НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
ННК «Інститут прикладного системного аналізу»
(повна назва інституту/факультету)

Кафедра системного проектування
(повна назва кафедри)

«На правах рукопису»
УДК _004.852

«До захисту допущено»

Завідувач кафедри
__________ _А.І. Петренко
(підпис) (ініціали, прізвище)

“___”_____________20__ р.

Магістерська дисертація

зі спеціальності (спеціалізації)__122 – Інформаційні системи і технології проектування (Системне проектування)_________
(код і назва спеціальності)

на тему: Мікросервісна архітектура у системах керування потоками даних
____

Виконав: студент _6_ курсу, групи ДА-61м
(шифр групи)

_________________ Сарапулов Віктор Сергійович _______________ ________
(прізвище, ім’я, по батькові) (підпис)

Науковий керівник _________ к.т.н., доц., Харченко Костянтин Васильович _
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант Розроблення стартап-проекту к.т.н., доц., Харченко Костянтин Васильович
(назва розділу) (науковий ступінь, вчене звання, , прізвище, ініціали) (підпис)

Рецензент ____ _________
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Нормконтроль к.т.н, доц., Кисельов Генадій Дмитрович_____ _________
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент ______________
(підпис)

Київ – 2018 року

Національний технічний університет України
«Київський політехнічний інститут
імені Ігоря Сікорського»

Інститут/факультет «Інститут прикладного системного аналізу»
(повна назва)

Кафедра ___________________Системного проектування _______________
(повна назва)

Рівень вищої освіти – другий (магістерський) за освітньо-професійною (освітньо-науковою) програмою

Спеціальність (спеціалізація) _8.05010103 Системне проектування__
(код і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри
__________ _А.І. Петренко
(підпис) (ініціали, прізвище)
«___»_____________20__ р.

ЗАВДАННЯ
на магістерську дисертацію студенту
Сарапулову Віктору Сергійовичу
(прізвище, ім'я, по батькові)

1. Тема дисертації «Мікросервісна архітектура у системах керування потоками даних»
науковий керівник дисертації Харченко К.В к.т.н., доц.,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «_27_»_березня__ 2018 р. №_____

2. Строк подання студентом дисертації ______________________________

3. Об'єкт дослідження __ системи керування потоками даних у контексті мікросервісної архітектури

4. Предмет дослідження побудова мікросервісної архітектури на основі систем керування потоками даних

5. Перелік завдань, які потрібно розробити дослідити системи керування потоками даних, визначити їх переваги та недоліки у порівнянні з системами фон Неймана,

розробити власну мікросервісну архітектуру, розробити стратегію по розширенню розробленої архітектури, представити систему керування потоками даних у вигляді мікросервісів, доповнити архітектуру мікросервісами, що побудовані на основі систем керування даних.

6. Орієнтовний перелік публікацій Сарапулов В. С. Мікросервісна архітектура у системах керування потоками даних / В. С. Сарапулов. // Міжнародний науковий журнал "Інтернаука". – 2018. – №20. – С. 81–101.

7. Консультанти розділів дисертації*

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Реалізація стартап-проекту	к.т.н., доц., Харченко Костянтин Васильович		

8. Дата видачі завдання _01.02.2018_

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Строк виконання етапів магістерської дисертації	Примітка
1	Отримання завдання	01.02.2018	
2	Збір інформації та аналіз літератури	15.02.2018	
3	Проведення огляду існуючих методів побудови мультиагентних систем	28.02.2018	
4	Формалізація задачі розробки методу прийняття колективного рішення в мультиагентних системах	11.03.2018	
5	Реалізація поставленої задачі	13.04.2018	
6	Аналіз та порівняння результатів моделювання	25.04.2018	
7	Оформлення дипломної роботи	30.04.2018	
8	Отримання допуску до захисту та подача роботи в ДЕК	09.05.2018	

Студент ____________ ____Сарапулов В.С._____
(підпис) (ініціали, прізвище)

Науковий керівник дисертації ____________ _____Харченко К.В.______
(підпис) (ініціали, прізвище)

* Консультантом не може бути зазначено наукового керівника

РЕФЕРАТ НА МАГІСТЕРСЬКУ ДИСЕРТАЦІЮ

виконану на тему: Мікросервісна архітектура у системах керування потоками даних

студентом: Сарапуловим Віктором Сергійовичем

Загальний обсяг роботи: 73 сторінок, 15 рисунків, 22 таблиці, 18 бібліографічних найменувань.

Актуальність теми

Дана магістерська дисертація присвячена дослідженню мікросервісних архітектур, побудованих на системах керування потоками даних. Метою роботи є розробка власної мікросервісної архітектури із використанням систем керування потоками даних.

Отже, актуальною науково-прикладною проблемою є створення підходу до побудови більш гнучкої та швидкої архітектури, побудованої на мікросервісах.

Мета та задачі дослідження

Метою роботи є розроблення архітектури, швидкість та гнучкість якої збільшуватиметься за рахунок використання систем керування потоками даних задля поліпшення продуктивності сучасних архітектур та змінити уявлення мікросервісних архітектур в цілому.

Рішення поставлених завдань та досягнуті результати

В роботі розглянуто засоби оркеструвания та масштабування мікросервісних архітектур та визначено найкращий спосіб її реалізації із використанням системи керування потоками даних під назвою DFVM. Розроблена архітектура дає можливість визначити переваги та недоліки у використанні систем керування потоками даних в контексті мікросервісів.

Об’єкт досліджень

Системи керування потоками даних у контексті мікросервісної архітектури.

Предмет досліджень

Побудова мікросервісної архітектури на основі систем керування потоками даних.

Методи досліджень

Для вирішення проблеми в даній роботі використовуються методи аналізу і синтезу, системного аналізу, порівняння, логічного узагальнення результатів.

Наукова новизна

Наукова новизна роботи полягає у створенні нових моделей для вирішення задач перетворення монолітних архітектур у мікросервісні.

Практичне значення одержаних результатів

Отримані результати можуть використовуватись у майбутніх дослідженнях за напрямком покращення запропонованої моделі, враховуючи переваги та недоліки даних результатів. Також дана модель може бути використана для покращення результатів роботи існуючих мікросервісних систем.

Публікації

Сарапулов В.С. Мікросервісна архітектура у системах керування потоками даних / В.С. Сарапулов // Міжнародний науковий журнал "Інтернаука". – 2018. – №20. – С. 81–101.

Ключові слова

Мікросервіси, система керування потоками даних, системи фон Неймана, оркестрування, масштабування.

ABSTRACT ON MASTER'S THESIS

on topic: Microservice architecture in dataflow control systems

student: Viktor S Sarapulov

Work carried out on 73 pages containing 15 figures, 22 tables. The paper was written with references to 18 different sources.

Topic

This master's dissertation is devoted to research of microservice architectures, built on data flow control systems. The purpose of the work is to develop its own microservice architecture using data flow management systems.

Consequently, the actual scientific and applied problem is the creation of an approach to building a more flexible and fast architecture built on microservices.

Purpose

The purpose of the work is to develop architecture, the speed and flexibility of which will increase with the use of data flow management systems to increase the productivity of modern architectures and change the representation of micro-server architectures in general.

Solution

The paper considers the means of orchestration and scaling of microservice architectures and identifies the best way to implement it using the data flow management system called DFVM. The developed architecture makes it possible to determine the advantages and disadvantages of using data flow control systems in the context of the microservice.

Object of research

Systems for managing dataflows in the context of a microservice architecture.

Subject of research

Building a microservice architecture based on dataflow management systems..

Research methods

To solve the problem in this paper we use methods of analysis and synthesis, system analysis, comparison, logical generalization of the results.

Scientific novelty

The scientific novelty of the work consists in the creation of new models for solving the problems of converting monolithic architectures into microspheres.

The practical value of the results

The results obtained can be used in future studies to improve the proposed model, taking into account the advantages and disadvantages of these results. Also, this model can be used to improve the performance of existing microsystems.

Publications

Sarapulov V. S. Microservice architecture in dataflow control systems V. S. Sarapulov. // International scientific magazine "Internet Science". – 2018 - №20. - P. 81-101.

Keywords

Microservices, dataflow, fon Neyman systems, orchestration, scaling.

ЗМІСТ

СПИСОК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ ТА ТЕРМІНІВ10

ВСТУП..11

1 КОМП'ЮТЕРНІ СИСТЕМИ КЕРУВАННЯ 12

1.1 Системи фон Неймана 12

1.1.1 Передумови створення та першоджерела......................................13

1.1.2 Формальне визначення 14

1.1.3 Машина фон Неймана (класична структура) 14

1.1.4 ЕОМ та принципи її функціонування 15

1.1.5 Недоліки та сучасні перспективи архітектури фон Неймана 16

1.2 Гарвардська система 18

1.2.1 Класична гарвардська архітектура 19

1.3 Системи керування потоками даних 20

1.3.1 Передумови створення та першоджерела......................................21

1.3.2 Статична система керування потоками даних 22

1.3.3 Динамічна система керування потоками даних............................22

1.4 Системи керування з високорівневою машинною мовою 23

1.5 Основні результати та висновки з розділу 1 23

2 АРХІТЕКТУРИ КОМП'ЮТЕРНИХ ПРОГРАМ 24

2.1 Монолітна архітектура..24

2.2 Мікросервісна архітектура 29

2.3 Висновки до розділу 31

3 ПОБУДОВА МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ НА ОСНОВІ СИСТЕМИ КЕРУВАННЯ ПОТОКАМИ ДАНИХ 33
3.1 Система керування потоками даних DFVM 33
3.2 Система швидкого запуску сервісів Vagrant 38
3.3 SaltStack як метод оркестрування та масштабування мікросервісної архітектури 41
3.4 Висновки до розділу 44
4 ДОСЛІДЖЕННЯ ПОБУДОВАННОЇ АРХІТЕКТУРИ, ЗБІР ТА АНАЛІЗ ДАНИХ 45
4.1 Первинна характеристика тестового середовища 45
4.2 Представлення DFVM в якості мікросервісу 46
4.3 Порівняльна характеристика 47
4.4 Висновки до розділу 52
5 РОЗРОБЛЕННЯ СТАРТАП-ПРОЕКТУ «МІКРОСЕРВІСНА АРХІТЕКТУРА У СИСТЕМАХ КЕРУВАННЯ ПОТОКАМИ ДАНИХ» 53
5.1 Опис ідеї проекту 53
5.2 Технологічний аудит ідеї проекту 54
5.3 Аналіз ринкових можливостей 56
5.4 Розробка ринкової стратегії проекту 63
5.5 Розробка маркетингової програми 66
5.6 Висновки до розділу 69
ВИСНОВКИ 71
ПЕРЕЛІК ПОСИЛАНЬ 72

СПИСОК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ ТА ТЕРМІНІВ

DFVM – Data flow virtual machine

МКА – мікросервісна архітектура

XML – eXtensible Markup Language

ВСТУП

Підвищення вимог до сучасних інформаційних систем призводить до появи нових архітектурних підходів до їх створення. Сучасні підприємства стикаються з проблемами, обумовленими зростанням обсягу даних, необхідністю інтеграції з партнерами і новими підрозділами, оновленням програмної платформи підприємства зі збереженням старих, але перевірених часом систем. Зі зростанням таких проблем монолітні архітектури потроху втрачають свою актуальність через велику складність підтримувати її та вносити зміни для її покращення. Таким чином все більшої популярності набувають мікросервісні архітектури. Головний принцип таких архітектур – розбиття програми на певні модулі (сервіси), які з легкістю можна вилучати або додавати нові без виведення з ладу всієї системи. Такий підхід дає можливість розширювати інформаційну систему, масштабувати її та забезпечувати швидку та гнучку оркестрацію всіх модулів.

У даній роботі буде розглянуто сучасні підходи оркестрування та масштабування мікросервісних архітектур. В якості ядра архітектури буде використано систему керування потоками даних. Дана система має ряд переваг у порівнянні зі стандартною системою фон Неймана, що теоретично має покращити та значно прискорити обробку інформації сервісів. Після розробки такої архітектури буде проведено ряд тестувань, що спростують або підтвердять дану теорію. В разі підтвердження теорії, такий підхід може перевернути сучасне бачення взаємодії мікросервісів між собою та впровадити новий стандарт розгортання мікросервісних архітектур в цілому.

1 КОМП'ЮТЕРНІ СИСТЕМИ КЕРУВАННЯ

На даний момент світові відомо чотири види комп'ютерних систем керування: Гарвардська система, система фон Неймана, система керування потоками даних та система керування з високорівневою машиною мовою. У цьому розділі розглянемо кожну з них окремо.

1.1 Системи фон Неймана

Архітектура, яку також називають фон Нейманівською – це певний тип комп'ютерних архітектур, що була побудована на основі статей та висновків різних вчених. Побудована була у 1945р. Математик і фізик Джон фон Нейман сформував вимоги до такої архітектури беручи за основу статті з першого проекту доповіді на EDVAC. Це описує дизайн-архітектуру для електронного цифрового комп'ютера з частинами, що складаються з обробного блоку, що містить арифметичну логічну одиницю та регістри процесорів; блок керування, що містить реєстр інструкцій та лічильник програм; пам'ять для зберігання даних і інструкцій; зовнішній накопичувач; і механізми введення та виведення. Сенс перетворився на будь-який комп'ютер із збереженими програмами, в якому команда завантажується, і операція передачі даних не може відбуватися одночасно, оскільки вони мають спільну шину. Це називається вузьким місцем фон Неймана і часто обмежує продуктивність системи.

Дизайн такої моделі архітектури виявився простішим, за машину Гарварду, яка також була системою, що мала один набір адресних та спеціальних інформаційних зчитуючих шин даних та їх записів в пам'ять, а інший – набір шин та адрес для отримання даних.

1.1.1 Передумови створення та першоджерела

Насправді, дана концепція, а точніше її авторство, належить цілому колективу авторів. Серед них були Дж. Екарт, Дж. Моклі і, звісно ж, фон Нейман. Всі вони працювали над створенням першої ЕОМ загального призначення, що була б доступна для перепрограмування – ENIAC (фон Нейман виступав у якості консультанта у цьому проекті, 1943). У 1952 році така технологія була глибше реалізована в EDVAC. Принципи формулювалися у різних публікаціях, серед яких можна було б виділити «Попередню дискусію про логічний устрій електронного обчислювального інструмента» та «Перший варіант доповіді про EDVAC». В обох випадках фон Нейман приймав участь і зробив свій внесок.

Найголовнішою проблемою, що поставала перед тодішніми математиками у сфері обчислювальної техніки велика складність введення алгоритмів різного роду обчислень в ЕОМ. Часто доводилось мати справу з великою кількістю різних тумблерів, роз'ємів, перемичок, що нерідко значно ускладнювало увесь процес розробки. Як наслідок, такі проблеми призводили до помилок, а універсальнішими ці машини не ставали.

Натомість, у зазначеній вище публікації фон Нейман запропонував разом із даними для впровадження обчислень зберігати і алгоритм. Збереження відбувалося б у пам'яті обчислювальної машини і такий підхід давав би можливість перепрограмовувати систему, з даними і з командами поводитися однаково, проводити над ними різні обчислювальні операції. Відкривалися можливості для написання програм, які зможуть модифікувати самі себе. Так і пропонувалась організація проведення обчислень, що перетворювала машину на інструмент, що можна було б використовувати універсально.

Хоча ця структура і унеможливлювала процес розрізнення даних в пам'яті від збережених команд, однак фон Нейман вказав, що такий підхід міг би бути можливим за умови, що машина буде розрізняти ці дві речі як різні окремі сутності. Згодом була впроваджена концепція підрахунку команд. Лічильник команд –

частина пам’яті, в якій зберігалися адреси поточних команд. Після виконання програми, стара адреса змінювалася наступною. Адреси операндів були збережені в самій команді.

1.1.2 Формальне визначення

Машина є обчислювальною архітектурою фон Неймана, якщо:

1. Дані і програму зберігають в загальній пам'яті. Це дає можливість прирівнювати команди до даних і виконувати над ними ті ж дії.
2. У кожної комірки пам'яті машини є власний ідентифікатор, що представляє собою унікальний номер, який називається адресою.
3. Різні інформаційні слова (команди або дані) розрізняються не за способом кодування і структурою представлення, а за способом використання в пам'яті.
4. Програми виконуються послідовно, починаючи від першої команди. Виключення – наявність спецвказівок. Щоб змінити цю послідовність, використовують команди передачі управління.

1.1.3 Машина фон Неймана (класична структура)

“Машина фон Неймана, як кожна ЕОМ загального призначення нашого часу, складається з чотирьох основних компонентів:

1. Операційний пристрій (ОП). Виконує команди з певного набору, який називається системою команд, над частинами інформації, яку зберігають у відокремленій від ОП пам'яті. Сучасні пристрої мають в складі ОП додаткову пам'ять (так званий, банк регістрів), де операнди або команди зберігаються відносно короткий час в процесі проведення обчислення.
2. Пристрій управління (ПУ). Організовує послідовне виконання алгоритмів, виконує розшифрування команд, які згодом надходять із пристрою пам'яті. Також він реагує на аварійні ситуації та виконує

функції управління всіма вузлами машини обчислення. Зазвичай операційний пристрій та пристрій управління об'єднуються в структуру, що називається центральним процесором (ЦП). Відзначимо, що вимога саме послідовного, в порядку надходження з пам'яті (в порядку зміни адрес в лічильнику команд) виконання команд є принциповою. Архітектури, що не дотримуються такого принципу, взагалі не вважають фон-нейманівськими.

3. Пристрій пам'яті (ПП). Масив, що складається з комірок з унікальними ідентифікаторами, в яких зберігаються дані та команди.
4. Пристрій вводу-виводу (ПВВ). Забезпечує зв'язок ЕОМ з різними пристроями, які приймають результати та передають інформацію на переробку в ЕОМ.” [2]

1.1.4 ЕОМ ТА ПРИНЦИПИ ЇЇ ФУНКЦІОНУВАННЯ

Після завантаження програми (алгоритму й даних для обробки) в запам'ятовуючий пристрій, машина фон Неймана може працювати автоматично, без втручання оператора. Кожна комірка пам'яті машини має унікальний номер — адресу, спеціальний механізм, найчастіше — лічильник команд — забезпечує автоматичне виконання необхідної послідовності команд, і визначає на кожному етапі адресу комірки, з якої необхідно завантажити наступну команду.

Перед початком виконання програми в лічильник записується адреса її першої команди. Визначення адреси наступної команди відбувається за одним з наступних сценаріїв:

Якщо поточна команда не є командою передачі управління (тобто це просто арифметична або логічна операція над даними), то до поточного значення лічильника додається число, яке дорівнює довжині поточної команди в мінімально адресованих одиницях інформації (зрозуміло, що це можливо за умови, якщо звичайні команди в блоках, не розділених командами передачі управління,

розташовуються послідовно в пам'яті, інакше адреса наступної команди може зберігатись, наприклад, безпосередньо в команді).

Якщо поточна команда — команда передачі управління (команда умовного або безумовного переходу), яка змінює послідовний хід виконання програми, то в лічильник примусово записується адреса тої команди, яка була замовлена при виконанні переходу, де б вона не знаходилась.

1.1.5 Недоліки та сучасні перспективи архітектури фон Неймана

“Хоча це не стосується безпосередньо принципів фон Неймана, але часто апелюють саме до «класичної архітектури фон Неймана» в критиці її досить примітивного та низькорівневого набору команд, який, на думку критиків, абсолютно не відповідає сучасному стану справ в індустрії розробки програмного забезпечення, зокрема в наявності мов високого рівня, які набагато підвищують продуктивність праці програміста за рахунок пропонування йому більш високорівневих абстракцій, і потрібно зазвичай до декількох сот машинних команд замість однієї команди мови високого рівня. Цей дисбаланс в принципі успішно вирішується на програмному рівні за допомогою компіляторів, але в 60—70 роки XX століття було досить багато намагань реалізувати машинні мови високого рівня апаратно (див. Архітектура з розвинутими засобами інтерпретації). Серед вітчизняних розробок в цьому напрямі слід виділити ЕОМ серії «МИР», а серед серйозних критиків системи фон-Неймана, в тому числі і за низький семантичний рівень команд, академіка В. М. Глушкова. Певною мірою, намаганням «підвищити семантичний рівень» можна вважати і CISC-архітектури системи команд, хоча як довів час, перспективнішим виявився прямо зворотний напрямок максимальної «примітивізації» набору команд, реалізований в RISC-архітектурах.

Розділення операційного пристрою та пам'яті в класичній архітектурі фон-Неймана вважається її суттєвим недоліком. Полюбляють казати про так звану «шийку пляшки» фон-нейманівської архітектури (термін, запропонований Джоном Бекусом (John Backus) в 1977. Ця «шийка» виникає між операційним пристроєм

(центральним процесором) і пам'яттю, адже швидкість обробки інформації в процесорі зазвичай є набагато більшою, аніж швидкість роботи пристрою пам'яті, який не встигає забезпечувати процесор новими порціями інформації, що призводить до простоїв. Проблема вирішується за рахунок побудови складнішої ієрархії пам'яті, зокрема введенням кеш-пам'яті (швидшої, але й дорожчої за основну). У кеш-пам'ять зберігаються дані, які часто використовуються в обчисленнях, що зменшує кількість звертань до повільнішої основної пам'яті. Існують також і радикальні пропозиції, які в останній час почали втілюватись в життя, і полягають в створенні так званої «розумної пам'яті», яка б інтегрувала комірки пам'яті зі схемами обробки даних.

Архітектура фон Неймана є принципово послідовною. І це є суттєвим обмежуючим фактором в підвищенні швидкодії машин з такою організацією, унеможливлює введення явного паралелізму в систему. Передусім це питання не технічне, а концептуальне і пов'язане з самою парадигмою програмування для фон-нейманівських машин. Саме тому паралельні обчислювальні машини, хоча й успішно виконують свої завдання, ще довго, мабуть, не зможуть витіснити цю класичну архітектуру.

Разом з тим, хоча майже всі ЕОМ загального призначення є фон-нейманівськими, вони суттєво використовують механізми розпаралелювання обчислень, хоча це відбувається й неявно, на рівні внутрішньої організації процесора, який непомітно для програміста виявляє схований паралелізм в послідовних програмах для фон-нейманівських машин.

Така «непомітність» є принциповою. Фактично фоннейманівською в сучасних ЕОМ залишається саме архітектура обчислювальної машини (тобто програмна організація). Внутрішня організація сучасних процесорів радикально використовує нефоннейманівські принципи виконання команд, але «виведення» цих принципів безпосередньо в архітектуру ЕОМ, тобто відкриття їх для програміста, яке на перший погляд може здаватись доцільним, насправді може

зруйнувати всю індустрію, і саме в цьому є секрет привабливості фоннейманівської архітектури. Фактично, ця концепція пропонує програмісту надзвичайно просту модель виконання програми, послідовну модель, яка збігається з образом мислення більшості програмістів, і тому найчастіше використовується при створенні програм. Явне паралельне програмування — це надзвичайно складна галузь, яка потребує повної перебудови образу мислення програміста, оперування складнішими абстракціями, застосування зовсім інших алгоритмів та структур даних. Тому збереження фоннейманівської архітектури, яким би стримуючим фактором воно не було, є абсолютно принциповим для проектувальників ЕОМ загального призначення.”[3]

1.2 Гарвардська система

Гарвардська архітектура (англ. Harvard architecture) — така архітектура обчислювальних машин, відмінністю якої було головним від інших архітектур те, що дані та оператори (алгоритм) зберігаються окремо.

Така структура має одну важливу перевагу над фон-нейманівською архітектурою: дані можна завантажувати для обробки з запам'ятовуючого пристрою одночасно з командами. В фон-нейманівській архітектурі для зв'язку операційного та керувального пристроїв (які разом складають центральний процесор), використовується одна шина, тому необхідно спочатку завантажити в процесор команду, а вже потім, звернувшись по тій же шині за адресою, яка вказана в команді — завантажити дані. Наявність в гарвардській архітектурі двох незалежних підсистем пам'яті з окремими шинами дозволяє вести процес завантаження команд і даних практично паралельно.

Головним недоліком гарвардської архітектури є порівняна з фон-нейманівською складність реалізації. Адже для кожного з запам'ятовуючих пристроїв необхідний свій контролер і своя шина, що зі збільшенням розрядності

призводить до зростання кількості з'єднань у системі, і це негативно впливає як на складність проектування, так і на швидкодію.

Гарвардська архітектура широко застосовується в спеціалізованих обчислювачах, зокрема в мікроконтролерах та цифрових сигнальних процесорах, де необхідний високоінтенсивний обмін даними. Також за гарвардською архітектурою зазвичай організується кеш-пам'ять в ЕОМ загального призначення, яка розділяється окремо на кеш-пам'ять команд та кеш-пам'ять даних (але, точніше, це стосується внутрішньої організації процесора, а не архітектури ЕОМ).

1.2.1 Класична гарвардська архітектура

“Типові операції (додавання та множення) вимагають від будь-якого обчислювального пристрою кількох дій:

- вибірку двох операндів;
- вибір інструкції та її виконання;
- збереження результатів.

Ідея, реалізована Ейкеном, полягала у фізичному поділі ліній передачі команд та даних. У першому комп'ютері Ейкена «Марк I» для зберігання інструкцій використовувалася перфострічка, а для роботи з даними — електромеханічні регістри. Це дозволяло одночасно пересилати й обробляти команди і дані, завдяки чому значно підвищувалася загальна швидкодія комп'ютера.”[6]

1.3 Системи керування потоками даних

Архітектура керування потоків даних (dataflow architecture) — така архітектура обчислювальної техніки, у якій потік даних керує процесом обчислень. Такий потік переміщуються між обчислювальною технікою машини, де перероблюються, поступово перетворюючи себе до нормального вигляду.

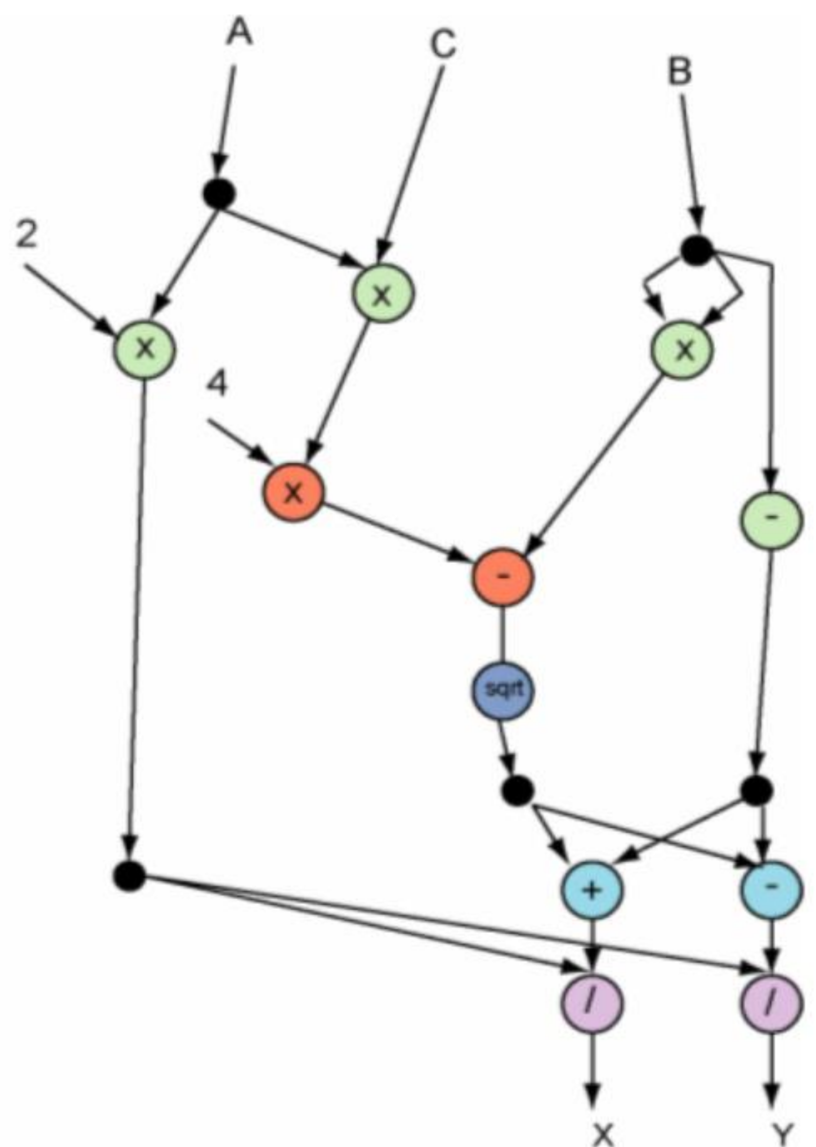

Рис. 1.1 – Схематична модель системи керування потоками даних [2]

В архітектурах керування потоками даних процесом обчислень керує потік даних, на відміну від інших архітектур. Ознакою того, що до обчислень в машині може приступити деякий ОП виступає певна готовність необхідних операндів до

проведення обчислення. І саме тому у деяких машинах з розгляненим типом керування, операції можуть бути виконані паралельно та без залежності одна від одного. Система керування потоком даних не має необхідності у централізованому керуванн у лічильниках команд, послідовністю виконання, тощо, та взагалі у будь-якому іншому керуванні потоками даних.

1.3.1 Передумови створення та першоджерела

Апаратні архітектури керування потоками даних були основною темою дослідження комп'ютерних архітектур в 1970-х і на початку 1980-х років. Джек Денніс з MIT став першим у галузі статичних потоків архітектур потоку даних, тоді як Manchester Dataflow Machine та MIT Tagged Token були основними проектами в динамічному потоці даних.

Дослідження, однак, ніколи не подолали проблеми, пов'язані з:

- Ефективністю трансляції токенів даних у паралельній системі.
- Ефективністю відправки токенів інструкцій в паралельній системі.
- Побудова досить великої асоціативної пам'яті, щоб тримати всі залежності реальної програми.

Інструкції та їх дані залежностей виявилися надто дрібними, щоб ефективно розповсюджуватися у великій мережі. Тобто час інструкцій та позначених результатів для подорожей через велику мережу з'єднання було довшим, ніж час, щоб фактично виконувати обчислення.

Тим не менш, позачергове виконання (ПВ) стало основною комп'ютерною парадигмою з 1990-х років. Це форма обмеженого потоку даних. Ця парадигма представила ідею виконання вікна. Вікно виконання виконує послідовний порядок архітектури фон Неймана, однак в межах вікна інструкції можуть бути заповнені в порядку залежності від даних. Це відбувається в процесорах, які динамічно позначають залежності даних коду в вікні виконання. Логічна складність

динамічного відстеження залежностей даних обмежує процесорів ПВ до невеликої кількості виконавчих пристроїв (2-6) і обмежує розміри виконавчого вікна в діапазоні від 32 до 200 інструкцій, набагато менших, ніж передбачено для повних машин потоку даних.

1.3.2 Статична система керування потоками даних

Описана вище схема називається статичною (static dataflow). У ній кожен обчислювальний вузол представлений в єдиному екземплярі, число вузлів заздалегідь відомо, також заздалегідь відомо число токенов, що циркулюють в системі. Як приклад реалізації статичної архітектури можна привести MIT Static Dataflow Machine - потоковий комп'ютер, створений в Массачусетському технологічному інституті в 1974 році. Машина складалася з безлічі обробних елементів (Processing Element), пов'язаних комунікаційною мережею.

Роль пристрою зіставлення тут виконувала пам'ять взаємодій (activity store). У ній зберігалися пари токенов разом з адресою вузла призначення, прапорами готовності і кодом операції. Будь-обчислювальний вузол в цій архітектурі мав тільки два входи і складався з одного оператора. При виявленні готовності обох операндів пристрій вибірки (fetch unit) прочитував код операції, і дані відправлялися на обробку в виконавчий пристрій (operation unit).

1.3.3 Динамічна система керування потоками даних

У динамічної потокової архітектурі (dynamic dataflow) кожен вузол може мати безліч екземплярів. Для того, щоб розрізняти токени, адресовані в різні екземпляри одного вузла, в структуру токена вводиться додаткове поле - контекст. Зіставлення токенов тепер ведеться не тільки по мітках, а й за значеннями контексту. У порівнянні зі статичної архітектурою з'являється цілий ряд нових можливостей.

Рекурсія. Вузол може направляти дані в свою копію, яка буде відрізнятися контекстом (але при цьому мати ту ж мітку).

Підтримка процедур. Процедурою в рамках даної моделі обчислень буде послідовність вузлів, пов'язаних між собою і що має входи і виходи. Можна одночасно викликати кілька примірників однієї й тієї ж процедури, які будуть відрізнятися контекстом.

Розпаралелювання циклів. Якщо між ітераціями циклу немає залежності за даними, можна обробляти відразу все ітерації одночасно. Номер ітерації, як ви вже напевно здогадалися, буде міститися в поле контексту.

Однією з перших реалізацій динамічної потокової архітектури була система Manchester Dataflow Machine (1980 рік). Машина містила апаратні засоби для організації рекурсій, виклику процедур, розкриття циклів, копіювання та об'єднання гілок обчислювального графа. Також в окремий модуль була винесена пам'ять команд (instruction store unit).

Динамічна dataflow-архітектура, в порівнянні зі статичною, демонструє більш високу продуктивність, за рахунок кращого паралелізму обчислень. Крім того, вона дає більше можливостей для програміста. З іншого боку, динамічна система складніше по апаратної реалізації, особливо це стосується пристроїв зіставлення і блоків формування контексту токенов.

1.4 Системи керування з високорівневою машинною мовою

“Архітектура з розвинутими засобами інтерпретації (архітектура ЕОМ з високорівневою машинною мовою) — архітектура обчислювальних машин, в машинній мові якої інтегровані елементи мов високого рівня, або яка на апаратному рівні підтримує деякі складні програмні абстракції.”[8]

1.5 Основні результати та висновки з розділу 1

1. Розглянуто 4 види комп’ютерних систем.

2. Досліджено переваги та недоліки кожної системи.

3. Проведено детальний аналіз еомп’ютерних систем.

2 АРХІТЕКТУРИ КОМП'ЮТЕРНИХ ПРОГРАМ

У цьому розділі описані основні два типа архітектури комп'ютерів. Також було проведено порівняння цих двох типів та виявлення їх переваг та недоліківо

2.1 Монолітна архітектура

У програмній інженерії монолітна архітектура описує однорівневий програмний додаток, в якому інтерфейс користувача і код доступу до даних об'єднуються в єдину програму з однієї платформи.

Монолітна архітектура автономна і незалежна від інших обчислювальних додатків або компонентів. Філософія дизайну полягає в тому, що програма відповідає не лише за певне завдання, але й може виконувати кожен крок, необхідний для виконання певної функції. Сьогодні деякі фінансові програми є монолітними в тому сенсі, що вони допомагають користувачеві виконувати завдання у повному обсязі, являються "приватними інформаційними сховищами", а не частинами більшої системи програм, які працюють разом. Деякі текстові процесори монолітними додатками.

У програмній інженерії монолітна архітектура описує додаток, розроблений без модульності. Модульність бажана, в цілому, оскільки вона підтримує повторне використання частин логіки додатків, а також полегшує технічне обслуговування, дозволяючи ремонтувати або замінювати частини програми без необхідності оптової заміни.

Модульність досягається різними масштабами за допомогою різних модулярних підходів. Кодова модульність дозволяє розробникам повторно використовувати та відновлювати частини програми, але для виконання цих функцій технічного обслуговування потрібні інструменти розробки (наприклад, можливо, потрібно буде перекомпілювати програму). Об'єктна модульність перетворює додаток у сукупність окремих виконуваних файлів, які можна незалежно підтримувати та замінювати без перерозподілу всієї програми

(наприклад, файли Microsoft DLL, файли спільного об'єкта Sun UNIX). Деякі можливості обміну повідомленнями об'єктів дозволяють розповсюджувати об'єктові програми на кількох комп'ютерах (наприклад, Microsoft COM+). Сервісні архітектури використовують специфічні стандарти зв'язку / протоколи для зв'язку між модулями.

У своєму оригінальному використанні термін "моноліт" описував величезні програми без використання модульності. Це у поєднанні з швидким збільшенням обчислювальної потужності і, отже, стрімким зростанням складності проблем, які можуть бути вирішені програмним забезпеченням, призвели до незмінних систем та «кризи програмного забезпечення».

Уявімо, що ви створюєте програму електронної комерції, яка приймає замовлення від клієнтів, перевіряє інвентар та доступний кредит, а також відповідає за відправку замовлень. Додаток складається з декількох компонентів, включаючи StoreFrontUI, який реалізує інтерфейс користувача, а також деякі служби підтримки для перевірки кредиту, ведення замовлень на інвентар та доставку.

Програма розгортається як єдиний монолітний додаток. Наприклад, веб-додаток Java складається з одного WAR-файлу, який працює на веб-контейнері Tomcat. Додаток Rails складається з єдиної ієрархії каталогів, розгорнутих за допомогою Phusion Passenger в Apache / Nginx або JRuby на Tomcat. Ви можете запускати кілька примірників програми за балансувальником навантаження, щоб масштабувати та покращувати доступність.

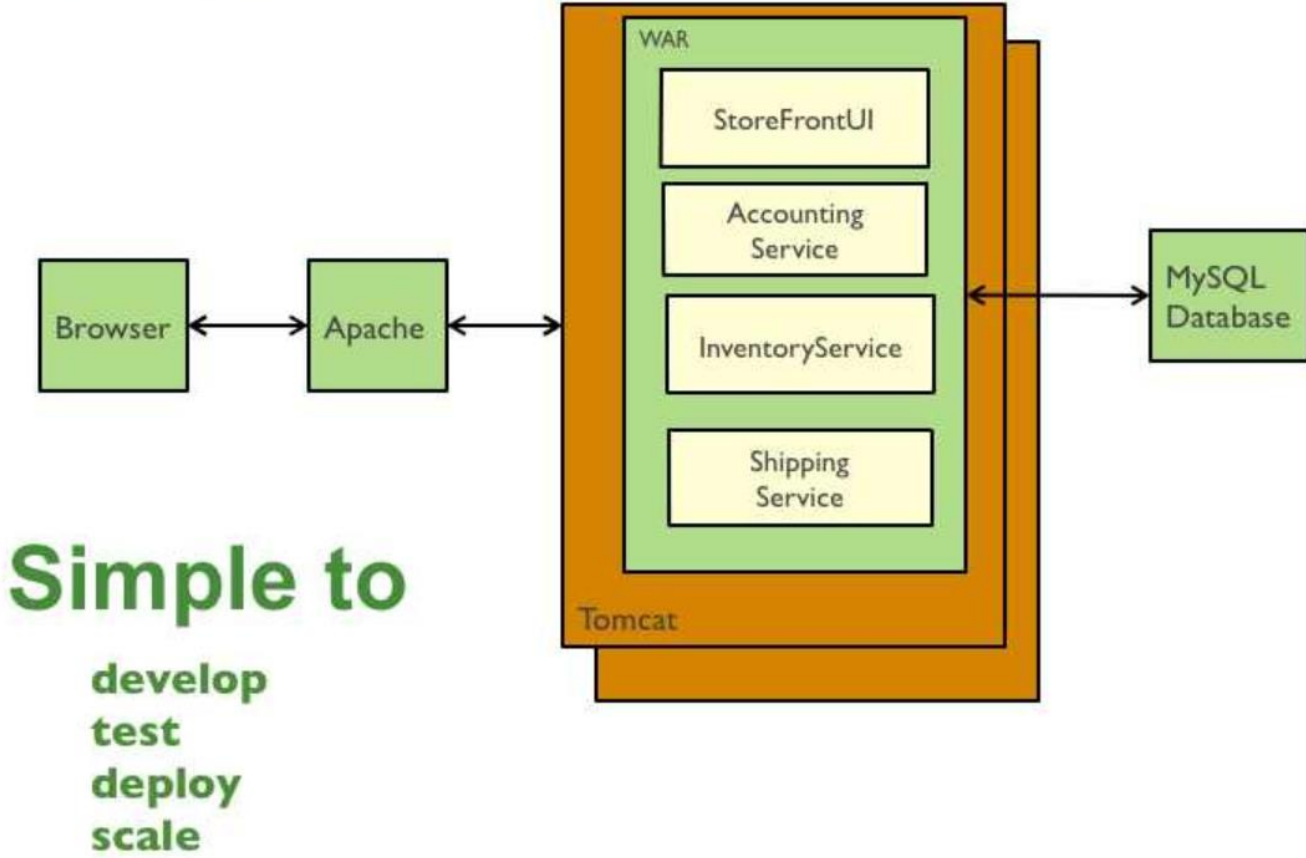

Рис. 2.1 – Приклад типової архітектури веб-додатків[3]

Таке рішення має ряд переваг:

- Проста розробка - мета сучасних інструментів розробки та середовищ розробки - це підтримка розробки монолітних додатків
- Просте розгортання - вам просто потрібно розгорнути файл WAR (або ієрархію каталогів) у відповідний час виконання
- Просте масштабування - ви можете масштабувати додаток, запустивши кілька копій програми за балансувальником навантаження

Проте, як тільки додаток стає великим, і команда зростає в розмірах, цей підхід має ряд недоліків, які стають все більш значущими:

- Велика монолітна кодова база лякає розробників, особливо тих, хто є новачком у команді. Додаток може бути важко зрозуміти та змінити. Як результат, розвиток, як правило, сповільнюється. Крім того, оскільки не існує жорстких зв'язкі між модуля, модульність з часом ламається. Крім того, оскільки складно зрозуміти, як правильно впровадити зміни, якість коду з часом зменшується. Це спіраль, що йде вниз.
- Перевантажена IDE - чим більша кодова база, тим повільніше є IDE та менш продуктивні розробники.
- Перевантажений веб-контейнер - чим більше додаток, тим довше це потрібно для запуску. Це має величезний вплив на продуктивність розробників через те, що витрачється багато часу на очікування збору контейнера. Це також впливає на розгортання.
- Постійна розробка складна - велике монолітне застосування також є перешкодою для частого впровадження змін. Щоб оновити один компонент, потрібно перегорнути всю програму. Це призведе до переривання фонових завдань (наприклад, кварцових завдань у програмі Java), незалежно від того, чи впливають на них зміни та іноді виникають проблеми та фатальні сбої. Існує також шанс, що компоненти, які не були оновлені, не зможуть працювати правильно. У результаті збільшується ризик, пов'язаний з перебудовою компонента, що перешкоджає частому впровадженю змін. Особливо цу є проблемою для розробників інтерфейсу користувача, оскільки вони, як правило, потребують швидкого ітераційного переміщення та частої перебудови всього проекту.
- Масштабування програми може бути складним - монолітна архітектура полягає в тому, що вона може масштабуватися лише в одному вимірі. З одного боку, він може масштабуватися за рахунок збільшення обсягу транзакцій, запускаючи більше копій програми. Деякі хмари навіть можуть регулювати кількість примірників динамічно, залежно від завантаження.

Але, з іншого боку, ця архітектура не може масштабуватися за рахунок збільшення обсягу даних. Кожна копія екземпляра програми матиме доступ до всіх даних, що робить кешування менш ефективним та збільшує споживання пам'яті та трафік вводу-виводу. Крім того, різні компоненти програми мають різні вимоги до ресурсів - певні з них потребують більш інтенсивної роботи процесора, тоді як інші можуть використовувати пам'ять більш інтенсивно. З монолітною архітектурою ми не можемо масштабувати кожен компонент самостійно

- Перешкода для розвитку масштабу - монолітне застосування також є перешкодою для розширення масштабу. Після того, як додаток досягне певного розміру, корисно поділити інженерну організацію на групи, що зосереджуються на конкретних функціональних областях. Наприклад, ми можемо мати команду інтерфейсу користувача, групу обліку, команду інвентаризації тощо. Проблема з монолітною програмою полягає в тому, що вона не дозволяє командам працювати самостійно. Команди повинні координувати свої зусилля щодо розвитку та перерозподілу. Команді значно складніше внести зміни та оновлювати продукцію.
- Монолітна архітектура змушує вас бути прив'язаним до стеку технології (і в деяких випадках до конкретної версії цієї технології), яку ви обрали на початку розробки. За допомогою монолітної програми важко поступово впроваджувати нову технологію. Наприклад, припустімо, ви вибрали JVM. Ви маєте деякі варіанти вибору мови, так як Java, ви можете використовувати інші мови JVM, які чудово взаємодіють з Java, такими як Groovy та Scala. Але компоненти, написані мовами, відмінними від JVM, не мають місця у вашій монолітній архітектурі. Крім того, якщо ваша програма використовує платформу, яка згодом стає застарілою, то може бути складно поступово перенести додаток на нову та кращу структуру. Можливо, для

того, щоб прийняти нову платформу, потрібно переписати весь додаток, що є ризикованим заходом.

2.2 Мікросервісна архітектура

Мікросервісна архітектура або просто мікросервіси - це дещо інший спосіб розробки програмних систем, що набуває все більшой популярності. Завдяки своїй масштабованості цей архітектурний метод вважається особливо ідеальним, коли вам потрібно ввімкнути підтримку для різних платформ і пристроїв, що охоплюють веб, мобільний Інтернет, Інтернет речей, або просто, коли ви не впевнені, які пристрої вам потрібно буде підтримувати у майбутньому.

Незважаючи на відсутність формального визначення мікросервісу, існують певні характеристики, які допомагають нам визначити сутність. По суті, мікросервісна архітектура являє собою метод розробки програмних додатків як набір незалежно розгортаючих, маленьких модульних сервісів, в якій кожен сервіс виконує унікальний процес і спілкується за допомогою чітко визначеного, легкого механізму для обслуговування.

Спілкуються сервісів між собою залежить від вимог вашого застосування, але багато розробників використовують HTTP / REST з JSON або Protobuf. Фахівці DevOps, звичайно, можуть обрати будь-який протокол зв'язку підходить, але в більшості випадків, REST (Representational State Transfer) є кращим методом інтеграції через його порівняно низьку складність в порівнянні з іншими протоколами.

Що ж особливого є у міеросервісній архітектурі:

По-перше, програмне забезпечення, побудоване як мікросервіси, може, за визначенням, бути розбите на послуги з декількома компонентами. Таким чином, кожна з цих служб може бути розгорнута, налаштована, а потім перерозподілена самостійно, не посягаючи на цілісність програми. У результаті вам може знадобитися лише змінити одну або декілька різних служб, замість того, щоб передислокувати всі програми. Але цей підхід має свої недоліки, в тому числі дорогі

віддалені виклики (замість внутрішньопроцесориних викликів), складніші інтерфейси API, а також підвищена складність при перерозподілі відповідальності між компонентами.

По-друге, стиль мікросервісів, як правило, орієнтується на можливості та пріоритети бізнесу. На відміну від традиційного підходу до монолітної розробки, де кожна команда має свою специфіку роботи (інтерфейс користувача, бази даних, технологічні шари або логіка серверної сторони), мікросервісна архітектура використовує міжфункціональні команди розробників. Обов'язки кожної команди полягають у тому, щоб робити конкретні продукти на основі однієї або кількох індивідуальних послуг, що передаються через шину повідомлень. Це означає, що коли в систему доведеться внести зміни, не доведеться гаяти купу часу на перезібрання всієї системи, або виділення виділення бюджету під окремий сервіс. Більшість методів розробки зосереджуються на проектах: частина коду, яка має запропонувати певну визначену вартість бізнесу, повинна бути передана клієнту, а потім періодично підтримується командою. Але в мікросервісах команда володіє продуктом протягом всього свого життя.

По-третє, мікрослужби діють дещо як класична система UNIX: вони отримують запити, обробляють їх і відповідно формують відповідь. Можна сказати, що мікрослужби мають розумні кінцеві точки, які обробляють інформацію та застосовують логіку, а також шлюзи, через які поширюється інформація.

По-четверте, оскільки мікросервіси включають різні технології та платформи, методи централізованого управління є застарілими та неоптимальними. Децентралізоване управління - певний прототип мікросервісної архітектури, оскільки її розробники прагнуть створювати корисні інструменти, які потім можуть використовувати інші, щоб вирішувати ті самі проблеми. На практичному прикладі це Netflix - служба відповідає за приблизно 30% трафіку в Інтернеті. Компанія заохочує своїх розробників заощаджувати час, завжди використовуючи бібліотеки кодів, створені іншими, і надаючи їм свободу знайомитися з альтернативними

рішеннями, коли це необхідно. Так само як децентралізоване управління, мікросервісна архітектура також сприяє децентралізації управління даними. Монолітні системи використовують єдину логічну базу даних у різних програмах. У програмі мікросервісів кожна служба зазвичай керує своєю унікальною базою даних.

По-п'яте, мікрослужби призначені для усунення несправностей. Оскільки кілька унікальних і різноманітних служб спілкуються разом, цілком можливо, що сервіс може не відповідати на запити з тієї чи іншої причини (наприклад, коли постачальник недоступний). У таких випадках клієнт повинен дозволити працювати своїм сусіднім сервісам і паралельно цьому усувати несправність попереднього. Однак моніторинг мікрослужби може допомогти запобігти появі ризику невдачі. З очевидних причин ця вимога додає складності мікросервісній архітектурі в порівнянні з архітектурою монолітних систем.

Нарешті, дизайн мікросервісів є еволюційним і, знову ж, ідеально підходить для еволюційних систем, де ви не можете повністю передбачити типи пристроїв, які можуть коли-небудь мати доступ до вашої програми. Хороший приклад цього сценарію можна побачити на веб-сайті The Guardian (до реконструкції наприкінці 2014 року). Основна програма була спочатку заснована на монолітній архітектурі, але, як виникло декілька непередбачених проблем, замість перепрограмування всього додатку розробники стали використовувати мікросервіси, які взаємодіють з більш старою монолітною архітектурою через API.

2.3 Висновки до розділу

Кожен тип архітектури має свої переваги на недоліки. Монолітні програми прості для розгортання, оскільки зазвичай вони вимагають одноразового розгортання на одному сервері. Є випадки, коли монолітний додаток треба було розгорнути на декількох серверах; однак, монолітний додаток є тісно пов'язаним і, як правило, має цілий стек IT, присвячений йому. Масштабуються моноліти одним єдиним суцільним блоком.

На відміну від цього, архітектура мікросервісів включає в себе менші дискретні програмні компоненти, орієнтовані на сервіси, які є вільно пов'язаними, але нерідко згруповані за спільною ознакою(функціональна суміжність, відомча власність тощо) у так званому обмеженому контексті. Ці індивідуальні послуги можуть бути розгорнуті та масштабовані незалежно один від одного.

3 ПОБУДОВА МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ НА ОСНОВІ СИСТЕМИ КЕРУВАННЯ ПОТОКАМИ ДАНИХ

У цьому розділі буде описуються всі кроки побудови мікросервісної архітектури, сервісами якої виступатиме система керування потоками даних DFVM.

3.1 Система керування потоками даних DFVM

DFVM – (Dataflow Virtual Machine) система керування потоками даних, що була створена Костянтином Васильовичем Харченко. Дана система написана на мові програмування C++, являється кроссплатформенною, а це означає, що запускати її можна на різних операційних системах. Основна ідея даної системи полягає у тому, що в пям’яті комп’ютера зберігаються так звані ноди, або вузли, які зберігають інформацію щодо зв’язків вхідних і вихідних даних. Так як система досить нова, її можливості на даний момент обмежені, але в якості дослідницьких цілей її функціоналу буде достатньо. Система складається з декількох модулів:

- CommandProcessor – командний процесор, що зберігає в собі набір доступних команд, виконує їх, а також парсить поток даних
  Component – модуль, що відповідає за сбереження проміжних результатів (компонентів)
- VirtualMachine – відповідає за завантаження компонентів
- Dfvm – інтерфейс користувача
- ReadBuffer – буфер зчитування кількості байтів з потоку вводу

CommandProcessor.cpp	5.41 KB	2017-03-17
CommandProcessor.h	1.14 KB	2016-04-08
Component.cpp	10.54 KB	2018-01-05
Component.h	1.75 KB	2017-03-17
Flow.h	1.05 KB	2017-03-18
VirtualMachine.cpp	1.38 KB	2016-04-15
VirtualMachine.h	1.14 KB	2016-04-15
dfvm.cpp	2.6 KB	2017-03-17
readbuffer.cpp	576 B	2016-02-07

Рис. 3.1 – Структура системи керування потоками даних DFVM

На вхід програми необхідно подавати файли-паттерни, у яких вказана вся необхідна інформація, а саме: вхідні\вихідні дані, тип даних, які дії треба виконати над цими змінними. Паттерни dfvm мають дуже просту структуру. Розглянемо її на прикладі:

```
4 7 1 3
a b c d e f g
int int int int int int int
a b c d
1 2 4 3
g
mul 2 1 e f g
add 2 1 a b e
sub 2 1 c d f
```

Рис. 3.2 – Приклад вхідного файлу «example1.dfvm»

Перша стрічка відповідає за кількість вхідних нод, кількість нод загалом, кількість вихідних нодів та кількість команд, які треба провести. Друга і третя стрічка – назва змінних та їх тип. Четверта і п’ята стрічки відповідають за присвоєння значень нашим вхідним нодам. Шоста – яка нода є вихідною. Три останніх стрічки вказують на те, які саме операції треба виконати. Не дивлячись на те, що множення та підрахунок вихідної ноди відбувається на початку, її буде пораховано останньою. В цьому заключається одна з особливостей систем керування потоками даних – за рахунок збереження в пам’яті дерева зв’язків, програма розбиває дії за пріорітетами.

Таким чином, ми отримали вхідну інструкцію, що має порахувати значення виразу:

$$g = e * f, \quad (1)$$

де ‘e’ – сума чисел ‘a’ та ‘b’, а ‘f’ – різниця чисел ‘c’ і ‘d’. Так як вхідним параметрам ‘a’, ‘b’, ‘c’, ‘d’ були присвоєні значення 1, 2, 4, 3 відповідно, на виході ми маємо отримати наступне:

$$g = (1 + 2) * (4 - 3) = 3 \quad (2)$$

Поглянемо на роботу даної системи на практиці. Робота з нею проходить у командній строці. Використаємо паттерн, що було розглянуто раніше:

```
C:\Users\AdminDell\Desktop\Универ\Диплом\dfvm\windows_dfvm\Debug>windows_dfvm.exe example1.dfvm
Dataflow Virtual Machine, dfvm (c) 2015, 2016, 2017
Result 3
```

Рис. 3.3 Результат роботи системи керування потоками даних

Як ми бачимо, в кінці справді було одержано вірний результат. Однак чи правильно було виконано підрахунок? Щоб дізнатися, як треба користуватися DFVM, додайте -h.

```
C:\Users\AdminDell\Desktop\Универ\Диплом\dfvm\windows_dfvm\Debug>windows_dfvm.exe -h
Dataflow Virtual Machine, dfvm (c) 2015, 2016, 2017
Use: dfvm file.dfvm
 -h: this help
 -r: use old solver
 -v: verbose mode
 -I/path/: inclide path to .dfvm files
```

Рис. 3.4 – Інтерфейс користувача DFVM

Для виведення покрокового підрахунку, додайте до команди виклику системи керування потоками даних -v.

```
C:\Users\AdminDell\Desktop\Универ\Диплом\dfvm\windows_dfvm\Debug>windows_dfvm.exe -v example1.dfvm
Dataflow Virtual Machine, dfvm (c) 2015, 2016, 2017
Non-option argument example1.dfvm
Input filename example1.dfvm
inputs 4 nodes 7 outputs 1 commands 3
Flow names a
Flow names b
Flow names c
Flow names d
Flow names e
Flow names f
Flow names g
Flow type int
Flow type int
Flow type int
Flow type int
Flow type int
Flow type int
Flow type int
Inputs a 0
Inputs b 1
Inputs c 2
Inputs d 3
Input values1
Input values2
Input values4
Input values3
Outputs g 6
command mul
command code 2
Component inputs e 4
Component inputs f 5
Component outputs g 6
command add
command code 0
Component inputs a 0
Component inputs b 1
Component outputs e 4
command sub
command code 1
Component inputs c 2
Component inputs d 3
Component outputs f 5
```

Рис. 3.5 – Покрокове збереження даних у системі DFVM

Як ми бачимо, програма зберігла всю інформацію, що було подано на вхід. Command code – унікальний ідентифікатор дії, порядковий номер, під яким вона зьерігається у командному процесорі.

Наступним кроком, програма визначатиме, в якій послідовності дії необхідно виконувати:

```
New solver
4
5
0
1
2
3
0 0 1
1 0 1
2 0 2
3 0 2
4 0 0
5 0 0
0
1
2
3
States
need to solve component 1
need to solve component 1
need to solve component 2
need to solve component 2
Solve list
1
C = add
Op = 1
Op = 2
R = 3
2
C = sub
Op = 4
Op = 3
R = 1
Check output flow 6 0
need to solve component 0
need to solve component 0
Solve list
0
C = mul
Op = 3
Op = 1
R = 3
Check output flow 6 1
Result 3
```

Рис. 3.6 – Покрокове обчислення даних у системі DFVM

Як ми бачимо, усі операції було проведено коректно та у правильній послідовності.

3.2 Система швидкого запуску сервісів Vagrant

Vagrant - це інструмент для створення і управління віртуальними машинами в одному робочому потоці. Завдяки зручному робочому потоці та зосередженню на автоматизації, Vagrant знижує час встановлення середовища розробки та збільшує продуктивність розробки.

Vagrant забезпечує легкі в налаштуванні, відтворювані та портативні робочі середовища, побудовані на основі сучасних технологій та керуються єдиним послідовним робочим процесом, що допомагає максимально підвищити продуктивність та гнучкість команди розробників.

До того ж, Vagrant та SaltStack чудово взаємодіють одне з іншим та дозволяє навіть на дуже повільному компьютері розгорнути як мінімум 2 віртуальні системи.

Подивимось, що дана технологія може нам запропонувати:

```
Admin@Admin-▒▒ MINGW64 ~/Desktop/salt-vagrant-demo-master/salt-vagrant-demo-mast
er
$ vagrant -h
Usage: vagrant [options] <command> [<args>]

    -v, --version                    Print the version and exit.
    -h, --help                       Print this help.

Common commands:
     box             manages boxes: installation, removal, etc.
     destroy         stops and deletes all traces of the vagrant machine
     global-status   outputs status Vagrant environments for this user
     halt            stops the vagrant machine
     help            shows the help for a subcommand
     init            initializes a new Vagrant environment by creating a Vagrantfile
     login           log in to HashiCorp's Vagrant Cloud
     package         packages a running vagrant environment into a box
     plugin          manages plugins: install, uninstall, update, etc.
     port            displays information about guest port mappings
     powershell      connects to machine via powershell remoting
     provision       provisions the vagrant machine
     push            deploys code in this environment to a configured destination
     rdp             connects to machine via RDP
     reload          restarts vagrant machine, loads new Vagrantfile configuration
     resume          resume a suspended vagrant machine
     snapshot        manages snapshots: saving, restoring, etc.
     ssh             connects to machine via SSH
     ssh-config      outputs OpenSSH valid configuration to connect to the machine
     status          outputs status of the vagrant machine
     suspend         suspends the machine
     up              starts and provisions the vagrant environment
     validate        validates the Vagrantfile
     version         prints current and latest Vagrant version

For help on any individual command run `vagrant COMMAND -h`

Additional subcommands are available, but are either more advanced
or not commonly used. To see all subcommands, run the command
`vagrant list-commands`.
```

Рис. 3.7 – Інтерфейс користувача системи Vagrant

Як бачимо, даний інструментарій дає широкий спектр можливостей. Для того, щоб запустити Vagrant бокси, необхідно підготувати конфігураційний файл «Vagrantfile». Під боксами мається на увазі дистриб’ютиви віртуальних машин,

максимально стиснуті у контейнери. Конфігураційний файл необхідно зробити для того, щоб дана технологія чітко могла налаштувати всі віртуальні машини, які розгортатиме.

```
config.vm.define :master, primary: true do |master_config|
  master_config.vm.provider "virtualbox" do |vb|
      vb.memory = "2048"
      vb.cpus = 1
      vb.name = "master"
  end
    master_config.vm.box = "#{os}"
    master_config.vm.host_name = 'saltmaster.local'
    master_config.vm.network "private_network", ip: "#{net_ip}.10"
    master_config.vm.synced_folder "saltstack/salt/", "/srv/salt"
    master_config.vm.synced_folder "saltstack/pillar/", "/srv/pilla

    master_config.vm.provision :salt do |salt|
      salt.master_config = "saltstack/etc/master"
      salt.master_key = "saltstack/keys/master_minion.pem"
      salt.master_pub = "saltstack/keys/master_minion.pub"
      salt.minion_key = "saltstack/keys/master_minion.pem"
      salt.minion_pub = "saltstack/keys/master_minion.pub"
      salt.seed_master = {
                          "minion1" => "saltstack/keys/minion1.pub"
                          "minion2" => "saltstack/keys/minion2.pub"
                          "minion3" => "saltstack/keys/minion3.pub"
                          "minion4" => "saltstack/keys/minion4.pub"
                          "minion5" => "saltstack/keys/minion5.pub"
                         }

      salt.install_type = "stable"
      salt.install_master = true
      salt.no_minion = true
      salt.verbose = true
      salt.colorize = true
      salt.bootstrap_options = "-P -c /tmp"
    end
  end
```

Рис. 3.8 – Приклад файлу конфігурації “Vagrantfile”

Після налаштування конфігураційного файлу, введемо команду vagrant up.

```
Admin@Admin-▒▒▒ MINGW64 ~/Desktop/salt-vagrant-demo-master/salt-vagrant-demo-mast
er
$ vagrant up
Bringing machine 'master' up with 'virtualbox' provider...
Bringing machine 'minion1' up with 'virtualbox' provider...
Bringing machine 'minion2' up with 'virtualbox' provider...
==> master: Importing base box 'bento/ubuntu-16.04'...
==> master: Matching MAC address for NAT networking...
==> master: Checking if box 'bento/ubuntu-16.04' is up to date...
==> master: Setting the name of the VM: master
==> master: Clearing any previously set network interfaces...
==> master: Preparing network interfaces based on configuration...
    master: Adapter 1: nat
    master: Adapter 2: hostonly
==> master: Forwarding ports...
    master: 22 (guest) => 2222 (host) (adapter 1)
==> master: Running 'pre-boot' VM customizations...
==> master: Booting VM...
==> master: Waiting for machine to boot. This may take a few minutes...
    master: SSH address: 127.0.0.1:2222
    master: SSH username: vagrant
    master: SSH auth method: private key
    master:
    master: Vagrant insecure key detected. Vagrant will automatically replace
    master: this with a newly generated keypair for better security.
    master:
    master: Inserting generated public key within guest...
    master: Removing insecure key from the guest if it's present...
    master: Key inserted! Disconnecting and reconnecting using new SSH key...
==> master: Machine booted and ready!
==> master: Checking for guest additions in VM...
    master: The guest additions on this VM do not match the installed version of
    master: VirtualBox! In most cases this is fine, but in rare cases it can
    master: prevent things such as shared folders from working properly. If you
see
    master: shared folder errors, please make sure the guest additions within th
e
    master: virtual machine match the version of VirtualBox you have installed o
n
    master: your host and reload your VM.
    master:
    master: Guest Additions Version: 5.2.6
    master: VirtualBox Version: 5.1
==> master: Setting hostname...
==> master: Configuring and enabling network interfaces...
==> master: Mounting shared folders...
    master: /vagrant => C:/Users/Admin/Desktop/salt-vagrant-demo-master/salt-vag
rant-demo-master
    master: /srv/salt => C:/Users/Admin/Desktop/salt-vagrant-demo-master/salt-va
grant-demo-master/saltstack/salt
    master: /srv/pillar => C:/Users/Admin/Desktop/salt-vagrant-demo-master/salt-
vagrant-demo-master/saltstack/pillar
==> master: Running provisioner: salt...
Copying salt master config to vm.
Uploading minion keys.
Uploading master keys.
Checking if salt-master is installed
salt-master was not found.
Using Bootstrap Options:  -P -c /tmp -F -c /tmp -k /tmp/minion-seed-keys -M -N s
table
Bootstrapping Salt... (this may take a while)


%

T
o
t
a
```

Рис. 3.9 – Результат введення команди “vagrant up”

Після звернення до файлу Vagrantfile почалися розгортатися віртуальні машини. У випадку їх відсутності, програма сама шукала відповідні вказані бокси в інтернеті. Дана система також зразу встановлює всі пакети на вказані машини у відповідності із конфігураційним файлом. Як можна відзначити, перші спроби

розгортання мікросервісної архітектури проходили з використанням двох виконавчих машин та однієї командуючої.

Після закінчення налаштування та завантаження всіх необхідних пакетів, можна під’єднатися до віртуальної машини-командувача та розпочати налагодження внутрішньої роботи мікросервісної архітектури.

```
Configuration file '/etc/salt/minion'
 ==> File on system created by you or by a script.
 ==> File also in package provided by package maintainer.
 ==> Using current old file as you requested.
Processing triggers for libc-bin (2.23-0ubuntu10) ...
Processing triggers for systemd (229-4ubuntu21.2) ...

Processing triggers for ureadahead (0.100.0-19) ...
 *  INFO: Running install_ubuntu_stable_post()
 *  INFO: Running install_ubuntu_check_services()
 *  INFO: Running install_ubuntu_restart_daemons()
 *  INFO: Running daemons_running()
 *  INFO: Salt installed!
Salt successfully configured and installed!
run_overstate set to false. Not running state.overstate.
run_highstate set to false. Not running state.highstate.
orchestrate is nil. Not running state.orchestrate.

Admin@Admin-▒▒ MINGW64 ~/Desktop/salt-vagrant-demo-master/salt-vagrant-demo-mast
er
$ vagrant ssh master
Welcome to Ubuntu 16.04.4 LTS (GNU/Linux 4.4.0-87-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

43 packages can be updated.
20 updates are security updates.
```

Рис. 3.10 – Підключення до боксу Vagrant системи

3.3 SALTSTACK ЯК МЕТОД ОРКЕСТРУВАННЯ ТА МАСШТАБУВАННЯ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

Серед знайдених систем оркестрації мікросервісів було обрано SaltStack. SaltStack, або просто Salt – система одночасного керування сервісів. В концепції цього інструменту існує дві основні сутності: командувач (master) та виконувачі (minion). Усі виконувачі отримують команди від командувача одночасно. Це означає, що час, який потрібно для оновлення 10 або 10000 систем майже не відрізняється, а запити до тисячі різних систем можна виконати за лічені секунди. Особливість інструменту у тому, що він робить запити до необхідної інфраструктури в режимі реального часу, а не покладається на такий застарілий метод, як перевірка бази даних.

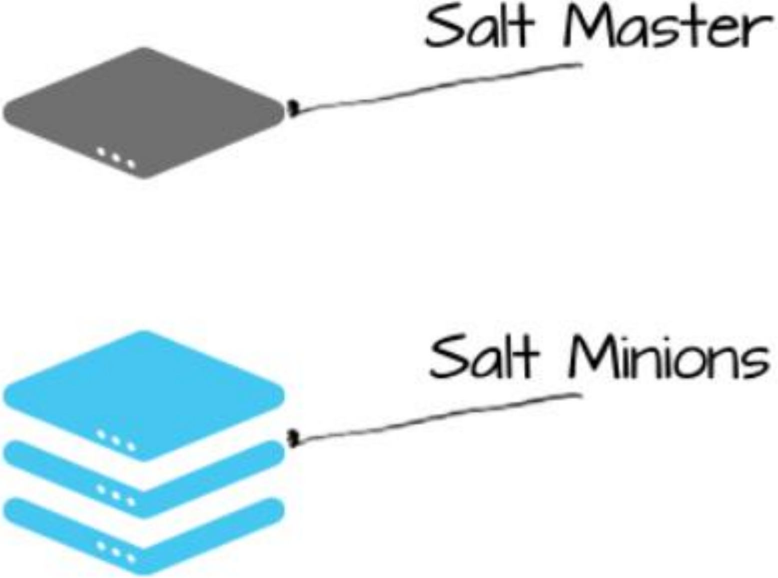

Рис. 3.11 – Основні сутності системи SaltStack

Кожен виконувач займається виключно своєю роботою. Дані, які передає командувач є надлегкими наборами інструкцій, які чітко описують, що мае робити конкретний виконувач, з конкретними характеристиками. Мільйони виконувачів визначають, чи підходять вони під вказані характеристики при отримані команди. Виконувачі зберігають всі команди локально, що дає змогу швидко їх виконати в необхідний момент та передати необхідні результати командувачеві.

Коли виконувач запускається вперше, він шукає мережу під назвою Salt (хоча це може бути легко змінено на IP або інше ім'я хоста). Коли знайдено, міньйон ініціює рукостискання, а потім надсилає свій відкритий ключ до командувача.

Рис. 3.12 – Регістрація публічного ключа міньйона у мастері

Однією з найголовніших критеріїв вибору інструментарію була доступність. SaltStack надають своїм користувачам можливість використовувати їх технології задарма та ще й на різних платформах. Як стверджують самі розробники, дана технологія працює скрізь, де працює Python.

Salt спроектовано для високої продуктивності та масштабування. Система передачі даних налагоджена таким чином, щоб був постійний непреривний зв’язок командувача та виконувачів на основі ZeroMQ або сирих пакетів TCP, що надає даній системі значні переваги у порівнянні з конкуруючими технологіями. Внутрішньо, Salt використовує Python Tornado як асинхронну мережеву бібліотеку.

Знаходячись вже у системі віртуальної машини-командувача, необхідно під’єднати приватні ключі виконавчих комп’ютерів. Для того, щоб дізнатися, чи всі ключі були під’єднані, необхідно ввести команду salt-key –list-all

```
vagrant@saltmaster:~$ sudo su
root@saltmaster:/home/vagrant# salt-key --list-all
\Accepted Keys:
minion1
minion2
Denied Keys:
Unaccepted Keys:
Rejected Keys:
```

Рис. 3.13 – Перевірка на присутність всіх ключів міньйонів

В нашому випадку, за рахунок правильно написаного конфігураційного файлу, усі ключі вже були під’єднані. В іншому разі, необхідно було б скористатися командою salt-key –accept=<key>. У випадку, коли всы ключі є валідними, та необхідно додати всі виконавчі машини, довелось би скористатися командою salt-key –accept-all.

Після того, як всі ключі було додано до машини-командувача, слід перевірити, чи всіх виконувачів під’єднано:

```
root@saltmaster:/home/vagrant# salt '*' test.ping
minion1:
    True
minion2:
    True
```

Рис. 3.14 – Перевірка на присутність зв’язку між мастером та міньйонами

Завершивши перевірку, можна приступати до оркестрації. Для одночасного звернення до всіх міньйонів, необхідно ввести команду salt ‘*’.

3.4 Висновки до розділу

Було виявлено, які інструментарії будуть використовуватися при побудові фінальної архітектури. На практиці було розглянено такі інструментарії як: Vagrant, SaltStack, DFVM.

Кожен з інструментів чудово підходить для реалізації власної мікросервісної архітектури, є доступними, та надають можливість гнучко і швидко обмінюватися даними між сервісами.

Vagrant – система, що дозволяє оптимальвикористовувати ресурси комп’ютера-хоста, на якому розгортаються мікросервіси, а SaltStack – інструмент, що дозволяє виконувати оркестрування мікросервісів та забезпечує зручне масштабування всієї системи.

4 ДОСЛІДЖЕННЯ ПОБУДОВАННОЇ АРХІТЕКТУРИ, ЗБІР ТА АНАЛІЗ ДАНИХ

4.1 Первинна характеристика тестового середовища

Під час проведення дослідження та проектування, було використано таке тестове середовище:

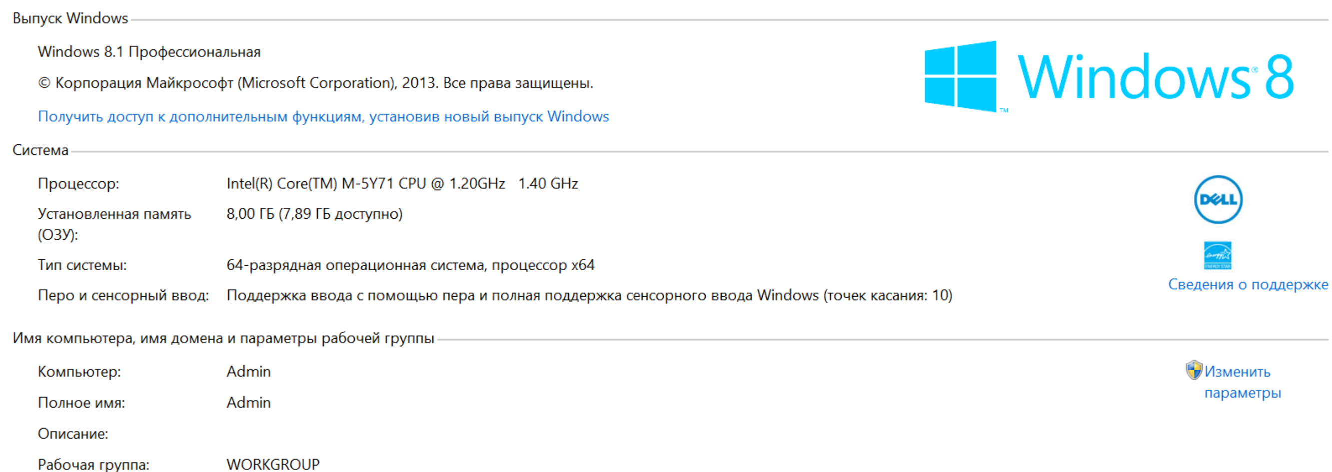

Рис. 4.1 – Характеристика тестової обчислювальної машини

В ході розробки мікросервісної архітектури, на даному комп’ютері вдалося розгорнути максимум 6 різних мікросервісів. Кожен мікросервіс представляв з себе окремий бокс системи Vagrant.

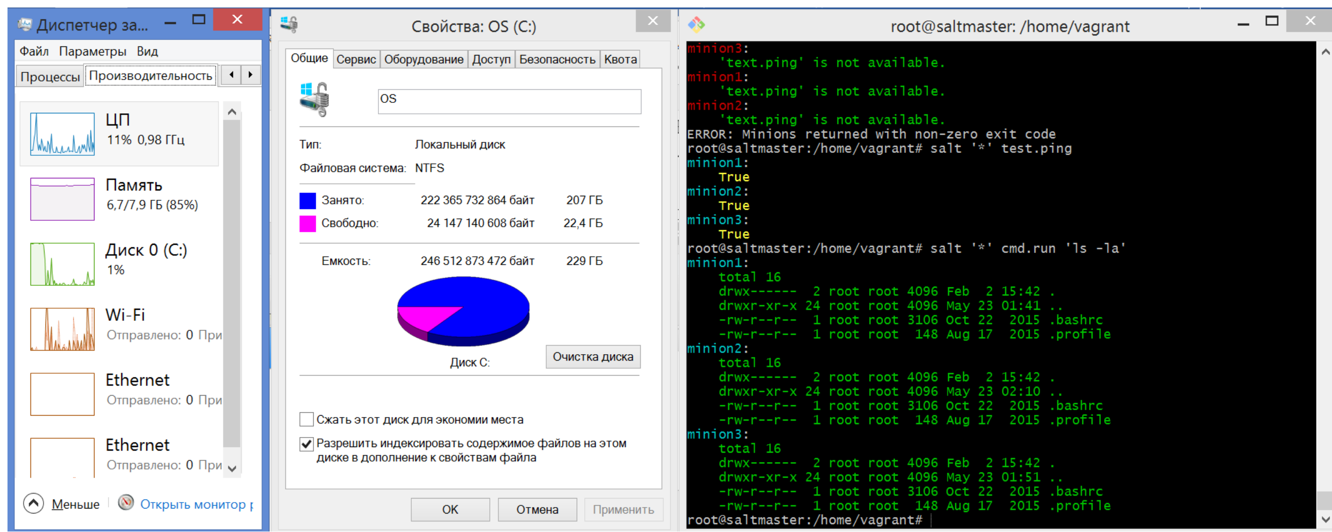

Рис. 4.2 – Нагруженість системи при розгортуванні 4 мікросервісів

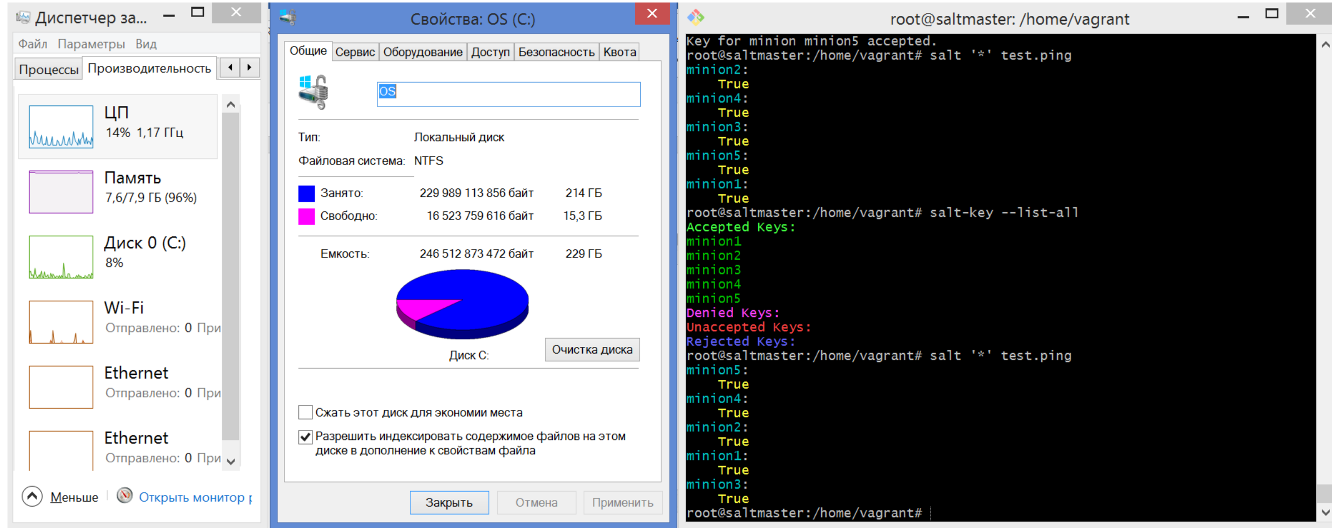

Рис. 4.3 – Нагруженість системи при розгортуванні 6 мікросервісів

Судячи з представлених даних, на кожен мікросервіс в середньому відводилося майже 3.6 Гб пам'яті. При більшій кількості розгорнутих виконувачів, обчислювальна машина починала сильно зависати, що робило роботу на ній неможливою.

4.2 Представлення DFVM в якості мікросервісу

Завдяки системам Vagrant та SaltStack, ми отримаємо зручний інструментарій для поширення файлів та даних між мікросервісами-виконувачами. У кожного з них є спільна папка на машині-хості. Для оновлення коду системи потоками даних у кожному міньйоні, було складено такий виконавчий файл:

```
salt '*' cmd.run 'sudo mount -t vboxsf vagrant test'
salt '*' cmd.run 'sudo cp -a test/dfvm /root/'
salt '*' cmd.run 'sudo umount test'
```

Рис. 4.4 – Виконавчий файл для оновлення міньйонами версії системи керування потоками даних

Ці три команди повідомляють міньйонам, що треба під'єднати зовнішню папку хоста, скопіювати код в кореневу папку, та від'єднати зовнішню спільну папку аби випадково не видалити якісь файли через міньйонів у майбутньому. Внесення змін у систему керування потоками даних відбувається на хост-комп'ютері.

```
root@saltmaster:/home/vagrant/workspace# salt '*' cmd.run 'ls dfvm'
minion4:
    Debug
    dfvm
    readme.txt
    src
minion5:
    Debug
    dfvm
    readme.txt
    src
minion1:
    Debug
    dfvm
    readme.txt
    src
minion2:
    Debug
    dfvm
    readme.txt
    src
minion3:
    Debug
    dfvm
    readme.txt
    src
root@saltmaster:/home/vagrant/workspace# |
```

Рис. 4.5 – Перевірка наявності системи керування потоками даних в усіх міньйонів

4.3 Порівняльна характеристика

Вирахуємо час вирішення примітивних обчислень.

Для 5 міньйонів час обчислень на мікросервісній архітектурі з системою керування потоками даних дорівнює 35 мілісекунд. Час на саме обчислення ~ 1 млс. Час на передачу команди між міньйонами – 588 млс.

```
root@saltmaster:/home/vagrant/workspace# ./work_all.sh
1527052670052
minion5:
    Dataflow Virtual Machine, dfvm (c) 2015, 2016, 2017
    Start time = 1527052670585
    Result 966
    End time = 1527052670585
minion1:
    Dataflow Virtual Machine, dfvm (c) 2015, 2016, 2017
    Start time = 1527052670595
    Result 966
    End time = 1527052670595
minion2:
    Dataflow Virtual Machine, dfvm (c) 2015, 2016, 2017
    Start time = 1527052670610
    Result 966
    End time = 1527052670610
minion3:
    Dataflow Virtual Machine, dfvm (c) 2015, 2016, 2017
    Start time = 1527052670607
    Result 966
    End time = 1527052670607
minion4:
    Dataflow Virtual Machine, dfvm (c) 2015, 2016, 2017
    Start time = 1527052670619
    Result 966
    End time = 1527052670620
1527052670811
```

Рис. 4.6 – Обчислення даних на 5 DFVM мікросервісах

Для 3 міньйонів час обчислень на мікросервісній архітектурі з системою керування потоками даних дорівнює 3 мілісекунд. Час на саме обчислення ~ 1 млс. Час на передачу команди між міньйонами – 550 млс.

```
root@saltmaster:/home/vagrant/workspace# ./work_3.sh
1527052777592
minion1:
    Dataflow Virtual Machine, dfvm (c) 2015, 2016, 2017
    Start time = 1527052778199
    Result 966
    End time = 1527052778199
minion2:
    Dataflow Virtual Machine, dfvm (c) 2015, 2016, 2017
    Start time = 1527052778202
    Result 966
    End time = 1527052778202
minion3:
    Dataflow Virtual Machine, dfvm (c) 2015, 2016, 2017
    Start time = 1527052778199
    Result 966
    End time = 1527052778199
1527052778404
```

Рис. 4.7 – Обчислення даних на 3 DFVM мікросервісах

Для 1 міньйона час обчислень дорівнює ~ 1 млс. Час на передачу команди між міньйонами – 511 млс.

```
root@saltmaster:/home/vagrant/workspace# ./work_1.sh
1527052822697
minion1:
    Dataflow Virtual Machine, dfvm (c) 2015, 2016, 2017
    Start time = 1527052823208
    Result 966
    End time = 1527052823208
1527052823384
```

Рис. 4.8 – Обчислення даних на 1 DFVM мікросервісі

Розглянемо монолітну архітектуру з декількома потоками. Код програми:

```
void main() {
#pragma omp parallel num_threads(THREADS)
    {
        std::cout << "\nMonolith multithreading architecture (openmp)\n";
        SYSTEMTIME time;
        GetSystemTime(&time);
        LONG time_ms = (time.wSecond * 1000) + time.wMilliseconds;
        std::stringstream stream;
        stream << "Start time = " << time_ms << std::endl;
        std::cout << stream.str();
        int a, b, c, d, e, f, g;
        a = 1;
        b = 20;
        c = 49;
        d = 3;
        e = sum(a, b);
        f = sub(c, d);
        g = mul(e, f);
        time_ms = (time.wSecond * 1000) + time.wMilliseconds;
        stream << "Result " << g << "\nEnd time = " << time_ms << std::endl;
        std::cout << stream.str();
    }
    system("PAUSE");
    return;
}
```

Рис. 4.9 – Код монолітної багатопотокової програми

При 5 потоках час всього обчислення дорівнює 22 мілісекунд. Час на саме обчислення ~ 1 млс.

```
C:\Users\Administrator\source\repos\Project1\Debug\Project...

Monolith multithreading architecture (openmp)

Monolith multithreading architecture (openmp)

Monolith multithreading architecture (openmp)

Monolith multithreading architecture (openmp)

Monolith multithreading architecture (openmp)
Start time = 55838
Start time = 55843
Start time = 55841
Start time = 55847
Start time = 55860
Start time = 55838
Result 966
End time = 55838
Start time = 55843
Result 966
End time = 55843
Start time = 55841
Result 966
End time = 55841
Start time = 55847
Result 966
End time = 55847
Start time = 55860
Result 966
End time = 55860
Для продолжения нажмите любую клавишу . . .
```

Рис. 4.10 – Обчислення даних на 5 потоках

При 3 потоках час всього обчислення дорівнює 11 мілісекунд. Час на саме обчислення ~ 1 млс.

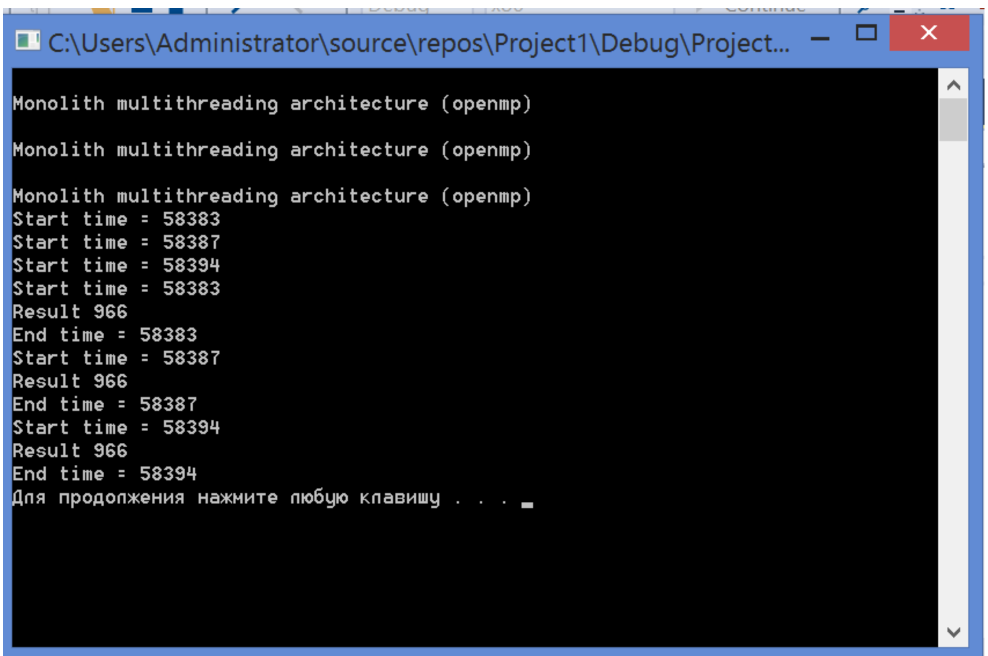

Рис. 4.11 – Обчислення даних на 3 потоках

При 1 потоку час всього обчислення дорівнює ~ 1 млс.

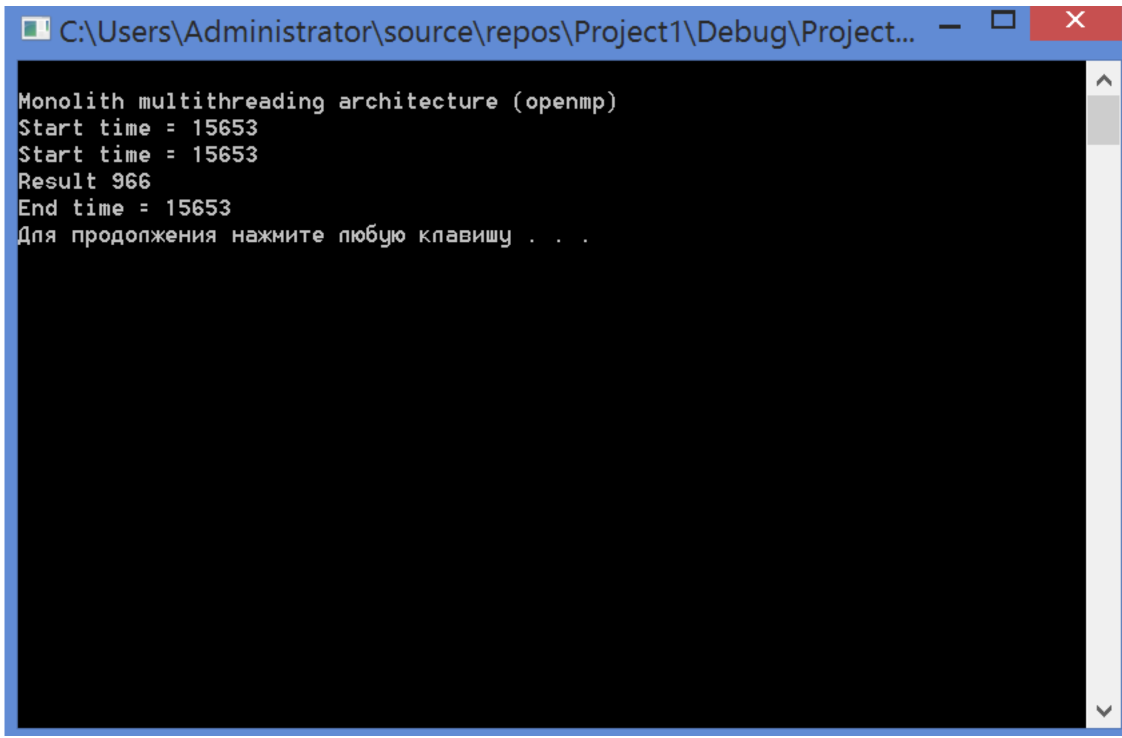

Рис. 4.12 – Обчислення даних на 1 потоці

Розглянемо мікросервісну архітектуру с фон-неймaнівськими мікросервісами.

При 5 сервісах час всього обчислення дорівнює 28 мілісекунд. Час на саме обчислення ~ 1 млс. Час на передачу команди між міньйонами –953 млс.

```
root@saltmaster:/home/vagrant/workspace# ./work_m_all.sh
1527057519270
minion4:

    Monolith multithreading architecture (openmp)
    Start time = 1527057520015
    Result 966
    End time = 1527057520015
minion1:

    Monolith multithreading architecture (openmp)
    Start time = 1527057520030
    Result 966
    End time = 1527057520030
minion2:

    Monolith multithreading architecture (openmp)
    Start time = 1527057520034
    Result 966
    End time = 1527057520034
minion5:

    Monolith multithreading architecture (openmp)
    Start time = 1527057520043
    Result 966
    End time = 1527057520043
minion3:

    Monolith multithreading architecture (openmp)
    Start time = 1527057520031
    Result 966
    End time = 1527057520031
1527057520342
```

Рис. 4.13 – Обчислення даних на 5 мікросервісах фон Неймана

При 3 сервісах час всього обчислення дорівнює 34 мілісекунд. Час на саме обчислення ~ 1 млс. Час на передачу команди між міньйонами – 846 млс.

```
root@saltmaster:/home/vagrant/workspace# ./work_m_3.sh
1527057557434
minion1:

    Monolith multithreading architecture (openmp)
    Start time = 1527057558280
    Result 966
    End time = 1527057558280
minion2:

    Monolith multithreading architecture (openmp)
    Start time = 1527057558314
    Result 966
    End time = 1527057558314
minion3:

    Monolith multithreading architecture (openmp)
    Start time = 1527057558311
    Result 966
    End time = 1527057558311
1527057558546
```

Рис. 4.14 – Обчислення даних на 3 мікросервісах фон Неймана

При 1 сервісі час всього обчислення дорівнює 34 мілісекунд. Час на саме обчислення ~ 1 млс. Час на передачу команди між міньйонами – 745 млс.

```
root@saltmaster:/home/vagrant/workspace# ./work_m_1.sh
1527057580337
minion1:

    Monolith multithreading architecture (openmp)
    Start time = 1527057581290
    Result 966
    End time = 1527057581290
1527057581561
```

Рис. 4.15 – Обчислення даних на 1 мікросервісі фон Неймана

4.4 Висновки до розділу

На проведених тестах реалізована архітектура показала кращі результати ніж архітектура з системою фон Неймана, однак гірші за монолітний підхід. Слід зазначити, що при монолітному підході паралелізація проекту напряму залежить від кількості ядер обчислювальної техніки

5 РОЗРОБЛЕННЯ СТАРТАП-ПРОЕКТУ «МІКРОСЕРВІСНА АРХІТЕКТУРА У СИСТЕМАХ КЕРУВАННЯ ПОТОКАМИ ДАНИХ»

5.1 Опис ідеї проекту

Розділ має на меті проведення маркетингового аналізу стартап проекту “Мікросервісна архітектура у системах керування потоками даних” задля визначення принципової можливості його ринкового впровадження та можливих напрямів реалізації цього впровадження.

Метою розділу є формування інноваційного мислення, підприємницького духу та формування здатностей щодо оцінювання ринкових перспектив і можливостей комерціалізації основних науково-технічних розробок, сформованих у попередній частині магістерської дисертації у вигляді розроблення концепції стартап-проекту “ Мікросервісна архітектура у системах керування потоками даних ” в умовах висококонкурентної ринкової економіки глобалізаційних процесів.

Опис стартап-проекту “Мікросервісна архітектура у системах керування потоками даних” наведено у Таблиці 5.1.

Таблиця 5.1. Опис ідеї стартап-проекту

Зміст ідеї	*Напрямки застосування*	*Вигоди для користувача*
Створення та розгортання мікросервісної архітектури на основі системи керування потоками даних.	1. Перетворення монолітних систем у мікросервісні.	Мікросервісна архітектура надає більшої гнучкості системі та зручності у впровадженні змін логіки, дизайну тощо.
	2. Використання системи керування потоками даних в якості мікросервісів мікросервісної архітектури.	Системи керування потоками даних можуть прискорити роботу всієї архітектури.

Отже, проект “Мікросервісна архітектура у системах керування потоками даних” може бути використаним як покращення вже існуючого підходу розробки комп’ютерних систем.

Таблиця 5.2 – Визначення сильних, слабких та нейтральних характеристик ідеї проекту

№ п/п	*Техніко-економічні характерис-тики ідеї*	*(потенційні) товари/концепції конкурентів*				*W (слабка сторона)*	*N (нейтр а- льна сторона)*	*S (сильна сторона)*
		Мій проект	*Конкурент 1*	*Конкурент2*	*Конкурент 3*			
1.	Форма виконання	Мікросервіс + система керування потоками даних	Моноліт + система керування потоками даних	Мікросервіс + система фон Неймана	Моноліт + система фон Неймана			+
2.	Собівартість	Висока	Висока	Низька	Низька	+		
3.	Кросплатформність	Так	Ні	Так	Ні			+
4.	Продуктивність	Дуже висока	Висока	Висока	Низька			+
5.	Швидкість	Дуже висока	Висока	Висока	Низька			+

Сильними сторонами проекту є форма виконання у вигляді мікросервісної архітектури та долучанням систем керування потоками даних, кросплатформність, продуктивність та швидкість обробки матеріалу. Слабкою стороною є собівартість, адже така система буде дорожчою за інші через наявність систем керування потоками даних. З таблиці видно, що запропоноване рішення є найбільш ефективним поміж конкурентів, а також тип подачі інформації є найкращим серед конкурентів, що доводить конкурентоспроможність даного проекту.

5.2 Технологічний аудит ідеї проекту

В межах даного підрозділу необхідно провести аудит технології, за допомогою якої можна реалізувати ідею проекту (технології створення товару).

Таблиця 5.3 - Технологічна здійсненність ідеї проекту

№ п/п	*Ідея проекту*	*Технології її реалізації*	*Наявність технології*	*Доступність технології*
1.	Створення мікросервісної архітектури на основі системи керування потоками даних	Docker	Наявна	Безкоштовно, доступна, нульове поширення
		SaltStack	Наявна	Умовно платна, доступна, середнє поширення, підходить по технологіям
		Amazon WebServices	Наявна	Умовно платна, доступна, широко використовується, слабко підходить по технологіям
		Власна система	Потрібна розробка	Безкоштовна, доступна, нульове поширення, ідеально підходить по технологіям

Обрана технологія реалізації ідеї проекту – SaltStack, оскільки розробка власного двигуну майже не під силу одній людині (комерційні ігрові двигуни розробляються сотнями людей). До того ж, SaltStack майже весь функціонал задарма. Docker та Amazon являють собою гарну альтернативу, проте коштують дорожче.

5.3 Аналіз ринкових можливостей

Визначення ринкових можливостей, які можна використати під час ринкового впровадження проекту, та ринкових загроз, які можуть перешкодити реалізації проекту, дозволяє спланувати напрями розвитку проекту із урахуванням стану ринкового середовища, потреб потенційних клієнтів та пропозицій проектів-конкурентів.

Спочатку проводимо аналіз попиту: наявність попиту, обсяг, динаміка розвитку ринку.

Таблиця 5.4 - Попередня характеристика потенційного ринку стартап-проекту

№ п/п	*Показники стану ринку (найменування)*	*Характеристика*
1.	Кількість головних гравців, од	3
2.	Загальний обсяг продаж, грн/ум.од	20000 грн./ум.од
3.	Динаміка ринку (якісна оцінка)	Зростає
4.	Наявність обмежень для входу (вказати характер обмежень)	Немає
5.	Специфічні вимоги до стандартизації та сертифікації	Немає
6.	Середня норма рентабельності в галузі (або по ринку), %	R = (3000000 * 100) / (1000000 * 12) = 25%

Виходячи з таблиці 5.4, можна зробити висновки, що на ринку небагато головних гравців, що дозволяє сконцентрувати зусилля на порівнянні свого продукту з їх слабкостями. Позитивна динаміка ринку говорить про актуальність розробки такого проекту и його необхідність на ринку.

Таблиця 5.5 - Характеристика потенційних клієнтів стартап-проекту

№ п/п	*Потреба, що формує ринок*	*Цільова аудиторія (цільові сегменти ринку)*	*Відмінності у поведінці різних потенційних цільових груп клієнтів*	*Вимоги споживачів до товару*
1.	Прискорення роботи системи та потреба у модульності.	Будь-які сфери діяльності, у яких використовують мікросервісну архітектуру	Цільова група, відмінностей між групами не має, всі можуть використовувати продукт	Рішення повинне бути зручним у користуванні, надійним, ефективним

Судячи з таблиці 5.5, на ринку є потреба в наведеному продукті, оскільки різниця між швидкостями роботи мікросервісних архітектур є невисокою. Продукт може бути використаний для будь-якої групи населення, що зацікавлена в даній сфері так само як і у випадку звичайних мікросервісних архітектур.

Таблиця 5.6 - Фактори загроз

№ п/п	*Фактор*	*Зміст загрози*	*Можлива реакція компанії*
1.	Конкуренція	Вихід на ринок великої компанії	вихід з ринку; запропонувати великій компанії поглинути себе; передбачити додаткові переваги власного сервісу для того, щоб повідомити про них саме після виходу міжнародної компанії на ринок

2.	Матеріальна частина	Подорожчання послуг	Орієнтація на якість послуг, виграш на масштабах, пошук інших, більш доступних технологій
3.	Маловідомість	Малопоширенність відомостей про даний підхід	Реклама, публікації

Виходячи з таблиці можна зробити висновок, що є небагато ризиків зв'язаних с заснуванням компанії, серед найбільш ймовірних – малопоширенність відомостей про даний підхід та подорожчання послуг обраної технології, проте було розроблено стратегії боротьби із цими ризиками.

Таблиця 5.7 - Фактори можливостей

№ п/п	*Фактор*	*Зміст можливості*	*Можлива реакція компанії*
1.	Зростання можливостей потенційних клієнтів	Зростання фінансування у підприємств, які займаються предметами побуту	Запропонувати їм свої послуги
2.	Зниження довіри до конкурента	У конкурента було помічено негативні відгуки	При виході на ринок звертати увагу покупців на надійність нашого сервісу
3.	Залучення компаній	Люди будуть охочіше користуватися нашим продуктом, якщо будему співпрацювати зі знайомими ним компаніями	Надавати ексклюзиву інформацію компаніям

Є багато можливостей, котрі можна використати для поліпшення позицій компанії на ринку. Найбільш цікаво виглядає варіант залучення мікросервісних компаній до розробки, що дозволить як покращити якість сервісів, так і отримати більшої популярності на ринку.

Надалі проводиться аналіз пропозиції: визначаються загальні риси конкуренції на ринку.

Таблиця 5.8 - Ступеневий аналіз конкуренції на ринку

Особливості конкурентного середовища	*В чому проявляється дана характеристика*	*Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентоспроможною)*
1. Вказати тип конкуренції: - досконала	Існує 3 фірми-конкуренти на ринку	Врахувати ціни конкурентних компаній на початкових етапах створення бізнесу, реклама (вказати на конкретні переваги перед конкурентами)
2. За рівнем конкурентної боротьби: - міжнародний	Всі компанії з інших країн	Додати можливість вибору системи керування потоками даних
3. За галузевою ознакою: - внутрішньогалузева	Конкуренти мають ПЗ, який використовується лише всередині даної галузі	Створити основу ПЗ таким чином, щоб можна було легко його переробити для використання у інших галузях
4. Конкуренція за видами товарів: - товарно-видова	Види товарів є однаковими, а саме – мікросервісна архітектура на основі системи керування потоками даних	Створити ПЗ, враховуючи недоліки конкурентів
5. За характером конкурентних переваг: - нецінова	Вдосконалення технології створення ПЗ, щоб собівартість була нижчою	Використання менш дорогих технологій для розробки, ніж використовують конкуренти
6. За інтенсивністю: - не марочна	Бренди відсутні	-

Ступеневий аналіз конкуренції на ринку наведений в таблиці 5.8 показує, що всі компанії-конкуренти з інших країн, що означає що український ринок фактично вільний для запропонованого продукту.

Методикою виділяються п'ять сил, які визначають рівень конкуренції, і, отже, привабливості ведення бізнесу в конкретній галузі.

Таблиця 5.9 – Аналіз конкуренції в галузі за М. Портером

Складові аналізу	*Прямі конкуренти в галузі*	*Потенційні конкуренти*	*Постачальники*	*Клієнти*	*Товари-замінники*
	Навести перелік прямих конкурентів	*Визначити бар'єри входження в ринок*	*Визначити фактори сили постачальників*	*Визначити фактори сили споживачів*	*Фактори загроз з боку замінників*
Висновки	Існує 3 конкуренти на ринку. Найбільш схожим за виконанням є конкурент 2, так як його рішення також представлене у вигляді мікросервісів.	Так, можливості для входу на ринок є, оскільки в Україні подібних компаній немає	Постачальники відсутні	Важливим для користувача є зручність у користуванні, ефективність та швидкість системи	Товари-замінники можуть зробити ефективнішу програму

Аналіз конкуренції в галузі за М. Портером для даного проекту дозволяє побачити можливі перешкоди з боку конкурентів. В даному випадку, замінники

можуть зробити більш ефективну систему, що дозволить їм випередити представлений проект. З огляду на це, потрібно буде постійно покращувати представлену пропозицію.

На основі аналізу конкуренції із урахуванням характеристик ідеї проекту, вимог споживачів до товару та факторів маркетингового середовища визначається та обґрунтовується перелік факторів конкурентоспроможності (табл. 4.10).

Таблиця 5.10 - Обґрунтування факторів конкурентоспроможності

№ п/п	*Фактор конкурентноспроможності*	*Обґрунтування (наведення чинників, що роблять фактор для порівняння конкурентних проектів значущим)*
1.	Швидкість	Наявність в архітектурі системи керування потоками даних має значно прискорити процес обробки іеформації
2.	Простота оркестрації	Інтерфейс програми оркестрування сервісами інтуітивно зрозумілий кожному

Згідно з таблицею 5.10, можна побачити, що інтерфейс програми оркестрування сервісами інтуітивно зрозумілий кожному, що дозволить вийти на інший рівень розуміння студентам представленого курсу.

За визначеними факторами конкурентоспроможності (табл. 4.10) проводиться аналіз сильних та слабких сторін стартап-проекту (табл. 4.11). Фінальним етапом ринкового аналізу можливостей впровадження проекту є складання SWOT-аналізу (матриці аналізу сильних (Strength) та слабких (Weak) сторін, загроз (Troubles) та можливостей (Opportunities) на основі виділених ринкових загроз та можливостей, та сильних і слабких сторін.

Таблиця 5.11 - Порівняльний аналіз сильних та слабких сторін проекту

№ п/п	*Фактор конкурентноспроможності*	*Бали 1-20*	*Рейтинг товарів-конкурентів у порівнянні з нашим підприємством*						
			-3	*-2*	*-1*		*1*	*2*	*3*
.	Простота оркестрації	15			+				
.	Швидкість	20		+					

Перелік ринкових загроз та ринкових можливостей складається на основі аналізу факторів загроз та факторів можливостей маркетингового середовища. Ринкові загрози та ринкові можливості є наслідками (прогнозованими результатами) впливу факторів, і, на відміну від них, ще не є реалізованими на ринку та мають певну ймовірність здійснення.

Таблиця 5.12 – SWOT-аналіз стартап-проекту

Сильні сторони: простота оркестрації, швидкість обробки інформації.	Слабкі сторони: слабка рекламна компанія
Можливості: у конкурента 1 виявлена проблема із надійністю ПЗ, додаткове фінансування для розповсюдження даної технології	Загрози: конкуренція, зміна потреб користувачів, зміна популярності

На основі SWOT-аналізу розробляються альтернативи ринкової поведінки (перелік заходів) для виведення стартап-проекту на ринок та орієнтовний оптимальний час їх ринкової реалізації з огляду на потенційні проекти конкурентів, що можуть бути виведені на ринок. Визначені альтернативи аналізуються з точки зору строків та ймовірності отримання ресурсів.

Таблиця 5.13 – Альтернативи ринкового впровадження стартап-проекту

№ п/п	*Альтернатива (орієнтовний комплекс заходів) ринкової поведінки*	*Ймовірність отримання ресурсів*	*Строки реалізації*
1.	Створення на основі університету	80%	6 місяців
2.	Створення власної незалежної компанії	30%	12 місяців

З означених альтернатив обирається та, для якої: а) отримання ресурсів є більш простим та ймовірним; б) строки реалізації – більш стислими. Тому обираємо альтернативу 1.

5.4 РОЗРОБКА РИНКОВОЇ СТРАТЕГІЇ ПРОЕКТУ

Розроблення ринкової стратегії першим кроком передбачає визначення стратегії охоплення ринку: опис цільових груп потенційних споживачів.

Таблиця 5.14 - Вибір цільових груп потенційних споживачів

№ п/п	*Опис профілю цільової групи потенційних клієнтів*	*Готовність споживачів сприйняти продукт*	*Орієнтовни й попит в межах цільової групи (сегменту)*	*Інтенсивніст ь конкуренції в сегменті*	*Простота входу у сегмент*
1.	Бізнес сфера	Можливість покращити роботу вже	Великий	Існує 3 конкуренти, які надають	Ефективність, простота оркеструання

		задіяної системи		схожі, але гірші рішення. Також усі вони знаходяться за межами країни.	сервісів, актуальність
2.	Зацікавлені люди (нові в галузі)	Можливість потрапити в бажану галузь	Великий		Ефективність, простота оркеструання сервісів, актуальність

Виходячі з таблиці, можна зробити висновок, що найбільш доцільним буде вибрати такі цільові групи, як бізнес сфера та нові зацікавлені люди.

За результатами аналізу потенційних груп споживачів (сегментів) автори ідеї обирають цільові групи, для яких вони пропонуватимуть свій товар, та визначають стратегію охоплення ринку.

Таблиця 5.15 – Визначення базової стратегії розвитку

№ п/п	*Обрана альтернатива розвитку проекту*	*Стратегія охоплення ринку*	*Ключові конкурентоспроможні позиції відповідно до обраної альтернативи*	*Базова стратегія розвитку*
1.	Розвиток обраної архітектури на основі іншої системи керування потоками даних	Ринкове позиціонування	Ефективність, простота у оркеструванні, простота доступу	Диференціація

Виходячі з таблиці, можна побачити, що було обрано ринкове позиціонування як стратегію охоплення ринку. Також, ключовими позиціями конкурентоспроможності, при виборі альтернативи в вигляді обрання іншої

системи керування потоками даних, буде ефективність, простота у оркеструанні та простота доступу.

Таблиця 5.16 - Визначення базової стратегії конкурентної поведінки

№ п/п	*Чи є проект «першопрохідцем» на ринку?*	*Чи буде компанія шукати нових споживачів, або забирати існуючих у конкурентів?*	*Чи буде компанія копіювати основні характеристики товару конкурента, і які?*	*Стратегія конкурентної поведінки*
1.	На українському – Так На міжнародному - Ні	Так	Ні, не буде	Зайняття конкурентної ніші

Проект є першопрохідцем на українському ринку, це водночас і добре (через те, що нема конкурентів) і погано (через те, що нема прихильників даного підходу). В будь якому разі, планується не копіювати характеристики міжнародних конкурентів, а знайти свій шлях на цьому ринку.

На основі вимог споживачів з обраних сегментів до постачальника (стартап-компанії) та до продукту, а також в залежності від обраної базової стратегії розвитку та стратегії конкурентної поведінки розробляється стратегія позиціонування, що полягає у формуванні ринкової позиції (комплексу асоціацій), за яким споживачі мають ідентифікувати торгівельну марку/проект.

Таблиця 5.17 - Визначення стратегії позиціонування

№ п/п	*Вимоги до товару цільової аудиторії*	*Базова стратегі я розвитку*	*Ключові конкурентоспроможні позиції власного стартап- проекту*	*Вибір асоціацій, які мають сформувати комплексну позицію власного проекту (три ключових)*

1.	Ефективність, простота у оркеструванні, простота доступу	Диференціація	Простота оркеструванняь та ефективність дозволить здобути прихільність клієнтів	Ефективність, простота оркестрування та швидкість

В таблиці визначена стратегія позиціонування продукту, визначено вимоги, головною із яких є ефективність і простота оркестрування, що дозволить здобути прихільність нових клієнтів.

5.5 Розробка маркетингової програми

Першим кроком є формування маркетингової концепції товару, який отримає споживач. Для цього у табл. 5.18 потрібно підсумувати результати попереднього аналізу конкурентоспроможності товару.

Таблиця 5.18 - Визначення ключових переваг концепції потенційного товару

№ п/п	*Потреба*	*Вигода, яку пропонує товар*	*Ключові переваги перед конкурентами (існуючі або такі, що потрібно створити)*
1.	Ефективніть	Розроблена архітектура надасть більшу ефекствивність	Перевага у ефективності
2.	Простота оркестрації	Оркестрування сервісів буде зручним та інтуітивно зрозумілим кожному	Користувачі мають зручний інтерфейс для взаємодії з системою

Ключовими перевагами потенційного товару, згідно таблиці 5.18, є ефективність так простота оркестрації, що виділяються перед конкурентами.

Надалі розробляється трирівнева маркетингова модель товару: уточнюється ідея продукту та/або послуги, його фізичні складові, особливості процесу його надання (табл. 5.19).

Таблиця 5.19 - Опис трьох рівнів моделі товару

<table>
<tr><th>Рівні товару</th><th colspan="3">Сутність та складові</th></tr>
<tr><td>I. Товар за задумом</td><td colspan="3">Мікросервісна архітектура у системах керування потоками даних</td></tr>
<tr><td rowspan="5">II. Товар у реальному виконанні</td><td>Властивості/характеристики</td><td>М/Нм</td><td>Вр/Тх /Тл/Е/Ор</td></tr>
<tr><td>Простота викладання, ефективність, простота доступу</td><td>НЕ МАТЕРІАЛЬНА</td><td>ОРГАНІЗАЦІЙНА</td></tr>
<tr><td colspan="3">Якість: згідно до стандарту ISO 4444 буде проведено тестування</td></tr>
<tr><td colspan="3">Маркування відсутнє</td></tr>
<tr><td colspan="3">Моя компанія: “MicroDFVM”</td></tr>
<tr><td rowspan="2">III. Товар із підкріплюнням</td><td colspan="3">1 місяц пробного користування</td></tr>
<tr><td colspan="3">Постійна підтримка для користувачів</td></tr>
<tr><td colspan="4">За рахунок чого потенційний товар буде захищено від копіювання: за рахунок унікальності подачі та інформації, що дозволить завоювати репутацію.</td></tr>
</table>

Три рівні моделі описують «мікросервісну архітектуру у системах керування потоками даних», компанія буде мати назву «MicroDFVM», з додатковою підтримкою для користувацiв і пробними періодом користування.

Після формування маркетингової моделі товару слід особливо відмітити – чим саме проект буде захищено від копіювання. Захист може бути організовано за рахунок захисту ідеї товару (захист інтелектуальної власності), або ноу-хау, чи комплексне поєднання властивостей і характеристик, закладене на другому та третьому рівнях товару.

Наступним кроком є визначення цінових меж, якими необхідно керуватись при встановленні ціни на потенційний товар (остаточне визначення ціни відбувається під час фінансово-економічного аналізу проекту), яке передбачає аналіз ціни на товари-аналоги або товари субститути, а також аналіз рівня доходів цільової групи споживачів (табл. 5.20). Аналіз проводиться експертним методом.

Таблиця 5.20 - Визначення меж встановлення ціни

№ п/п	*Рівень цін на товари-замінники*	*Рівень цін на товари-аналоги*	*Рівень доходів цільової групи споживачів*	*Верхня та нижня межі встановлення ціни на товар/послугу*
1.	25000 грн	30000 грн	200000 грн	20000 грн

Наступним кроком є визначення оптимальної системи збуту, в межах якого приймається рішення (табл. 5.21).

Таблиця 5.21 - Формування системи збуту

№ п/п	*Специфіка закупівельної поведінки цільових клієнтів*	*Функції збуту, які має виконувати постачальник товару*	*Глибина каналу збуту*	*Оптимальна система збуту*
1.	Купують підписку та роблять щорічні внески для подовження	Продаж	0(напряму), 1(через одного посередника)	Власна та через посередників

Останньою складової маркетингової програми є розроблення концепції маркетингових комунікацій, що спирається на попередньо обрану основу для позиціонування, визначену специфіку поведінки клієнтів (табл. 5.22).

Таблиця 5.22 - Концепція маркетингових комунікацій

№ п/п	*Специфіка поведінки цільових клієнтів*	*Канали комунікацій, якими користуютьс я цільові клієнти*	*Ключові позиції, обрані для позиціонування*	*Завдання рекламного повідомлення*	*Концепція рекламного звернення*
1.	Локальне встановлення	Інтернет	Ефективність, простота оркестрації, швидкість	Показати переваги сервісу, у тому числі і перед конкурентами	Демо-ролик із використання
2.	Використання за допомогою сайту	Інтернет	Простота оркестрації, простота доступу	Показати переваги сервісу, у тому числі і перед конкурентами	Демо-ролик із використання

Згідно таблиці, є декілька варінтів надання послуг, кожен із яких має свою специфіку, переваги так недоліки. Кожен користувач сам зможе вибрати, що йому більш потрібно.

5.6 Висновки до розділу

У даному розділі описано економічне обґрунтування реалізації стартап-проекту на тему «Мікросервісна архітектура у системах керування потоками даних». В проекті пропонується розробка, яка дозволить створити новий тип мікросервісної архітектури, що значно покращувативе ефективність та зручність архітектури.

Данна система може бути використана у будь-якій сфері діяльності, адже немає конкретних усталених сфер використання мікросервісного підходу. Його використання залежить від потреб клієнта та його цілей.

Запропоноване рішення є найбільш ефективним поміж конкурентів, а також тип подачі інформації є найкращим серед конкурентів, що доводить конкурентоспроможність даного проекту.

Є багато можливостей, котрі можна використати для поліпшення позицій компанії на ринку. Найбільш цікаво виглядає варіант залучення інших компаній до розробки, що дозволить як покращити якість сервісів, швидкість обробки даних

Аналіз конкуренції в галузі за М. Портером для даного проекту дозволяє побачити можливі перешкоди з боку конкурентів. В даному випадку, замінники можуть зробити більш ефективну архітектуру, що дозволить їм випередити представлений проект. З огляду на це, потрібно буде постійно покращувати представлену пропозицію.

ВИСНОВКИ

Було досліджено різні види комп’ютерних систем, у чому їх відмінність, переваги, недоліки. Розглянуто, у чому полягає різниця між монолітними та мікросервісними архітектурами. Було розроблено новий тип мікросервісних архітектур, досліджено його переваги у порівнянні зі звичайними архітектурами, що побудовані на системах фон Неймана. Під час дослідження літератури було знайдено багато підходів, інструментів та фреймворків, що були розроблені для покращення роботу мікросервісів, однак даний експеримент був унікальний.

У порівнянні з системами фон Неймана, системи керування потоками даних мають один значний недолік, яким можна нехтувати в умовах роботи з мікросервісами.

Даний підхід може стати іноваційним проривом, якщо продовжити роботу над ним та виділити більше ресурсів та потужності на проведення дослідів.

Основні проблеми дослідження – низька потужність комп’ютера та відсутність використовувати велику кількість сервісів. Через постійні зависання обчислювальної машини, відбувалося постійне гальмування процесу досліджень

Також, під час виконання дипломної роботи була досліджена конкурентоспроможність даного проекту у якості стартапу. Оцінка ринку показує високий попит на дану розробку. Запропоноване рішення є найбільш ефективним поміж конкурентів, а також тип подачі інформації є найкращим серед конкурентів, що доводить конкурентоспроможність даного проекту.

ПЕРЕЛІК ПОСИЛАНЬ

1. Архітектура фон Неймана [Електронний ресурс]. – Режим доступу: https://uk.wikipedia.org/wiki/Архітектура_фон_Неймана. – Назва з екрану.
2. Джон фон Нейман [Електронний ресурс]. – Режим доступу: http://wiki.kspu.kr.ua/index.php/Джон_фон_Нейман – Назва з екрану.
3. Сарапулов В. С. Мікросервісна архітектура в системах керування потоками даних / В. С. Сарапулов // Міжнародний науковий журнал "Інтернаука". – 2018. – №20. – С. 81–101.
4. ИТ инфраструктура и телекоммуникации [Електронний ресурс]. – Режим доступу: http://www.anylogic.ru/consulting/information-and- telecommunication-networks. – Назва з екрану.
5. Архітектура фон Неймана [Електронний ресурс]. – Режим доступу: https://wikivisually.com/lang-uk/wiki/Архітектура_фон_Неймана. – Назва з екрану.
6. Гарвардська архітектура [Електронний ресурс]. – Режим доступу: https://wikivisually.com/lang-uk/wiki/Архітектура_фон_Неймана. – Назва з екрану.
7. Управління проектами [Електронний ресурс]. – Режим доступу: http://uk.wikipedia.org/wiki/Управління_проектами. – Назва з екрану.
8. Архітектура з розвинутими засобами інтерпретації [Електронний ресурс]. – Режим доступу: http://uk.wikipedia.org/wiki/Архітектура_з_розвинутими_засобами_інтерпретації. – Назва з екрану.
9. Smart Python Agent Development Environment [Електронний ресурс]. – Режим доступу: https://github.com/javipalanca/spade. – Назва з екрану.
10. Fabio Luigi Bellifemine, Giovanni Caire, Dominic Greenwood. Developing Multi-Agent Systems with JADE / Fabio Luigi Bellifemine, Giovanni Caire,

Dominic Greenwood . Wiley, 2007. –P. 1-40. – ISBN: 978-0-470-05747-6

11. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: A BDI reasoning engine. In Bordini et al. [5], chap- ter 6, pages 149–174.
12. AnyLogic [Електронний ресурс]. – Режим доступу: https://ru.wikipedia.org/wiki/AnyLogic. – Назва з екрану.
13. S. Railsback and V. Grimm. /Agent-based and Individual-based Modeling: A Practical Introduction. // Princeton University Press, 2011.
14. Odd Myklebust, Enterprise Modelling supported by Manufacturing Systems Theory: Doctoral dissertation, Norwegian University of Science and Technology Department of Production- and Quality Engineering.
15. Mark S. Fox and Michael Gruninger/ Enterprise Modeling //AI MAGAZINE, Fall 1998
16. Abar, S., Theodoropoulos, G.K., Lemarinier, P., OHare, G.M., 2017. Agent based modelling and simulation tools: A review of the state-of-art software. Computer Science Review 24, 13–33.
17. Allan, R., 2009. Survey of agent based modelling and simulation tools.
18. Baˇdicaˇ, C., Budimac, Z., Burkhard, H.D., Ivanovic, M., 2011. Software agents: Languages, tools, platforms. Computer Science and Information Systems 8, 255–298.