УДК 004.41:004.75

МЕТОД РЕКОНСТРУКЦІЇ ДОДАТКІВ ДЛЯ СТВОРЕННЯ КОНТЕЙНЕРІВ РОЗПОДІЛЕНОЇ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

О. М. Купріянов, В. С. Ситников
Національний університет «Одеська політехніка»

Анотація. *Мікросервіси – це невеликі, автономні сервіси, які можна розробляти окремо, причому кожен мікросервіс обробляє свою частину логіки без взаємодії з іншими. Вони легко розгортаються в контейнерах, а технології оркестрації контейнерів допомагають керувати їхнім розподілом та оновленням. З цієї причини багато компаній переходять від монолітних архітектур до мікросервісів. У цій статті розглядається метод трансформації монолітного додатку в мікросервісну архітектуру розподіленої системи на основі контейнерів за прикладом телекомунікаційної компанії. Також пропонується метод створення шаблонних сценаріїв для розгортання, щоб підтримувати та поширювати мікросервіси в контейнерному середовищі. Результати дослідження підтвердили ефективність запропонованого підходу. Оцінка ефективності часу виконання для викликів API на різних етапах розробки підтвердила переваги мікросервісної архітектури. Висновки демонструють, що мікросервіси, створені за запропонованим методом, мають вищу можливість повторного використання порівняно з тими, що створені традиційними методами.*

Ключові слова*: контейнер, мікросервіс, API, масштабованість, розподілена система*

Вступ

Мікросервіси є основним концептом безсерверних обчислень і являють собою невеликі незалежні служби. Програми, що складаються з мікросервісів, характеризуються слабким зв'язком між окремими компонентами, що дозволяє оновлювати їх без необхідності змінювати внутрішню структуру всього додатка. Така архітектура швидше розробляється та розгортається порівняно з монолітною структурою, і її компоненти можуть використовуватися повторно [1].

Нові технології оркестрації контейнерів, такі як Kubernetes [2], Docker Swarm [3] і Mesospher [4], спрощують управління контейнеризованими мікросервісами. Вони забезпечують автоматичне створення контейнерів і управління великими додатками, дозволяючи ефективно керувати розподілом численних контейнерів. Завдяки перевагам мікросервісів і контейнерної оркестрації багато компаній переходять від монолітних архітектур до мікросервісів, розробляючи та розгортаючи свої додатки на основі контейнерів [5].

1. Постановказадачі

Основною метою дослідження є створення гнучкої та масштабованої розподіленої системи для української телекомунікаційної компанії, яка дасть змогу виявляти аварії, обробляти трафік та легко адаптуватися до розширення сервісів. Це дозволить скоротити час впровадження нових функцій та підвищити загальну ефективність компанії, що покращить якість мобільного сигналу та сприятиме розвитку інформаційного суспільства й економічному зростанню України, що в свою чергу закріплено у Законі України №497/2019 "Про деякі заходи з покращення доступу до мобільного Інтернету".

Наразі перехід від монолітних додатків до мікросервісів залишається недостатньо вивченим. При плануванні реконструкції складного монолітного додатка на мікросервіси необхідно враховувати такі аспекти, як розмір мікросервісу, конфігурація API, обробка баз даних та забезпечення безпеки [6]. Однією з проблем є те, що розмір мікросервісу не є фіксованим, що ускладнює його зміну. Крім того, існує потреба вручну визначати параметри для управління контейнерами, такі як початковий номер керування контейнером і конфігурація мережі [7].

Перетворення монолітних додатків на мікросервіси вимагає глибокого розуміння їхньої структури. Для цього були запропоновані методи, що базуються на архітектурі програмного забезпечення або потоках даних [8]. Також використовується підхід до проектування та розробки додатків на основі UML (Unified Modeling Language), що вимагає перетворення існуючих UML-даних у мікросервіси.

Щоб вирішити ці проблеми, у цій статті представлено метод аналізу даних проектування монолітних додатків та їхньої реконструкції у мікросервіси на основі контейнерів для телекомунікаційної компанії з метою покращення зв'язку. Метод класифікує UML-дані монолітного додатка за ієрархією, будує граф, який перетворюється на мікросервіси, і створює мікросервіси для кожної сутності.

Цей метод автоматизованого створення шаблонів контейнерів дозволить зменшити час, необхідний для розгортання та управління мікросервісними системами в середовищі оркестрації Kubernetes, а також підвищить ефективність та точність процесів розгортання і конфігурації, зменшуючи ймовірність помилок і спрощуючи управління ресурсами.

2. Аналіз існуючих систем

Розподілена мікросервісна система повинна бути створена з урахуванням стійкості до відмов. Це означає, що вона повинна бути здатна обробляти відмови певних частин або вузлів без зниження продуктивності чи доступності всієї системи. Структуру розподілених програмованих адаптивних систем викладено на Рис.1.

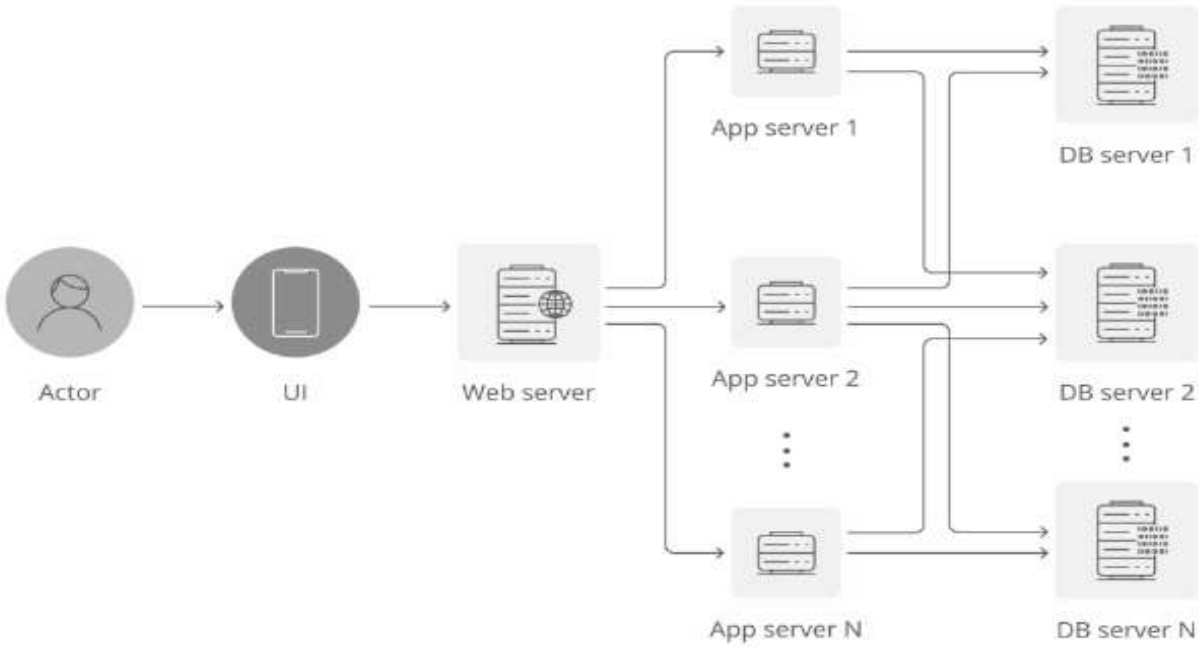

Рис. 1. – Структура розподілених програмованих адаптивних систем

Розподілені системи використовують стратегії резервування, реплікації або розділення для досягнення стійкості до відмов. Мікросервісна архітектура Рис. 2 розділяє додаток на більш дрібні сервіси та повністю незалежні компоненти, що забезпечує їм більшу гнучкість і масштабованість. Це логічна еволюція сервіс-орієнтованої архітектури (SOA), яка відповідає сучасним бізнес-процесам і вимогам [9]. Мікросервіси вирішують проблеми застарілих монолітних систем.

Цей тип архітектури складається з великої кількості невеликих сервісів, кожен з яких виконує свої власні функції і може бути незалежно розгорнутий.

Такий сервіс простіший для розуміння та розробки, що, в свою чергу, надає можливість безперервного постачання і поліпшення продукту [10].

За допомогою Docker можно створювати контейнеризовані програми, які включають всі залежності (бібліотеки, конфігурація) необхідні для роботи. Це забезпечує стандартизоване середовище виконання для обробки даних незалежно від базової операційної системи вузлів кластера [11]. Гарантує, що код обробки даних працюватиме однаково на всіх вузлах, що спрощує розробку та тестування.

Сервіси обмінюються даними через легкі механізми, такі як HTTP протокол, і кожен сервіс працює незалежно в своєму власному процесі. Іншими словами, в архітектурі мікросервісів вся функціональність розділена на незалежні модулі, які взаємодіють один з одним

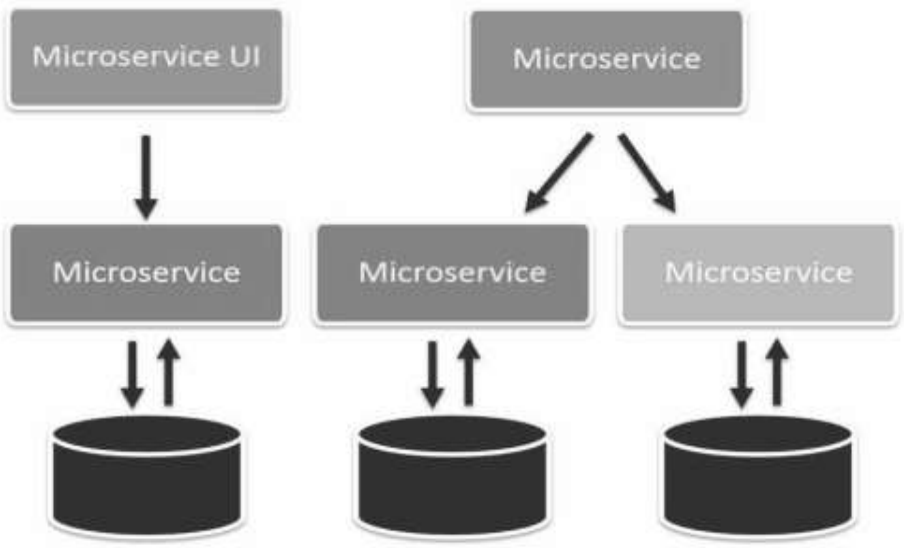

Рис. 2. – Мікросервісна архітектура

через API (інтерфейси прикладного програмування). Кожен сервіс охоплює свою власну область і може оновлюватися, розгортатися і масштабуватися незалежно.

Мікросервісна архітектура виявляється надзвичайно ефективною з кількох причин. По-перше, вона забезпечує високий рівень відмовостійкості, оскільки навіть при виникненні помилки в одному сервісі, інші функціонують без перебоїв. Це робить процес змін та експериментів менш ризикованим та зменшує кількість помилок. Далі, мікросервісна архітектура відзначається великою гнучкістю у виборі технологій. Інженерні команди не зв'язані з технологічними обмеженнями, тому можуть вільно використовувати різні рішення та структури для кожного сервісу. Це сприяє швидкому інноваційному розвитку без обмежень, які часто виявляються в монолітних системах.

Крім того, мікросервіси спрощують управління та розуміння системи. Розбивка додатку на менші та прості компоненти полегшує аналіз та контроль кожного сервісу окремо. Такий підхід дозволяє концентруватися на конкретних функціях та сервісах, забезпечуючи велику гнучкість та швидкість реакції на зміни. Помилка в одному мікросервісі не поширюється на інші, що забезпечуєстабільність системи та уникнення серйозних перебоїв у роботі. Крім того, додавання нових функцій в мікросервісну систему виявляється набагато простішим, що призводить до кращої масштабованості. Кожен елемент може масштабуватися незалежно, що забезпечує ефективне використання ресурсів та економію часу та коштів.

Мікросервісна архітектура забезпечує високу гнучкість і адаптивність, що дозволяє легко інтегрувати нові функції та технології без зупинки системи. Цей підхід зменшує ризики зупинки окремих сервісів і покращує загальну продуктивність системи.

Масштабованість та гнучкість є в тому, що легко запускати та зупиняти контейнеризовані служби ОБРОБКИ даних. Дозволяє горизонтальне масштабування додаванням нових вузлів до кластера та запуском на них додаткових контейнерів. Це дозволяє швидко реагувати на зміни обсягу даних та обчислювальних потреб.

Ізоляція та безпека полягає в тому, що Docker забезпечує ізоляцію контейнерів один від одного. Це запобігає конфліктам між залежностями програм та підвищує безпеку обробки даних. Контейнери мають обмежені ресурси (CPU, пам'ять), що запобігає впливу одного контейнера на інші.

Так, мікросервісна архітектура може допомогти зробити розподілену обробку даних більш гнучкою та масштабованою, яка буде більш стійкою до відмов та швидкого відновлення. А розподілена обробка даних може допомогти зробити мікросервісну архітектуру більш продуктивною та більш доступною.

Використання сервісів як окремих компонентів у мікросервісній архітектурі вимагає особливого підходу до проектування

додатків, оскільки можливі відмови окремих сервісів можуть вплинути на загальну функціональність системи. Кожен запит до сервісу може зазнати невдачі через його недоступність, тому необхідно забезпечити, щоб система могла ефективно реагувати на такі ситуації. Це ускладнює розробку порівняно з монолітними системами.

При розробці мікросервісів слід враховувати, як відмови окремих сервісів можуть вплинути на загальну роботу системи. Для перевірки відмовостійкості потрібно штучно симулювати збої, щоб оцінити, як система справляється з відмовами і як ефективно працюють механізми моніторингу. Важливо забезпечити швидке виявлення і, по можливості, автоматичне відновлення роботи сервісів. Це може включати використання інструментів для моніторингу, автоматичного масштабування та стратегій відновлення після збоїв.

Розміщення компонентів у сервісах додає можливість більш точного планування релізу системи. З монолітом будь-які зміни вимагають повторної збірки і розгортання всієї програми. У разі мікросервісної архітектури потрібно розгорнути тільки ті сервіси, які змінилися. Процес розгортання мікросервісів зображено на Рис. 3.

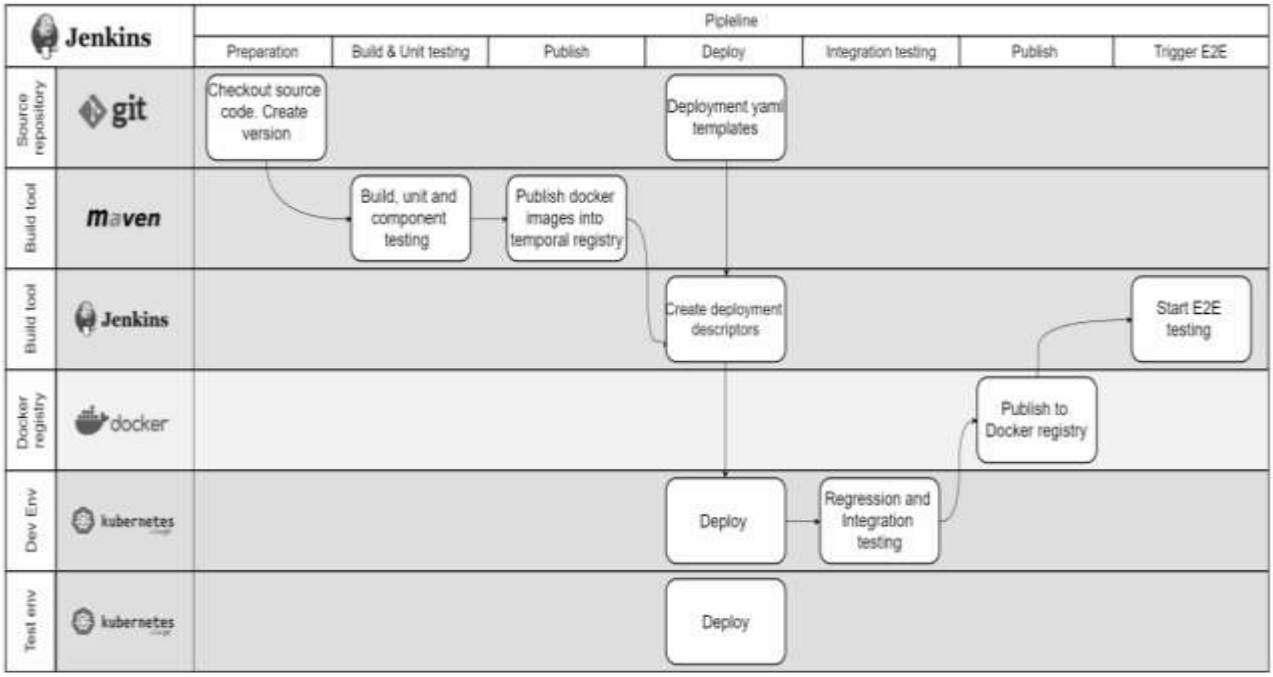

Рис. 3. – Процес розгортання мікросервісів

Це дозволяє спростити і прискорити процес релізу або швидкого розгортання додаткового локального середовища для розробників, яке не буде відрізнятися від середовища на сервері. Існують деякі недоліки мікросервісного підходу на Docker, які варто врахувати:

- Складність конфігурації мережі: Керування мережею у середовищі з великою кількістю контейнерів може бути складним завданням. Докер пропонує кілька різних методів мережевої конфігурації, і вибір найбільш підходящого може бути викликом.

- Складність моніторингу та налагодження: З великою кількістю мікросервісів у роботі стає важко відстежувати та моніторити їх стан. Розгортання, налагодження та відлагодження також можуть виявитися складними завданнями через розподілений характер архітектури.

- Збільшення складності управління: Керування кількома контейнерами, що працюють на різних сервісах, може призвести до збільшення складності управління і розгортанням.

- Проблеми з безпекою: З ростом числа контейнерів збільшується ймовірність вразливостей безпеки. Недоліки в конфігурації контейнерів або служб можуть викликати проблеми безпеки.

Але загалом, мікросервісний підхід на Docker в порівнянні з монолітною системою, дозволяє компаніям швидко реагувати на зміни [8], забезпечуючи більшу гнучкість, надійність та ефективність у розробці та впровадженні

програмного забезпечення, що в свою чергу перекриває всі незначні недоліки.

Мікросервіси повинні складатися з одиниць, які виконують одну бізнес-функцію [12]. Якщо бізнес-функція занадто велика, її слід розділити на декілька менших одиниць. У мікросервісах база даних повинна бути розподіленого типу, а не централізованою. У даній статті визначено бізнес-функцію як одиницю бази даних.

В дослідженнях про відмінності мікросервісів та порівнюючи їх з існуючими концепціями сервіс-орієнтованої архітектури (SOA) зазначено, що мікросервіси є більш автономними, ніж SOA, і що розмір мікросервісів повинен відповідати обсягу роботи, яку може виконати одна команда розробників. Також показано, що зв'язок між мікросервісами є мінімальним, а їхні інтерфейси дуже абстрактні.

У статті концепція інтерфейсу використовується як метод побудови, і він визначений для підтримки зв'язку між користувачем і мікросервісами [13].

В моделі еталонної архітектури для побудови мікросервісного середовища в корпоративному контексті виділяють три типи мікросервісів у своїй архітектурній моделі [14]. Інтерфейс складається з користувацького введення, бізнес-логіки для обробки цього введення та рівня збереження, який містить дані. Тому мікросервіси класифіковані на рівні презентації, бізнес-логіки та збереження даних, використовуючи концепцію трьох рівнів, представлену в цьому дослідженні. Однак, коли застосували цей метод, повністю незалежні мікросервіси не були налаштовані, оскільки дані можна пов'язувати з іншими функціями.

3. Метод реконструкції мікросервісу на основі контейнерів

Процес перетворення монолітних проектних даних у мікросервіси зображено на Рис. 4.

Для здійснення реконструкції проекту у вигляді мікросервісів, зображеної на Рис. 5, виконується ретельний аналіз та дотримуються чотири етапи:

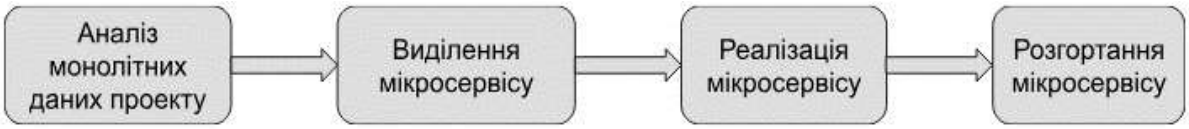

Рис. 4. – Процес розгортання мікросервісів

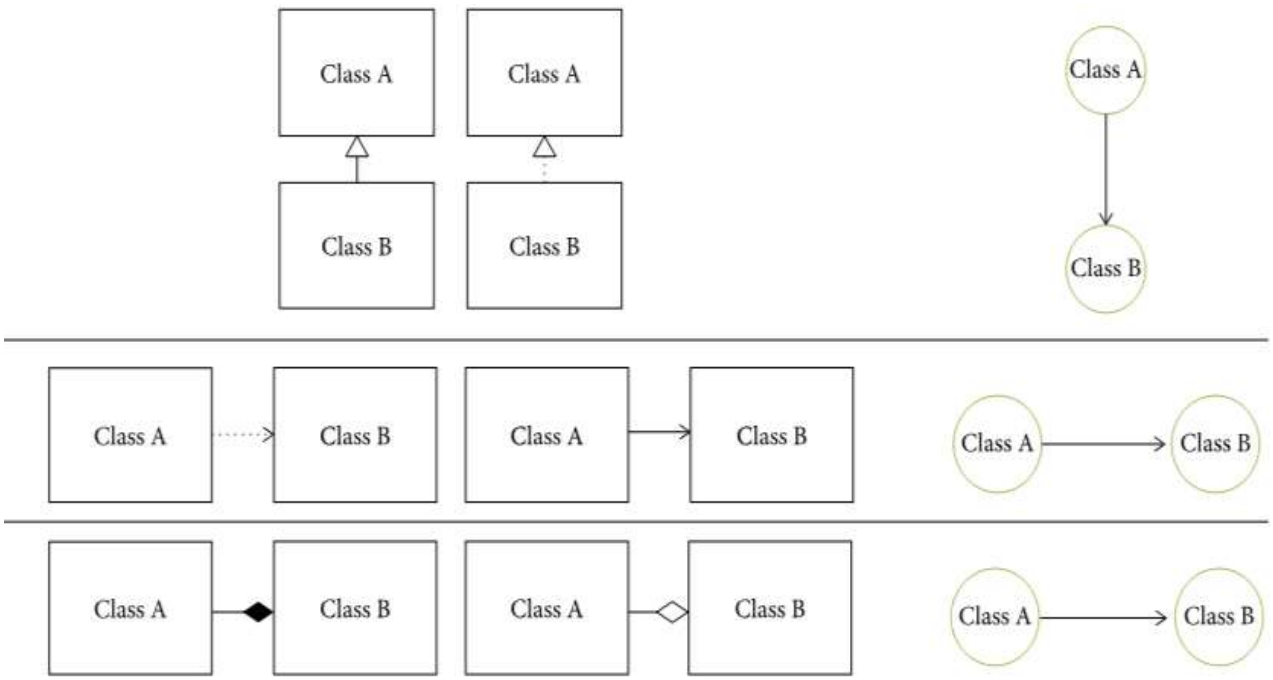

Рис. 5. – Підхід до реконструкції додатків для розробки мікросервісів на основі контейнерів

- Аналіз даних монолітного проекту. На цьому етапі збираються та аналізуються дані монолітного проекту, який планується реконструювати в мікросервіси. Він включає наступні заходи:

- Збір даних проектування моноліту включає діаграму класів UML, визначену шаблоном ECB.

- Трирівнева класифікація: класи, визначені як стереотипи, збираються з даних дизайну класу UML і класифікуються в трьох рівневу ієрархію, що охоплює презентацію, бізнес-логіку та збереження.

- Відображення класів за типом: визначається трирівнева ієрархія та виконується ієрархічне відображення. Тип "Межа" відповідає рівню представлення, тип "Контроль" — рівню бізнес-логіки, а тип "Об'єкт" — рівню збереження.

На етапі виділення мікросервісів на основі зібраної та класифікованої інформації про класи будується граф; потім виконуються наступні дії для виділення мікросервісів одиниці сутності:

Аналіз відносин класів і побудова графа: створюється граф, що складається з вершин, які відповідають класу, і ребер, які представляють відносини виклику. Вершини, класифіковані на презентаційному рівні згідно з відносинами відповідності, визначаються як граничні вершини. Вершини, класифіковані на рівні бізнес-логіки, визначаються як керуючі вершини. Вершини, класифіковані на рівні збереження даних, визначаються як сутнісні вершини. Крім того, відносини виклику між класами в даних UML-дизайну представлені як ребра.
Узагальнення та реалізація: у відношенні до узагальнення та реалізації, клас A є батьківським класом, і він успадковується або реалізується. Тому клас A впливає на клас B і викликає його.
Залежність та асоціація: у відношенні до залежності та асоціації, клас A викликає та використовує об'єкт і метод класу B; таким чином, клас A зазнає впливу, коли клас B змінюється. Композиція та агрегація: у відношенні до композиції та агрегації, клас A включений як частина класу B; отже, якщо клас B змінюється, це впливає на клас A.

Перебудова графа, орієнтованого на бізнес-логіку: граф мікросервісного перетворення перебудовується так, щоб конфігурований граф міг бути виділений за одиницею мікросервісу. Вершини та ребра презентаційного і рівня збереження даних, які не безпосередньо пов'язані з вершинами бізнес-логіки, видаляються, і граф перебудовується.

Виділення мікросервісу на основі основної сутності: на цьому завершальному етапі виділення мікросервісу створюється одиниця мікросервісу, що містить одну сутність. Сутність - це елемент класу вершини, класифікований на рівні збереження даних. Вона відстежує вершини бізнес-логіки та презентаційного рівня, які викликаються у цій сутності, і будує їх. Однак, оскільки певний керуючий клас, що є вершиною рівня бізнес-логіки, може бути викликаний багатьма класами сутностей, область конфігурації може перекриватися. Щоб подолати цю проблему, визначається основна сутність для контролю, що усуває надмірність у конфігурації. Вона визначається шляхом обчислення відношення методу, який викликає клас сутності, та середнього значення подібності назв методів.

Якщо головна сутність викликає іншу сутність безпосередньо з точки зору керування після створення мікросервісу одиниці сутності в елементі керування визначеної сутності, дані проекту змінюються для виклику сутності через граничний клас рівня представлення. Це пов'язано з тим, що зв'язок між мікросервісами має здійснюватися через API. Це створює нову межу у зв'язку «контроль-межа-сутність», яка викликається безпосередньо та змінює дані проекту у зв'язку «контроль-межа-сутність» [15].

Якщо клас управління викликає більше однієї сутності, обчисліть співвідношення методу, який викликає сутність, і подібність назви методу сутності та визначте сутність із вищим коефіцієнтом як головну сутність. В іншому випадку клас, пов'язаний з однією сутністю, автоматично визначить головну сутність, яку буде пов'язано з сутністю. Коли головну сутність визначено, повторіть стільки, скільки сутностей, і перейдіть до конфігурації мікросервісу в одиницях сутності, що відповідають головній сутності. Сутності, для яких не визначено головну сутність, виключаються з конфігурації мікросервісу.

Нарешті, коли елемент керування підключається до сутності або класу керування іншого мікросервісу, створюється нова межа, а підключення змінюється, щоб підключити мікросервіс через створену межу.

Приклад реалізації API підключення між мікросервісами зображено на Рис. 6.

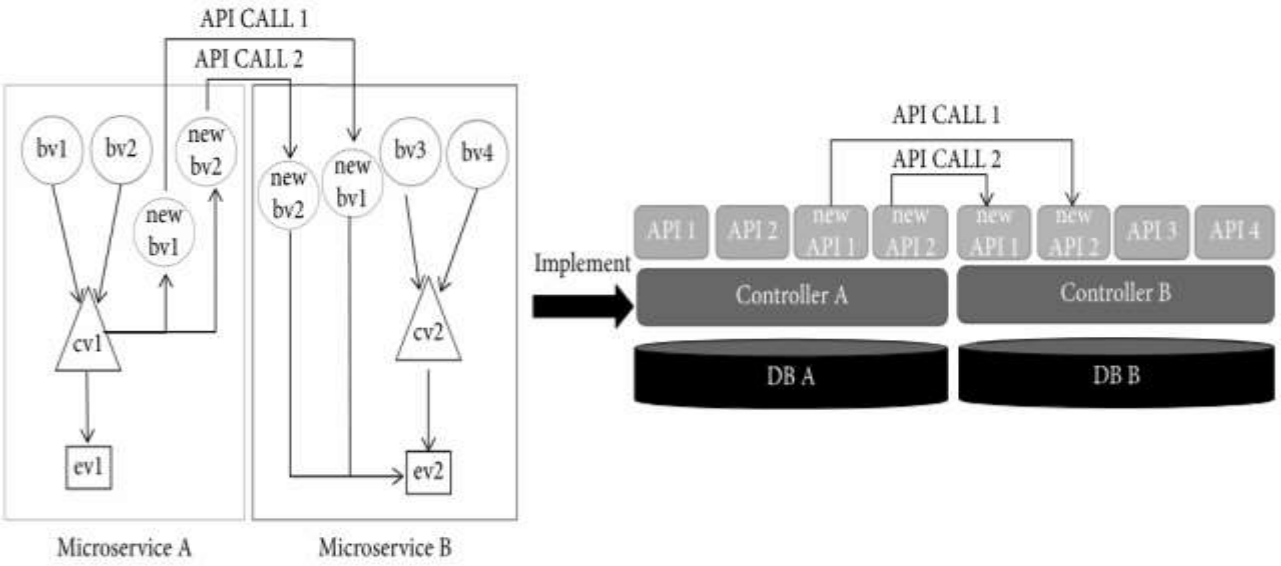

Рис. 6. – Приклад реалізації API підключення між мікросервісами

На етапі впровадження API-з'єднання між ідентифікованими мікросервісами реалізується відповідно до ієрархії:

- Реалізація мікросервісу на 3 рівнях: впровадження класів вершин, класифікованих на кожному рівні. Розробники можуть будувати на основі проектних даних, використовуючи інструменти розробки або мови для кожного рівня.

- Реалізація мікросервісу на 3 рівнях: впровадження класів вершин, класифікованих на кожному рівні.

Останнім етапом процесу реконструкції мікросервісу є розгортання мікросервісу, під час якого генерується сценарій шаблону, який підтримує розгортання та керування в середовищі оркестровки контейнера на основі реалізованих мікросервісів і даних дизайну розгортання UML. Тут дані про проект збирають дані про клас і проект розгортання, а мікросервіс посилається на мікросервіс, реалізований на етапі впровадження мікросервісу. Основними сценаріями шаблонів є Pod, ReplicaSet і Service, які є трьома основними типами сценаріїв шаблонів, які працюють на Kubernetes, платформі оркестровки контейнерів. Процес створення сценарію шаблону на основі даних дизайнувключає кілька етапів, що забезпечують узгодженість і точність.

Створення сценарію шаблону для розгортання контейнера: отримано елементи колекції з монолітних проектних даних, щоб створити елементи моделі специфікації сценарію шаблону.

Дані проекту розгортання: вони класифікуються на інформацію про вузли та інформацію про з'єднання. Інформація про вузол містить назву вузла, ресурс, ОС і програму для створення контейнера. Інформація про підключення збирає зовнішній і внутрішній порт, протокол і IP-адресу зовнішнього з'єднання для зв'язку із зовнішньою частиною контейнера.

Важливо забезпечити правильну конфігурацію всіх компонентів для уникнення можливих збоїв.Таким чином, структурований відповідним методом дослідження та запропонований спосіб, показує, що метод конфігурації мікросервісу має високу зв'язність і низький зв'язок.

На Рис. 7. порівнюється загальне середнє значення часу виконання API для кожної ітерації.

Як показано, у випадку монолітного можна побачити, що час виконання швидко зростає на 500 ітерацій і далі. І навпаки, у випадку з мікросервісами на основі методу, представленого в цьому документі, можна побачити, що продуктивність стабільна при 500 ітераціях і 1000 ітерацій.

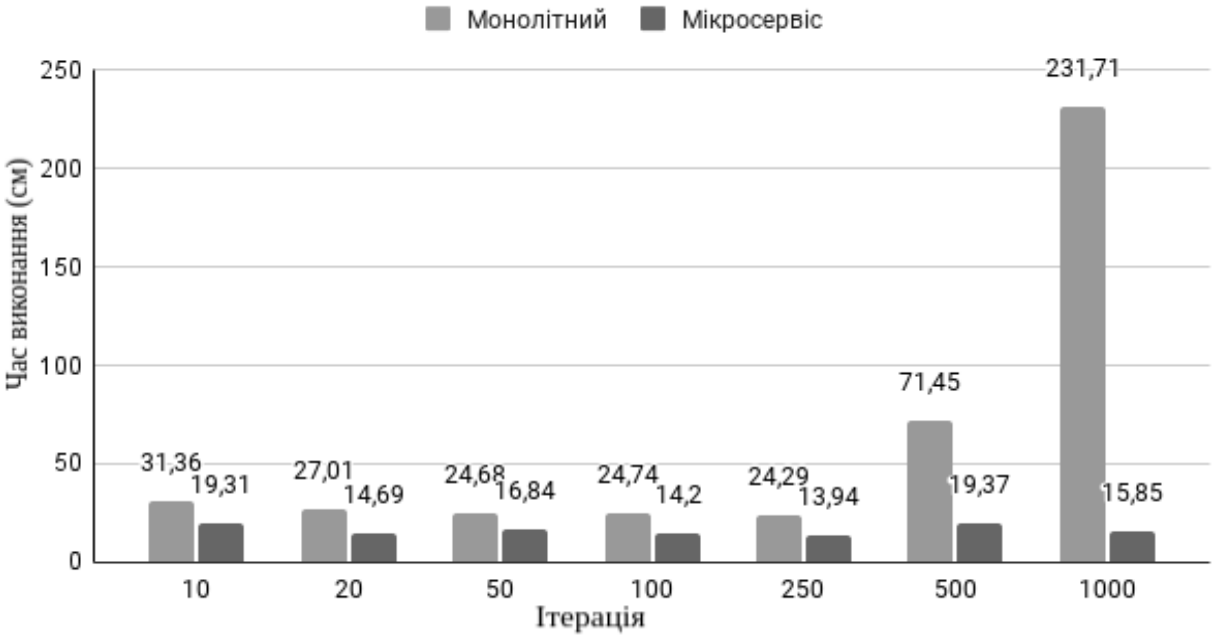

Рис. 7. - Середнє значення часу для моноліту та мікросервісу

Як показано, у випадку монолітного можна побачити, що час виконання швидко зростає на 500 ітерацій і далі. І навпаки, у випадку з мікросервісами на основі методу, представленого в цьому документі, можна побачити, що продуктивність стабільна при 500 ітераціях і 1000 ітерацій.

Висновки

У цій роботі представлено метод аналізу даних проектування монолітних додатків та їх перетворення на мікросервісні одиниці на основі контейнерів. Запропонований метод включає процеси аналізу даних проектування моноліту, вилучення мікросервісів, їх реалізації та розгортання. На етапі аналізу даних проектування моноліту збиралися та класифікувалися різні типи даних. На етапі вилучення мікросервісів граф будувався шляхом аналізу класів та асоціацій, після чого визначалися мікросервіси як одиниці сутності. На етапі реалізації мікросервісів здійснювалась їх імплементація по рівнях.

Майбутні дослідження будуть зосереджені на розробці інструменту моніторингу для управління ефективною роботою контейнерів після розподілу мікросервісів, створених за запропонованим методом, та на покращенні продуктивності API Gateway для вирішення проблеми мережевого навантаження між мікросервісами.

Проведене дослідження підтвердило ефективність запропонованого методу: реорганізовані мікросервіси успішно розгортаються та працюють у контейнерному середовищі. Порівняльна оцінка показала, що запропонований метод перевершує існуючі з точки зору можливості повторного використання.

Список використаної літератури

1. Amazon AWS Lambda, https://aws.amazon.com/lambda/?nc1=f_ls .

2. Kubernetes, https://kubernetes.io/.

3. Docker Swarm, https://docs.docker.com/engine/swarm/swarm-tutorial/ .

4. Мезосфера, https://mesosphere.com/ .

5. І. Бальдіні, П. Кастро, К. Чанг та ін., «Безсерверні обчислення: поточні тенденції та відкриті проблеми», у дослідницьких досягненнях у хмарних обчисленнях , стор. 1–20, Springer, Берлін, Німеччина, 2017 р. Переглянути на: Сайт видавця | Google Scholar.

6. C. Esposito, A. Castiglione, and K.-KR Choo, "Challenges in delivery software in cloud as microservices", IEEE Cloud Computing , vol. 3, № 2016. – № 5. – С. 10–14. Переглянути на: Сайт видавця | Google Scholar.

7. E. Casalicchio, «Автономна оркестровка контейнерів: визначення проблеми та проблеми дослідження», у матеріалах 10-ї міжнародної конференції EAI з методологій та інструментів

оцінки продуктивності , Таорміна, Італія, жовтень 2016 р. Переглянути на: Сайт видавця | Google Scholar.

8. Р. Чен, С. Лі та З. Лі, «Від моноліту до мікросервісів: підхід, керований потоком даних», у матеріалах 24-ї Азіатсько-Тихоокеанської конференції з розробки програмного забезпечення (APSEC), стор. 466–475, Нанкін, Китай, грудень 2017 року.

9. F. Rademacher, S. Sachweh, and A. Z¨undorf, "Різниці між керованою моделлю розробкою сервіс-орієнтованої та мікросервісної архітектури", у матеріалах Міжнародної конференції IEEE з семінарів з архітектури програмного забезпечення (ICSAW), стор. 38–45, Гетеборг, Швеція, квітень 2017 р.

10. N. Dragoni, S. Giallorenzo, A. L. Lafuente et al., "Micro-services: yesterday, today, and tomorrow," in Present and Ulterior Software Engineering, pp. 195–216, Springer, Berlin, Germany, 2017.

11. Docker, https://www.docker.com/.

12. Ісіль Карабей Аксакалли «Керована модель архітектури для автоматизованого розгортання мікросервісів» у докладі, який знаходиться за посиланням: https://www.mdpi.com/2076-3417/11/20/9617.

13. Антони Бучегонни «Відмінності мікросервісів» у виданні https://books.google.com.ua/books?id=WqjDDwAAQBAJ&printsec=copyright&redir_esc=y#v=onepage&q&f=false.

14. Y. Yu, H. Silveira, and M. Sundaram, "A microservice based reference architecture model in the context of enterprisearchitecture," in Proceedings of the IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference, стор. 1856 –1860, Сіань, Китай, жовтень 2016.

15. Електроннакнига Microservice–Architecture: Creating Composite UIBased on Microservices, https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/architect-microservice-container-applications/microservice-based-composite-ui - форма-макет.

References list

1. Amazon AWS Lambda, https://aws.amazon.com/lambda/?nc1=f_ls.

2. Kubernetes, https://kubernetes.io/.

3. Docker Swarm, https://docs.docker.com/engine/swarm/swarm-tutorial/ .

4. Mesosphere, https://mesosphere.com/.

5. I. Baldini, P. Castro, K. Chang et al., "Serverless computing: current trends and open problems," in *Research Advances in Cloud Computing*, pp. 1–20, Springer, Berlin, Germany, 2017.
View at: Publisher Site | Google Scholar.

6. C. Esposito, A. Castiglione, and K.-K. R. Choo, "Challenges in delivering software in the cloud as microservices," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 10–14, 2016.
View at: Publisher Site | Google Scholar.

7. E. Casalicchio, "Autonomic orchestration of containers: problem definition and research challenges," in *Proceedings of the 10th EAI International Conference on Performance Evaluation Methodologies and Tools*, Taormina, Italy, October 2016.
View at: Publisher Site | Google Schola.

8. R. Chen, S. Li, and Z. Li, "From monolith to microservices: adataflow-driven approach,," in Proceedings of the 24th Asia-Pacific Software Engineering Conference(APSEC), pp. 466–475, Nanjing, China, December 2017.

9. F. Rademacher, S. Sachweh, and A. Z ¨undorf, "Differencesbetween model-driven development of service-oriented andmicroservice architecture," in Proceedings of the IEEE International Conference on Software Architecture Workshops(ICSAW), pp. 38–45, Gothenburg, Sweden, April 2017.

10. N. Dragoni, S. Giallorenzo, A. L. Lafuente et al., "Micro-services: yesterday, today, and tomorrow," in Present andUlterior Software Engineering, pp. 195–216, Springer, Berlin,Germany, 2017.

11. Docker, https://www.docker.com/.

12. Isil Karabey Aksakalla "Managed architecture model for automated deployment of microservices" in the report, which can be found at the link: https://www.mdpi.com/2076-3417/11/20/9617.

13. Antony Buchegonna "Distinctions of microservices" in the edition https://books.google.com.ua/books?id=WqjDDwAAQBAJ&printsec=copyright&redir_esc=y#v=onepage&q&f=false.

14. Y. Yu, H. Silveira, and M. Sundaram, "A microservice basedreference architecture model in the context of enterprisearchitecture," in Proceedings of the IEEE Advanced Infor-mation Management, Communicates, Electronic and Auto-mation Control Conference, pp. 1856–1860, Xi'an, China,October 2016.

15. Microservice–Architecture e-book: Creating Composite UIBased on Microservices, https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/architect-microservice-container-applications/microservice-based-composite-ui-shape-layout.

APPLICATION RECONSTRUCTION METHOD FOR CREATING CONTAINERS OF DISTRIBUTED MICROSERVICE ARCHITECTURE AND TRANSACTIONS BETWEEN THEM

O. M. Kupriyanov, V. S. Sytnikov
Odesa Polytechnic National University

Abstract. *Microservices are small, autonomous services. An application consisting of such microservices can be developed separately, with each microservice processing its own part of the logic without interacting with the others. Microservices are easy to deploy in containers, and container orchestration technologies help manage their distribution and updates. Therefore, many companies are moving from monolithic architectures to microservices. This article describes a method for converting a monolithic application into a microservice architecture of a container-based distributed system. Microservices are defined through data analysis of a monolithic application. It also provides a method for creating templated deployment scripts to support and distribute microservices in a containerized environment. The results of the study confirmed the effectiveness of the proposed approach. A runtime performance evaluation for API calls at various stages of development was conducted, which confirmed the benefits of a microservice architecture. The findings show that microservices created by the proposed method have higher reusability compared to those created by traditional methods.*

Keywords: *container, microservice, API, scalability, distributed system*

Отримано 16.09.2024

Купріянов Олександр Миколайович, Національний університет «Одеська політехніка», здобувач освітньо-наукового рівня доктора філософії. Проспект Шевченка, 1, Одеса, 65044, Україна,
E-mail: kupriyanov.a062@gmail.com, тел.: +380995633790

Oleksandr Kupriyanov, Odesa Polytechnic National University, recipient of the educational and scientific level of Doctor of Philosophy. Shevchenko Avenue, 1, Odesa, 65044, Ukraine,
E-mail: kupriyanov.a062@gmail.com, ph: +380995633790.

ORCID: https://orcid.org/0009-0003-8155-1321

Ситников Валерій Степанович, Національний університет «Одеська політехніка», завідувач кафедри комп'ютерних систем, д.т.н., проф. Просп. Шевченка, 1, Одеса, Україна,
E-mail: sitnikov@op.edu.ua, тел. +38-067-4567165

Sytnikov Valery Stepanovich, Odesa Polytechnic National University, Computer System Department Head, Dr. Eng. Sci., Prof., Shevchenko ave., 1, Odessa, Ukraine, , E-mail: sitnikov@op.edu.ua, ph: +38-067-4567165

ORCID: https://orcid.org/0000-0003-3229-5096

Web of Science ResearcherID: 57190377358