DOI: https://doi.org/10.36910/6775-2524-0560-2024-55-03
УДК 004:02
Анісімов Віктор Геннадійович, магістр
https://orcid.org/0009-0006-0146-081X
Кунанець Наталія Едуардівна, д.н.с.к., професор
https://orcid.org/0000-0003-3007-2462
Національний університет «Львівська політехніка», м. Львів, Україна

ПЕРЕХІД ВІД МОНОЛІТНОЇ ДО МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ: МЕТОДОЛОГІЯ ТА ДОСВІД ВПРОВАДЖЕННЯ

Анісімов В. Г., Кунанець Н.Е. Перехід від монолітної до мікросервісної архітектури: методологія та досвід впровадження. У статті розглядається проблема переходу від монолітної архітектури до мікросервісної з розподіленою базою даних на прикладі інформаційної системи для розвиваючого контролю використання мобільних пристроїв. Монолітна архітектура часто стає неефективною при активному розвитку системи, створюючи проблеми з масштабованістю, гнучкістю та управлінням даними. У роботі запропоновано метод поступової декомпозиції монолітної системи на незалежні мікросервіси з власними базами даних. Проаналізовано етапи аналізу початкової структури, визначення потенційних проблем та обмежень монолітної архітектури, а також кроки переходу до розподіленої архітектури. Особливу увагу приділено питанням проектування розподіленої бази даних, зокрема, розподілу даних між регіонами з урахуванням вимог до захисту персональних даних, забезпечення узгодженості та цілісності даних у розподіленому середовищі. Отриманий досвід та рішення можуть бути корисними для розробників та архітекторів при модернізації інформаційних систем зі схожими викликами.
Ключові слова: монолітна архітектура, мікросервісна архітектура, розподілена база даних, декомпозиція, цілісність даних, системний аналіз, інформаційна система.

Anisimov V., Kunanets N. Transition from Monolithic to Microservice Architecture: Methodology and Implementation Experience. The article addresses the problem of transitioning from a monolithic architecture to a microservice architecture with a distributed database, using the example of an information system for the developmental controlling of mobile devices. Monolithic architecture often becomes inefficient with the active development of the system, creating issues with scalability, flexibility, and data management. The study proposes a method of gradual decomposition of the monolithic system into independent microservices with their own databases. It describes the stages of analyzing the initial structure, identifying potential problems and limitations of the monolithic architecture, and the steps for transitioning to a distributed architecture. Special attention is paid to the design of the distributed database, particularly the distribution of data across regions considering personal data protection requirements, and ensuring data consistency and integrity in a distributed environment. The experience and solutions obtained may be useful for developers and architects in modernizing information systems facing similar challenges.
Keywords: monolithic architecture, microservice architecture, distributed database, decomposition, data integrity, system analysis, information system.

Постановка наукової проблеми. У сучасному світі інформаційних технологій, де системи постійно розвиваються та ускладнюються, питання ефективної архітектури та управління даними стає все більш актуальним. Переважна більшість інформаційних систем починають своє життя як монолітна архітектура з єдиною централізованою базою даних. Такий підхід часто є простим та зручним на початкових етапах розроблення, коли функціональність інформаційної системи обмежена, а обсяг даних не надто великий. Однак, з плином часу, коли інформаційна система активно розвивається і до неї постійно додається новий функціонал, монолітна архітектура створює певні проблеми, такі як складність розгортання та оновлення окремих компонентів, обмеження у горизонтальному масштабуванні, труднощі в управлінні великою монолітною базою даних та інші.

Постає наукова проблема дослідження методів та підходів до переходу від монолітної архітектури до більш гнучкої та масштабованої архітектури, зокрема, мікросервісної архітектури з розподіленою базою даних. Необхідно розробити ефективні стратегії декомпозиції монолітних систем на незалежні сервіси, механізми забезпечення узгодженості та цілісності даних у розподіленому середовищі, а також методи проектування та управління розподіленими базами даних для забезпечення високої продуктивності, доступності та відмовостійкості системи.

Аналіз досліджень та публікацій. Проблеми масштабованості, гнучкості та складності розгортання і підтримки монолітних архітектур інформаційних систем широко висвітлені в науковій літературі. Багато дослідників та практиків в галузі інформаційних технологій вивчають переваги та виклики переходу до мікросервісних архітектур та розподілених баз даних.

У дослідженні [1] проведено ґрунтовний аналіз концепцій, характеристик та переваг мікросервісних архітектур. Автори розглянули різні аспекти проектування, розроблення та розгортання мікросервісів, а також виділили ключові виклики, такі як забезпечення узгодженості

даних та міжсервісної комунікації. Це дослідження є важливим для розуміння основних принципів та особливостей мікросервісної архітектури, а також для виявлення потенційних проблем та шляхів їх вирішення. У праці [2], автори детально описали принципи та підходи до декомпозиції монолітних систем на мікросервіси. Вони запропонували методи виділення функціональних областей, визначення меж сервісів та організації взаємодії між ними. Ці методи є ключовими для успішного переходу від монолітної до мікросервісної архітектури, оскільки дозволяють систематично розділити систему на незалежні компоненти та забезпечити їх ефективну взаємодію.

Дослідження [3] зосереджено на шаблонах та практиках проектування мікросервісних архітектур, зокрема питаннях розподілу даних між сервісами, забезпечення узгодженості та цілісності даних у розподіленому середовищі. Автор розглядає різні підходи до організації даних у мікросервісах та надає рекомендації щодо вибору відповідних стратегій в залежності від вимог системи. Ці знання є важливими для проектування ефективної та надійної мікросервісної архітектури з розподіленою базою даних.

Щодо розподілених баз даних, у дослідженні [4] розглянуто різні стратегії розподілу даних, такі як реплікація, шардування та створення федеративних баз даних. Автори проаналізували переваги та недоліки кожного підходу з точки зору продуктивності, масштабованості та доступності даних. Це дослідження надає цінну інформацію для вибору оптимальної стратегії розподілу даних у мікросервісній архітектурі в залежності від специфічних вимог та обмежень системи.

У роботі [5], автор сформулював теорему CAP, яка визначає фундаментальні обмеження розподілених систем щодо узгодженості, доступності та стійкості до розділення. Ця теорема є важливим фактором при проектуванні розподілених баз даних та виборі моделі узгодженості даних. Вона допомагає зрозуміти компроміси, які необхідно враховувати при розробленні розподілених систем, та прийняти обґрунтовані рішення щодо балансу між узгодженістю, доступністю та стійкістю до розділення.

Незважаючи на значну кількість досліджень у галузі побудови мікросервісних архітектур, процес переходу від монолітної до мікросервісної системи залишається унікальним викликом для кожної окремої інформаційної системи. Це пов'язано зі специфічними вимогами, обмеженнями та бізнес-логікою, притаманними кожній інформаційній системі [3, 6].

Згідно з дослідженнями [1, 6, 7], перехід до розподіленої архітектури та бази даних може бути необхідним у наступних випадках:

1. Масштабованість. Коли обсяг даних або кількість користувачів зростає настільки, що одного вузла бази даних стає недостатньо для опрацювання навантаження.

2. Висока доступність. Розподілена база даних з репліками даних забезпечує вищу доступність системи, оскільки у разі збою одного з вузлів, інші вузли можуть продовжувати обслуговувати запити.

3. Географічна розподіленість. Якщо користувачі системи розподілені географічно, розподілена база даних може забезпечити кращу продуктивність за рахунок зменшення затримок при обробці запитів.

4. Відмовостійкість. Розподілена база даних з репліками даних є більш відмовостійкою, оскільки у разі збою одного з вузлів, дані залишаються доступними на інших вузлах.

Дана робота спрямована на те, щоб поділитися досвідом здійснення переходу від монолітної до мікросервісної архітектури на прикладі інформаційної системи розвиваючого контролю використання мобільних пристроїв. Метою є запропонувати практичний метод такого переходу, який враховує специфічні вимоги та обмеження подібних систем. Цей метод може слугувати орієнтиром та основою для інших розробників та архітекторів, які стикаються з подібними викликами при модернізації своїх інформаційних систем.

Мета роботи. Запропонувати методологію трансформації архітектури інформаційної системи, що активно використовується, від монолітної до мікросервісної з розподіленою базою даних, дослідити процес трансформації, виділити ключові етапи, виклики та запропонувати ефективні рішення для успішної модернізації системи без переривання її роботи.

Виклад основного матеріалу й обґрунтування отриманих результатів дослідження. В сучасному світі інформаційних технологій, де системи постійно розвиваються та ускладнюються, питання ефективної архітектури та управління даними стає все більш актуальним. Багато систем починають своє життя як монолітна архітектура з єдиною централізованою базою даних. Такий підхід часто є простим та зручним на початкових етапах розроблення, коли функціональність системи обмежена, а обсяг даних не надто великий.

Однак, з плином часу, коли система активно розвивається і до неї постійно додається новий функціонал, монолітна архітектура починає створювати певні проблеми. База даних стрімко розширюється, що може призвести до зниження продуктивності та ускладнення масштабування системи. Крім того, монолітна архітектура часто обмежує можливості незалежного розгортання та оновлення окремих компонентів системи, що уповільнює процес розроблення та ускладнює впровадження нових функцій [6].

У певний момент стає зрозуміло, що для забезпечення подальшого розвитку та ефективного функціонування інформаційної системи необхідно змінювати її архітектуру. Одним із популярних підходів є перехід до мікросервісної архітектури з розподіленою базою даних. Такий перехід дозволяє розбити монолітну систему на менші, незалежні сервіси, кожен з яких відповідає за певну функціональність та має власну базу даних. Це забезпечує кращу масштабованість, гнучкість та стійкість системи до збоїв.

Для розгляду процесу переходу від монолітної архітектури та монолітної бази даних до мікросервісної архітектури з розподіленою базою даних, ми використаємо приклад інформаційної системи для розвиваючого контролю використання мобільних пристроїв. Основна функція даної інформаційної системи полягає у можливості тимчасово блокувати доступ до певних мобільних додатків, поки користувач не вирішить поставлене перед ним завдання або головоломку. Ця функціональність дозволяє користувачам розвивати свої когнітивні здібності, логічне мислення та вирішувати цікаві завдання, перш ніж отримати доступ до бажаних додатків.

Однією з ключових особливостей системи є можливість для користувачів самостійно створювати та додавати нові завдання до бази даних. Це дозволяє інформаційній системі постійно розвиватися та розширюватися за рахунок внеску спільноти користувачів. Користувачі можуть проявляти свою креативність, розробляючи унікальні та захоплюючі завдання, якими потім зможуть користуватися інші учасники системи.

Завдяки цій функціональності, база даних інформаційної системи постійно збільшується та збагачується новим контентом. Це приваблює нових користувачів, які зацікавлені в різноманітних та складних завданнях, а також стимулює існуючих користувачів до активної участі в розвитку системи.

На рис.1 наведена спрощена схема бази даних для кращого розуміння функціоналу даної інформаційної системи.

Наше дослідження складалося з кількох етапів.

Аналіз інформаційної системи. Першим кроком було проведення аналізу початкової монолітної структури системи. Ми детально вивчили існуючу архітектуру, визначили основні компоненти системи та їх взаємозв'язки. Це дало нам чітке розуміння поточного стану системи та дозволило виявити проблеми та обмеження, які існують на даний момент.

Обов'язковою частиною даного етапу стало визначення основних потенційних проблем та обмежень, які можуть виникнути при активному розвитку даної інформаційної системи при збереженні монолітної архітектури. Ми проаналізували, як зростання функціональності та обсягу даних впливає на продуктивність, масштабованість та гнучкість системи.

Були виявлені такі потенційні проблеми, притаманні більшості монолітних інформаційних систем:

1. Складність розгортання та оновлення окремих компонентів. Монолітна архітектура ускладнює процес оновлення окремих частин системи. Будь-які зміни в одному компоненті вимагають перезбирання та розгортання всієї системи, що збільшує час та ресурси, необхідні для внесення змін, а це в свою чергу збільшує час, протягом якого сервіс є недоступним для користувачів.

2. Обмеження у горизонтальному масштабуванні та висока вартість масштабування. Монолітна архітектура має значні обмеження щодо горизонтального масштабування, тобто додавання нових екземплярів системи для опрацювання зростаючого навантаження. Кожен новий інстанс монолітної системи повинен мати достатньо ресурсів для забезпечення ефективного функціонування всіх компонентів інформаційної системи, що робить горизонтальне масштабування досить витратним. Вартість одного інстанса монолітної інформаційної системи є високою, оскільки він має бути достатньо потужним для опрацювання всіх запитів та забезпечення прийнятної продуктивності. Це призводить до неефективного використання ресурсів та значних фінансових витрат при необхідності збільшення масштабів інформаційної системи.

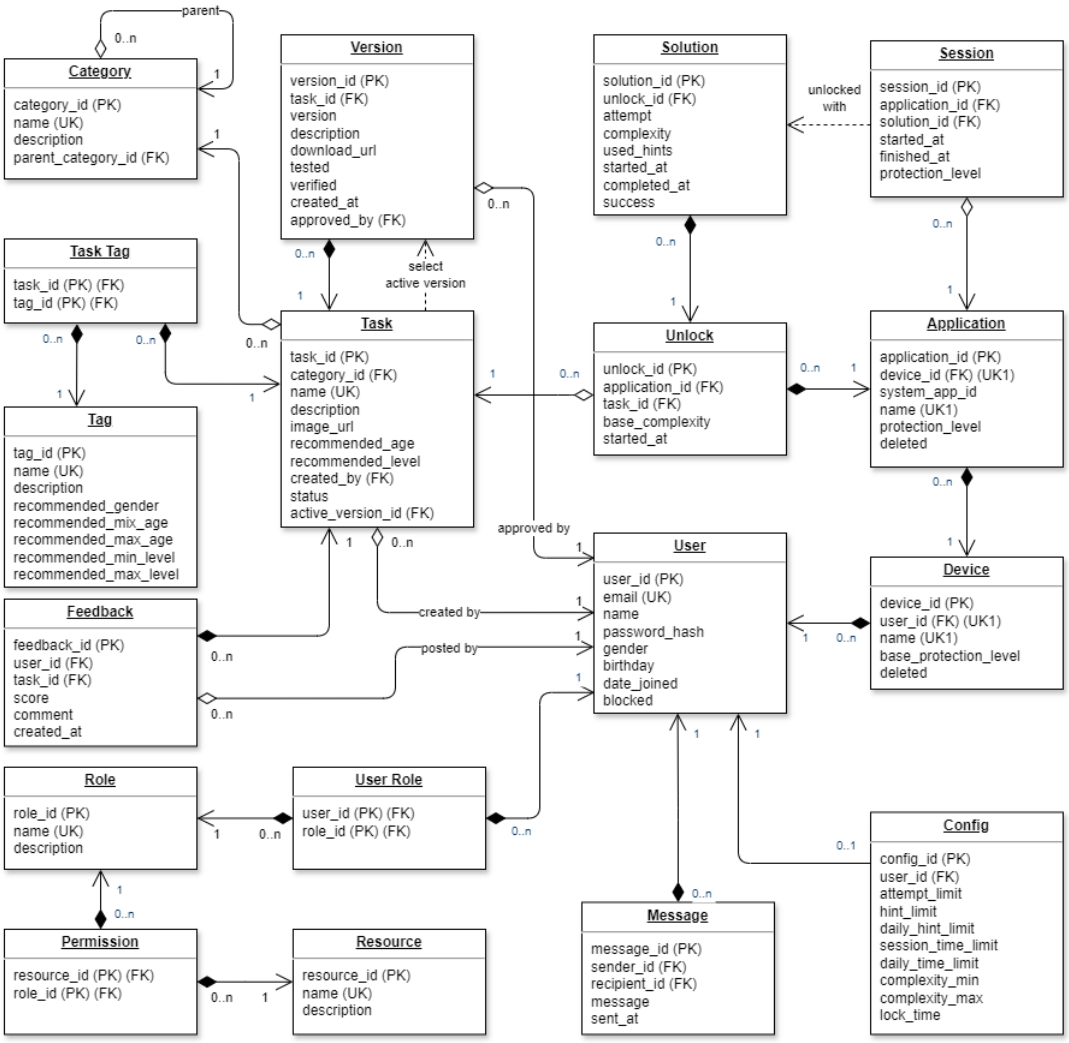

Рис. 1. UML діаграма структури бази даних

3. Труднощі в управлінні великою монолітною базою даних. З ростом системи та збільшенням обсягу даних, управління монолітною базою даних стає все більш складним. Виникають проблеми з продуктивністю запитів, оптимізацією індексів, резервним копіюванням та відновленням даних.

4. Обмеження гнучкості та адаптивності інформаційної системи до змін бізнес-вимог. Внесення змін до однієї частини інформаційної системи може вимагати змін в інших частинах, що ускладнює процес розроблення та збільшує ризики появи помилок, що в свою чергу тягне зростання витрат на тестування інформаційної системи при внесені навіть незначних змін.

5. Складність в організації командної роботи. Розроблення монолітної інформаційної системи, розподіл процесів роботи між командами розробників може бути ускладненим. Залежності між компонентами інформаційної системи можуть призводити до конфліктів та необхідності частої синхронізації роботи різних команд.

6. Обмеження в технологічному стеку. Монолітна архітектура обмежує вибір технологій та фреймворків, які можуть бути використані для розроблення інформаційної системи. Технології вибрані на початковому етапі дуже важко змінити, оскільки це вимагатиме одночасної переробки всього наявного коду. Це призводить до продовження використання застарілих або неоптимальних технологій для певних частин системи.

Визначення цілей. Зважаючи на всі перелічені проблеми, рішення про зміну архітектури інформаційної системи назрівало давно. Важливим чинником необхідності зміни архітектури стала потреба у побудові географічно розподіленої бази даних. Специфіка інформаційної системи, яка працює з користувачами з різних країн та регіонів, згенерувала необхідність забезпечити ефективне опрацювання та зберігання даних у різних географічних локаціях.

Однією з ключових вимог стало забезпечення відповідності регіональним нормативним актам щодо зберігання персональної інформації користувачів. Різні країни мають свої специфічні вимоги до збору, опрацювання та зберігання персональних даних. Наприклад, «Загальний регламент про захист даних (GDPR)» в Європейському Союзі вимагає, щоб персональні дані європейських користувачів зберігалися та опрацьовувалися в межах ЄС. Аналогічні вимоги існують і в інших регіонах світу.

Для забезпечення відповідності цим нормативним актам, необхідно було розробити архітектуру, яка дозволяла б зберігати персональні дані користувачів у тих географічних регіонах, де вони знаходяться. Це означає, що дані користувачів з Європи повинні зберігатися на серверах в Європі, дані користувачів зі США - на серверах у США і так далі. Така архітектура вимагає розподілу бази даних на окремі регіональні сегменти та забезпечення синхронізації даних між ними.

Крім того, географічний розподіл бази даних сприяє покращенню продуктивність та швидкості доступу до даних для користувачів з різних регіонів. Розміщення даних ближче до користувачів зменшить затримки в мережі та забезпечить більш швидке завантаження та оновлення інформації в додатку.

Якщо зробити звичайну повну копію середовища в різних локаціях, ми зіткнемося з проблемою роздільного використання бази даних завдань у цих локаціях. Кожна копія середовища матиме свою окрему базу даних завдань, які будуть розвиватися незалежно одна від одної. Це означає, що нові завдання, створені в одній локації, не будуть доступні в інших локаціях. Синхронізація та перенесення завдань між регіонами стане основним викликом для команди розробників.

Підтримка цілісності та консистентності бази даних завдань у такій розподіленій конфігурації буде надзвичайно складною операцією. Будь-які зміни, внесені в одній локації, доведеться вручну синхронізувати з іншими локаціями, що може призвести до конфліктів версій та втрати даних. Такий підхід також ускладнить процес розроблення та тестування нових завдань. Розробникам доведеться враховувати відмінності між базами даних завдань у різних локаціях, що може призвести до помилок та неузгодженостей. Тестування нових завдань доведеться проводити окремо для кожної локації, що збільшить час та ресурси, необхідні для випуску оновлень.

Тепер, після визначення ключових цілей та драйверів змін, ми можемо перейшли до опису кроків, здійснених для успішного переходу від моноліту з єдиною базою даних до мікросервісної архітектури з вертикально розподіленою базою даних.

Визначення меж мікросервісів. Для декомпозиції інформаційної системи на менші, незалежні мікросервіси, ми проаналізували її функціональність та структуру бази даних (рис.1). Для визначення меж мікросервісів доцільно використовувати методологію, засновану на понятті "обмежених контекстів" (bounded contexts) з предметно-орієнтованого проектування (domain-driven design, DDD) [2].

Обмежений контекст - це явна межа, в рамках якої існує модель домену. Всередині обмеженого контексту всі терміни та поняття мають специфічне значення, і модель відображає узгоджене уявлення про предметну область в рамках цього контексту.

Кожен обмежений контекст зазвичай відповідає певній частині бізнесу або підрозділу організації, що займається конкретним аспектом інформаційної системи. Обмежені контексти часто мають власні моделі даних, які можуть відрізнятися від моделей в інших контекстах.

Кожен мікросервіс повинен реалізовувати функціональність в рамках одного обмеженого контексту і не виходити за його межі. Це забезпечує високу зв'язність всередині сервісу та низьку зв'язність між сервісами.

На основі проведеного аналізу, ми виділили наступні "обмежені контексти", які можуть існувати незалежно один від одного як мікросервіси:

1. User Service - відповідає за управління користувачами інформаційної системи, їх аутентифікацію, авторизацію та збереження персональних даних. Цей сервіс відповідає за взаємодію з таблицями user, role, user_role.
2. Device Service - відповідає за управління мобільними пристроями користувачів, їх реєстрацію в системі та збереження інформації про встановлені додатки. Взаємодіє з таблицями device та application.
3. Task Service - відповідає за управління завданнями та головоломками в системі, їх створення, редагування, категоризацію та зберігання. Цей сервіс працює з таблицями task, category, task_tag, tag, version.

4. Solution Service - відповідає за процес розв'язання завдань користувачами, перевірку правильності відповідей та розблокування додатків. Взаємодіє з таблицями solution, session, unlock.

5. Feedback Service - відповідає за збір та збереження відгуків користувачів про завдання, їх оцінки та коментарі. Працює з таблицею feedback.

6. Recommendation Service - відповідає за підбір оптимальних завдань для користувачів на основі їх активності, рівня знань та налаштувань складності. Аналізує дані з різних сервісів та формує персоналізовані рекомендації.

7. Analytics Service - відповідає за збір та аналіз статистичних даних про активність користувачів, популярність завдань, ефективність розблокування додатків тощо. Агрегує дані з різних сервісів та надає аналітичну інформацію для прийняття рішень.

8. Notification Service - відповідає за надсилання сповіщень та повідомлень користувачам про нові завдання, досягнення, оновлення системи тощо.

Ітераційний процес декомпозиції. Декомпозиція функціональності монолітної інформаційної системи на мікросервіси - це поступовий та ітеративний процес (рис. 2). Не рекомендується намагатися переробити одразу весь моноліт на мікросервіси, оскільки це може призвести до хаосу та труднощів в управлінні процесом.

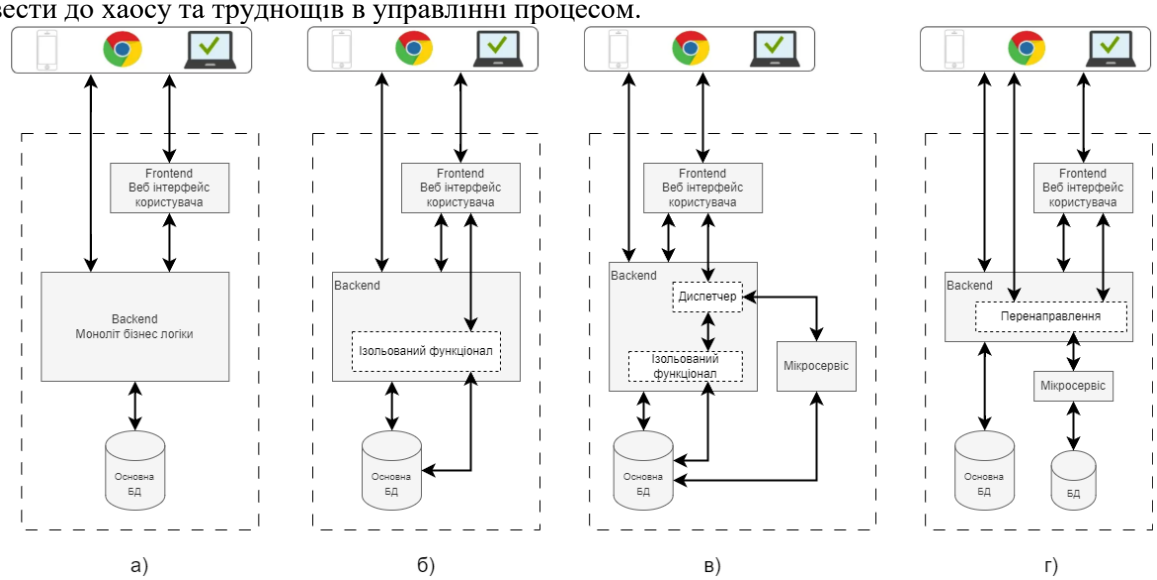

Рис. 2. Ітераційний процес декомпозиції частини функціоналу на мікросервіси:
а) початковий стан; б) ізоляція функціоналу; в) створення мікросервісу;
г) декомпозиція частини бази даних.

При виборі функціональності для здійснення декомпозиції, ми починаємо з найбільш незалежної та ізольованої частини інформаційної системи. У нашому випадку, ми вибрали Feedback Service як кандидата для першого мікросервісу.

Ізоляція функціоналу. Якщо код монолітної системи з самого початку був правильно спроектований з використанням принципів об'єктно-орієнтованого програмування (ООП), то вибрана частина функціональності буде вже достатньо ізольованою. Однак, якщо це не так, необхідно провести рефакторинг коду, щоб максимально ізолювати цю функціональність як незалежний модуль або аплікацію в межах моноліту (рис. 2б). Цей модуль повинен взаємодіяти з іншими частинами інформаційної системи тільки через чітко визначений інтерфейс. Саме цей інтерфейс згодом буде визначати інтерфейс нашого мікросервісу.

У нашому випадку, ми провели рефакторинг коду, пов'язаного з Feedback Service, та виділили його в окремий пакет Python разом з усіма необхідними тестами. Загальні утиліти та допоміжні функції, які використовувалися по всій системі, також були винесені в окремий пакет. Це дозволило нам пізніше легко перенести цей код в окремий мікросервіс.

Після завершення цього епапу, ми ретельно протестували монолітну систему, щоб переконатися, що вона функціонує належним чином і всі зміни не призвели до появи нових помилок або збоїв.

Створення мікросервісу. Наступним кроком стала побудова мікросервісу для вибраної функціональності. Оскільки необхідна функціональність була ізольована, ми легко перенесли код з

монолітної системи в окремий сервіс, зберігаючи при цьому зв'язок з базою даних моноліту. На цьому етапі важливо уникати внесення змін в існуючий функціонал моноліту, щоб запобігти виникненню функціональних розбіжностей між монолітом та новим мікросервісом.

Оскільки мікросервіс це незалежна частина нашої інформаційної системи, ми проводимо його повне незалежне тестування, щоб переконатися в коректності роботи та відповідності очікуваній поведінці.

Перевірка. На наступному етапі, ми переробляємо інтерфейсні методи в моноліті таким чином, щоб вони стали диспетчерами (dispatchers). Ці диспетчери дозволяють перемикати реалізацію функціональності між внутрішніми модулями моноліту та зовнішнім мікросервісом в залежності від налаштувань конфігурації системи. Це дає нам можливість поступово переходити на використання мікросервісу, не порушуючи роботу існуючої системи (рис. 2в).

Протягом певного періоду часу, функціональність буде існувати одночасно в двох варіантах - як мікросервіс та як внутрішній модуль моноліту. Це дозволить нам, у випадку виявлення будь-яких недоліків або проблем з мікросервісом, швидко повернутися до попередньої версії функціональності в моноліті, поки ми вносимо необхідні виправлення та покращення в мікросервіс.

Коли всі необхідні тести успішно завершені і ми переконалися, що мікросервіс функціонує коректно та надійно, ми можемо остаточно видалити відповідний функціонал з монолітної системи. При цьому, ми залишаємо тільки інтерфейсні методи, які перетворюють всі функціональні виклики на запити до мікросервісу. Це означає, що будь-які звернення до даного функціоналу в моноліті тепер будуть автоматично перенаправлятися до мікросервісу для опрацювання.

Після цього етапу, мікросервіс стає повноцінною частиною нашої інформаційної системи, а моноліт виступає в ролі маршрутизатора, який перенаправляє виклики на мікросервіс. Однак, він все ще не є незалежним, оскільки використовує спільну базу даних з монолітом. Наступним кроком є декомпозиція частини бази даних, за яку відповідає мікросервіс.

Декомпозиція домену бази даних. При декомпозиції, важливо забезпечити відповідність обмеженому контексту. База даних мікросервісу повинна містити тільки ті дані, які безпосередньо стосуються функціональності сервісу та його предметної області. Необхідно уникати включення даних, які виходять за межі відповідальності сервісу, щоб зберегти його автономність та незалежність. Це дозволяє мікросервісу зосередитися на своїй основній функції та мінімізувати залежності від інших частин інформаційної системи.

В мікросервісній архітектурі допускається денормалізація даних для забезпечення автономності сервісів. На відміну від традиційного підходу до проектування бази даних, де дані нормалізуються для уникнення дублювання, кожен мікросервіс може мати свою власну оптимізовану схему бази даних. Це означає, що деякі дані можуть бути продубльовані в різних сервісах, якщо це необхідно для забезпечення незалежності та продуктивності. Денормалізація даних дозволяє мікросервісу швидко отримувати необхідну інформацію без звернення до інших сервісів, що зменшує затримки та підвищує автономність.

Під час декомпозиції частини бази, ми стикнемося з новим викликом - забезпечення цілісності даних у розподіленій системі. У монолітній архітектурі, цілісність бази даних зазвичай забезпечується засобами самої системи управління базами даних (СУБД) за допомогою зовнішніх ключів. Однак, при переході до мікросервісної архітектури з незалежними базами даних, ми змушені реалізувати власні механізми для підтримки цілісності та узгодженості даних. Замість цього, рекомендується використовувати асинхронні механізми комунікації, такі як черги повідомлень або подій, для підтримки узгодженості даних між сервісами [3]. Для прикладу, коли з інформаційної системи видаляється користувач, ця подія повинна бути поширена на всі мікросервіси, які зберігають дані, пов'язані з цим користувачем. Кожен мікросервіс повинен отримати відповідне повідомлення і видалити всі дані користувача, за які він відповідає, зі своєї бази даних. Це може бути реалізовано за допомогою шини повідомлень (message bus) або інших механізмів асинхронної комунікації між сервісами.

Забезпечення цілісності даних у розподіленій інформаційній системі вимагає ретельного проектування та реалізації додаткових механізмів синхронізації та узгодженості даних. Це може включати використання розподілених транзакцій, компенсуючих операцій (compensating transactions), подійно-орієнтованої архітектури (event-driven architecture) тощо.

Після успішного розділення бази даних і реалізації механізмів забезпечення цілісності даних, наш мікросервіс стає повністю незалежним і самодостатнім компонентом системи. Він має

власну базу даних, яка оптимізована під його потреби, і може розвиватися та масштабуватися окремо від інших частин інформаційної системи (рис. 2г).

Ітерація завершена - переходимо до декомпозиції наступного функціонального блоку. Процес повторюється, доки моноліт не буде повністю розділений на мікросервіси. У колишньому коді моноліту залишаться лише методи, які перенаправляють виклики на відповідні мікросервіси. На цьому етапі моноліт можна замінити на API Gateway сервіс (рис. X), який буде слугувати єдиною точкою входу для клієнтських запитів та забезпечувати маршрутизацію до відповідних мікросервісів (рис. 3).

Кожен з мікросервісів в кінцевому результаті отримає свою власну базу даних, яка буде містити тільки ті таблиці та дані, які безпосередньо стосуються функціональності даного сервісу. Це означає, що кожен мікросервіс буде реалізовувати лише частину загальної схеми бази даних, яка відповідає його потребам та обов'язкам. Фізично це може бути окремий сервер бази даних, віртуальна машина або контейнер, залежно від обраної технології та інфраструктури. Кожен інстанс бази даних буде мати свої власні ресурси, налаштування та засоби управління.

Така архітектура забезпечує високий рівень незалежності між мікросервісами. Кожен сервіс має повний контроль над своєю базою даних і може вільно змінювати схему даних, оптимізувати запити та масштабувати базу даних відповідно до своїх потреб, не впливаючи на інші сервіси.

Це також дозволяє кожному мікросервісу обрати найбільш підходящу для нього технологію баз даних. Наприклад, один сервіс може використовувати реляційну базу даних, таку як PostgreSQL, для зберігання структурованих даних, тоді як інший сервіс може обрати документо-орієнтовану базу даних, таку як MongoDB, для зберігання неструктурованих даних.

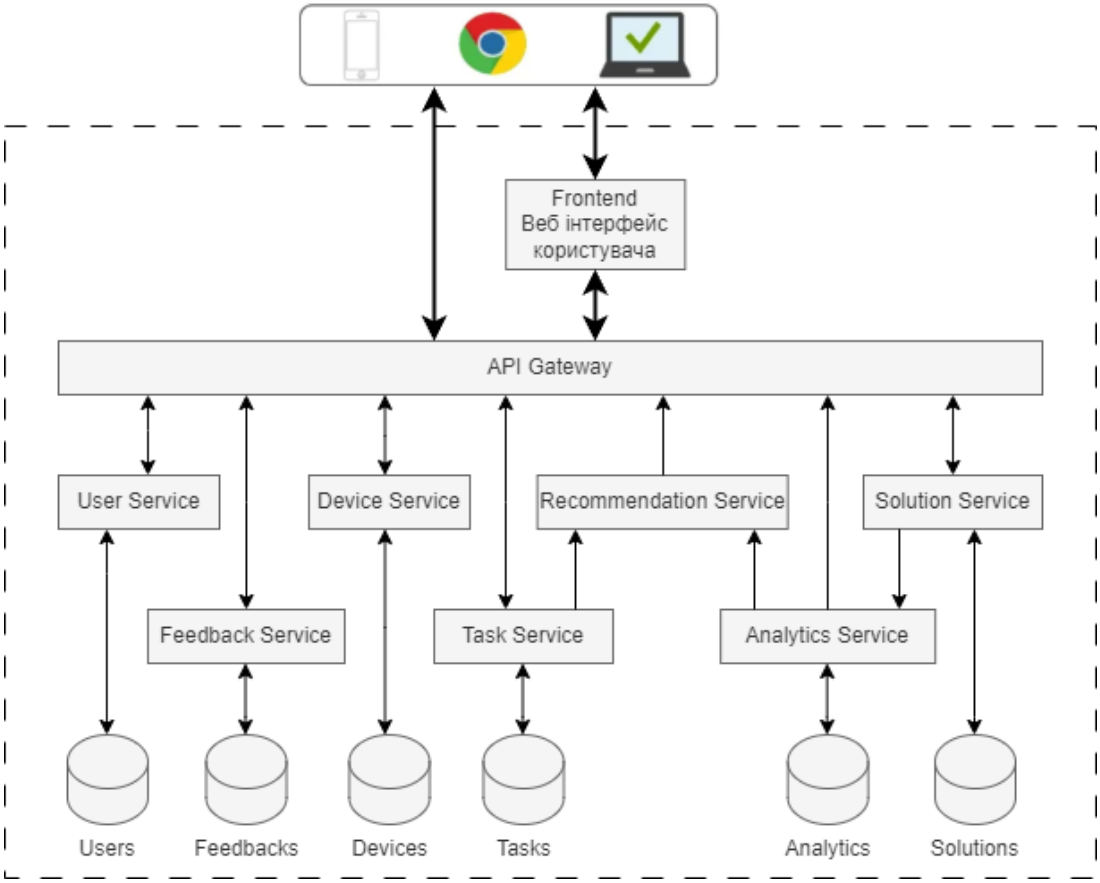

Рис. 3. Схема мікросервісної архітектури з розподіленою базою даних.

Незалежність баз даних мікросервісів також спрощує процес розгортання та масштабування системи. Кожен сервіс може бути розгорнутий та масштабований окремо, без необхідності координації з іншими сервісами. Це дозволяє більш ефективно використовувати ресурси та адаптуватися до змін навантаження.

Під час переходу до мікросервісної архітектури, ми провели ретельний аналіз та проектування API та протоколів обміну даними між сервісами. Для кожного мікросервісу ми визначили набір API-ендпоінтів, які дозволяють іншим сервісам та клієнтським додаткам

взаємодіяти з ним та отримувати необхідні дані або виконувати певні операції. Ми задокументували ці ендпоінти за допомогою специфікацій OpenAPI (Swagger) [8], що забезпечує чітке розуміння доступних методів, параметрів та форматів даних.

Для обміну даними між мікросервісами ми обрали формат JSON (JavaScript Object Notation), який дозволяє структуровано та ефективно передавати дані через мережу, незалежно від мови програмування чи технологічного стеку кожного сервісу.

Ми також реалізували додаткові протоколи та механізми для забезпечення безпечної та надійної комунікації між мікросервісами:

1. Ми використали HTTPS (Hypertext Transfer Protocol Secure) для шифрування та захисту даних під час передачі через мережу.

2. Для аутентифікації та авторизації запитів між сервісами ми впровадили використання JWT (JSON Web Tokens) [8].

3. Для асинхронної та подійно-орієнтованої взаємодії між сервісами ми використали Message Broker Apache Kafka. Це дозволяє сервісам публікувати події та підписуватися на них, забезпечуючи механізми обміну управляючою інформацією між різними мікросервісами.

В результаті проведеної роботи, ми успішно перейшли від монолітної архітектури до мікросервісної з розподіленою базою даних. Однак, незважаючи на досягнутий прогрес, ми все ще не досягли нашої основної мети - можливості використання спільної бази даних завдань у різних регіонах.

Географічний розподіл бази даних. Завдяки ізольованості кожного мікросервісу, ми можемо легко переконфігурувати Tasks Service таким чином, щоб він використовував не локальну базу даних, а спільну віддалену базу, яка знаходиться в іншому регіоні. Це дозволить нам досягти нашої цілі: завдання будуть спільними для всіх регіонів, тоді як приватні дані користувачів зберігатимуться у відповідній локації.

Проте, перш ніж розпочати таке об'єднання баз даних завдань з різних регіонів, необхідно ретельно продумати механізми забезпечення узгодженості даних. Фактично, ми плануємо об'єднати кілька незалежних баз даних в єдину горизонтально розподілену базу, що може призвести до виникнення певних проблем з цілісністю даних.

Розглянемо, наприклад, записи в таблиці завдань, які містять зовнішні ключі, що посилаються на користувачів, які завантажили ці завдання. В кожному регіоні є свій власний набір користувачів, і при об'єднанні баз даних можуть виникнути конфлікти ідентифікаторів. Два різних користувача з різних регіонів можуть мати однаковий ідентифікатор, що призведе до невідповідності зовнішніх ключів та порушення цілісності даних.

Для забезпечення коректної роботи об'єднаної бази даних, нам необхідно гарантувати унікальність ідентифікаторів для всіх сутностей незалежно від локації. Одним із можливих підходів є використання концепції multi-tenancy, де кожен ідентифікатор складається з двох частин: tenant ідентифікатора (ідентифікатора регіону в нашому випадку) та ідентифікатора сутності [9]. При цьому, всі основні та зовнішні ключі перетворюються на комплексні ключі, що складаються з двох значень.

Такий підхід надає значну перевагу - завжди можна легко визначити, якому регіону належить кожен запис у базі даних. Це дозволяє реалізувати ефективний роутинг запитів до інформаційної системи на основі ідентифікатора регіону. Коли надходить запит, інформаційна система аналізує ідентифікатор регіону в ключах і автоматично перенаправляє запит до відповідного регіонального сервера.

Завдяки цьому механізму, ми можемо забезпечити прозорість розподілу даних між регіонами для користувачів та інших компонентів системи. З точки зору клієнта, система виглядає як єдине ціле, в той час як внутрішньо дані розподілені та опрацьовуються у відповідних регіонах.

Якщо використання комплексних ідентифікаторів, з якоїсь причини не бажано, як альтернативу, можна запропонувати використання універсальних унікальних ідентифікаторів (UUID) для всіх сутностей у системі. UUID є 128-бітним значенням, яке генерується за допомогою спеціального алгоритму. Це стандартний формат ідентифікаторів, який гарантує унікальність не тільки в межах однієї інформаційної системи, але й у глобальному масштабі [10].

Змінивши тип всіх ідентифікаторів на UUID, ми можемо забезпечити унікальність ідентифікаторів для всіх сутностей у нашій розподіленій системі незалежно від регіону. Кожен мікросервіс зможе генерувати ідентифікатори для своїх сутностей самостійно, без необхідності координації з іншими сервісами.

На даному етапі в інформаційній системи залишається важливий недолік. Розміщення бази даних завдань лише в одному, умовно головному, регіоні, до якого віддалено підключаються інші регіони, може призвести до певних проблем з продуктивністю та масштабованістю системи.

По-перше, необхідно враховувати затримки при підключенні до географічно віддаленої локації. Якщо база даних завдань знаходиться в одному регіоні, а користувачі з інших регіонів звертаються до неї через мережу, це може призвести до значних затримок при опрацюванні запитів.

По-друге, концентрація великої кількості запитів в єдиній базі даних може перетворити її на "вузьке місце" (bottleneck) в інформаційній системі. Це може призвести до перевантаження бази даних, збільшення часу на отримання відгуку та потенційних збоїв. У результаті, затримки при опрацюванні запитів можуть стати неприйнятними для користувачів.

Крім того, з точки зору розгортання та підтримки інфраструктури, бажано, щоб усі регіони розгорталися однаково. Якщо кожен регіон має ідентичну структуру та конфігурацію, це спрощує процес розгортання нових регіонів та управління ними. Однакова інфраструктура полегшує моніторинг, оновлення та масштабування інформаційної системи, оскільки всі регіони мають однакові компоненти та налаштування.

Мікросервіс завдань на даному етапі виконує дві незалежні функції: повернення метаданих завдань для успішного функціонування мобільного додатку, функцію поповнення і модерації бази даних завдань. Операції редагування метаданих завдань не є критичними з точки зору швидкодії, оскільки виконуються набагато рідше, ніж операції читання. Тому було вирішено розділити Task Service на два мікросервіси: перший дозволяє лише читати дані про завдання (Task Reader Service), інший відповідає за адміністрування бази даних завдань (Task Admin Service).

Task Admin Service, розгортається разом з базою даних завдань в окремому центральному регіоні. Цей сервіс відповідає за додавання нових завдань, редагування існуючих завдань, модерацію контенту та інші адміністративні операції. Зміни, внесені в базу завдань, автоматично реплікуються засобами СУБД до локальних баз даних завдань у кожному регіоні, забезпечуючи глобальну актуальність даних.

Task Reader Service розгортається в кожному регіоні та взаємодіє з локальною read-only реплікою бази завдань. Цей сервіс опрацьовує запити від мобільних додатків та інших компонентів інформаційної системи, які потребують доступу до метаданих завдань. Завдяки локальному розміщенню бази даних, Task Reader Service забезпечує швидкий доступ до даних та мінімізує затримки при опрацюванні запитів.

Для забезпечення безпеки та авторизації доступу до Task Admin Service використовується механізм JWT авторизації. JWT токен генерується в тому регіоні, де авторизується адміністратор або модератор, після успішної перевірки їх облікових даних та прав доступу.

Кінцевий вигляд розподіленої бази даних для даної інформаційної системи зображено на рис.1.

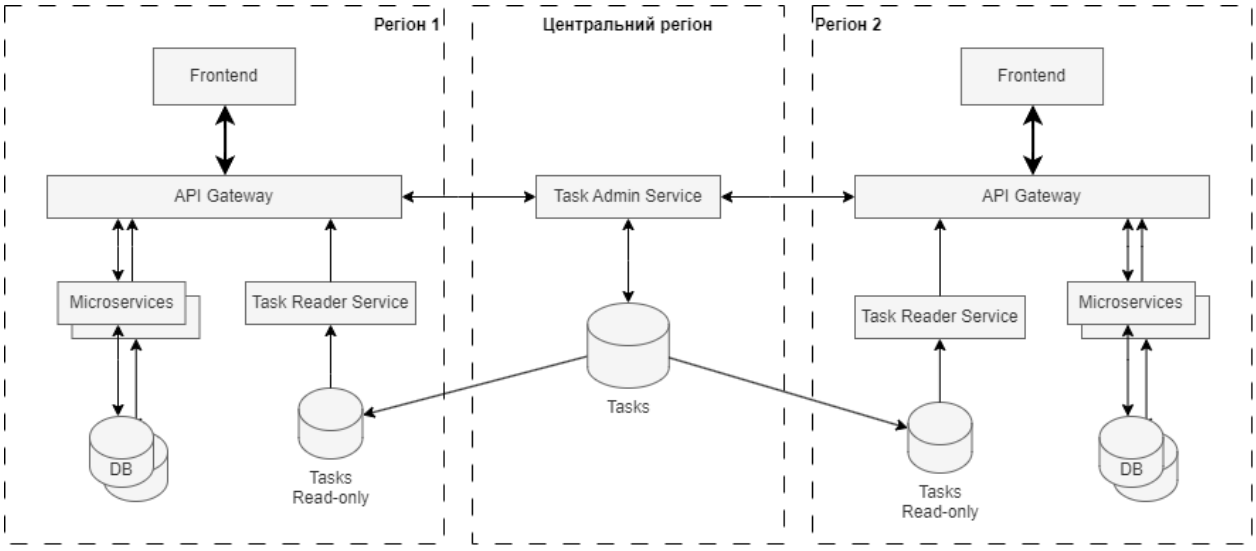

Рис. 4. Схема розподіленої інформаційної системи.

В результаті проведеної роботи, інформаційна система для розвиваючого контролю роботи мобільних пристроїв була успішно переведена з монолітної архітектури на мікросервісну з розподіленою базою даних. Система тепер відповідає всім поставленим цілям і вимогам:

1. Забезпечена можливість зберігання та опрацювання персональних даних користувачів у відповідності до регіональних нормативних вимог. Дані користувачів зберігаються в окремих базах даних у кожному регіоні, що дозволяє дотримуватися законодавства про захист персональних даних.

2. Забезпечена можливість використання спільної бази завдань у всіх регіонах.

3. Досягнута висока доступність та відмовостійкість інформаційної системи завдяки розподіленій архітектурі.

4. Досягнута висока продуктивність та низька затримка опрацювання запитів завдяки локальному розгортанню сервісів у кожному регіоні. Запити користувачів опрацьовуються в найближчому географічному регіоні, що мінімізує затримки та покращує швидкість відгуку інформаційної системи.

5. Забезпечена масштабованість та гнучкість інформаційної системи завдяки мікросервісній архітектурі. Це дозволить ефективно справлятися зі зростаючим навантаженням та швидко додавати нові функції.

Висновки. У статті запропоновано та досліджено метод переходу від монолітної архітектури до мікросервісної з розподіленою базою даних на прикладі інформаційної системи для розвиваючого контролю роботи мобільних пристроїв. Розроблений метод включає в себе етапи поступової декомпозиції монолітної системи на незалежні мікросервіси, розподілу бази даних між регіонами, а також реалізації механізмів для підтримки узгодженості та цілісності даних у розподіленій системі.

Головною перевагою запропонованого методу є можливість здійснювати всі зміни короткими ітераціями на працюючій інформаційній системі, без необхідності зупинки розроблення основної функціональності. Це дозволяє поступово та керовано переходити від монолітної до мікросервісної архітектури, не порушуючи роботу системи та не призупиняючи її розвиток.

Дослідження показали, що перехід до мікросервісної архітектури забезпечує високу модульність, масштабованість та адаптивність системи до змін бізнес-вимог. Отриманий досвід може бути корисним для розробників та архітекторів, які стикаються з подібними викликами при розробленні інформаційних систем.

Подальші дослідження будуть спрямовані на всебічний аналіз ефективності переходу до мікросервісної архітектури з розподіленою базою даних. Одним з ключових аспектів є аналіз економічної доцільності та обґрунтованості такого переходу для різних типів інформаційних систем та бізнес-вимог. Це включатиме детальну оцінку витрат на розроблення, розгортання та підтримку мікросервісної архітектури, а також порівняння цих витрат з потенційними перевагами та вигодами, які може принести така трансформація.

Також важливо дослідити вплив мікросервісної архітектури на процеси розроблення, тестування та розгортання програмного забезпечення. Потрібно проаналізувати, як перехід до мікросервісів впливає на швидкість розроблення, частоту релізів, якість коду та можливості автоматизації процесів. Це дозволить оцінити, наскільки ефективною є мікросервісна архітектура з точки зору підвищення продуктивності розроблення та скорочення часу виходу на ринок.

Окрім економічних факторів, важливо дослідити технологічну ефективність мікросервісної архітектури. Необхідно провести емпіричні дослідження та збір даних з реальних систем, щоб оцінити, наскільки ефективно мікросервісна архітектура справляється з різними навантаженнями та сценаріями використання.

Для формального обґрунтування ефективності переходу до мікросервісної архітектури з розподіленою базою даних необхідно розробити математичні моделі та провести кількісний аналіз. Це може включати моделювання продуктивності системи, аналіз масштабованості та надійності з використанням теорії масового обслуговування [11], а також застосування методів оптимізації для визначення оптимальної конфігурації та розподілу ресурсів у мікросервісній архітектурі.

Список бібліографічного опису

1. Драгоні Н. Мікросервіси: вчора, сьогодні та завтра / Н. Драгоні, С. Джіаллоренцо, А. Л. Лафуенте, М. Маззара, Ф. Монтезі, Р. Мустафін, Л. Сафіна // Present and ulterior software engineering. – 2017. – С. 195-216. - DOI: https://doi.org/10.1145/343477.343502

2. Фаулер М. Мікросервіси: визначення нового архітектурного терміну [електронний ресурс] / М. Фаулер, Дж. Льюїс // martinfowler.com. – 2014. – Режим доступу: https://martinfowler.com/articles/microservices.html
3. Ньюмен С. Будування мікросервісів: проектування дрібнозернистих систем / С. Ньюмен. – O'Reilly Media, 2015. – 280 с. – ISBN 978-1491950357.
4. Прамод Д. Огляд стратегій розподілу та реплікації даних у розподілених базах даних / Д. Прамод, К. Венкатарамана, С. Фані Кумар // International Journal of Advanced Computer Research. – 2018. – Том 8, № 36. – С. 80-90. DOI: http://dx.doi.org/10.30534/ijatcse/2019/117852019
5. Брюер Е. На шляху до надійних розподілених систем / Е. Брюер // Матеріали дев'ятнадцятого щорічного симпозіуму ACM з принципів розподілених обчислень. – 2000. – С. 7. – DOI: https://doi.org/10.1145/343477.343502.
6. Ньюмен С. Від моноліту до мікросервісів: еволюційні шаблони для перетворення вашого моноліту / С. Ньюмен. – O'Reilly Media, 2019. – 272 с. – ISBN 978-1492047841.
7. Гролінгер К. Виклики великих даних: технології, моделі та парадигми / К. Гролінгер, В. А. Хігашіно, А. Тіварі, М. А. Капрец // Управління даними в хмарних, грід та P2P системах. – Springer, Berlin, Heidelberg, 2013. – С. 1-15. – DOI: https://doi.org/10.1007/978-3-642-40053-7_1.
8. Амундсен М. Проектування та створення чудових веб-API: надійних, стабільних та стійких / М. Амундсен. – Pragmatic Bookshelf, 2020. – 275 с. – ISBN 978-1680506808.
9. Вайлдер Б. Шаблони хмарної архітектури: використання Microsoft Azure / Б. Вайлдер. – O'Reilly Media, 2020. – 622 с. – ISBN 978-1492040675.
10. Пачеко Ф. Практична архітектура програмного забезпечення з C# 10 та .NET 6: проектування програмних рішень з використанням мікросервісів, DevOps та шаблонів проектування для Azure, 2-е видання / Ф. Пачеко. – Packt Publishing, 2022. – 650 с. – ISBN 978-1800566040.
11. Гаутам Н. Аналіз черг: методи та застосування / Н. Гаутам. – CRC Press, 2012. – 802 с. – ISBN 978-1439806586.

References

1. Dragoni N. Microservices: yesterday, today, and tomorrow / N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, L. Safina // Present and ulterior software engineering. – 2017. – P. 195-216. - DOI: https://doi.org/10.1145/343477.343502
2. Fowler M. Microservices: a definition of this new architectural term [internet resource] / M. Fowler, J. Lewis // martinfowler.com. – 2014. – Access mode: https://martinfowler.com/articles/microservices.html
3. Newman S. Building Microservices: Designing Fine-Grained Systems / S. Newman. – O'Reilly Media, 2015. – 280 p. – ISBN 978-1491950357.
4. Pramod D. A survey on data partitioning and replication strategies in distributed databases / D. Pramod, K. Venkataramana, S. Phani Kumar // International Journal of Advanced Computer Research. – 2018. – Vol. 8, No. 36. – P. 80-90. DOI: http://dx.doi.org/10.30534/ijatcse/2019/117852019
5. Brewer E. Towards robust distributed systems / E. Brewer // Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing. – 2000. – P. 7. – DOI: https://doi.org/10.1145/343477.343502.
6. Newman S. Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith / S. Newman. – O'Reilly Media, 2019. – 272 p. – ISBN 978-1492047841.
7. Grolinger K. Challenges in Big Data: Technologies, Models, and Paradigms / K. Grolinger, W. A. Higashino, A. Tiwari, M. A. Capretz // Data Management in Cloud, Grid and P2P Systems. – Springer, Berlin, Heidelberg, 2013. – P. 1-15. – DOI: https://doi.org/10.1007/978-3-642-40053-7_1.
8. Amundsen M. Design and Build Great Web APIs: Robust, Reliable, and Resilient / M. Amundsen. – Pragmatic Bookshelf, 2020. – 275 p. – ISBN 978-1680506808.
9. Wilder B. Cloud Architecture Patterns: Using Microsoft Azure / B. Wilder. – O'Reilly Media, 2020. – 622 p. – ISBN 978-1492040675.
10. Pacheco F. Hands-On Software Architecture with C# 10 and .NET 6: Architecting software solutions using microservices, DevOps, and design patterns for Azure, 2nd Edition / F. Pacheco. – Packt Publishing, 2022. – 650 p. – ISBN 978-1800566040.
11. Gautam N. Analysis of Queues: Methods and Applications / N. Gautam. – CRC Press, 2012. – 802 p. – ISBN 978-1439806586.