше забезпечити цілісність даних та узгодженість логіки.

#### **Недоліки монолітної архітектури:**

- Складність масштабування — неможливо масштабувати лише окремий компонент.
  - Будь-яка зміна потребує повного перескладання та повторного розгортання застосунку.
  - Зростання коду ускладнює тестування та супровід.
  - Підвищений ризик впливу локальних змін на всю систему.

## 1.2 Клієнт-серверна архітектура (Client-Server Architecture)

Клієнт-серверна архітектура — це модель побудови системи, у якій функціональність розподіляється між двома основними компонентами: клієнтом, що відповідає за інтерфейс користувача (UI) та взаємодію з ним, і сервером, який обробляє бізнес-логіку, запити та керує доступом до даних [14;15].

У такій архітектурі клієнт ініціює запит, а сервер виконує його обробку і повертає результат. Цей підхід став ключовим у розвитку мережесистем та веб-систем, оскільки забезпечує централізоване управління даними та спрощує адміністрування [16].

Microsoft [17] визначає клієнт-серверну модель як основу більшості сучасних корпоративних систем, де логіка застосунку розподілена: користувацький інтерфейс — на клієнті, а процесинг запитів, валідація даних і доступ до бази — на сервері. Це дозволяє централізовано контролювати бізнес-правила, без необхідності оновлення застосунку на кожному клієнтському пристрої окремо.

Red Hat [18] підкреслює, що головною перевагою цієї архітектури є централізація управління логікою: усі перевірки, розрахунки та правила виконуються на серверній стороні, а клієнт лише надсилає запити та відображає результати. Завдяки цьому система стає більш керованою, а підтримка - простішою.

Також до переваг належить кросплатформність клієнтів: у межах однієї архітектури можуть працювати веб-браузери, мобільні застосунки та десктопні клієнти, використовуючи єдиний API-інтерфейс [19].

Разом із тим, клієнт-серверна архітектура має й свої обмеження. Основною проблемою є навантаження на серверну частину: оскільки сервер виступає центральною точкою обробки запитів, він може стати «вузьким місцем» при великій кількості користувачів [20]. IBM [21] зазначає, що при масштабуванні таких систем основним підходом зазвичай є вертикальне масштабування — збільшення ресурсів одного сервера, що має фізичні

обмеження і з часом стає неефективним. Крім того, між клієнтом і сервером існує потенційна затримка у передачі даних, яка залежить від швидкості мережі, кількості користувачів та обсягу даних, що передаються. Ці затримки часто компенсуються за рахунок кешування або оптимізації протоколів [22].



Рисунок 1.2 - Клієнт-серверна архітектура

#### **Переваги клієнт-серверної архітектури:**

- Централізоване управління логікою системи.
- Просте оновлення — достатньо змінити серверну частину.
- Підтримка різних клієнтів (веб, мобільний, десктоп).
- Зручність адміністрування та контролю доступу.
- Можливість повторного використання серверних компонентів різними клієнтами.

#### **Недоліки клієнт-серверної архітектури:**

- Обмежене масштабування при великому навантаженні.
- Сервер може стати єдиною точкою відмови системи.
- Затримки при обміні даними між клієнтом і сервером.
- Залежність продуктивності від стабільності мережі.

### **1.3 Мікросервісна архітектура (Microservices Architecture)**

Мікросервісна архітектура — це підхід до побудови програмних систем, у якому застосунок складається з набору невеликих, автономних сервісів, кожен із яких реалізує окрему бізнес-функцію та взаємодіє з іншими через стандартизовані інтерфейси (зазвичай REST або gRPC API) [23; 24].

Кожен сервіс має власний цикл життя, логіку, базу даних (або сховище) та може розгортатися незалежно від інших. Такий підхід дозволяє створювати гнучкі, масштабовані й легко підтримувані системи [25].

Згідно з Newman [26], головна ідея мікросервісів полягає в декомпозиції великого моноліту на набір незалежних сервісів, які можна розробляти, тестувати й розгортати окремо. Це значно підвищує швидкість розробки, адже різні команди можуть працювати над різними частинами системи одночасно, не заважаючи одна одній.

Microsoft [27] зазначає, що мікросервісна архітектура забезпечує гнучке масштабування: якщо один із сервісів починає відчувати перевантаження, його можна масштабувати незалежно, не змінюючи конфігурацію інших компонентів. Це знижує витрати на інфраструктуру та підвищує ефективність використання ресурсів.

Red Hat [28] та AWS [29] підкреслюють, що розподілена природа мікросервісів також підвищує відмовостійкість системи. Якщо один сервіс виходить із ладу, решта продовжують працювати, забезпечуючи контрольовану деградацію функціоналу без повного збою. Такий підхід критично важливий для високонавантажених або безперервно доступних систем.

Разом із тим, IBM [30] застерігає, що гнучкість мікросервісів має ціну — зростання складності взаємодії між сервісами, потребу у надійній комунікаційній інфраструктурі, централізованому логуванні, моніторингу та автоматизації CI/CD. Крім того, збільшується кількість мережових викликів, що призводить до зростання затримок і накладних витрат на передачу даних.

Переваги мікросервісної архітектури:

- Незалежна розробка, тестування та розгортання окремих сервісів.
- Масштабування лише тих компонентів, які цього потребують.
- Підвищена відмовостійкість і стійкість системи до збоїв.
- Можливість використання різних технологій і мов програмування для різних сервісів.

- Підвищена гнучкість і швидкість оновлення бізнес-функціоналу.

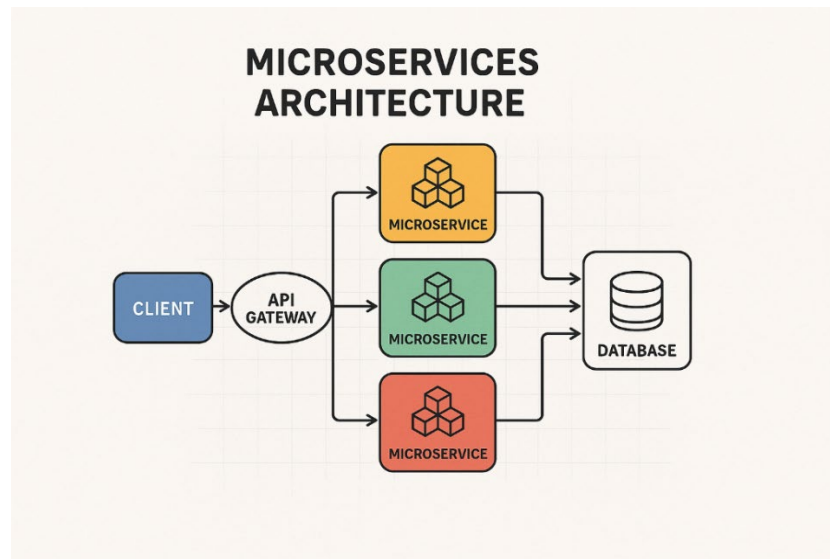


Рисунок 1.3 - Мікросервісна архітектура

Недоліки мікросервісної архітектури:

- Підвищена складність комунікацій між сервісами.
- Необхідність розвинених DevOps-практик, CI/CD, моніторингу та логування.
- Зростання мережових накладних витрат і потенційних затримок.
- Складність відстеження транзакцій, що охоплюють кілька сервісів.
- Збільшення вимог до інфраструктури та досвіду команди.

#### 1.4 Сервіс-орієнтована архітектура (SOA — Service-Oriented Architecture)

Сервісно-орієнтована архітектура (SOA) — це підхід до побудови програмних систем, у якому застосунок складається з набору незалежних сервісів, що взаємодіють між собою через стандартизовані протоколи та інтерфейси. В основі SOA лежить концепція повторного використання сервісів і чітко визначених контрактів взаємодії, зазвичай реалізованих за допомогою протоколів SOAP та форматів XML або WSDL [31; 32].

Ключовим елементом цієї архітектури є Enterprise Service Bus (ESB) — проміжний шар, який відповідає за маршрутизацію, трансформацію даних, безпеку, оркестрацію сервісів та управління транзакціями [33]. За даними The Open Group [34], ESB виступає як “центральна шина” взаємодії між сервісами, абстрагуючи їхню внутрішню реалізацію і забезпечуючи сумісність навіть між різними технологічними стекками.

Oracle [35] підкреслює, що SOA стала основою для побудови великих корпоративних систем, оскільки дозволяє інтегрувати різноманітні застосунки — CRM, ERP, білінгові чи аналітичні рішення — у єдину логічну систему. Це дає змогу централізовано керувати бізнес-процесами, забезпечуючи узгодженість даних між підсистемами. Gartner [36] додає, що SOA сприяє повторному використанню бізнес-логіки — один сервіс може бути викликаний із різних застосунків або модулів, що підвищує ефективність розробки й знижує дублювання функціоналу.

Однак, як зазначають IBM [37] та MuleSoft [38], архітектура SOA має і низку обмежень. Основним викликом є складність налаштування та управління інфраструктурою, оскільки необхідно забезпечити правильну конфігурацію ESB, а також підтримувати каталоги сервісів (Service Registry) і метадані. Будь-яка помилка або перевантаження ESB може призвести до збоїв у всій системі, адже ESB є єдиною точкою інтеграції між усіма сервісами. Крім того, SOA потребує ретельного моніторингу, масштабування та резервування, щоб уникнути перетворення ESB на «вузьке горлечко» [39].

### **Переваги сервісно-орієнтованої архітектури:**

- Повторне використання бізнес-логіки та сервісів у різних бізнес-процесах.
- Централізований контроль доступу, безпеки та транзакцій.
- Високий рівень інтеграції між різними системами (CRM, ERP, Billing тощо).



- Незалежність технологічних стеків між сервісами (можливість гетерогенного середовища).
- Підтримка оркестрації складних бізнес-процесів через ESB.
- 

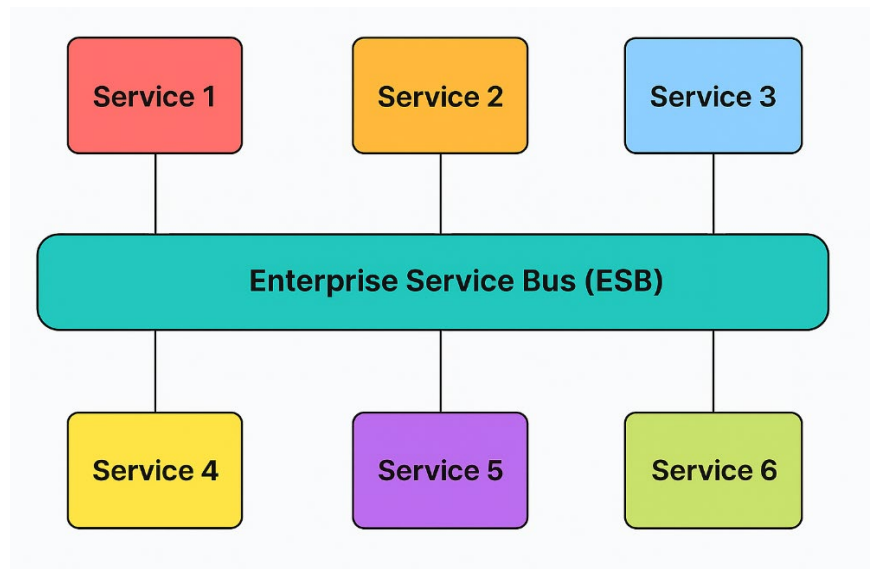


Рисунок 1.4 - Сервіс-орієнтована архітектура

#### **Недоліки сервісно-орієнтованої архітектури:**

- Висока складність налаштування та підтримки інфраструктури.
- Залежність від центрального елемента — Enterprise Service Bus (ESB).
- Підвищені вимоги до моніторингу, логування і керування сервісами.
- Потенційна затримка при маршрутизації запитів через посередницький шар.
- Менша гнучкість порівняно з мікросервісним підходом.

### **1.5. Serverless / Function-as-a-Service Architecture (FaaS)**

Serverless або Function-as-a-Service (FaaS) — це архітектурний підхід, у якому розробники можуть запускати окремі функції або частини бізнес-логіки без необхідності керувати серверами чи інфраструктурою вручну. Хмарний провайдер автоматично забезпечує масштабування, моніторинг, розподіл навантаження та оплату лише за фактичний час виконання функцій [40; 41].

Згідно з AWS [42], у Serverless-підході застосунок складається з невеликих автономних функцій, які виконуються у відповідь на певні події — HTTP-запити, повідомлення у черзі, зміну файлів у сховищі тощо. Це дозволяє створювати подійно-орієнтовані системи, у яких компоненти ізольовані, а інфраструктура повністю керується постачальником послуг.

Google Cloud [43] наголошує, що FaaS дає можливість масштабувати застосунок автоматично, виходячи з кількості викликів функцій, без ручного втручання або налаштування серверів. Таким чином, компанія сплачує лише за фактичне використання обчислювальних ресурсів, що робить цей підхід економічно ефективним для непостійних або змінних навантажень.

IBM [44] визначає основною перевагою Serverless-архітектури **спрощення життєвого циклу розробки**: розробники можуть зосередитися на бізнес-логіці, не витрачаючи час на адміністрування серверів, резервування, безпеку чи оновлення системи. Крім того, ThoughtWorks [45] зазначає, що FaaS підвищує **гнучкість системи**, оскільки дозволяє комбінувати функції з різних сервісів (наприклад, AWS Lambda, Google Cloud Functions або Azure Functions) у єдині робочі процеси.

Разом із тим, як підкреслює IEEE [46], Serverless має і свої недоліки. Основним викликом є **обмежена тривалість виконання функцій** та залежність від провайдера (vendor lock-in). Також складно реалізувати тривалі транзакційні процеси або низькорівневий контроль ресурсів. Cloud Native Computing Foundation (CNCF) [47] звертає увагу, що відсутність постійного сервера ускладнює налагодження, моніторинг і трасування подій, а також збільшує ризики холодного старту функцій, що може вплинути на продуктивність у критичних сценаріях.

#### **Переваги Serverless / FaaS архітектури:**

- Відсутність необхідності керування серверами або інфраструктурою.
- Автоматичне масштабування залежно від навантаження.
- Оплата лише за фактичний час виконання функцій.
- Висока гнучкість і швидкість розробки.

- Природна інтеграція з подійно-орієнтованими системами.

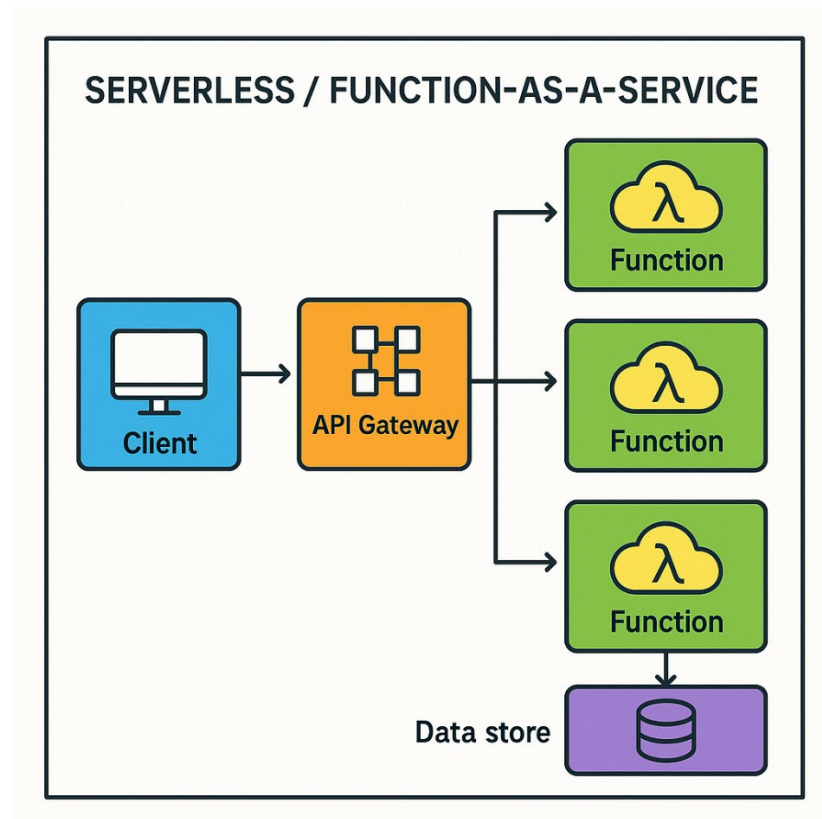


Рисунок 1.5 - Serverless / Function-as-a-Service Architecture

#### Недоліки Serverless / FaaS архітектури:

- Залежність від хмарного провайдера (vendor lock-in).
- Обмеження за тривалістю виконання функцій.
- Складність у налагодженні, моніторингу та тестуванні.
- Проблеми з холодним стартом і латентністю.
- Ускладнення при реалізації складних транзакцій або станів.

### 1.6 Подійно-орієнтована архітектура (EDA — Event-Driven Architecture)

Подійно-орієнтована архітектура (Event-Driven Architecture, EDA) — це підхід до побудови розподілених систем, у якому компоненти взаємодіють між собою через **події** — повідомлення, що сигналізують про зміну стану або виконання певної дії в системі [48; 49].

Замість прямого виклику методів, як у клієнт-серверній моделі, EDA використовує **асинхронну комунікацію**, що забезпечує незалежність компонентів і підвищує гнучкість системи.

Gartner [50] визначає подійно-орієнтовану архітектуру як один із ключових стилів інтеграції для сучасних високонавантажених систем, що дозволяє організаціям швидко реагувати на бізнес-події у режимі реального часу.

Згідно з Apache [51], реалізація EDA зазвичай базується на **публікаційно-підписній моделі (publish/subscribe)**, де один компонент (видавець) генерує події, а інші (споживачі) підписуються на ті, які їх цікавлять.

Confluent [52] підкреслює, що найпоширенішою технологічною реалізацією EDA є **платформи потокової обробки подій**, такі як *Apache Kafka*, *RabbitMQ* або *AWS Kinesis*, які дозволяють передавати, зберігати й обробляти події у реальному часі. Завдяки цьому система може миттєво реагувати на зміни даних або станів без необхідності централізованих запитів.

Oracle [53] зазначає, що подійно-орієнтовані системи особливо ефективні у сценаріях із високою інтенсивністю даних — таких як моніторинг, аналітика, обробка транзакцій чи IoT-рішення. EDA забезпечує **високу масштабованість і стійкість**, оскільки події зберігаються у чергах або потоках і можуть бути повторно оброблені при збої. Red Hat [54] додає, що така архітектура природно підходить для мікросервісного середовища, де кожен сервіс реагує лише на ті події, які йому потрібні, що дозволяє досягти слабкого зв'язку між компонентами системи.

Разом із тим, як зазначає IEEE [55], EDA має і свої обмеження. Серед них — **складність управління потоками подій**, необхідність у точному проєктуванні схем подій та забезпеченні ідемпотентності операцій. NIST [56] підкреслює, що відстеження транзакцій і відлагодження подій у таких системах є нетривіальним завданням, адже комунікація відбувається асинхронно й у великому обсязі. Крім того, зростає потреба у централізованому моніторингу, трасуванні та контролі порядку подій.

## Event-Driven Architecture (EDA) pattern

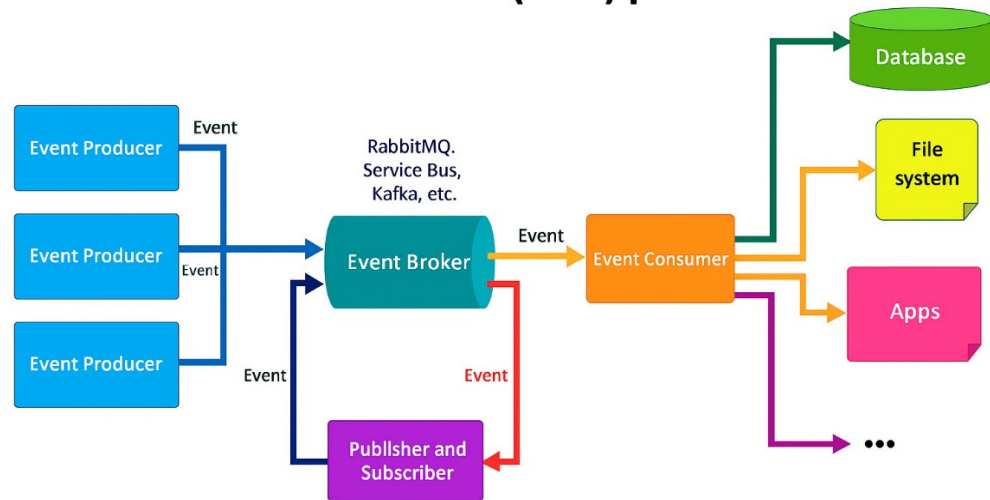


Рисунок 1.6 - Подійно-орієнтована архітектура

### Переваги подійно-орієнтованої архітектури:

- Висока масштабованість і асинхронна взаємодія між компонентами.
- Реактивна модель — система реагує на події у режимі реального часу.
- Слабке зв'язування між сервісами, що підвищує гнучкість і надійність.
- Підтримка повторної обробки подій у разі збоїв.
- Ефективна інтеграція з мікросервісами, IoT, аналітикою та потоковими системами.

системами.

### Недоліки подійно-орієнтованої архітектури:

- Підвищена складність проектування та відлагодження.
- Необхідність забезпечення ідемпотентності операцій.
- Високі вимоги до моніторингу, трасування та керування подіями.
- Можливі затримки при великій кількості одночасних подій.
- Ускладнення гарантій порядку обробки повідомлень.

Таблиця 1.1

## Порівняльна таблиця основних архітектурних підходів

Архітектура	Короткий опис	Масштабування	Відмовостійкість	Супроводжуваність
<b>Монолітна</b>	Єдиний застосунок, що розгортається одним пакетом	Вертикальне, «все або нічого»	Низька — збій одного модуля впливає на всю систему	Низька — з часом зростає складність підтримки й тестування
<b>Клієнт-серверна</b>	Поділ на клієнтську (UI) та серверну частини	Переважно вертикальне	Середня — залежить від стабільності сервера	Середня — проста логіка, але потребує підтримки обох частин
<b>Мікросервісн а</b>	Система з незалежних сервісів, що взаємодіють через API	Горизонтальне за сервісами	Висока — збій локалізований	Висока — ізольовані модулі легко оновлювати та тестувати
<b>SOA (Service-Oriented)</b>	Сервіси з'єднані через ESB, взаємодія SOAP/XML	Частково горизонтальне	Середня — критична роль ESB	Середня — складність документації та оновлень
<b>Serverless / FaaS</b>	Виконання функцій у хмарі, інфраструктура керується провайдером	Автоматичне, динамічне	Середня — залежить від провайдера	Висока — мінімальна підтримка інфраструктури
<b>EDA (Event-Driven)</b>	Компоненти реагують на події через брокер (Kafka, RabbitMQ)	Горизонтальне на рівні підписників	Висока — слабо зв'язані компоненти	Середньо-висока — потрібна централізована система логів і трейсингу

У таблиці 1.1 наведено порівняльну характеристику основних архітектурних підходів, що застосовуються при проєктуванні високонавантажених систем. Порівняння здійснено за ключовими критеріями, зокрема масштабованістю, відмовостійкістю та супроводжуваністю, що дозволяє оцінити доцільність використання кожного підходу залежно від вимог до системи.

## 1.7. Принципи проєктування високонавантажених систем

У наш час стрімкого розвитку технологій високонавантажені системи стали основою сучасного світоустрою, адже вони є необхідними для функціонування великих інформаційних платформ, таких як соціальні мережі, інтернет-магазини та сервіси потокової передачі даних. Сьогодні важко уявити наше життя без таких систем. [1]

На думку N.Badwaik, з якою ми погоджуємося, важливо створювати платформи, які не лише ефективно працюють за різного рівня навантаження, а й здатні масштабуватись у відповідь на зміну ринкових вимог. Проектування архітектури системи з високою доступністю та масштабованістю є критичним завданням у сьогоднішньому світі, де бізнес і сервіси залежать від безперебійної роботи IT-інфраструктури. [2]

Під час проєктування високонавантажених систем у першу чергу варто звернути увагу на такі три аспекти [3]:

### *Надійність (Reliability):*

Система повинна коректно функціонувати (виконувати необхідні дії з потрібною продуктивністю) навіть у разі збоїв обладнання, програмних помилок або людського фактора.

### *Масштабованість (Scalability):*

У міру зростання обсягу даних, кількості користувачів або складності логіки має існувати розумне рішення, яке дозволяє системі адаптуватися.

### *Супроводжуваність (Maintainability):*

Із системою працюватимуть багато людей — розробники, адміністратори, інженери з експлуатації — і вони повинні мати можливість робити це ефективно й без труднощів.

### **1.7.1. Питання надійності при проектуванні високонавантажених систем**

Кожен користувач має інтуїтивне уявлення про те, що означає надійність чи ненадійність певної системи. У контексті програмного забезпечення під надійністю зазвичай мають на увазі такі характеристики:

- Застосунок виконує функціональні завдання відповідно до очікувань користувача.
- Система здатна витримувати помилки користувача або її використання непередбачуваним способом.
- Продуктивність є задовільною для визначених сценаріїв використання, з урахуванням прогнозованого навантаження та обсягу даних.
- Система забезпечує захист від несанкціонованого доступу та потенційного зловживання.

У разі коли всі ці аспекти виконуються одночасно, йдеться про «коректну роботу» системи. Відповідно, поняття надійності можна трактувати як здатність системи продовжувати функціонування в штатному режимі навіть за умов виникнення збоїв.

Під збоями розуміють порушення роботи окремих компонентів системи, що відхиляються від передбачених специфікацій. Системи, що передбачають можливість виникнення таких відхилень та здатні адекватно реагувати на них, називаються відмовностійкими чи резилієнтними. Варто зазначити, що термін "відмовностійкість" може вводити в обман, оскільки створити систему, стійку до всіх типів збоїв, є технічно неможливо. Наприклад, повне знищення інфраструктури (внаслідок, умовно, астрофізичних подій) апріорі не передбачає наявності інженерних рішень для збереження працездатності. Отже, раціонально говорити лише про обмежений набір типових збоїв, до яких система повинна бути адаптована.

При цьому важливо розрізняти поняття «збій» та «відмова» [4]. Збій стосується часткової несправності, тобто порушення роботи окремого компонента, тоді як відмова — це неспроможність системи в цілому



забезпечити надання передбаченого сервісу. Зважаючи на те, що повне усунення ймовірності виникнення збоїв є недосяжним, необхідно впроваджувати механізми, що мінімізують ризики перетворення збоїв на повноцінні відмови.

Практика розробки відмовостійких систем передбачає використання інженерних підходів, що можуть включати надмірне моделювання збоїв (наприклад, довільне завершення процесів) для верифікації надійності відповідних механізмів. Значна частка критичних помилок у сучасному програмному забезпеченні зумовлена некоректною або неповною обробкою виключних ситуацій [5]. Тому активне тестування відмовостійкості дозволяє підвищити впевненість у стабільності поведінки системи в умовах реальних інцидентів. Відомим прикладом такого підходу є інструмент Netflix Chaos Monkey [6], який навмисне підсвічує збої з метою перевірки системної стійкості.

Незважаючи на переважне прагнення до адаптації систем до збоїв, існують сценарії, в яких запобігання є єдиною можливим варіантом. Зокрема, йдеться про питання інформаційної безпеки: у разі компрометації системи та витоку чутливих даних подію неможливо відкотити. Втім, у цьому дослідженні основну увагу приділено збоям, які можуть бути усунені або локалізовані за допомогою відповідних інженерних рішень, що будуть розглянуті в наступних розділах.

### **1.7.2. Питання масштабованості при проектуванні високонавантажених систем**

Навіть якщо система зараз функціонує стабільно, це зовсім не означає, що вона залишиться такою в майбутньому. Однією з основних причин зниження ефективності часто стає зростання навантаження: наприклад, кількість одночасних користувачів може зрости з десяти тисяч до ста тисяч, або обсяг оброблюваних даних — у десятки разів.

У таких випадках постає питання масштабованості — здатності системи адаптуватися до збільшення навантаження. Водночас не зовсім коректно просто стверджувати, що якась система «масштабується» або «не масштабується». Така оцінка набуває змісту лише тоді, коли розглядаються конкретні сценарії росту та відповідні варіанти реагування на них.

Тобто, замість формальних ярликів, варто ставити практичні запитання:

— Що відбудуватиметься зі системою, якщо кількість користувачів продовжить зростати?

— Як ми можемо забезпечити стабільну роботу в умовах збільшеного навантаження?

— Які ресурси можна додати, щоб підтримати систему в нових умовах?

Таким чином, масштабованість — це не властивість "у вакуумі", а радше здатність системи гнучко реагувати на зміни та зростання.

### **1.7.3. Питання супроводжуваності при проектуванні високонавантажених систем**

Загальновідомо, що більшість витрат на програмне забезпечення виникає не під час його створення, а вже після запуску — на підтримку. Це включає усунення помилок, забезпечення стабільної роботи, аналіз причин збоїв, адаптацію до нових середовищ, внесення змін для нових потреб, виправлення архітектурних недоліків і розширення функціоналу.

Попри це, підтримка вже існуючих систем часто сприймається інженерами як рутинна або навіть неприємна робота. Чужий код, незручні технології, застарілі рішення — усе це робить роботу з такими системами непростою. У кожної з них — свої складнощі, тому дати універсальну інструкцію, як із ними працювати, доволі складно.

Проте ми можемо спроектувати систему так, щоб зменшити ризик перетворення її на застарілу та важко підтримувану в майбутньому. Для цього слід звернути увагу на кілька базових принципів, які допомагають створювати більш життєздатні рішення:

Зручність у підтримці.

Система повинна бути простою в експлуатації — її легко моніторити, оновлювати та відновлювати у разі проблем. Це важливо для команд, які відповідають за її роботу.

Зрозуміла структура.

Складність системи має бути мінімальною настільки, наскільки це можливо. Новий розробник повинен мати змогу швидко розібратися, як вона влаштована. Це значно полегшує подальшу роботу з кодом.

Готовність до змін.

Система повинна мати таку архітектуру, яка дозволяє безболісно вносити зміни, коли змінюються бізнес-вимоги. Це означає хорошу розширюваність, модульність і чітку внутрішню логіку.

Як і у випадку з надійністю чи масштабуванням, не існує «чарівної кнопки», яка гарантує ці властивості. Але якщо з самого початку підходити до проектування з думкою про майбутню підтримку, це дозволить уникнути багатьох проблем у перспективі.

Таким чином, з точки зору надійності мікросервісний підхід забезпечує підвищену відмовостійкість за рахунок ізоляції функціональних компонентів. Незалежність окремих сервісів дозволяє локалізувати збої та запобігати каскадним відмовам, унаслідок чого порушення роботи одного модуля не призводить до зупинки всієї системи. Така властивість відповідає вимогам резилієнтності, які є критичними для розподілених платформ з високим рівнем доступності.

У контексті масштабованості мікросервісна архітектура забезпечує гнучку реакцію на зростання навантаження. На відміну від монолітних систем, де масштабування передбачає розгортання всієї програми, мікросервісний підхід дозволяє збільшувати обчислювальні ресурси лише для окремих сервісів, які створюють вузькі місця продуктивності. Це сприяє оптимальному використанню інфраструктури та ефективній інтеграції з хмарними обчислювальними середовищами.

З позиції супроводжуваності мікросервісна архітектура значно підвищує ефективність процесів розробки та експлуатації. Модульна структура системи дає змогу різним командам незалежно розробляти, тестувати й розгортати окремі сервіси без негативного впливу на інші компоненти. У результаті знижуються ризики впровадження змін, скорочується час виходу нових версій та підвищується стабільність релізів.

Крім того, мікросервісна архітектура підтримує принципи розширюваності та технологічної гнучкості, що дозволяє поступово впроваджувати нові функціональні можливості та технологічні рішення без суттєвих архітектурних змін. Це є особливо важливим в умовах динамічного розвитку ринку та постійного зростання вимог до продуктивності й масштабів систем.

## Висновки до розділу 1

Проведений аналіз основних архітектурних підходів до проектування програмних систем — зокрема монолітної, клієнт–серверної, сервіс-орієнтованої (SOA), serverless та подійно-орієнтованої архітектур — засвідчує, що кожен із розглянутих підходів має як переваги, так і суттєві обмеження, які проявляються залежно від масштабів системи, характеру навантаження та вимог до надійності й гнучкості.

Монолітна архітектура характеризується простотою реалізації та розгортання, однак демонструє низьку масштабованість і ускладнює супровід при зростанні функціональності та навантаження. Клієнт–серверна модель забезпечує централізоване управління бізнес-логікою, проте має обмеження щодо продуктивності та створює єдину точку відмови. Архітектура SOA сприяє повторному використанню бізнес-компонентів, але водночас ускладнює інфраструктуру через залежність від шини корпоративних сервісів (ESB). Serverless-підхід зменшує обсяг інфраструктурного адміністрування, однак накладає технічні обмеження на час виконання, споживання ресурсів і стабільність роботи. Подійно-орієнтована архітектура забезпечує високу асинхронність і гнучкість, проте потребує складних механізмів моніторингу, логування та гарантій доставки повідомлень.

У цьому контексті мікросервісна архітектура виступає найбільш збалансованим рішенням для побудови високонавантажених розподілених систем у хмарних середовищах. Вона поєднує ключові переваги розподілених архітектур, зокрема незалежність компонентів, горизонтальне масштабування, ізольоване оновлення сервісів та підвищену відмовостійкість, з можливістю чіткого контролю життєвого циклу кожного мікросервісу. Додатковою перевагою є органічна інтеграція мікросервісного підходу з сучасними хмарними платформами та контейнерними оркестраторами.

Таким чином, для складних програмних систем, що повинні забезпечувати високу продуктивність, безперервну доступність і швидку адаптацію до змін бізнес-вимог, мікросервісна архітектура є найбільш доцільним архітектурним підходом. Вона забезпечує ефективне використання обчислювальних ресурсів, підвищує стабільність системи та створює надійну основу для подальшого розвитку в умовах динамічних хмарних інфраструктур.

Також встановлено, що на основі розглянутих аспектів проєктування програмних систем, зокрема надійності, масштабованості та супроводжуваності, можна зробити обґрунтований висновок, що мікросервісна архітектура є найбільш доцільною моделлю для побудови сучасних високонавантажених систем. Мікросервісна архітектура комплексно поєднує ключові характеристики високонавантажених програмних систем — надійність, масштабованість і супроводжуваність, забезпечуючи стабільну роботу, зручність модернізації та стійкість до зростаючих вимог користувачів і бізнесу. Саме це обґрунтовує її доцільність як базового архітектурного підходу для сучасних розподілених систем.

## РОЗДІЛ 2. ПРАКТИЧНА ЧАСТИНА РЕАЛІЗАЦІЇ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ В ХМАРНИХ СЕРЕДОВИЩАХ

### 2.1. Загальні відомості та архітектура практичної частини

З метою дослідження доцільності та практичної ефективності використання мікросервісної архітектури в хмарних середовищах у межах даної роботи було спроектовано та реалізовано експериментальну програмну платформу, побудовану відповідно до принципів мікросервісного підходу [57, 58]. Основною метою практичної частини є перевірка теоретичних положень, розглянутих у попередніх розділах, шляхом створення реальної системи та проведення практичних експериментів, спрямованих на оцінку масштабованості, надійності та гнучкості обраної архітектури [59].

У якості об'єкта практичного дослідження було розроблено платформу електронної комерції, надалі іменовану мікросервісний онлайн-магазин. Дана платформа імітує типову сучасну систему онлайн-торгівлі та складається з набору незалежних сервісів, кожен з яких відповідає за окрему бізнес-функцію [60]. Такий вибір предметної області зумовлений її репрезентативністю для високонавантажених систем та широким застосуванням у реальних комерційних рішеннях [61].

Архітектура платформи включає шість основних мікросервісів [57, 62]:

`billing_service` — сервіс, що відповідає за внутрішній білінг платформи, облік фінансових операцій та взаєморозрахунки між компонентами системи;

`payment_service` — сервіс, призначений для інтеграції з зовнішніми платіжними системами та обробки платежів користувачів [63];

`product_service` — сервіс, який забезпечує управління товарами, зокрема зберігання інформації про продукти, їх характеристики та доступність;

`user_service` — сервіс, відповідальний за управління користувачами, включаючи реєстрацію, автентифікацію, авторизацію та обробку профілів [64];

`api_gateway` — координаційний сервіс, який виступає єдиною точкою входу до платформи, маршрутизує запити до відповідних мікросервісів та реалізує базові кроссервісні функції [65];

`web_ui` — веб-інтерфейс платформи, призначений для взаємодії з кінцевими користувачами та адміністраторами, забезпечуючи можливість придбання товарів і керування контентом.

Кожен із зазначених сервісів реалізований як автономний компонент із власною зоною відповідальності, що відповідає принципам слабкої зв'язаності та високої модульності, характерним для мікросервісної архітектури [57, 58]. Взаємодія між сервісами здійснюється через стандартизовані інтерфейси, що дозволяє незалежно розгортати, масштабувати та оновлювати окремі компоненти системи [66].

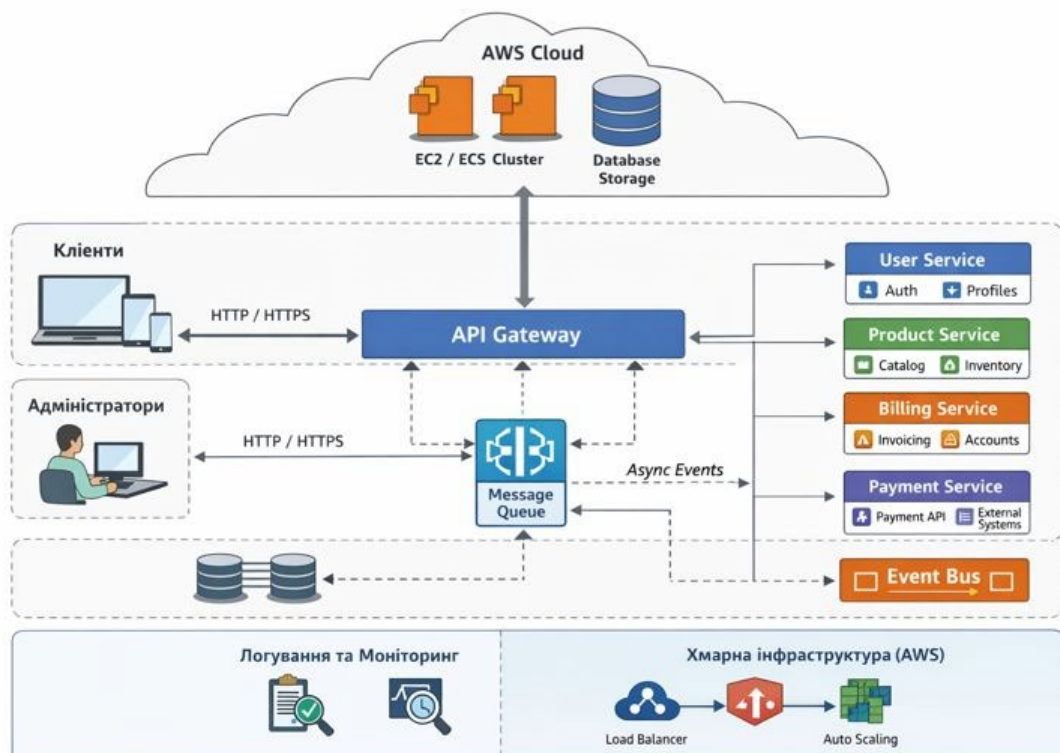


Рисунок 2.1 – Загальна архітектура мікросервісного онлайн-магазину



У якості хмарного середовища для розгортання платформи було обрано інфраструктуру Amazon Web Services (AWS), яка є одним із найбільш поширених та надійних хмарних провайдерів у світі [67]. Використання AWS дозволило реалізувати практичну частину з урахуванням сучасних підходів до хмарних обчислень, а також забезпечити можливість горизонтального масштабування, високої доступності та відмовостійкості системи [68].

Загальна архітектура розробленої платформи, а також взаємодія її основних компонентів представлена на рисунку 2.1, який ілюструє логічну структуру мікросервісного онлайн-магазину та принципи організації взаємодії між сервісами.

## **2.2. Розгортання інфраструктури**

Розгортання інфраструктури практичної частини магістерської роботи було виконано відповідно до сучасної методології Infrastructure as Code (IaC), яка передбачає опис усіх інфраструктурних компонентів у вигляді програмного коду. Застосування даного підходу забезпечує відтворюваність середовищ, автоматизацію процесів розгортання, зменшення ризику конфігураційних помилок та спрощення подальшого супроводу інфраструктури.

Для реалізації підходу IaC було використано інструмент Pulumi, який дозволяє описувати хмарну інфраструктуру за допомогою мов програмування загального призначення та має повноцінну інтеграцію з сервісами Amazon Web Services. Це надало можливість створювати, змінювати та видаляти інфраструктурні ресурси у декларативному та керованому вигляді.

```
import pulumi
import pulumi_aws as aws
import json

account_id = aws.get_caller_identity().account_id
region = "eu-north-1"
```

На початковому етапі було визначено конфігураційні параметри для кожного мікросервісу платформи. До них належать мережеві порти, кількість екземплярів сервісу, що запускаються, а також змінні середовища, необхідні для коректної роботи контейнерів. Така централізована конфігурація забезпечує уніфікацію налаштувань і спрощує масштабування системи.

```
services_config = {  
    "user_service": {  
        "port": 8000,  
        "replicas": 1,  
        "env": {  
            "AWS_REGION": "eu-north-1",  
            "AWS_ACCESS_KEY_ID": "...",  
            "AWS_SECRET_ACCESS_KEY": "...",  
            "USER_TABLE_NAME": "users",  
            "SECRET_KEY": "...",  
        },  
    },  
    "product_service": {  
        "port": 8000,  
        "replicas": 1,  
        "env": {  
            "AWS_REGION": "eu-north-1",  
            "AWS_ACCESS_KEY_ID": "...",  
            "AWS_SECRET_ACCESS_KEY": "...",  
            "PRODUCT_TABLE_NAME": "products",  
            "INVENTORY_TABLE_NAME": "inventory",  
            "HISTORY_TABLE_NAME": "history",  
        },  
    },  
}
```

Наступним кроком стало створення обчислювального кластера, у межах якого здійснюється виконання контейнеризованих сервісів. Для логічного розмежування компонентів і забезпечення коректної міжсервісної взаємодії було визначено окремий простір імен, що дозволяє сервісам взаємодіяти між собою в межах ізольованого середовища.

```
cluster = aws.ecs.Cluster("cluster")
```

```

▼ namespace = aws.servicediscovery.PrivateDnsNamespace(
    "local",
    name="local",
    vpc=aws.ec2.get_vpc(default=True).id,
)

```

Для забезпечення можливості виконання контейнерів у хмарному середовищі було налаштовано ролі доступу, необхідні для роботи з ресурсами Amazon Web Services. Кожному контейнеру було призначено відповідну роль, яка визначає його права доступу до сервісів AWS.

```

▼ execution_role = aws.iam.Role(
    "execution-role",
    assume_role_policy=json.dumps({
        "Version": "2012-10-17",
        "Statement": [{
            "Action": "sts:AssumeRole",
            "Principal": {"Service": "ecs-tasks.amazonaws.com"},
            "Effect": "Allow"
        }]
    })
)

```

Додатково до ролей були прив'язані політики доступу, що регламентують дозволені операції, зокрема запуск і управління контейнерними завданнями. Такий підхід відповідає принципу мінімально необхідних привілеїв і підвищує загальний рівень безпеки інфраструктури.

```
aws.iam.RolePolicy(
    "execution-role-logs",
    role=execution_role.name,
    policy=json.dumps({
        "Version": "2012-10-17",
        "Statement": [{
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:PutLogEvents"
            ],
            "Resource": "*"
        }]
    })
)
```

У процесі розгортання було використано стандартну віртуальну приватну мережу (VPC), а також її підмережі, у межах яких розміщуються контейнерні сервіси.

```
default_vpc = aws.ec2.get_vpc(default=True)
```

Для контролю мережевого трафіку були налаштовані групи безпеки, які виконують функції мережевого екрану та визначають правила для вхідних і вихідних з'єднань.

```
sg = aws.ec2.SecurityGroup(
    "app-sg",
    vpc_id=default_vpc.id,
    ingress=[aws.ec2.SecurityGroupIngressArgs(
        protocol="-1",
        from_port=0,
        to_port=0,
        cidr_blocks=["0.0.0.0/0"],
    )],
    egress=[aws.ec2.SecurityGroupEgressArgs(
        protocol="-1",
        from_port=0,
        to_port=0,
        cidr_blocks=["0.0.0.0/0"],
    )],
)
```

Окрема група безпеки була створена для балансувальника навантаження з відкриттям портів 80 (HTTP), 443 (HTTPS) та 8080 для доступу до API.

```
▼ alb_sg = aws.ec2.SecurityGroup(  
    "alb-sg",  
    vpc_id=default_vpc.id,  
    ingress=[  
        aws.ec2.SecurityGroupIngressArgs(  
            protocol="tcp",  
            from_port=80,  
            to_port=80,  
            cidr_blocks=["0.0.0.0/0"],  
        ),  
        aws.ec2.SecurityGroupIngressArgs(  
            protocol="tcp",  
            from_port=443,  
            to_port=443,  
            cidr_blocks=["0.0.0.0/0"],  
        ),  
    ],  
)
```

Для забезпечення рівномірного розподілу вхідного трафіку між сервісами платформи було розгорнуто балансувальник навантаження Amazon.

```
▼ alb = aws.lb.LoadBalancer(  
    "alb",  
    load_balancer_type="application",  
    subnets=default_subnets.ids,  
    security_groups=[alb_sg.id],  
)
```

У межах його конфігурації були визначені цільові групи, які відповідають за маршрутизацію трафіку до API-шару та веб-інтерфейсу платформи.

```

▼ api_tg = aws.lb.TargetGroup(
    "api-tg",
    port=8000,
    protocol="HTTP",
    target_type="ip",
    vpc_id=default_vpc.id,
    health_check=aws.lb.TargetGroupHealthCheckArgs(
        path="/",
        interval=30,
        timeout=5,
        healthy_threshold=2,
        unhealthy_threshold=2,
        matcher="200,404",
    ),
)

```

Також були налаштовані слухачі балансувальника, що визначають порти прийому запитів і правила їх перенаправлення, зокрема маршрут за замовчуванням до веб-інтерфейсу.

```

▼ listener = aws.lb.Listener(
    "listener",
    load_balancer_arn=alb.arn,
    port=80,
    protocol="HTTP",
    default_actions=[aws.lb.ListenerDefaultActionArgs(
        type="forward",
        target_group_arn=web_tg.arn,
    )],
)

```

Розгортання кожного мікросервісу здійснювалося відповідно до визначених конфігурацій шляхом створення контейнерних сервісів у середовищі Amazon ECS. Для цього використовувалися контейнерні образи, розміщені в реєстрі Amazon Elastic Container Registry (ECR), який забезпечує централізоване зберігання та керування Docker-образами.

```

for service_name, config in services_config.items():

```

Для цього використовувалися контейнерні образи, розміщені в реєстрі Amazon Elastic Container Registry (ECR), який забезпечує централізоване зберігання та керування Docker-образами:

```
ecr_url = f"{account_id}.dkr.ecr.{region}.amazonaws.com/{service_name}:latest"
```

Для кожного контейнера було визначено опис, що включає використовуваний образ, відкриті порти, змінні середовища та параметри логування.

```
container_def = {
    "name": service_name,
    "image": ecr_url,
    "essential": True,
    "portMappings": [{
        "containerPort": config["port"],
        "protocol": "tcp"
    }],
    "environment": [{"name": k, "value": v} for k, v in config["env"].items()],
    "logConfiguration": {
        "logDriver": "awslogs",
        "options": {
            "awslogs-group": f"/ecs/{service_name}",
            "awslogs-region": region,
            "awslogs-stream-prefix": "ecs",
            "awslogs-create-group": "true"
        }
    }
}
```

Окремим етапом стало визначення контейнерних завдань, у межах яких задаються обчислювальні ресурси, необхідні для роботи сервісів, зокрема обсяг процесорного часу та оперативної пам'яті. Для виконання завдань було використано режим керованого виконання, при якому хмарний провайдер автоматично виділяє необхідні обчислювальні ресурси без потреби ручного управління серверами.

```
task_def = aws.ecs.TaskDefinition(
    f"{service_name}-task",
    family=service_name,
    cpu="256",
    memory="512",
    network_mode="awsvpc",
    requires_compatibilities=["FARGATE"],
    execution_role_arn=execution_role.arn,
    container_definitions=json.dumps([container_def])
)
```

Для забезпечення коректної міжсервісної взаємодії було налаштовано механізм сервісного виявлення, який реєструє сервіси в системі доменних імен і дозволяє іншим компонентам звертатися до них за логічними ідентифікаторами. Це усуває залежність від статичних мережевих адрес і підвищує гнучкість архітектури.

```
discovery = aws.servicediscovery.Service(
    f"{service_name}-discovery",
    name=service_name,
    dns_config=aws.servicediscovery.ServiceDnsConfigArgs(
        namespace_id=namespace.id,
        dns_records=[aws.servicediscovery.ServiceDnsConfigDnsRecordArgs(
            ttl=10,
            type="A",
        )],
    ),
)
```

Балансування навантаження було застосовано лише до сервісів, що виконують роль точок входу до системи, а саме API Gateway та веб-інтерфейсу. Внутрішні мікросервіси взаємодіють між собою безпосередньо через внутрішню мережу кластера.



```

lb_config = []
if service_name == "api_gateway":
    lb_config = [aws.ecs.ServiceLoadBalancerArgs(
        target_group_arn=api_tg.arn,
        container_name=service_name,
        container_port=config["port"],
    )]
elif service_name == "web_ui":
    lb_config = [aws.ecs.ServiceLoadBalancerArgs(
        target_group_arn=web_tg.arn,
        container_name=service_name,
        container_port=config["port"],
    )]

```

Завершальним етапом стало створення та налаштування ECS-сервісів, які забезпечують запуск і підтримку заданої кількості контейнерів та автоматичне відновлення у разі збоїв.

```

service = aws.ecs.Service(
    f"{service_name}-service",
    name=service_name,
    cluster=cluster.arn,
    task_definition=task_def.arn,
    desired_count=config["replicas"],
    launch_type="FARGATE",
    network_configuration=aws.ecs.ServiceNetworkConfigurationArgs(
        subnets=default_subnets.ids,
        security_groups=[sg.id],
        assign_public_ip=True,
    ),
    service_registries=aws.ecs.ServiceServiceRegistriesArgs(
        registry_arn=discovery.arn,
    ),
    load_balancers=lb_config,
)

```

Після завершення процесу розгортання було отримано ключові параметри інфраструктури, зокрема ідентифікатор кластера, назву простору імен та мережеві адреси для доступу до API й веб-інтерфейсу платформи. Це підтвердило коректність реалізації інфраструктури та готовність системи до подальших експериментів і навантажувального тестування.

```
pulumi.export("cluster_name", cluster.name)
pulumi.export("namespace", namespace.name)
pulumi.export("web_url", alb.dns_name.apply(lambda dns: f"http://{dns}"))
pulumi.export("api_url", alb.dns_name.apply(lambda dns: f"http://{dns}:8080"))
```

### 2.3 Функціонування мікросервісів платформи

Функціонування розробленої платформи базується на принципах слабкої зв'язаності та чіткого розмежування відповідальності між мікросервісами. Кожен сервіс реалізує окрему бізнес-функцію та взаємодіє з іншими компонентами системи виключно через стандартизовані інтерфейси, що забезпечує незалежність їх життєвих циклів та спрощує подальший розвиток платформи.

Взаємодія між клієнтським рівнем і серверними компонентами здійснюється через мікросервіс `Api_gateway`, який виконує роль єдиної точки входу до системи. Такий підхід дозволяє централізувати автентифікацію запитів, контроль доступу та маршрутизацію, а також мінімізувати прямі залежності між клієнтами та внутрішніми сервісами платформи. Усі запити від `web_ui` до мікросервісів `User_service`, `Product_service`, `Billing_service` та `Payment_service` проходять через `API Gateway`, що відповідає сучасним рекомендаціям щодо побудови мікросервісних систем.

Для забезпечення узгодженості даних і коректної роботи бізнес-процесів застосовано чітке розмежування зон відповідальності сервісів. Зокрема, платіжні операції обробляються виключно мікросервісом `Payment_service`, тоді як внутрішній облік фінансових транзакцій та балансів користувачів реалізовано в `Billing_service`. Такий підхід знижує ризики порушення цілісності фінансових даних і підвищує надійність системи в цілому.

Архітектура платформи також враховує вимоги до масштабованості та відмовостійкості. Завдяки ізоляції мікросервісів можливе незалежне масштабування окремих компонентів залежно від характеру навантаження, а відмова одного сервісу не призводить до повного припинення роботи

системи. Це особливо важливо для високонавантажених платформ електронної комерції, де безперервна доступність є критичною вимогою.

Таким чином, організація взаємодії між мікросервісами, використання API Gateway та чітке розмежування бізнес-логіки забезпечують цілісність, гнучкість і масштабованість розробленої платформи, що підтверджує відповідність реалізованого рішення сучасним підходам до проектування хмарних мікросервісних систем.

### 2.3.1 Мікросервіс User\_service


Мікросервіс User\_service відповідає за управління користувачами платформи та реалізацію основних процесів автентифікації й авторизації. До основних функціональних можливостей даного сервісу належать:

- реєстрація нових користувачів платформи;
- автентифікація та авторизація з використанням JWT-токенів;
- зберігання та обробка персональних даних користувачів.

Застосування токенів JSON Web Token (JWT) дозволяє реалізувати безпечний і масштабований механізм авторизації без збереження стану сесій на сервері, що є важливою вимогою для розподілених систем.

На рисунку 2.2 представлено веб-інтерфейс, призначений для взаємодії користувачів з мікросервісом User\_service, який забезпечує виконання операцій реєстрації та входу в систему.

#### Реєстрація / Авторизація



The screenshot shows a web form with two input fields. The first field contains the email address 'Leon.vl777@gmail.com'. The second field contains three dots '...'. To the right of these fields are three blue buttons: 'Реєстрація' (Registration), 'Увійти' (Login), and 'Вийти' (Logout).

Користувач: Leon.vl777@gmail.com (Авторизовано)

Рисунок 2.2 – Веб-інтерфейс взаємодії користувача з мікросервісом User\_service

На рисунку 2.3 наведено приклад взаємодії з даним сервісом через API Gateway, що демонструє використання єдиної точки доступу до сервісів платформи.

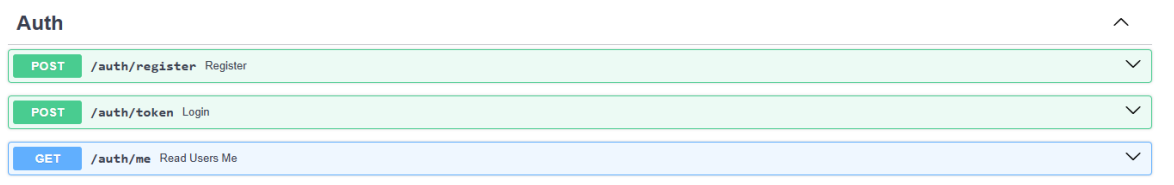


Рисунок 2.3 – Взаємодія користувача з мікросервісом User\_service через API Gateway

### 2.3.2 Мікросервіс Payment\_service

Мікросервіс Payment\_service відповідає за реалізацію процесів поповнення балансу користувачів шляхом інтеграції з зовнішніми платіжними системами. Основним призначенням сервісу є обробка платіжних запитів та взаємодія з зовнішніми джерелами фінансових транзакцій.

Даний сервіс ізольований від інших компонентів системи, що дозволяє змінювати або розширювати платіжну логіку без впливу на внутрішні бізнес-процеси платформи. Такий підхід підвищує безпеку та спрощує подальшу інтеграцію з новими платіжними провайдерами.

### 2.3.3 Мікросервіс Billing\_service

Мікросервіс Billing\_service призначений для реалізації внутрішнього білінгу платформи. Він відповідає за облік фінансових операцій, управління балансами користувачів та взаєморозрахунки між компонентами системи.

Функціональність даного сервісу охоплює фіксацію всіх фінансових подій, що відбуваються в межах платформи, а також забезпечення узгодженості даних між платіжним сервісом і іншими бізнес-компонентами. Відокремлення білінгової логіки в окремий мікросервіс підвищує надійність і прозорість фінансових операцій.

На рисунку 2.4 представлено веб-інтерфейс для взаємодії з мікросервісом Billing\_service.

Профіль та баланс оновлені.

**Профіль та Баланс**

Email: **Leon.vl777@gmail.com**

Баланс: **100,00 USD USD**

**Оновити**

Рисунок 2.4 – Веб-інтерфейс для взаємодії коистувача з мікросервісом Billing\_service

На рисунку 2.5 наведено приклад взаємодії з даним сервісом через API Gateway, що ілюструє централізований підхід до маршрутизації запитів.

**Billing** ^

**POST** /billing/deposit Deposit ▾

**GET** /billing/account/{user\_id} Get Account Balance ▾

Рисунок 2.5 – Взаємодія коистувача з мікросервісом Billing\_service через API Gateway

### 2.3.4 Мікросервіс Product\_service

Мікросервіс Product\_service забезпечує управління товарами платформи та відповідає за зберігання і обробку інформації про продукти, їх характеристики та доступність.

Даний сервіс реалізує бізнес-логіку, пов'язану з каталогом товарів, і надає інтерфейси для отримання, оновлення та адміністрування даних про продукти. Винесення цієї функціональності в окремий мікросервіс дозволяє незалежно масштабувати та розвивати підсистему управління товарами.

На рисунку 2.6 представлено веб-інтерфейс для взаємодії з мікросервісом Product\_service.

Створити тестовий товар

Laptop

Ціна

Кількість

Створити товар

Купівля товару

ID товару

1

Придбати

Каталог товарів

Оновити каталог

Назва	Ціна	В наявності
TestItem-34588	500.00	1
Monitor	1000.00	29
TestItem-10388	500.00	1
TestItem-59463	500.00	1

Рисунок 2.6 – Веб-інтерфейс для взаємодії користувача з мікросервісом Product\_service

На рисунку 2.7 наведено API Gateway, через який здійснюється доступ до функціональних можливостей даного сервісу.

Shop

POST /purchase Purchase Product

GET /products Get Products List

POST /products Create Product

Рисунок 2.7 – Взаємодія користувача з мікросервісом Product\_service через API Gateway

### 2.3.5 Мікросервіс Api\_gateway

Мікросервіс Api\_gateway виконує роль координаційного компонента та є єдиною точкою входу до платформи. Він відповідає за маршрутизацію

клієнтських запитів до відповідних мікросервісів, а також реалізує базові кроссервісні функції, зокрема автентифікацію, логування та контроль доступу.

Використання API Gateway дозволяє приховати внутрішню структуру системи від клієнтів, спростити керування доступом і забезпечити централізований контроль над зовнішніми викликами. Такий підхід підвищує безпеку, масштабованість і зручність супроводу платформи.

### **2.3.6. Мікросервіс Web\_ui**

Компонент web\_ui є клієнтським веб-інтерфейсом платформи та призначений для забезпечення взаємодії між системою й кінцевими користувачами, а також адміністраторами. Даний компонент виконує роль презентаційного рівня архітектури та забезпечує доступ до основних функціональних можливостей платформи електронної комерції.

Веб-інтерфейс реалізує користувацькі сценарії, пов'язані з переглядом каталогу товарів, вибором продуктів, ініціацією процесу придбання, а також взаємодією з користувацьким профілем. Для адміністративних користувачів web\_ui надає інструменти для керування товарами, оновлення інформації про продукти та контролю стану платформи. Таким чином, компонент підтримує розмежування ролей і забезпечує доступ до функціональності відповідно до прав користувачів.

Компонент web\_ui не містить бізнес-логіки системи, а взаємодіє з серверною частиною платформи виключно через API Gateway. Такий підхід відповідає принципам багаторівневої архітектури та дозволяє ізолювати клієнтський інтерфейс від внутрішньої реалізації мікросервісів, що підвищує гнучкість і безпеку системи.

На рисунку 2.8 представлено загальний вигляд веб-інтерфейсу платформи, який демонструє основні елементи навігації та структуру користувацького інтерфейсу.

## 2.4 Навантажувальне тестування платформи

З метою оцінки продуктивності та масштабованості розробленої мікросервісної платформи було проведено навантажувальне тестування. Основною метою даного етапу експериментального дослідження є перевірка здатності системи стабільно обробляти зростаючу кількість користувацьких запитів та підтвердження ефективності горизонтального масштабування мікросервісів.

Профіль та баланс оновлені.

Профіль та Баланс

Email: Leon.vl777@gmail.com

Баланс: 100,00 USD USD

Оновити

Реєстрація / Авторизація

Leon.vl777@gmail.com

...

Реєстрація

Увійти

Вийти

Користувач: Leon.vl777@gmail.com (Авторизовано)

Дії з гаманцем та товарами

Поповнення гаманця

100

Поповнити

Створити тестовий товар

Лартор

Ціна

Кількість

Створити товар

Купівля товару

ID товару

1

Придбати

Каталог товарів

Оновити каталог

Назва	Ціна	В наявності
TestItem-34588	500.00	1
Monitor	1000.00	29
TestItem-10388	500.00	1
TestItem-59463	500.00	1
TestItem-98598	500.00	1

Рисунок 2.8 – Загальний вигляд веб-інтерфейсу мікросервісу web\_ui



Як тестовий сценарій було обрано процес реєстрації користувачів, оскільки він є типовою операцією для систем електронної комерції та створює навантаження як на прикладну логіку, так і на підсистему зберігання даних. На першому етапі тестування було ініційовано одночасну реєстрацію до 360 користувачів на хвилину за умови використання лише однієї репліки мікросервісу User\_service. У результаті проведеного експерименту було встановлено, що стабільна робота сервісу забезпечується при навантаженні до приблизно 180 реєстрацій на хвилину. Подальше збільшення інтенсивності запитів призводило до зростання часу обробки та зниження стабільності роботи сервісу, що зафіксовано на рисунку 2.9.

## Charts

Total Requests per Second

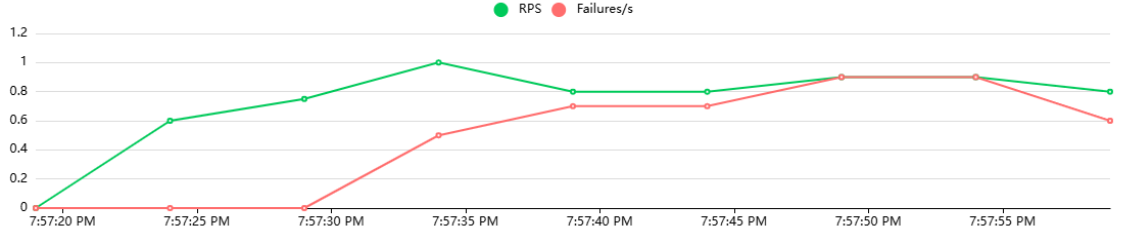


Рисунок 2.9 – Результати тестування процес реєстрації користувачів на мікросервісі User\_service

На наступному етапі кількість реплік мікросервісу реєстрації було збільшено до трьох, що відповідає принципам горизонтального масштабування, характерним для мікросервісної архітектури. Схему оновленої конфігурації наведено на рисунку 2.10. Після цього навантажувальний тест було повторено з інтенсивністю 360 реєстрацій користувачів на хвилину.

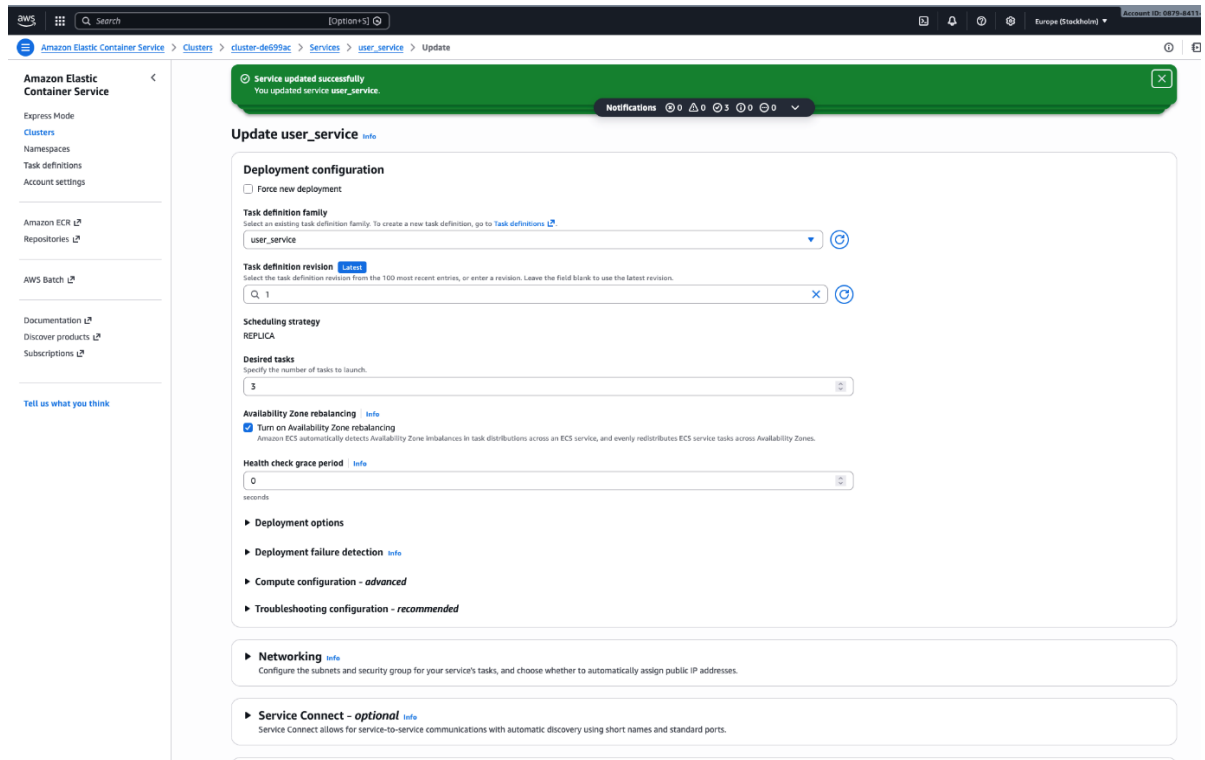


Рисунок 2.10 – Повторне тестування реєстрації користувачів на мікросервісі User\_service

Результати тестування показали, що за умов використання трьох реплік сервісу платформа стабільно обробляє зазначене навантаження без деградації продуктивності та помітного збільшення часу відповіді. Це підтверджує ефективність масштабування мікросервісів і здатність системи адаптуватися до зростання навантаження. Відповідні результати наведено на рисунку 2.11.

## Charts

### Total Requests per Second

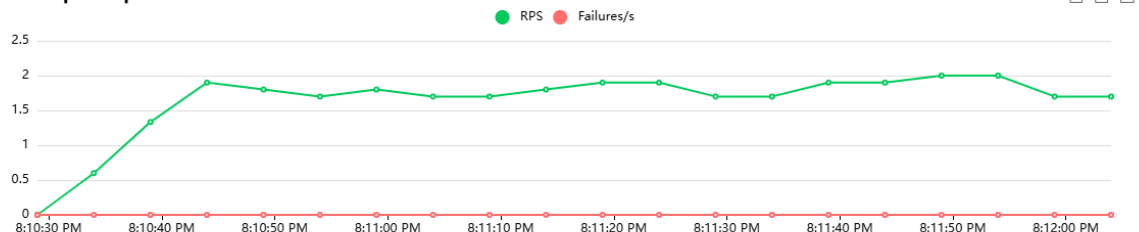


Рисунок 2.11 – Результати повторного тестування реєстрації користувачів на мікросервісі User\_service

## 2.5. Тестування відмовостійкості платформи

Однією з ключових переваг мікросервісної архітектури є підвищена відмовостійкість системи, яка досягається за рахунок ізоляції окремих компонентів. Для перевірки даної властивості було проведено серію експериментів, спрямованих на моделювання відмов окремих мікросервісів платформи та аналіз поведінки системи в таких умовах.

У першому експерименті було свідомо відключено мікросервіс `Payment_service`, імітуючи його повну недоступність. Відповідний стан системи наведено на рисунку 2.12. Метою даного експерименту було перевірити, чи призводить відмова платіжного сервісу до повного припинення роботи платформи.

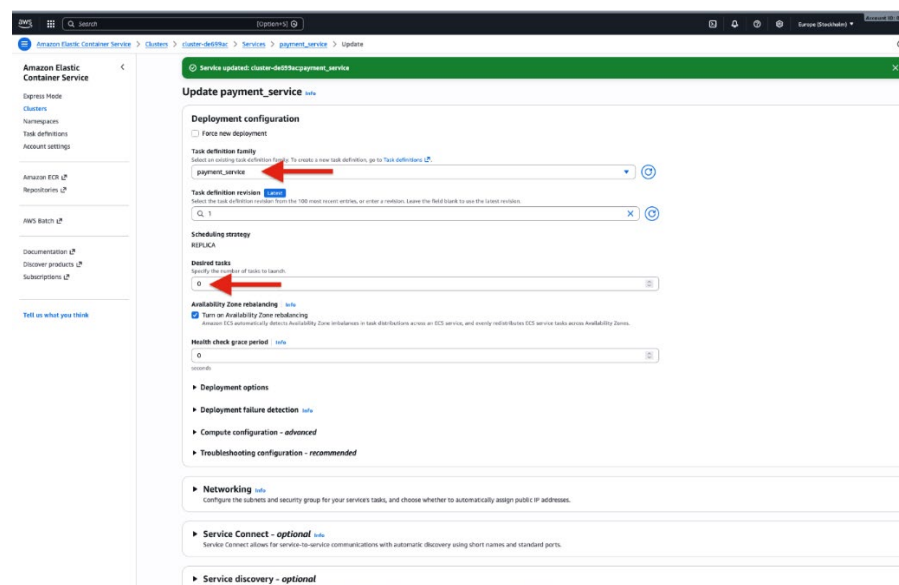


Рисунок 2.12 – Тестування відмовостійкості на мікросервісі `Payment_service`

Результати тестування показали, що за відсутності `Payment_service` основні функції платформи, зокрема автентифікація користувачів, перегляд товарів та виконання операцій, не пов'язаних безпосередньо з оплатою, продовжують працювати коректно. Як показано на рисунку 2.13, система

зберігає працездатність, а відмова одного сервісу не спричиняє каскадного збою всієї платформи.

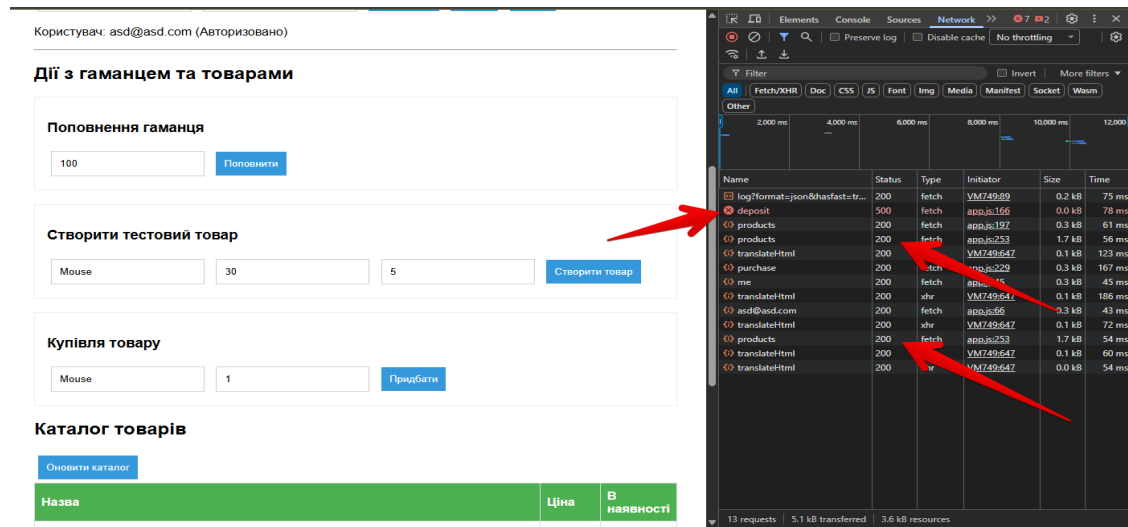


Рисунок 2.13 – Результати тестування відмовостійкості на мікросервісі Payment\_service

У другому експерименті було імітовано відмову мікросервісу Product\_service, що відповідає за управління товарами. Стан системи в умовах відключення даного сервісу представлено на рисунку 2.14.

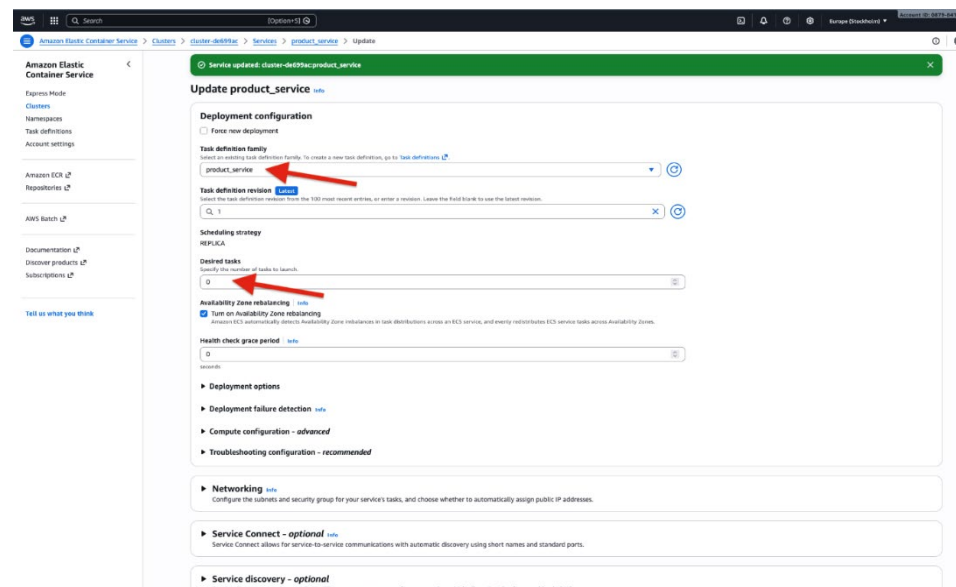


Рисунок 2.14 – Тестування відмовостійкості мікросервісу Product\_service

Аналіз поведінки платформи показав, що функції реєстрації користувачів та поповнення балансу залишаються доступними, незважаючи на недоступність сервісу управління товарами. Це підтверджується результатами, наведеними на рисунку 2.15.

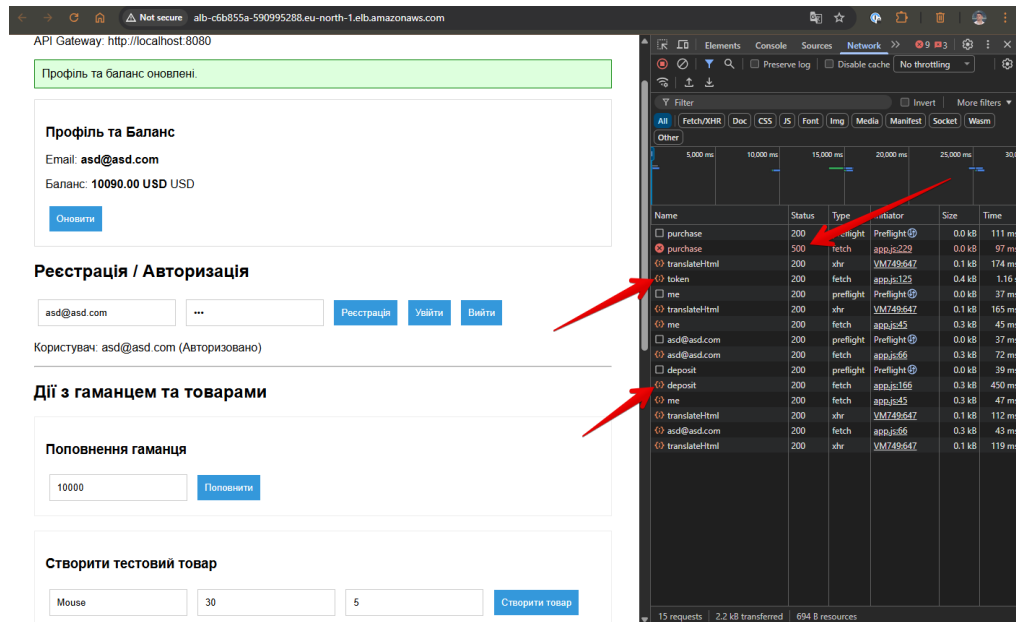


Рисунок 2.15 – Результати тестування відмовостійкості мікросервісу Product\_service

Отримані результати експериментів свідчать про те, що розроблена платформа відповідає принципам відмовостійкості та резилієнтності. Відмова окремого мікросервісу не призводить до повного припинення роботи системи, а функціональність, не пов'язана з недоступним компонентом, зберігається. Це підтверджує доцільність використання мікросервісної архітектури для побудови надійних високонавантажених систем.

## **Висновки до розділу 2**

За результатами проведеного навантажувального тестування та тестування відмовостійкості було підтверджено практичну ефективність обраної мікросервісної архітектури та доцільність її використання для побудови високонавантажених систем у хмарних середовищах.

Навантажувальне тестування продемонструвало, що система здатна стабільно обробляти зростаючу кількість користувацьких запитів за умови застосування горизонтального масштабування окремих мікросервісів. Збільшення кількості реплік сервісу реєстрації дозволило пропорційно підвищити пропускну здатність платформи без деградації продуктивності, що підтверджує ефективність використання хмарної інфраструктури та контейнерної оркестрації.

Результати тестування відмовостійкості засвідчили, що відмова окремих мікросервісів не призводить до повного припинення роботи платформи. Ізоляція компонентів забезпечує збереження працездатності функціональних підсистем, не пов'язаних безпосередньо з недоступним сервісом, що відповідає принципам резилієнтності та підвищує загальну надійність системи.

Таким чином, отримані експериментальні результати підтверджують, що мікросервісна архітектура у поєднанні з хмарною інфраструктурою Amazon Web Services забезпечує масштабованість, стабільність та відмовостійкість платформи, а також створює ефективну архітектурну основу для подальшого розвитку високонавантажених програмних систем.

## ВИСНОВКИ

У магістерській роботі досліджено сучасні методології проектування високонавантажених програмних систем із використанням мікросервісної архітектури в хмарних середовищах. У ході виконання роботи було розглянуто еволюцію архітектурних підходів до побудови розподілених систем, зокрема монолітну, клієнт–серверну, сервіс-орієнтовану, serverless та подійно-орієнтовану архітектури, а також проведено їх порівняльний аналіз за ключовими характеристиками масштабованості, надійності та супроводжуваності.

За результатами теоретичного аналізу встановлено, що мікросервісна архітектура є найбільш доцільним та збалансованим підходом для побудови сучасних високонавантажених систем у хмарних інфраструктурах. Вона забезпечує незалежність компонентів, гнучке горизонтальне масштабування, ізольоване розгортання сервісів та підвищену відмовостійкість, що є критично важливим для систем, орієнтованих на безперервну роботу та обслуговування великої кількості користувачів.

У практичній частині роботи було спроектовано та реалізовано мікросервісну платформу електронної комерції, розгорнуту в хмарному середовищі Amazon Web Services. Архітектура платформи базується на принципах мікросервісного підходу з чітким розмежуванням бізнес-функцій між окремими сервісами, використанням API Gateway як єдиної точки входу та застосуванням сучасних механізмів контейнеризації та оркестрації. Розгортання інфраструктури здійснювалося із застосуванням методології Infrastructure as Code, що забезпечило відтворюваність, керованість і масштабованість інфраструктурного середовища.

Проведене навантажувальне тестування підтвердило здатність розробленої платформи ефективно обробляти зростаюче навантаження шляхом горизонтального масштабування окремих мікросервісів. Збільшення кількості реплік сервісів дозволило пропорційно підвищити пропускну

здатність системи без деградації продуктивності, що свідчить про ефективність обраної архітектури та хмарної інфраструктури.

Результати тестування відмовостійкості продемонстрували, що відмова окремих мікросервісів не призводить до повного припинення роботи платформи. Ізоляція компонентів та відсутність жорстких залежностей між ними забезпечують збереження працездатності основного функціоналу системи, що відповідає принципам резилієнтності та підвищує загальний рівень надійності програмного рішення.

Таким чином, результати теоретичного аналізу та практичної реалізації підтверджують, що використання мікросервісної архітектури в поєднанні з хмарними технологіями Amazon Web Services є ефективним підходом до проєктування високонавантажених програмних систем. Запропоноване рішення забезпечує масштабованість, відмовостійкість і зручність супроводу, а також створює надійну архітектурну основу для подальшого розвитку та модернізації сучасних розподілених систем.



## СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Shelmanov D. *Analysis and Design of High-Load Systems Based on Microservice Architecture*. DOI: 10.21694/2572-2921.24015. URL: <https://arjonline.org/papers/arjcsit/v7-i1/15.pdf> (дата звернення: 11.11.2025).
2. Badwaik N. *Designing System Architecture with High Availability and Scalability*. DOI: 10.21694/2572-2921.24002. URL: <https://arjonline.org/papers/arjcsit/v7-i1/2.pdf> (дата звернення: 11.11.2025).
3. Kleppmann M. *Designing Data-Intensive Applications*. Sebastopol : O'Reilly Media, 2017. 616 с.
4. Heimerdinger W. L., Weinstock C. B. *A Conceptual Framework for System Fault Tolerance*. Pittsburgh : Software Engineering Institute, Carnegie Mellon University, 1992.
5. Yuan D., Luo Y., Zhuang X. et al. *Simple Testing Can Prevent Most Critical Failures*. Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation. Broomfield, 2014. С. 249–265.
6. Izrailevsky Y., Tseitlin A. *The Netflix Simian Army*. Netflix TechBlog, 2011.
7. Fowler M. *Patterns of Enterprise Application Architecture*. Boston : Addison-Wesley, 2003. 533 с.
8. Newman S. *Building Microservices: Designing Fine-Grained Systems*. Sebastopol : O'Reilly Media, 2015. 280 с.
9. Monolithic Application Architecture. Microsoft Learn. URL: <https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/monolithic> (дата звернення: 11.11.2025).
10. Fowler M. *Monolith First*. Martinfowler.com. URL: <https://martinfowler.com/bliki/MonolithFirst.html> (дата звернення: 11.11.2025).
11. Monolithic vs Microservices. Red Hat. URL: <https://www.redhat.com/en/topics/microservices/monolithic-vs-microservices> (дата звернення: 11.11.2025).

12. Newman S. *Monolith to Microservices*. Sebastopol : O'Reilly Media, 2019. 272 с.
13. What is Monolithic Architecture? IBM Developer. URL: <https://developer.ibm.com/articles/what-is-monolithic-architecture/> (дата звернення: 11.11.2025).
14. Tanenbaum A. S., Van Steen M. *Distributed Systems: Principles and Paradigms*. Upper Saddle River : Pearson, 2017. 749 с.
15. Coulouris G., Dollimore J., Kindberg T., Blair G. *Distributed Systems: Concepts and Design*. Boston : Addison-Wesley, 2012. 1080 с.
16. Pressman R. S., Maxim B. R. *Software Engineering: A Practitioner's Approach*. New York : McGraw-Hill Education, 2014. 976 с.
17. Client-Server Architecture. Microsoft Learn. URL: <https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/client-server> (дата звернення: 11.11.2025).
18. What is a Client-Server Architecture? Red Hat. URL: <https://www.redhat.com/en/topics/cloud-native-apps/what-is-client-server> (дата звернення: 11.11.2025).
19. Client-Server Model Overview. Oracle. URL: <https://docs.oracle.com/javase/tutorial/networking/overview/clientServer.html> (дата звернення: 11.11.2025).
20. Cisco Systems. Client-Server Network Model. Indianapolis : Cisco Press, 2020.
21. Client-Server vs Peer-to-Peer Architecture. IBM Developer. URL: <https://developer.ibm.com/articles/client-server-vs-peer-to-peer/> (дата звернення: 11.11.2025).
22. Architectural Patterns: Client-Server Applications. Amazon Web Services. URL: <https://aws.amazon.com/architecture/client-server/> (дата звернення: 11.11.2025).

23. Fowler M., Lewis J. *Microservices: a Definition of This New Architectural Term*. URL: <https://martinfowler.com/articles/microservices.html> (дата звернення: 11.11.2025).

24. Dragoni N., Giallorenzo S., Lafuente A. L., Mazzara M. *Microservices: Yesterday, Today, and Tomorrow*. Cham : Springer, 2017. С. 195–216.

25. Microservices Architecture Style. Microsoft Learn. URL: <https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices> (дата звернення: 11.11.2025).

26. What are Microservices? Red Hat. URL: <https://www.redhat.com/en/topics/microservices/what-are-microservices> (дата звернення: 11.11.2025).

27. Microservices on AWS. Amazon Web Services. URL: <https://aws.amazon.com/microservices/> (дата звернення: 11.11.2025).

28. Microservices Architecture Explained. IBM Developer. URL: <https://developer.ibm.com/articles/microservices-architecture/> (дата звернення: 11.11.2025).

29. Reference Model for Service-Oriented Architecture 1.0 (SOA-RM). OASIS. URL: <https://docs.oasis-open.org/soa-rm/v1.0/> (дата звернення: 11.11.2025).

30. Web Services Architecture. W3C. URL: <https://www.w3.org/TR/ws-arch/> (дата звернення: 11.11.2025).

31. Chappell D. A. *Enterprise Service Bus*. Sebastopol : O'Reilly Media, 2004. 259 с.

32. Service-Oriented Architecture: A Reference Architecture. The Open Group. URL: <https://publications.opengroup.org/g071> (дата звернення: 11.11.2025).

33. Luckham D. *The Power of Events*. Boston : Addison-Wesley, 2002. 376 с.

34. Hohpe G., Woolf B. *Enterprise Integration Patterns*. Boston : Addison-Wesley, 2003. 736 с.

35. Event-Driven Architecture (EDA) Definition. Gartner. URL: <https://www.gartner.com/en/information-technology/glossary/event-driven-architecture> (дата звернення: 11.11.2025).
36. Apache Kafka Documentation. Apache Software Foundation. URL: <https://kafka.apache.org/documentation/> (дата звернення: 11.11.2025).
37. Event-Driven Architecture Explained. Confluent. URL: <https://www.confluent.io/learn/event-driven-architecture/> (дата звернення: 11.11.2025).
38. Event-Driven Architecture Overview. Oracle. URL: <https://docs.oracle.com/en/solutions/event-driven-architecture/> (дата звернення: 11.11.2025).
39. Understanding Event-Driven Architecture. Red Hat. URL: <https://www.redhat.com/en/topics/integration/what-is-event-driven-architecture> (дата звернення: 11.11.2025).
40. Papazoglou M. P., van den Heuvel W.-J. *Service-Oriented Design and Development Methodology*. International Journal of Web Engineering and Technology. 2006. Vol. 2, no. 4. С. 412–442.
41. Jonas E., Schleier-Smith J., Sreekanti V. *Cloud Programming Simplified: A Berkeley View on Serverless Computing*. Berkeley : University of California, Berkeley, 2019.
42. Baldini I., Castro P., Chang K., Cheng P. *Serverless Computing: Current Trends and Open Problems*. Cham : Springer, 2017.
43. Serverless Computing. Amazon Web Services. URL: <https://aws.amazon.com/serverless/> (дата звернення: 11.11.2025).
44. Cloud Functions Overview. Google Cloud. URL: <https://cloud.google.com/functions/docs> (дата звернення: 11.11.2025).
45. What is Serverless Computing? IBM Cloud. URL: <https://www.ibm.com/cloud/learn/serverless> (дата звернення: 11.11.2025).

46. Serverless Architecture Radar Report. ThoughtWorks. URL: <https://www.thoughtworks.com/radar/techniques/serverless-architecture> (дата звернення: 11.11.2025).

47. CNCF Serverless Whitepaper v1.0. Cloud Native Computing Foundation. URL: [https://www.cncf.io/wp-content/uploads/2018/11/CNCF\\_Serverless\\_Whitepaper.pdf](https://www.cncf.io/wp-content/uploads/2018/11/CNCF_Serverless_Whitepaper.pdf) (дата звернення: 11.11.2025).

48. Luckham D. *The Power of Events*. Boston : Addison-Wesley, 2002. 376 с.

49. Hohpe G., Woolf B. *Enterprise Integration Patterns*. Boston : Addison-Wesley, 2003. 736 с.

50. Event-Driven Architecture (EDA) Definition. Gartner. URL: <https://www.gartner.com/en/information-technology/glossary/event-driven-architecture> (дата звернення: 11.11.2025).

51. Apache Kafka Documentation. Apache Software Foundation. URL: <https://kafka.apache.org/documentation/> (дата звернення: 11.11.2025).

52. Event-Driven Architecture Explained. Confluent. URL: <https://www.confluent.io/learn/event-driven-architecture/> (дата звернення: 11.11.2025).

53. Event-Driven Architecture Overview. Oracle. URL: <https://docs.oracle.com/en/solutions/event-driven-architecture/> (дата звернення: 11.11.2025).

54. Understanding Event-Driven Architecture. Red Hat. URL: <https://www.redhat.com/en/topics/integration/what-is-event-driven-architecture> (дата звернення: 11.11.2025).

55. Event-Driven Systems and Stream Processing: Trends and Challenges. IEEE Internet Computing. 2021.

56. Event-Driven Architecture for Distributed Systems. National Institute of Standards and Technology, 2020.

57. Newman S. *Building Microservices: Designing Fine-Grained Systems*. 2nd ed. Sebastopol : O'Reilly Media, 2021. 582 с.
58. Fowler M., Lewis J. *Microservices: a Definition of This New Architectural Term*. URL: <https://martinfowler.com/articles/microservices.html> (дата звернення: 11.11.2025).
59. Richards M. *Software Architecture Patterns*. Sebastopol : O'Reilly Media, 2015. 162 с.
60. Evans E. *Domain-Driven Design*. Boston : Addison-Wesley, 2004. 560 с.
61. Kleppmann M. *Designing Data-Intensive Applications*. Sebastopol : O'Reilly Media, 2017. 616 с.
62. Richardson C. *Microservices Patterns*. New York : Manning Publications, 2019. 520 с.
63. Payments architecture overview. Stripe Engineering. URL: <https://stripe.com/docs> (дата звернення: 11.11.2025).
64. Hardt D. *The OAuth 2.0 Authorization Framework: RFC 6749*. Internet Engineering Task Force (IETF), 2012.
65. API Gateway architecture. NGINX. URL: <https://www.nginx.com/learn/api-gateway> (дата звернення: 11.11.2025).
66. Fielding R. T. *Architectural Styles and the Design of Network-based Software Architectures*. Irvine : University of California, 2000.
67. AWS Well-Architected Framework. Amazon Web Services. URL: <https://docs.aws.amazon.com/wellarchitected> (дата звернення: 11.11.2025).
68. Burns B., Beda J., Hightower K. *Kubernetes: Up and Running*. 2nd ed. Sebastopol : O'Reilly Media, 2019. 278 с.

**Додаток А. Вихідний код додатку**

```

import pulumi
import pulumi_aws as aws
import json

account_id = aws.get_caller_identity().account_id
region = "eu-north-1"

services_config = {
    "user_service": {
        "port": 8000,
        "replicas": 1,
        "env": {
            "AWS_REGION": "*****",
            "AWS_ACCESS_KEY_ID": "*****",
            "AWS_SECRET_ACCESS_KEY": "*****",
            "USER_TABLE_NAME": "*****",
            "SECRET_KEY": "*****",
        },
    },
    "product_service": {
        "port": 8000,
        "replicas": 1,
        "env": {
            "AWS_REGION": "*****",
            "AWS_ACCESS_KEY_ID": "*****",
            "AWS_SECRET_ACCESS_KEY": "*****",
            "PRODUCT_TABLE_NAME": "*****",
            "INVENTORY_TABLE_NAME": "*****",
        },
    },
}

```

```

    "HISTORY_TABLE_NAME": "*****",
  },
},
"billing_service": {
  "port": 8000,
  "replicas": 1,
  "env": {
    "AWS_REGION": "*****",
    "AWS_ACCESS_KEY_ID": "*****",
    "AWS_SECRET_ACCESS_KEY": "*****",
    "ACCOUNT_TABLE_NAME": "*****",
    "TRANSACTION_TABLE_NAME": "*****",
    "PAYMENT_SERVICE_URL": "*****",
  },
},
"payment_service": {
  "port": 8000,
  "replicas": 1,
  "env": {},
},
"api_gateway": {
  "port": 8000,
  "replicas": 1,
  "env": {
    "USER_SERVICE_URL": "*****",
    "PRODUCT_SERVICE_URL": "*****",
    "BILLING_SERVICE_URL": "*****",
  },
},
"web_ui": {

```



```

    "port": 80,
    "replicas": 1,
    "env": {
        "API_URL": "*****",
    },
},
}

```

```
cluster = aws.ecs.Cluster("cluster")
```

```

namespace = aws.servicediscovery.PrivateDnsNamespace(
    "local",
    name="local",
    vpc=aws.ec2.get_vpc(default=True).id,
)

```

```

execution_role = aws.iam.Role(
    "execution-role",
    assume_role_policy=json.dumps({
        "Version": "2012-10-17",
        "Statement": [{
            "Action": "sts:AssumeRole",
            "Principal": {"Service": "ecs-tasks.amazonaws.com"},
            "Effect": "Allow"
        }]
    })
)

```

```

aws.iam.RolePolicyAttachment(
    "execution-role-policy",

```

```

        role=execution_role.name,
        policy_arn="arn:aws:iam::aws:policy/service-
role/AmazonECSTaskExecutionRolePolicy"
    )

```

```

aws.iam.RolePolicy(
    "execution-role-logs",
    role=execution_role.name,
    policy=json.dumps({
        "Version": "2012-10-17",
        "Statement": [{
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:PutLogEvents"
            ],
            "Resource": "*"
        }]
    })
)

```

```

default_vpc = aws.ec2.get_vpc(default=True)

```

```

default_subnets = aws.ec2.get_subnets(
    filters=[aws.ec2.GetSubnetsFilterArgs(name="vpc-id",
values=[default_vpc.id])]
)

```

```

sg = aws.ec2.SecurityGroup(

```

```

"app-sg",
vpc_id=default_vpc.id,
ingress=[aws.ec2.SecurityGroupIngressArgs(
    protocol="-1",
    from_port=0,
    to_port=0,
    cidr_blocks=["0.0.0.0/0"],
)],
egress=[aws.ec2.SecurityGroupEgressArgs(
    protocol="-1",
    from_port=0,
    to_port=0,
    cidr_blocks=["0.0.0.0/0"],
)],
)

```

```

alb_sg = aws.ec2.SecurityGroup(
    "alb-sg",
    vpc_id=default_vpc.id,
    ingress=[
        aws.ec2.SecurityGroupIngressArgs(protocol="tcp",    from_port=80,
to_port=80, cidr_blocks=["0.0.0.0/0"]),
        aws.ec2.SecurityGroupIngressArgs(protocol="tcp",    from_port=443,
to_port=443, cidr_blocks=["0.0.0.0/0"]),
        aws.ec2.SecurityGroupIngressArgs(protocol="tcp",    from_port=8080,
to_port=8080, cidr_blocks=["0.0.0.0/0"]),
    ],
    egress=[aws.ec2.SecurityGroupEgressArgs(
        protocol="-1",
        from_port=0,

```

```

        to_port=0,
        cidr_blocks=["0.0.0.0/0"],
    )],
)

```

```

alb = aws.lb.LoadBalancer(
    "alb",
    load_balancer_type="application",
    subnets=default_subnets.ids,
    security_groups=[alb_sg.id],
)

```

```

api_tg = aws.lb.TargetGroup(
    "api-tg",
    port=8000,
    protocol="HTTP",
    target_type="ip",
    vpc_id=default_vpc.id,
    health_check=aws.lb.TargetGroupHealthCheckArgs(
        path="/",
        interval=30,
        timeout=5,
        healthy_threshold=2,
        unhealthy_threshold=2,
        matcher="200,404",
    ),
)

```

```

web_tg = aws.lb.TargetGroup(
    "web-tg",

```

```

    port=80,
    protocol="HTTP",
    target_type="ip",
    vpc_id=default_vpc.id,
    health_check=aws.lb.TargetGroupHealthCheckArgs(
        path="/",
        interval=30,
        timeout=5,
        healthy_threshold=2,
        unhealthy_threshold=2,
    ),
)

```

```

listener = aws.lb.Listener(
    "listener",
    load_balancer_arn=alb.arn,
    port=80,
    protocol="HTTP",
    default_actions=[aws.lb.ListenerDefaultActionArgs(
        type="forward",
        target_group_arn=web_tg.arn,
    )],
)

```

```

api_listener = aws.lb.Listener(
    "api-listener",
    load_balancer_arn=alb.arn,
    port=8080,
    protocol="HTTP",
    default_actions=[aws.lb.ListenerDefaultActionArgs(

```

```

        type="forward",
        target_group_arn=api_tg.arn,
    )],
)

```

```

for service_name, config in services_config.items():
    ecr_url =
    f"{account_id}.dkr.ecr.{region}.amazonaws.com/{service_name}:latest"

```

```

    container_def = {
        "name": service_name,
        "image": ecr_url,
        "essential": True,
        "portMappings": [{
            "containerPort": config["port"],
            "protocol": "tcp"
        }],
        "environment": [{"name": k, "value": v} for k, v in
config["env"].items()],
        "logConfiguration": {
            "logDriver": "awslogs",
            "options": {
                "awslogs-group": f"/ecs/{service_name}",
                "awslogs-region": region,
                "awslogs-stream-prefix": "ecs",
                "awslogs-create-group": "true"
            }
        }
    }
}

```

```

task_def = aws.ecs.TaskDefinition(
    f'{service_name}-task',
    family=service_name,
    cpu="256",
    memory="512",
    network_mode="awsvpc",
    requires_compatibilities=["FARGATE"],
    execution_role_arn=execution_role.arn,
    container_definitions=json.dumps([container_def])
)

discovery = aws.servicediscovery.Service(
    f'{service_name}-discovery',
    name=service_name,
    dns_config=aws.servicediscovery.ServiceDnsConfigArgs(
        namespace_id=namespace.id,

dns_records=[aws.servicediscovery.ServiceDnsConfigDnsRecordArgs(
    ttl=10,
    type="A",
)],
),
)

lb_config = []
if service_name == "api_gateway":
    lb_config = [aws.ecs.ServiceLoadBalancerArgs(
        target_group_arn=api_tg.arn,
        container_name=service_name,
        container_port=config["port"],

```

```

    )]
elif service_name == "web_ui":
    lb_config = [aws.ecs.ServiceLoadBalancerArgs(
        target_group_arn=web_tg.arn,
        container_name=service_name,
        container_port=config["port"],
    )]

aws.ecs.Service(
    f'{service_name}-service',
    name=service_name,
    cluster=cluster.arn,
    task_definition=task_def.arn,
    desired_count=config["replicas"],
    launch_type="FARGATE",
    network_configuration=aws.ecs.ServiceNetworkConfigurationArgs(
        subnets=default_subnets.ids,
        security_groups=[sg.id],
        assign_public_ip=True,
    ),
    service_registries=aws.ecs.ServiceServiceRegistriesArgs(
        registry_arn=discovery.arn,
    ),
    load_balancers=lb_config,
)

```

```

pulumi.export("cluster_name", cluster.name)
pulumi.export("namespace", namespace.name)
pulumi.export("web_url", alb.dns_name.apply(lambda dns: f"http://{dns}"))

```



```
pulumi.export("api_url", alb.dns_name.apply(lambda dns:
f"http://{dns}:8080"))
```

### **Додаток Б. Вихідний код додатку**

```
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware

from .routers.billing_router import billing_router
from .routers.user_router import user_router
from .routers.shop_router import shop_router

app = FastAPI(title="API Gateway")

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

app.include_router(user_router)
app.include_router(shop_router)
app.include_router(billing_router)

@app.get("/health")
def health_check():
    return {"status": "ok", "service": "api_gateway"}
```

**Додаток В. Вихідний код додатку**

```

from decimal import Decimal

from fastapi import FastAPI, Depends, HTTPException

from .clients.payment_client import PaymentClient, get_payment_client
from .db.billing_repository import BillingRepository, get_billing_repository
from .models import DepositRequest, WithdrawalRequest, AccountItem,
PaymentStatus, TransactionItem


def decimal_default(obj):
    if isinstance(obj, Decimal):
        return str(obj)
    raise TypeError(f"Object of type {type(obj).__name__} is not JSON
serializable")


app = FastAPI(
    title="Billing Service",
    json_encoders={Decimal: decimal_default}
)


# services/billing_service/app/main.py


from decimal import Decimal

from fastapi import FastAPI, Depends, HTTPException

from botocore.exceptions import ClientError # <-- НОВЫЙ ИМПОРТ

```

```

# ... (другие импорты и app = FastAPI)

@app.post("/deposit", response_model=AccountItem)
async def process_deposit(
    request: DepositRequest,
    repo: BillingRepository = Depends(get_billing_repository),
    payment_client: PaymentClient = Depends(get_payment_client)
):
    repo.create_or_get_account(request.user_id)

    pending_transaction = TransactionItem(
        user_id=request.user_id,
        amount=request.amount,
        type="DEPOSIT",
        status="PENDING"
    ).model_dump()
    repo.add_transaction(pending_transaction) # <-- Здесь транзакция
    создается с 'timestamp'

    payment_result = await payment_client.initialize_deposit(request.user_id,
    request.amount)

    payment_status = PaymentStatus(**payment_result)

    final_status = "FAILED"

    if payment_status.status == "COMPLETED":
        # repo.update_balance_atomic() работает успешно!
        if repo.update_balance_atomic(request.user_id, request.amount):
            final_status = "COMPLETED"

```

```

else:
    final_status = "FAILED"
try:
    repo.transaction_table.update_item(
        Key={
            'user_id': request.user_id,
            'timestamp': pending_transaction['timestamp']
        },
        UpdateExpression="SET #s = :status_val, reference_id = :ref_id",
        ExpressionAttributeNames={'#s': 'status'},
        ExpressionAttributeValues={
            ':status_val': final_status,
            ':ref_id': payment_status.reference_id
        }
    )
except ClientError as e:

    print(f"CRITICAL DB ERROR (Transaction Status): {e}")
    raise HTTPException(
        status_code=500,
        detail=f"CRITICAL ERROR: Balance update succeeded, but failed
to log transaction status: {e.response['Error']['Code']}. Check transaction key
schema (user_id + timestamp)."
    )

if final_status == "FAILED":
    raise HTTPException(status_code=500, detail="Deposit failed during
payment or balance update.")

```

```

return repo.get_account(request.user_id)

@app.post("/withdraw", response_model=AccountItem)
async def process_withdrawal(
    request: WithdrawalRequest,
    repo: BillingRepository = Depends(get_billing_repository)
):
    try:
        success = repo.update_balance_atomic(request.user_id, -
request.amount)
    except Exception:
        raise HTTPException(status_code=400, detail="Insufficient funds or
account error.")

    if not success:
        raise HTTPException(status_code=400, detail="Insufficient funds.")

    transaction = TransactionItem(
        user_id=request.user_id,
        amount=request.amount,
        type="WITHDRAWAL",
        status="COMPLETED"
    ).model_dump()
    repo.add_transaction(transaction)

    return repo.get_account(request.user_id)

@app.get("/account/{user_id}", response_model=AccountItem)

```

```
async def get_account(
    user_id: str,
    repo: BillingRepository = Depends(get_billing_repository)
):
    account = repo.get_account(user_id)
    if not account:
        raise HTTPException(status_code=404, detail="Account not found.")
    return account


@app.get("/health")
def health_check():
    return {"status": "ok", "service": "billing_service"}
```

**Додаток Г. Вихідний код додатку**

```

from fastapi import FastAPI, Depends, HTTPException, status
from typing import Dict
import uuid
import random
import time

from .models import PaymentRequest, PaymentResponse

app = FastAPI(title="Payment Service")

@app.post("/process_payment", response_model=PaymentResponse)
async def process_payment(request: PaymentRequest):

    time.sleep(random.uniform(0.1, 0.5))

    reference_id = str(uuid.uuid4())

    return PaymentResponse(
        status="COMPLETED",
        reference_id=reference_id,
        message=f"Payment of {request.amount} {request.currency} processed
successfully."
    )

@app.get("/health")
def health_check():
    return {"status": "ok", "service": "payment_service"}

```

**Додаток Д. Вихідний код додатку**

```

import uuid

from datetime import datetime

from typing import List, Dict


from fastapi import FastAPI, Depends, HTTPException


from .db.product_repository import ProductRepository,
get_product_repository

from .models import ProductCreate, ProductItem, ProductPurchase


app = FastAPI(title="Product Service")


@app.post("/products", response_model=ProductItem)
async def create_product(
    product_data: ProductCreate,
    repo: ProductRepository = Depends(get_product_repository)
):
    product_name = product_data.name
    product_item = ProductItem(
        id=product_name,
        name=product_data.name,
        description=product_data.description,
        price=product_data.price
    )

    repo.create_product(product_item.model_dump(by_alias=True),
product_data.initial_quantity)

```



```
return product_item
```

```
@app.get("/products/{product_id}")
```

```
async def get_product(
```

```
    product_id: str,
```

```
    repo: ProductRepository = Depends(get_product_repository)
```

```
):
```

```
    product = repo.get_product(product_id)
```

```
    inventory = repo.get_inventory(product_id)
```

```
    if not product:
```

```
        raise HTTPException(status_code=404, detail="Product not found")
```

```
    product['quantity'] = inventory['quantity'] if inventory else 0
```

```
    return product
```

```
@app.post("/products/purchase")
```

```
async def process_purchase(
```

```
    purchase_data: ProductPurchase,
```

```
    repo: ProductRepository = Depends(get_product_repository)
```

```
):
```

```
    try:
```

```
        success = repo.decrement_inventory(purchase_data.product_id,
        purchase_data.quantity)
```

```
    except Exception:
```

```
        raise HTTPException(status_code=400, detail="Insufficient stock or
        invalid product.")
```

```

    if not success:
        raise HTTPException(status_code=400, detail="Inventory update
failed.")

```

```

    history_item = {
        'user_id': purchase_data.user_id,
        'transaction_id': str(uuid.uuid4()),
        'product_id': purchase_data.product_id,
        'quantity': purchase_data.quantity,
        'timestamp': datetime.utcnow().isoformat()
    }
    repo.add_purchase_history(history_item)

    return {"status": "success", "message": "Purchase recorded and stock
updated."}

```

```

@app.get("/products", response_model=List[Dict])
async def get_all_products(repo: ProductRepository =
Depends(get_product_repository)):
    return repo.get_all_products_with_inventory()

```

```

@app.get("/health")
def health_check():
    return {"status": "ok", "service": "product_service"}

```

**Додаток Е. Вихідний код додатку**

```

import uuid

from fastapi import FastAPI, Depends, HTTPException, status
from fastapi.security import OAuth2PasswordRequestForm

from .db.user_repository import UserRepository, get_user_repository
from .models import UserRegister, UserResponse, Token, UserInDB
from .security import (
    get_password_hash,
    verify_password,
    create_access_token,
    get_current_user
)

app = FastAPI(title="User Service")

@app.post("/register", response_model=UserResponse)
async def register_user(
    user_data: UserRegister,
    repo: UserRepository = Depends(get_user_repository) # <-- DI
):
    existing_user_item = repo.get_user_by_email(user_data.email)
    if existing_user_item:
        raise HTTPException(status_code=400, detail="Email already
registered")

    hashed_password = get_password_hash(user_data.password)
    new_user_item = {

```

```

        'email': user_data.email,
        'hashed_password': hashed_password,
        'role': 'user',
        'id': str(uuid.uuid4())
    }

    repo.create_user(new_user_item)

    return UserResponse(**new_user_item)

@app.post("/token", response_model=Token)
async def login_for_access_token(
    form_data: OAuth2PasswordRequestForm = Depends(),
    repo: UserRepository = Depends(get_user_repository) # <-- DI
):
    user_item = repo.get_user_by_email(form_data.username)

    if not user_item:
        user = None
    else:
        user = UserInDB(**user_item)

    if not user or not verify_password(form_data.password,
user.hashed_password):
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Incorrect username or password",
            headers={"WWW-Authenticate": "Bearer"},
        )

```

```
access_token = create_access_token(data={"sub": user.email})
return {"access_token": access_token, "token_type": "bearer"}


@app.get("/me", response_model=UserResponse)
async def read_users_me(
    current_user: UserInDB = Depends(get_current_user)
):
    return current_user


@app.get("/health")
def health_check():
    return {"status": "ok", "service": "user_service"}
```

## Додаток Є. Вихідний код додатку

```

<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="UTF-8">
  <title>Microservice Shop</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <header>
    <h1>Микросервисный Магазин (FastAPI & Docker)</h1>
    <p>API Gateway: http://localhost:8080</p>
  </header>

  <div id="status" class="status"></div>

  <section id="profile-info" style="border: 1px solid #ddd; padding: 15px;
margin-top: 15px;">
    <h3>Профиль и Баланс</h3>
    <p>Email: <strong id="profile-email">N/A</strong></p>
    <p>Баланс: <strong id="profile-balance">0.00</strong> USD</p>
    <button onclick="fetchProfileAndBalance()">Обновить</button>
  </section>

  <section id="auth-section">
    <h2>Регистрация / Авторизация</h2>
    <input type="email" id="email" placeholder="Email">
    <input type="password" id="password" placeholder="Пароль">
    <button onclick="register()">Регистрация</button>
  </section>

```

```

    <button onclick="login()">Войти</button>
    <button onclick="logout()">Выйти</button>
    <p id="user-info">Статус: Гость</p>
</section>

<hr>

<section id="actions-section">
    <h2>Действия с кошельком и товарами</h2>

    <div class="action-group">
        <h3>Пополнение кошелька</h3>
        <input type="number" id="deposit-amount" placeholder="Сумма"
value="100">
        <button onclick="deposit()">Пополнить</button>
    </div>

    <div class="action-group">
        <h3>Создать тестовый товар</h3>
        <input type="text" id="product-name" placeholder="Название
товара" value="Laptop">
        <input type="number" id="product-price" placeholder="Цена"
value="500">
        <input type="number" id="product-quantity"
placeholder="Количество" value="10">
        <button onclick="createProduct()">Создать товар</button>
    </div>

    <div class="action-group">
        <h3>Покупка товара</h3>

```

```

        <input type="text" id="buy-product-id" placeholder="ID товара">
        <input type="number" id="buy-quantity" placeholder="Количество"
value="1">
        <button onclick="purchase()">Купить</button>
    </div>
</section>

```

```

<section id="catalog-section">
    <h2>Каталог товаров</h2>
    <button onclick="fetchProducts()">Обновить каталог</button>
    <table id="products-table">
        <thead>
            <tr>
                <th>Название</th>
                <th>Цена</th>
                <th>В наличии</th>
            </tr>
        </thead>
        <tbody>
            </tbody>
        </table>
    </section>

```

```

    <script src="app.js"></script>
</body>
</html>

```

```

body { font-family: sans-serif; max-width: 800px; margin: 0 auto; padding:
20px; }

input, button { padding: 10px; margin: 5px; border: 1px solid #ccc; }

```



```
button { cursor: pointer; background-color: #3498db; color: white; border:
none; }
```

```
button:hover { background-color: #2980b9; }
```

```
.action-group { margin-top: 20px; padding: 15px; border: 1px solid #eee; }
```

```
.status { padding: 10px; margin-bottom: 10px; border: 1px solid red;
background-color: #fdd; }
```

```
table th:first-child,
```

```
table {
  border-collapse: collapse;
  width: 100%;
}
```

```
table th, table td {
  border: 1px solid #ddd;
  padding: 8px;
  text-align: left;
}
```

```
table tr:nth-child(even) {
  background-color: #f2f2f2;
}
```

```
table th {
  background-color: #4CAF50;
  color: white;
}
```

```
table tr:hover {
  background-color: #ddd;
}
```

```
const GATEWAY_URL = `http://${window.location.hostname}:8080`;
let ACCESS_TOKEN = localStorage.getItem('access_token') || null;
let CURRENT_USER_ID = null;
```

```
window.onload = function() {
  fetchProfileAndBalance();
  fetchProducts();
}
```

```
function setStatus(message, isError = false) {
  const statusDiv = document.getElementById('status');
  statusDiv.textContent = message;
  statusDiv.style.borderColor = isError ? 'red' : 'green';
  statusDiv.style.backgroundColor = isError ? '#fdd' : '#dfd';
  console.log(message);
}
```

```
function updateUserInfo(email = 'Гість') {
  // Оновлює статус під кнопками
  document.getElementById('user-info').textContent =
    ACCESS_TOKEN ? `Користувач: ${email} (Авторизований)` :
'Sтатус: Гість';
}
```

```
function getAuthHeaders() {
  return {
```

```

    'Authorization': `Bearer ${ACCESS_TOKEN}`,
    'Content-Type': 'application/json'
  };
}

```

```

async function fetchProfileAndBalance() {
  // Скидання полів при кожному виклику (щоб показати, що оновлення
  почалося)
  document.getElementById('profile-email').textContent =
  'Завантаження...';
  document.getElementById('profile-balance').textContent = '...';

  if (!ACCESS_TOKEN) {
    CURRENT_USER_ID = null;
    document.getElementById('profile-email').textContent = 'N/A';
    document.getElementById('profile-balance').textContent = '0.00 USD';
    updateUserInfo();
    return;
  }

  try {
    // 1. Отримання профілю (Email, ID)
    const profileResponse = await fetch(`${GATEWAY_URL}/auth/me`, {
      method: 'GET',
      headers: getAuthHeaders()
    });

    if (profileResponse.status === 401) {
      setStatus('Токен недійсний. Будь ласка, увійдіть.', true);
      return logout();
    }
  }
}

```

```

    }

    const profileData = await profileResponse.json();

    if (profileResponse.ok) {
        // ВИКОРИСТОВУЄМО EMAIL, ОСКІЛЬКИ ВІН Є РК У
        BILLING SERVICE (за результатами дебагу)
        CURRENT_USER_ID = profileData.email;

        // УСПІШНЕ ОНОВЛЕННЯ EMAIL
        document.getElementById('profile-email').textContent =
        profileData.email;
        updateUserInfo(profileData.email);

        // 2. Отримання балансу
        const balanceResponse = await fetch(
            `${GATEWAY_URL}/billing/account/${CURRENT_USER_ID}`,
            {
                method: 'GET',
                headers: getAuthHeaders()
            }
        );

        if (balanceResponse.ok) {
            const balanceData = await balanceResponse.json();
            document.getElementById('profile-balance').textContent =
                parseFloat(balanceData.balance).toFixed(2) + ' USD';
            setStatus('Профіль і баланс оновлено.', false);
        } else if (balanceResponse.status === 404) {

```

```

        document.getElementById('profile-balance').textContent = '0.00
USD';

        setStatus('Рахунок не знайдено (буде створений під час
поповнення).', false);
    } else {
        const errorData = await balanceResponse.json();
        setStatus(
            `Помилка балансу (5xx): ${errorData.detail} || 'Невідома
помилка'`,
            true
        );
    }
}
} catch (e) {
    setStatus('Мережева помилка під час завантаження профілю.', true);
    document.getElementById('profile-email').textContent = 'N/A';
    document.getElementById('profile-balance').textContent = 'Error';
}
}
}

```

// --- ФУНКЦІЇ АВТЕНТИФІКАЦІЇ ---

```

async function register() {
    const email = document.getElementById('email').value;
    const password = document.getElementById('password').value;

    try {
        const response = await fetch(`${GATEWAY_URL}/auth/register`, {
            method: 'POST',

```

```

        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ email, password })
    });

    const data = await response.json();

    if (response.ok) {
        setStatus('Реєстрація ${data.email} успішна.', false);
        await login();
    } else {
        setStatus('Помилка реєстрації: ${data.detail}', true);
    }
} catch (e) {
    setStatus('Мережева помилка: ${e.message}', true);
}

}

async function login() {
    const email = document.getElementById('email').value;
    const password = document.getElementById('password').value;

    try {
        const formData = new URLSearchParams();
        formData.append('username', email);
        formData.append('password', password);

        const response = await fetch(`${GATEWAY_URL}/auth/token`, {
            method: 'POST',
            headers: { 'Content-Type': 'application/x-www-form-urlencoded' },
            body: formData.toString()
        });
    }
}

```

```
});
```

```
const data = await response.json();
```

```
if (response.ok) {
```

```
    ACCESS_TOKEN = data.access_token;
```

```
    localStorage.setItem('access_token', ACCESS_TOKEN);
```

```
    setStatus('Вхід успішний. Токен отримано.', false);
```

```
    await fetchProfileAndBalance();
```

```
} else {
```

```
    setStatus('Помилка входу: ${data.detail}', true);
```

```
}
```

```
} catch (e) {
```

```
    setStatus('Мережева помилка: ${e.message}', true);
```

```
}
```

```
}
```

```
function logout() {
```

```
    ACCESS_TOKEN = null;
```

```
    localStorage.removeItem('access_token');
```

```
    CURRENT_USER_ID = null;
```

```
    setStatus('Вихід виконано.', false);
```

```
    fetchProfileAndBalance();
```

```
}
```

```
async function deposit() {
```

```
    if (!ACCESS_TOKEN) return setStatus('Спочатку виконайте вхід.',  
true);
```

```
    if (!CURRENT_USER_ID) {
```

```

        setStatus(
            'Зачекайте, триває оновлення профілю, або спробуйте увійти
знову.',
            true
        );
        return;
    }

    const amount = parseFloat(document.getElementById('deposit-
amount').value);

    try {
        const response = await fetch(`${GATEWAY_URL}/billing/deposit`, {
            method: 'POST',
            headers: getAuthHeaders(),
            body: JSON.stringify({ user_id: CURRENT_USER_ID, amount })
        });

        const data = await response.json();

        if (response.ok) {
            setStatus(
                `Поповнення успішне! Новий баланс:
${parseFloat(data.balance).toFixed(2)} USD`,
                false
            );
            await fetchProfileAndBalance();
        } else {
            setStatus(`Помилка поповнення: ${data.detail || data.message}`,
true);

```



```

    }
  } catch (e) {
    setStatus(`Мережева помилка: ${e.message}`, true);
  }
}

async function createProduct() {
  if (!ACCESS_TOKEN) return setStatus('Спочатку виконайте вхід.',
true);

  const name = document.getElementById('product-name').value;
  const price = parseFloat(document.getElementById('product-
price').value);
  const quantity = parseInt(document.getElementById('product-
quantity').value);

  try {
    const response = await fetch(`${GATEWAY_URL}/products`, {
      method: 'POST',
      headers: getAuthHeaders(),
      body: JSON.stringify({
        name,
        description: 'Тестовий товар з UI',
        price,
        initial_quantity: quantity
      })
    });

    const data = await response.json();

```

```

    if (response.ok) {
        setStatus(`Товар створено! ID: ${data.id}`, false);
        document.getElementById('buy-product-id').value = data.id;
        await fetchProducts();
    } else {
        setStatus(`Помилка створення товару: ${data.detail}`, true);
    }
} catch (e) {
    setStatus(`Мережева помилка: ${e.message}`, true);
}
}

async function purchase() {
    if (!ACCESS_TOKEN) return setStatus('Спочатку виконайте вхід.',
true);

    const product_id = document.getElementById('buy-product-id').value;
    const quantity = parseInt(document.getElementById('buy-
quantity').value);

    if (!product_id) return setStatus('Вкажіть ID товару.', true);

    try {
        const response = await fetch(`${GATEWAY_URL}/purchase`, {
            method: 'POST',
            headers: getAuthHeaders(),
            body: JSON.stringify({ product_id, quantity })
        });

        const data = await response.json();

```

```

    if (response.ok) {
        setStatus(` ПОКУПКА УСПІШНА: ${data.message}`, false);
        await fetchProfileAndBalance();
        await fetchProducts();
    } else {
        setStatus(` ПОМИЛКА ПОКУПКИ: ${data.detail || data.message}`,
true);
    }
} catch (e) {
    setStatus(` Мережева помилка: ${e.message}`, true);
}
}

```

```

async function fetchProducts() {
    const tableBody = document.querySelector('#products-table tbody');
    tableBody.innerHTML =
colspan="3">Завантаження...</td></tr>';

```

```

try {
    const response = await fetch(`${GATEWAY_URL}/products`);
    const products = await response.json();

    if (response.ok) {
        tableBody.innerHTML = "";
        if (products.length === 0) {
            tableBody.innerHTML =
                '<tr><td colspan="3">Каталог порожній. Створіть
товар.</td></tr>';
        }
    }
}

```

```

        products.forEach(product => {
            const row = tableBody.insertRow();
            row.insertCell().textContent = product.name;
            row.insertCell().textContent =
parseFloat(product.price).toFixed(2);
            row.insertCell().textContent = product.quantity;
        });
    } else {
        setStatus(
            `Помилка завантаження каталогу: ${products.detail || 'Невідома
помилка'}`,
            true
        );
        tableBody.innerHTML =
            '<tr><td colspan="3">Не вдалося завантажити
каталог.</td></tr>';
    }
} catch (e) {
    setStatus(`Мережева помилка під час завантаження каталогу:
${e.message}`, true);
    tableBody.innerHTML =
        '<tr><td colspan="3">Помилка мережі.</td></tr>';
    }
}

```