Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Навчально-науковий центр заочної форми навчання
(повна назва)

Кафедра Інформаційно-мережної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)
Аналіз варіантів розгортання додатку в хмарі
(тема)

Виконав:
здобувач 2 року навчання,
групи ІМІзм-23-1
Лаврик Д. Р.
(прізвище, ініціали)

Спеціальність 172 Електронні комунікації та радіотехніка
(код і повна назва спеціальності)
Тип програми Освітньо-професійна
(освітньо-професійна або освітньо-наукова)
Освітня програма «Інформаційно-мережна інженерія»
(повна назва освітньої програми)

Керівник доц. Костромицький А.І.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ Безрук В.М.
(підпис) (прізвище, ініціали)

2025 р.

Харківський національний університет радіоелектроніки

Факультет Навчально-науковий центр заочної форми навчання

Кафедра Інформаційно-мережної інженерії

Рівень вищої освіти другий (магістерський)

Спеціальність 172 Електронні комунікації та радіотехніка
(код і повна назва)

Тип програми Освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма «Інформаційно-мережна інженерія»
(повна назва)

ЗАТВЕРДЖУЮ:

Зав. кафедри _____
(підпис)

«_07__»__листопада___ 20 24 р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Лаврику Дмитру Романовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Аналіз варіантів розгортання додатку в хмарі

затверджена наказом університету від _07_ _____11_____2024 р. № 186Стз

2. Термін подання здобувачем роботи до екзаменаційної комісії _15__ _січня_____ 2025 р.

3. Вихідні дані до роботи

Провести порівняльний аналіз основних патернів, що використовуються для побудови та подальшого розгортання додатків в хмарах. Зокрема, розглянути багаторівневу, монолітну, сервісно-орієнтовану, мікросервісну, подійно-орієнтовану, та MACH.
Провести експериментальне порівняння розгортання WORDPRESS у варіантах мікросервісної та монолітної архітектури у хмарі AWS

4. Перелік питань, що потрібно опрацювати в роботі

Вступ
1 Найпоширеніші архітектурні патерни
2 Аналіз обраного застосунку
3 Розгортання монолітної архітектури
4 Розгортання мікросервісної архітектури
5 Дослідження характеристик розгорнутих архітектур
Висновки

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) Слайди у форматі PowerPoint (назва та мета роботи, актуальність, огляд архітектур, структура застосунку, розгорнуті архітектури, висновки)

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата
Основна частина	доц. Костромицький А.І.		07.11.2024

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Ознайомлення із завданням. Уточнення ТЗ.	07.11.24	
2	Підбір літератури за темою роботи.	08.11.-15.11.24	
3	Виконання розділу 1	16.11-22.11.24	
4	Виконання розділу 2	23.11-29.11.24	
5	Виконання розділу 3	30.11-06.12.24	
6	Виконання розділу 4	07.12-18.12.24	
7	Виконання розділу 5	19.12-31.12.24	
8	Оформлення презентаційного матеріалу, підготовка	01.01-15.01.25	
	до захисту у ЕК		

Дата видачі завдання _07__ ____листопада___2024 р.

Здобувач _____
(підпис)

Керівник роботи _____ доц. Костромицький А.І.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка 81с., 27 рис., 4 табл., 1 додаток, 28 джерел.

Об’єкт роботи – архітектура веб-застосунку.

Мета роботи – аналіз різних підходів до побудови архітектури застосунку в хмарному середовищі.

Розглянуто ряд архітектурних підходів, за якими побудовані сучасні застосунки. Було обрано застосунок, над яким ставитимуться експерименти, описано його внутрішню структуру. Було виконано розгортання обраного застосунку в монолітній та мікросервісній архітектурах. Було виконано тестування різних показників якості застосунку, розгорнутого в різних архітектурних конфігураціях. Проведено порівняння та аналіз результатів тестування.

АРХІТЕКТУРА, МІКРОСЕРВІСИ, МОНОЛІТ, MACH, SOA, WORDPRESS, DEVOPS.

ABSTRACT

Explanatory slip 81p., 37 fig., 4 tab., 1 app., 28 sources.

The object of work is the web application architecture.

The goal of the work is to analyse different of different approaches on deploying the application architecture on the cloud computing environment.

Overview of the most popular architecture patterns, that are used in the modern software development. Selection of the application, that will be used in the further experiment, overview of the internal structure. Deployment of the chosen application in monolithic and microservice architectures. Testing of the deployed architectures and measurement of quality of service. Analysis and comparison of the collected metrics.

ARCHITECTURE, MICROSERVICES, MONOLITH, MACH, SOA, WORDPRESS, DEVOPS.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ І ТЕРМІНІВ

ActiveMQ – програмне забезпечення для обміну повідомленнями, яке реалізує Java Message Service (JMS) і забезпечує можливості обміну даними між додатками;

Amazon Aurora – хмарна реляційна база даних від AWS, створена для високої продуктивності та доступності, сумісна з MySQL і PostgreSQL;

AMQP (Advanced Message Queuing Protocol) – протокол обміну повідомленнями, розроблений для забезпечення взаємодії між різними системами;

Apache – відкрите програмне забезпечення для роботи веб-серверів, яке широко використовується для розміщення веб-сайтів та додатків;

Apache JMeter – інструмент з відкритим кодом для тестування продуктивності веб-додатків та сервісів;

API (Application Programming Interface) – набір правил та інструментів для створення та інтеграції програмного забезпечення;

API-first – підхід до розробки, який ставить розробку API на перше місце, забезпечуючи узгодженість і можливість інтеграції;

AWS (Amazon Web Services) – хмарна платформа, яка пропонує широкий спектр послуг, включаючи обчислювальні ресурси, зберігання даних, бази даних, аналітику, DevOps тощо;

AWS Lambda – безсерверна обчислювальна служба AWS, яка виконує код у відповідь на події без необхідності керувати серверами;

CloudFormation – сервіс AWS для визначення та розгортання інфраструктури за допомогою шаблонів у форматах JSON або YAML;

CloudFront – служба доставки контенту (CDN) від AWS для швидкої доставки даних, відео та додатків по всьому світу;

Cloud-native – архітектурний підхід до створення і розгортання додатків за допомогою хмарних технологій, орієнтований на масштабованість і стійкість;

CloudTrail – сервіс AWS для реєстрації та відстеження API-активності в обліковому записі, що використовується для аудиту;

CloudWatch – сервіс моніторингу AWS для відстеження метрик, логів і подій у хмарних додатках;

CSV (Comma-Separated Values) – формат файлу для зберігання табличних даних у вигляді тексту, де колонки розділені комами;

CSS (Cascading Style Sheets) – мова стилізації для оформлення зовнішнього вигляду HTML-документів (розташування, кольори, шрифти тощо);

DevOps – набір практик, що об'єднує розробку програмного забезпечення (Dev) і IT-операції (Ops) для скорочення циклу розробки та підвищення якості;

DNS (Domain Name System) – система, яка перетворює домени на IP-адреси;

DynamoDB – NoSQL-база даних від AWS, створена для низької затримки і високої продуктивності;

EBS (Elastic Block Store) – служба AWS для блочного сховища, яка використовується з EC2 для зберігання даних;

EC2 (Elastic Compute Cloud) – сервіс AWS для масштабованих віртуальних серверів у хмарі;

ELB (Elastic Load Balancer) – балансувальник навантаження від AWS, який автоматично розподіляє вхідний трафік між декількома серверами;

ESB (Enterprise Service Bus) – проміжне програмне забезпечення для інтеграції додатків і сервісів у сервісно-орієнтованій архітектурі;

FastCGI (Fast Common Gateway Interface) – Оптимізований протокол для інтеграції веб-серверів і програм;

Framework – програмна структура або платформа, яка забезпечує основу для розробки додатків;

Headless – підхід до створення додатків, де фронтенд (інтерфейс користувача) відділений від бекенду (логіки і даних), що дозволяє використовувати різні технології для кожного компонента;

HornetQ – високопродуктивна платформа для обміну повідомленнями, яка підтримує протокол JMS;

HTML (HyperText Markup Language) – мова розмітки для створення веб-сторінок, яка структурує контент у вигляді тексту, зображень, посилань тощо;

HTTP (HyperText Transfer Protocol) – протокол передачі гіпертексту, що використовується для комунікації між веб-браузерами і серверами;

HTTPS (HTTP Secure) – безпечна версія HTTP, яка використовує шифрування через SSL/TLS для захисту даних;

IaC (Infrastructure as Code) – автоматизація інфраструктури через програмний код;

ID (Identifier) – унікальний ідентифікатор, який використовується для позначення об'єктів у системі;

IP (Internet Protocol) – основний протокол, що забезпечує адресацію та маршрутизацію пакетів даних у мережі;

JavaScript (JS) – мова програмування, що використовується для створення інтерактивності на веб-сторінках;

JMS (Java Message Service) – API від Java для обміну повідомленнями, який дозволяє додаткам взаємодіяти через асинхронний обмін повідомленнями;

Kubernetes – платформа з відкритим кодом для оркестрації контейнерів, яка дозволяє автоматизувати розгортання, масштабування та управління контейнеризованими додатками;

MACH (Microservices, API-first, Cloud-native, Headless) – архітектурний підхід, що поєднує мікросервіси, орієнтацію на API, хмарні рішення та відділення фронтенда від бекенда;

MariaDB – реляційна база даних з відкритим кодом, що є форком MySQL;

Microservices – архітектурний стиль, де додаток складається з невеликих автономних сервісів, які взаємодіють між собою через API;

MVC (Model-View-Controller) – архітектурний шаблон для розробки програмного забезпечення, який розділяє дані (Model), представлення (View) та логіку управління (Controller);

MySQL – реляційна база даних з відкритим кодом, яка широко використовується для зберігання даних у веб-додатках;

Nginx – веб-сервер з відкритим кодом, що використовується для обробки HTTP-трафіку, а також як балансувальник навантаження чи зворотний проксі-сервер;

NoSQL – бази даних для неструктурованих або великих даних, без реляційних таблиць;

PHP – серверна мова програмування, яка використовується для створення веб-додатків;

PHP-FPM (FastCGI Process Manager) – альтернатива стандартному PHP FastCGI для обробки великих навантажень у веб-додатках;

Pipeline – процес автоматизації завдань, таких як побудова, тестування та розгортання програмного забезпечення;

PostgreSQL – потужна реляційна база даних з відкритим кодом, яка підтримує розширення і додаткові функції для роботи з даними;

REST (Representational State Transfer) – архітектурний стиль для створення веб-сервісів, які використовують HTTP для передачі даних;

RMI (Remote Method Invocation) – механізм Java, що дозволяє викликати методи об'єктів, розташованих на віддалених серверах;

Route53 – служба DNS від AWS для управління доменними іменами;

S3 (Simple Storage Service) – об'єктне сховище від AWS, яке використовується для зберігання і резервування даних;

SOAP (Simple Object Access Protocol) – протокол для обміну структурованими повідомленнями в мережах;

SOA (Service-Oriented Architecture) – архітектурний стиль, де функціональність додатків поділена на окремі сервіси з чіткими інтерфейсами;

SOA-Lite – полегшена версія SOA, орієнтована на менші системи;

SQL (Structured Query Language) – мова для роботи з реляційними базами даних;

SSD (Solid State Drive) – тип накопичувача, який забезпечує високу швидкість зчитування і запису даних;

SSL (Secure Sockets Layer) – протокол для захисту даних у мережі через шифрування;

Terraform – інструмент для управління інфраструктурою як кодом (IaC), що дозволяє створювати, змінювати та версіонувати інфраструктуру;

TLS (Transport Layer Security) – Оновлена і безпечніша версія SSL;

VPC (Virtual Private Cloud) – віртуальна приватна мережа AWS для ізольованого запуску ресурсів у хмарі;

WordPress – відкрита платформа для створення веб-сайтів та блогів, яка підтримує плагіни та теми.

ВСТУП

У сучасному світі інформаційних технологій створення ефективних програмних застосунків є ключовим для забезпечення стабільності, масштабованості та гнучкості бізнесу. Швидкий технологічний розвиток, зростаючі потреби користувачів і економічні вимоги спонукають організації розробляти архітектури, які можуть підтримувати динамічні навантаження та швидко адаптуватися до мінливих умов. Архітектура програми забезпечує структуру для створення програмного забезпечення, забезпечуючи структуру для розподілу обов'язків і організації зв'язку між компонентами. Кожен архітектурний підхід має свої переваги та недоліки, які визначають сферу його застосування.

Наразі найбільшу популярність мають монолітна архітектура, а також архітектури, засновані на мікросервісному принципі. Новим витком в цьому є архітектура MACH, яка дозволяє відносно легко підтримувати, модифікувати та масштабовувати великі застосунки. В даній роботі ми розглядатимемо лиш монолітну та мікросервісну архітектури відносно малого застосунку.

Актуальність питання. Правильна побудова архітектури є основою правильної побудови складних систем. Аналіз різних підходів до побудови дозволить обрати необхідну нам архітектуру при розробці такої складної системи, як застосунок.

Метою цієї роботи є аналіз найпоширеніших архітектур програмного забезпечення, практична реалізація та порівняння ряду архітектурних підходів. Під час виконання даної дипломної роботи ми повинні набути навичок роботи з технічною документацією, з сучасними технологіями розгортання та тестування архітектур; навчитись аналізувати дані, отримані під час проведених експериментів та приймати відповідні рішення.

1 НАЙПОШИРЕНІШІ АРХІТЕКТУРНІ ПАТЕРНИ

1.1 Короткий огляд популярних архітектурних підходів

Побудова архітектури застосунку це процес визначення структури, який відповідає всім технічним і операційним вимогам до застосунку. Це креслення, за яким будується системний дизайн та визначаються комунікації всередині застосунку.

Архітектура включає в себе ряд ключових рішень щодо організації системи програмного забезпечення. Сюди входять основні компоненти системи, їхні інтерфейси, інтеграція між ними, та композиція та взаємозв’язок між цими елементами. Всі ці рішення ґрунтуються на вимогах до функціоналу, надійності, продуктивності та можливості повторного використання коду, а також економічних та технологічних обмеженнях.

В цілому, архітектура програмного забезпечення це план, за яким розробники розробляють рішення для виконання певних бізнес-вимог, при цьому впевнюючись в тому, що застосунок масштабований, легкий у підтримці та адактований до зміни потреб бізнесу [1].

Тут ми розглянемо наступні найпоширеніші архітектурні патерни, які зустрічаються в програмному забезпеченні.

1. Багаторівнева архітектура.

Багаторівнева архітектура це один з найпоширеніших типів архітектури застосунків. В цій моделі, програмні компоненти організовані горизонтальними рівнями, кожен з власною специфічною роллю. Зазвичай ці рівні включають представницький, бізнес-логіки, доступу до даних. Це також має назву трирівневої архітектури (N-tier architecture).

Прикладом такої архітектури служить дизайн типового веб-застосунку. Користувацький інтерфейс формує представницький рівень; серверна логіка формує рівень бізнес-логіки; база даних формує рівень доступу до даних. Перевагами такої моделі є простота й чіткий розподіл ролей, що спрощує підтримку продукту, та розширення за необхідності.

Втім, багаторівнева архітектура має і власні недоліки. Це надмірна зв’язанвсть між рівнями, що ускладнює внесення змін. Зміна в одному рівні впливає на верхні рівні, таким чином весь застосунок стає нестабільним[1].

2. Монолітна архітектура.

В монолітній архітектурі, всі компоненти програмного забезпечення тісно зв’язані і взаємозалежні. Вся система являє собою одне ціле, і окремі компоненти не можуть функціонувати незалежно від інших. Всі функції застосунку та залежності запаковані разом, що робить застосунок простим в розробці, тестуванні та розгортанні.

Втім, монолітні архітектури є доволі проблемними в плані підтримки та розширення, особливо коли розмір та складність застосунку зростає. Більш того, несправність в одному компоненті тягне за собою всю систему, що призводить до серйозних порушень в роботі сервісу [1].

3. Сервісно орієнтована архітектура.

Сервісно-орієнтована архітектура (SOA, Service-Oriented Architecture) це одна зі старих архітектур, що досі використовується деякими організаціями, яку вважають предком мікросервісної архітектури. Це паттерн, де сервіси надаються іншим компонентам мережею, використовуючи комунікаційний протокол. Основний принцип SOA полягає в обміні послугами між компонентами програмного забезпечення за допомогою протоколу, який незалежний від платформи.

Найкращим прикладом SOA є банківська система де такі сервіси, як обробка даних карти, керування користувацькими даними та обробка даних про заборгованості представлені різним корстувацькми застосункам як окремі сервіси.

Перевагою сервісно-орієнтованої архітектури є можливість використання тої самої кодової бази в функціонально різних застосунках. Втім, це може бути складним для впровадження та підтримки. Також не найкращою є продуктивність таких застосунків через необхідність у надто частій комунікації між сервісами [1].

4. Мікросервісна архітектура.

В мікросервісній архітектурі, застосунок побудований у вигляді колекції малих незалежних одне від одного сервісів. Кожен сервіс виконаний як окремий процес і комунікує з іншими через добре продуману систему API.

Найкращим прикладом такої архітектури є Netflix. Кожна функція, як автентифікація користувачів, потокова передача відео, рекомендації, тощо, є окремими сервісами.

Такий підхід до побудови застосунку має такі переваги як легке масштабування та простота в підтримці продукту, оскільки кожен сервіс можна масштабувати та оновлювати незалежно.

Втім, мікросервісна архітектура має і власні недоліки. Такий підхід може призвести дл надмірного ускладнення системи що призводить до збільшення витрат. Більш того, необхідність в комунікації між сервісами може призвести до затримок в мережі та проблем з узгодженістю даних [1].

5. Подійно-орієнтована архітектура.

В подійно-орієнтованій архітектурі, робочий потік визначається такими подіями як користувацькі дії, виведення сенсорів чи повідомлень від інших програм. Компоненти застосунку реагують на події та створюють нові, створюючи нескінченний динамічний процес.

Прикладом такої архітектури є застосунок чату в реальному часі. В ньому користувацькі повідомлення є подіями, які ініціюють відповідь сервера та оновлення інших користувацьких інтерфейсів. Така архітектура має високу швидкість відповіді та дозволяє оновлення в реальному часі.

Втім, керування потоком подій в складній системі може бути складним, а асинхронна природа подійно-орієнтованої архітектури ускладнює пошук і вирішення проблем в таких застосунках [1].

6. Архітектура MACH

Архітектура MACH це сучасний підхід до системного дизайну, який полягає в інкапсуляції мікросервісного підходу (Microservices), переваги використанню API (API-first), використання в основному у хмарі (Cloud-native),

та відсутності тісних зв’язків користувацького інтерфейсу з серверною логікою (Headless). Такий підхід дає основу для побудови гнучких, масштабованих та прогресивних програм, в тому числі і в контексті веб-сайтів електронної комерції.

Дана архітектура також асоційована з розробкою сучасних, витривалих та легко пристосованих до змін застосунків, і має за мету подолати обмеження традиційних монолітних архітектур використовуючи модульність, гнучкість і використання хмарних сервісів.

Прикладом такої архітектури є Shopify. Це хмарна платформа, яка дозволяє кому-завгодно створювати власні інтернет-магазини, використовуючи headless архітектуру. Також ця платформа використовує принцип API-first, що дозволяє розробникам інтегрувати інші сервіси [2].

Розглянемо наведені вище архітектури в деталях.

1.2 Багаторівнева архітектура

Багаторівнева архітектура - це один з найпоширеніших підходів до побудови архітектури затосунків. Цей архітектурний підхід є де-факто стандартом для більшості сучасних застосунків через простоту, зрозумілість і малу вартість імплементації. Це також природний підхід до розробки застосунків через закон Конвея, згідно з яким, організації, які розробляють системи схильні розробляти архітектури, які є копіями власної організаційної структури. В більшості організацій є розробники користувацького інтерфейсу, розробники бізнес-логіки, розробники правил комунікації та експерти з баз даних. Такі організаційні шари добре накладаються на рівні традиційної багаторівневої архітектури, що робить її природним вибором для багатьох бізнес-застосунків [3].

Компоненти багаторівневої архітектури організовані в горизонтальні рівні, кожен з них виконує специфічну роль застосунку, наприклад, інтерфейс чи бізнес-логіка. Хоча багаторівнева архітектура не визначає кількість рівнів, в

класичному вигляді існує 4 рівні: представницький, бізнес-логіки, рівень доступу до бази даних та рівень баз даних. Зазвичай, бізнес-логіку й сервісний рівень об’єднують в один рівень, утворюючи трирівневу архітектуру [4].

Стандартна чотирьохрівнева архітектура зображена на рисунку 1.1.

Рисунок 1.1 – Багаторівнева архітектура [4].

Представницький рівень це та частина застосунку, яка відповідає за взаємодію застосунку з користувачем. Основна робота цього рівня полягає у наданні інформації в інтерактивній манері та інтерпретація користувацьких дій у вигляд, який дозволяє програмі розуміти й відповідати на ці дії. Головна мета цього рівня це створення інтуїтивного та приємного користувацького досвіду, будь то інтерфейс веб-сайту, мобільний застосунок чи потужний десктопний програмний пакет.

Рівень бізнес-логіки це власне вся робоча логіка застосунку, яка є проміжним шаром між користувацьким інтерфейсом та керуванням даними. На цьому рівні відбувається найбільш важка на ресурси робота: верифікація користувацького введення, різного роду математичні розрахунки, прийняття ключиових рішень та координація потоків даних.

Рівень доступу до баз даних це той рівень де відбуваються маніпуляції з даними, що робить її незамінною частиною і міцним фундаментом застосунків.

Робота цього рівня полягає в дотриманні постійності та стандартності збережених даних, в правильному отриманні та записі інформації при роботі з базами даних чи іншими системами сховищ [5].

Рівень баз даних чи рівень сховищ даних це рівень, де вся інформація, яка обробляється застосунком зберігається та управляється.

Багаторівнева архітектура, основний принцип в розробці програмного забезпечення, організовує застосунки в окремі рівні, кожна з власною роллю в системі. При широкій поширеності, існує багато різних типів багаторівневих архітектур, кожна з власними позитивними й негативними рисами.

Традиційна багатошарова архітектура.

Традиційна багаторівнева архітектура це класична модель, яка зазвичай складається з представницького рівня, рівня бізнес-логіки та рівня доступу до даних. Такий підхід має дуже прямий і чітко розділений розподіл ролей, що робить його дуже легким в обслуговуванні та знаходженні проблем. Цей підхід найкращий для малих та середніх застосунків. При такому підході кожен компонент зосереджується на власному завданні, від користувацького інтерфейсу до керування взаємодією з базами даних.

Трирівнева архітектура.

Трирівнева архітектура це пряме застосування багаторівневого підходу, поділяючи застосунок на три окремі рівні: преставницький для користувацького інтерфейсу, логічний рівень для виконання основної роботи застосунку, та рівень даних для управління базами даних. Дана модель набула широкого поширання в розробці веб-застосунків завдяки збалансованому підходу до розподілу ролей, що спрощує розробку, обслуговування і масштабування [5]. Один з прикладів даної моделі зображений на рисунку 1.2.

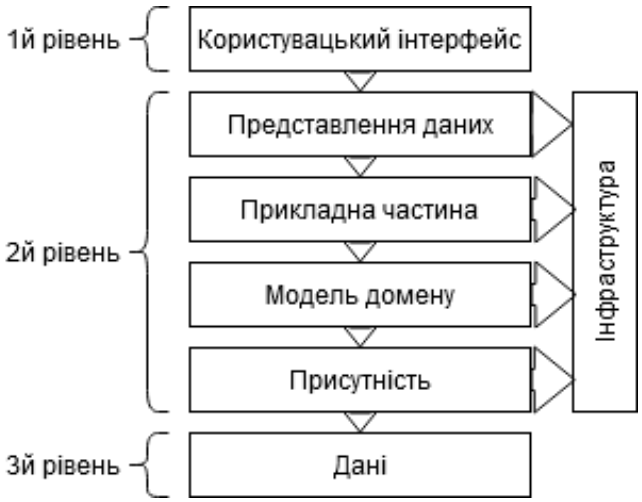

Рисунок 1.2 – Трирівнева архітектура.

N-рівнева архітектура.

N-рівнева розширює стандартну трирівневу модель вводячи нові додаткові рівні, як, наприклад, рівень обслуговування або інтеграційний рівень. Даний підхід збільшує гнучкість та масштабованість, але такий підхід потребує більш системного підходу до побудови комунікації між рівнями. Такий підхід знаходить своє застосування у великих продуктів, де необхідна інтеграія з різноманітними базами даних, сервісами та зовнішніми застосунками [5].

Мікросервісна архітектура.

Мікросервісна архітектура бере за основу принципи побудови багаторівневої архітектури та застосовує це в розподілених системах. Кожен мікросервіс являє собою окремим рівнем, який відповідає за ту чи іншу бізнес-функцію; мікросервіси комунікують між собою за допомогою добре визначених інтерфейсів. Така архітектура підтримує Agile підхід до розробки, має найкращу масштабованість та може значно підвищити гнучкість та надійність складних систем.

Таким чином, багаторівнева архітектура являє собою логічну організацію застосунку в рівні, які чітко розділені. Розроблені таким чином застосунки прості в розгортанні, масштабуванні та мають високу надійність. Втім, такий підхід має такі недоліки як високий час відповіді, пов’язана з необхідністю проходити кілька рівнів, неефективність використання бази даних, а також проблеми з

комунікацією між рівнями. При всьому цьому, багаторівнева архітектура є найпоширенішою в розробці застосунків завдяки своїй простоті та суттєвим перевагам і зручностям, які вона надає [5].

1.3 Монолітна архітектура

У монолітній архітектурі всі елементи програми — від інтерфейсу користувача та бізнес-логіки до коду доступу до даних — створені та об’єднані в єдину кодову базу та репозиторій. Ця архітектура зазвичай використовує такі концепції, як шаблони/теми та шаблон проектування Model-View-Controller (MVC) (рис.1.3).

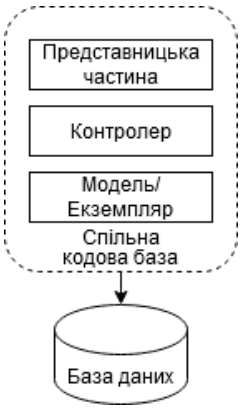

Рисунок 1.3 – Традиційна монолітна архітектура

Монолітна архітектура з невеликою внутрішньою структурою за межами простих класів демонструє узгодження майже так само погано, як так звана “велика куля бруду” (big ball of mud). Таким чином, зміни в одній частині коду можуть мати непередбачені побічні ефекти в іноді далекосяжних частинах бази коду.

При цьому ця архітектура має властивість до розвитку, запобігаючи утворенню “великої кулі бруду”. Цю архітектуру досить легко дегенерувати, тому що є кілька структурних обмежень, щоб запобігти цьому.

Багато переваг, які архітектори наголошують стосовно мікросервісів — ізоляція, незалежність, невелика одиниця змін — можна досягти в монолітних архітектурах при умові, якщо розробники будуть надзвичайно дисциплінованими щодо пропрацювання комунікацій в коді.

Добре спроектований модульний моноліт є хорошим прикладом даного пропрацювання. Кожен компонент є функціонально зв’язаним, з хорошими інтерфейсами між ними та низьким зв’язком [9].

Модульна монолітна архітектура полягає в розбитті логіки програми на модулі, де кожен модуль є незалежним та ізольованим. Тоді кожен модуль має власну бізнес-логіку — і, за необхідності, власну базу даних та інші компоненти [10].

Крім того, кожен модуль може дотримуватися власного логічного розділення, наприклад, як показано на рисунку 1.4, кожен модуль відповідає стилю багаторівневої архітектури, але також вони можуть дотримуватися чистої архітектури всередині модуля, коли організовують логічні рівні. Таким чином є можливість створювати та змінювати рівні кожного модуля, не впливаючи на інші. Кожен модуль підключається лише до інших модулів, які надають необхідні послуги.

Рисунок 1.4 – Модульна монолітна архітектура [10].

При монолітному архітектурному підході весь код програми розгортається та виконується як єдиний процес на одному вузлі. Важливо зрозуміти, чому ключовими словами тут є «розгортання» і «вузол». Що стосується першого, розгортання, це означає, що не має значення, де фізично зберігається код, чи він організований в одному чи кількох репозиторіях, а те, як він організований під час виконання. Щодо другого ключового слова, вузол, це означає, що це все ще моноліт. Тобто розгортання на кількох серверах відбувається у випадках горизонтального масштабування.

У сервері з одним вузлом усі модулі моноліту збираються в один образ пам’яті, який виконується як єдиний процес на одному вузлі. Комунікації відбуваються шляхом стандартних викликів процедур через той самий стек. Один образ пам’яті робить програму монолітною. У випадку запуску модулів в різних процесах, необхідно пророблювати міжпроцессову комунікацію. Оскільки модулі підпадають під різні обмеження, може виникати проблема розподілених обчислень, таким чином торкаючись мікросервісів.

Хоча ця архітектура має погану репутацію серед архітекторів, цей підхід може працювати добре навіть при розробці великих застосунків. Монолітна архітектура програє у випадках, коли необхідне незалежне масштабування різних компонентів домену, коли різні модулі необхідно написати різними мовами програмування, або коли необхідне швидке незалежне розгортання застосунків [11].

1.4 Сервісно орієнтована архітектура

Сервісно орієнтована архітектура (SOA) це підхід до архітектури програмного забезпечення, в якому компоненти програмного забезпечення, які використовуються для створення бізнес-застосунку, називаються сервісами. Кожен сервіс надає певний функціонал для застосунку. Сервіси при цьому коммунікують між собою незалежно від платформ і мов програмування.

Розробники використовують SOA для застосування тих самих сервісів у різних системах або для використання кількох незалежних сервісів для вирішення комплексних завдань [12].

Існує багато сервісно-орієнтованих архітектур, включаючи кілька гібридних паттернів. На рисунку 1.5 зображений один з найпоширеніших підходів – на основі загальної сервісної шини (ESB). При даному підході до побудови архітектури, медіатором для складної обробки подій виступає ESB, яка також регулює різні типові архітектурні процеси, такі як перетворення повідомлень та хореографія [9].

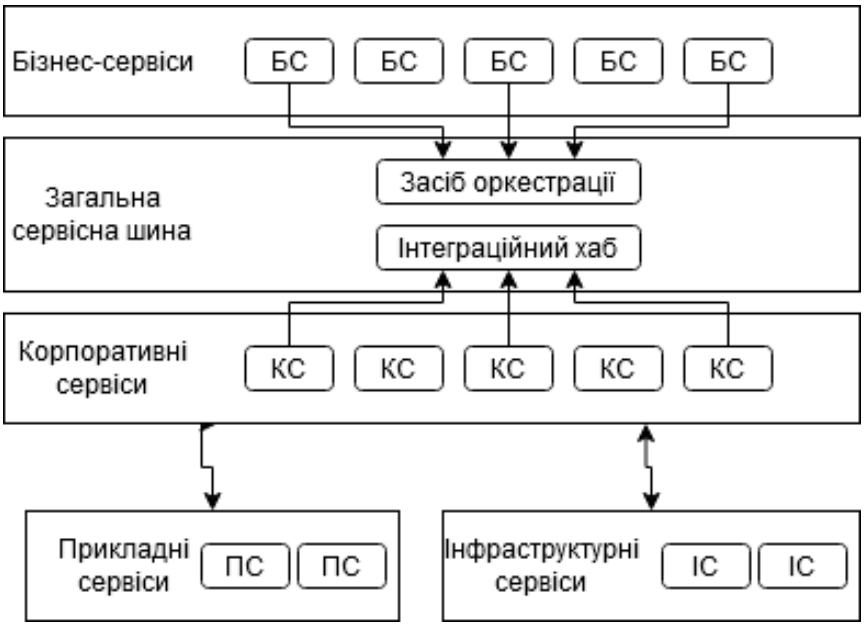

Рисунок 1.5 – Сервісно орієнтована архітектура на основі ESB.

Не всі приклади такого архітектурного стилю мають структуру, наведену вище, але вони всі побудовані на загальному принципі встановлення системного підходу до контролю різноманітних сервісів всередині архітектури, кожен шар з власною зоною відповідальності [3].

Розглянемо основні принципи даного архітектурного підходу.

Систематизація.

В даній архітектурі основна філософія архітектора фокусується на можливості повторного використання компонентів в перспективі. Оскільки

багато великих компаній не хочуть переписувати програмний код, необхідна стратегія, яка дозволяє вирішити дану проблему при реалізації нових застосунків. Кожен рівень систематизації повинен рухатись до даної мети.

Бізнес-сервіси

Бізнес-сервіси знаходяться нагорі даної архітектури та є точками входу. Наприклад, такі сервіси, як ExecuteTrade чи PlaceOrder представляють поведінку домену. Розділення на сервіси відбувається за принципом можливості надання чіткої позитивної відповіді на запитання: “Чи займається компанія…?”.

Дані сервіси мають мінімум коду — введення, виведення, прості перетворення даних. Вони зазвичай визначені користувачами бізнесів, як і власте назви цих сервісів [3].

Корпоративні сервіси.

Дані сервіси містять добре продумані імплементації логіки, які загальні для всього застосунку. Зазвичай, команда розробників має завдання побудови компоненту програми, яка зосереджена на певному бізнес-домені. Такі сервіси є тими блоками, з яких складають різні бізнес-сервіси та пов’язані між собою за використання оркестрації.

Даний розподіл відповідальності витікає з мети у повторному використанні кодової бази в даному архітектурному підході. Якщо розробники зможуть побудувати добре продумані корпоративні сервіси при правильному рівні розподілу задач, бізнес не матиме необхідності в переписуванні тієї чи іншої частини робочого процесу. Таким чином, бізнес матиме можливість будувати різні застосунки з готових корпоративних сервісів.

Втім, динамічна натура бізнес-процесів накладає власні обмеження. Ринок та технології змінюються, інженерні практики вдосконалюються, проводяться різні спроби встановити стабільність в світі програмного забезпечення [3].

Прикладні сервіси.

Не всі сервіси в архітектурі вимагають того самого рівня розділення чи повторного використання, як корпоративні сервіси. Прикладні сервіси є одноразовими сервісами з однією імплементацією. Наприклад, одному

застосунку потрібна геолокацію, але організація не хоче витрачати час чи зусилля для розробки повторно використоуваного сервісу, в таких випадках використовуються дані сервіси .

Інфраструктурні сервіси.

Інфраструктурні сервіси забезпечують операційні потреби, такі як моніторинг, логінг, автентифікацію і авторизацію. Такі сервіси зазвичай є імплементаціями, якими володіє інфраструктурна команда, яка близько співпрацює з командами, які обслуговують фізичні сервери.

Засіб оркестрації.

Засіб оркестрації формує серце даної розподіленої архітектури, яка поєднує в єдине імплементації бізнес-сервісів через оркестрацію, включаючи такі речі, як координація транзакцій та обробку повідомлень. Дана архітектура типічно підв’язана до однієї реляційної бази даних, або кількох, на відміну від мікросервісів, де є одна база даних для кожного з сервісів. Таким чином, транзакційна поведінка визначається в засобі оркестрації, аніж в базі даних.

Засіб оркестрації визначає відносини між бізнес та корпоративними сервісами, як вони поєднуються, та де пролягають межі транзакцій. Він також грає роль інтеграційного концентратору, дозволяючи архітекторам інтегрувати власний код з пакетами та системи підтримки старого коду.

Оскільки такий механізм формує серце архітектури, закон Конвея правильно передбачає, що команда інтеграційних архітекторів, відповідальних за даний засіб стають політичною силою всередині організації, та можливо сповільнювачем частини процесів всередині компанії.

В той час, як даний підхід звучить добре, на практиці зазвичай ця архітектура проявляє себе вкрай погано. Транзакційна поведінка орекстраційного засобу звучить добре, але знаходження правильного рівня розподілу транзакцій стає складнішим зі зростанням застосунку. В той час як побудова кількох сервісів в розподілену транзакційну систему можлива, архітектура сильно ускладнюється коли необхідно визначити правильні межі між сервісами [3].

Мета сервісно орієнтованої архітектури є широка мережа об’єднаних між собою сервісів, які розгорнуті і готові до комунікації через сервісну шину. Запровадження SOA забезпечує достатню надійність і гнучкість для сервісів. Дані переваги забезпечуються не тільки дивлячись на архітектуру сервісів з технологічної перспективи, а й запровадженням сервісно-орієнтованого середовища, побудованого за рядом основних принципів даного підходу.

Найбільш важливим концептом є сервіс. Веб-сервіси це набір протоколів, завдяки яким сервіси можуть бути доступними та використаними в технологічно-нейтральній, стандартній формі.

З технологічної перспективи це не тільки архітектура сервісів, але більш за все набір політик, практик та фреймворків, завдяки яким можна впевнитись в тому, що відбувається правильний обмін повідомленнями між сервісами.

Критично важливо при такому підході впевнитись що є принаймні два процеси – для провайдера й споживача.

Замість того, щоб залишати розробників розбиратись з окремими сервісами та збирати їх у застосунок, загальна сервісна шина допомогає також визначити правильні сервіси, які треба використовувати для забезпечення потрібного користувацького досвіду в їхньому домені [13].

1.5 Мікросервісна архітектура

Мікросервісна архітектура являє собою набір слабко пов’язаних між собою розподілених сервісів. Кожен мікросервіс розроблений для виконання певної специфічної ролі в застосунку та може розроблятись, розгортатись та масштабуватись незалежно від інших. Це дозволяє розбивати великі складні застосунки в малі компоненти з вузько визначеними функціями. Мікросервіси можуть бути написані будь-якою мовою програмування з використанням всіх можливих фреймворків, і кожен сервіс виступає як малий застосунок сам по собі [14].

Існує декілька керівних принципів, які застосовуються до мікросервісного підходу до архітектури застосунків. Як показано на рисунку 5.1, кожен компонент мікросервісу розгорнутий як окрема одиниця, що спрощує розгортання завдяки налагодженню ефективних та добре оптимізованих пайплайнів. Також даний підхід спрощує масштабування та забезпечує високу програмну та компонентну незалежність.

Одним з найбільш важливих концептів даного паттерну є поняття сервісного копоненту. В даній архітектурі структурною одиницею є не окремо взятий модуль чи мікросервіс, а один або об’єднання кількох модулів у функціональну одиницю, яка виконує певну визначену функцію сервісу, що і є сервісним компонентом застосунку. Однією з найскладніших задач архітектора при розробці системи це задати правильне розподілення застосунку на сервісні компоненти [4].

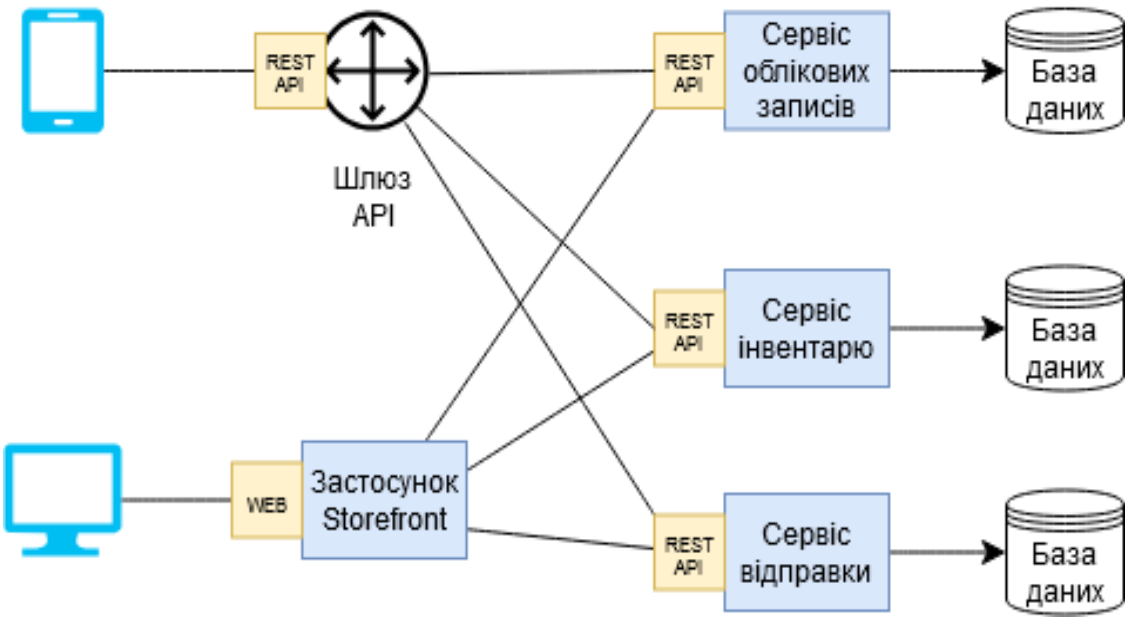

Рисунок 1.6 – Типова архітектура мікросервісного застосунку [15].

Інший ключовий концепт мікросервісного архітектурного патерну це те, що це є розподілена архітектура, що означає, що всі компоненти в даній архітектурі повністю незалежні одне від одного та зв’язуються між собою за допомогою певних протоколів віддаленого доступу (наприклад, JMS, AMQP,

REST, SOAP, RMI, тощо). Розподілена природа даного архітектурного патерну є головним чинником найкращої масштабованості з-поміж інших архітектур та кращі показники розгортання застосунків.

Сама мікросервісна архітектура виникла для компенсації існуючих проблем з традиційних архітектур. Основні два патерни з яких виник даний підхід до архітектури це монолітна архітектура, з якої було взято багатошаровий підхід та сервісно-орієнтована архітектура з якої було взято розподілений підхід до побудови застосунків. Еволюція даної архітектури з монолітної зумовлена розвитком інструментів безперервної доставки коду, поняття пайплайну безперервної доставки коду з розробки в робочий застосунок дозволяє добре організувати розгортання застосунків. Монолітні застосунки типово складаються з тісно зв’язаними компонентами, що робить дані застосунки складним при оновлені, тестуванні та розгортанні. Зміни в коді часто призводять до помилок в роботі всього застосунку. Дану проблему мікросервісна архітектура вирішує виділенням функціональних блоків у незалежні сервісн компоненти, які можуть розгортатись, тестуватись та оновлюватись окремо від інших.

Є три основні топології в даній архітектурі. API REST топологія, прикладна REST топологія та топологія з централізованою системою обміну повідомленнями.

Топологія API REST корисна для веб-сайтів, які мають маленькі самодостатні сервіси, які викликаються через API. Дана топологія, яка зображена на рисунку 1.7, складається з ряду добре розподілених сервісних компонентів (або мікросервісів), які мають один або два модулі, які виконують специфічні бізнес-функції, незалежні від інших сервісів. В даній топології, ці мікросервіси типово доступні через інтерфейс на базі REST, який виконаний через окремо розгорнутий рівень API [4].

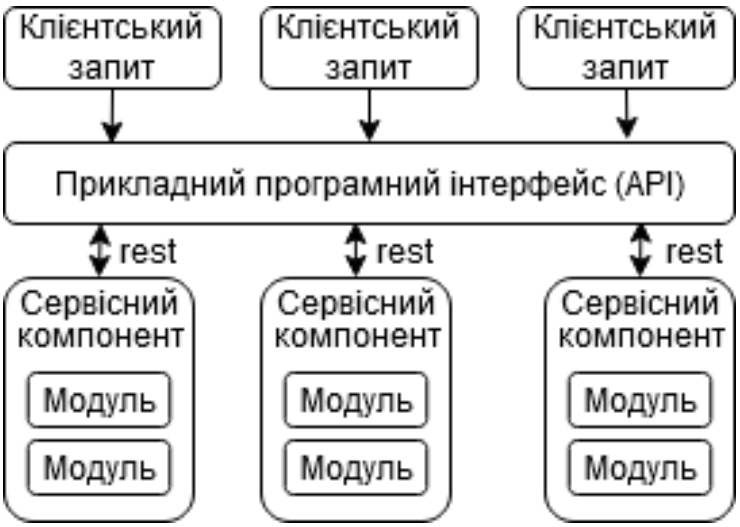

Рисунок 1.7 – Топологія API REST.

Прикладна REST топологія відрізняється від тої, що має API в тому, що користуваціькі запити обробляються через традиційний веб-застосунок, аніж через API. Як показано на рисунку 1.8, рівень користувацького інтерфейсу розгорнутий у вигляді окремого застосунку, який віддалено зв’язується з різними розгорнутими сервісними компонентами через інтерфейси REST. Сервісні комопненти в даній топології відрізняються від аналогічних в API-REST в тому розумінні що вони зазвичай більші, слабше розбиття та в цілому дана топологія має дуже обмежене застосування в сучасних бізнес-застосунках. Дана топологія знаходить своє застосування в малих та середніх застосунках, які є відносно простими [4].

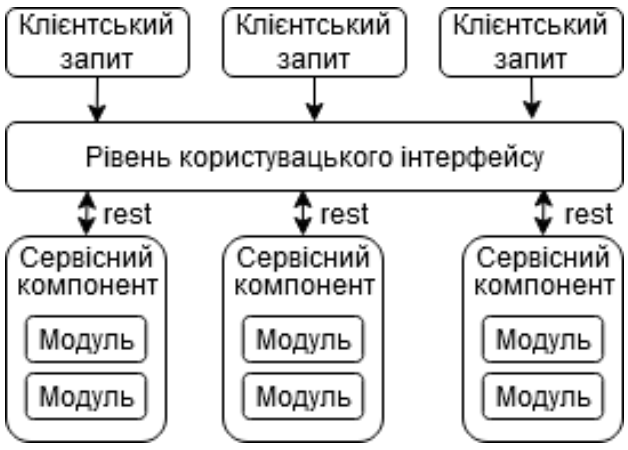

Рисунок 1.8 – Прикладна REST топологія.

Наступний підхід до організації мікросервісної архітектури, який ми розглянемо в даній роботі, це топологія з централізованим обміном повідомлень. Дана топологія подібна до попередньої прикладної REST топології за винятком того, що замість використання інтерфейсів REST для віддаленого доступу, дана топологія використовує полегшений централізований брокер повідомлень (наприклад, ActiveMQ, HornetQ, тощо). Хоча його важко переплутати з полегшеною версією сервісно-орієнтованої архітектури “SOA-Lite”, на відміну від неї, полегшений брокер повідомлень не виконує ніякої оркестрації, перетворення чи складної маршрутизації. Це скоріше легкий транспорт для отримання доступу до віддалених сервісних компонентів.

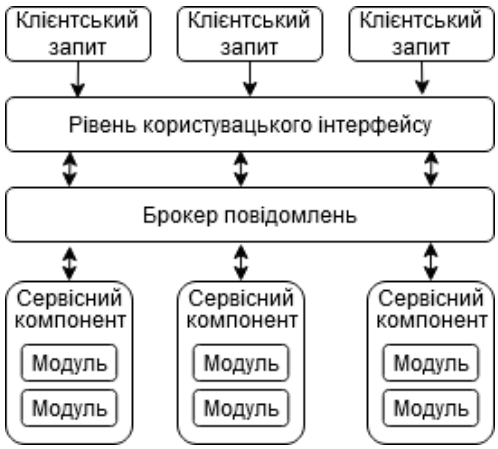

Рисунок 1.9 – Топологія з системою обміну повідомлень.

Топологія з централізованим обміном повідомлень типово присутній у великих бізнес-застосунках, або там, де необхідно більш складний контроль транспортного рівня між користувацьким інтерфейсом і сервісними компонентами [4].

Мікросервісна архітектура вирішує багато складних проблем, які мали розробники з традиційними монолітними рішеннями. Втім, розподілення розробки та надмірна кількість різних попереджень створюють нові виклики.

Таким чином, важливо мати моніторингову платформу, розроблену спеціально для мікросервісної архітектури. Аналізуючи величезну кількість логів та виділення важливого, оптимізація моніторингу допоможе розробникам

користуватись перевагами мікросервісної архітектури на повну, маючи легкість моніторингу, як в монолітних архітектурах [16].

1.6 Подійно-орієнтована архітектура

Подійно-орієнтований архітектурний патерн — це популярний розподілений асинхронний архітектурний паттерн, який використовується для побудови високо масштабованих застосунків. Ця архітектура має високу пристосовуваність та може використовуватись однаково ефективно як в малих застосунках, так і складних. Подійно-орієнтована архітектура заснована на ряді високо розділених компонентів обробки подій, які асинхронно приймають та обробляють події. Як показано на рисунку 1.10, на відміну від мікросервісної архітектури, в подійно-орієнтовану архітектуру вирізняє наявність маршрутизатора подій, який є центром архітектури, на відміну від мікросервісів, де комунікації децентралізовано побудовані індивідуально для кожного з сервісних компонентів [4].

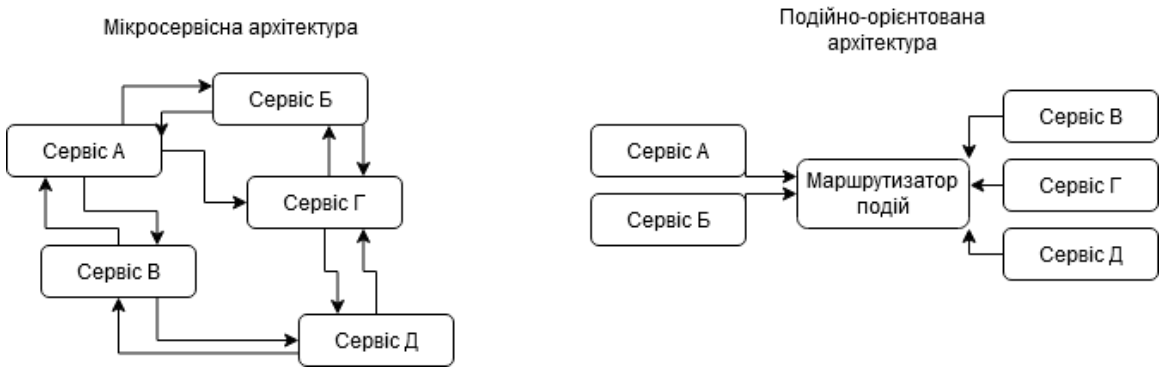

Рисунок 1.10 – Подійно-орієнтована архітектура в порівнянні з мікросервісною архітектурою [17].

Центральним поняттям в даній архітектурі є подія. Подією може будь-яка зміна в системі, будь-то зовнішній вплив від користувачів, чи системні

оновлення. Дані події можуть нести як детальну інформацію, так і сповіщення про зміну стану системи [17].

В подійно-орієнтованій архітектурі типово є три головні компоненти:

1. Ініціатори подій. Даний компонент слугує джерелом подій, які генерують сповіщення про зміну станів чи оновлення.

2. Маршрутизатори подій. Також відомий як подійний брокер чи шина. Даний компонент фільтрує та спрямовує події від ініціаторів подій до необхідних отримувачів.

3. Отримувачі подій. Даний компонент являє собою сервіси чи компоненти, які реагують на події, обробляючи за необхідності.

Існує дві основні топології в подійно-орієнтованій архітектурі. Медіаторна топологія і брокерна топологія. Медіаторна топологія знаходить своє застосування там, де потрібний контроль робочих потоків, в той час як брокерна там, де необхідно висока швидкість відповіді та динамічний контроль обробки подій. Оскільки архітектурні характеристики та імплементаційні стратегії відрізняються між двома топологіями, важливо розуміти принципи роботи кожен з них для визначення, що є кращим рішенням в тій чи іншій ситуації.

Медіаторна топологія краща для таких подій, які мають кілька кроків, та вимагають певного рівня оркестрацію для обробки подій. Наприклад, одна подія виставлення торгів на біржі вимагає спочатку верифікувати факт торгу, потім перевірити відповідність даних торгів поточним правилам, призначити цю подію брокеру, вирахувати комісію та в кінці виставити торги з даним брокером. Всі ці процедури потребують певного рівня оркестрацію для визначення порядку кроків та тих кроків які можуть виконуватись серійно чи паралельно.

Існує чотири основні типи архітектурних компонентів в медіаторній топології: черги подій, подійний медіатор, подійні канали, та обробники подій. Потік подій починається з того, що клієнт надсилає подію у чергу, яка використовується для передачі даної події на медіатор. Медіатор подій приймає первинну подію та орекструє її, надсилаючи додаткові асинхронні події в подійні канали для виконання кожного кроку процесу. Обробники подій, які

слухають канали отримують подію від медіатору та виконують необхідну бізнес-логіку для обробки події. Рисунок 1.11 ілюструє типову медіаторну топологію в подійно-орієнтованому архітектурному патерні.

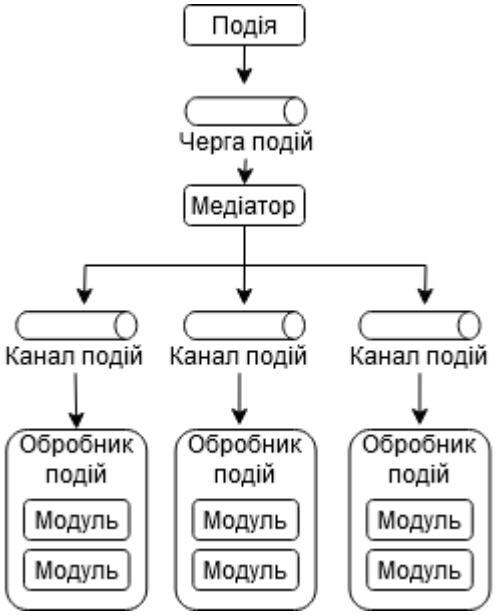

Рисунок 1.11 – Медіаторна топологія подійно-орієнтованої архітектури

Є два типи подій в даному патерні: первинна подія та сервісна подія. Первинна подія є оригінальною подією, надісланою на медіатор, в той час як сервісні події генеруються медіатором та отримуються компонентами обробки подій.

Подійний медіаторний компонент відповідальний за оркестрацію кроків, які містяться в первинній події.

Подійні канали використовуються медіаторами для асинхронної передачі специфічні сервісні події, які асоційовані з тим чи іншим кроком первинної події, до обробників подій. Подійні канали можуть бути організованими або в черги повідомлень або за темами. Найпоширенішим при цьому є тематичне розділення на канали, таким чином події можуть одночасно оброблюватись кількома обробниками подій. Компоненти обробників подій містять бізнес-логіку застосунку для того, щоб обробити необхідну поді. Ці обробники самодостатні,

незалежні, високо відділені архітектурні компоненти, які виконують специфічне завдання в застосунку чи системі.

Брокерна топологія, в свою чергу, відрізняється від медіаторної тим, що немає центрального подійного медіатора, а потік повідомлень розподіляється між обробниками подій у ланцюговий манер через полегшений брокер повідомлень.

Існує два головні види архітектурних компонентів з брокерною топологією: брокерні компоненти та компоненти обробників подій. Брокерні компоненти можуть бути як централізованими, так і децентралізованими та містять всі подійні канали, які використовуються в подійному потоці.

Подійні канали, які містяться всередині брокерних компонентів можуть мати черги повідомлень, теми повідомлень або комбінацію цих двох методів розподілу.

Дана топологія зображена на рисунку 1.12. Як можна побачити на діаграмі, тут відсутній центральний медіаторний компонент, який би контролював та оркестрував початкову подію, натомість кожен обробний компонент відповідальний за опрацювання події та генерацію нової події, яка б вказувала на виконання дії. При цьому можуть бути моменти, коли нова подія не приймається жодним обробником, що є дуже поширеним у процесі розвитку застосунку чи продумування на випадок розширення функціоналу [3].

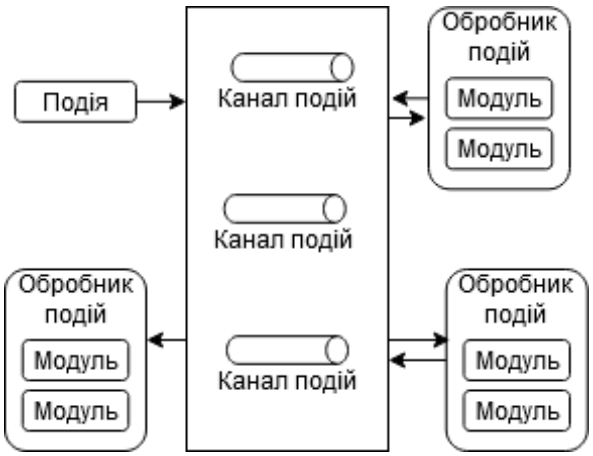

Рисунок 1.12 – Брокерна топологія подійно-орієнтованої архітектури.

Для того, щоб проілюструвати як працює брокерна топологія, можна скористатись прикладом, який ми використовували в медіаторній топології. Через відсутність централізованого медіатора для отримання первинної події в брокерній топології, компонент, який обробляє користувацькі події отримує їх напряму, змінює користувацьку адресу та надсилає подію, яка б казала, що користувацька адреса була змінена [3].

В даному прикладі два обробники мають інтерес в даній події: той, який цитує і той, який запитує.

Той компонент, який цитує, заново обчислює необхідні параметри згідно зміни адреси та публікує нову подію в систему, показуючи, що було зроблено. Той же компонент, який запитує, в свою чергу отримує подію зміни адреси, але в даному випадку, відбувається оновлення адреси та публікація події в систему як подія про запит обчислених даних. Ці нові події потім підібрані іншими компонентами, та ланцюжок подій проходить крізь систему допоки не знаходиться нова подія, яка була б згенерована для даної події.

Брокерна топологія більше про поєднання подій в ланцюжок для виконання бізнес-функції. Найкраща паралель із реального життя це естафета. В естафеті спортсмени отримують палочку, пробігають певну відстань та передають наступному, і так відбувається аж до подолання всієї естафетної дистанції. Так і в брокерній топології, де після отримання функції, один оброблювач виконує першу подію, передає результати обробки іншому оброблювачу і так допоки не будуть виконані необхідні дії для задоволення користувацького запиту [3].

1.7 Архітектура MACH

Архітектура MACH — це сучасний підхід до побудови цифрових застосунків, який пріоретизує гнучкість, масштабованість та швидкість [18]. Дана абревіатура означає набір керівних принципів даної архітектури:

- Microservices, використання мікросервісів
- API-first, пріоритезація API
- Cloud-native, застосунки розроблені спеціально для роботи в хмарі.
- Headless, відокремлення бізнес-логіки від користувацького інтерфейсу.

В архітектурі MACH, застосунки побудовані як колекція незалежних модульних сервісів, які комунікують між собою за допомогою API. Представницький рівень відділений при цьому від системи керування контентом та іншою бізнес-логікою. Таким чином, вміст застосунку може змінюватись незалежно від поведінки користувацької частини. Архітектура MACH дозволяє компаніям створювати застосунки, які легко підтримувати, масштабувати та пристосовувати до постійно змінюваних потреб, та надавати більш персоналізований та цікавий користувацький досвід [19].

Мікросервісна компонента в даній архітектурі має такі переваги, як швидкість та гнучкість в розгортанні та масштабуванні, дозволяє більше використовувати вже написаний код, та має зручність у підтримці. Також мікросервіси дозволяють скорочувати цикл розробки, бо різні функції можуть розроблятись та розгортатись окремо від інших. Як результат, мікросервісні застосунки користуються високою популярністю, включаючи сферу хмарних обчислень.

Стратегія пріоритезації API — це стратегія для розробки програмних продуктів, при якій прикладний програмний інтерфейс має пріоритет над іншими компонентами. Даний підхід має перевагу в тому, що достатньо розробити уніфікований API, який легко інтегрувати в інші застосунки, що дозволяє легше обмінюватись даними й функціоналом між різними системами. Також використання такого підходу дозволяє впевнитись, що користувацький інтерфейс постійний при даному API, що спрощує користувачам розуміти й використовувати застосунок.

Поняття “Cloud-Native” використовується для того, щоб описати застосунки, які розроблені спеціально для розгортання в хмарі. Такі застосунки типово побудовані з використанням мікросервісів, які являють собою малі

самодостатні компоненти, які можуть розгортатись і масштабуватись самостійно. Даний підхід відрізняється від традиційних великих та складних монолітних застосунків з тісно пов’язаними залежностями. Дані застосунки також розроблені для збільшення надійності, де кожен мікросервіс може незалежно замінюватись та масштабовуватись. Даний підхід дозволяє компаніям швидко адаптуватись до змінюваних умов та впевнитись в тому, що їхні застосунки залишаться доступними навіть в моменти несправностей.

Архітектура Headless — це тип програмної архітектури, в якій користувацький інтерфейс відділяють від основної логіки застосунку, використовуючи незалежну від фреймворків структуру. В такій системі, бізнес-частина надає API, який доставляє вміст сторінки до користувацького інтерфейсу. Дане розділення покращує гнучкість та масштабованість.

Відділяючи інтерфейс від логіки, розробники можуть простіше робити зміни кожної з частин не впливаючи на іншу. Даний підхід також спрощує масштабованість окремих компонентів системи за потреби [19].

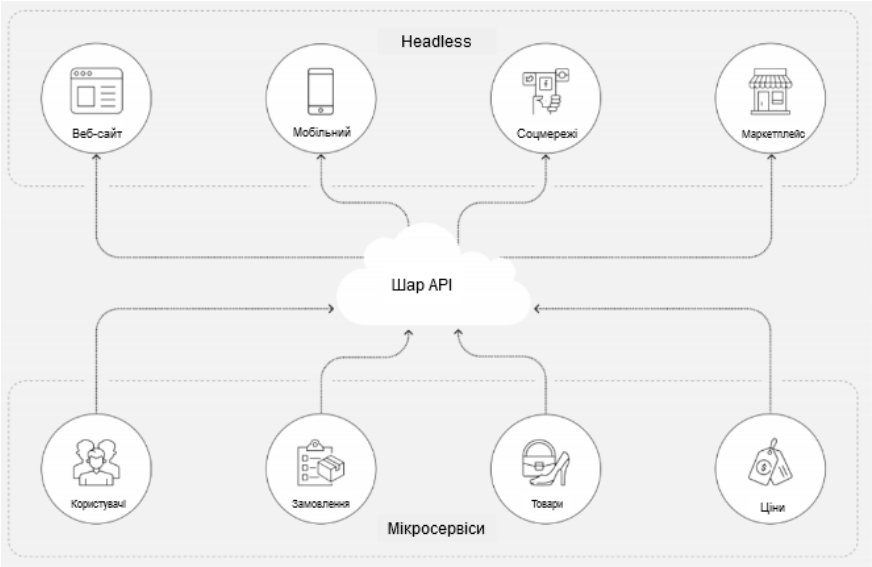

Рисунок 1.13 – Спрощена діаграма архітектури MACH

Спрощена діаграма даної архітектури показана на рисунку 1.13. На ній видно, що користувацький інтерфейс, представлений в різноманітних формах, повинен мати певну постійність у функціоналі, що забезпечується

відокремленням від бізнес-логіки за допомогою принципа переваги API. Вся логіка застосунку реалізована у мікросервісний манер. Хоча це й не показано на рисунку, самі мікросервісні компоненти розгорнуті на хмарі [20].

Хоча архітектура MACH є відносно новою, вона набуває все більшої популярності в провідних світових компаній та багато хто вбачає в ній майбутнє цифрового користувацького досвіду. Як результат, дана архітектура швидко набуває форми нового стандарту для застосунків [21].

Архітектура MACH є сучасним підходом до побудови застосунків, який пріоритезує гнучкість, швидкість та масштабованість. Використовуючи мікросервіси, пріоритет API, хмарні обчислення та відокремлення інтерфейсу від логіки, організації можуть створювати такі застосунки, які є легкими у підтримці, масштабованості та пристосуванні до змінюваних потреб бізнесу. Також ця архітектура дозволяє компаніям створювати більш персоналізований користувацький досвід та спрощує інтеграцію з іншими системами. Переходячи на архітектуру MACH, організації можуть випереджувати інших та розробляти застосунки з задатком на майбутнє, і які добре конкурують на сучасному IT ринку [18].

1.8 Порівняльний аналіз архітектур

Всі описані вище архітектури мають власні переваги і недоліки, деякі архітектури виникли як спроби компенсувати недоліки існуючих архітектур, і набули власних рис, а як наслідок і неідеальність людського мислення призводить до утворення нових недоліків, які в свою чергу компенсуються наступними архітектурами. Кожна з них завдяки своїм унікальним рисам є актуальною та оптимальною в своїй сфері.

Отже, оглянемо коротко переваги й недоліки кожної з описаних вище архітектур, та дамо оцінку за основними факторами: масштабованість, складність у підтримці (складність), гнучкість, продуктивність, простота у

розгортанні (DevOps), вартість та окреслимо сфери, де ті чи інші архітектури найкраще себе проявляють.

1) Багаторівнева архітектура.

Переваги:

- Проста, добре організована структура;
- Простота в керуванні малими проєктами;
- Зрозумілий розподіл ролей.

Недоліки:

- Може з часом набувати надмірної зв’язаності компонентів;
- Слабка масштабованість;
- Зміни в одному рівні впливають на інші.

Оцінка архітектурного підходу:

Масштабованість: Обмежена, підходить до малих та середніх застосунків.

Складність: Середня, легко імплементувати, але може бути сильно зв’язаним з часом.

Гнучкість: Слабка, додавання нових рівнів чи внесення змін потребує ґрунтовного рефакторингу.

Продуктивність: Може страждати від перевантаження міжрівневих зв’язків.

DevOps: Прості при першому розгортанні, але складно масштабувати.

Вартість: Малий при першому розгортанні, зростають витрати при необхідності подальшого ґрунтовного рефакторингу.

Сфера застосування: Для традиційних корпоративних застосунків.

2) Монолітна архітектура.

Переваги:

- Простота в розробці, тестуванні та первинному розгортанні;
- Добре підходить для малих та середніх застосунків;
- Мала складність для малих команд.

Недоліки:

- Складність у масштабуванні окремих компонентів;

- Надмірна зв’язаність. Одна помилка тягне за собою всю систему
- Складно адаптувати до змінного середовища.

Оцінка архітектурного підходу:

Масштабованість: Складність у масштабуванні окремих компонентів, проте вожливе вертикальне масштабування всієї системи.

Складність: Мала, збільшується при зростанні системи.

Гнучкість: Мала, зміни в одному компоненті має вплив на всю систему.

Продуктивність: Висока, падає при зростанні системи.

DevOps: Легко розгортати малі застосунки, складність збільшується при зростанні системи.

Вартість: Мала, зростають експлуатаційні витрати зі зростанням системи.

Сфера застосування: Для малих команді і стартапів.

3) Сервісно-орієнтована архітектура.

Переваги:

- Сервіси слабко зв’язані та можуть незалежно масштабовуватись;
- Код сервісів може повторно використовуватись в інших застосунках;
- Добре підходить для високих розподілених систем.

Недоліки:

- Складність в управлінні та підтримці;
- Комунікація між сервісами може викликати затримки;
- Вимагає високої дисципліни та стандартизації.

Оцінка архітектурного підходу:

Масштабованість: Висока, сервіси можуть масштабовуватись незалежно.

Складність: Висока, потребує координації кількох слабо зв’язаних сервісів.

Гнучкість: Висока, сервіси можуть розвиватись незалежно.

Продуктивність: Має затримки через комунікацію між сервісами.

DevOps: Складніше монолітної, потребує складні засоби оркестрації.

Вартість: Вища вартість початкового налаштування та експлуатаційні витрати, але має вищу довгострокову гнучкість.

Сфера застосування: Для великих корпоративних застосунків.

4) Мікросервісна архітектура.

Переваги:

- Незалежна масштабованість та розгортання;
- Незалежність технологічного стеку кожного з мікросервісів;
- Гнучкість та висока надійність.

Недоліки:

- Висока складність в керуванні розподіленими системами;
- Висока ймовірність перевантаження міжсервісної комунікації;
- Вимагає складної оркестрації та моніторингу.

Оцінка архітектурного підходу:

Масштабованість: Дуже висока, сервіси повністю відділені та масштабовані.

Складність: Дуже висока, потребує керування розподіленою системою.

Гнучкість: Дуже висока, кожен мікросервіс може бути побудованим, розгорнутим та масштабованим незалежно від інших.

Продуктивність: Сервіси можуть бути оптимізованими, але перевантаження комунікацій може давати затримки.

DevOps: Вимагає продвинутої оркестрації та CI/CD.

Вартість: Висока початкова вартість, але малі експлуатаційні витрати.

Сфера застосування: Для великих застосунків, які потребують в частому оновленні.

5) Подійно-орієнтована архітектура.

Переваги:

- Ідеально для обробки в реальному часі та асинхронної комунікації;
- Висока масштабованість та гнучкість;
- Слабка зв’язаність між компонентами.

Недоліки:

- Складність в розробці та керуванні;
- Складність у відладці та підтримці через асинхронність потоків;
- Вимагає уважного дотримання постійності подій.

Оцінка архітектурного підходу:

Масштабованість: Висока, можливе незалежне масштабування джерел та обробників подій.

Складність: Висока, вимагає налагодження асинхронної комунікації та постійності подій.

Гнучкість: Висока, відокремлення компонентів спрощує додавання нових обробників подій.

Продуктивність: Відмінна для асинхронних даних в реальному часі

DevOps: Вимагає обережного моніторингу потоку подій, важко відлагоджувати.

Вартість: Висока початкова вартість, низькі експлуатаційні витрати для обробки подій в реальному часі.

Сфера застосування: Для систем обробки в реальному часі, таких як інтернет речей чи фінанси.

6) Архітектура MACH.

Переваги:

- Висока гнучкість та масштабованість;
- Висока оптимізація для хмарних обчислень та веб-застосунків;
- Підтримка швидкої розробки та інтеграції з різними інтерфейсами.

Недоліки:

Висока початкова складність та вартість;

Вимагає складної хмарної інфраструктури та керування;

Потребує чіткої орекстрації API та сервісів.

Оцінка архітектурного підходу:

Масштабованість: Дуже висока, розроблена для динамічного масштабування розгорнутого у хмарі застосунку.

Складність: Висока, включає в себе багато змінних компонентів (мікросервіси, API, хмарні обчислення).

Гнучкість: Дуже висока, може інтегруватись з більшістю систем.

Продуктивність: Висока, з оптимізованою комунікацією між сервісами через API.

DevOps: Дуже динамічна, вимагає складної оркестрації та управління хмарними сервісами.

Вартість: Висока початкова вартість, вигідно у достроковій перспективі завдяки гнучкості.

Сфера застосування: Електронна комерція або інші системи для великої кількості користувачів, які швидко розвиваються.

Зведемо результати в порівняльну таблицю 1.1.

Таблиця 1.1 – Порівняльний аналіз найпоширеніших архітектур

Критерій	Багаторівнева	Монолітна	SOA	Мікросервіси	EDA	MACH
Масштабованість	Низька	Середня	Висока	Дуже висока	Висока	Дуже висока
Складність	Середня	Низька	Висока	Дуже висока	Висока	Висока
Гнучкість	Низька	Низька	Висока	Дуже висока	Висока	Дуже висока
Продуктивність	Середня	Середня	Низька	Висока	Висока	Висока
DevOps	Проста	Проста	Складна	Складна	Складна	Складна
Вартість	Низька	Низька	Висока	Висока	Висока	Висока
Сфера застосування	Корпоративні рішення	Невеликі системи	Великі корпорації	Великі застосунки	Системи в реальному часі	Електронна комерція

2 АНАЛІЗ ОБРАНОГО ЗАСТОСУНКУ

В даній роботі для розгортання було обрано один з найбільш популярних веб-застосунків під назвою WordPress, який забезпечує роботу 43% веб-сайтів по всьому світу.

Цей застосунок був розроблений в 2003 році Майком Літтлом та Меттом Муленвіґом як гілка застосунку b2/cafelog, призначеного для блогінгу. Потреба в елегантному добре спроєктованому застосунку для розміщення особистих публікацій була нагальна ще в ті часи. На сьогодні, WordPress побудований із застосуванням PHP та MySQL, та розповсюджується за ліцензією GPLv2 [22].

Особливість даного застосунку полягає в тому, що люди з обмеженим технічним досвідом можуть використовувати цей застосунок “з коробки” без жодних проблем; просунуті ж користувачі можуть підлаштовувати застосунок під власні цілі.

В даній роботі ми розглянемо розгортання варіанту “з коробки” у двох найпоширеніших архітектурних підходах – монолітний та мікросервісний.

2.1 Внутрішня архітектура застосунку

WordPress являє собою просту систему, яка поділена на наступні структурні компоненти:

- публікації блогу (пости);
- сторінки;
- користувачі;
- коментарі;
- тема;
- плагіни;
- налаштування;

- медіа, інструменти та ін. [23]

Застосунок має доступ до всіх цих частин, і навпаки, при створенні власних плагінів та тем, є доступ до всіх частин WordPress до застосування в коді. Основною філософією застосунку (яку ми, втім, коротко оглянемо при розгляді даного питання) є необмежені можливості користувачів до створення власних модулів для розширення функціоналу застосунку.

Загальна структура застосунку зображена на рисунку 2.1. При розгортанні застосунку, в користувача є доступ до активної теми, множини плагінів, постів блогу, різних сторінок, користувачів, медіа файлів та налаштувань [23].

Кожен з даних прямокутників, умовно назвемо модулями.

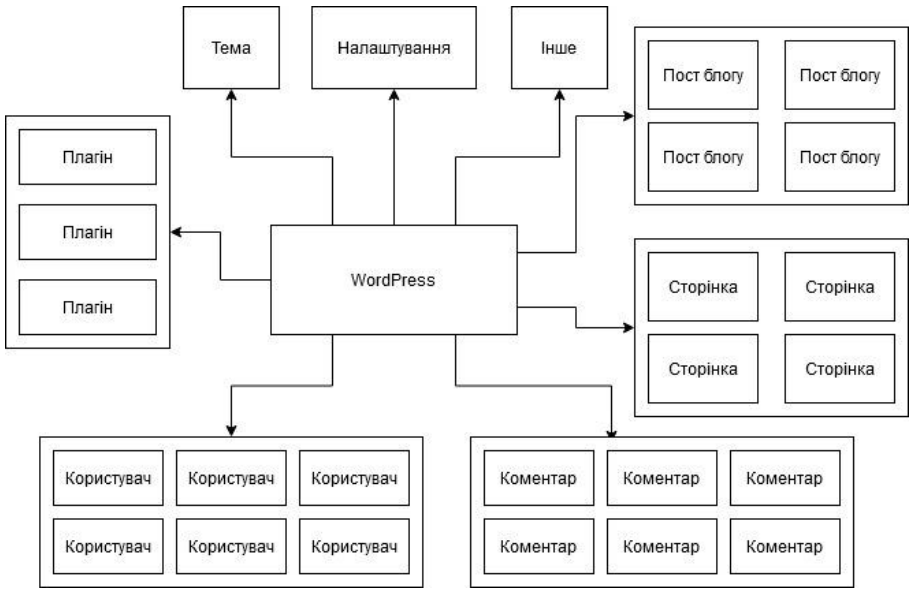

Рисунок 2.1 – Структура додатку WordPress [23].

Модулі працюють наступним чином. Кожен модуль являє собою список. Наприклад – користувачів. Кожен користувач представлений в списку користувачів, пости представлені у списку постів, тощо. Таким чином доволі легко працювати з модулями.

Кожен модуль можна редагувати за допомогою Панелі Адміністратора. Також ця панель дає доступ до інших інтерфейсів, наприклад “Customize”, де редагується власне тема інтерфейсу веб-застосунку.

Плагіни та теми можуть бути відредаговані за допомогою Панелі Адміністратора, або написані самотужки. Плагіни написані мовою PHP. Теми написані мовами PHP, HTML, CSS, JS. Під час написання модулів, є можливість дуже легко викликати за необхідності за допомогою власного PHP API. [23]

2.2 Структура бази даних WordPress.

Стандартна інсталяція WordPress містить в собі 11 таблиць бази даних, таким чином цей застосунок вважається дуже ощадливим стосовно баз даних. Сама структура розроблена таким чином, щоб бути дуже мінімальною, при цьому маючи високу гнучкість при розробці різних модулів для WordPress. Для спрощення розуміння структури, на рисунку 2.2 наведена блок-схема бази даних для даного застосунку. Ця схема ілюструє структуру баз даних, яка утвроюється під час встановлення застосунку. Втім, окремі плагіни та теми можуть створювати власні таблиці таким чином, структура бази даних може відрізнятись на практиці. [24]

Нижче наведені назви таблиць та те, які дані вони зберігають:

- wp_comments: Містить всі коментарі додатку. Кожен з коментарів прикріплюється до поста блогу за допомогою ідентифікатора поста;

- wp_commentsmeta: Містить метадату для коментарів;

- wp_links: Містить всі посилання, які створені за допомогою Link Manager;

- wp_options: Зберігає всі налаштування веб-сайту, які були визначені в підпанелі Settings. Також зберігає налаштування плагінів, активні плагіни та теми, тощо;

- wp_postmeta: Містить всю метадату постів (окремі поля);

- wp_posts: Містить всі пости, сторінки, записи медіа та редагування. В більшості випадків це найбільша таблиця в базі даних;

- wp_terms: Містить всі таксономічні терміни, які були визначені для вебсайту;

- wp_term_relationships: Поєднує таксономічні терміни зі вмістом (пости, посилання, тощо);

- wp_term_taxonomy: Визначає, до чого призначений той чи інший таксономічний термін;

- wp_users: Містить дані користувачів, які створені на веб-сторінці (логін, пароль, адреса електронної пошти);

- wp_usermeta: Містить метадані користувачів (прізвище-ім’я, нікнейм, рівень користувача, тощо).

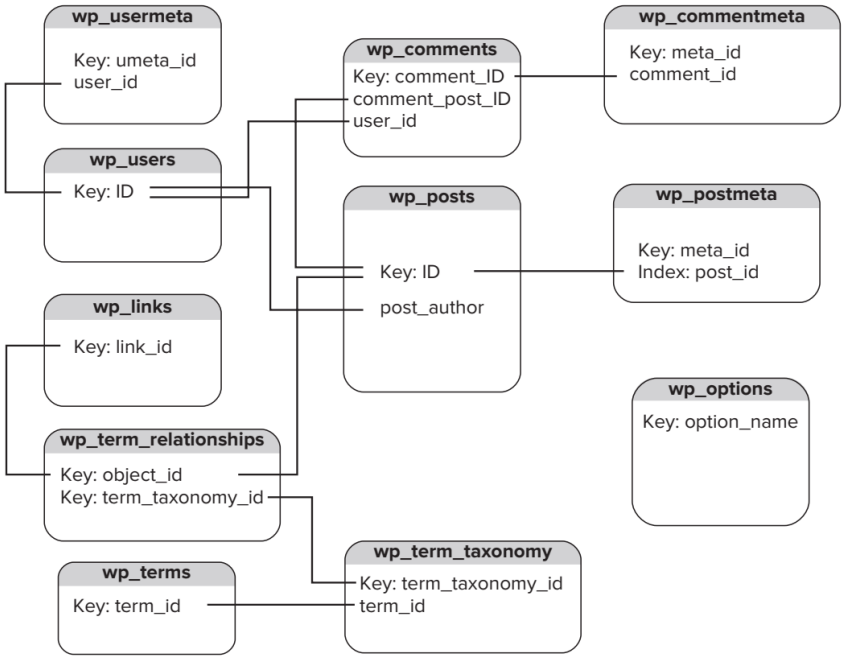

Рисунок 2.2 – Архітектура бази даних [24].

Стуктура таблиць у WordPress дуже послідовна. Кожна таблиця в базі даних містить поле з унікальним ідентифікатором, що є головним ключем таблиці. Також кожна таблиця містить один або більше індексів на полях, що покращує швидкість отримання даних під час запитів.

Найважливішим полем кожної таблиці є поле з унікальним ідентифікатором. Це поле не завжди має назву ID, але є полем з автоматичним інкрементом, який двє кожному запису в таблиці унікальний ідентифікатор. Наприклад, при першому встановленні застосунку, створюється найперший пост

блогу під назвою “Hello world!”. Оскільки це перший пост, утворений у таблиці wp_posts, ідентифікатор, призвоєний цьому посту має значення 1. Кожен пост має унікальний ідентифікатор, який може використовуватись для того, щоб завантажити інформацію специфічну для тої чи іншої публікації, а також може використовуватись для поєднання даних з різних таблиць в базі даних.

Втім, існує один нюанс з нумерацією, пов’язаний з редагуванням публікацій та вкладеннями до них. Оскільки кожне редагування та вкладення зберігаються як окремий запис в таблиці wp_posts, кожній ревізії публікації та вкладенню присвоюється власний унікальний ідентифікатор, що означає, що ідентифікатори публікації постів не слідують один за одним. Наприклад, найперший пост може мати номер 1 в базі даних, в той час як наступна публікація може мати номер 15. Це залежить від кількості вкладень та редагувань було створено між цими двома публікаціями [24].

2.3 Особливості розгортання застосунку

Оскільки на практиці, розгортання інфраструктури під різні архітектури виконується за допомогою інструменту, що дозволяє автоматично розгортати хмарну інфраструктуру за заздалегідь написаними специфікаціями, під час виконання даної роботи, ми використовуватимемо інструмент під назвою Terraform. Цей інструмент дозволяє працювати з різними хмарними провайдерами, та має високу гнучкість і функціонал та регулярно оновлюється по взаємодії з такими компаніями.

Виконання даної роботи відбуватиметься в навчальному середовищі на базі хмари AWS, яке надає доступ до наступних ресурсів:

- екземпляри EC2 t2 та t3 розмірів від t.nano до t.medium;
- томи EBS – розміром понад 35ГіБ та типу Genetal Purpose SSD (gp2);
- операційни системи – Amazon Linux та Windows;

– класи екземплярів баз даних – db.t2 та db.t3 розміром від db.t.micro до db.t.medium;

– тільки одна зона доступу;

– розміри томів – до 100ГіБ типу General Purpose SSD (gp2);

– можливі рушії бази даних – Amazon Aurora, MySQL, PostgreSQL та MariaDB;

– автоматичне масштабування EC2;

– CloudFormation;

– CloudWatch;

– обмежений доступ до S3 (можливі проблеми з Terraform);

– обмежений доступ до Glacier (можливі проблеми з Terraform);

– балансувальник навантаження Elastic Load Balancer;

– Route53;

– CloudFront;

– AWS Lambda;

– Amazon DynamoDB;

– CloudTrail.

Спираючись на даний перелік можливостей середовища, необхідно виконувати планування розгортання застосунку. Як можна побачити, тут немає доступу до EKS, що дає можливість використання технології Kubernetes. Це накладає обмеження на можливість створення наближених до реальності складних мікросервісних архітектур, оскільки доступні типи й розміри екземплярів EC2 забезпечують вкрай обмежену роботу Kubernetes [25][26].

Розглянувши середовище, перейдемо до особливостей розгортання застосунку. Як було вказано вище, WordPress в стандартній конфігурації являє собою монолітний застосунок. Це спрощує написання специфікації для монолітної архітектури.

Втім, розгортання мікросервісної архітектури вкрай обмежене. По-перше, через відсутність доступу до EKS, налаштування автоматично балансованої

архітектури розгорнутої в контейнерах. По-друге, через вкрай обмежений доступ до S3 (сервіс який надає доступ до файлового сховища), відсутня можливість автоматичного розгортання даного ресурсу для перенесення частини файлів, наприклад, веб-сторінок з контейнерів чи екземплярів в дане сховище. По-третє, обмежені обчислювальні можливості в свою чергу обмежують горизонтальне масштабування.

Враховуючи дані зауваження, перейдемо до практичної частини нашої роботи.

3 РОЗГОРТАННЯ МОНОЛІТНОЇ АРХІТЕКТУРИ

3.1 Інфраструктура монолітного застосунку

Почнемо розгортання монолітної архітектури WordPress з планування хмарної інфраструктури. В даній роботі буде найпростіше розгортання на базі екземпляру EC2, на якому працюватиме додаток, а також екземпляр RDS з базою даних для застосунку. Також необхідно створити групи безпеки для цих екземплярів, щоб обмежити доступ до них ззовні.

Стосовно рушія бази даних та налаштування екземпляру, задля встановлення необхідних залежностей для застосунку, звернемося до системних вимог до WordPress [27]. Вимоги наступні:

– підтримка PHP версії 7.4 або вище;

– MySQL версії 8.0 та вище, або MariaDB версії 10.5 або вище (можлива підтримка MySQL версії 5.5.5 або вище);

– підтримка HTTPS;

– рекомендовано використання веб-серверу Apache або Nginx.

Таким чином, ми маємо змогу спланувати інфраструктуру застосунку. Вона зображена на рисунку 3.1.

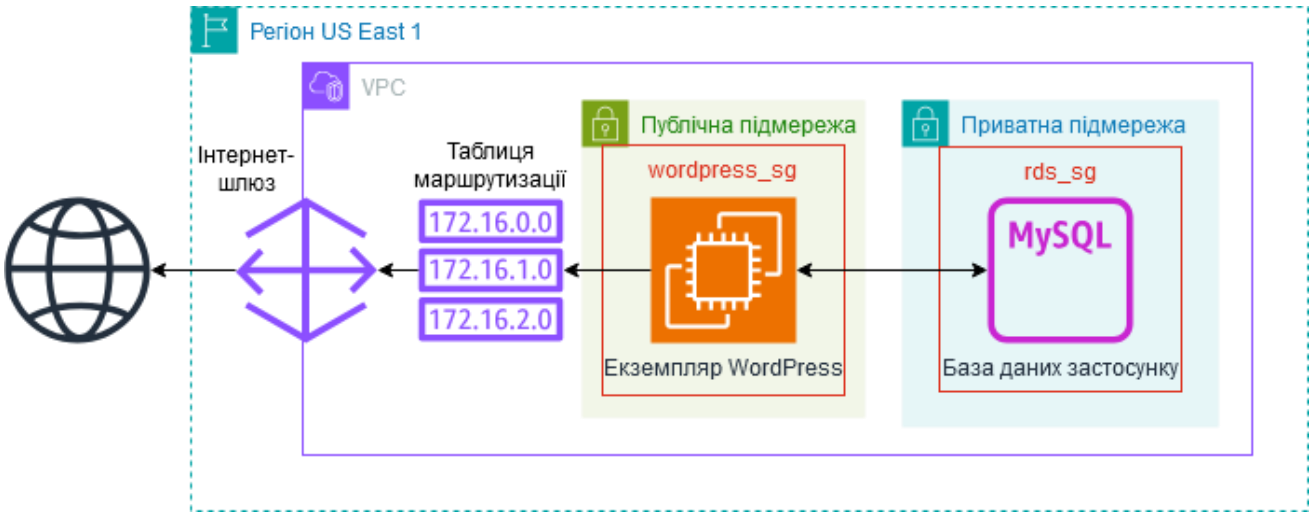

Рисунок 3.1 – Інфраструктура монолітного застосунку

В одному регіоні розташуємо по екземпляру EC2 та RDS, кожній з них створена та присвоєна власна група безпеки а також відповідні підмережі всередині віртуальної хмарної мережі (VPC). Для доступу до мережі Інтернет, необхідно створити шлюз та таблицю маршрутизації, яка б дозволяла користуватись даним шлюзом.

3.2 Розгортання монолітного застосунку

Специфікація для розгортання застосунку умовно поділена на ряд структурних компонентів, які спрощують розуміння та редагування коду. Це змінні та конфігурація середовища, мережна складова, групи безпеки, та власне екземпляри EC2 та RDS.

Почнемо зі змінних. В специфікації було визначено три змінні, які допомагатимуть при автоматичній конфігурації застосунку. Це назва бази даних (db_name), логін (db_username) і пароль адміністратора (db_password). Частину цих змінних зберігати в такому вигляді є небезпечним, такі ризики пов’язані умовами середовища, які роблять взаємодію багатьох компонентів непередбачуваною.

Перейдемо до груп безпеки. Для екземпляру EC2 (wordpress-sg) надаємо доступ за протоколом TCP за портами 80 (для HTTP) та 22 (для дистанційного доступу за SSH), для висхідного потоку обмеження відсутні. Для бази даних (rds-sg) вхідний потік відкриємо виключно за TCP портом 3306, для висхідного потоку обмеження відсутні.

Стосовно мережної складової, створимо хмарну віртуальну приватну мережу (VPC).

```
resource "aws_vpc" "main" {
  cidr_block = "10.0.0.0/16"
```

```
  tags = {
    Name = "main-vpc"
  }
}
```

Для можливості доступу зсередини віртуальної мережі до мережі Інтернет, також створимо шлюз. Для екземпляру EC2 створимо публічну мережу та для маршрутизації зі шлюзом створимо таблицю маршрутизації. Створимо приватну підмережу для бази даних застосунку. Також налаштуємо відповідні асоціації.

Таким чином була створена мережна частина застосунку.

Екземпляр RDS, створимо з наступними параметрами:

1. Виділений об’єм – 20 ГіБ;
2. Рушій бази даних – MySQL;
3. Версія рушія бази даних – 5.7 (зумовлено подальшими помилками сумісності при налаштуванні екземпляру EC2);
4. Тип екземпляру – db.t3.micro;
5. Назва бази даних – визначена змінною db_name;
6. Ім’я користувача – визначене змінною db_username;
7. Пароль користувача – визначений змінною db_password;
8. Доступ з кількох зон доступності – заборонений;
9. Публічинй доступ – заборонений;
10. Група безпеки – rds-sg.

Таким чином, була створена працююча база даних.

Перейдемо до екземпляру EC2. Було створено екземпляр з наступними параметрами, які забезпечують нормальну роботу застосунку:

1. Операційна система – AWS Linux 2;
2. Тип екземпляру – t2.micro;
3. Ключова пара – vockey (стандартна ключова пара згенерована середовищем);
4. Група безпеки – wordpress-sg.

Таким чином, ми маємо базу для застосунку. Самий застосунок розгортається та налаштовується за допомогою користувацького скрипту, написаного мовою Bash.

Перш ніж встановити сам застосунок, необхідно провести підготовчі дії, а саме оновити наявні пакети, та встановити необхідні залежності; запустити веб-сервер Nginx та обробник запитів PHP-FPM.

```
yum update -y

amazon-linux-extras enable nginx1 php8.0
yum install -y nginx php php-fpm php-mysqlnd wget unzip mysql

systemctl enable nginx
systemctl enable php-fpm
```

Після цього налаштуємо веб-сервер та обробник запитів.

```
sed -i 's|^listen = .*|listen = /var/run/php-fpm/www.sock|' /etc/php-
fpm.d/www.conf

cat >> /etc/php-fpm.d/www.conf <<-EOL
listen.owner = nginx
listen.group = nginx
EOL

systemctl restart php-fpm

cat > /etc/nginx/conf.d/wordpress.conf <<-EOL
server {
    listen 80;
    server_name _;
    root /usr/share/nginx/html;
    index index.php index.html index.htm;

    location / {
        try_files \$uri \$uri/ /index.php;
```

```
    }

    location ~ \.php\$ {
        include /etc/nginx/fastcgi_params;
        fastcgi_pass unix:/var/run/php-fpm/www.sock;
        fastcgi_index index.php;
        fastcgi_param SCRIPT_FILENAME \$document_root\$fastcgi_script_name;
    }
}
EOL

systemctl restart nginx
```

Після того, як були налаштовані веб-сервер та обробник, проведемо встановлення застосунку на екземпляр EC2. Для цього необхідно завантажити архів з кодом, розпакувати його та встановити дозволи на виконання файлів застосунку.

```
wget https://wordpress.org/latest.tar.gz
tar -xzf latest.tar.gz
mkdir -p /usr/share/nginx/html
cp -r wordpress/* /usr/share/nginx/html/
chown -R nginx:nginx /usr/share/nginx/html
```

Оскільки необхідно мінімізувати людський фактор при налаштуванні застосунку, проведемо також конфігурацію, використовуючи змінні та дані, отримані в результаті розгортання інфраструктури застосунку.

```
cat > /usr/share/nginx/html/wp-config.php <<-EOL
<?php
define('DB_NAME', '${var.db_name}');
define('DB_USER', '${var.db_username}');
define('DB_PASSWORD', '${var.db_password}');
define('DB_HOST', '${aws_db_instance.mysql.endpoint}');
define('DB_CHARSET', 'utf8');
define('DB_COLLATE', '');
```

```
// Authentication Unique Keys and Salts (Predefined for simplicity)
define('AUTH_KEY',         '$(openssl rand -base64 32)');
define('SECURE_AUTH_KEY',  '$(openssl rand -base64 32)');
define('LOGGED_IN_KEY',    '$(openssl rand -base64 32)');
define('NONCE_KEY',        '$(openssl rand -base64 32)');
define('AUTH_SALT',        '$(openssl rand -base64 32)');
define('SECURE_AUTH_SALT', '$(openssl rand -base64 32)');
define('LOGGED_IN_SALT',   '$(openssl rand -base64 32)');
define('NONCE_SALT',       '$(openssl rand -base64 32)');

\$table_prefix = 'wp_';

// Absolute path to the WordPress directory.
if (!defined('ABSPATH'))
    define('ABSPATH', __DIR__ . '/');

// Sets up WordPress vars and included files.
require_once ABSPATH . 'wp-settings.php';
EOL

// Absolute path to the WordPress directory.
if (!defined('ABSPATH'))
    define('ABSPATH', __DIR__ . '/');

// Sets up WordPress vars and included files.
require_once ABSPATH . 'wp-settings.php';
EOL

chown nginx:nginx /usr/share/nginx/html/wp-config.php
```

Після цього необхідно перезапустити веб-сервер та обробник запитів для застосування нової конфігурації.

```
systemctl restart nginx
systemctl restart php-fpm
```

Таким чином була проведена конфігурація екземпляру EC2, завершаючи специфікацію. Для перевірки даної конфігурації, проведемо розгортання застосунку за допомогою інструменту Terraform (рис. 3.2).

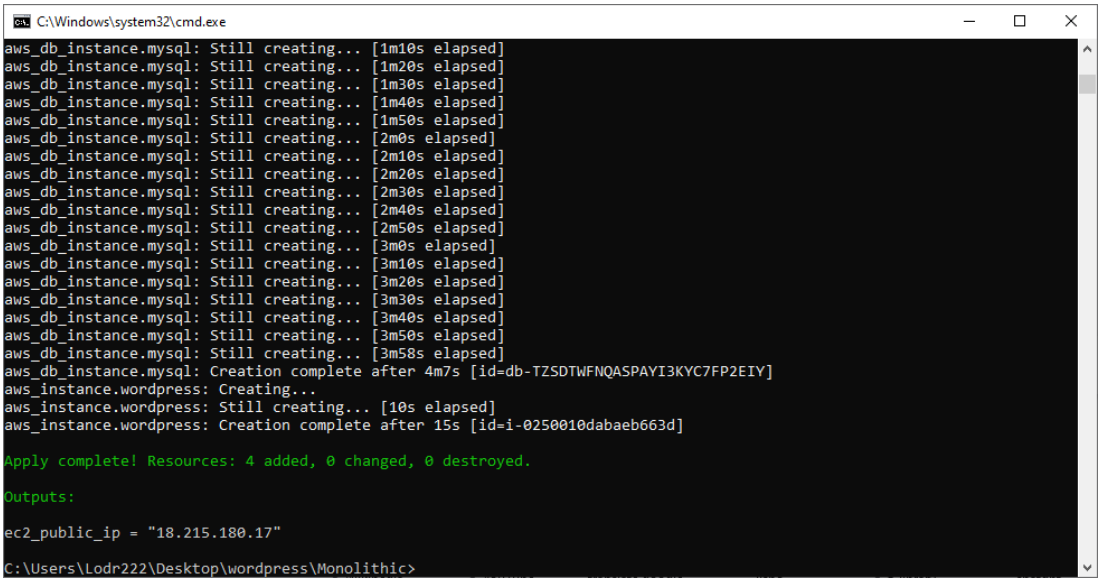

Рисунок 3.2 – Виведення консолі

Після завершення розгортання та виконання користувацького скрипту, зайшовши за публічною IP-адресою екземпляра за протоколом HTTP, отримаємо наступну сторінку (рис. 3.3).

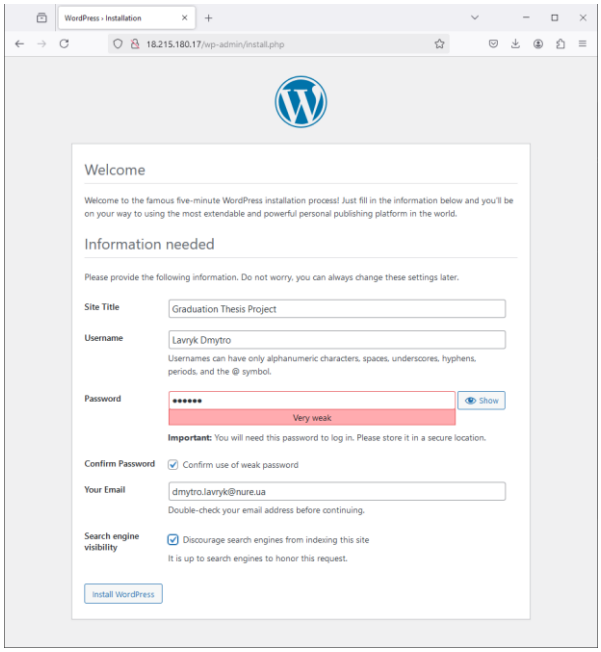

Рисунок 3.3 – Сторінка створення веб-сайту на базі застосунку WordPress

Для перевірки працездатності бази даних, завершимо налаштування назви сайту та подивимось на наявні публікації (рис.3.4). При справній роботі бази даних повинна бути найперша публікація з назвою “Hello, world!”.

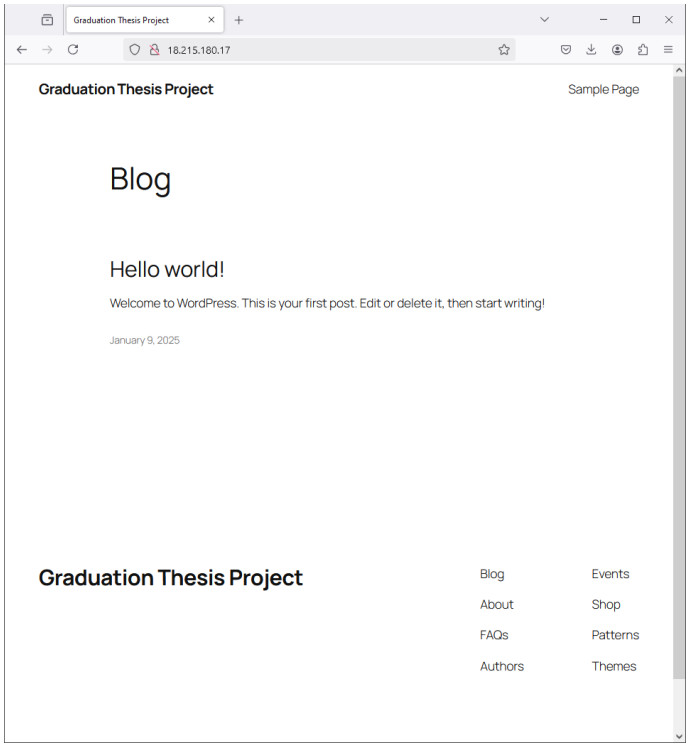

Рисунок 3.4 – Головна сторінка веб-сайту, створеного застосунком Wordpress

Таким чином, під час виконання даного розділу, ми спланували найпростішу інфраструктуру для розгортання застосунку, написали специфікацію та організували автоматичне розгортання; була виконана перевірка працездатності WordPress.

4 РОЗГОРТАННЯ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

4.1 Інфраструктура мікросервісного застосунку

Як було вказано в розділі 1, мікросервісний застосунок складається з ряду слабо пов’язаних компонентів, які можуть незалежно масштабуватись.

Відповідно змінимо архітектуру нашого застосунку. Ця видозмінена інфраструктура має наступний вигляд, як показано на рисунку 4.1

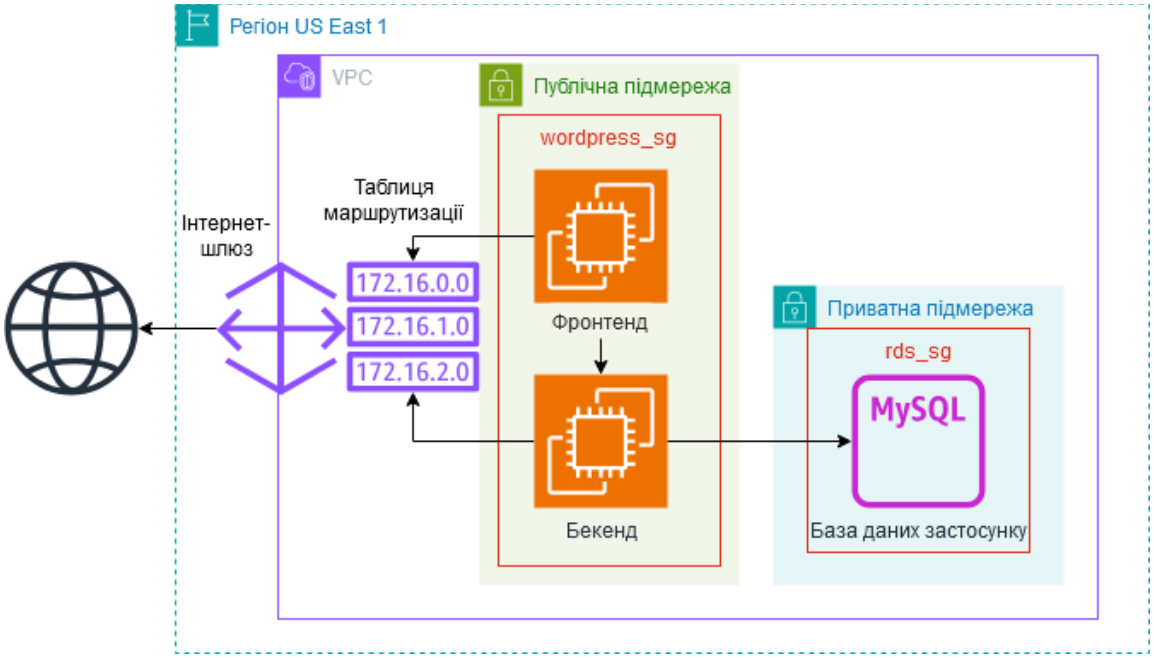

Рисунок 4.1 – Інфраструктура мікросервісного застосунку.

Як показано на схемі, ми виділили наступні два мікросервіси: фронтенд, який має в собі веб-сервер Nginx, який спрямовуватиме запити самому сайту, та бекенд, який міститиме в собі основну логіку застосунку та обробник запитів PHP-FPM. Кожен з цих сервісів розташований на власному екземплярі EC2, збільшення кількості екземплярів вимагає більш складної мережної складової, тому створимо дві приватні мережі, вихід до зовнішньої мережі при цьому відбувається через шлюз, щоб це все працювало необхідно також описати таблицю маршрутизації. Комунікація між сервісами відбуватиметься за допомогою внутрішньої мережі хмари.

4.2 Розгортання мікросервісної архітектури.

Після планування інфраструктури, приступимо до модифікації наявної специфікації. Такі ресурси як змінні, параметри середовища, групи безпеки та екземпляр бази даних залишимо незмінними. Стосовно мережної частини, змінимо асоціації підмереж таким чином, щоб була комунікація між мікросервісами.

Перейдемо до екземплярів EC2. За параметрами вони ідентичні монолітному застосунку. Почнемо зі створення користувацького скрипту для фронтенду. Для початку встановимо необхідні програмні пакети. Сервіс розгорнемо на койнтейнері, тому встановимо Docker та Git для роботи з репозиторіями на GitHub.

```
amazon-linux-extras enable docker
yum install -y docker git
systemctl enable docker
systemctl start docker
```

Створимо Dockerfile.

```
FROM nginx:alpine

COPY nginx.conf /etc/nginx/nginx.conf

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]
```

Запустимо контейнер.

```
docker build -t thesis-frontend /home/ec2-user/frontend
```

```
docker run -d --name wordpress-frontend --network wordpress-network -p 80:80
thesis-frontend
```

Таким чином, ми маємо екземпляр EC2 з налаштованим контейнером, який братиме файли з контейнеру бекенду, який розташований на іншому екземплярі, комунікація з яким відбуватиметься всередині хмарної мережі.

Перейдемо до бекенду. Напишемо користувацький скрипт для відповідного екземпляру EC2. Встановимо залежності.

```
amazon-linux-extras enable docker
yum install -y docker git
systemctl enable docker
systemctl start docker
```

Вміст Dockerfile наступний.

```
FROM php:8.0-fpm

RUN apt-get update && apt-get install -y \
    libjpeg62-turbo-dev \
    libpng-dev \
    libfreetype6-dev \
    libzip-dev \
    unzip \
    wget \
    && docker-php-ext-configure gd --with-freetype --with-jpeg \
    && docker-php-ext-install gd zip mysqli pdo pdo_mysql

WORKDIR /var/www/html

RUN wget https://wordpress.org/latest.tar.gz && \
    tar -xzf latest.tar.gz --strip-components=1 && \
    rm latest.tar.gz

RUN cp wp-config-sample.php wp-config.php && \
    sed -i "s/database_name_here/${var.db_name}/g" wp-config.php && \
    sed -i "s/username_here/${var.db_username}/g" wp-config.php && \
```

```
    sed -i "s/password_here/${var.db_password}/g" wp-config.php && \
    sed -i "s/localhost/${aws_db_instance.mysql.endpoint}/g" wp-config.php

RUN chown -R www-data:www-data /var/www/html

EXPOSE 9000

CMD ["php-fpm"]
```

Після чого, створюємо та запускаємо контейнер.

```
docker build -t thesis-backend /home/ec2-user/backend
docker run -d --name wordpress-backend --network host thesis-backend
```

Таким чином, ми створили специфікацію для нашого мікросервісного застосунку. Проведемо розгортання даного застосунку на хмарі та протестуємо доступність та роботу бази даних. Спочатку розгорнемо застосунок (рис. 4.2).

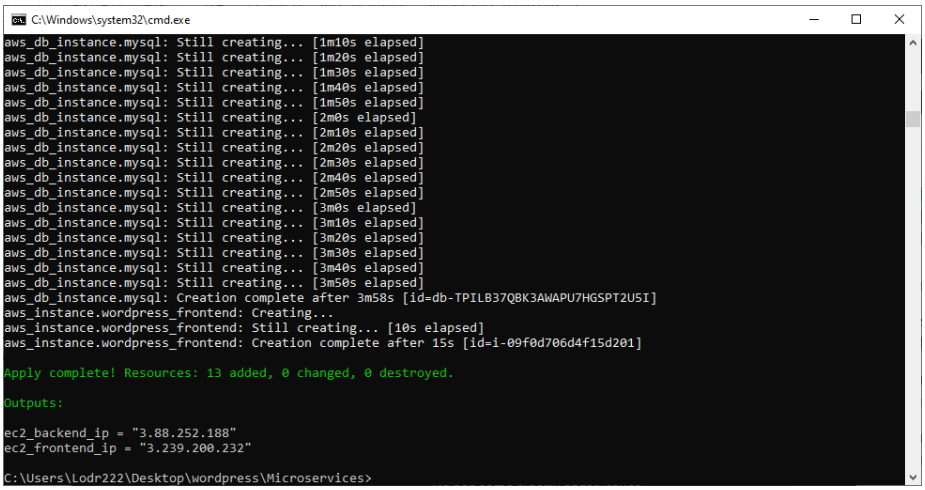

Рисунок 4.2 – Процес розгортання мікросервісного застосунку

Після розгортання, перейдемо за IP адресою за протоколом HTTP. Маємо отримати сторінку встановлення застосунку. Втім, як ми можемо побачити на рисунках 4.3 та 4.4, сторінка відрізняється за оформленням. Це пов’язане з тим, що через обмеження у виділених ресурсах на контейнер, застосунок переходить в режим підтримки старішої версії.

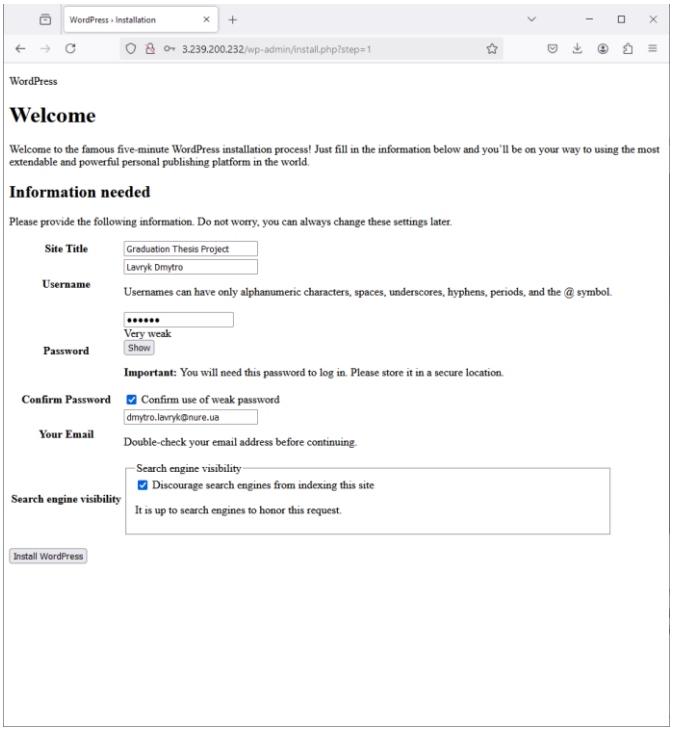

Рисунок 4.3 – Сторінка встановлення веб-сторінки на базі застосунку Wordpress

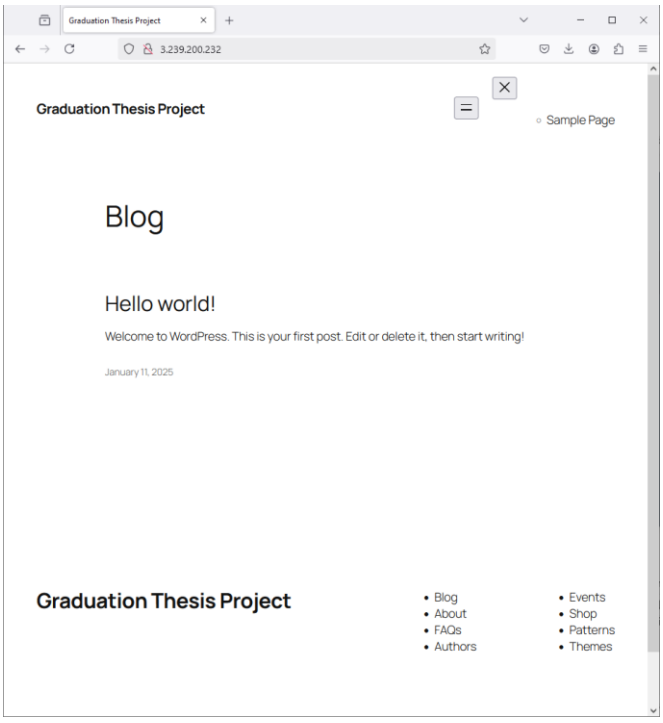

Рисунок 4.4 – Головна сторінка застосунку, розгорнутого за мікросервісною схемою.

Таким чином, ми спланували зміни в архітектурі та розробили відповідну специфікацію для Terraform, за якою успішно розгорнули веб-застосунок та випробували працездатність.

5 ДОСЛІДЖЕННЯ ХАРАКТЕРИСТИК РОЗГОРНУТИХ АРХІТЕКТУР

5.1 Опис середовища

Дане дослідження виконуватиметься в навчальному середовищі, умови та обмеження якого описані в розділі 2.3. Самі конфігурації були описані в розділах 3 та 4. Втім, для збору даних, необхідно виконати деякі зміни. Для моніторингу було використано хмарний сервіс CloudWatch.

Для генерації запитів HTTP було використано інструмент Apache JMeter. Такий інструмент є доволі гнучким і дозволяє ретельно перевірити застосунок на наявність проблем зі звичайним користувацьким трафіком. Всі зібрані дані після цього збирались в форматі CSV для подальшого аналізу.

5.2 Дослідження затримок та використання ресурсів

Для даного експерименту було виконано 5 досліджень, під час аналізу використовували усереднені значення з даних 5 спроб. Спроби являють собою серії симуляції запитів від 90 користувачів, кількість яких збільшується арифметично кожні 5 секунд.

Почнемо з аналізу затримок, дані яких були отримані за допомогою Apache JMeter (рис. 5.1). При цьому, запити мікросервісного застосунку надходили на фронтенд, і тут включається час обробки запиту всією системою, від надсилання запиту до отримання відповіді.

Рисунок 5.1 – Затримки при обробці запитів.

Як бачимо, монолітний застосунок має значно менші затримки. Це пов’язано з тим, що монолітна архітектура не потребує комунікації мережею, мережний обмін дещо збільшує затримки. Мікросервісна архітектура має вищий час, що пов’язано зі встановленням з’єднань між екземплярами та передачею даних між ними.

Перейдемо до використання ресурсів. Були виконані вимірювання використання ресурсів із фіксацією кожні 5 секунд, проміжки збору даних були синхронізовані з Apache JMeter. Почнемо з використання ЦП. На рисунку 5.2 наведено графік завантаженості центрального процесора залежно від кількості одночасних запитів.

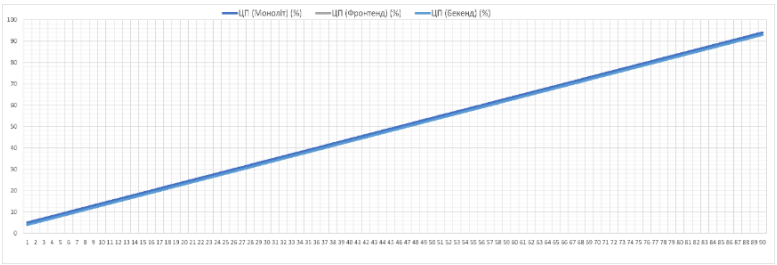

Рисунок 5.2 – Завантаженість центральних процесорів екземплярів EC2

Як можна побачити, тут спостерігається лінійна залежність, і рівень завантаження майже не відрізняється. Дивним чином завантаження рівномірно розподіляється між екземплярами незалежно від експерименту. На рисунку 5.3 наведене завантаження оперативної пам’яті екземплярів EC2.

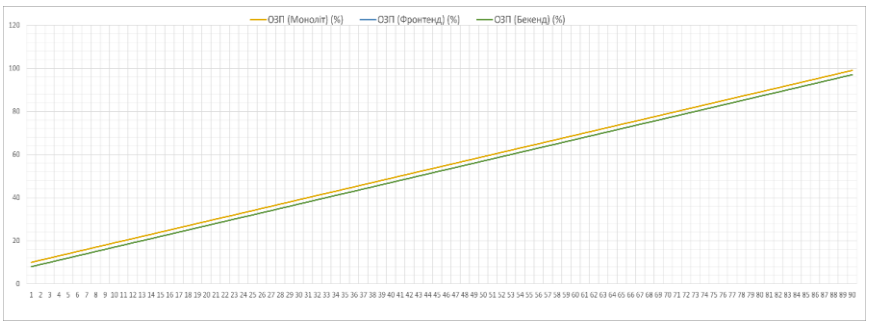

Рисунок 5.3 – Завантаженість ОЗП екземплярів EC2

Тут теж можна простежити схожу залежність. Втім, монолітна архітектура має трохи більшу завантаженість ресурсів. Втім, підсумувавши, загальне використання ресурсів мікросервісної архітектури вище за монолітну. Найбільш імовірно, це пов’язано з невдалим плануванням інфраструктури та розбиттям застосунку на сервіси.

Як підсумок, в плані затримок монолітна архітектура має найменші затримки при обробці запитів та краще використовує наявні ресурси.

5.3 Дослідження надійності розгорнутих застосунків

Дослідимо надійність систем в двох експериментах: ймовірність помилки при звичайному користувацькому трафіку та вимірювання часу відсутності сервісу при різних сценаріях розвитку небажаних для нас подій.

Для першого експерименту використовуватимемо дані, отримані з минулого експерименту. Під час тестування за допомогою інструменту JMeter, було також зібрано дані про кількість запитів, які не були оброблені системою. Побудуємо графік з усереднених значень з проведених п’яти спроб (рис.5.4).

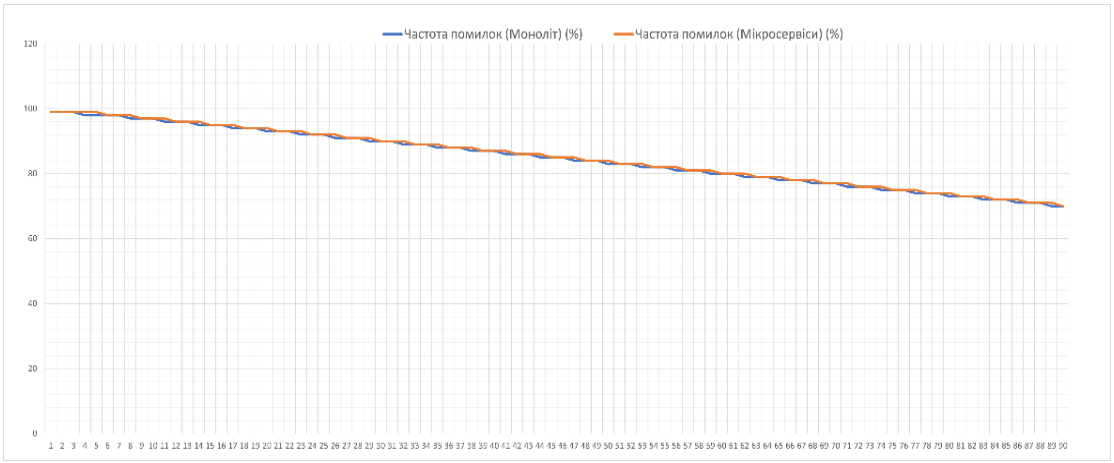

Рисунок 5.4 – Частота помилок

Звідси ми бачимо що частота помилок майже не відрізняється і запити обробляються однаково успішно в обох випадках.

Перейдемо до другого експерименту. В цьому експерименті ми виконаємо симуляцію ряду найбільш поширених сценаріїв, пов’язаних з несправністю архітектурних складових додатків. А саме:

– раптова несправність екземпляру EC2 (з вимкненням екземпляру);

– проблема з мережною складовою (несправність шлюзу, таблиці маршрутизації, комунікації між мікросервісами);

– несправність бази даних.

Також для порівняння наведемо дані без несправностей. Усунення несправностей буде ручним. Оскільки при постановці експерименту природа несправності заздалегідь відома, час витрачений на усунення мінімальний. Кожен з цих сценарії повторений 5 разів; наведені значеня усереднені, деякі значення округлені до найближчого числа кратного 5 або 10.

Почнемо з монолітного застосунку. Результати експерименту занесені в таблицю 5.1

Таблиця 5.1 – Результати експерименту з монолітною архітектурою

Сценарій	Затримка, мс	Коефіцієнт помилок, %	Доступність, %	Час на відновлення, хв	Час простою застосунку, хв
Несправності відсутні	100	1	100	0	0
Проблема з екземпляром	300	50	50	15	30
Несправність мережі	350	60	40	10	20
Несправність бази даних	500	70	10	30	50

Як можемо побачити, час, витрачений на усунення помилки в монолітній архітектурі є надто великим, що пов’язане з тим, що структура застосунку містить в собі одну точку несправності, тобто якщо стається помилка на екземплярі EC2, то немає можливості переадресації на інші. Концентрація всього застосунку в одній точці також сповільнює час необхідний на розгортання. Також відсутність заздалегідь приготованих відновлювальних скриптів збільшує час на відновлення системи в робочий стан.

Перейдемо до мікросервісного застосунку. Результати експерименту занесені в таблицю 5.2.

Таблиця 5.2 – Результати експерименту з мікросервісною архітектурою

Сценарій	Затримка, мс	Коефіцієнт помилок, %	Доступність, %	Час на відновлення, хв	Час простою застосунку, хв
Несправності відсутні	130	1	100	0	0
Проблема з екземпляром	150	10	95	10	15
Несправність мережі	180	15	92	12	20
Несправність бази даних	250	20	85	18	30

Звідси бачимо, що час, який зайняв в нас на відновлення значно менший за монолітну архітектуру. Вища доступність пов’язана з тим, що тут відсутня одна точка помилки, є можливість тимчасово змінити конфігурацію застосунку на час усунення помилок зберігаючи часткову доступність. Також кодова база окремого сервісу потребує менше часу на розгортання, чим зумовлені низький час простою при несправності.

В плані надійності, мікросервісна архітектура значно переважає монолітну архітектуру через ряд факторів, такі як розділення ролей, відсутність однієї точки несправності та малий розмір окремих компонентів.

5.4 Порівняння вартості розгорнутих застосунків

Розрахунок порівняльної вартості проводитиметься за допомогою офіційного калькулятора цін від хмарного провайдера [28]. Згідно з архітектур, наведених на рисунках 3.1 та 4.1, визначимо які ресурси підлягають тарифікації. Це EC2, VPC та RDS. Далі визначимо яке навантаження в кожного з цих ресурсів і яким чином тарифікувати ці ресурси.

Для екземплярів EC2 визначимо наступне:

– Тип екземпляру – t2.micro;

– Стратегія тарифікації – використання за потреби, 10 год/добу;

– Збір метрик – дозволено.

Для екземпляру RDS визначимо наступне:

– Тип екземпляру – db.t3.micro;

– Стратегія тарифікації – використання за потреби, 5 год/добу;

– Тип розгортання – одна зона доступності.

Для VPC тарифікація відбувається за кількістю віртуальних з’єднань, встановимо кількість віртуальних з’єднань рівною одному.

Ціни

Таблиця 5.3 – Порівняння місячної вартості розгорнутих застосунків

Назва ресурсу	Монолітний застосунок	Мікросервісний застосунок
EC2	5,63 дол. США	11,26 дол. США*
RDS	35,99 дол. США	35,99 дол. США
VPC	47,50 дол. США	47,50 дол. США
Сума	89,12 дол. США	94,75 дол. США

* - кількість запущених екземплярів збільшено до 2 відповідно до архітектури

Таким чином, наша мікросервісна архітектура вийшла дорожче за монолітну. Це пов’язано з тим, що застосунок не було добре оптимізовано під розгортання в мікросервісному форматі, через що потребує такого самого розміру екземпляри EC2, як і монолітна конфігурація. Втім, дані розрахунки стосуються початкової конфігурації, без врахування автомасштабування, що в довгостроковій перспективі збільшує вартість монолітного застосунку при постійному прирості користувачів.

ВИСНОВКИ

Одним з важливих етапів розробки застосунків є планування архітектури. Від даного етапу залежить, наскільки просто та економічно вигідно буде підтримувати застосунок, а також надійність всієї системи.

В даній роботі ми розглянули найпопулярніші архітектурні підходи та практичні реалізації кількох з них на прикладі застосунку WordPress.

В першому розділі ми описали найпопулярніші архітектурні підходи та надали порівняльну характеристику. Одним з найновіших принципів побудови застосунків є MACH, який побудований на мікросервісній архітектурі. Втім, для малих застосунків краще за все монолітна або мікросервісна архітектура, оскільки вони мають малу початкову вартість.

В другому розділі ми прийшли до вибору застосунку. Наш вибір пав на WordPress, який має широку спроможність до модифікації та є найпопулярнішим вибором рушія для веб-сайту. Було описано його внутрішню структуру та середовище, в якому відбуватиметься практична реалізація ряду обраних архітектур.

В третьому та четвертому розділах, ми спланували та успішно розгорнули монолітну та мікросервісну архітектури. Під час практичної реалізації виникали деякі проблеми, які, втім, було вирішено і в розділах були описані успішні спроби.

В п’ятому розділі було порівняно розгорнуті архітектури за чотирьма категоріями: затримка обробки запитів, використання ресурсів, надійність та економічна доцільність. В результаті, мікросервісна архітектура показала себе більш надійною в цілому, проте дорожчою та викликала трохи більші затримки, а ефективність використання ресурсів бажала кращого.

Це пов’язано з тим, що ми намагались розгорнути монолітний застосунок у вигляді мікросервісів без ґрунтовної перебудови кодової бази.

В реальності не існує однаково ефективної для усіх випадків архітектури, що даний експеримент й продемонстрував.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Application Architecture [Електронний ресурс]. URL: https://www.codesee.io/learning-center/application-architecture (дата звернення: 18.12.2024).

2. MACH Architecture [Електронний ресурс]. URL: https://alokai.com/blog/mach-architecture (дата звернення: 18.12.2024).

3. Ford N., Richards M. Fundamentals of Software Architecture: A Comprehensive Guide to Patterns, Characteristics, and Best Practices. Sebastopol, CA: O'Reilly Media, 2020. 500p.

4. Richards M. Software Architecture Patterns. Sebastopol, CA: O'Reilly Media, 2015. 55p.

5. Yokwejuste. Dissecting Layered Architecture [Електронний ресурс]. URL: https://dev.to/yokwejuste/dissecting-layered-architecture-2ppb (дата звернення: 18.12.2024).

6. Three-Tier Architecture [Електронний ресурс]. URL: https://www.ibm.com/topics/three-tier-architecture (дата звернення: 18.01.2025).

7. Herberto Graca. Layered Architecture [Електронний ресурс]. URL: https://herbertograca.com/2017/08/03/layered-architecture/ (дата звернення: 18.12.2024).

8. Monolith Architecture [Електронний ресурс]. URL: https://tech.tamara.co/monolith-architecture-5f00270f384e (дата звернення: 18.01.2025).

9. Ford N., Parsons R., Kua P. Building Evolutionary Architectures: Support Constant Change. Sebastopol, CA: O'Reilly Media, 2017.

10. Modular Monolithic Architecture [Електронний ресурс]. URL: https://medium.com/design-microservices-architecture-with-patterns/microservices-killer-modular-monolithic-architecture-ac83814f6862 (дата звернення: 18.12.2024).

11. Herberto Graca. Monolithic Architecture [Електронний ресурс]. URL: https://herbertograca.com/2017/07/31/monolithic-architecture/ (дата звернення: 18.12.2024).

12. Service-Oriented Architecture [Електронний ресурс]. URL: https://aws.amazon.com/what-is/service-oriented-architecture/ (дата звернення: 18.12.2024).

13. Microsoft. Three-Tier Architecture Documentation [Електронний ресурс]. URL: https://learn.microsoft.com/en-us/previous-versions/aa480021(v=msdn.10)?redirectedfrom=MSDN (дата звернення: 18.12.2024).

14. Microservices Overview [Електронний ресурс]. URL: https://www.geeksforgeeks.org/microservices/ (дата звернення: 18.12.2024).

15. Spiceworks. What Are Microservices? [Електронний ресурс]. URL: https://www.spiceworks.com/tech/devops/articles/what-are-microservices/ (дата звернення: 18.12.2024).

16. Middleware.io. Microservices Architecture [Електронний ресурс]. URL: https://middleware.io/blog/microservices-architecture/ (дата звернення: 18.12.2024).

17. Encore. Event-Driven Architecture [Електронний ресурс]. URL: https://encore.dev/resources/event-driven-architecture (дата звернення: 18.12.2024).

18. Solutelabs. MACH Architecture [Електронний ресурс]. URL: https://www.solutelabs.com/blog/mach-architecture (дата звернення: 18.12.2024).

19. Elasticpath. What Is MACH Architecture? [Електронний ресурс]. URL: https://www.elasticpath.com/blog/what-is-mach-architecture (дата звернення: 18.12.2024).

20. Medium. MACH Architecture [Електронний ресурс]. URL: https://medium.com/google-cloud/mach-architecture-what-is-it-b3d34d02c8cd (дата звернення: 18.12.2024).

21. Ninetailed. Everything About MACH Architecture [Електронний ресурс]. URL: https://ninetailed.io/blog/everything-about-mach-architecture/ (дата звернення: 18.12.2024).

22. About WordPress [Електронний ресурс]. URL: https://uk.wordpress.org/about/ (дата звернення: 28.12.2024).

23. Ericson E. Learn WordPress: From Beginner to Advanced User. Independently Published, 2017.

24. Williams B., Damstra D., Stern H. Professional WordPress: Design and Development. Hoboken, NJ: John Wiley & Sons, 2015.

25. Kublr. Hardware Recommendation [Електронний ресурс]. URL: https://docs.kublr.com/installation/hardware-recommendation/ (дата звернення: 29.12.2024).

26. AWS EC2 Instance Types [Електронний ресурс]. URL: https://aws.amazon.com/ec2/instance-types/ (дата звернення: 29.12.2024).

27. WordPress Requirements [Електронний ресурс]. URL: https://wordpress.org/about/requirements/ (дата звернення: 1.01.2025).

28. AWS Calculator [Електронний ресурс]. URL: https://calculator.aws/ (дата звернення: 2.01.2025).