Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет ___________________*Комп’ютерних наук*___________________________
(повна назва)
Кафедра_______________________*Системотехніки*______________________________
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти ________________другий (магістерський)________________

_________Розробка мікросервісної архітектури вебзастосунків_________
(тема)

Виконав:
студент _2__ курсу, групи _*ІТПм-22-1*
______________*Янчинський І.В.*___________
(прізвище, ініціали)

Спеціальність *122 Інформаційні* ______________*технології проектуванняі*__
(код і повна назва спеціальності)
Тип програми __освітньо-професійна____
Освітня програма _*Комп'ютерні науки*___
____
(повна назва освітньої програми)

Керівник _______*проф. Ситніков. Д.Е*____
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри ____________ __*Гребеннік І.В.*__
(підпис) (прізвище, ініціали)

2024 р.

Харківський національний університет радіоелектроніки

Факультет *Комп'ютерних наук*

Кафедра *Системотехніки*

Рівень вищої освіти другий (магістерський)

Спеціальність *122 Інформаційні технології проектування*
(код і повна назва)

Тип програми освітньо-професійна

Освітня програма *Комп'ютерні науки*
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри ___________________
(підпис)
«___»_________________20___ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові *Янчнському Ігорю Всеволодовичу*
(прізвище, ім'я, по батькові)

1. Тема роботи: *Розробка мікросервісної архітектури вебзастосунків*
затверджена наказом по університету від *20.11* 2023 р. № *1373Ст*
2. Термін подання студентом роботи до екзаменаційної комісії: 05.02.2022 р
3.Вихідні дані до роботи: *розробити компоненти інформаційної системи вебзастосунку, що забезпечить автоматизацію основних бізнес процесів та бізнес функцій та порівняти отриману систему з монолітною архітектурою.*

4. Перелік питань, що потрібно опрацювати в роботі: *Вступ. Огляд критеріїв для розробки методу порівняння. Огляд критеріїв порівняльного аналізу. Опис математичної моделі порівняльного аналізу. Розробка експериментальних систем. Проведення експерименту.. Висновки.*

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп’ютерних ілюстрацій: *діаграма IDEF0, ER-діаграми розроблених баз даних, діаграма Use Case, діаграма послідовностей, моделі архітектур, моделі розробляємих елементів архітектур.*

6. Консультанти розділів роботи

Найменування розділу	Консультант (посада, прізвище, ім’я, по батькові)	Позначка консультанта про виконання розділу	
		підп	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	*Отримання завдання на виконання роботи*		
2	*Огляд критеріїв для розробки методу порівняння*		
3	*Огляд критеріїв порівняльного аналізу*		
4	*Огляд літератури*		
5	*Опис математичної моделі порівняльного аналізуі*		
6	*Розробка експериментальних систем*		
7	*Проведення експерименту*		
8	*Збор та аналіз даних експерименту*		
9	*Оформлення пояснювальної записки*		
10	*Представлення на рецензування*		

Дата видачі завдання 16.10.2023 р.

Студент ____________________
(підпис)

Керівник роботи ____________________ *проф Ситніков Д.Е.*
(підпис) (посада, прізвище, ініціали)

// РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи магістра містить: 85 с., 14 табл., 14 рис., 2 додатки, 24 джерел інформації.

ІНФОРМАЦІЙНА СИСТЕМА, МІКРОСЕРВІСНА АРХІТЕКТУРА, МОНОЛІТНА АРХІТЕКТУРА, UML, IDEF0, ПОРІВНЯЛЬНИЙ АНАЛІЗ, БАГАТОКРИТЕРІАЛЬНА МОДЕЛЬ ОПТИМІЗАЦІЇ

Об'єктом дослідження цієї роботи є розробка та впровадження системи з мікросервісною архітектурою.

Предметом дослідження є архітектурні особливості мікросервісних компонентів, їх структура та способи взаємодії між ними, з акцентом на гнучкість, масштабованість та незалежність цих компонентів.

Метою дослідження є визначення кращих практик та стратегій для ефективного застосування мікросервісної архітектури, а також аналіз умов, за яких вона є найбільш підходящою.

Методи дослідження включають системний підхід, структурний аналіз, моделювання взаємодій та процесів у мікросервісних системах, а також застосування концепцій реляційної алгебри та реляційного числення для оптимізації баз даних у контексті мікросервісів.

Робота включає теоретичний аналіз сучасних методів розробки мікросервісних архітектур, встановлення параметрів та критеріїв для їхньої оцінки та порівняння. Також передбачено проведення експериментального дослідження для перевірки розробленої системи.

Сфера застосування дослідження розширюється до практичного впровадження мікросервісної архітектури в реальних проектах програмного забезпечення та її моделювання.

ABSTRACT

Explanatory note to the qualification work of the masters contains: 85 pages, 14 tables, 14 figures, 2 appendices, 24 sources of information.

INFORMATION SYSTEM, MICROSERVICE ARCHITECHTURE, MONOLYTH ARCHITECHTURE, UML, IDEF0, COMPARATIVE ANALYSIS, MULTI-OBJECTIVE OPTIMIZATION MODEL

The object of study in this work is the development and implementation of a system with a microservice architecture.

The subject of study is the architectural features of microservice components, their structure, and ways of interaction among them, with an emphasis on the flexibility, scalability, and independence of these components.

The purpose of the study is to determine the best practices and strategies for the effective application of microservice architecture, as well as to analyze the conditions under which it is most suitable.

The research methods include a systemic approach, structural analysis, modeling of interactions and processes in microservice systems, and the application of concepts of relational algebra and relational calculus for the optimization of databases in the context of microservices.

The work includes a theoretical analysis of modern methods of developing microservice architectures, establishing parameters and criteria for their evaluation and comparison. An experimental study is also planned to test the developed system.

The scope of the research extends to the practical implementation of microservice architecture in real software projects and its modeling.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ 9
ВСТУП 10
1 ТЕОРЕТИЧНІ ОСНОВИ ТА МЕТОДОЛОГІЯ ПОРІВНЯЛЬНОГО АНАЛІЗУ МІКРОСЕРВІСНОЇ ТА МОНОЛІТНОЇ АРХІТЕКТУРИ 13
1.1 Огляд існуючих архітектурних парадигм 13
1.2 Критерії оцінювання мікросервісної і монолітної архітектури 19
1.3 Теоретичні аспекти мікросервісної та монолітної архітектури 22
1.4 Аналіз впливу архітектурного вибору на розробку та експлуатацію систем 24
2 РОЗРОБКА МЕТОДУ ТА ПЛАНУВАННЯ ЕКСПЕРИМЕНТАЛЬНОГО ДОСЛІДЖЕННЯ 26
2.1 Визначення параметрів для кожного критерію порівняльного методу 26
2.2 Розробка метрик оцінювання 32
2.3 Створення моделі оцінювання 38
2.4 Планування експерементального дослідження 41
3 РОЗРОБКА ЕКСПЕРЕМЕНТАЛЬНИХ ІС 43
3.1 Опис предметної області 43
3.2 Опис основних бізнес процесів ІС 45
3.3 Проектування і розробка компонентів ІС 50
4 ПРОВЕДЕННЯ ЕКСПЕРИМЕНТУ ОТРИМАНОГО ПОРІВНЯЛЬНОГО МЕТОДУ69
4.1 Загальні відомості предметної області розробки архітектури ПЗ 69
4.2 Збір даних 70
4.3 Застосування отриманої математичної моделі 71
4.4 Перевірка гіпотез 81
4.5 Можливі майбутні дослідження 81
ВИСНОВКИ 83
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 85
Додаток А Графічний матеріал кваліфікаційної роботи**Ошибка! Закладка не определена.**
Додаток Б Відомість кваліфікаційної роботи**Ошибка! Закладка не определена.**

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

БД – база даних;

ІС – інформаційна система;

СУБД – система керування базами даних;

CI/CD (Continuous Integration/Continuous Deployment) – неперервна інтеграція/неперервне розгортання;

API (Application Programming Interface, API) — набір визначень підпрограм;

ПЗ – програмне забезпечення.

ВСТУП

У контексті сучасного світу розробки програмного забезпечення, концепція архітектурних стилів є ключовою. Серед найбільш обговорюваних підходів є мікросервісна архітектура, яка в останні роки набула значної популярності, ставши вагомою альтернативою традиційним монолітним структурам. Головні переваги мікросервісів полягають у їх гнучкості, масштабованості та здатності до незалежного розгортання, що сприяє адаптації системи до змінних вимог бізнесу та швидшому розгортанню.

В той же час, монолітна архітектура, яка характеризується єдиним кодовим репозиторієм та тісною інтеграцією компонентів, продовжує займати своє місце, особливо в контексті менших або менш складних систем. Ця архітектура пропонує простоту у розробці та тестуванні, але зі зростанням проекту може стати обтяжливою для управління та масштабування, створюючи виклики при внесенні змін або оновленнях.

Перехід до мікросервісної архітектури, хоча й привабливий з багатьох точок зору, супроводжується своїми викликами. Ключовим серед них є управління залежностями між сервісами, організація ефективної мережевої взаємодії та забезпечення безпеки даних. Особливу увагу слід приділити механізмам комунікації між сервісами, наприклад, через API-інтерфейси чи шини повідомлень, а також забезпеченню надійності та консистентності даних у розподілених системах.

У сучасному світі інформаційних технологій, де швидкість розвитку та адаптивність до змін стають вирішальними, вибір між мікросервісною та монолітною архітектурою є критичним рішенням для багатьох організацій. Тому розробка методу порівняльного аналізу цих двох підходів стає особливо актуальною. Такий аналіз дасть змогу організаціям об'єктивно оцінити обидві архітектури з погляду вартості, ефективності, гнучкості, масштабованості та безпеки.

З огляду на швидку еволюцію технологічного середовища, з'являється потреба в методології, яка дозволить компаніям зробити обґрунтований вибір, виходячи з їх конкретних потреб та довгострокової стратегії. Порівняльний аналіз мікросервісної та монолітної архітектур може допомогти виявити потенційні переваги та недоліки кожної з них, а також сприяти кращому розумінню їхньої придатності для різних типів застосунків та бізнес-процесів.

Враховуючи це, ця робота спрямована на розробку комплексної мікросервісної системи, акцентуючи на її архітектурні особливості та потенціал. Центральним завданням є створення системи, яка використовує як кількісні, так і якісні показники для оцінки ефективності та продуктивності мікросервісної архітектури. Це надасть можливість стейкхолдерам, включаючи розробників, проектних менеджерів та кінцевих користувачів, приймати обґрунтовані рішення, виходячи з конкретних вимог та цілей їхніх проектів та організацій.

Об'єктом дослідження в цьому контексті є процес розробки та впровадження мікросервісних архітектур у програмному забезпеченні.

Предметом дослідження є вивчення та розробка мікросервісної системи, з особливим фокусом на її архітектурні особливості та можливості для оптимізації в контексті різних типів проектів програмного забезпечення.

В рамках дослідження був розроблений набір інноваційних метрик для оцінки мікросервісної та монолітної архітектур, які виходять за рамки традиційних технічних параметрів. Ці метрики забезпечують більш глибоке розуміння архітектурних рішень, охоплюючи аспекти, які впливають на загальну ефективність, вартість, масштабованість, та адаптивність систем. Вони спрямовані на всебічний аналіз і можуть бути застосовані для різноманітних сценаріїв розробки, що робить їх цінним інструментом для прийняття обґрунтованих архітектурних рішень. Також було розроблено ІС, що втілює мікросервісну архітектуру і перевірено як добре вона справляється з поставленими завдачами.

Розроблений метод порівняльного аналізу мікросервісної та монолітної архітектур істотно впливає на оптимізацію процесів розробки програмного

забезпечення. Цей вплив проявляється через здатність методики забезпечити більш ефективний вибір архітектури, виходячи з конкретних потреб та особливостей проекту, що сприяє зменшенню часу на розробку та випробування різних підходів. Водночас, стандартизований підхід до вибору архітектури забезпечує консистентність у процесах розробки, знижуючи ймовірність помилок та неефективних рішень.

Результати цієї наукової роботи були представлені та опубліковані у наступних наукових форумах та виданнях, що свідчить про визнання та цінність дослідження в академічній спільноті:

онференція «Інформаційні системи та технології» IST-2023;

урнал UNIVERSUM від Громадської організації «Молодіжна наукова ліга».

1 ТЕОРЕТИЧНІ ОСНОВИ ТА МЕТОДОЛОГІЯ ПОРІВНЯЛЬНОГО АНАЛІЗУ МІКРОСЕРВІСНОЇ ТА МОНОЛІТНОЇ АРХІТЕКТУРИ

1.1 Огляд існуючих архітектурних парадигм

Архітектурні підходи до розробки програмного забезпечення еволюціонували з часом, постійно відповідаючи на змінні потреби технологій та бізнесу. Від ранніх днів комп'ютингу, коли програми були прив'язані до величезних, монолітних машин, інженери шукали шляхи, як зробити програмне забезпечення більш зрозумілим, легким для розробки та обслуговування. Структурне програмування принесло перший значний прорив, вводячи чіткі правила для створення більш організованого та читабельного коду, що значно полегшило розробку та відладку.

Як розвивалися технології, з'явилася потреба у більш модульних підходах, що дозволили розробникам розділяти програми на менші, взаємозамінні частини. Модульне програмування та подальший перехід до об'єктно-орієнтованого програмування дозволили програмістам створювати програми з високим рівнем абстракції та повторного використання компонентів [1].

Проте, зі зростанням складності програмних систем з'явилася потреба у більш гнучких архітектурних рішеннях. Монолітні архітектури, хоча і забезпечували зручність управління та розгортання, стикалися з проблемами масштабованості та складності оновлень. Відповіддю на це стала сервісно-орієнтована архітектура (SOA), яка впровадила концепцію розділення функціональності на окремі сервіси, що взаємодіють через добре визначені інтерфейси. SOA дозволила підприємствам краще управляти великими системами та полегшила інтеграцію різноманітних технологій.

Останнім кроком в еволюції архітектур стали мікросервіси, які розширили ідеї SOA, пропонуючи ще більшу гнучкість та незалежність. Мікросервісна архітектура розглядає кожну функцію як окремий, самодостатній сервіс, що може бути розроблений, розгорнутий та масштабований незалежно від інших. Такий підхід забезпечив революційні зміни у способах розробки, управління та

розгортання програмного забезпечення, відповідаючи на потреби сучасних динамічних бізнес-середовищ.

Кожен етап цієї еволюції був відгуком на обмеження та проблеми попереднього, водночас відкриваючи нові можливості для розвитку програмного забезпечення. В результаті, сьогоднішні архітектури є надзвичайно розмаїтими, що дозволяє командам вибирати найбільш підходящі підходи для їх конкретних потреб та цілей [2-3].

Мікросервісна архітектура є сучасним підходом до розробки програмного забезпечення, який базується на використанні маленьких, самостійних сервісів. Кожен з цих сервісів відповідає за певний бізнес-процес або функціональність і може бути розроблений, розгорнутий та масштабований незалежно від інших. Це надає значну гнучкість, дозволяючи використовувати різні технологічні стеки та мови програмування для різних сервісів.

Модульність є ключовою характеристикою мікросервісної архітектури, що дозволяє розглядати кожен сервіс як окрему одиницю, що спрощує розробку, тестування та розгортання. Це також сприяє більшій відмовостійкості системи, оскільки збої в одному сервісі менше впливають на решту системи.

У контексті мережевого спілкування, мікросервіси взаємодіють один з одним за допомогою легковагових протоколів, часто використовуючи RESTful API. Це дозволяє сервісам спілкуватися через мережу, що приносить гнучкість у виборі розташування та масштабуванні сервісів. Проте, ця розподіленість може вести до збільшення мережевих затримок та складності управління транзакціями.

Одним з головних переваг мікросервісної архітектури є незалежність розгортання. Кожен сервіс може бути розгорнутий окремо, що дозволяє швидше впроваджувати зміни, оновлення та нові функції. Це також сприяє децентралізації управління даними, оскільки кожен сервіс може використовувати свою власну базу даних, забезпечуючи вищу ступінь незалежності та гнучкості.

Мікросервісна архітектура також підтримує незалежний розвиток, дозволяючи різним командам розробників працювати на різних сервісах одночасно, що значно спрощує управління проектами та збільшує швидкість розробки. Кожен сервіс може бути оновлений і масштабований незалежно від інших, що забезпечує високу масштабованість та гнучкість системи [4-6].

Однак, мікросервісна архітектура несе із собою певні виклики. Складність управління значно зростає через необхідність координації численних незалежних сервісів, а також через складності, пов'язані з мережевими затримками та управлінням транзакціями. Крім того, необхідність в облаштуванні ефективних систем моніторингу та логування стає критично важливою для забезпечення надійності та ефективності в роботі мікросервісів.

Таким чином, мікросервісна архітектура пропонує значні переваги у гнучкості, масштабованості та незалежності, але також потребує ретельного планування та управління для вирішення специфічних викликів, що вона пропонує.

На рис 1.1 була представлена схема типової мікросервісної архітектури.

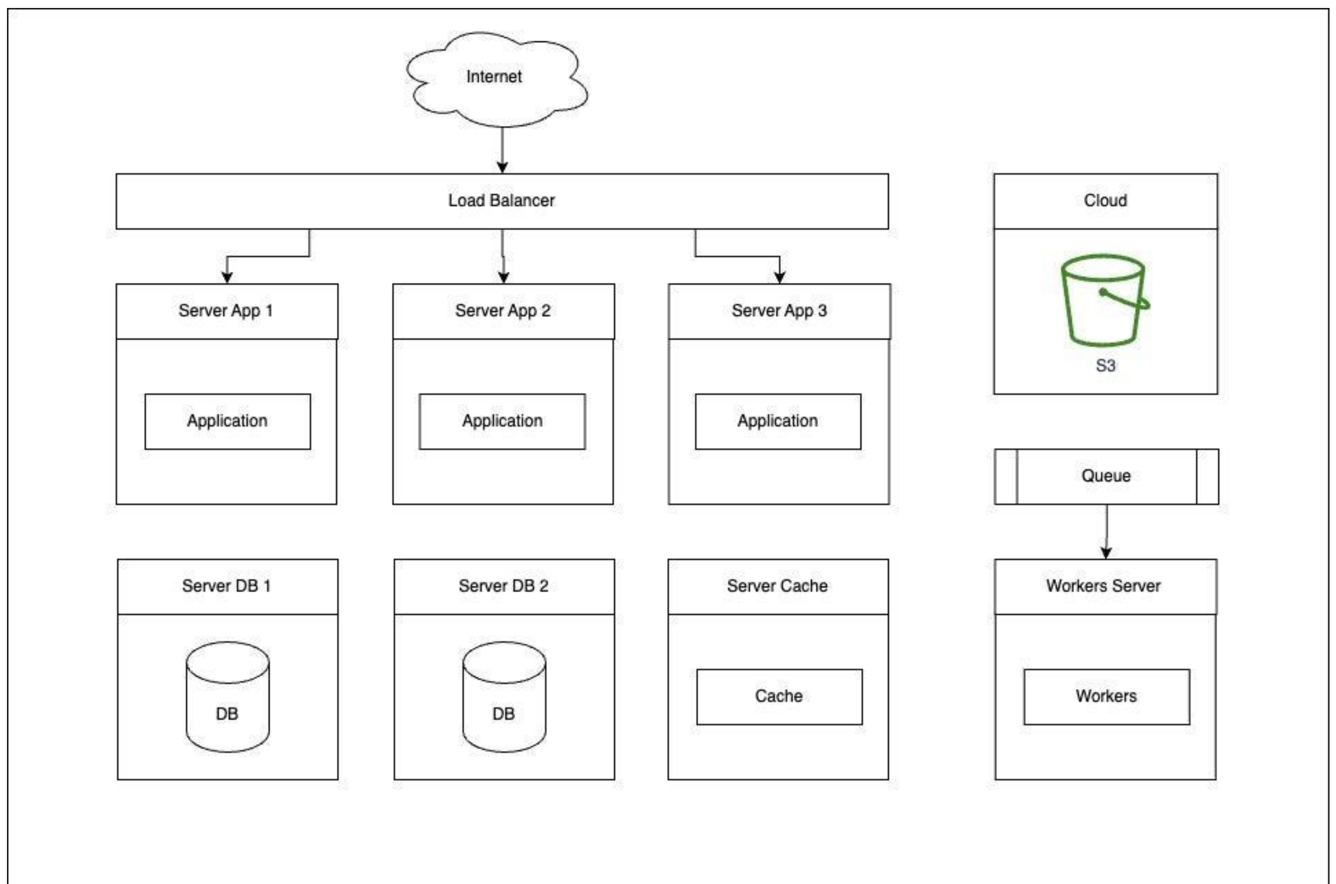

Рис.1.1 – Схема мікросервісної архітектури

На зображенні представлена схема, що ілюструє загальний вигляд мікросервісної архітектури. Хоча схема не показує всі характерні риси традиційної мікросервісної архітектури, вона відображає декілька важливих елементів розподіленої системи:

Інтернет»: зображений як хмара, що символізує зовнішню мережу, через яку користувачі можуть звертатися до системи;

Load Balancer (Балансувальник навантаження)»: відіграє роль в розподілі вхідних звернень з Інтернету між різними серверами додатків, забезпечуючи оптимальне розподілення навантаження та підвищення доступності та надійності системи;

Server App 1, 2, 3 (Сервери додатків)»: представляють собою індивідуальні інстанції або контейнери, кожен з яких містить копію додатку. Ці сервери можуть обробляти різні бізнес-завдання або слугувати різним користувацьким групам;

Server DB 1, 2 (Сервери баз даних)»: ці компоненти є базами даних, які використовуються серверами додатків для зберігання та витягування даних.

Можливо, вони містять різні схеми або набори даних, в залежності від архітектури системи;
Server Cache (Сервер кешування)»: забезпечує проміжне зберігання часто запитуваної інформації для швидкого доступу, знижуючи затримку та навантаження на сервери баз даних;
відноситься до сервісу Amazon S3, який є об'єктним сховищем у хмарі і може використовуватися для зберігання різноманітних даних, включаючи медіа-файли, дані резервних копій тощо;
Queue (Черга)»: використовується для асинхронної обробки завдань, управління потоками даних між сервісами, та забезпечення послідовної обробки завдань, які не можуть бути виконані відразу;
Workers Server (Сервер обробників)»: отримує завдання з черги та виконує їх. Обробники можуть виконувати фонові завдання, такі як обробка даних, відправлення електронних листів тощо.

Монолітна архітектура є традиційним підходом у розробці програмного забезпечення, де додаток розробляється як єдине цілісне рішення. У такому підході, інтерфейс користувача, бізнес-логіка, обробка даних та інші функціональні компоненти тісно інтегровані в одну програму, яка виконується як один процес. Основною перевагою цього методу є спрощення розробки та розгортання: оскільки всі частини додатку зібрані разом, розробники можуть легко координувати зміни, а розгортання зводиться до запуску єдиного виконуваного файлу [7-8].

Проте, монолітна структура може створювати складнощі у міру зростання додатку. З огляду на те, що всі компоненти програми міцно зв'язані, масштабування окремих функцій може вимагати масштабування всієї програми, що не завжди є оптимальним рішенням з точки зору використання ресурсів. Крім того, оскільки всі функції залежать одна від одної, невеликі зміни або оновлення можуть виявитися непростими, вимагаючи повного тестування та розгортання всього додатку.

Ще одним важливим моментом є те, що в монолітних системах код може стати дуже заплутаним, що ускладнює його розуміння та подальше розширення. Це створює залежності, які ускладнюють внесення змін, оскільки розробники повинні бути обізнані про всі взаємозв'язки в коді, щоб уникнути помилок, які можуть вплинути на роботу додатку в цілому.

Крім того, у випадку виникнення збоїв, монолітна архітектура може призвести до повного зупинення системи, оскільки всі компоненти взаємозалежні. Відновлення після таких збоїв часто вимагає значних зусиль та часу, що може бути критичним для бізнесу.

Попри це, на ранніх етапах життєвого циклу проекту, монолітна архітектура може бути вигідною, оскільки вона дозволяє швидко створити робочий продукт і легше контролювати всі елементи програми. Однак, зі збільшенням масштабів та складності, додаток може стати важко керованим і малоадаптивним до нових вимог ринку [9-11].

Таким чином, монолітна архітектура має місце в певних сценаріях, але із розвитком технологій та підвищенням вимог до швидкості та гнучкості розробки, багато компаній віддають перевагу більш модульним та адаптивним підходам, таким як мікросервісна архітектура.

Приклад монолітної архітектури показано на рис 1.2.

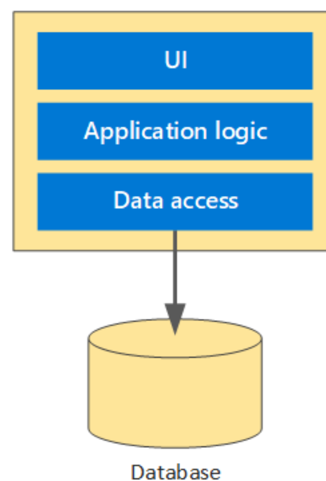

Рис 1.2 – Схема монолітної архітектури

Зображення показує типову структуру монолітної архітектури програмного забезпечення, де всі основні компоненти додатку інтегровані в один єдиний стек. На верху знаходиться шар користувацького інтерфейсу (UI), який є точкою взаємодії з користувачами. Цей шар відповідає за представлення даних та прийняття вхідних даних від користувачів.

Нижче UI розташований шар бізнес-логіки (Application logic), який містить основну функціональність додатку. Він обробляє вхідні дані, виконує бізнес-правила, розрахунки та приймає рішення на основі логіки, запрограмованої в системі. Цей шар відокремлює логіку від інтерфейсу користувача та шару доступу до даних, але в монолітній структурі він тісно пов'язаний з іншими компонентами.

Шар доступу до даних (Data access) знаходиться під шаром бізнес-логіки та відповідає за взаємодію з базою даних. Цей шар виконує запити до бази даних та обробляє отримані дані для використання в бізнес-логіці або для відображення в користувацькому інтерфейсі. Він абстрагує логіку доступу до даних від інших частин системи, дозволяючи змінювати способи зберігання та обробки даних без впливу на загальну функціональність [12].

У нижній частині схеми розташована база даних (Database), яка є централізованим сховищем всіх даних програми. Вона взаємодіє безпосередньо з шаром доступу до даних, приймаючи запити на зберігання, оновлення, видалення та витягування даних.

Ця структура демонструє як у монолітному додатку компоненти тісно інтегровані та залежні один від одного. Такий підхід може спростити початкову розробку та розгортання, але також може привести до складнощів із масштабуванням та утриманням програми в довгостроковій перспективі.

1.2 Критерії оцінювання мікросервісної і монолітної архітектури

В архітектурі програмного забезпечення існує набір критеріїв, які використовуються для оцінювання та порівняння різних архітектурних рішень.

Ці критерії відображають ключові аспекти системи, які впливають на її проектування, реалізацію та експлуатацію [13-14].

Час до ринку, як поняття, вказує на швидкість, з якою новий продукт або функція може бути представлена користувачам. Він залежить від багатьох факторів, включаючи складність розробки, ефективність процесів розробки та розгортання, а також від спроможності команди швидко адаптуватися до змін.

Масштабованість описує здатність системи ефективно працювати при збільшенні обсягів роботи або кількості користувачів. Вона тісно пов'язана з архітектурними рішеннями, оскільки вимагає від системи здатності додавати ресурси для обробки додаткового навантаження без шкоди для продуктивності.

Відновлення після збоїв стосується здатності системи швидко відновити роботу після помилок або проблем. Цей аспект включає не тільки технічні характеристики системи, але й процедури, які можуть бути впроваджені для мінімізації часу простою та збитків.

Складність розробки відноситься до того, наскільки легко чи важко розробники можуть додавати нові функції, вносити зміни чи відлагоджувати існуючі. Вона залежить від багатьох факторів, включаючи чистоту архітектурного дизайну, якість коду, а також наявність і якість документації.

Простота управління відіграє ключову роль у підтримці та експлуатації системи. Вона включає зручність моніторингу системи, логування, резервне копіювання та відновлення, а також управління конфігурацією та версіями.

Вартість розгортання та експлуатації системи включає не тільки безпосередні витрати на обладнання та програмне забезпечення, але й витрати на розробку, тестування, підтримку та можливі зміни в архітектурі.

Модифікованість та гнучкість системи визначають, наскільки легко систему можна адаптувати до змінних вимог. Це включає здатність до рефакторингу, введення нових технологій та інтеграції з іншими системами.

Безпека охоплює заходи, що запобігають несанкціонованому доступу та забезпечують захист даних. Це стосується всього: від фізичного захисту серверів до захисту програмного забезпечення та даних.

Ці критерії є фундаментальними для розуміння та оцінки архітектур програмного забезпечення, і їх слід розглядати як частину більшого контексту, який включає бізнес-цілі, ринкові умови, командну динаміку та технологічний ландшафт.

У світі програмної інженерії аналіз архітектури займає центральне місце, оскільки він допомагає зрозуміти, як різні дизайнерські рішення впливають на загальне функціонування та ефективність систем. Аналітики використовують суміш кількісних та квалітативних методів, які сприяють глибокому розумінню різних аспектів системи. Вони оцінюють, як система впорається з навантаженням, її здатність до масштабування та відновлення після збоїв, розглядаючи при цьому і час до ринку, і економічну ефективність [15-16].

Кількісний аналіз, наприклад, забезпечує цифри та факти, які можна легко порівняти, але він може не відображати повної картини, особливо коли мова йде про якість та стійкість коду. Тим часом квалітативний аналіз може запропонувати більш нюансоване розуміння, але його результати часто варіюються в залежності від досвіду та перспективи аналітика.

Моделювання та симуляція доповнюють ці методи, надаючи можливість прогнозувати поведінку системи в контрольованих умовах. Це корисно для випробування потенційних майбутніх змін, хоча не завжди може точно відтворити реальні умови експлуатації.

Сценарійний аналіз вносить в розгляд реалістичні ситуації, оцінюючи, як система впорається з конкретними завданнями та викликами. Цей метод особливо корисний для розуміння бізнес-аспектів системи, але він може не врахувати всі можливі сценарії.

Експертні огляди залучають досвідчених професіоналів, чий глибокий аналіз може виявити проблеми, які інші методи можуть пропустити. Водночас, навіть найкращі експерти не можуть бути абсолютно об'єктивними, тому їх оцінки можуть містити суб'єктивність.

Всі ці методи складають багатогранний підхід до аналізу архітектур, дозволяючи аналітикам отримати глибоке та всебічне розуміння системи.

Використання комбінації цих методів може допомогти виявити не тільки сильні сторони архітектури, але й потенційні ризики та області для поліпшення.

1.3 Теоретичні аспекти мікросервісної та монолітної архітектури

Мікросервісна та монолітна архітектури знайшли своє застосування у різних сферах, кожна з яких має свої унікальні вимоги та контекст використання.

Мікросервісна архітектура часто застосовується у великих, складних системах, де потрібна висока масштабованість та гнучкість. Такі системи зазвичай зустрічаються у наступних галузях:

Веб-сервіси та облачні платформи»: де необхідно швидко реагувати на зміну навантаження та запитів користувачів;

Електронна комерція»: для оптимізації процесів купівлі, вибору продуктів та управління інвентаризацією;

Стартапи та інноваційні компанії»: які цінують швидкість випуску продукту та часті ітераціїж

Фінансові технології (FinTech)»: де потрібна висока надійність та безпека при швидкому розвитку нових продуктів;

Інтернет речей (IoT)»: через потребу в обробці великої кількості даних та взаємодії з різноманітними пристроями.

Монолітна архітектура, в свою чергу, добре підходить для проектів, де стабільність та простота є більш важливими, ніж швидкість змін або масштабування. Типові застосування включають:

Внутрішні корпоративні системи»: такі як управління ресурсами компанії (ERP) або управління відносинами з клієнтами (CRM);

Малі та середні веб-додатки»: де вимоги до масштабування є помірними;

Стартові проекти та MVPs (мінімально життєздатні продукти)»: де швидкість випуску на ринок є критичною, а ресурси та складність управління мають бути мінімізовані;

Додатки з високим рівнем транзакцій»: де транзакційна цілісність та простота транзакційної логіки є важливими.

Вибір між мікросервісною та монолітною архітектурою залежить від багатьох факторів, включаючи бізнес-цілі, командні навички, очікуваний час життєвого циклу продукту та готовність до управління складністю. Важливо враховувати, що обрана архітектура може впливати на довгострокову стратегію продукту та його здатність адаптуватися до змін у ринкових умовах.

1.4 Аналіз впливу архітектурного вибору на розробку та експлуатацію систем

Вибір архітектури програмного забезпечення глибоко впливає на всі аспекти розробки та тестування, від управління проектом до безпеки та ефективності систем. У монолітній архітектурі, де всі компоненти інтегровані, розробка та тестування часто зосереджуються навколо єдиного кодового репозиторію, що може полегшити управління залежностями та версіонуванням. Однак, це також може створювати складності, оскільки невеликі зміни вимагають повного розгортання всієї системи, що підвищує ризик помилок та ускладнює тестування.

З іншого боку, мікросервісна архітектура дозволяє розробникам незалежно працювати над окремими сервісами, сприяючи спеціалізації та оптимізації процесів під конкретні потреби. Це підхід вимагає більш складної інфраструктури для неперервної інтеграції та розгортання, але пропонує підвищену гнучкість у тестуванні та швидше реагування на зміни. Мікросервіси забезпечують більшу модульність у тестуванні, дозволяючи зосередитися на мережевих взаємодіях та інтеграції API, однак координація між сервісами та управління спільними ресурсами може стати викликом.

Управління проектами в мікросервісних архітектурах часто включає децентралізовані команди, що працюють над окремими сервісами, що може прискорити розробку та полегшити впровадження нових функцій. Це забезпечує високу адаптивність до змін, але вимагає ефективної комунікації та координації. Натомість, монолітні проекти часто керуються більш централізовано, що сприяє єдності рішень та легшому контролю, але може уповільнити процеси прийняття рішень та адаптації до змін [17].

З точки зору безпеки, мікросервісна архітектура має потенційно більшу кількість точок входу, що вимагає складнішого управління безпекою, в той час як монолітні системи можуть бути вразливішими до повсюдних збоїв у разі виявлення вразливості. Вибір архітектури також впливає на ефективність

системи: мікросервіси пропонують гнучкість та оптимальне використання ресурсів через незалежне масштабування, тоді як моноліти забезпечують стабільність та передбачуваність, але можуть стикатися з викликами при швидкому масштабуванні.

Таким чином, вибір між мікросервісною та монолітною архітектурою має широкий вплив на управління проектом, від методології розробки та тестування до управління безпекою та ефективністю систем. Цей вибір вимагає ретельного розгляду потреб проекту, можливостей команди, та стратегічних цілей організації, оскільки він визначає не тільки технічні аспекти роботи, але й загальний підхід до управління та розвитку проекту.

2 РОЗРОБКА МЕТОДУ ТА ПЛАНУВАННЯ ЕКСПЕРИМЕНТАЛЬНОГО ДОСЛІДЖЕННЯ

2.1 Визначення параметрів для кожного критерію порівняльного методу

У цій роботі було вирішено зосередитись на розробці покращеної версії методу кількісного аналізу для порівняння мікросервісної та монолітної архітектур програмного забезпечення. Цей метод розроблений з метою надання більш точної, об'єктивної та всебічної оцінки ключових показників ефективності обох архітектур. Він включає в себе розширений набір кількісних критеріїв, таких як час впровадження змін, продуктивність, масштабованість, надійність, використання ресурсів, вартість розгортання та експлуатації, модифікованість, та безпека.

Кожен з цих критеріїв ретельно вибраний та адаптований для забезпечення точного вимірювання та порівняння архітектурних характеристик. Цей підхід дозволяє не лише виявити сильні та слабкі сторони кожної архітектури, але й надає змогу краще зрозуміти вплив різних дизайнерських рішень на загальну продуктивність та ефективність систем. Ця робота прагне заповнити існуючі прогалини у кількісному аналізі, надаючи більш глибокий інсайт у вплив архітектурних виборів на загальну ефективність програмного забезпечення в різних бізнес-контекстах.

Для розробки вдосконаленого методу кількісного аналізу мікросервісної та монолітної архітектур, мною було вибрано наступні ключові критерії, які вважаються визначальними для оцінки ефективності та функціональності обох підходів до архітектури програмного забезпечення:

ас впровадження змін (Time to Market): цей критерій важливий для оцінки швидкості, з якою нові функції або оновлення можуть бути впроваджені на ринок. Він безпосередньо впливає на конкурентоспроможність та інноваційність продукту;

родуктивність (Performance): продуктивність є ключовим фактором, що впливає на користувацький досвід та загальну ефективність системи. Оцінка

продуктивності дозволяє зрозуміти, як система справляється з обробкою запитів та виконанням завдань;
асштабованість (Scalability): масштабованість визначає здатність системи адаптуватися до зростаючих або змінюваних вимог без компромісу до продуктивності. Це критично для систем, що зазнають змінних рівнів навантаження;
адійність (Reliability): надійність вказує на стабільність та безперебійність роботи системи, що є важливим для підтримки довіри користувачів та неперервності бізнес-процесів;
ас відновлення після збою (Recovery Time): цей критерій вимірює ефективність системи у відновленні після несподіваних збоїв або помилок, що є ключовим аспектом для забезпечення високої доступності сервісів;
икористання ресурсів (Resource Utilization): вимірювання, наскільки ефективно система використовує обчислювальні ресурси (наприклад, пам'ять, процесор), дозволяє оцінити її оптимізацію та вартісну ефективність;
езпека (Security): у сучасному цифровому світі безпека є пріоритетним аспектом. Оцінка безпеки включає в себе аналіз вразливостей, засобів захисту даних та спроможності системи протистояти зловмисним атакам.

Ці критерії були обрані на основі їхньої важливості для забезпечення комплексної оцінки архітектурних рішень. Вони дозволяють не тільки оцінити поточний стан обох архітектур, але й прогнозувати їхню ефективність у різних операційних та бізнес-умовах.

Час до ринку – критерій, який охоплює кілька ключових аспектів, які разом формують загальний час, необхідний для того, щоб нові функції чи оновлення були введені на ринок. Важливим елементом є тривалість розробки, яка включає час від початку планування до завершення розробки нової функції або оновлення. Далі йде час тестування, який визначає, скільки часу потрібно для проведення повного циклу тестування змін. Це включає юніт-тестування, інтеграційне тестування, та приймальне тестування. Також важливим є час

розгортання, який оцінює, скільки часу необхідно для впровадження нової версії продукту в продуктивне середовище.

Окремою складовою є час до першої відповіді користувача, який вимірює, скільки часу проходить від розгортання продукту до моменту отримання першої зворотньої відповіді від кінцевих користувачів. Це може включати фідбек щодо користувальницького досвіду, знайдених помилок, або інших важливих показників. Частота релізів є ще одним важливим показником, який вимірює, як часто команда може випускати оновлення або нові функції. Вища частота релізів часто вказує на більш швидкий час до ринку. Нарешті, час на усунення помилок/дефектів, який вимірює час від ідентифікації помилки до її виправлення та розгортання оновлення, та гнучкість у впровадженні змін, що оцінює здатність системи вносити зміни без значного перероблення коду або архітектури, є ключовими параметрами для оцінки цього критерію [18].

У контексті розробки вдосконаленого методу кількісного аналізу, особлива увага приділяється критерію "Продуктивність (Performance)" у порівнянні мікросервісної та монолітної архітектур. Для глибокого розуміння цього аспекту було визначено ряд ключових параметрів, які охоплюють різні грані продуктивності системи.

Першим з параметрів є "Час відповіді системи", який вимірюється для оцінки тривалості часу від отримання запиту до моменту надання відповіді системою. Цей параметр вважається важливим для оцінки швидкості обробки інформації. Далі йде "Пропускна здатність", яка визначається як кількість операцій або запитів, що система може обробити за одиницю часу, і є критичною для забезпечення високої продуктивності під час пікових навантажень.

"Час завантаження ресурсів" також вимірюється, щоб оцінити тривалість завантаження та ініціалізації необхідних системних ресурсів. "Ефективність використання ресурсів" аналізується для визначення, наскільки ефективно система використовує обчислювальні ресурси, включаючи ЦПУ, пам'ять та дисковий простір.

"Стабільність системи під навантаженням" вивчається, щоб зрозуміти, як змінюється продуктивність системи при збільшенні або зменшенні обсягу роботи. Аналізується також "Масштабування під навантаженням", що визначає здатність системи збільшувати свої ресурси для забезпечення необхідної продуктивності.

Останній параметр, "Оптимізація запитів", вивчається для оцінки ефективності обробки запитів до баз даних та інших зовнішніх систем, що є ключовим для забезпечення високої продуктивності системи.

Розуміння масштабованості системи є ключовим для визначення її здатності адаптуватися до зростаючих або змінюваних вимог. Для цього було визначено низку параметрів, які допомагають оцінити цей аспект.

Перш за все, "Вертикальна масштабованість" аналізується для визначення здатності системи збільшувати свою продуктивність шляхом додавання ресурсів до існуючих вузлів, таких як ЦПУ або пам'ять. Це важливо для розуміння, наскільки ефективно може бути збільшена потужність системи без зміни її архітектури.

"Горизонтальна масштабованість" також оцінюється, що включає аналіз здатності системи розширюватися шляхом додавання додаткових вузлів або інстансів. Це дає уявлення про можливості системи розширюватися для обробки збільшеного обсягу запитів або транзакцій.

Далі "Час на масштабування" вивчається, що визначає швидкість, з якою система може бути масштабована відповідно до змінних вимог. Це включає час, необхідний для додавання нових ресурсів або інстансів, та їх інтеграції у систему.

Оцінюється також "Вартість масштабування", яка включає аналіз фінансових витрат на збільшення масштабу системи, включаючи витрати на додаткове обладнання, ліцензування та інфраструктуру.

"Ефективність розподілу навантаження" розглядається для оцінки здатності системи рівномірно розподіляти навантаження між різними вузлами або компонентами, що є важливим для забезпечення стабільної продуктивності під час масштабування.

Останнім параметром є "Гнучкість архітектури", яка аналізується для визначення, наскільки легко можна внести зміни в архітектуру системи для її масштабування, включаючи здатність до адаптації під змінювані бізнес-вимоги та технологічні умови [19].

Для оцінки критерію "Надійність (Reliability)" у контексті порівняльного аналізу мікросервісної та монолітної архітектур, було визначено декілька ключових параметрів, які допомагають всебічно оцінити цей аспект.

Першим параметром є "Частота збоїв", яка оцінюється для визначення регулярності непередбачуваних збоїв чи відмов у системі. Це включає аналіз інцидентів, що призводять до часткового або повного переривання функціонування системи.

Далі вивчається "Час до відмови (Mean Time Between Failures, MTBF)", який вимірює середній час між збоями або відмовами у системі, що є показником стабільності та надійності.

"Час на відновлення після збою (Mean Time To Recovery, MTTR)" також аналізується, визначаючи середній час, необхідний для відновлення системи після збою. Це важливо для оцінки спроможності системи швидко повертатися до нормального функціонування.

"Запас надійності" розглядається для визначення ступеня, до якого система може витримувати пікові навантаження або стресові ситуації, перш ніж вона зазнає збою. Це дає уявлення про межі витривалості системи.

"Автоматизація відновлення" оцінюється для аналізу здатності системи автоматично відновлюватися після збоїв без втручання людини. Це включає механізми відновлення, такі як автоматичне перезавантаження або переключення на резервні системи.

"Журналювання помилок та моніторинг" також враховується, оскільки забезпечення належного ведення журналів та моніторингу системи є критично важливим для своєчасного виявлення та вирішення проблем.

"Резервне копіювання та відновлення даних" аналізується для оцінки здатності системи забезпечувати безпеку даних та їх відновлення у випадку втрати або пошкодження.

У рамках розробки методу кількісного аналізу для порівняння мікросервісної та монолітної архітектур, важливим аспектом є оцінка "Використання ресурсів (Resource Utilization)". Цей критерій охоплює декілька ключових параметрів, які дозволяють зрозуміти, наскільки ефективно система використовує доступні ресурси.

Першим параметром є "Використання ЦПУ (CPU Utilization)", що вимірює відсоток часу, протягом якого центральний процесор зайнятий обробкою завдань. Це важливий показник, що відображає навантаження на процесор і допомагає оцінити, наскільки ефективно система використовує цей критичний ресурс.

"Використання пам'яті (Memory Utilization)" також аналізується, що відображає кількість використовуваної оперативної пам'яті. Цей параметр важливий для оцінки ефективності роботи системи, особливо в контексті управління пам'яттю та запобігання витокам пам'яті.

Далі враховується "Використання дискового простору (Disk Space Utilization)", яке вимірює, скільки місця на диску використовується системою. Це допомагає зрозуміти, наскільки ефективно система використовує дисковий простір та чи є достатньо ресурсів для зберігання даних.

При оцінці критерію "Безпека (Security)" у порівняльному аналізі мікросервісної та монолітної архітектур, було визначено ряд параметрів, які дозволяють всебічно оцінити цей важливий аспект.

"Рівень захисту даних" аналізується для оцінки способів шифрування, зберігання та обробки даних у системі. Це включає механізми захисту від несанкціонованого доступу та витоку інформації, що є важливим для забезпечення конфіденційності даних [20].

"Заходи проти зловмисних атак" оцінюються, щоб визначити здатність системи відстежувати та протидіяти спробам несанкціонованого доступу, включаючи захист від DDoS-атак, вірусів та інших видів кіберзагроз.

"Аудит та моніторинг безпеки" аналізується для оцінки наявності та ефективності систем аудиту та моніторингу, які дозволяють виявляти та реагувати на потенційні інциденти безпеки.

"Політики безпеки та процедури відповіді на інциденти" також вивчаються, щоб оцінити наявність та дієвість процедур і політик, які визначають правила використання системи та способи реагування на інциденти безпеки.

"Комплексність системи управління ідентифікацією та авторизацією" оцінюється, включаючи механізми аутентифікації, контролю доступу та управління користувацькими ролями. Це важливо для забезпечення того, що доступ до ресурсів системи мають лише уповноважені особи.

"Регулярність оновлень безпеки та патчів" аналізується, щоб визначити частоту та оперативність оновлень безпеки, що допомагає у забезпеченні захисту від нових та відомих уразливостей.

"Сертифікації та стандарти безпеки" також враховуються, оскільки наявність міжнародних та галузевих сертифікацій може свідчити про відповідність системи встановленим стандартам безпеки.

2.2 Розробка метрик оцінювання

Метрики у контексті розробки програмного забезпечення — це кількісні показники, які використовуються для оцінки, вимірювання та порівняння різних аспектів процесу розробки, якості продукту, та ефективності команди. Вони служать інструментом для об'єктивного аналізу та прийняття обґрунтованих рішень, підвищення продуктивності та ефективності процесів, а також забезпечення відповідності продукту вимогам та стандартам [21-22].

Метрика у сфері розробки програмного забезпечення — це стандартизований метод вимірювання певного елемента процесу розробки або його результатів. Метрики можуть бути як кількісними, так і якісними, та часто виражаються у вигляді числових значень, що дозволяє проводити порівняльний аналіз.

Основна мета використання метрик — це поліпшення процесів розробки, підвищення якості кінцевого продукту, та забезпечення прозорості та відстежуваності процесів. Метрики можуть використовуватися для ідентифікації слабких місць у розробці, оцінки продуктивності команди, а також для планування ресурсів та бюджету.

Метрики можна класифікувати за різними категоріями, зокрема:

Метрики Продуктивності»: Вимірюють швидкість та ефективність розробки, наприклад, час від визначення вимог до запуску продукту;

Метрики Якості»: Фокусуються на надійності, безпеці, та здатності продукту задовольняти визначені вимоги;

Метрики Ефективності»: Оцінюють, наскільки ефективно використовуються ресурси та час під час розробки;

Метрики Задоволення Клієнтів»: Вимірюють рівень задоволення кінцевих користувачів продуктом.

Використання метрик дозволяє керівникам проектів, розробникам, та зацікавленим сторонам отримувати чітке уявлення про стан проекту та приймати своєчасні та ефективні рішення. Вони також сприяють встановленню чітких цілей та очікувань, а також допомагають у визначенні пріоритетів та керуванні ризиками.

Необхідно збалансувати використання метрик, уникаючи надмірної зосередженості на числах за рахунок якості та інноваційності. Важливо вибирати відповідні метрики, які відображають ключові аспекти проекту та узгоджені з цілями організації.

Метрики в розробці програмного забезпечення є фундаментальним інструментом для забезпечення ефективності, якості та успішності проектів у сфері IT. Вони допомагають структурувати розробку, сприяють забезпеченню прозорості процесів, та стають ключем до безперервного поліпшення та інновацій.

У контексті вимірювання часу до ринку, розглядаємо наступні ключові параметри: тривалість розробки, час тестування, час розгортання, час до першої

відповіді користувача, частота релізів, та час на усунення помилок/дефектів. Кожен із цих параметрів може бути квантифікований(кількісний) за допомогою відповідної математичної формули.

Тривалість розробки оцінюється як середній час від початку до завершення розробки, виражений як:

$$D_{\text{розробки}} = \frac{\sum_{i=1}^{n}(t_{\text{завершення},i} - t_{\text{початку},i})}{n} \quad (2.1)$$

де:

$t_{\text{завершення},i}$ та $t_{\text{початку},i}$ – час початку і завершення проекту.

Час тестування вимірюється як середня кількість часу, витраченого на тестування одного релізу, виражена через:

$$T_{\text{тестуваня}} = \frac{\sum_{j=1}^{m} t_{\text{тестування},j}}{m} \quad (2.2)$$

де:

$t_{\text{тестування},j}$ – час тестування j-го релізу.

Час розгортання визначається як середній час, необхідний для розгортання одного релізу, і обчислюється за формулою:

$$R_{\text{розгортання}} = \frac{\sum_{k=1}^{p} t_{\text{разгортання},k}}{p} \quad (2.3)$$

де:

$t_{\text{разгортання},k}$ – час розгортанняk-го релізу.

Час до першої відповіді користувача вимірюється як середній час між релізом продукту та отриманням першої відповіді від користувача, представлений як:

$$C_{\text{відповіді}} = \frac{\sum_{l=1}^{q} t_{\text{відповіді},l}}{q} \quad (2.4)$$

де:

$t_{\text{відповіді},l}$ – час до отримання першої відповіді для l-го релізу.

Частота релізів визначається кількістю релізів за певний період, зазвичай місяць або квартал, і може бути виражена як:

$$F_{\text{релізи}} = \frac{\text{кількість_релізів}}{\text{період_часу}} \quad (2.5)$$

Час на усунення помилок/дефектів вимірюється як середній час, витрачений на виправлення одного дефекту, і обчислюється за формулою

$$E_{\text{дефекти}} = \frac{\sum_{m=1}^{s} t_{\text{виправлення},m}}{s} \quad (2.6)$$

де:

$t_{\text{виправлення},m}$ – час виправлення m-го дефекту.

Для критерію "Продуктивність", оцінюючи мікросервісну та монолітну архітектуру, можемо використовувати наступні метрики для кожного параметра:

Середній час, який система витрачає на обробку запиту та надання відповіді:

$$T_{\text{відповідь}} = \frac{\sum_{\mathrm{i}=1}^{n} t_{\text{відповідь},i}}{\mathrm{n}} \quad (2.7)$$

де:

$t_{\text{відповідь},i}$ – час відповіді на і-й запит.

Максимальна кількість запитів, які система може обробити за одиницю часу:

$$P_{\text{запити}} = \frac{\text{кількість_запитів}}{\text{час_періоду}} \quad (2.8)$$

Середній час, потрібний для завантаження всіх необхідних ресурсів системою:

$$Z_{\text{ресурси}} = \frac{\sum_{\mathrm{j}=1}^{m} t_{\text{завантаження},j}}{\mathrm{m}} \quad (2.9)$$

де:

$t_{\text{завантаження},j}$ – час завантаження для j-го ресурсу.

Ступінь використання системних ресурсів (наприклад, CPU, пам'ять) для виконання задач:

$$\mathrm{E}_{\text{ресурси}} = \frac{\text{використані_ресурси}}{\text{загальні_ресурси}} \quad (2.10)$$

Здатність системи підтримувати стабільну роботу при збільшенні навантаження визначається експериментально, може включати відсоток успішних запитів під час пікового навантаження.

Здатність системи збільшувати свої ресурси для підтримки високого рівня обслуговування запитів визначається за допомогою тестування масштабування, фіксуючи зміни в часі відповіді та пропускної здатності при збільшенні ресурсів.

Ефективність обробки запитів, виміряна як час відповіді та використання ресурсів:

$$O_{\text{запити}} = \frac{\sum_{\mathrm{k}=1}^{p} t_{\text{відповіді},k}}{\text{потреби_ресурсів}} \quad (2.11)$$

де:

$t_{\text{відповіді},k}$ – час відповіді на k-й оптимізований запит.

Для критерію "Масштабованість", який є ключовим у порівнянні мікросервісної та монолітної архітектур, можна розглянути наступні параметри та відповідні метрики.

Здатність системи підвищувати продуктивність шляхом додавання більше ресурсів до існуючого середовища (наприклад, збільшення CPU або пам'яті):

Здатність системи підвищувати продуктивність шляхом додавання додаткових інстансів або вузлів:

Час, необхідний для виконання масштабування системи:

де:

$t_{\text{початок}}$– час початку процесу масштабування.

Загальні витрати, пов'язані з масштабуванням системи:

де:

$p_{\text{ресурсів}}$– вартість ресурсів;

$p_{\text{реалізації}}$ - вартість реалізації.

Ступінь рівномірності розподілу навантаження між вузлами в масштабованій системі:

Здатність архітектури адаптуватися до змін у навантаженні або вимогах без необхідності повної перебудови. Ця метрика часто оцінюється квалітативно, на основі аналізу архітектури та здатності до швидких змін.

Для критерію "Надійність", який є важливим при оцінці мікросервісної та монолітної архітектур, можна визначити наступні параметри та метрики:

Кількість збоїв, які виникають за певний період часу:

Середній час між послідовними збоями системи:

де:

$t_{\text{між_збоями}}$- час між i-м та (i+1)-м збоєм.

Середній час, необхідний для відновлення системи після збою.

де:

$t_{\text{відновлення}, j}$- час відновлення після j-го збою.

Ступінь, до якого система може витримувати перевантаження або збої, не виходячи з ладу часто оцінюється квалітативно, засновано на тестуванні меж витривалості та аналізі архітектури.

Здатність системи автоматично відновлюватися після збоїв може вимірюватися як відсоток збоїв, які були вирішені автоматично, порівняно з загальною кількістю збоїв.

Обсяг та якість журналювання помилок та моніторингу системи може включати кількісні показники, такі як кількість журналів помилок, покриття моніторингу та частоту перевірок.

Ефективність процесу резервного копіювання та відновлення даних може включати частоту успішних резервних копій та час, необхідний для відновлення даних.

Для критерію "Використання ресурсів" можна розглянути наступні параметри та відповідні метрики.

Процент використання центрального процесора (ЦПУ) системою:

Процент використання оперативної пам'яті системою:

де:

$Mвикористана$ - використана пам’ять;

$M_{закальна}$ - загальна пам’ять;

$M_{використання}$ - використання пам’яті.

Процент використання дискового простору системою:

де:

$U_{використаний_дисковий_простір}$- використанний дисковий простір;

$S_{дискового_простору}$ – загальний дисковий простір.

Для критерію "Безпека" можна використовувати наступні параметри та метрики.

Оцінка рівня захисту даних на основі використання шифрування, контролю доступу та інших заходів безпеки - квалітативна оцінка, може включати відсоток зашифрованих даних, кількість випадків витоку даних тощо.

Ефективність захисту від зловмисних атак, таких як DDoS, SQL ін'єкції та інше - процент успішно відбитих атак від загальної кількості спроб.

Обсяг і глибина аудиту та моніторингу безпеки може включати кількість аудитів, частоту моніторингу, відсоток виявлених вразливостей тощо.

Наявність та дієвість політик безпеки та процедур реагування на інциденти - квалітативна оцінка, може включати аналіз політик, процедур та часу відповіді на інциденти.

Ступінь комплексності та безпеки систем ідентифікації та авторизації може включати оцінку рівнів доступу, методів аутентифікації, політик паролів тощо.

Частота та своєчасність випуску оновлень безпеки та патчів:

$$T_{оновлення} = \frac{N_{оновлень}}{t_{періоду}}$$

де:

$t_{періоду}$- час періоду;

$N_{оновлень}$ – кількість оновлень.

Наявність та дотримання міжнародних стандартів та сертифікацій безпеки: кількість наявних сертифікацій, відсоток відповідності стандартам.

$T_{оновлення}$ регулярність оновлень:

2.3 Створення моделі оцінювання

У процесі розробки методу порівняльного аналізу мікросервісної та монолітної архітектур важливе місце займає створення моделі оцінювання. Ця модель є фундаментальним інструментом, який дозволяє систематизувати процес оцінки та забезпечує структурований підхід до аналізу та порівняння обох архітектур.

Головна мета створення такої моделі полягає у встановленні об'єктивних критеріїв та метрик для оцінки визначених параметрів, що дозволяє робити виважені та обґрунтовані висновки. Модель оцінювання повинна бути гнучкою, щоб враховувати різноманітність сценаріїв використання та можливі зміни в технологічному ландшафті.

Створення моделі оцінювання передбачає глибоке розуміння як мікросервісної, так і монолітної архітектур, а також включає в себе аналіз ключових аспектів, таких як продуктивність, масштабованість, надійність, використання ресурсів та безпека. Це вимагає інтеграції даних з різних джерел, включаючи емпіричні дані, експертні оцінки та літературні джерела [23-24].

Модель оцінювання служить як основа для здійснення об'єктивного порівняльного аналізу, надаючи не тільки кількісні, але й якісні оцінки, які є невід'ємною частиною наукового дослідження у цій області.

Спочатку було розроблено моделі для кожного з критерію а потім розроблено цілісну модель.

Створення математичної моделі для оцінки критерію "Час до ринку" вимагає інтеграції визначених метрик у єдину формулу або набір формул, які дозволять кількісно оцінити та порівняти мікросервісну та монолітну архітектури. Ось приклад такої моделі:

$$T_{періоду} = w1 * D_{розробки} + w2 * T_{тестування} + w3 * R_{розгортання} + w4 * С_{відповідь} + ...$$

де:

$D\text{розробки}$- середній час розробки;
$T\text{тестування}$- середній час тестування;

$R_{\text{розгортання}}$ – середній час разгортання;

$C\text{відповідь}$- середній час до першої відповіді користувача;
$F\text{релізи}$- частота релізів;
$E\text{дефекти}$- середній час на усунення помилок/дефектів;
$w1, 2, \dots, 6$- вагові коефіцієнти для кожної метрики.

Кожна метрика в моделі має відповідну вагу, яка відображає її важливість у загальному оцінюванні часу до ринку.

Вагові коефіцієнти можуть бути визначені на основі аналізу важливості кожного аспекту часу до ринку або можуть бути адаптовані для конкретного проекту або організації.

Ця модель дозволяє отримати кількісну оцінку загального часу до ринку, яка може бути використана для порівняння між мікросервісною та монолітною архітектурами.

Аналогічну математичну модель можливо створити для критерію «Продуктивність»:

$=$

де:

$T\text{відповіді}$- час відповіді системи;

$P_{\text{пропускна}}$ – пропускна здатність;

$Z_{\text{ресурси}}$ – час завантаження ресурсів;

$O\text{запити}$- оптимізація запитів;

$w_{1,2,\dots,4}$- вагові коефіцієнти для кожної метрики.

Модель дозволяє отримати кількісну оцінку загальної продуктивності, що є корисним для порівняння між різними архітектурами.

Також була розроблена математична модель для критерію «Масштабованість». Вона виглядає наступним планом:

$=$

$$w1* V\text{масштабування}+w2* H\text{масштабування}+ w3* T\text{масштабування}+ w4*$$

де:

$V\text{масштабування}$- вертикальна масштабованість;
$H\text{масштабування}$- горизонтальна масштабованість;

$T_{\text{масштабування}}$ – час на масштабування;

$C\text{масштабування}$- вартість масштабування;
$E\text{розподіл}$- Ефективність розподілу навантаження;

$w_{1,2,\dots,5}$- вагові коефіцієнти для кожної метрики.

Ця модель дозволяє отримати загальну кількісну оцінку масштабованості, яка може бути використана для порівняння між різними архітектурами.

Далі було створено математичну модель для критерію «Надійність».

де:

$F_{збої}$- частота збоїв;

$MTBR$ – (Mean Time Between Failures);

час до відновлення після збою (Mean Time To Recovery);

$w_{1,2,\dots,3}$- вагові коефіцієнти для кожної метрики.

Метрики відображають різні аспекти надійності, включаючи частоту та вплив збоїв, здатність системи до швидкого відновлення та забезпечення безперебійної роботи.

Далі було створено математичну модель для критерію «Використання ресурсів».

де:

$U_{ЦПУ}$- використання ЦПУ;

$U_{пам'ять}$- використання пам'яті;

$U_{дисковий_простір}$–використання дискового простору;

$w_{1,2,3}$- вагові коефіцієнти для кожної метрики.

Модель дозволяє кількісно оцінити та порівняти загальне використання ресурсів між різними архітектурами.

За аналогією було створено модель для критерію «Безпека».

$$w1* S_{дані}+w2* S_{атаки}+ w3* A_{безпека}+ w4* P_{політика}+ w5$$

де:

$S_{дані}$ – рівень захисту даних;

$S_{атаки}$- заходи проти зловмисних атак;

$A_{безпека}$–аудит та моніторінг безпеки;

$P_{політика}$- політики безпеки та процедури відповіді на інциденти;

$I_{ідентифікація}$ – комплексність системи управління ідентифікацією та авторизацією;

$U_{оновлення}$- регулярность оновлень безпеки та патчів;

$C_{сертифікації}$ – сертифікації та стандарти безпеки;

$w_{1,2,\dots,7}$- вагові коефіцієнти для кожної метрики.

Після цього було розроблено загальну формулу оцінку архітектури:

де:

оцінка «Час до ринку»;
оцінка «Продуктивність»;
TMF — оцінка «Масштабованість»;

оцінка «Надійність»;
оцінка «Використання ресурсів»;
α
оцінка «Безпека»;
1
Ця спільна модель є потужним інструментом для оцінки та порівняння мікросервісних та монолітних архітектур, враховуючи різні важливі аспекти їх функціонування.

2
,

2.4 Планування експерементального дослідження

...

Мета експерименту полягає у валідації та перевірці ефективності розробленої спільної математичної моделі, яка оцінює архітектури з точки зору Часу до ринку, Продуктивності, Масштабованості, Надійності, Використання ресурсів та Безпеки. Ключовим завданням експерименту є визначення того, наскільки точно та об'єктивно модель може відображати реальні переваги та недоліки мікросервісної та монолітної архітектур у різних умовах та сценаріях.

6
вагові коефіцієнти для кожного критерію.

Це включає перевірку здатності моделі коректно оцінювати різні архітектури: Переконатися, що модель ефективно розрізняє між мікросервісними та монолітними архітектурами на основі важливих метрик.

Також треба, щоб модель відображала реальні умови, для цього потрібно перевірити, чи результати, отримані з моделі, відповідають дійсним даним і спостереженням у реальних умовах експлуатації систем.

Ще одним пунктом треба підтвердити надійність та відтворюваність, тобто довести, що модель є надійною та її результати можна відтворити при повторних тестуваннях в аналогічних умовах.

Були виділені наступні гіпотези:

ульова гіпотеза точності моделі (H0): модель не забезпечує точного оцінювання архітектур з точки зору Часу до ринку, Продуктивності, Масштабованості, Надійності, Використання ресурсів та Безпеки;

льтернативна гіпотеза точності моделі (H1): модель точно оцінює архітектури за зазначеними критеріями;

ульова гіпотеза порівняння архітектур (H0): між мікросервісною та монолітною архітектурами немає статистично значущої різниці за оцінками моделі;

льтернативна гіпотеза порівняння архітектур (H1): існує статистично значуща різниця між мікросервісною та монолітною архітектурами за оцінками моделі.

У рамках цього дослідження було вирішено взяти за основу додаток для служби доставки в ресторані, розроблений власноруч. Даний додаток унікальний тим, що він створений як дві окремі інформаційні системи. Перша система розроблена з використанням монолітної архітектури, де всі компоненти - доставка, управління замовленнями, обробка платежів та інтерфейс користувача - інтегровані в одне цілісне програмне забезпечення. Це дозволяє зосередитись на простоті розробки та розгортання. Друга система, в свою чергу, розроблена з використанням мікросервісної архітектури, де функціонал розділений на декілька незалежних сервісів, кожен з яких відповідає за свою функціональну область і спілкується з іншими через визначені інтерфейси, що сприяє гнучкості та масштабуванню.

Обраний підхід до дослідження надає унікальну можливість для безпосереднього порівняння монолітної та мікросервісної архітектур в контексті реального застосування. Крім того, оскільки обидві системи розроблені для однакових бізнес-процесів, це створює ідеальні умови для об'єктивного аналізу. Використання власноруч розроблених систем дозволяє отримати детальне розуміння внутрішньої структури та функціональності, що є важливим для глибокого аналізу їхньої ефективності.

Для збору та аналізу даних у цьому дослідженні будуть використані статистичні методи аналізу. Це дозволить об'єктивно оцінити продуктивність, надійність та інші ключові показники обох систем, забезпечуючи точне та вірогідне порівняння їх ефективності. Використання статистичних методів також забезпечить наукову точність у висновках та рекомендаціях, виведених із дослідження.

Очікується, що через це дослідження буде отримано вичерпні дані про переваги та недоліки кожної архітектури у контексті конкретної доменної області. Це дозволить зробити обґрунтовані висновки щодо оптимального вибору архітектури для подібних майбутніх проектів.

3 РОЗРОБКА ЕКСПЕРЕМЕНТАЛЬНИХ ІС

3.1 Опис предметної області

Для проведеня експерименту було обрано розробити ІС доставки їжі для ресторану.

Розробка інформаційної системи для доставки в ресторані є важливою та актуальною у сучасному бізнес-середовищі, особливо з огляду на зростаючу популярність онлайн-замовлень та доставки їжі. Така система дозволяє ресторанам значно підвищити ефективність, оптимізувавши процеси замовлення та доставки, що в свою чергу скорочує час очікування для клієнтів та покращує використання ресурсів. Це також сприяє покращенню якості обслуговування, оскільки автоматизація зменшує ймовірність помилок, які можуть виникати при ручному введенні даних.

Системи збору та аналізу даних можуть надавати цінну інформацію про поведінку споживачів та попит на певні страви, дозволяючи ресторанам оптимізувати меню та маркетингові стратегії. Це не лише збільшує доходи за рахунок приваблення більшої кількості клієнтів, але й створює конкурентні переваги, особливо в міських районах з високим рівнем конкуренції.

Адаптація до змін у споживацьких звичках та перевагах, особливо з урахуванням поширення мобільних технологій та інтернету, є ключовою. Клієнти все частіше віддають перевагу цифровим каналам для замовлення продуктів і послуг, у тому числі їжі. Також важливою є гнучкість і можливість швидкої адаптації до змінних ринкових умов, як-от пандемії або економічні коливання.

Крім того, ефективні системи доставки можуть допомогти зменшити викиди вуглецю завдяки оптимізації маршрутів доставки, що є важливим з точки зору екологічної відповідальності. Загалом, розробка такої системи вимагає глибокого розуміння потреб ринку та технологічних можливостей, але її потенціал для покращення бізнес-процесів та збільшення прибутків є значним.

Додатки для доставки їжі від ресторанів включають ряд ключових функціональних можливостей, спрямованих на забезпечення зручності та ефективності як для клієнтів, так і для бізнесу.

Додатки дозволяють користувачам переглядати різноманітні меню, включаючи детальні описи страв, ціни та фотографії. Користувачі можуть швидко знайти бажані страви або ресторани, використовуючи фільтри за типом кухні, рейтингу, часу доставки та іншим критеріям.

Додатки надають можливість здійснення замовлення в кілька кліків, з можливістю вибору різних опцій (наприклад, рівень гостроти блюда, інгредієнти тощо). Оплата замовлень здійснюється через інтегровані платіжні системи, що підтримують кредитні картки, електронні гаманці та інші платіжні методи.

Користувачі можуть в режимі реального часу відслідковувати статус своїх замовлень, від моменту прийняття рестораном до часу доставки.

Сучасні додатки часто використовують алгоритми машинного навчання для надання персоналізованих рекомендацій страв і ресторанів на основі попередніх замовлень та переваг користувача.

Користувачі мають можливість залишати відгуки та оцінки стосовно якості страв, рівня обслуговування та часу доставки, що сприяє підвищенню прозорості та допомагає іншим користувачам у виборі ресторанів.

Часто додатки пропонують можливості швидкого ділення досвіду або улюблених страв у соціальних мережах.

Користувачі можуть керувати своїми обліковими записами, зберігаючи інформацію про адреси доставки, улюблені страви, історію замовлень та персональні дані.

Ці функції не тільки підвищують зручність та задоволеність клієнтів, але й сприяють оптимізації бізнес-процесів ресторану, включаючи управління запасами, аналіз продажів та маркетинг. В сукупності це створює потужний інструмент для зростання та розвитку бізнесу в галузі громадського харчування.

3.2 Опис основних бізнес процесів ІС

На початку розгляду основних бізнес процесів було створено діаграму IDEF0. Діаграми IDEF0, або Integrated DEFinition for Function Modeling, відіграють ключову роль у розробці та аналізі бізнес-процесів, завдяки їх здатності візуалізувати та структурувати складні процеси. Ці діаграми дозволяють розділити процеси на менші, легко управлінні частини, що сприяє кращому розумінню та ефективності в компаніях. Однією з ключових переваг є стандартизація документації процесів, що полегшує комунікацію між різними відділами та співробітниками.

Крім того, діаграми IDEF0 є потужним інструментом для аналізу та оптимізації, дозволяючи ідентифікувати зайві кроки, вузькі місця та можливості для покращення. Вони сприяють управлінню змінами в рамках процесу неперервного вдосконалення та служать важливим засобом для навчання нових співробітників. Ці діаграми можуть ефективно інтегруватися з іншими методами управління якістю та проектного менеджменту, такими як Lean та Six Sigma, та сприяють більш обгрунтованому прийняттю управлінських рішень. У цілому, діаграми IDEF0 є ключовим елементом у розробці та вдосконаленні бізнес-процесів, забезпечуючи глибше розуміння та ефективність в їх управлінні.

На рис 3.1 можна переглянути діаграму IDEF0 для розроблювальної ІС.

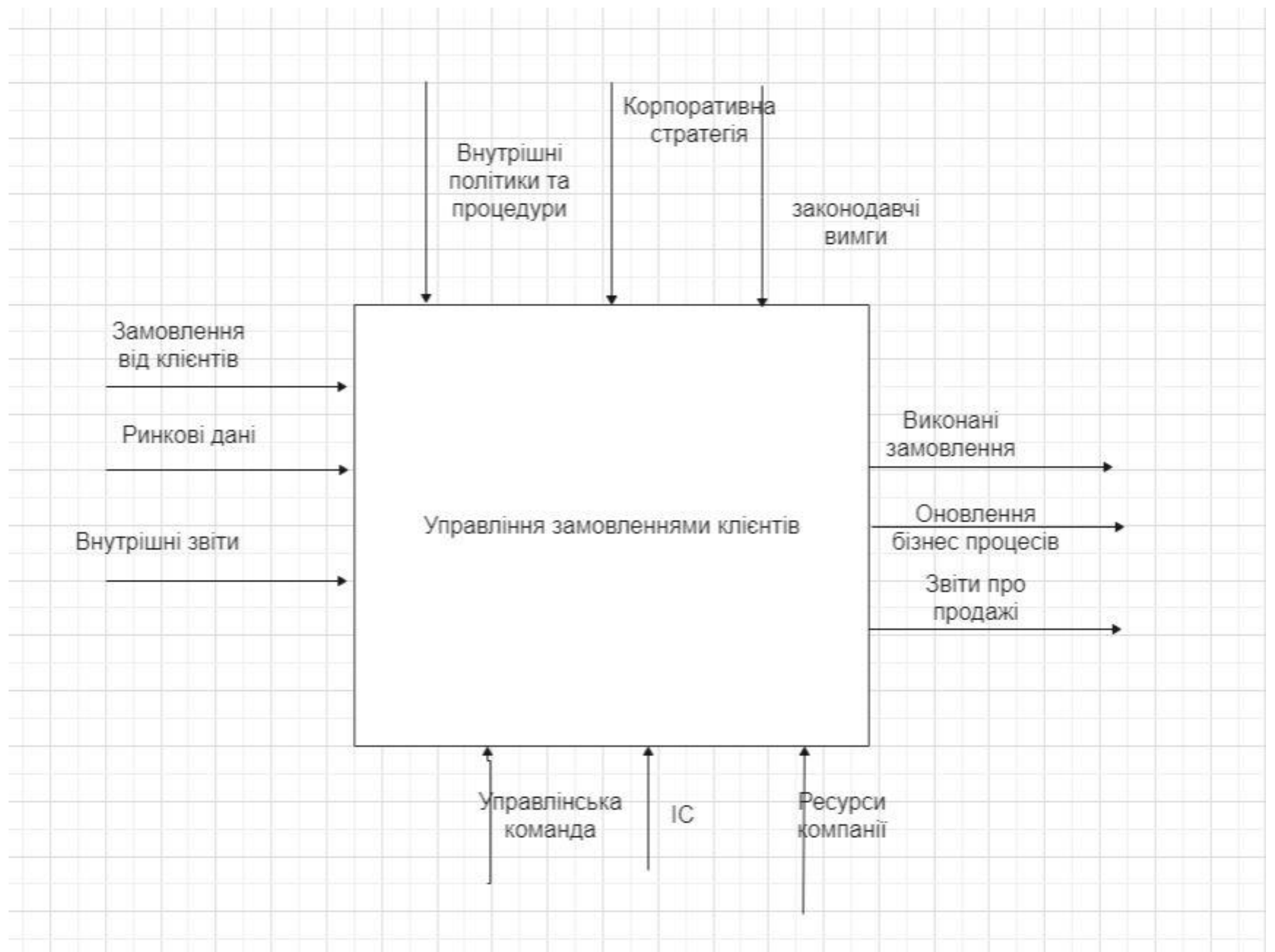

Рис. 3.1 – IDEF0 розроблювальної системи

Діаграма містить наступні елементи.

Основний блок (A0) - Управління замовленнями клієнтів:

Вхід:

Замовлення від клієнтів»: включає деталі продуктів, кількості, вимоги до доставки;

Ринкові дані»: включає інформацію про попит, конкурентний аналіз, тенденції споживання;

Внутрішні звіти»: включає даних про стан запасів, ефективність виробництва, історичні дані замовлень.

Вихід:

Виконані замовлення»: остаточні продукти, доставлені та задоволені потреби клієнтів;

Звіти про продажі та доставку»: документація щодо обсягу продажів, ефективності доставки, зворотнього зв'язку клієнтів;
Оновлення бізнес-процесів»: рекомендації щодо покращення процесів, засновані на аналізі продуктивності.

Механізм:
Управлінська команда»: керівники, що відповідають за прийняття стратегічних рішень;
Інформаційні системи»: ERP-системи, CRM, системи аналітики даних;
Ресурси компанії»: фінансові, людські, технічні ресурси для підтримки процесів.

Керівництво:
Корпоративна стратегія»: загальна місія та цілі компанії, довгострокові плани;
Внутрішні політики та процедури»: правила управління замовленнями, стандарти обслуговування, політика якості;
Законодавчі та регуляторні вимоги»: дотримання законів та стандартів, що стосуються торгівлі, доставки, конфіденційності.

Далі було створено Use Case діаграму

Діаграми випадків використання (Use Case Diagrams) відіграють ключову роль у процесі аналізу та проектування систем, особливо в рамках об'єктно-орієнтованого підходу. Вони служать для візуалізації функціональних вимог системи, показуючи, як зовнішні користувачі (актори) будуть взаємодіяти з нею. Це спрощує комунікацію між розробниками, аналітиками та клієнтами, надаючи зрозумілий і доступний спосіб висвітлення ключових функцій та можливостей системи.

Через діаграми можна ефективно ідентифікувати основні функціональні вимоги системи, враховуючи всі важливі потреби користувачів. Вони також важливі для планування процесу розробки, допомагаючи визначити пріоритети різних випадків використання та обсяг роботи. Крім того, діаграми є корисним інструментом при розробці сценаріїв тестування, забезпечуючи всебічну перевірку ключових функцій системи.

На ранніх етапах проектування вони також допомагають виявляти та вирішувати потенційні конфлікти у вимогах, що дозволяє уникнути проблем у майбутньому розвитку проекту. В цілому, Use Case діаграми є ефективним засобом для забезпечення відповідності системи потребам користувачів та бізнес-цілям.

На рис. 3.2 можна переглянути розроблену Use Case діаграму.

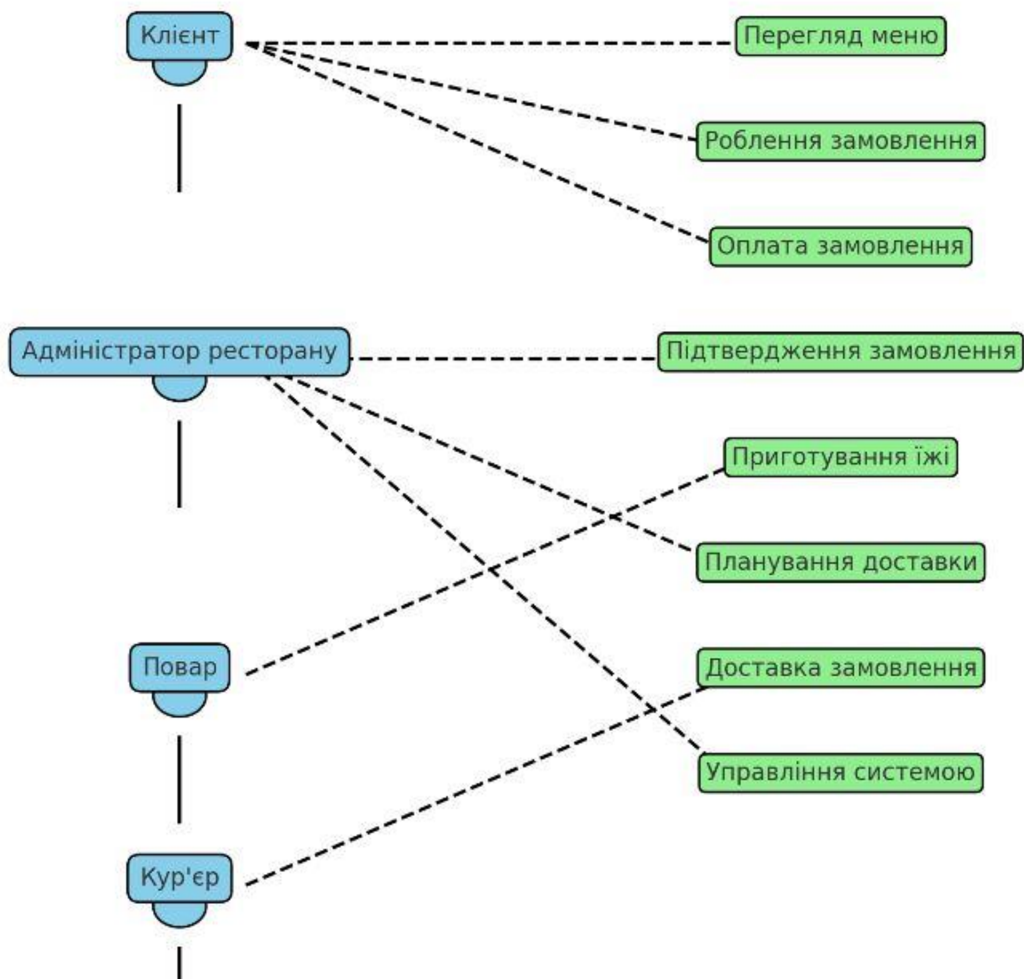

Рис 3.2 – Діаграма Use Case

Після цього було створено діаграму послідовності.

Діаграма послідовності є одним з типів діаграм, які використовуються в рамках Unified Modeling Language (UML) для візуалізації взаємодії між об'єктами у певному бізнес-процесі або сценарії. Ця діаграма зосереджується на показі того, як об'єкти спілкуються між собою через послідовність повідомлень

або дій у часі для виконання конкретного завдання або досягнення цілі. На діаграмі кожен об'єкт представлений у вигляді вертикальної лінії, відомої як лінія життя, а горизонтальні стрілки між цими лініями вказують на повідомлення або дії, що відбуваються між об'єктами. Вертикальна організація діаграми служить як часова вісь, ілюструючи порядок подій від верху до низу.

Діаграма послідовності має значну цінність у процесі аналізу бізнес-процесів, оскільки вона надає чітке візуальне представлення складних взаємодій. Вона допомагає ідентифікувати залежності та потенційні затримки, що сприяє ефективному плануванню та оптимізації процесів. Крім того, діаграма послідовності є корисним інструментом для поліпшення комунікації між різними учасниками проекту, такими як розробники, аналітики та зацікавлені сторони. Це також цінний ресурс для створення тестових сценаріїв та аналізу потенційних проблем, що можуть виникнути під час реалізації процесу. В цілому, діаграми послідовності сприяють підвищенню якості проектування та розробки програмного забезпечення, забезпечуючи більш ефективний і зрозумілий підхід до візуалізації бізнес-процесів.

Розроблену діаграму можна переглянути на рис 3.3. Діаграма показує послідовність взаємодій між різними акторами (Клієнт, Система, Адміністратор, Повар, Кур'єр) та як ці взаємодії відбуваються у часі від моменту замовлення клієнтом до доставки їжі кур'єром.

Кожна стрілка представляє повідомлення або дію, яка переходить від одного актора до іншого, ілюструючи взаємодію та потік процесу. Ця діаграма допомагає зрозуміти, як обробляються замовлення в системі, та визначити ключові точки взаємодії.

Ось перелік процесів у системі при управлінні замовленнями клієнта:
лієнт робить замовлення через інтерфейс системи;
истема обробляє замовлення та пересилає його адміністратору ресторану;
дміністратор ресторану перевіряє та підтверджує замовлення;
ідтверджене замовлення відправляється до повара для приготування;

ісля приготування, повар відправляє повідомлення про готовність страви до адміністратора;
дміністратор планує доставку і надсилає інформацію про замовлення кур'єру;
ур'єр доставляє їжу клієнту.

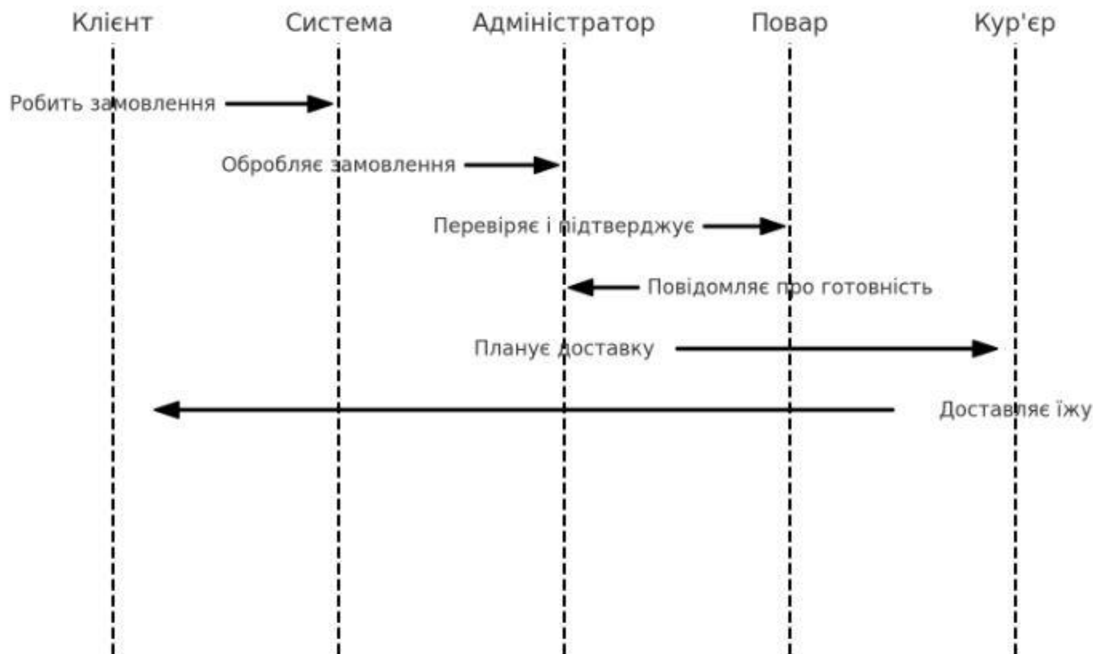

Рис. 3.3 – Діаграма послідовності

3.3 Проектування і розробка компонентів ІС

У ході реалізації експериментального дослідження, спрямованого на порівняння мікросервісної та монолітної архітектур, було вирішено розпочати з розробки системи, що базується на монолітній архітектурі.

В процесі розробки монолітної системи була використана класична трьохрівнева модель, що включає презентаційний рівень, бізнес-логіку та рівень даних. Дана модель була обрана через її здатність забезпечити чітке розмежування між різними компонентами системи, що сприяє легкості внесення змін та підтримці.

Архітектура розробленої монолітної системи демонструє централізовану структуру, де всі компоненти тісно взаємопов'язані та взаємодіють між собою.

Ця структура була вибрана з метою забезпечення високої продуктивності та надійності для конкретного типу застосувань, які не вимагають широкого масштабування або великої гнучкості. Важливо відзначити, що монолітна архітектура забезпечує простоту розгортання та управління, оскільки вся система розгортається як єдиний блок.

Детальна модель архітектури монолітної системи представлена на рисунку 3.4.

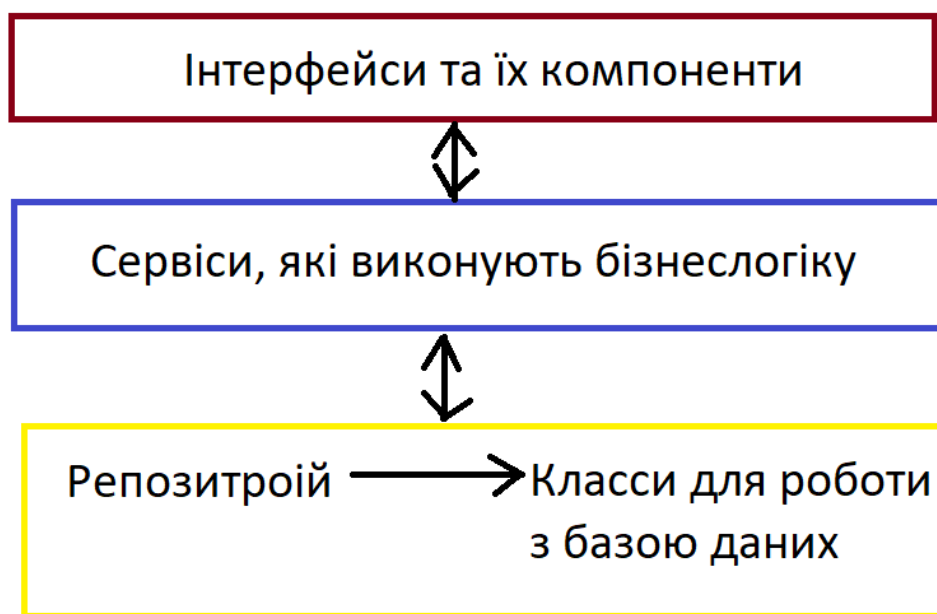

Рис. 3.4 – Модель монолітної архітектури

В рамках проектування монолітної системи особлива увага була приділена розробці ефективної моделі бази даних, ключовим елементом якої є Entity-Relationship (ER) діаграма. ER-діаграма була розроблена для забезпечення чіткого візуального представлення структури даних, що використовуються в системі, та їх взаємозв'язків. Це інструмент, який дозволяє систематизувати дані та оптимізувати процеси їх зберігання та обробки.

ER-діаграма для монолітної системи включає основні сутності, такі як користувачі, транзакції, продукти та інші необхідні елементи, які є частиною

бізнес-процесів системи. Зв'язки між сутностями чітко визначені, щоб забезпечити цілісність та неперервність даних. Така структура допомагає уникнути дублювання даних та сприяє більш ефективному доступу та управлінню інформацією.

Важливим аспектом ER-діаграми є її здатність забезпечувати легке розуміння структури бази даних навіть для осіб, які не є експертами у сфері IT. Це особливо важливо для команди розробників, оскільки воно дозволяє швидко адаптуватися до проекту та ефективно співпрацювати над ним.

ER-діаграма також відіграє ключову роль у визначенні та оптимізації запитів до бази даних, що є критично важливим для підтримки високої продуктивності системи. Вона дозволяє точно визначати, як дані будуть пов'язані та яким чином ці зв'язки будуть використані в різних частинах системи.

Таким чином, ER-діаграма стала не лише засобом візуалізації структури бази даних, а й важливим інструментом для ефективного проектування та реалізації монолітної системи. Її використання дозволило досягти високої координації між різними компонентами системи та оптимізувати обробку даних.

Розроблену ER можливо переглянути на рисунку 3.5. На ній вміщено мінімально необхідну інформацію яку треба зберігати в рамках інформаційної системи. В ній міститься таблиця користувачів, яка має інформацію про кухарів, клієнтів, кур'єрів, адміністраторів. Також для розрізняння користувачів системи використовується допоміжна таблиця ролей. В ІС також зберігається інформація про блюда там, та для зручності є додаткова таблиця для поділення блюд на окремі меню. Зв'язок багато до багатьох розв'язується за допомогою таблиці замовлення, де ще є і адреса для доставки.

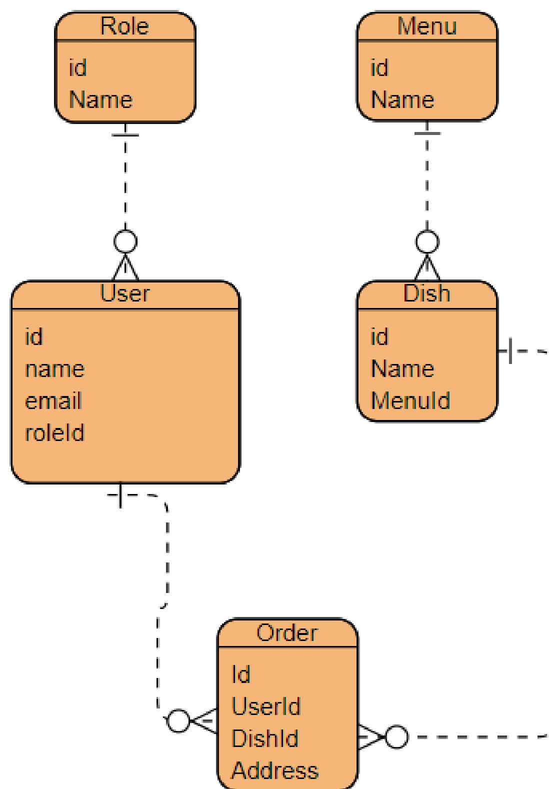

Рис. 3.5 – Пілотна ER –діаграма

При розробці мікросервісної архітектури розділяється одна система на набір менших служб, кожна з яких працює в своєму власному процесі та спілкується через легкі механізми, такі як HTTP API. Цей підхід починається з ідентифікації окремих доменних областей бізнес логіки та створення окремого мікросервісу для кожної функції, таких як обслуговування користувачів, обробка замовлень, управління меню, платежів та доставки. Кожен мікросервіс є автономним та може мати свою власну базу даних, що дозволяє досягати високого рівня розподілу та ізоляції. Взаємодія з мікросервісами здійснюється через API Gateway, який служить єдиною точкою входу та направляє запити до відповідних сервісів. Щоб сервіси могли легко виявляти один одного, використовується механізм виявлення служб. Балансувальники навантаження забезпечують розподіл запитів між екземплярами мікросервісів, підвищуючи надійність та доступність. Конфігураційний сервер централізує управління конфігурацією, а служби автентифікації та авторизації підтримують безпеку

системи. Централізовані служби логування, моніторингу та трасування спрощують відстеження проблем та оцінку ефективності системи. Окрім того, кожний мікросервіс потрібно документувати, щоб забезпечити зрозумілість і спростити інтеграцію між сервісами. Мікросервісна архітектура може бути складною у розробці та управлінні, проте вона пропонує значні переваги у гнучкості, масштабованості та відмовостійкості системи.

При аналізі предметної області були виділені наступні мікросервіси:
Управління замовленнями» - відповідає за прийом замовлень від клієнта, відстеження статусу замовлення. З сервісом доставки та кухні є зв'язок;
Управління користувачами» - відповідає за реєстрацію та авторизацію користувачів, управління профілями клієнтів та зберігання про них інформації;
Меню» - відповідає за перелік страв і напоїв та управління ним, оновленням цін та описом страв, також відповідає за категоризацію меню;
Управління доставкою» - відповідає за визначення маршрутів доставки, відстреження кур'єрів у реальному часі, управління графіками доставки;
Звітність та Аналітика» - відповідає за збір даних про продажі та відгуки клієнтів, аналіз ефективності роботи ресторану та сервісу доставки;
Інтеграція з платіжними системами» - відповідає за обробку платежів, інтеграцію з іншими платіжними системами, забезпечення безпеки транзакцій.

Кожен мікросервіс розроблений таким чином, щоб бути незалежним, легко масштабованим та легко інтегрованим з іншими сервісами. Це забезпечить гнучкість та ефективність системи. Важливо також розглянути впровадження API Gateway для спрощення взаємодії між різними мікросервісами та забезпечення безпеки системи.

Схему того як влаштований API Gateway показано на рис. 3.6. На ній зображенні наступні елементи:
- мобільний пристрій чи інший клієнтський застосунок, який надсилає запити до API Gateway;
блок, що відображає процес зняття шифрування SSL/TLS з запитів до API

Gateway, що може бути використано для покращення продуктивності шляхом зменшення навантаження на шифрування на сервері;
центральний компонент схеми, який приймає всі вхідні запити від клієнтських застосунків і управляє їхньою подальшою обробкою;
 - сервіс, який забезпечує аутентифікацію користувачів. Цей компонент взаємодіє з API Gateway для верифікації ідентифікаційних даних користувача;
 - процес верифікації користувача, що здійснюється перед перенаправленням запиту до відповідного сервісу;
 - процес визначення маршруту запиту від API Gateway до конкретного мікросервісу;
різні мікросервіси, які обробляють певні аспекти бізнес-логіки або функціональності;
механізм, який дозволяє зберігати і повторно використовувати відповіді API для підвищення швидкості відгуку і зменшення навантаження на сервіси;
 - запис подій та інформації про запити і відповіді, що може використовуватися для аналізу та відстеження.

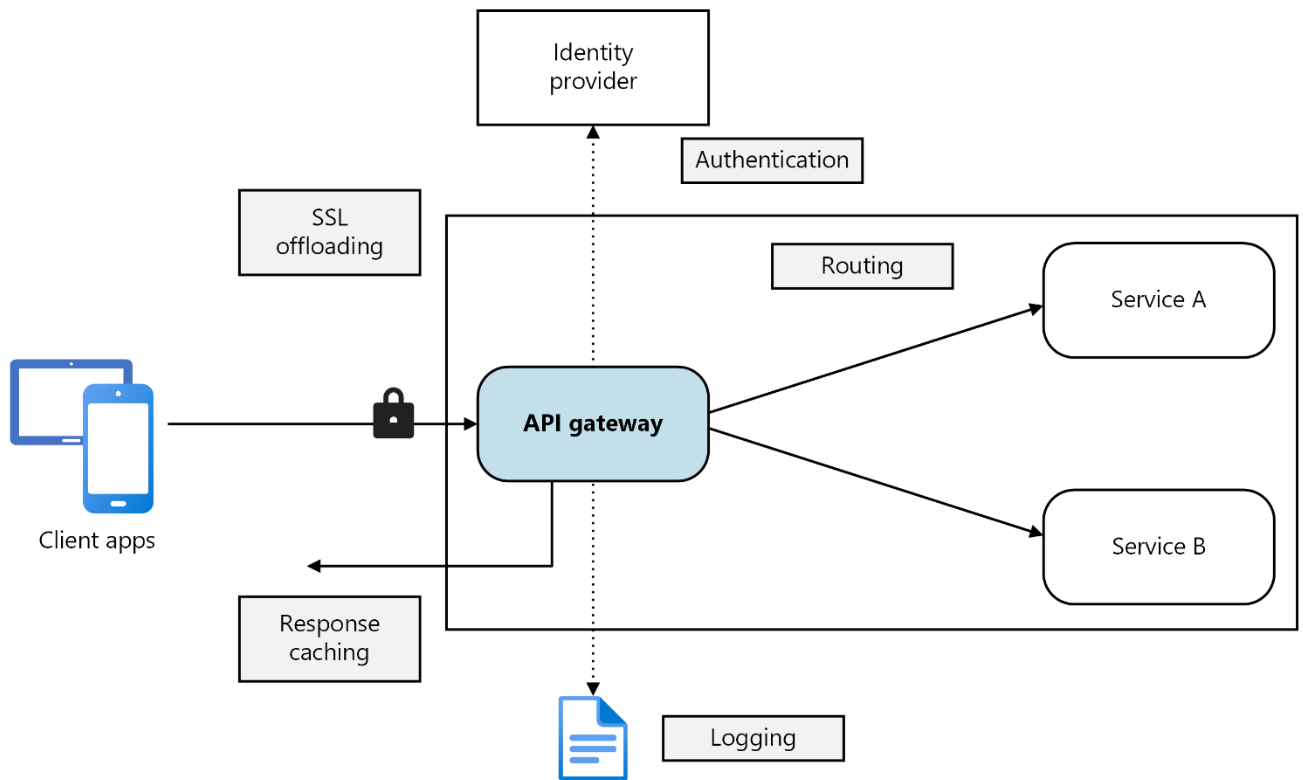

Рис. 3.6 – Схема API Gateway

Після розгортання API Gateway, який слугує центральним вузлом для обробки та маршрутизації запитів у нашій мікросервісній архітектурі, було вирішено почати наступний етап: розробку оптимізованих баз даних для кожного мікросервісу. Цей крок життєво важливий для забезпечення ефективного зберігання та доступу до даних, специфічних для кожного сервісу, таких як управління замовленнями, користувачами, меню та доставкою. Особлива увага приділялася розробці ER-схем, які детально відображають структуру та взаємозв'язки даних в кожній базі. Це дозволило нам ефективно ізолювати та управляти даними в масштабованому середовищі, забезпечуючи високу продуктивність та гнучкість системи. Кожна база даних була спроектована з урахуванням специфічних вимог мікросервісу, забезпечуючи таким чином оптимальну підтримку бізнес-процесів та операцій.

Спочатку було розроблено БД для мікросервісу доставки. На рисунку 3.7 можна побачити діаграму ER для нього.

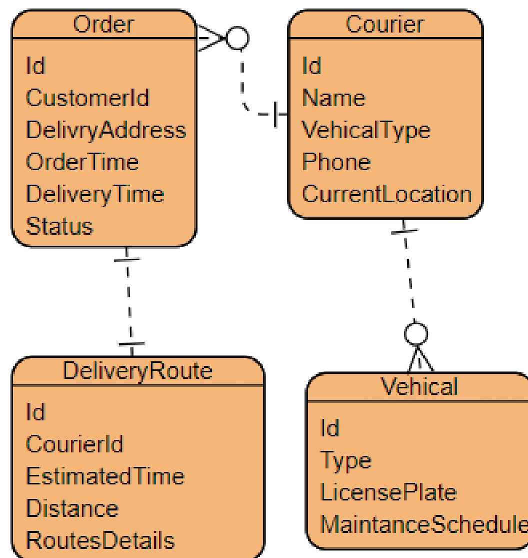

Рис. 3.7 – ER діаграма сервісу доставки

На наведеній діаграмі ER представлено структуру бази даних, призначеної для управління доставкою в системі доставки їжі. Діаграма включає чотири основні сутності: Order (Замовлення), Courier (Кур'єр), DeliveryRoute (Маршрут Доставки) та Vehicle (Транспортний Засіб). Кожна сутність має унікальний ідентифікатор (Id) та набір атрибутів, які описують її характеристики. Замовлення містить дані про клієнта, адресу доставки, час замовлення та доставки, а також статус замовлення. Кур'єр характеризується ім'ям, типом транспортного засобу, контактним телефоном та поточним місцезнаходженням. Маршрут Доставки визначається оціненим часом доставки, дистанцією та деталями маршруту, що пов'язує його з конкретним кур'єром. Транспортний засіб описується типом, номерним знаком та графіком обслуговування.

Зв'язки між сутностями показують, що один кур'єр може бути відповідальний за декілька замовлень, а також що для кожного замовлення існує один маршрут доставки. Також ілюструється, що один транспортний засіб

асоційований з одним кур'єром. Стрілки з ключами біля сутностей вказують на тип зв'язку: "один до багатьох" або "один до одного". Наприклад, кожне замовлення має один маршрут доставки, але один кур'єр може мати кілька замовлень. Така структура бази даних дозволяє ефективно управляти процесом доставки, оптимізувати маршрути та координувати роботу кур'єрів.

Після цього було розроблено БД для мікросервісу, що відповідає за меню. Розроблену діаграму можна подивитися на рисунку 3.8.

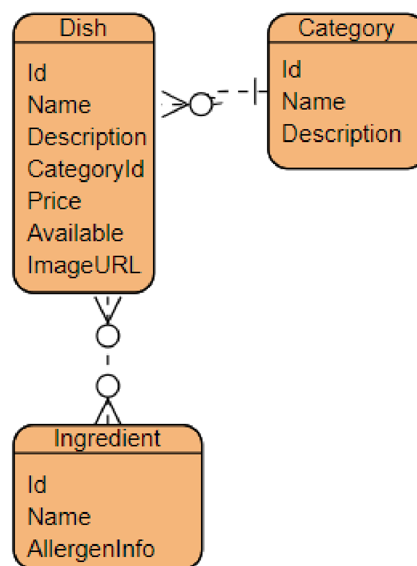

Рис. 3.8 – ER діаграма сервісу меню

Діаграма представляє ER-модель для системи управління меню в рамках послуги доставки їжі. Вона складається з трьох основних сутностей: "Dish" (Страва), "Category" (Категорія) і "Ingredient" (Інгредієнт), кожна з яких містить ряд атрибутів. Страва описується ідентифікатором, назвою, описом, ідентифікатором категорії, ціною, інформацією про наявність та URL зображення. Категорія включає ідентифікатор, назву та опис. Інгредієнт має ідентифікатор, назву та інформацію про алергени.

Замовлення містить відношення "багато до одного" (Many-to-One) з Категорією, вказуючи на те, що багато страв можуть бути класифіковані в одну категорію. Відносини між Стравою та Інгредієнтом є "багато до багатьох" (Many-

to-Many), що означає, що кожна страва може містити багато інгредієнтів, і один інгредієнт може використовуватися у багатьох стравах. Ця взаємодія зазвичай вимагає додаткової зв'язуючої таблиці, яка не показана на діаграмі, але необхідна для управління такими відносинами.

Ця ER-схема дозволяє ресторанам або службам доставки їжі керувати своїм меню, розподіляючи страви по категоріях для зручності та організації, а також відстежувати, які інгредієнти використовуються в кожній страві, що є критично важливим для клієнтів з алергією або специфічними дієтичними обмеженнями.

Після цього було створено БД для сервісу з управління замовленнями. Розроблена діаграма ER можна побачити на рисунку 3.9.

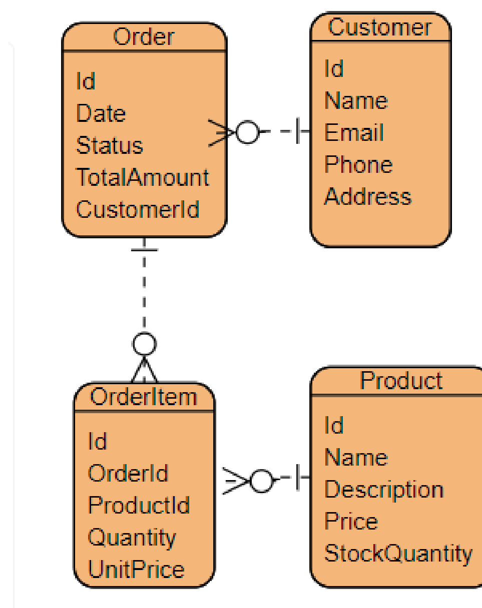

Рис. 3.9 – ER діаграма сервісу управління замовленнями

На ER-діаграмі представлено структуру бази даних для управління замовленнями. Діаграма ілюструє чотири основні сутності: "Order" (Замовлення), "Customer" (Клієнт), "OrderItem" (Позиція Замовлення) та "Product" (Продукт), які взаємопов'язані.

Сутність "Order" відображає інформацію про замовлення, включаючи унікальний ідентифікатор, дату, статус, загальну суму замовлення та

ідентифікатор клієнта, що здійснив замовлення. Вона має відношення "один до багатьох" з сутністю "Customer", що означає, що один клієнт може зробити кілька замовлень.

Сутність "Customer" містить деталі про клієнтів, включаючи їхній ідентифікатор, ім'я, електронну адресу, телефонний номер та адресу. Ця сутність є ключовою для встановлення відносин з замовленнями та для персоналізації досвіду покупця.

"OrderItem" є зв'язуючою сутністю між "Order" та "Product" та відображає кожен товар чи послугу, що входить до складу конкретного замовлення. Вона включає унікальний ідентифікатор, ідентифікатор замовлення, ідентифікатор продукту, кількість замовленого товару та ціну за одиницю. Відношення "багато до багатьох" між "OrderItem" та "Product" показує, що один продукт може бути включений у багато різних замовлень, і одне замовлення може містити багато різних продуктів.

Сутність "Product" містить інформацію про товари, доступні для замовлення, включаючи ідентифікатор, назву, опис, ціну та кількість на складі. Ця інформація є важливою для управління запасами та аналізу продажів.

Ця ER-діаграма дозволяє розуміти, як дані про замовлення організовані та взаємодіють, що є необхідним для ефективного процесу управління замовленнями, від обробки замовлень клієнтів до відстеження запасів продуктів.

Наступним кроком було розроблено БД для сервісу, що відповідає за транзакції у ІС. Розроблену схему можна побачити на рис 3.10.

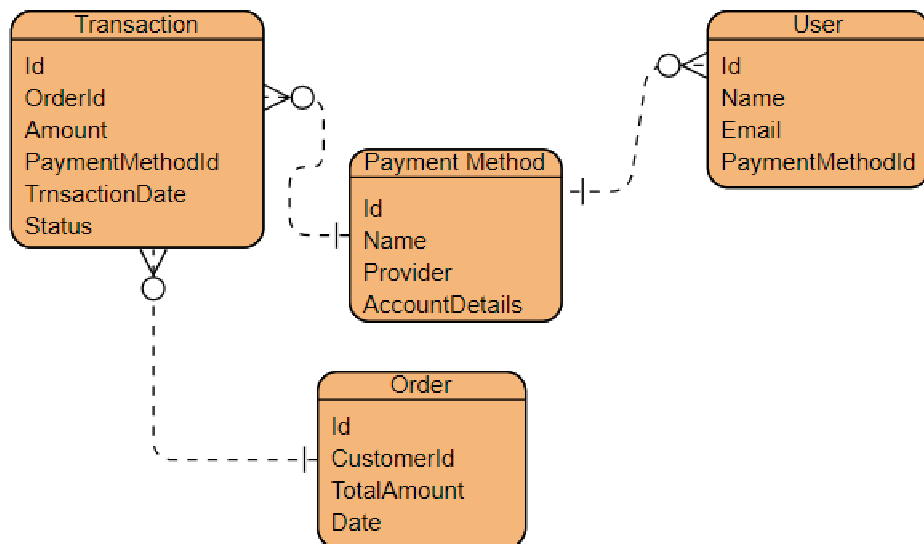

Рис. 3.10 – ER діаграма сервісу управління транзакціями

На цій ER-діаграмі показано структуру бази даних, яка фокусується на процесі обробки транзакцій і пов'язаних із ними даних у системі доставки їжі. Діаграма описує взаємозв'язки між чотирма сутностями: "Transaction" (Транзакція), "Payment Method" (Метод Оплати), "Order" (Замовлення) та "User" (Користувач).

Сутність "Transaction" включає атрибути, які відображають унікальний ідентифікатор транзакції, ідентифікатор замовлення, суму транзакції, ідентифікатор методу оплати, дату транзакції та її статус. Ця сутність відображає кожну фінансову операцію, що відбувається в системі.

Сутність "Payment Method" містить ідентифікатор, назву методу оплати, провайдера та деталі рахунку. Це вказує на способи, які користувачі можуть використовувати для здійснення платежів.

"Order" представляє конкретне замовлення з унікальним ідентифікатором, ідентифікатором клієнта, загальною сумою замовлення та датою замовлення. Це важливо для відстеження замовлень, які користувачі здійснюють.

Кінцева сутність "User" включає інформацію про користувачів системи, таку як їхній ідентифікатор, ім'я, електронну пошту та асоційований

ідентифікатор методу оплати, що дозволяє зв'язати користувачів із їхніми платіжними методами та замовленнями.

Взаємозв'язки на діаграмі відображають, що кожна транзакція пов'язана з одним замовленням та одним методом оплати, що вказує на "один до багатьох" відношення між методами оплати і транзакціями. Крім того, замовлення пов'язані з користувачами, що демонструє, як користувачі здійснюють замовлення. Така структура бази даних є критично важливою для точного відстеження фінансових транзакцій і управління користувацькими даними в системі доставки їжі.

Після цього було розроблено БД для управління користувачами. Розроблену діаграму можна побачити на рисунку 3.11.

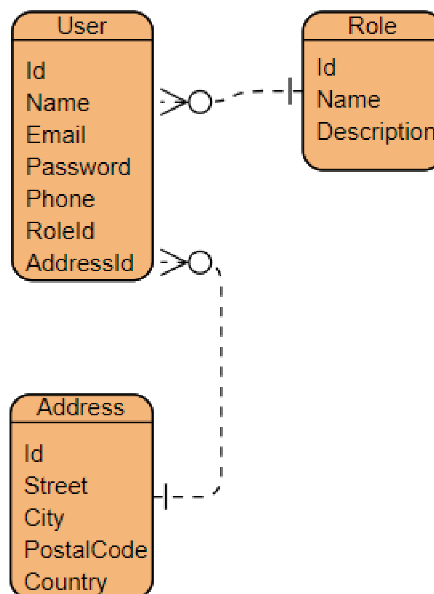

Рис. 3.11 – ER діаграма сервісу управління користувачами

На ER-діаграмі зображено структуру бази даних для управління користувачами в рамках сервісу доставки їжі. Діаграма включає три основні сутності: "User" (Користувач), "Role" (Роль) та "Address" (Адреса), які між собою взаємопов'язані.

Сутність "User" містить інформацію про користувачів системи, включаючи їх ідентифікатор, ім'я, електронну адресу, пароль, телефон, а також ідентифікатори для ролі та адреси. Це надає основні відомості, які ідентифікують користувача в системі та забезпечують контактну інформацію.

Сутність "Role" описує можливі ролі, які можуть бути призначені користувачам, включаючи такі атрибути, як ідентифікатор ролі, її назву та опис. Ролі визначають рівень доступу користувачів до різних функцій системи.

"Address" представляє адресні дані, з якими користувачі можуть бути асоційовані, включаючи вулицю, місто, поштовий код та країну. Ці дані використовуються для доставки замовлень або як частина контактної інформації користувача.

На діаграмі показано відношення "один до багатьох" між "User" та "Role", що означає, що одна роль може бути призначена багатьом користувачам. Відношення "один до одного" між "User" та "Address" свідчить, що кожен користувач асоційований з однією адресою. Ця структура бази даних важлива для організації та управління інформацією про користувачів, їх ролями та адресами в контексті сервісу доставки їжі.

Після цього було розроблено БД для сервісу з аналітики. Розроблену діаграму можна побачити на рис. 3.12.

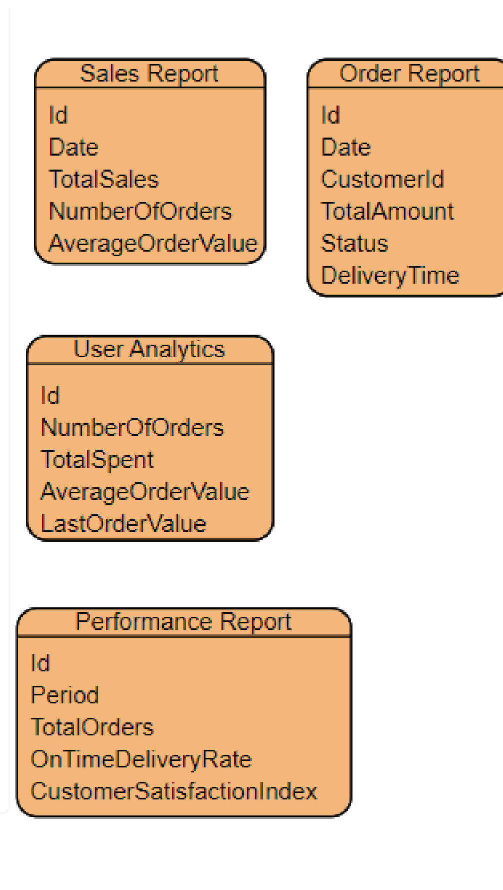

Рис. 3.12 – ER діаграма сервісу управління аналітики

Діаграма відображає зв'язки між чотирма різними типами звітів у системі, яка використовує MongoDB, нереляційну базу даних, оптимізовану для роботи з великими обсягами даних та гнучкою схемою документів. Використання MongoDB у цьому контексті дозволяє легко розширювати і змінювати структуру даних без необхідності перегляду жорстко визначених схем реляційних баз даних.

На діаграмі представлені такі сутності як Sales Report (Звіт Продажів), яка включає поля, що відображають ідентифікатор звіту, дату, загальні продажі, кількість замовлень та середню вартість замовлення. Order Report (Звіт Замовлень), що містить інформацію про кожне замовлення, включаючи ідентифікатор, дату, ідентифікатор клієнта, загальну суму, статус та час доставки. User Analytics (Аналітика Користувачів), що забезпечує дані для аналізу активності користувачів, включаючи ідентифікатор, кількість замовлень, загальну витрату, середню вартість замовлення та вартість останнього

замовлення. Performance Report (Звіт Продуктивності), яка оцінює загальну ефективність системи, охоплюючи період, загальну кількість замовлень, рівень своєчасної доставки та індекс задоволення клієнтів.

В MongoDB ці сутності будуть реалізовані як окремі колекції документів. Відсутність жорстких відносин між колекціями спрощує структуру бази даних і дозволяє кожній колекції ефективно масштабуватись незалежно від інших. Такий підхід добре підходить для систем, де звіти генеруються та використовуються незалежно один від одного, а швидкість читання та запису є важливішою, ніж складність транзакційних операцій, що характерна для реляційних баз даних. Обравши MongoDB, можна отримати можливість гнучко змінювати структуру документів у відповідь на змінені аналітичні потреби та бізнес-вимоги, що є надзвичайно цінним у швидко змінюваному середовищі доставки їжі.

Після цього було продовжено налаштовувати інфраструктуру ІС. Було налаштовано службу виявлення.

Kubernetes надає вбудовану службу виявлення через концепції Services та Pods. Кожен Pod отримує власну IP-адресу, але оскільки Pods можуть динамічно створюватися та знищуватися, Services діють як стабільні точки доступу до групи Pods, що виконують однакові функції.

Коли Service створюється, він отримує стабільну внутрішню класерну IP-адресу. Kubernetes використовує DNS для виявлення сервісів, тобто сервіси всередині кластера можна знаходити за їх DNS-іменами. Це дозволяє сервісам легко знаходити та спілкуватися один з одним.

Kubernetes дозволяє балансувати навантаження між Pods для сервісу:

Існує внутрішнє балансування навантаження це коли Service спрямовує трафік до Pods, воно автоматично балансує навантаження на ці Pods.

Kubernetes може використовувати кілька стратегій для балансування навантаження, таких як round-robin.

Також існує зовнішнє балансування навантаження, коли для доступу зовні кластера, Kubernetes підтримує типи сервісів, такі як LoadBalancer, який

інтегрується з зовнішніми балансувальниками навантаження, що надаються хмарними провайдерами.

Це дозволяє зовнішньому балансувальнику автоматично виявляти та направляти трафік на Pods всередині кластера.

Для більш складних сценаріїв балансування навантаження можуть бути використані сторонні рішення або додаткові інструменти, такі як Istio або Linkerd, які надають більш деталізоване управління трафіком, політиками безпеки, моніторингом та трасуванням.

Служби автентифікації та авторизації можна розробляти вручну як частину системи, або ж використовувати готові рішення, які можуть бути інтегровані у мікросервісний ландшафт.

Kubernetes сам по собі не надає прямих інструментів для розробки служб автентифікації та авторизації для додатків, які він оркеструє. Втім, він пропонує механізми безпеки на рівні інфраструктури, такі як контроль доступу на основі ролей (Role-Based Access Control, RBAC), які управляють доступом до Kubernetes API та ресурсів у кластері.

Існують готові рішення для автентифікації та авторизації, такі як Auth0, Okta, Amazon Cognito, Keycloak, які пропонують багато функціоналу "з коробки" і можуть бути легко інтегровані з мікросервісами. Ці платформи зазвичай надають SDK та API для інтеграції, а також можуть взаємодіяти з Kubernetes через конфігурації та секрети.

Конфігураційний сервер у мікросервісній архітектурі дозволяє централізовано управляти конфігурацією додатків. Це особливо корисно у випадку, коли існують багато сервісів, що потребують однакової конфігурації, або коли конфігурація має динамічно змінюватися без необхідності перезапуску сервісів.

ConfigMaps та Secrets у Kubernetes дозволяють управляти конфігураційними даними та чутливою інформацією відповідно. Вони можуть бути монтувані як томи або експортовані як змінні середовища у Pods.

ConfigMaps - використовуються для збереження конфігураційних даних, які не є чутливими, таких як параметри запуску або налаштування програми.

Secrets - використовуються для збереження чутливих даних, таких як паролі, токени OAuth, ssh ключі тощо.

Обидва ці ресурси дозволяють додаткам перезапускатися та автоматично отримувати останні налаштування, коли конфігурація змінюється.

Для мікросервісних архітектур, особливо тих, що розгорнуті в Kubernetes, ефективна система логування є важливою для моніторингу, відстеження проблем та забезпечення відповідності. В Kubernetes, логи зазвичай збираються з контейнерів, що працюють у pods.

Kubernetes надає базову підтримку логування, дозволяючи отримувати логи з конкретного pod за допомогою команди kubectl logs. Такий підхід є корисним для швидкого перегляду логів, але не масштабується для великих систем.

Також важливою частиною розробки ІС є додавання тестування. Існують архітектурні подходи де тести займають центральне місце, як то TDD.

Тестування мікросервісної архітектури значно відрізняється від монолітної через розподіленість та незалежність її компонентів. У мікросервісах кожен сервіс розробляється, тестується та деплоїться незалежно, що дозволяє командам працювати паралельно та вносити зміни без впливу на роботу всієї системи. Це вимагає окремих тестових підходів для кожного сервісу, а також ретельного міжсервісного тестування для забезпечення правильної інтеграції та взаємодії між ними.

У монолітних системах, де всі компоненти тісно зв’язані, інтеграційне тестування може охоплювати велику частину додатку і часто проводиться через користувацький інтерфейс. Натомість, у мікросервісних системах інтеграційне тестування стає складнішим через необхідність забезпечення комунікації між незалежними сервісами, що може включати використання стабів або моків для імітації залежних сервісів.

Продуктивність у мікросервісах тестується не тільки на рівні окремого сервісу, але й у контексті загальної взаємодії сервісів, з особливою увагою до мережевих затримок та навантаження на API шлюзи. Крім того, важливою частиною є версіонування API, що дозволяє забезпечити безперебійну роботу додатку під час оновлень одного або декількох сервісів.

Неперервна інтеграція та деплоймент у мікросервісах вимагають складніших пайплайнів CI/CD, які можуть автоматично впроваджувати зміни в окремі сервіси без порушення роботи всієї системи. У монолітних системах, натомість, розгортання зазвичай відбувається одночасно для всього додатку.

Загалом, тестування мікросервісної архітектури є більш розподіленим та комплексним процесом, що вимагає додаткових зусиль для координації між сервісами та управлінням залежностями. Водночас це надає більшу гнучкість та можливість швидкішого впровадження змін і ітерацій розробки.

4 ПРОВЕДЕННЯ ЕКСПЕРИМЕНТУ ОТРИМАНОГО ПОРІВНЯЛЬНОГО МЕТОДУ

4.1 Загальні відомості предметної області розробки архітектури ПЗ

Перед початком проведення експерименту слід розглянути загальні відомості щодо мікросервісної та монолітної архтектури. Цей теоретичний мінімум допоможе отримати правильні результати під час перевірці отриманої математичної моделі.

Отже мікросервісна архітектура та монолітна архітектура представляють два фундаментально різних підходи до структурування і розробки програмного забезпечення, кожен з яких має свої переваги та обмеження, що робить їх більш або менш придатними для певних видів проектів.

Монолітна архітектура, традиційний підхід до розробки програмного забезпечення, передбачає створення єдиної і неділимої програми, де всі компоненти системи тісно інтегровані і працюють як одне ціле. Цей підхід може бути ефективним для невеликих або середніх проектів, де важлива простота розгортання та управління. Монолітні системи часто є вибором для стартапів або проектів з обмеженим бюджетом та ресурсами, оскільки вони забезпечують швидкий старт та менш складне технічне обслуговування на ранніх етапах розвитку. Однак, монолітні системи можуть стикатися з проблемами масштабованості та гнучкості при рості проекту, а оновлення та внесення змін можуть бути складними і ризикованими, оскільки зміни в одній частині системи можуть впливати на інші.

З іншого боку, мікросервісна архітектура полягає у розбитті додатку на набір невеликих, автономних сервісів, кожен з яких виконує певну функцію і спілкується з іншими через легкі протоколи, такі як REST. Цей підхід забезпечує високу гнучкість та масштабованість, оскільки кожен мікросервіс може бути розроблений, розгорнутий, масштабований і оновлений незалежно від інших. Мікросервіси ідеально підходять для великих, складних додатків та організацій, які потребують високого рівня гнучкості та швидкої реакції на зміни ринку або

вимог користувачів. Вони також ефективні в середовищах з континуальною інтеграцією та доставкою. Однак, мікросервісна архітектура може бути складнішою у впровадженні та вимагає більш глибокого розуміння взаємодії між сервісами, а також розробки надійних механізмів моніторингу та відновлення.

Отже, вибір між мікросервісною та монолітною архітектурою залежить від багатьох факторів, включаючи розмір та складність проекту, вимоги до масштабованості та гнучкості, наявність ресурсів для розробки та підтримки, а також стратегічні цілі організації. Важливо зважити всі ці аспекти перед прийняттям рішення про вибір архітектури для вашого проекту.

4.2 Збір даних

Для перевірки ефективності розроблених формул через порівняння мікросервісної та монолітної систем, можна використовувати кілька методів збору даних.

Можна використовувати системи логування та моніторінгу. Для цього методу треба встановити системи моніторингу для збору даних про продуктивність, використання ресурсів, час відповіді системи та інші важливі метрики. Логування використовується для фіксації подій, пов'язаних із збоями, помилками та іншими важливими аспектами роботи систем.

Іншим методом є тестування продуктивності. Для цього методу треба провести навантажувальні та стрес-тести для обох систем, щоб визначити їх продуктивність, пропускну здатність та стабільність під навантаженням. Для цього методу використовується спеціалізоване програмне забезпечення для тестування продуктивності.

Також використовуються засоби аналіз логів. Для цього проводиться аналіз логів обох систем для виявлення помилок, збоїв та інших проблем, які можуть вплинути на надійність та ефективність. Для цього методу збору інформації використовуються інструменти для аналізу логів, які можуть допомогти в ідентифікації тенденцій та закономірностей.

Ще одним важливим методом є вимірювання часу відповіді та обробки запитів. Для цього методу використовують інструменти для вимірювання часу відповіді системи на запити користувачів. Цей метод допоможе оцінити час обробки запитів та швидкість відповіді сервера.

Для оцінки заходів безпеки проводять аудит безпеки для обох систем, це полегшує процес аналізу заходів безпеки, а ще допомагає в протидії зловмисним атакам. Цей метод допомагає оцінити відповідність систем стандартам безпеки та політикам.

Ще одним необхідним методом отримання інформації є аналіз робочого навантаження та вартості експлуатації. Для цього треба визначити вартість підтримки та експлуатації обох систем, включаючи ресурси, час розробки та оновлень. Це допоможе оцінити робоче навантаження команди, пов'язане з управлінням та підтримкою систем.

Зібрані дані дозволять об'єктивно порівняти обидві системи за різними критеріями, що передбачені у математичній моделі, та зробити виважені висновки щодо їхньої ефективності та придатності для різних цілей і умов.

4.3 Застосування отриманої математичної моделі

При застосуванні було вирішено почати з критерію «Час до ринку». У формулі 2.23 можна знайти необхідні дані які потрібно отримати. Вони були винесині у таблицю 4.1.

Таблиця 4.1 – Дані для розрахунку критерію «Час до ринку».

Параметр	Значення монолітної архітектури	Значення мікросервісної архітектури
$D_{\text{розробки}}$	60 днів	120 днів
$T_{\text{тестування}}$	5 днів	10 днів
$R_{\text{розгортання}}$	2 дні	1 день
$C_{\text{відповіді}}$	5 днів	5 днів

Продовження таблиці 4.1

$F_{\text{релізи}}$	1 рел/міс	2 рел/міс
$E_{\text{дефекти}}$	4 дні	2 дні

Отже можемо підставити в рівняння і отримати наступне:

$$TDR_{\text{моно}} = w_1 * 60 + w_2 * 5 + w_3 * 2 + w_4 * 5 + w_5 * 1 + w_6 * 4$$

$$TDR_{\text{мікро}} = w_1 * 120 + w_2 * 10 + w_3 * 1 + w_4 * 5 + w_5 * 2 + w_6 * 2$$

Треба підібрати потрібні вагові коефіцієнти. З усіх параметрів лише частота релізів має пряму квантитивну ефективність, тому лише він має використовувати додатне число. Коефіцієнти з обернено квантитивною ефективностью є від'ємними числами. Відповідь користувача як було з'ясовано не залежить від архітектури, тож коефіцієнт стає 0. Коефіцієнти, що було отримано - винесено у таблицю 4.2

Таблиця 4.2 вагові коефіцієнти

Ваговий коефіцієнт	Значення
w_1	-0.4
w_2	-0.3
w_3	-0.2
w_4	0
w_5	0.3
w_6	-0.2

Після отриманих коефіцієнтів отримуємо:

$$TDR_{\text{моно}} = (-0.4) * 60 + (-0.3) * 5 + (-0.2) * 2 + 0 * 5 + 0.3 * 1 + (-0.2) * 4 = -26.4$$

$$TDR_{\text{мікро}} = (-0.4) * 120 + (-0.3) * 10 + (-0.2) * 1 + 0 * 5 + 0.3 * 2 + (-0.2) * 2 = \text{-51}$$

Як можемо бачити в критерії «Час до ринку» виграє Монолітна архітектура, бо найважливіший коефіцієнт «Тривалість розробки» менше саме у монолітної архітектури.

Після цього потрібно провести розрахунки для критерію «Продуктивність». Було зібрано узагальнену інформацію яку можна побачити у таблиці 4.3.

Таблиця 4.3 – Дані для розрахунку критерію «Продуктивність».

Параметр	Значення монолітної архітектури	Значення мікросервісної архітектури
$T_{\text{відповіді}}$	200 мсек	172 мсек
$P_{\text{пропускна}}$	300 од	600 од
$Z_{\text{ресурси}}$	250 мсек	270 мсек
$O_{\text{запити}}$	2 мсек/мсек	1.72 мсек/мсек

Отже, щоб порахувати по формулі 2.24 можемо використати отримані дані та отримуємо:

$$TPR_{\text{моно}} = w_1 * 200 + w_2 * 300 + w_3 * 250 + w_4 * 2$$

$$TPR_{\text{мікро}} = w_1 * 172 + w_2 * 600 + w_3 * 270 + w_4 * 1.72$$

Тепер потрібно підібрати потрібні вагові коефіцієнти. З усіх параметрів лише час відповіді та оптимізація запитів має обернену квантитивну ефективність, тому лише вони мають використовувати негативні числа. Ці коефіцієнти було винесено у таблицю 4.4.

Таблиця 4.4 вагові коефіцієнти

Ваговий коефіцієнт	Значення
w_1	-0.3
w_2	0.2
w_3	-0.2

Продовження таблиці 4.4

w_4	-0.3

Отже, отримаємо наступні значення

$$TPR_{\text{моно}} = (-0.3) * 200 + 0.2 * 300 + (-0.2) * 250 + \ (-0.3) * 2 = -50.6$$

$$TPR_{\text{мікро}} = \ (-0.3) * 172 + \ 0.2 * 600 + \ (-0.2) * 270 + \ (-0.3) * 1.72 = 13.884$$

Як можемо бачити, мікросервісна архітектура, значно продуктивніша, через більшу пропускну здатність та оптимізованість.

Наступним кроком було розглянуто архітектури з критерію «Масштабованість». Щоб використати формулу 2.25 було зібрано необхідну інформацію у таблиці 4.5

Таблиця 4.5 – Дані для розрахунку критерію «Масштабованість».

Параметр	Значення монолітної архітектури	Значення мікросервісної архітектури
$V_{\text{масштабування}}$	0.6	0.8
$H_{\text{масштабування}}$	0.5	0.9
$T_{\text{масштабування}}$	120	48
$C_{\text{масштабування}}$	50	35
$E_{\text{масштабування}}$	0.85	0.9

Отже, щоб порахувати по формулі 2.24 можемо використати отримані дані та отримуємо:

$$TMS_{\text{моно}} = \ w_1 * 0.6 + \ w_2 * 0.5 + \ w_3 * 120 + \ w_4 * 50 + \ w_5 * 0.85$$

$$TMS_{\text{мікро}} = \ w_1 * 0.8 + \ w_2 * 0.9 + \ w_3 * 48 + \ w_4 * 35 + \ w_5 * 0.9$$

Після цього було знайдено вагові коефіцієнти для рівняння. Від’ємним він буде тільку у часі масштабування та її вартість, показані вони у таблиці 4.6.

Таблиця 4.6 – Коефіцієнти для критерію «Масштабованність»

Ваговий коефіцієнт	Значення
w_1	0.2
w_2	0.25
w_3	-0.2
w_4	-0.15
w_5	0.2

Отже, отримаємо наступні значення

$$TMS_{\text{моно}} = 0.2 * 0.6 + 0.25 * 0.5 + -0.2 * 120 + -0.15 * 50 + 0.2 * 0.85 = -31.085$$

$$TMS_{\text{мікро}} = 0.2 * 0.8 + 0.25 * 0.9 + -0.2 * 48 + -0.15 * 35 + 0.2 * 0.9 = -14{,}285$$

Мікросервісна архітектура знов краще монолітної у масштабованості, бо саме для покращення масштабованості вона і була створена, тож результат очікуваний.

Після цього було використано математичну модель для критерію «Надійність». Для того щоб порахувати формулу 2.27 було отримано необхідні дані і поміщено в таблицю 4.7.

Таблиця 4.7 – Дані для розрахунку критерію «Надійність».

Параметр	Значення монолітної архітектури	Значення мікросервісної архітектури
$F_{\text{збоїв}}$	8	6
$MTBF$	4,29	7,5
$MTBR$	1	0,8

Тепер можна підсавити значення у рівняння:

$$TRR_{\text{моно}} = w_1 * 8 + w_2 * 4{,}29 + w_3 * 1$$

$$TRR_{\text{мікро}} = w_1 * 6 + w_2 * 7{,}5 + w_3 * 0{,}8$$

Для того, щоб використати математичну модель потрібно визначитись з коефіцієнтами. Частота збоїв та час на відновлення мають від’ємні коефіцієнти бо чим нижче число в них тим краще працює система. Коефіцієнти містяться у таблиці 4.8.

Таблиця 4.8 – Коефіцієнти для критерію «Надійність»

Ваговий коефіцієнт	Значення
w_1	-0.25
w_2	0.2
w_3	-0.2

$$TRR_{\text{моно}} = -0.25 * 8 + 0.2 * 4.29 + -0.2 * 1 = -1.342$$

$$TRR_{\text{мікро}} = -0.25 * 6 + 0.2 * 7.5 + -0.2 * 0.8 = -0{,}16$$

Знов мікросервісна система має кращі показники, бо має більшу збоєстійкість в порівнянні до монолітної системи бо спроектованна для запобігання їм.

Наступна модель «Використання ресурсів» містить інформацію про те як інформаційні системи використовують надані їм ресурси. Отримані дані показано у таблиці 4.9.

Таблиця 4.9 – Дані для розрахунку критерію «Використання ресурсів».

Параметр	Значення монолітної архітектури	Значення мікросервісної архітектури
$U_{\text{ЦПУ}}$	0.3	0.275
$U_{\text{пам'ять}}$	12,5	14
$U_{\text{дисковий_простір}}$	0,02	0.025

Ці показники відображають загальну тенденцію, що мікросервісні системи можуть бути більш ефективними у використанні ресурсів, оскільки вони

дозволяють більш гранулярне управління ресурсами та масштабування. Водночас, монолітні системи можуть мати вище загальне навантаження на ресурси через їх централізовану структуру.

Математична модель має наступний вигляд:

$$TUR_{\text{моно}} = w_1 * 0.3 + w_2 * 12.5 + w_3 * 0.02$$

$$TUR_{\text{мікро}} = w_1 * 0.275 + w_2 * 14 + w_3 * 0.025$$

Для розрахунку моделі використовуються від'ємні коефіцієнти, які показані на таблиці 4.10

Таблиця 4.10 – Коефіцієнти для критерію «Викоритання ресурсів»

Ваговий коефіцієнт	Значення
w_1	-0.3
w_2	-0.3
w_3	-0.4

Ці коефіцієнти відображають, що низьке використання ключових ресурсів, таких як ЦПУ, пам'ять і дисковий простір, є важливим для оцінки загальної ефективності системи.

$$TUR_{\text{моно}} = -0.3 * 0.3 + -0.3 * 12.5 + -0.4 * 0.02 = -3{,}848$$

$$TUR_{\text{мікро}} = -0.3 * 0.275 + -0.3 * 14 + -0.4 * 0.025 = -4{,}2925$$

Остання окрема модель яку буде розглянуто це «Безпека».

Дані які було зібрано поміщено у таблицю 4.11.

Таблиця 4.11 – Дані для розрахунку критерію «Безпека».

Параметр	Значення монолітної архітектури	Значення мікросервісної архітектури
$S_{\text{дані}}$	0.75	0.75
$S_{\text{атаки}}$	0.7	0.9
$A_{\text{безпека}}$	0.7	0.7

Продовження таблиці 4.11

$P_{\text{політика}}$	0.8	0.8
$I_{\text{іднетифікація}}$	0.8	0.8
$U_{\text{оновлення}}$	0.65	0.85
$C_{\text{сертифікації}}$	0.78	0.8

Отримані дані дозволяють заповнити математичну модель і виглядатиме вона таким чином

$$TSR_{\text{моно}} = w_1 * 0.75 + w_2 * 0.7 + w_3 * 0.7 + w_4 * 0.8 + w_5 * 0.8 + w_6 * 0.65 + w_7 * 0.78$$

$$TSR_{\text{мікро}} = w_1 * 0.75 + w_2 * 0.9 + w_3 * 0.7 + w_4 * 0.8 + w_5 * 0.8 + w_6 * 0.85 + w_7 * 0.8$$

Після цього було створено коефіцієнти для аналізу математичної моделі. Їх наведено в таблиці 4.12.

Таблиця 4.12 – Коефіцієнти для критерію «Безпека»

Ваговий коефіцієнт	Значення
w_1	0.2
w_2	0.2
w_3	0.15
w_4	0.15
w_5	0.1
w_6	0.1
w_7	0.1

Отже можна порахувати

$$TSR_{\text{моно}} = 0.2 * 0.75 + 0.2 * 0.7 + 0.15 * 0.7 + 0.15 * 0.8 + 0.1 * 0.8 + 0.1 * 0.65 + 0.1 * 0.78 = 0.738$$

$$TSR_{\text{мікро}} = 0.2 * 0.75 + 0.2 * 0.9 + 0.15 * 0.7 + 0.15 * 0.8 + 0.1 * 0.8 + 0.1 * 0.85 + 0.1 * 0.8 = 0.8$$

Як можемо побачити через свою складність мікросервісна архітектура стає набагато безпечнішою.

Всі отримані данні було занотовано у таблицю 4.13. Після їх розгляду можна перевірити головну узагальнену модель.

Таблиця 4.13 – Отримані дані

Назва	Параметр	Значення монолітної архітектури	Значення мікросервісної архітектури
Час до ринку	TDR	-26.4	-51
Продуктивність	TPR	-1.342	-0.16
Масштабованність	TMS	-31.085	-14.285
Надійність	TRR	-1,342	-0.16
Використання Ресурсів	TUR	-3.848	-4.2925
Безпека	TSR	0.738	0.8

Після отримання даних можна внести їх у математичну модель.

$$TA_{\text{моно}} = \propto_1 * (-26.4) + \propto_2 * (-1.342) + \propto_3 * (-31.085) + \propto_4 * (-1.342) + \propto_5 * (-3.848) + \propto_6 * 0.738$$

$$TA_{\text{мікро}} = \propto_1 * (-51) + \propto_2 * (-0.16) + \propto_3 * (-14.285) + \propto_4 * (-0.16) + \propto_5 * (-4.2925) + \propto_6 * 0.8$$

При аналізу математичної моделі можна зрозуміти, що як монолітна так мікросервісна архітектура мають свої сильні та слабкі сторони. Тож для різних стратегій розробки інформаційних систем будуть використовуватися різні архітектури.

Отже було розроблено дві стратегії застосування моделі:

видка розробка ІС для якомога швидшого виходу на ринок;

ільш повільна розробка ІС для створення більш якісної системи.

Отже було отримано таблицю 4.14 яка містить всі необхідні коефіцієнти для загальної математичної моделі.

Таблиця 4.14 – Коефіцієнти загальної математичної моделі

Стратегія I		Стратегія II	
$\propto_1$	0.8	$\propto_1$	0.2
$\propto_2$	0.2	$\propto_2$	0.4
$\propto_3$	0.2	$\propto_3$	0.4
$\propto_4$	0.2	$\propto_4$	0.4
$\propto_5$	0.2	$\propto_5$	0.2
$\propto_6$	0.2	$\propto_6$	0.2

Після цього отримуємо перший розрахунок:

$$TA_{\text{моно}} = 0.8 * (-26.4) + 0.2 * (-1.342) + 0.2 * (-31.085) + 0.2 * (-1.342) + 0.2 * (-3.848) + 0.2 * 0.738 = -28.4958$$

$$TA_{\text{мікро}} = 0.8 * (-51) + 0.2 * (-0.16) + 0.2 * (-14.285) + 0.2 * (-0.16) + 0.2 * (-4.2925) + 0.2 * 0.8 = -44.4195$$

Отже можемо бачити, що при використанні монолітної архітектури можна швидше вийти на ринок та отримувати прибуток за розробляємий додаток.

Це можливий варіант для деяких компаній, але є ті, які намагаються розробити більш якісний застосунок, тож у цій стратегії головний акцент поставлено на продуктивності, масштабованості та надійності. Було отримано наступне рівняння:

$$TA_{\text{моно}} = 0.2 * (-26.4) + 0.4 * (-1.342) + 0.4 * (-31.085) + 0.4 * (-1.342) + 0.2 * (-3.848) + 0.2 * 0.738 = -19.4096$$

$$TA_{\text{мікро}} = 0.2 * (-51) + 0.4 * (-0.16) + 0.4 * (-14.285) + 0.4 * (-0.16) + 0.2 * (-4.2925) + 0.2 * 0.8 = -16.7405$$

Найоптимальнішою архітектурою у цьому випадку виявилась мікросервісна архітектура, що очікувано навіть для малих додатків.

4.4 Перевірка гіпотез

На початку роботи були виділені гіпотези, щодо точності моделі та різниці між архітектрами, після виконання роботи було отримано наступну інформацію.

По-перше, після ретельного аналізу та валідації, можна з упевненістю стверджувати, що розроблена математична модель високою точністю у відображені ключових параметрів обох архітектур. Модель продемонструвала здатність ефективно працювати у різних умовах, що свідчить про її вправність та надійність. Особливо варто відзначити гнучкість моделі, яка досягається завдяки використанню вагових коефіцієнтів, що дозволяють адаптувати її до різних реалій бізнесу. Ця адаптивність робить модель особливо цінною для аналізу та порівняння мікросервісних та монолітних систем у різноманітних комерційних застосуваннях. Таким чином, представлена математична модель не тільки відповідає поставленим вимогам точності та ефективності, але й є вагомим інструментом для планування та розробки архітектурних рішень у сучасному бізнес-середовищі.

По-друге, виходячи з результатів експерименту, було встановлено, що монолітна архітектура надає перевагу у швидкості створення пілотної версії проекту. Це обумовлено меншою складністю інтеграції та відсутністю необхідності у координації між різними сервісами, що є характерним для мікросервісної архітектури. Проте, у довгостроковій перспективі мікросервісна архітектура виявилася більш продуктивною, масштабованою та надійною. Ця архітектура дозволяє ефективніше управляти великими наборами даних та користувацькими запитами, забезпечуючи вищу ступінь відмовостійкості і гнучкості у разі необхідності внесення змін чи оновлень. Таким чином, хоча монолітна архітектура може виявитися кращим рішенням для швидкого запуску проекту, мікросервісна архітектура володіє більшим потенціалом для розширення та підтримки великомасштабних і складних систем.

4.5 Можливі майбутні дослідження

Було проаналізовано які можуть бути проведені дослідження далі у майбутньому.

По-перше, можливо проаналізувати вплив різних бізнес-сценаріїв. Дослідити, як різні бізнес-моделі та стратегії можуть впливати на вибір між мікросервісною та монолітною архітектурами. Це може включати аналіз таких факторів, як розмір компанії, ринкова ніша, швидкість змін у технологічному ландшафті тощо.

По-друге, можливо дослідити оптимізацію витрат та ресурсів. Розробити моделі для оцінки витрат та ресурсів, необхідних для міграції з монолітної архітектури на мікросервісну. Це може включати аналіз вартості розробки, підтримки, а також потенційних ризиків та вигод.

Також ще одним варіантом дослідження є експерименти з різними технологічними стеками. А саме дослідження впливу вибору технологічних стеків на ефективність та продуктивність обох архітектур. Це може допомогти у визначенні найбільш ефективних технологій для кожної з архітектур.

Ще можливо додати, що дослідження безпеки та відмовостійкості. А саме поглиблене дослідження аспектів безпеки та відмовостійкості у мікросервісних та монолітних архітектурах. Це може включати аналіз вразливостей, стратегій запобігання збоям та методів швидкого відновлення після збоїв.

Можливий варіант майбутніх досліджень - вплив на продуктивність команди розробників. Дослідження того, як вибір архітектури впливає на продуктивність та ефективність команд розробників, включаючи аспекти управління проектами, співпраці та розподілу завдань.

Також, можна розглянути розробку гібридних моделей. Дослідити можливості інтеграції елементів мікросервісної та монолітної архітектур, щоб створити гнучкі гібридні моделі, які можуть поєднувати переваги обох підходів.

ВИСНОВКИ

В результаті виконання професійної практики було досліджено створення ІС на мікросервісній архітектурі та порівняно за допомогою математичної моделі мікросервісну та монолітну архітектуру.

У висновках цієї кваліфікаційної роботи варто підкреслити, що розроблена математична модель вдало демонструє свою ефективність та точність у порівнянні мікросервісної та монолітної архітектур. Повнота виконання завдання підтверджується детальним аналізом досягнутих кількісних показників, які відображають ключові аспекти обох архітектурних стилів. Модель успішно інтегрує різні вимірювання, такі як продуктивність, масштабованість, вартість впровадження та експлуатації, надаючи комплексний погляд на переваги та недоліки кожного підходу.

Завдяки використанню вагових коефіцієнтів, модель здатна гнучко адаптуватися до різних бізнес-сценаріїв, що робить її цінним інструментом для планування та стратегічного рішення у різних умовах реального бізнесу. Таким чином, розроблена модель не лише відповідає науковим та практичним вимогам кваліфікаційної роботи, але й вносить суттєвий вклад у розуміння складних питань, що стосуються вибору архітектури для інформаційних систем.

У рамках цієї кваліфікаційної роботи було розроблено вебзастосунок на мікросервісній архітектурі та перевірено як впливає вибір архітектури на розробку програмного забезпечення за допомогою власної багатокритеріальної моделі оптимізації. Основною перевагою цієї моделі є її простота у використанні, що робить її доступною для широкого кола фахівців, незалежно від їхнього рівня технічної експертизи. Модель була розроблена таким чином, щоб забезпечити високу точність у визначенні та аналізі ключових характеристик обох архітектур, включаючи їх продуктивність, масштабованість, вартість впровадження та експлуатаційні витрати.

За допомогою проведеного експерименту було отримано інформацію щодо доцільного використання мікросервісної архітектури. За допомогою математичної моделі було досліджено, що мікросервісна архітектура підходить

для створення надійних та високопродуктивних застосунків, та не підходить там де потрібна швидкість виходу на ринок.

Розробка та дослідження власної Багатокритеріальної Моделі Оптимізації в контексті порівняння мікросервісної та монолітної архітектур ідеально вписується в основні наукові напрями кафедри системотехніки, які включають системний аналіз, моделювання та оптимізацію складних організаційно-технічних систем. Ця робота розширює існуючі знання в області математичного моделювання, надаючи новий інструмент для структурно-топологічного синтезу та оцінки інформаційних систем. А розроблена система демонструє переваги та недоліки обох архітектур, що дозволить більш доцільно спрямовувати ресурси при розробці ІС.

Ця математична модель може бути використана як для навчальних цілей, так і в якості основи для досліджень, спрямованих на розв’язання задач комбінаторної оптимізації у геометричному проектуванні.

Таке використання може допомогти в підготовці фахівців, які зможуть ефективно інтегрувати теоретичні знання з практичним досвідом, забезпечуючи глибоке розуміння сучасних вимог до системного аналізу та проектування інформаційних систем.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Методичні вказівки до організації виконання та захисту атестаційної роботи на здобуття першого (бакалаврського) рівня вищої освіти для студентів усіх форм навчання спеціальності 151 – «Автоматизація та комп'ютерноінтегровані технології» за освітньою програмою «Системна інженерія» / Упорядники: І.В. Гребеннік, В.Г. Іванов, Б.О. Колесник, А.С. Нечипоренко, П.Е. Ситнікова, О.С.Чорна – Харків: ХНУРЕ, 2019. – 58 с.

2. Microservices.io [Електронний Ресурс] – Режим доступу: https://microservices.io/ (Дата звернення: 12.10.2023).

3. Spring Documentation [Електронний Ресурс] – Режим доступу: https://docs.spring.io/spring-framework/docs/current/reference/html/ (Дата звернення: 12.10.2023).

4. Docker Documentation [Електронний Ресурс] – Режим доступу: https://docs.docker.com/ (Дата звернення: 12.10.2023).

5. Kubernetes Documentation [Електронний Ресурс] – Режим доступу: https://kubernetes.io/docs/ (Дата звернення: 12.10.2023).

6. RabbitMQ Documentation [Електронний Ресурс] – Режим доступу: https://www.rabbitmq.com/documentation.html (Дата звернення: 12.10.2023).

7. Martin Fowler's Blog [Електронний Ресурс] – Режим доступу: https://martinfowler.com/articles/microservices.html (Дата звернення: 12.10.2023).

8. Azure Architecture Center [Електронний Ресурс] – Режим доступу: https://docs.microsoft.com/en-us/azure/architecture/ (Дата звернення: 12.10.2023).

9. AWS Microservices Architecture [Електронний Ресурс] – Режим доступу: https://aws.amazon.com/microservices/ (Дата звернення: 12.10.2023).

10. Medium [Електронний Ресурс] – Режим доступу https://medium.com/@inverita/frontend-optimization-8-tips-to-improve-web performance-29af4b00efe7

11. NGINX Microservices Reference Architecture [Електронний Ресурс] – Режим доступу: https://www.nginx.com/blog/introducing-the-nginx-microservices-reference-architecture/ (Дата звернення: 12.10.2023).

12. Red Hat Microservices [Електронний Ресурс] – Режим доступу: https://www.redhat.com/en/topics/microservices (Дата звернення: 12.10.2023).

Microservices vs. monolithic architecture - Atlassian [Електронний Ресурс] – Режим доступу: https://www.atlassian.com/microservices (Дата звернення: 12.10.2023).

13. Microservices vs. Monolithic Architectures - Baeldung on Computer Science [Електронний Ресурс] – Режим доступу: https://www.baeldung.com/cs/microservices-monolithic-architectures (Дата звернення: 12.10.2023).

14. Monolithic vs Microservices - Difference Between Software Development Architectures- AWS [Електронний Ресурс] – Режим доступу: https://aws.amazon.com/microservices/ (Дата звернення: 12.10.2023).

15. (PDF) Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation [Електронний Ресурс] – Режим доступу: https://www.researchgate.net/publication/346651284_Monolithic_vs_Microservice_Architecture_A_Performance_and_Scalability_Evaluation (Дата звернення: 12.10.2023).

16. Microservices vs Monolith: The Ultimate Comparison 2021 - DZone [Електронний Ресурс] – Режим доступу: https://dzone.com/articles/microservices-vs-monolith-the-ultimate-comparison (Дата звернення: 12.10.2023).

17. Monolithic Architecture Vs. Microservices: An Overall Comparison - Orient Software [Електронний Ресурс] – Режим доступу:

https://www.orientsoftware.com/blog/monolithic-architecture-vs-microservices/ (Дата звернення: 12.10.2023).

18. The Comparison of Microservice and Monolithic Architecture - ResearchGate [Електронний Ресурс] – Режим доступу: https://www.researchgate.net/publication/346892624_The_Comparison_of_Microservice_and_Monolithic_Architecture (Дата звернення: 12.10.2023).

19. Monolithic vs. Microservices Architectures: What to Choose - Digiteum [Електронний Ресурс] – Режим доступу: https://www.digiteum.com/monolithic-vs-microservices-architecture (Дата звернення: 12.10.2023).

20. Monolith vs Microservice Architecture: A Comparison - Camunda [Електронний Ресурс] – Режим доступу: https://camunda.com/learn/whitepapers/monolith-vs-microservice/ (Дата звернення: 12.10.2023).

21. Monolithic vs. Microservices Architecture: Ultimate Comparison Guide - Light IT [Електронний Ресурс] – Режим доступу: https://light-it.net/blog/monolithic-vs-microservices-architecture-ultimate-comparison-guide/ (Дата звернення: 12.10.2023).

22. Comparing monolith and microservice architectures for software delivery - Chronosphere [Електронний Ресурс] – Режим доступу: https://chronosphere.io/learn/comparing-monolith-and-microservice-architectures-for-software-delivery/ (Дата звернення: 12.10.2023).

23. Сучасні напрями розвитку інформаційно-комунікаційних технологій та засобів управління, Тези доповідей дванадцатої міжнародної науково-технічної конференції 28 листопада – 01 грудня 2023 року, «Розробка методу порівняльного аналізу Мікросервісної і Монолітної архітектури Янчинський І. В., Ситніков Д. Е.. – Харків: ХНУРЕ, 2023. – 76с.

24. Студентський науковий журнал «UNIVERSUM», стаття «Розробка методу порівняльного аналізу Мікросервісної і Монолітної архітектури Янчинський І. В., Ситніков Д. Е.. – Вінниця: УКРЛОГОС Груп, 2023. – 135с.