Vol. 6, No. 2, 2024

Назар Плесканка[1], Мар'яна Плесканка[2], Тарас Слободзян[3], Богдан Марко[4]

[1] Кафедра систем автоматизованого проектування, Національний університет "Львівська політехніка", вул. С. Бандери, Львів, Україна, E-mail: nazarii.m.pleskanka@lpnu.ua, ORCID 0009-0002-2341-5113

[2] Кафедра Телекомунікації Національний університет "Львівська політехніка", вул. Професорська, 2, Львів, Україна, E-mail: mariana__p.m.v.9@ukr.net,

[3] Кафедра систем автоматизованого проектування, Національний університет "Львівська політехніка", вул. С. Бандери, Львів, Україна, E-mail: taras.slobodzjan@lpnu.ua

[4] Кафедра систем автоматизованого проектування, Національний університет "Львівська політехніка", вул. С. Бандери, Львів, Україна, E-mail: bohdan.marko.knm.2020@lpnu.ua

АНАЛІЗ ЕФЕКТИВНОСТІ ВИКОРИСТАННЯ МІКРОСЕРВІСІВ ПРИ РОЗРОБЦІ WEB ДОДАТКІВ

Отримано: вересень 03, 2024 / Переглянуто: вересень 18, 2024 / Прийнято: Вересень 30, 2024

https://doi.org/

Анотація. В роботі проведено аналіз та дослідження продуктивності роботи WEB платформи та ефективності використання хмарних технологій і мікросервісної архітектури. У дослідженні розглядаються основні аспекти переходу від монолітної архітектури до використання мікросервісів і хмарних технологій, включно з декомпозицією системи на незалежні сервіси, що сприяє покращенню обробки та зберігання даних та дає можливість досягти більшої ефективності. Додатково, проведено порівняльний аналіз продуктивності роботи системи із використанням обох архітектурних підходів. Представлено графічні залежності, які показують як змінюється час відповіді в залежності від навантаження при використанні різних архітектурних підходів побудови аплікацій.

Ключові слова: мікросервіс, моноліт, інфраструктура, хмарне середовище, масштабування.

Вступ

У сучасному світі, де інформаційні технології стрімко розвиваються, трансформуючи всі сфери життя, вебплатформи та онлайн-сервіси відіграють все більш важливу роль. В таких умовах швидкого розвитку, зокрема хмарних обчислень та мікросервісної архітектури, постає нагальна потреба в модернізації та адаптації веб-платформ для забезпечення максимальної гнучкості, ефективної масштабованості та високої продуктивності. Сучасні рішення повинні враховувати зростаючі вимоги до оптимізації ресурсів, швидкої інтеграції нових компонентів та безперебійного функціонування систем у динамічних умовах. Веб додатки забезпечують безпрецедентний доступ до інформації, дозволяючи людям залишатися на зв'язку з глобальними подіями, не виходячи з дому. За переліченими зручностями та можливостями стоїть наполеглива праця IT-фахівців, які проектують, розробляють, обслуговують та вдосконалюють такі цифрові системи, гарантуючи їхню надійність, доступність та ефективність. Вибір правильної архітектури для вебплатформ є ключовим фактором їх успішного функціонування та масштабування.

Постановка проблеми

Стрімкий розвиток хмарних технологій призвів до того, що хмарні обчислення стають все більш популярним вибором для підприємств будь-якого розміру. Це в свою чергу ставить нові вимоги до архітектурних підходів побудови та розгортання сервісів. Все більше уваги приділяється продуктивності роботи самого сервісу та ефективному використанню ресурсів. Високої популярності набула мікросервісна архітектура побувдови аплікацій, яка дає можливість розділяти

монолітні системи на незалежні, гнучкі та легко масштабовані мікросервіси, що покращує продуктивність, гнучкість та адаптивність до мінливих потреб. Використання таких підходів дає можливість легко боротись із короткотривалими сплесками навантаження та при цьому масштабувати ресурси лише за необхідності.

Саме тому, дуже важливим є правильне планування процесу міграції в хмарні системи із використанням мікросервісів, що включає аналіз існуючої архітектури, розробку плану трансформації та поступову реалізацію змін з метою досягнення максимальної ефективності та мінімізації ризиків. Також окрім планування самої міграції, слід оцінити всі переваги та недоліки і провести порівняльний аналіз продуктивності роботи сервісів при використанні обох підходів, що дозволить зрозуміти, як кожна архітектура впливає на швидкість обробки даних, можливості масштабування та загальну гнучкість у відповідь на зміни в навантаженні.

Огляд сучасних джерел інформації за тематикою публікації

Мікросервісна архітектура, також відома як мікросервіси, являє собою методологію розробки програмного забезпечення, яка полягає у розділенні додатків на менші, автономні сервіси. Кожен із цих сервісів відповідає за певний аспект функціональності та організований навколо конкретних бізнес-задач аплікації [1-2]. Вони взаємодіють між собою через добре визначені API інтерфейси. Ці сервіси зазвичай мають слабкий зв'язок один з одним, що дозволяє легко їх підтримувати та незалежно розгортати, забезпечуючи гнучкість і надійність архітектури додатків. Для обробки запиту одного користувача додаток, побудований на основі мікросервісів, може звертатися до кількох внутрішніх компонентів, що працюють незалежно, для формування повної відповіді. Такий підхід не тільки сприяє масштабованості, але й підвищує продуктивність та швидкість реагування системи на зміни у вимогах бізнесу.

Базова схема архітектури побудови мікросервісів представлена на рисунку 1.

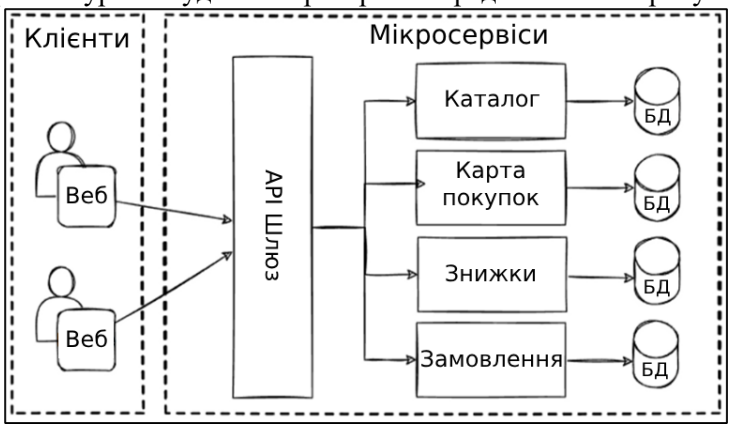

Рис. 1. Базова архітектура мікросервісного додатку

Мікросервіси базуються на ключових принципах, таких як принцип єдиної відповідальності, слабка взаємозалежність – тобто сервіси мінімально залежать один від одного, децентралізоване управління та автономне розгортання. Кожен мікросервіс спеціалізується на виконанні окремого набору функцій, зосереджуючись на вирішенні конкретної проблеми. Це дозволяє створювати гнучку і масштабовану архітектуру, яка легко адаптується до змін. Якщо розвиток сервісу призводить до збільшення його складності, його слід розділити на менші, більш керовані компоненти, що значно спрощує управління та підтримку.

Мікросервіси взаємодіють за допомогою протоколів, таких як HTTP із стилем REST або асинхронний обмін повідомленнями, до прикладу використання патерну Pub/Sub, забезпечуючи повну інформацію для обробки кожного запиту [3-4]. Кожен сервіс має свій власний стек технологій та, здебільшого, базу даних, що забезпечує автономність та масштабованість.

Серед основних переваг використання мікросервісної архітектури можна виділити наступні:

- *масштабованість:* Однією з найбільших переваг мікросервісів є можливість незалежного масштабування окремих компонентів системи. На відміну від монолітних додатків, де

масштабування потребує збільшення ресурсів для всієї програми, мікросервіси дозволяють розширювати лише ті сервіси, які найбільше потребують додаткових ресурсів. Це забезпечує оптимізацію витрат і підвищення продуктивності, зменшуючи навантаження на інші частини системи.

- *гнучкість у виборі технологій:* Мікросервісна архітектура надає командам розробників можливість вибору найбільш підходящих технологій для кожного окремого сервісу. Це дозволяє використовувати різні мови програмування, бази даних, платформи та інструменти в межах одного проекту. Такий підхід забезпечує технологічну незалежність, дозволяючи створювати сервіси, які найкраще відповідають специфічним вимогам бізнесу.

- *стійкість:* Децентралізована структура мікросервісів забезпечує високу відмовостійкість системи. Оскільки кожен сервіс функціонує автономно, збій в одному з них не призведе до зупинки роботи всієї програми. Це дозволяє ізолювати проблеми, швидко реагувати на інциденти, перезапускати або замінювати несправні сервіси без суттєвого впливу на решту системи. Як результат, забезпечується безперервність надання послуг та зниження часу простою.

- *підтримка CI/CD:* Мікросервіси добре інтегруються з підходом безперервної інтеграції та доставки (CI/CD) додатків на різні середовище. Завдяки модульності архітектури, окремі сервіси можуть бути розгорнуті та оновлені незалежно один від одного, що сприяє більш швидкому впровадженню нових функцій та виправленню помилок. Це дозволяє частіше випускати нові версії продукту, скорочуючи час виходу на ринок та забезпечуючи швидкий зворотний зв'язок з користувачами.

- *підвищення продуктивності розробників:* Завдяки мікросервісам, кілька команд можуть одночасно працювати над різними сервісами без потреби чекати завершення роботи інших команд. Такий підхід створює умови для паралельної розробки та підвищує продуктивність команди. Незалежність сервісів сприяє кращій організації процесів розробки та зменшенню залежностей між різними частинами проекту, що в свою чергу покращує співпрацю та пришвидшує загальний процес створення програмного забезпечення.

Загалом, мікросервісна архітектура забезпечує підвищену гнучкість, стійкість і масштабованість систем, що дозволяє ефективно адаптуватися до сучасних вимог бізнесу та технологій.

Цілі та проблеми дослідження

В даному дослідженні були поставлені наступні цілі:

1. Розробити процес міграції монолітної вебінфраструктури в хмарну систему за допомогою використання мікросервісної архітектури з метою досягнення максимальної ефективності та мінімізації ризиків.
2. Провести порівняльний аналіз ефективності роботи веб аплікації із використанням монолітної та мікросервісної архітектури
3. Дослідити особливості масштабування ресурсів та адаптації роботи сервісів при зміні навантаження на систему.

Виклад основного матеріалу

Монолітний додаток зберігається в одному репозиторії та випускається як єдина одиниця, тоді як мікросервіси розділяють додатки на незалежні функціональні блоки, що почалося з розвитку SOA архітектури. Раніше моноліти були стандартним підходом у ПЗ, дозволяючи організаціям мати єдиний контроль над всіма функціями та даними, але це призводило до складнощів при управлінні кількома схожими програмами різними командами. Вибір архітектури залежить від довгострокових цілей проекту, і системний дизайн може потребувати адаптації з часом.

Додаток є монолітним, якщо він містить такі компоненти, як інтерфейс користувача, бізнес-логіку та шар доступу до даних в єдиному блоці, який розробляється, тестується та розгортається разом [5]. Структуру базового монолітного додатку можна побачити на рисунку 2.

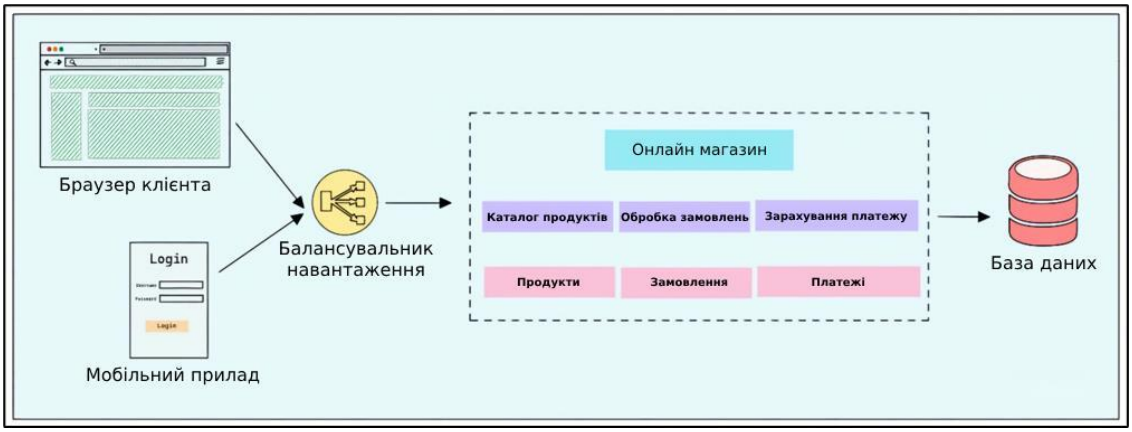

Рис. 2. Базова архітектура монолітного додатку

Однорівневий моноліт має інтерфейс, бізнес-логіку та логіку доступу до даних, написані як один додаток. Зміна будь-якої частини програми вимагає тестування та пере розгортання всієї програми. Такі додатки не потребують чітко визначених модулів, пов'язаних з функціональністю або бізнес-доменами [5-6].

Модульний моноліт схожий на однорівневий, оскільки вся функціональність програми розгортається разом. Однак, модульні додатки мають логічні межі, визначені між модулями, які відповідають бізнес або функціональним вимогам. Завдяки тісному зв'язку коду модулі можуть взаємодіяти один з одним. База даних модульного моноліту, як правило, має таку ж структуру і шаблони доступу додатків, як і в однорівневому [7-8].

Розробка додатків як моноліту має певні переваги, до яких можна віднести:

- *Тестування та налагодження* – розробники можуть тестувати та налагоджувати весь додаток, запускаючи його в IDE. Вони також можуть використовувати інструменти автоматизації тестування для запуску одного інтеграційного тесту або цілого набору для перевірки наскрізної функціональності.

- *Розгортання додатків* – розгортання монолітів є простим, оскільки існує лише один додаток, який розгортається як єдине ціле, що усуває необхідність координувати розгортання декількох додатків і компонентів одночасно.

- *Операційні накладні витрати на управління монолітними до*датками є відносно низькими, оскільки вони мають обмежену кількість зовнішніх залежностей. Комунікація між модулями відбувається всередині самої програми, тому діагностика та усунення проблем зосереджені на одній аплікації.

Незважаючи на перелічені вище переваги, також існують доволі вагомі обмеження монолітної архітектури, а саме [7]:

- *Зниження гнучкості* – додаток стає занадто великим і складним для того, щоб команда могла швидко вносити зміни без ризику виникнення дефектів високого рівня.

- *Зниження продуктивності* – розмір програми може впливати на такі характеристики як час компіляції, запуску та загальну продуктивність аплікації.

- *Ризик розгортання* – команда повинна розгортати весь додаток для кожної зміни. Наслідки невеликих змін можуть стати широкомасштабними.

- *Обмежене використання CI/CD* – великий розмір програми та специфічні вимоги до середовища виконання роблять безперервне розгортання складним. Для великих монолітних додатків не є чимось незвичайним те, що в процесі розгортання потрібно виконувати кроки вручну.

Вибір мікросервісної архітектури чи міграція до неї має бути свідомим і необхідним для досягнення бізнес-результатів.

Підсумовуючи основні результати порівняння, представимо їх у вигляді порівняльної таблиці 1.

Таблиця.1.

Порівняння монолітної та мікросервісної архітектури

Фактор порівняння	Моноліт	Мікросервіси
Структура	Єдина, тісно пов'язана аплікація	Багато маленьких, незалежних сервісів
Комунікація	Прямі виклики методів/функцій всередині програми	Міжсервісний зв'язок через API або події
Масштабованість	Масштабується як цілий додаток	Масштабується шляхом додавання екземплярів окремих сервісів
Гнучкість	Обмежений вибір технологій	Широке технологічне розмаїття в ізольованих сервісах
Обслуговування	Зміни можуть вплинути на всю програму	Зміни в мікросервісах мають ізольовані наслідки
Розгортання	Єдина одиниця розгортання	Незалежне розгортання мікросервісів
Командна співпраця	Вся команда працює на одній кодової базі	Команди працюють над окремими мікросервісами
Ізоляція несправностей	Один збій може вплинути на всю систему	Збої містяться в окремих мікросервісах
Керування складністю	Підвищена складність завдяки тісно з'єднаним компонентам	Менша складність завдяки сильно роз'єднаним компонентам

Одним із важливих кроків, про які слід пам'ятати при переході до мікросервісної архітектури є вибір Cloud провайдера та налаштування інфраструктури використовуючи підходи IaC. На сьогоднішній день існує досить багато провайдерів хмарних послуг, однак в межах даного дослідження основна увага була акцентована на Google Cloud Platform (GCP) [3]. Варто зазначити що за рахунок використання сервісів хмарного провайдера, можна отримати ряд переваг, серед яких можна виділити наступні:

- *Інноваційні технології та сучасна інфраструктура:* Google Cloud Platform (GCP) забезпечує компанії доступом до передових інструментів для роботи з штучним інтелектом (AI) та машинним навчанням (ML). Це відкриває нові можливості для організацій, які прагнуть інтегрувати AI/ML у свої процеси. Зокрема, Google BigQuery, як одна з найпотужніших аналітичних платформ, дозволяє обробляти величезні обсяги даних миттєво, що особливо корисно для великих корпорацій.

- *Глобальна інфраструктура та продуктивність:* GCP має в своєму розпорядженні одну з найбільших приватних оптоволоконних мереж у світі, яка забезпечує стабільну продуктивність та мінімальні затримки для користувачів, незалежно від їхнього місцезнаходження. Це робить GCP ідеальним вибором для компаній із глобальною присутністю, які потребують високої доступності та швидкості доступу до своїх сервісів.

- *Безпека та відповідність міжнародним стандартам:* Google активно інвестує в безпеку своїх хмарних сервісів, впроваджуючи такі заходи, як шифрування даних на всіх етапах та багатофакторну автентифікацію. GCP відповідає багатьом міжнародним стандартам, включаючи GDPR, HIPAA та PCI DSS, що є критично важливим для компаній, які працюють у регульованих галузях, таких як охорона здоров'я та фінанси.

- *Інтеграція з екосистемою Google:* Для компаній, що використовують Google Workspace, інтеграція з GCP може значно спростити управління IT-інфраструктурою. Google Kubernetes Engine (GKE) є одним з найбільш надійних сервісів для управління контейнеризованими додатками, і Google, як першопроходець у розвитку Kubernetes, надає потужні інструменти та експертизу у цій галузі.

- *Підтримка та активна спільнота:* GCP пропонує першокласну підтримку для своїх клієнтів, зокрема й для великих підприємств, які потребують індивідуального підходу. Крім того, широка партнерська екосистема та активна спільнота розробників забезпечують доступ до великої кількості ресурсів, інструментів та експертних знань, що сприяє швидкому вирішенню будь-яких проблем та розвитку нових рішень.

Етапи міграції до мікросервісів.

Міграція логіки із серверної (монолітної архітектури) інфраструктури на хмарну платформу GCP представляє собою доволі тривалий процес, який вимагає ретельного планування та тестування. Якщо говорити про міграцію монолітної бізнес логіки в нове середовище мікросервісів, то цей процес можна описати наступними основними кроками:

- Аналіз існуючої монолітної логіки.
- Виокремлення цілісної логічної частини.
- Розбиття даної частини на менші компоненти.
- Написання нового сервісу у хмарі.
- Тестування та розгортання.
- Поступове збільшення трафіку на новий сервіс.
- Моніторинг та підтримка нового функціоналу.
- Видалення старого коду.

Отже, давайте розглянемо кожен крок більш детально та обговоримо, як саме відбуваються ці етапи.

Аналіз існуючої монолітної логіки є критично важливим кроком у процесі міграції аплікації у хмарне середовище. Основне завдання цього етапу полягає у глибокому розумінні механізмів роботи аплікації та її архітектури, виявленні взаємодій між внутрішніми компонентами та визначенні їх меж відповідальності. Зокрема, у моноліті існує значна кількість тісно пов’язаних функціональних блоків. Розробники та архітектори повинні чітко розуміти, як саме ці компоненти взаємодіють, чи можна їх ефективно сегментувати для подальшої міграції, або ж необхідно спершу здійснити зміни існуючого моноліту. Зазвичай цей процес займає значний час, оскільки аналіз великої і складної системи вимагає витонченої уваги до деталей.

Виокремлення цілісної логічної частини. Наступним важливим етапом міграції є виокремлення специфічних логічних блоків аплікації. Ці блоки виконують окремі або тісно пов'язані бізнес функції. На цьому етапі відбувається вивчення визначених логічних блоків для повного розуміння їхніх залежностей та визначення потенційних ризиків, пов'язаних з міграцією.

Розбиття даної частини на менші компоненти. Після визначення частини моноліту, що підлягає міграції, наступний крок полягає у її подальшому поділі на менші, більш керовані компоненти, що полегшує процес міграції. Цей етап критично важливий для гарантії того, що кожен новий сервіс буде не лише функціональним, але й ефективно інтегрованим з іншими частинами системи, забезпечуючи плавний перехід від старої монолітної архітектури до нової, розподіленої.

Написання нового сервісу у хмарі. Після успішного поділу функцій моноліту на дрібніші компоненти, наступний етап полягає в створенні нового сервісу в хмарі. Розробка нового сервісу зазвичай починається зі створення базової структури або «скелета» аплікації. Це включає формування архітектури, яка часто використовує модель «Clean Architecture» зображену на рисунку 3, де визначаються ключові рівні:

Presentation Layer – управління користувацьким інтерфейсом.

Business Logic Layer – основні бізнес правила обробки даних програми.

Domain Layer – «серце» аплікації, що відображає реальні об'єкти домену.

Infrastructure Layer – роботи з базами даних, зовнішніми сервісами та іншою інфраструктурою.

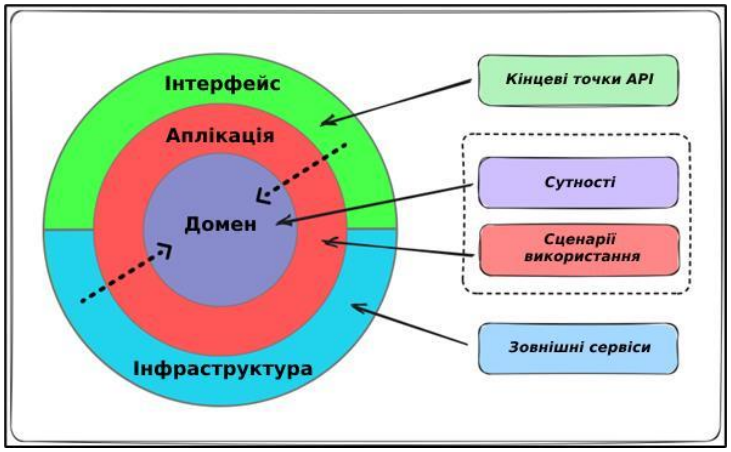

Рис. 3. Приклад структури проектів у мікросервісах

Крім того, створюється окремий проект для тестування функціональності сервісу. До складу сервісу додаються проміжні сервіси, що виконують різноманітні допоміжні функції, такі як ведення журналу подій, аутентифікація, кешування запитів, обробка помилок, що дозволяє зробити основний потік даних більш чистим і організованим.

Тестування та розгортання. Як згадувалось у попередніх етапах, тестування нового функціоналу частково покривається самими розробниками, оскільки одночасно з розробкою нового коду пишуться відповідні unit тести, що перевіряють його. Також, окрім тестів самого сервісу, команда QA паралельно створює власні інтеграційні та E2E тести. Приклад запуску та виконання тестів сервісу обробки продуктів можна побачити на рисунках 4. Таким чином, налагоджений процес тестування мікросервісу робить його надійним та безпечним до використання у виробничому середовищі.

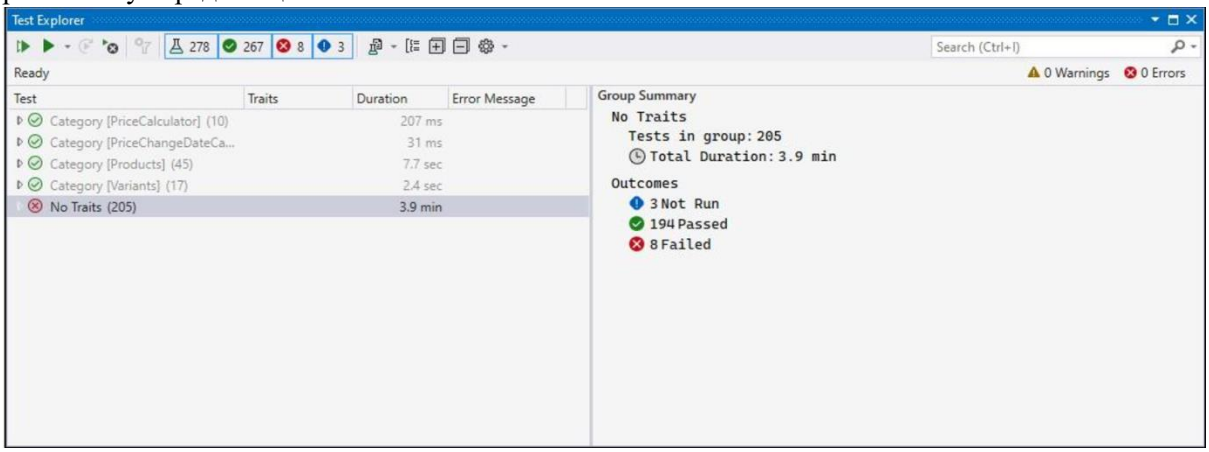

Рис. 4. Приклад запуску тестів мікросервісу

Розгортання нових аплікацій відбувається швидко завдяки інтегрованим CI/CD процесам. У ході цих процесів, скрипти автоматично виконують тестування мікросервісів, запускаючи внутрішні тести. Це значно спрощує роботу розробників, адже у разі виявлення проблем, процес розгортання негайно зупиняється та відправляє сповіщення про помилку через електронний лист. На рисунку 5 можна побачити конвеєр розгортання з усіма відповідними кроками для одного мікросервіса.

Рис. 5. Jenkins конвеєр розгортання мікросервісу

Моніторинг та підтримка нового функціоналу. Після успішного впровадження нового мікросервісу в робоче середовище, команда підтримки здійснює ретельний моніторинг його роботи за допомогою моніторингових систем. Ці системи дозволяють відстежувати різноманітні показники, такі як частота запитів до сервісу, виявлення помилок і їхніх джерел, аналізувати журнал подій, а також налаштовувати спеціальні правила для моніторингу аплікацій. Приклад одного із результатів системи моніторингу зображений на рисунку 6

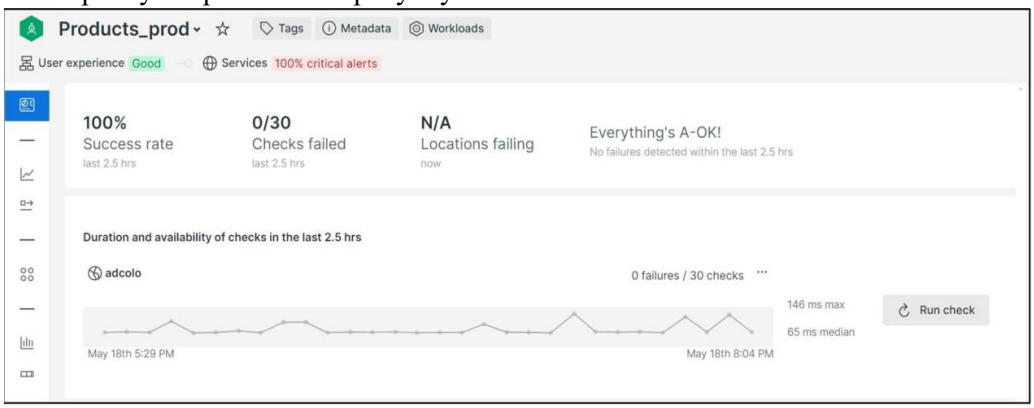

Рис. 6. Приклад моніторингу мікросервісу із використанням системи NewRelic

У випадку виявлення проблем, система автоматично генерує сповіщення, котре повідомляє про неполадки. Також можливе налаштування перевірок часу реакції сервісу або аналіз помилок. Такий підхід забезпечує швидке виявлення та усунення неполадок, збільшуючи надійність та ефективність мікросервісної архітектури.

Видалення старого коду. Під завершення, після того як мікросервіс успішно розгорнутий і функціонує в реальних умовах, настає фінальний етап, під час якого розробники приступають до видалення старих монолітних компонентів. Цей процес має велике значення, адже в моноліті можуть залишитися непередбачені взаємозв'язки в коді, які потенційно можуть спричинити проблеми в системі. Тому до цієї задачі необхідно підходити дуже уважно, щоб не упустити критичні деталі.

Результати та обговорення

Для оцінки ефективності використанні мікросервісної архітектури та необхідності міграції від монолітну було проведено аналіз результатів роботи web аплікації із використанням обох підходів (мікросервісний та монолітний). Основним критерієм порівняння в даному дослідженні було використано час відповіді для кінцевого користувача. Аплікація, на якій проводились

експериментальні дослдіження представляла собою E-Commerce платформу, яка складається із кількох основних компонентів. Базова архітектура представлена на рисунку 7

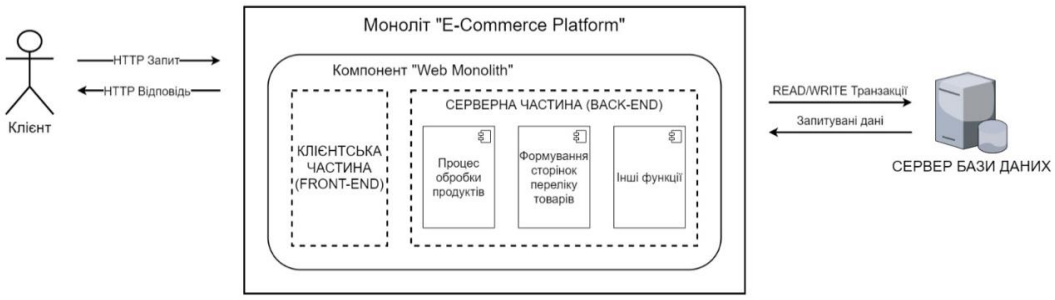

Рис. 7. Базова архітектура E-commerce платформи

Основна компонента, яка була мігрована та на якій проводилось дослідження – це каталог товарів, або ж у новому середовищі – мікросервіс продуктів (ProductMs). Цей компонент, або ж мікросервіс у новий архітектурі, відповідає за представлення інформації про товари, їхній опис, ціни, характеристики на всіх сторінках web аплікації.

Для проведення тестування обох архітектур було використано популярний інструмент під назвою Apache JMeter. Він надає зручний інтерфейс та дозволяє симулювати трафік у вигляді http запитів користувачів до сервісу. З метою отримання достовірних результатів, тестування проводилось у два сценарії.

Сценарій 1: 100 користувачів надсилатимуть по 1 запиту протягом 60 секунд, являє собою режим роботи з низьким навантаженням.

Сценарій 2: 1000 користувачів надсилатимуть по 1 запиту протягом 60 секунд, являє собою режим роботи з високим навантаженням.

Для сценарію 1 було створено тестові плани під назвою «Monolith Product Details Endpoint Test – 100 Users» та «Microservice Listing View Endpoint Test - 100 Users».

Плани було створено з наступними параметрами:

- значення кількості потоків або кількості користувачів - 100, період «розгону» - 60 секунд, а кількість ітерацій - 1, тобто кожен користувач зробить 1 запит.

- тип запиту який буде надсилатись - http запит.

Також сконфігуровано всі необхідні поля, а саме саме ім'я домену – www.E-Commerce Platform.com, протокол – https, метод http запиту – GET, шлях по якому будуть відправлятись ці запити – /als.mvc/nspc/ProductDetails?sku={ідентифікатори продуктів через кому}.

Результати тестування для першого сценарію представлено на рисунку 8.

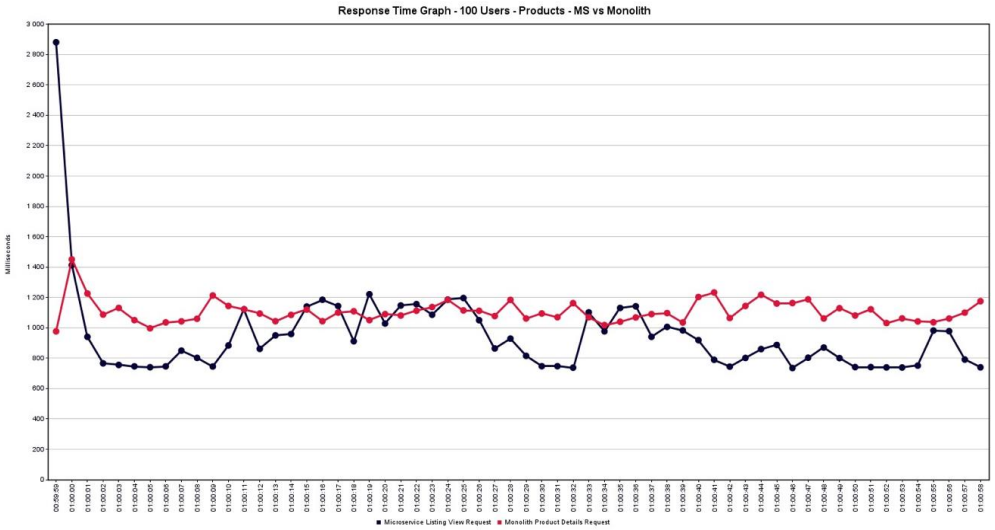

Рис. 8. Час відповіді моноліту та мікросервісу на запити 100 користувачів

Порівняння результатів тестування моноліту та мікросервісу для 100 користувачів:

- Середній час відповіді мікросервісу є меншим – 0.93442 секунди порівняно з монолітом – 1.10824 секунди.

- Мінімальний час відповіді мікросервісу має нижче значення – 0.732 секунди ніж моноліт – 0.995 секунди.

- Максимальне значення часу відповіді мікросервісу є вищим і становить - 2.881 секунди у порівнянні із монолітом – 1.699 секунди.

- Пропускна здатність мікросервісу є кращою – 1.65 транзакцій за секунду ніж у моноліта – 1.16 транзакцій за секунду.

По графіку, що представлений на рис.3.2 бачимо, що на початку експерименту, час відповіді від аплікації яка працює як мікросервіс починає зростати і навіть в певний момент час перевищує час відповіді від монолітної аплікації. Це пов'язано з ти, що йому потрібно певний час для виділення ресурсів та масштабування. Мікросервіс мав «холодний» страт, оскільки Kubernetes створював відповідну кількість його екземплярів. Після виділення ресурсів час відповіді відносно стабілізувався, і став меншим за час відповіді від монолітної аплікації. Результат монолітної аплікації, продемонстрував стабільніші показники з самого початку. Це відбувається тому, що монолітна архітектура не залежить від масштабування в реальному часі та використовує всі свої ресурси повноцінно та без змін.

Наступний експеримент буде проведено із використанням режиму високого навантаження, сценарій 2. Було створено окремі тестові плани під назвою «Monolith Product Details Endpoint Test – 1000 Users» для монолітної аплікації, та «Microservice Listing View Endpoint Test - 1000 Users» для мікросервісу.

Основні результати тестування монолітної аплікації:

- Середній час відповіді = 1905.25 мілісекунд = 1.90525 секунд.
- Мінімальний час відповіді = 1064 мілісекунд = 1.064 секунд.
- Максимальний час відповіді = 5215 мілісекунд = 5.215 секунд.
- Пропускна здатність = 16.37 транзакцій за секунду.

Основні результати тестування мікросервісу:

- Середній час відповіді = 869.26 мілісекунд = 0.86926 секунд.
- Мінімальний час відповіді = 728 мілісекунд = 0.728 секунд.
- Максимальний час відповіді = 3051 мілісекунд = 3.051 секунд.
- Пропускна здатність = 16.44 транзакцій за секунду.

Графічне представлення результатів відображено на рисунку 9

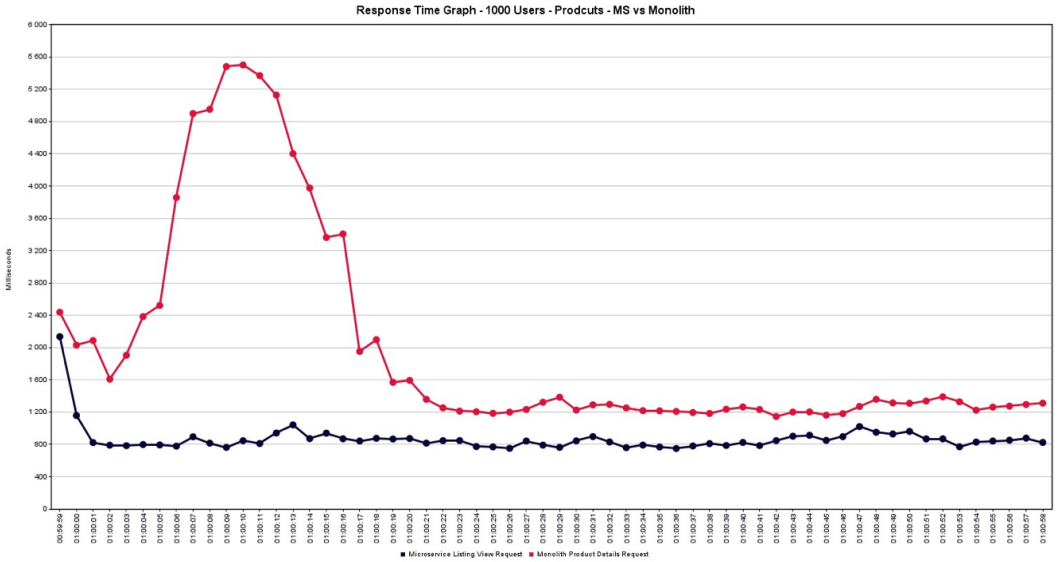

Рис. 9. Час відповіді моноліту та мікросервісу на запити 1000 користувачів

Проаналізувавши результати тестування, можна зробити висновки, що використання мікросервісної архітектури дає можливість отримати кращі значення часу відповіді на запити користувачів. На рисунку 9 можна побачити що час відповіді мікросервісної аплікації мав значний сплеск на початку моделювання, але після кількох секунд роботи результат коливався в межах 800 - 1000 мс. Можна стверджувати, що при більших кількостях користувачів та відповідно більшому навантаженні, мікросервісна аплікація швидко адаптується та масштабує свої ресурси. Це дозволяє мікросервісу якісно пристосовуватись до навантаження та використовувати лише ту частину ресурсів, яка необхідна для задовільної роботи сервісу, без втрати якості обслуговування кінцевих користувачів. На противагу, моноліт показував стабільніші результати для менших навантаженнях, однак при збільшенні трафіку він віддає позиції хмарному сервісу через недостатню оптимізацію коду та інфраструктури.

Висновки

В даній роботі проведено порівняльний аналіз ефективності роботи веб аплікації із використанням монолітної та мікросервісної архітектури. Представлено процес міграції веб аплікації від монолітної до мікросервісної архітектури. Перехід на мікросервіси дозволив адаптуватися до змін навантаження та забезпечити високу продуктивність роботи системи вцілому. Хмарна інфраструктура, яка використовувалась для запуску мікросервісів, забезпечила динамічне масштабування ресурсів, що дозволило ефективно справлятися з великими обсягами трафіку та даних. Результати проведених досліджень показують, що мікросервісна веб аплікація при більшій кількості користувачів та відповідно більшому навантаженні, швидко адаптується та масштабує свої ресурси і тим самим забезпечує надійність та стабільну роботу сервісу.

Результати дослідження допоможуть організаціям ефективно управляти процесом міграції, забезпечуючи краще розуміння викликів і можливостей, які надає перехід до мікросервісів та хмарних технологій. Впровадження сучасних технологічних стратегій дозволить компаніям не лише покращити поточну роботу, але й забезпечити стійкість для майбутнього розвитку та інновацій.

Перелік використаних джерел

[1] Microsoft. «What is cloud computing?». [Електронний ресурс]. Режим доступу: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-iscloud-computing (дата звернення 26.06.2024)

[2] Kev Zettler. «What is cloud computing? An overview of the cloud». [Електронний ресурс]. Режим доступу: https://www.atlassian.com/microservices/cloud-computing (дата звернення 29.05.2024)

[3] Google Cloud Platform. «What is Cloud Computing?». [Електронний ресурс]. Режим доступу: https://cloud.google.com/learn/what-is-cloud-computing (дата звернення 26.06.2024)

[4] Atlassian. «Microservices: understanding what it is and its benefits». [Електронний ресурс]. Режим доступу: https://www.atlassian.com/microservices (дата звернення 25.07.2024)

[5] Aditi Sharma, Craig Bossie, Runeet Vashisht, Tom Moore. «Monolithic to Microservice journey for .NET Applications». 21.10.2022. [Електронний ресурс].

Режим доступу: URL: https://d1.awsstatic.com/developer/Monolith-to-Microservice-Journey-net-framework-application-v1.5.pdf (дата звернення 08.07.2024)

[6] David Vellante. «Breaking Analysis: Uber's real-time architecture represents the future of data apps…meet the architects who built it». 17.06.2023. [Електронний ресурс]. Режим доступу: https://thecuberesearch.com/breaking-analysis-ubers-real-timearchitecture-represents-the-future-of-data-appsmeet-the-architects-who-built-it/ (дата звернення 29.08.2024)

[7] Mario Izquierdo. «Breaking the Monolith at Twitch: Part One». [Електронний ресурс]. Режим доступу: https://blog.twitch.tv/en/2022/03/30/breaking-themonolith-at-twitch/ (дата звернення 05.08.2024)

[8] Mario Izquierdo. «Breaking the Monolith at Twitch: Part Two». 12.04.2022. [Електронний ресурс]. Режим доступу: https://blog.twitch.tv/en/2022/04/12/breaking-the-monolith-at-twitch-part-2/ (дата звернення 15.08.2024)

Nazar Pleskanka[1], Maryana Pleskanka[2], Taras Slobodzian[3], Bogdan Marko[4]

[1] Computer Design Systems Department, Lviv Polytechnic National University, S. Bandery st. 12, Lviv, Ukraine, E-mail: nazarii.m.pleskanka@lpnu.ua, ORCID 0009-0002-2341-5113
[2] Telecommunication Department , Lviv Polytechnic National University S. Bandery st. 12, Lviv, Ukraine, E-mail: mariana__p.m.v.9@ukr.net,
[3] Computer Design Systems Department , Lviv Polytechnic National University S. Bandery st. 12, Lviv, Ukraine, E-mail: taras.slobodzjan@lpnu.ua
[4] Computer Design Systems Department , Lviv Polytechnic National University S. Bandery st. 12, Lviv, Ukraine, E-mail: bohdan.marko.knm.2020@lpnu.ua

ANALYSIS OF THE MICROSERVICES EFFICIENCY USING IN THE WEB APPLICATIONS DEVELOPMENT

Received: September 03, 2024 / Revised: September 18, 2024 / Accepted: September 30, 2024

Abstract. The paper presents an analysis and research on the performance of a WEB platform and the efficiency of using cloud technologies and microservice architecture. The study examines the key aspects of transitioning from a monolithic architecture to the microservices and cloud technologies, including the decomposition of the system into independent services, which improves data processing and storage and allows for greater efficiency. Additionally, a comparative performance analysis of the system using both architectural approaches is conducted. Graphical dependencies are presented, showing how response time changes depending on the load when using different architectural approaches to application development.

Keywords: microservice, monolith, infrastructure, cloud environment, scaling.