УДК 004.8:65.05:681.5 DOI: 10.31673/2786-8362.2024.010707

Катков Ю.І., д.т.н., **Зінченко О.В.,** д.т.н.,
Березовська Ю.В., PhD, **Кладько І.М.**

ОСОБЛИВОСТІ СЕРВІС-ДИЗАЙНУ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ В ХМАРНИХ ОБЧИСЛЕННЯХ В УМОВАХ КОНТЕЙНЕРНОЇ ВІРТУАЛІЗАЦІЇ

Katkov Y.I., Zinchenko O.V., Berezovska Yu.V., Kladko I.M. Special features of service design in microservices architecture in cloud computing with container virtualization. The article addresses the issue of service design in microservices architecture in cloud computing under conditions of container virtualization. Service design is the process of modeling a service to facilitate interaction between the provider and users. The foundation of service design lies in understanding user behavior, their needs, and motivations to create a service that is convenient and memorable, increasing the chances of reuse. Microservices architecture technology is an approach that involves breaking down applications into smaller and more manageable blocks (microservices), each performing a specific function. This approach differs from monolithic programs where all components are tightly interconnected and reside in a single codebase. The new approach opens up unique possibilities for utilizing various technologies, including containerization, orchestration, hybrid cloud, and more, offering excellent opportunities for optimizing applications through microservices. To address the identified challenges in the article: a) it examines how the new approach introduces complexities, such as the lack of proper monitoring and event logging; the possibility of incorrect service detailing; complexities in data management and synchronization; weaknesses in communication and automation processes; manifestations of bottlenecks in terms of productivity and scalability, among others; b) it is demonstrated that all these complexities can be resolved through effective service design in microservices architecture. This means that when designing microservices architecture in cloud computing with container virtualization, the challenge of effective service design is addressed by creating a balance between the detailing of services and their functionality to achieve maximum productivity and system functionality. Solving this task is possible primarily through an understanding of the principles of construction and application of microservices in cloud computing under conditions of container virtualization.

Keywords: microservices, microservices architecture, cloud computing, container virtualization

Катков Ю. І., Зінченко О.В., Березовська Ю.В Кладько І.М. Особливості сервіс-дизайну мікросервісної архітектури в хмарних обчисленнях в умовах контейнерної віртуалізації. Стаття присвячена проблемі сервіс-дизайну мікросервісної архітектури в хмарних обчислення в умовах контейнерної віртуалізації. Сервіс-дизайн — це процес моделювання послуги для створення взаємодії між постачальником та користувачами. Основою сервіс-дизайну є розуміння поведінки користувачів, їхніх потреб та мотивацій задля створення послуги, яка є зручною і запам'ятовується, збільшуючи шанси на повторне користування. Технологія мікросервісної архітектури - це підхід, що дозволяє створювати додатки, при якому вони розбиваються на дрібніші та більш керовані блоки (мікросервіси), кожен з яких виконує певну функцію. Цей підхід відрізняється від монолітних програм, де всі компоненти тісно пов'язані один з одним і знаходяться в одній базі коду. Новий підхід створює унікальні можливості застосування різних технологій, в тому числі: контейнеризації, оркестровки, гібридної хмари і так далі, що створює чудові можливості оптимізації додатків за допомогою мікросервісів. Для вирішення вказаної проблеми в статті: а) розглянути як новий підхід створює складності, наприклад, відсутність належного моніторингу та ведення журналу подій; можливість появи неправильної деталізації послуги; складності управління даними та синхронізацією; слабкі сторони процесів комунікації та автоматизації; прояви вузьких місць щодо продуктивності та масштабованості та інші; б) показано, що всі ці складності можна вирішити за допомогою грамотного сервіс-дизайну мікросервісної архітектури. Це означає, що при проектуванні мікросервісної архітектури в хмарних обчислення в умовах контейнерної віртуалізації проблема грамотного сервіс-дизайну мікросервісної архітектури вирішується за рахунок створення балансу між деталізацією сервісів та їх функціональністю, щоб досягти максимальної продуктивності та функціональності системи. Вирішення цього завдання можливе в першу чергу за рахунок розуміння принципів побудови та застосування мікросервісів в хмарних обчислення в умовах контейнерної віртуалізації.

Ключові слова: мікросервіси, мікросервісна архітектури, хмарні обчислення, контейнерна віртуалізація

Вступ

Мікросервісна архітектура - це підхід до створення додатків, при якому вони розбиваються на дрібніші та більш керовані блоки (мікросервіси), кожен з яких виконує певну функцію. Цей підхід відрізняється від монолітних програм, де всі компоненти тісно пов'язані один з одним і знаходяться в одній базі коду. Новий підхід створює можливості застосування різних технологій, в тому числі: контейнеризації, оркестровки, гібридної хмари і так далі, що створює можливості оптимізації додатків за допомогою мікросервісів. Але новий підхід створює нові складності під час експлуатації, наприклад, виникає відсутність належного моніторингу та ведення журналу подій; можливість появи неправильної деталізації послуги; складності управління даними та синхронізацією; виникають слабкі сторони процесів комунікації та автоматизації; створюються прояви вузьких місць щодо продуктивності та масштабованості та інші . Аналіз цих складностей під час експлуатації вказує на те, що всі ці проблеми можна вирішити за допомогою грамотного сервіс-дизайну мікросервісної архітектури.

Постановка завдання. Сервіс-дизайн – це процес моделювання послуги для створення взаємодії між постачальником та користувачами, що враховує всі відносини між ними і балансує потреби обох сторін. Основою сервіс-дизайну є розуміння поведінки користувачів, їхніх потреб та мотивацій задля створення послуги, яка є зручною і запам'ятовується, збільшуючи шанси на повторне користування [1]. Тому виникає проблема грамотного сервіс-дизайну мікросервісної архітектури. Це означає, що при проектуванні мікросервісної архітектури в хмарних обчислення в умовах контейнерної віртуалізації необхідно щоб розробник з одного боку враховував безліч факторів, таких як тип програми, потреби бізнесу та очікуваний обсяг роботи, а з іншого він повинен оцінити кожну послугу індивідуально та визначити, які з них можна об'єднати, а які, навпаки, поділити. Тобто проблема грамотного сервіс-дизайну мікросервісної архітектури вирішується за рахунок створення балансу між деталізацією сервісів та їх функціональністю, щоб досягти максимальної продуктивності та функціональності системи. Вирішення цього завдання можливе в першу чергу за рахунок опису та розуміння принципів побудови та застосування мікросервісів в хмарних обчислення в умовах контейнерної віртуалізації.

Аналіз останніх досліджень. Проблема грамотного сервіс-дизайну мікросервісної архітектури досить складна. Вона стосується багатьох питань. Так в [2] автори підкреслюють важливість та актуальність визначення якості обслуговування (QoS) у сфері хмарних обчислень. Дослідження вказує на значущі виклики та обмеження, такі як час відгуку, пропускна здатність, мінімальна недоступність послуг, вартість, терміни виконання та завершення, які визначають область QoS в хмарних обчисленнях за рахунок застосування технологій контейнеризації. В [3] розглядаються методи композиції мікросервісів, наводяться конкретні приклади мікросервісів та обговорюються переваги контейнеризації як ефективного середовища для розгортання та виконання мікросервісів. В багатьох інших роботах [4, 5, 6] розглядаються інші методи побудови мікросервісів. Але в загальному вигляді основи побудови мікросервісів відносно сервіс-дизайну мікросервісної архітектури в хмарних обчислення в умовах контейнерної віртуалізації не розглянути. Тому розгляд грамотного сервіс-дизайну мікросервісної архітектури в хмарних обчислення в умовах контейнерної віртуалізації у такому вигляді є актуальним та своєчасним для підготовки фахівців.

Метою роботи є вирішення проблеми грамотного сервіс-дизайну мікросервісної архітектури. Для цього ставляться завдання: а) розглянути можливості застосування мікросервісної архітектури в хмарних обчислення в умовах контейнерної віртуалізації; б) показати, що мікросервіси створює складності, наприклад, відсутність належного моніторингу та ведення журналу подій; можливість появи неправильної деталізації послуги; складності управління даними та синхронізацією; виникають слабкі сторони процесів комунікації та

автоматизації; створюються прояви вузьких місць щодо продуктивності та масштабованості та інші; в) показати, що всі ці складності можна вирішити за допомогою грамотного сервіс-дизайну мікросервісної архітектури.

Виклад основного матеріалу дослідження.

Сервіс-дизайн – це процес моделювання послуги для створення взаємодії між постачальником та користувачами. Основою сервіс-дизайну є розуміння поведінки користувачів, їхніх потреб та мотивацій задля створення послуги, яка є зручною і запам'ятовується, збільшуючи шанси на повторне користування. Для розуміння сервіс-дизайну мікросервісної архітектури в хмарних обчислення в умовах контейнерної віртуалізації розглянемо стисло основи побудови мікросервісної архітектури.

До появи мікросервісної архітектури основною технологією були монолітні додатки. Це був традиційних підхід до розробки програмного забезпечення, де весь додаток об'єднується в одному цілісному блоку. Монолітна програма містить функціональні можливості, специфічні для предметної області, і зазвичай поділено на функціональні рівні, такі як зовнішній інтерфейс, бізнес-логіка та сховище даних. Монолітна програма масштабується шляхом клонування всієї програми на кілька комп'ютерів. З появою інновацій, таких як хмарна інфраструктура Software-as-a-Service і Function-as-a-Service, переваги додатків з монолітною архітектурою зменшилися. Це призвело до підвищеного інтересу до мікросервісів [7 - 10].

Мікросервіси – це новий архітектурний та організаційний підхід до розробки та розгортання додатків, що відповідає вимогам гнучкості, масштабованості та надійності сучасних хмарних додатків, при якому програмне забезпечення складається з невеликих незалежних сервісів, які взаємодіють через чітко визначені прикладний програмний інтерфейс (Application Programming Interface – API). Сутність цього підходу у тому, що програма мікросервісів розбивається на незалежні компоненти, які разом забезпечують загальну функціональність програми. Термін «мікросервіс» підкреслює, що програми повинні складатися з сервісів, досить невеликих, щоб відображати незалежні завдання, щоб кожен мікросервіс реалізовував одну функцію. Тобто кожен мікросервіс представляє собою автономний ізольований модуль, якій має свою власну сферу відповідальності та може бути розроблений, розгорнутий та масштабований незалежно від інших. Тому кожен мікросервіс є самостійною одиницею масштабування, що дозволяє гнучке пристосовувати розмір служби згідно з її конкретними вимогами під час виконання. Всі мікросервіси мають чітко визначені алгоритми обміну даними та взаємодії з іншими мікросервісами через стандартизовані інтерфейси, що дозволяє забезпечити їх спільну роботу, навіть якщо вони написані різними мовами або розгорнуті на різних серверах, У цьому контексті мікросервіс виступає як одиниця розробки та розгортання, надаючи можливість кожній службі бути розробленою, розгорнутою та керованою незалежно. Для обслуговування одного запиту користувача до додатку на основі мікросервісів виконується звертання до безлічі мікросервісів для складання потрібної програми. Якщо виникає необхідність оновлення програми, то оновлюються потрібні мікросервіси, тобто підтримується незалежні оновлення. Такий слабкий зв'язок між мікросервісами забезпечує швидкий та надійний розвиток додатків. Їх незалежний, розподілений характер підтримує оновлення, що чергуються, при яких у будь-який момент часу оновлюється тільки підмножина екземплярів мікросервісів. Це дозволяє конкретизувати проблему, оновлення з помилкою можна відкотити до того, як усі екземпляри працювали до оновляться. Аналогічно, мікросервіси зазвичай використовують керування версіями додатку, щоб клієнти бачили узгоджену версію при застосуванні оновлень незалежно від того, з яким екземпляром мікросервісу здійснюється зв'язок.

Технологія мікросервісної архітектури – це підхід, що дозволяє створювати додатки, при якому вони розбиваються на дрібніші та більш керовані блоки (мікросервіси), кожен з яких

виконує певну функцію. Цей підхід відрізняється від монолітних програм, де всі компоненти тісно пов'язані один з одним і знаходяться в одній базі коду [7 - 10]. Новий підхід створює можливості застосування різних технологій, в тому числі: контейнеризації, оркестровки, гібридної хмари і так далі, що створює можливості оптимізації додатків за допомогою мікросервісів. Однією з основних переваг технології мікросервісної архітектури є те, що вона створює можливість масштабувати мікросервіс незалежно, на відміну від гігантських монолітних програм, які масштабуються разом. Це означає, що конкретну функціональну область, яка потребує більшої обчислювальної потужності або пропускної спроможності мережі для задоволення попиту, можна масштабувати, тобто немає необхідності масштабувати інші області програми. Таким чином, мікросервісна архітектура передбачає, що програмний продукт складається з множини таких самодостатніх мікросервісів, які працюють разом у гармонії, створюючи надійну та масштабовану інфраструктуру для сучасних хмарних рішень. Кожен процес мікросервісу може бути розгорнутий окремо на власних серверах чи контейнерах, надаючи гнучкість та швидкість реакції на зміни у навантаженні. Але такий підхід створює ускладнення під час сервіс-дизайну мікросервісної архітектури. Це виникає тому, що мікросервіси володіють такими властивостями: Continuous delivery (безперервною доставкою), Polyglot pipelines (поліглотними конвеєрами), Containers (контейнеризація), Highly distributed environments (високорозподілені середовища), Software-defined everything (все програмне визначення), Everything as a service (все як послуга). Це є їх важливою конкурентною перевагою.

Для розуміння причин ускладнення під час сервіс-дизайну мікросервісної архітектури треба розуміти, що існує три типи мікросервісів: без збереження стану, орієнтовані на дані та агрегатори.

Мікросервіс без збереження стану – це класична модель масштабованості додатків, яка забезпечує рівень з балансуванням навантаження, без збереження стану та загальним зовнішнім сховищем даних для зберігання постійних даних.

Мікросервіс орієнтовані на дані – це мікросервіс з відстеженням стану, він керує своїми власними постійними даними, зазвичай зберігаючи їх локально на серверах, де вони розміщені, щоб уникнути накладних витрат на доступ до мережі та складності міжсервісних операцій. Це забезпечує максимально швидку обробку даних та може усунути необхідність у системах кешування. Крім того, мікросервіси, що масштабуються, з відстеженням стану зазвичай поділяють дані між своїми екземплярами, щоб керувати розміром даних і пропускною здатністю передачі, яку може підтримувати один сервер.

Мікросервіс-агрегатор - це сервіс, який отримує запит, потім надсилає запити кільком сервісам, об'єднує результати та відповідає на запит, що ініціює. Мікросервіс-агрегатор надає однорідні послуги на своєму сайті. Вони об'єднують постачальників послуг на своєму веб-сайті та дозволяють клієнту/користувачу користуватися такими послугами зі свого веб-сайту або зі своєї програми. Класичний приклад агрегаторів сервісів - Uber або Ola.

Мікросервісні програми мають безліч переваг перед монолітними програмами, а саме:

• Надає можливість швидкого впровадження змін, великої гнучкості та високої доступності додатку.

• Кожен мікросервіс відносно невеликий, ним легко керувати та розвивати.

• Кожен мікросервіс можна розробляти та розгортати незалежно від інших сервісів.

• Кожен мікросервіс можна масштабувати незалежно. Наприклад, служба каталогу або корзина покупок може знадобитися більше, ніж служба замовлень. Отже, отримана інфраструктура ефективніше споживатиме ресурси при масштабуванні.

• Кожен мікросервіс ізолює будь-які проблеми. Наприклад, якщо є проблема у службі, це впливає лише на цю службу. Інші служби можуть продовжувати обробляти запити.

• Кожен мікросервіс може використовувати новітні технології. Оскільки мікросервіси автономні та працюють паралельно, можна використовувати новітні технології та платформи, а не використовувати стару платформу, яка могла б використовуватись монолітним додатком.

• Кожен мікросервіс має власну базу даних, що дозволяє повністю відокремити його з інших мікросервісів. За необхідності узгодженість між базами даних із різних мікросервісів досягається за допомогою подій рівня програми.

Однак рішення на основі мікросервісів також має потенційні недоліки:

• Вибір способу поділу програми на мікросервіси може виявитися непростим завданням, оскільки кожен мікросервіс має бути повністю автономним, наскрізним, включаючи відповідальність за джерела даних.

• Розробникам доводиться реалізовувати міжсервісний зв'язок, що ускладнює застосування та збільшує затримки.

• Атомарні транзакції між кількома мікросервісами, як правило, неможливі. Таким чином, бізнес-вимоги мають враховувати можливу узгодженість між мікросервісами.

• У виробничому середовищі виникають операційні складності при розгортанні та керуванні системою, скомпрометованою безліччю незалежних сервісів.

• Пряма взаємодія клієнта з мікросервісом може ускладнити рефакторинг контрактів мікросервісів. Наприклад, згодом може знадобитися зміна способу поділу системи на служби. Одна служба може бути поділена на дві або більше служб, а дві служби можуть об'єднатися. Коли клієнти безпосередньо взаємодіють із мікросервісами, цей рефакторинг може порушити сумісність із клієнтськими додатками.

• Існує загроза виникнення проблем безпеки мікросервісів, а саме: ізоляція (складна проблема безпеки для мікросервісів через особливості розподілу в архітектури); складність гібридної та мультихмарної хмари (оскільки збільшують кількість напрямків атак і ускладнюють управління та захист усієї системи); управління безпекою кількох хмарних провайдерів та локальних середовищ; захист даних, що передаються між службами та зовнішніми джерелами (оскільки кожна служба може мати власне сховище даних та протокол зв`язку, існує ризик витоку, підробки чи перехоплення даних зловмисниками); управління рівнями даних (труднощі забезпечення узгодженої безпеки даних у всіх сервісах); забезпечення мікросервісам відповідний рівень доступу до даних (в архітектурі мікросервісів кожна служба призначена для виконання певних бізнес-можливостей та може вимагати доступу до певних даних).

• Мікросервіси також можуть створювати проблеми, пов`язані з конфіденційністю та дотриманням вимог. У деяких галузях або регіонах правила конфіденційності даних можуть вимагати додаткового контролю над обробкою та зберіганням даних. Може виявитися непросто забезпечити дотримання вимог конфіденційності та відповідності вимогам всіх рівнях даних в архітектурі мікросервісів.

Тому для того щоб мікросервіс працював ефективно, потрібна розуміння основних принципів: безперервна інтеграція, безперервна доставка, безперервне розгортання, управління конфігурацією та розробка через тестування [2 - 4].

Безперервна інтеграція (Continuous integration - CI) гарантує, що код завжди інтегрується до загального репозиторію. Це дозволяє уникнути хаосу, що виникає в результаті громіздких і конфліктуючих комітів коду. Commit/Комміт - це спосіб збереження змін у коді. Коміти – основні конструктивні елементи тимчасової шкали проекту Git. Їх можна розглядати як знімки стану або контрольні точки на часовій шкалі проекту Git. Коміти створюються за допомогою команди git commit, яка робить знімок стану проекту на даний момент часу. Безперервна інтеграція забезпечується за допомогою спеціальних інструментів компілювання, наприклад, Jenkins, якій гарантує, що код компілюється, запуститься та протестується, перш ніж він буде інтегрований з іншими.

Безперервна доставка (Continuous delivery - CD) – це парадигма процесу створення, тестування та внесення покращень у програмний код чи середовище користувача за допомогою автоматизованих інструментів, які складаються в деякий процес-конвеєр CD, тобто під час яких набір кроків призводить до зміни коду, щоб забезпечити потрібний стан. Ось приклад CD: автоматичне оновлення програмного забезпечення на мобільному телефоні, що дозволяє розробникам вирішувати різні ситуації, перш ніж доставляти клієнтам що-небудь. Основна мета CD - зробити випуск програмного забезпечення безпечною та безболісною подією, яку можна виконувати на вимогу розгортання: автоматично і непомітного для користувача. Тобто CD забезпечує безперервну інтеграцію та включає проведення великих регресійних тестів, тестів інтерфейсу користувача та продуктивності, щоб гарантувати, що код готовий до роботи. Для цього типовий конвеєр складається з етапів: складання, тестування та розгортання. На різних етапах можна включити такі дії: вилучення коду із системи контролю версій та виконання збирання; увімкнення етапів для автоматизованих перевірок безпеки, якості та відповідності; підтримка тверджень, коли це необхідно. Типовий конвеєр включає такі елементи: автоматизовані зборки, тести, порядок розгортання та інші. По своїй суті CD є оптимізованим процесом. Конвеєр починається з того, що розробник чи група розробників фіксують свій код у репозиторії вихідного коду. Автоматичні тести (модульні, регресійні, продуктивні тощо) запускаються при кожній перевірці, щоб гарантувати високу якість коду. Після перевірки коду виконавчі файли автоматично розгортаються у проміжному середовищі. На цьому етапі код готовий до запуску у виробництво і може бути застосованій на вимогу, наприклад, автоматично оновити програмне забезпечення на мобільному телефоні.

Безперервне розгортання (Continuous deployment) – це процес, якій забезпечує розвиток застосування коду на крок уперед. Під час цього процесу код автоматично розгортається у робочому середовищі після фіксації на кожному етапі. Це працює за допомогою так званого конвеєра CI/CD (безперервна інтеграція/безперервне розгортання). Він спрямований на покращення доставки програмного забезпечення протягом усього життєвого циклу розробки програмного забезпечення.

Керування конфігураціями дозволяє абстрагувати складність продукту до простих конфігурацій. Це робить програмне забезпечення гнучким, масштабованим та легким.

Розробка через тестування - це гарантує, що код можна буде протестувати та розгорнути за кілька хвилин. Така модель доставки програмного забезпечення забезпечує автоматизоване розгортання і гарантує, що ручні процеси, що використовувалися раніше під час попереднього розгортання, тепер повністю виключені. Завдяки усуненню несподіванок, що виникають в останню хвилину, випуск виробничих версій став рутинною роботою, а не моментом паніки для розробників. Звідси, покладаючись на незалежні сервіси з чіткими межами, мікросервіси схожі на більш традиційну сервіс-орієнтовану архітектуру (Service-oriented architecture - SOA). Така сервіс-орієнтована архітектура дозволяє мікросервісам забезпечувати: зв'язок між клієнтом та мікросервісами, зв'язок між мікросервісами. [1].

Зв'язок між клієнтом та мікросервісами створюється за допомогою мобільного додатоку eShopOnContainers. Він надає взаємодію з контейнерними серверними мікросервісами, використовуючи прямий зв'язок між клієнтом та мікросервісом. Завдяки прямій взаємодії між клієнтом та мікросервісом мобільний додаток відправляє запити до кожного мікросервісу безпосередньо через його загальнодоступну кінцеву точку, використовуючи окремий TCP-порт для кожного мікросервісу. У робочому середовищі кінцева точка зазвичай зіставляється з балансувальником навантаження мікросервісу, який розподіляє запити на доступні екземпляри.

Зв'язок між мікросервісами. Додаток на основі мікросервісів – це розподілена система, яка потенційно працює на декількох сервірах. Кожен екземпляр служби зазвичай є процесом. Служби повинні взаємодіяти з використанням протоколу міжпроцесної взаємодії, такого як

HTTP, TCP, розширений протокол черги повідомлень (AMQP) чи двійкових протоколів, залежно від характеру кожної служби. Існує два поширені підходи до взаємодії між мікрослужбами – це зв'язок REST на основі HTTP при запиті даних та асинхронний обмін повідомленнями під час передачі оновлень між кількома мікрослужбами. REST (REpresentational State Transfer) - це архітектурний стиль, що забезпечує стандарти між комп'ютерними системами в мережі, що спрощує взаємодію систем один з одним. REST-сумісні системи, які часто називають системами RESTful, характеризуються тим, що вони не зберігають стан і поділяють завдання клієнта і сервера.

Асинхронний обмін повідомленнями, керований подіями, має вирішальне значення для поширення змін між кількома мікросервісами. За такого підходу мікросервіс публікує подію, коли відбувається щось примітне, наприклад, коли він оновлює бізнес-об'єкт. Інші мікросервіси підписуються на ці події. Потім, коли мікросервіс отримує подію, він оновлює власні бізнес-об'єкти, що, у свою чергу, може призвести до публікації більшої кількості подій. Ця функціональність публікації-підписки зазвичай досягається за допомогою шини подій. Шина подій дозволяє здійснювати зв'язок публікації та підписки між мікросервісами, не вимагаючи, щоб компоненти були явно обізнані один про одного. З точки зору програми шина подій це просто канал публікації-підписки, доступний через інтерфейс. Однак спосіб реалізації шини подій може бути різним. Наприклад, реалізація шини подій може використовувати RabbitMQ, службову шину Azure або інші службові шини, такі як NServiceBus та MassTransit. Шина подій eShopOnContainers, реалізована з використанням RabbitMQ, забезпечує функціональність асинхронної публікації-підписки «один-до-багатьох». Це означає, що після публікації події одна і та сама подія може прослуховувати кілька передплатників. Цей підхід зв'язку один до багатьох використовує події для реалізації бізнес-транзакцій, що охоплюють кілька служб, забезпечуючи кінцеву узгодженість між службами. Остаточно-узгоджена транзакція складається із серії розподілених кроків. Таким чином, коли мікрослужба профілю користувача отримує команду UpdateUser, вона оновлює інформацію про користувача у базі даних і публікує подію UserUpdated у шині подій. І мікросервіс корзини, і мікросервіс замовлень підписалися на отримання цієї події і у відповідь оновлюють інформацію про своїх покупців у відповідних базах даних. Кожен мікросервіс під час хмарних обчислювань використовує процеси контейнерної віртуалізації, тобто в кожен контейнер закладаються інструменти автоматичного тестування, що забезпечують максимальне покриття коду, тому безпека, продуктивність, інтеграція та інші функції коду не залишають нічого випадкового.

Контейнерна віртуалізація – представляє собою парадигму обчислення, що забезпечує ізоляцію та ефективне управління віртуальними середовищами, відомими як контейнери. Цей інноваційний підхід дозволяє упаковувати додатки та їх залежності в легкі та портативні контейнери, що можуть бути використані для виконання практично в будь-якому середовищі, яке підтримує контейнеризацію.

Контейнеризація – це підхід до розробки програмного забезпечення, у якому додаток та необхідне для його роботи програмне забезпечення для запуску програми (код, середовище виконання, системні інструменти, бібліотеки залежності, драйвера, файлова система та інші, крім загальної операційної системи на сервері) упаковано всередині об'єкта-контейнера. Це ізольована, керована ресурсами і переносне операційне середовище, в якому програма може працювати, не торкаючись ресурсів інших контейнерів або хоста. Перевага контейнерно-орієнтованого підходу до розробки та розгортання полягає в тому, що він усуває більшість проблем, що виникають через неузгоджені налаштування середовища, та проблем, які з ними пов'язані. Крім того, контейнери забезпечують швидке масштабування додатків шляхом створення екземплярів нових контейнерів у міру потреби. Таким чином, контейнер виглядає та діє як нещодавно встановлений фізичний комп'ютер чи віртуальна машина. Контейнери добре підходять для програми мікросервісів, оскільки різні мікросервіси, що становлять

додаток, можуть бути швидко розгортається у різних групах контейнерів. Програми також легко масштабуються у контейнерному середовищі, особливо коли виявлення служб автоматизовано, та оркестратори допомагають керувати конфігураціями автоматично. Ключові поняття при створенні та роботі з контейнерами наступні:

• Хост контейнера – фізична чи віртуальна машина, налаштована для розміщення контейнерів. Хост контейнера запускатиме один або кілька контейнерів.

• Зображення (образ) контейнера – образ складається з об'єднання багаторівневих файлових систем, що розташовані один над одним, і є основою контейнера. У образу немає стану, і він ніколи не змінюється при розгортанні в різних середовищах.

• Контейнер – це екземпляр зображення під час виконання.

• Образ операційної системи контейнера - це перший рівень серед потенційно багатьох верств образу, що має контейнер. Операційна система контейнера незмінна і може бути змінена.

• Репозиторій контейнерів - щоразу, коли створюється образ контейнера, образ та його залежності зберігаються у локальному репозиторії. Ці зображення можна використовувати багато разів. Образи контейнерів також можна зберігати в загальнодоступному або приватному реєстрі, наприклад Docker Hub, щоб їх можна було використовувати на різних хостах контейнерів. Такий підхід забезпечує легкий, ефективний та стандартний спосіб ізоляції та переміщення мікросервісів між середовищами та незалежність їх запуску, забезпечують зниження накладних витрат, швидшу розробку додатків та просте впровадження архітектури мікросервісів.

Дійсно кожен контейнер функціонує як самодостатня одиниця, ізольована від інших контейнерів та оптимізована для швидкого розгортання та масштабування. Природа контейнерів забезпечує консистентність та надійність додатків, а також сприяє зменшенню витрат на інфраструктуру. Важливою перевагою цієї технології є полегшення управління обчислювальними ресурсами, що робить її ключовим інструментом у розробці та розгортанні мікросервісів.

Контейнерна віртуалізація стала важливою технологією в розробці та розгортанні мікросервісів, забезпечуючи швидкість і надійність в процесі роботи з додатками. Виконання контейнерів виявилося економічно ефективним заходом з огляду на споживання ресурсів. Найпопулярніші додатки для контейнерної віртуалізації прийнято вважати Docker та Kubernetes. Хоча в одному контейнері можна запускати кілька мікросервісів, це не рекомендується. Щоб керувати кожним мікросервісом незалежно, кожен з них повинен мати власний контейнер Docker, який можна оркеструвати за допомогою Docker Compose або Kubernetes. Для цього необхідно створити Dockerfile для кожного мікросервісу. Проте найпоширенішими типами загроз безпеки контейнерів є: контейнерне шкідливе програмне забезпечення; небезпечні привілеї контейнера; контейнери із конфіденційними даними; трубопровід розвитку; контейнери зображення; реєстри контейнерів; середовище виконання контейнера [7, 8].

Наступним кроком, що необхідно розглянути, є розуміння проблем експлуатації мікросервісів в наслідок: складної залежності, безперервної доставки, все як послуга.

Проблеми експлуатації мікросервісів в наслідок складної залежності. Розуміння залежності у мікросервісах з навколишнім середовищем надзвичайно важливе, тому що воно надзвичайно складне та надзвичайно важко. Складність у тому, що у такі середовища, як монолітні або які засновані на SOA програми, міжсервісні залежності відносно нечисленні та стійкі, що робить їх просто і досить легко ідентифікувати. Зокрема, легко сказати, які на яких серверах працюють програми, наприклад, а також які служби зберігання даних пов'язані з якими додатками. Однак у мікросервісній архітектурі залежності, які пов'язують послуги та інфраструктуру разом набагато складніше. Наприклад, мікросервіси часто розгортаються з

використанням контейнерів, які розподілені по кластерах серверів; мережеві маршрути постійно коригуються залежно від навантаження на відповідь зміни попиту; порти, які мікросервіси використовують для спілкування один з одним, можуть змінюватися без попередження. Тому динамічне середовище мікросервісів контролювати надзвичайно складно, якщо взагалі можливо. Ця складність створює утруднені, наприклад, проблему із службою зберігання. Тобто служба зберігання є єдиним способом ідентифікувати точну причину інциденту для його усунення, але помилка в роботі служби зберігання може бути результатом збою сервера, помилки кодування, збою диску або уповільнення роботи віртуальної мережі, що з'єднує масив зберігання даних до решти додатка.

Проблеми експлуатації мікросервісів в наслідок безперервної доставки. Послуги з гнучкої розробки коду викликають необхідність швидкої та гнучкої доставки програмного забезпечення. Ефективне спостереження та моніторинг вимагають можливості бачити зміни у реальному часі, налаштування конфігурації моніторингу та активація картки служб кожного разу, коли служба оновлюється. Але у контейнерних середовищах існують значно більше компонентів для моніторингу всередині програми. Тобто замість кількох десятків серверів та додатків, кількість моніторингу об'єктів, які можна було б мати у традиційне середовище, контейнеризоване довкілля може складатися з тисяч окремих ефемерних контейнерів. Фактично, мікросервіс може призвести до експонентного зростання кількості об'єктів для спостереження та моніторингу. Тому в результаті збільшення кількості об'єктів швидкість змін безперервної доставки у навколишньому середовищі зменшується різко вниз.

Проблеми експлуатації мікросервісів в наслідок принципу «все як послуга». Прийняття принципу «все як послуга» означає, що багато організацій більше не володіють базовою інфраструктурою, на якій вони розміщують свої сервіси, мікросервіси та контейнери. Крім того це також означає, що немає простого способу порівняти будь-які залежності між мікросервісами. Це означає, що тепер не фахівець може звертатися до додатку та управляти якістю доставки та обслуговування, включаючи інженерів та адміністраторів, які не були спеціально навчені застосування технологіях чи архітектурі додатків. Це небезпечно.

Тепер, коли зрозуміло проблеми пов'язані з контролем та забезпеченням якості мікросервісів, треба визначити стратегії доставки програмного забезпечення. Всі стратегії та можливості для ефективне управління якістю обслуговування в середовищі мікросервісів, показують, що основна увага приділяється автоматизації робочих процесів та використанню інструментів для досягнення результатів в масштабах, які поодинці люди не можуть здійснити. Для цього треба застосовувати інтелектуальний аналіз, машинне навчання та автоматизацію для успішного надання послуг та якісного керування мікросервісами.

В першу чергу треба використання інтелектуального аналізу мікросервісів за такими напрямками: відокремлювати інциденти від шуму; розуміти інциденти та залежності; легко обмінюйтеся інформацією всередині команди розробників; забезпечення видимості у режимі реального часу та історична видимість; мінімізувати ручне налаштування; досягти постійного порозуміння.

Відокремлювання інцидентів від шуму. Оскільки середовище мікросервісів включає так багато рухомих частин і високій обсяг діяльності, важливо мати здатність швидко розділяти інциденти, які вимагають негайного втручання з боку оперативного шум, використовуючи інтелектуальний аналіз, вбудовані інструменти. *Інцидент* – це будь-яка подія чи проблема, яка негативно впливає на якість сервіс чи мікросервіс. Інциденти трапляються коли служба або мікросервіс не можуть бути доставлені протягом періоду часу, прийнятного для організації. *Шум* - це дані, що генерує нормальна діяльність. Шум не пов'язаний з проблемою якісті обслуговування. Поділ пригоди через шум означає можливість поглянути на програму та послугу через інформаційні вікна, щоб розпізнавати, які типи даних пов'язані з потенційною проблемою, наприклад мікросервісу не вистачає обчислювальної потужності ресурси та які

типи даних просто природні, нічим не примітної події. З тих же причин дані, що були отримані з будь-якого моніторингу даних, повинні негайно перекладені у дію. Звідси програмне забезпечення для моніторингу та для користувачів повинні мати можливість швидко використовувати інформацію моніторингу, щоб зрозуміти та усунути проблему. Щоб досягти цієї мети, слід шукати інструменти, що виходять за рамки збирання даних з використанням інтелектуального аналізу перетворити дані на корисну інформацію використовуючи переваги машинного навчання та автоматизація. Таким чином відокремлювання інцидентів від шуму можливе за рахунок застосовування інтелектуального аналізу в купе з машинним навчання та автоматизацію для успішного надання послуг та якісного керування мікросервісами. Такі інструменти також мають бути здатний виявляти аномалії та фіксувати залежності сервісів автоматично, що дозволяє розробникам більш ефективне проводить свій час для вирішення реальної проблеми, а не витрачати час на інтерпретація даних вручну.

Розуміти інциденти та залежності. Розуміння того, як взаємодіють послуги та як проблема з одним компонентом може вплинути інші важливі, тому що в мінливому середовище мікросервісів, що складається з багато рівнів інфраструктури, джерело інциденту може лежати в багатьох різних місця - на хост-сервері, в проміжному програмному забезпеченні, у коді програми, у контейнері, у мережу тощо.

Легко обмінюйтеся інформацією всередині команди розробників. У сучасних командах розробників DevOps надання доступу до моніторингу продуктивності в режимі реального часу і до даних спостереження може створити більше дружний та чуйний колектив внаслідок безперешкодного обміну інформацією по всій організації. Наприклад, спостережуваність та інструменти керування продуктивністю можуть забезпечити видимість всього стека додатків для всіх членів команди, що дозволяє безперервно зробити зворотній зв'язок для усунення проблеми.

Забезпечення видимості у режимі реального часу та історична видимість. Важливо вміти виявляти та розуміти проблеми в режимі реального часу в міру їх прояви. Реакція у реальному часі потребує здібності миттєво відокремлювати інциденти по шуму, визначити корінь проблеми сервісу та використовувати якість обслуговування та дані про продуктивність для негайної реакції на проблему. Не менш важливо мати можливість зсув у часі, тобто крок назад у часі на перегляд історичних даних та залежності на певний момент у минулому. Тимчасовий зсув дозволяє зрозуміти, як інцидент можливо, розвивалися з часом і які початкові умови викликали його, досліджуючи інформацію, таку як початковий макет та структура програми або попередніх розподіл контейнерів по хостах. Інструмент повинен підтримувати будь які типи сценаріїв. Ці знання особливо цінні при спробі досягти постійного розуміння справжні та минулі події.

Мінімізуйте ручне налаштування. У гипермасштабируемом середовищі інструменти, які автоматичне налаштовують моніторинг може виникати велика різниця у загальній ефективності розгортання. Причина в тому, що у мікросервісному середовище, оновлення інформації або додавання послуг вручну займає багато часу через велику кількість компонентів в середовищі мікросервісів та складні залежності, які можуть виникнути через мікросервісну архітектуру. Намагатися налаштувати інструменти для середовища вручну підриває здатність бути гнучкими та масштабованими і, отже, це суперечить більшій частині мети впровадження мікросервісної архітектури. З цієї причини стратегія управління якістю обслуговування повинна використовувати інструменти, які здатні автоматизувати налаштування та виявлення сервісів, як якомога більше. Розгортання нового коду, нові програми, нові послуги, нові сервери і так далі в середовищі має бути автоматичними. Як уже зазначалося, автоматизоване виявлення аномалій, що здійснюється за допомогою машинного навчання та картографування залежно від сервісів також допомагає мінімізувати обсяг ручного налаштування, яке потребує робочого процесу моніторингу. Коли інструменти

моніторингу можуть виявляти аномалії та залежності автоматично, команде не потрібно витрачати час на налаштування інструменти, що дозволяють зосередитись на цих точках інтересу.

Досягти постійного розуміння. Постійне розуміння – це процес розуміння даних, що отримані завдяки спостереження, моніторингу продуктивності додатків. Дійсно, завдяки постійним змінам у високо динамічному мікросервісному середовищі, продуктивність інформація про якість обслуговування та тенденції є надзвичайно важливими. Складність у тому, що в деяких випадках, дані можуть перестати бути актуальними за кілька хвилин після того, як їх отримали. Ось чому безперервне розуміння може принести величезну користь при реалізації робочих процесів для спостереження, моніторингу продуктивності додатків (APM) та управління якістю обслуговування робочі процеси рішень. Постійному розумінню сприяє постійне автоматичне повторне виявлення даних про систему та моніторинг її конфігурації, середовища дані та залежності. Автоматизація та інтелектуальний аналіз підвищує безперервність розуміння, тому що автоматизовані процеси може допомогти отримати уявлення про мікросервіси.

Висновки

Таким чином, розглянута проблема грамотного сервіс-дизайну мікросервісної архітектури. Показано, що проблема грамотного сервіс-дизайну мікросервісної архітектури вирішується за рахунок створення балансу між деталізацією сервісів та їх функціональністю, щоб досягти максимальної продуктивності та функціональності системи. Вирішення цього завдання можливе в першу чергу за рахунок опису та розуміння принципів побудови та застосування мікросервісів в хмарних обчислення в умовах контейнерної віртуалізації.

Що стосується перспективи подальших досліджень, то доцільно продовження досліджень щодо методів підвищення якості підготовки фахівців, які займаються сервіс-дизайном мікросервісної архітектури в хмарних обчислення в умовах контейнерної віртуалізації в наступних напрямках: а) розглянути можливості застосування мікросервісної архітектури в хмарних обчислення в умовах контейнерної віртуалізації; б) вивчати які складності створюють мікросервіси; в) розробляти методики проведення грамотного сервіс-дизайну мікросервісної архітектури.

Список використаної літератури:

1. Seven main problems of implementing a microservice architecture / [електронний ресурс] — режим доступу: https://levelup.gitconnected.com/seven-main-problems-of-implementing-a-microservice-architecture-481433ddd005

2. Containerized Microservices Orchestration and Provisioning in Cloud Computing: A Conceptual Framework and Future Perspectives/ [електронний ресурс] — режим доступу: https://www.mdpi.com/2076-3417/12/12/5793

3. Selection Mechanism of Micro-Services Orchestration Vs. Choreography/ [електронний ресурс] — режим доступу: https://www.researchgate.net/publication/330960463_Selection_Mechanism_of_Micro-Services_Orchestration_Vs_Choreography

4. N.M. Josuttis, SOA in Practice: The Art of Distributed System Design, O'Reilly, 2007.

5. Containerized Microservices/ [електронний ресурс] — режим доступу: https://learn.microsoft.com/en-us/xamarin/xamarin-forms/enterprise-application-patterns/containerized-microservices

6. Катков Ю. І., Кладько І. М. УМОВИ ЗАХИСТУ МІКРОСЕРВІСІВ В ХМАРНИХ ОБЧИСЛЕННЯХ В УМОВАХ КОНТЕЙНЕРНОЇ ВІРТУАЛІЗАЦІЇ // Науково-практична конференція «АКТУАЛЬНІ ПРОБЛЕМИ КІБЕРБЕЗПЕКИ» Збірник тез. – К.: ДУІКТ, 2023. 27

жовтня 2023, С-140-143. https://duikt.edu.ua/uploads/p_2626_52007398.pdf (date of access 24.12.2023)

7. Microservices monitoring / [електронний ресурс] — режим доступу: https://www.ibm.com/products/instana/microservices-monitoring?utm_content=SRCWW&p1=Search&p4=43700076214681427&p5=p&gclid=CjwKCAiAkp6tBhB5EiwANTCx1NLoJrIZ1cuk2kbkxHvHcj_ktO6e6zIilw508s3xDFsqO8dHsmoozhoCh-IQAvD_BwE&gclsrc=aw.ds

8. Continuous delivery - definition & overview/ [електронний ресурс] — режим доступу: https://www.sumologic.com/glossary/continuous-delivery/

9. Microservice observability/ [електронний ресурс] — режим доступу: https://www.ibm.com/downloads/cas/XWG4NDPN

10. Офіційна документація Docker // [Електронний ресурс] Режим доступу до ресурсу: https://docs.docker.com/

11. Офіційна документація Kubernetes // [Електронний ресурс] Режим доступу до ресурсу: https://kubernetes.io/docs/home/

12. J. Grogan et al., "A multivocal literature review of function-as-a-service (FaaS) infrastructures and implications for software developers" in Systems Software and Services Process Improvement, Cham, Switzerland:Springer, pp. 58-75, 2020, [online] Available: https://doi.org/10.1007/978–3-030-56441-4_5.

Автори статті

Катков Юрій - доктор технічних наук, доцент, Державний університет інформаційно-комунікаційних технологій, Київ, Україна.

Зінченко Ольга - доктор технічних наук, доцент, Державний університет інформаційно-комунікаційних технологій, Київ, Україна.

Березовська Юлія - PhD, Державний університет інформаційно-комунікаційних технологій, Київ, Україна.

Кладько Іван - студент, Державний університет інформаційно-телекомунікаційних технологій, Київ, Україна.

Authors of the article

Katkov Yuriy - Doctor of Science (technic), Associate Professor, State University of Information and Communication Technologies, Kyiv, Ukraine.

Zinchenko Olha - Doctor of Science (technic), Associate Professor, State University of Information and Communication Technologies, Kyiv, Ukraine.

Berezovska Yuliia - PhD, State University of Information and Communication Technologies, Kyiv, Ukraine.

Kladko Ivan - student, State University of Information and Communication Technologies, Kyiv, Ukraine.