Міністерство освіти і науки України
Чорноморський національний університет імені Петра Могили
Факультет комп'ютерних наук
Кафедра комп'ютерної інженерії

ДОПУЩЕНО ДО ЗАХИСТУ
Завідувач кафедри,
канд. техн. наук, доцент
_____ Я. М. Крайник
« __ » _____ 2022 р.

МАГІСТЕРСЬКА РОБОТА

Галузь знань: 12 Інформаційні технології
Спеціальність: 123 Комп'ютерна інженерія
Тема: **Комплексна система управління сервісами з використанням засобів контейнеризації**

Шифр: 123 – МР.ПЗ.00 – 605.21610503

Виконав:
студент 6 курсу, групи 605,
спеціальності
123 Комп'ютерна інженерія
_____ В.Б. Важоха

Керівник:
канд. техн. наук, доцент
_____ Я. М. Крайник

Миколаїв 2022

ЗАВДАННЯ
на виконання бакалаврської роботи

НЕ ВИДАЛЯТИ цю СТОРІНКУ з файлу !!!!!!!!!!!!!!!!!

ЗАРЕЗЕРВОВАНА Сторінка 1

ця сторінка після друку буде замінена

ЗАВДАННЯ

на виконання бакалаврської роботи

НЕ ВИДАЛЯТИ цю СТОРІНКУ з файлу !!!!!!!!!!!!!!!!!

ЗАРЕЗЕРВОВАНА Сторінка 2

ця сторінка після друку буде замінена

АНОТАЦІЯ

1 сторінка !!!!

НЕ ВИДАЛЯТИ цю СТОРІНКУ з файлу !!!!!!!!!!!!!!!!!

ЗАРЕЗЕРВОВАНА Сторінка 1

ця сторінка після друку буде замінена

ABSTRACT

1 сторінка !!!!

НЕ ВИДАЛЯТИ цю СТОРІНКУ з файлу !!!!!!!!!!!!!!!!!

ЗАРЕЗЕРВОВАНА Сторінка 2

ця сторінка після друку буде замінена

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

AOA	–	Angle of Arrival
COM	–	Communications Port
COM	–	COM
GPS	–	Global Positioning System
GPSM	–	Global Product Supply Management
IDE	–	Integrated Development Environment
IT	–	Information Technology
IRQ	–	Interrupt request
LRS	–	Layout Regularisation Scheme
OS	–	Operating System
PAYT	–	Pay-As-You-throw
PDA	–	Personal Digital Assistant
UART	–	Universal Asynchronous Receive Transmitter
USB	–	Universal Serial Bus
UID	–	User identifier
VCC	–	voltage common collector
СУБД	–	Система управління базами даних

ВСТУП

На сьогоднішній день середовище інформаційних технологій э трендовою частиною сучасного бізнесу. Більшу частину своєї життедіяльності люди проводять в комп’ютерах та смартфонах, з їх допомогою ми замовляємо їжу, купуємо речі, спілкуємося з рідними, розраховуємо свій бюджет, розраховуємося в магазині, плануємо подорожі, ведемо бізнес. Інтернет, мобільні додатки та додатки для персональних комп’ютерів стали нашими вірними асистентами в повсякденному житті.

Великий відсоток ринку праці займають різноманітні компанії, що проводять діяльність з розробки ПЗ, є учасниками команди стартапу або співробітниками величезних корпорацій. Майже в кожній компанії є свій відділ тестування та розробки в якому працюють співробітники різних спеціалізацій IT-сфери. Всі ці люди поєднуються в одну чи більше команду і разом працюють над розробкою якісного продукту, який буде приємно і зручно використовувати, який може вирішити певну бізнес-проблему. Тому важливо мати дієздатні бізнес-процеси в компанії та в рамках одної команди., адже кожен співробітник тісно поєднаний з іншими. Потрібно щоб в команді була налагоджена внутрішня взаємодія виконувати свої обов’язки якісно, а й бути націленому на успішний кінцевий продукт. Важливою складовою планування та керуванням процесу розробки є використання систем управління сервісами.

На шляху поширення такого рішення через цикл розробки зустрічається низька проблем. Крім підготовки програми для використання в різних ситуаціях, також можливо зіткнутися з проблемами відстеження залежності, масштабування додатків і оновлення окремих компонентів, що не зачіпають застосунок в цілому. Docker-контейнеризація і сервіс-орієнтоване проектування вирішують ці проблеми.

Контейнеризація ж покладається на вбудовані у ОС механізми ізоляції оточень використаних сервісів. Кожен контейнер є окремою частиною з

відділеною окремою пам’яттю та власним процесорним часом. Контейнеризація / віртуалізація на рівні операційної системи – метод віртуалізації, коли ядро операційної системи підтримує кілька ізольованих примірників простору користувача замість одного. Ці сутності або контейнер з точки зору людини повністю ідентичні окремому елементу операційної системи. Для систем на базі Unix використана технологія схожа на новітню реалізацію механізму chroot. Ядро забезпечує повноцінну ізольованість контейнерів, тому застосунки з різних контейнерів не впливають один на одного.

Docker – програмний інструмент для автоматизації розгортання і управління додатками в середовищах з підтримкою контейнеризації. У своєму ядрі він дозволяє виконувати практично будь-який застосунок, що повноцінно ізольований у контейнері. Повноцінна ізоляція дозволяє запускати на одному сервері багато контейнерів одночасно з оптимальним використанням обчислювальних ресурсів.

Мета: розробити комплексну система управління з використанням засобів контейнеризації

Об’єкт: модель управління процесами для злагодженої роботи дистанційного відділу.

Предмет: засоби контейнеризації та управління додатками в середовищах з підтримкою віртуалізації.

Методи дослідження: спостереження, порівняння, вимірювання, експеримент, аналіз і синтез, абстрагування, емпіричний аналіз, метод наукової індукції.

Гіпотеза: передбачається, що впроваджена модель процесів управління сервісами ПЗ значною мірою покращить ефективність та продуктивність роботи підприємства.

Практичне значення: використання системи управління сервісами допоможе реалізувати процеси задля росту ефективності, вмотивованості,

працездатності працівників підприємства, а процес розрбки був максимально автоматизованим.

Апробація результатів магістерської роботи відбулася в рамках Всеукраїнської науково-практичної конференції молодих вчених, аспірантів та студентів «Інформаційні технології та інженерія» (м. Миколаїв, ЧНУ ім. Петра Могили).

Публікації. За результатами магістерської роботи створено публікацію у збірнику матеріалів Всеукраїнської конференції “Інформаційні технології та інженерія” [1].

Для досягнення поставленої мети необхідно вирішити такі **завдання**:

1) Розглянути особливості системи управління сервісами.

2) Проаналізувати вихідну документацію, структури та обсягу даних аналогічних апаратно-програмних комплексів.

3) Дослідити математичні методи, моделі і алгоритми, які застосовуються для обробки даних.

4) Розробити сукупність апаратної частини та програм, призначених для розробки, функціонування та модернізації технічної системи.

5) Представити прототип апаратної частини комплексу у вигляді, готовому до використання.

6) Розробити програмну частину комплексу автоматизації.

Структура та обсяг роботи. Магістерська робота складається з анотації на 2 сторінках, вступу, чотирьох розділів, висновків, переліку джерел посилання з 34 найменувань, 3 додатки на 5 сторінках. Основна частина роботи становить 79 сторінок (без додатків), серед яких 35 рис. та 4 табл.

РОЗДІЛ 1
АНАЛІТИЧНА ЧАСТИНА

1.1 Аналіз предметної області

Поняття «сервіс» широко поширене у різних сферах науки, виробництва, навіть у побуті. Розглянемо існуючі у науковій літературі визначення поняття «сервіс»:

– організована діяльність, спрямована на створення унікальних продуктів, послуг чи результатів;

– окрема підприємницька діяльність з певними цілями, часто включає вимоги щодо часу, вартості та якості досягнутих результатів (відповідно до англійської Асоціації сервіс-менеджерів);

– діяльність, яка значною мірою характеризується неповторністю умов у їх сукупності, наприклад: завдання мети; тимчасові, фінансові, людські та інші обмеження;

– розмежування з інших намірів;

– специфічна для проекту організація його здійснення (відповідно до німецького стандарту DIN 6990) [2].

Відповідно до визначення за Бегьюлі Ф., сервіс - це послідовність взаємо-пов'язаних подій, в рамках обмеженого періоду часу і спрямовані на досягнення певного результату [3].

Івасенко О.Г. дає таке визначення: «Сервіс – це обмежене за часом цілеспрямована зміна окремої системи з самого початку чітко визначеними цілями, досягнення яких визначає завершення сервісу, із встановленими вимогами до термінів, результатів, ризику, рамок витрачання коштів та ресурсів та до організаційної структури » [4].

За Фунтов В.М., сервісом називається «цілеспрямована, обмежена у часі діяльність, що здійснюється для задоволення конкретних потреб за

наявності зовнішніх та внутрішніх обмежень та використання обмежених ресурсів» [5].

Виходячи з цього, наведені визначення відрізняються досить загальним підходом до терміну «сервіс», але все ж таки дозволяють виділити суттєві особливості цього поняття.

До таких відносяться:

– наявність чітко сформульованих цілей, а також ряду технічних, економічних та інших цільових показників;

– системний характер будь-якого сервісу, тобто наявність внутрішніх та зовнішніх зв’язків між усіма складовими системи, а саме цілями, завданнями, операціями, ресурсами (включаючи людські), шуканим результатом. Це дає можливість алгоритмізації сервісу, тобто уявлення його у вигляді комплексу взаємозалежних процесів

– наявність визначених часових інтервалів (терміни початку та кінця сервісу);

– обмежені ресурси;

– певний ступінь унікальності цілей сервісу та умов його здійснення.

Таким чином, сервіс – динамічна система дій, спрямованих на отримання заданих результатів у багатокритеріальному полі протягом встановленого строку та в рамках виділених ресурсів із залученням виконавців, які володіють необхідними навичками та знаннями.

На рис.1.1 комплексна діаграма представляє сервіс як систему.

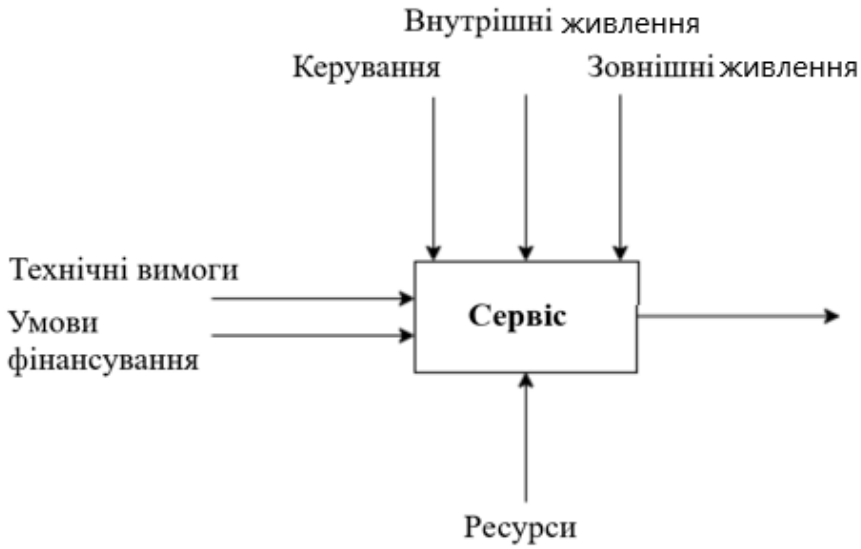


Рисунок 1.1 – Діаграма структури сервісу

Вхідними даними є технічні вимоги та умови фінансування; метою є досягнення необхідного результату. Наявність ресурсів у вигляді матеріалів, фінансів, людського ресурсу забезпечує виконання робіт. Ефективність визначається управлінням процесом реалізації сервісу [6].

До функцій управління входить розподіл ресурсів, координація виконуваної послідовності операцій, компенсація внутрішніх та зовнішніх впливів [7].

У таблиці 1.1 розкривається зміст складових елементів сервісу.

Таблиця 1.1 – Опис елементів системи «сервіс»

Елемент	Зміст
Ціль (результат)	Описуються нові продукти або послуги, які отримає замовник результаті реалізації проекту
Вартість проекту	Фінансові витрати, необхідні для виконання робіт проекту
Технічні вимоги	обсяги робіт (кількісні показники обсягу робіт проекту); термін виконання; якість (відповідність характеристик проекту та його продукції встановленим заздалегідь параметрам якості)
Ресурси	Устаткування, матеріали, персонал, програмне забезпечення, інформаційні системи; виробничі площі; фахівці та організації, залучені до виконання робіт проекту, їх кваліфікація
Внутрішні обурюючі впливу	Стиль керівництва проектом; організація проекту з погляду комунікації між основними учасниками проекту, розподілу прав, відповідальності та обов'язків; методи та засоби взаємодії між співробітниками всіх рівнів на проекті; умови праці та техніки безпеки, страхування та соціальне забезпечення та т.п.
Зовнішні обурюючі впливу	Взаємодія із замовником та конкурентами; ситуація на ринку та пов'язані з цим ризики; непередбачені обставини

1.2 Обґрунтування актуальності управління сервісами

Управління сервісами як вид професійної діяльності та як об'єкт наукових досліджень мав суттєвий розвиток у 1980-х роках, коли світова економіка виходила із кризи, при цьому зростало насичення ринку та виникала необхідність розв'язання нових, великих завдань. Саме в той час виходить перша значна робота PMBoK (A Guide to the Project Management Body of Knowledge), виконана в Project Management Institute (PMI). Дана організація на сьогоднішній день є найбільш авторитетною професійною асоціацією, яка розробляє стандарти в галузі управління сервісами. Серед інших інститутів, що займаються стандартизацією сервісного менеджменту слід назвати IPMA (International Project Management Association), OGC (The

Office of Government Commerce, стандарти PRINCE2), ISO (International Standartization Organization), APM(Association for Project Management) та інших. Слід зазначити, що, крім міжнародних стандартів, існують також національні системи стандартизації, галузеві та корпоративні.

Розвиток IT-сфери, інтенсивна робота в галузі ПЗ призвели до накопичення великого практичного досвіду ("best practice"). Комплекс таких «кращих практик», що реалізуються на різних стадіях життєвого циклу сервісу та базуються на спільній ідеології, стандарт SWEBOK [8] називає "методологія розробки програмного забезпечення". Методології, або методи розробки програмних продуктів на сьогодні областю інформаційних технологій, що найбільш швидко розвивається, так як спираються на реальні практичні знання.

Оцінюючи сучасну швидкість поширення методології управління сервісами у світі стає очевидним, що управління сервісами використовується не в виняткових випадках, а, навпаки, все частіше стає стандартизованим способом для ведення бізнесу. Все більша частка робіт у звичайних компаніях виконується як сервіси або інакше кажучи, сервіси. За оцінками експертів, у майбутньому очікується збільшення частки системного управління, зокрема в галузі стратегічного планування та розвитку організацій.

Суспільні та технологічні умови сучасного світового розвитку характеризуються постійним зростанням обсягу цифрових технологій, що ґрунтуються на використанні штучного інтелекту (ШІ), інноваційними процесами у суспільстві загалом, а також у сферах виробництва, науки, бізнесу. Зміни, які відбуваються в соціумі та в IT-області, породжують нові вимоги до якості управління сервісами, що втілюють у життя «розумні» програми: розпізнавання осіб, постановка діагнозів, безпілотне керування транспортом тощо.

Робота з розробки нейро-мереж (як математичної основи штучного інтелекту) за своїм змістом є науково-дослідною, тобто має творчий характер, і отже, структура виробничих відносин у процесі виготовлення програмного продукту не така жорстка, менш ієрархічна, причому ієрархія протягом часу створення програми може порушуватись. Це зумовлює необхідність вибору більш гнучких та стійких до змін методологій управління сервісами у компаніях, що спеціалізуються на розробці ШІ-продуктів. На сьогоднішній день найбільш поширеними та популярними методологіями є Agile-сімейство, що включає Scrum, Kanban, XP та інші методи. Але вони не забезпечують повною мірою запити управління ШІ-сервісами, як наукомісткими та великими масивами даних, що оперують. Тому сервіси віддають перевагу такому методу, як Crisp-Dm, що доказав свою ефективність при роботі з Big Data.

Сучасний ринок інформаційних технологій пропонує великий вибір платформ для автоматизованого керування сервісами. Асортимент таких систем дуже великий. Налічується близько ста автоматизованих систем керування процесом створення програмних продуктів. Найпопулярнішими з них є Trello, Jira та Asana. Кожен із пропонованих продуктів має свої переваги та недоліки. Так, Trello орієнтується на малі та короткострокові сервіси, добре зарекомендувала себе у роботі з легко формалізованими сервісами; Jira зручна у використанні у великих командах, але за умови попереднього налаштування та розширення базового набору функцій за допомогою плагінів та надбудов. Цей процес є досить трудомістким, потребує багато зусиль. Сама Jira має складну структуру та важка у розумінні для нових користувачів.

Велика різноманітність у клієнтських запитах на ринку ІТ-послуг, спектр застосування ШІ, що постійно розширюється, зумовлює існування різних видів компаній, що різняться як за кількістю співробітників, так і

виробленого продукту. Тому актуальними є питання адаптації існуючих моделей управління сервісами до конкретної компанії.

З погляду менеджменту, організована та продумана система є одним із найскладніших об'єктів управління. Це зумовлено особливостями робочого процесу співробітників, невизначеністю та важкою передбачуваністю кінцевих результатів, неможливістю повної алгоритмізації процесу пошуку нових рішень.

Таким чином, виявляється суперечність: з одного боку, об'єктивно зростає частка дослідницької роботи у сфері IT-бізнесу, посилюється необхідність постійного вдосконалення системи управління інноваційними та дослідницькими сервісами в галузі ШІ; з іншого боку, методи системи управління є недостатньо розробленими та часто не адекватними об'єкту управління. Така невідповідність призводить до того, що витрати, вкладені в сервіс, не окупається і робота, що виявляється нерентабельною.

1.3 Аналітичний огляд літератури та патентної інформації

1.3.1 Патент US10237118B2

Назва патенту – «Ефективне створення/розгортання програми для хмарної платформи розподіленого контейнера».

Управління центром обробки даних стало важливим аспектом в інформаційних технологіях (IT) і дисциплінах управління об’єктами, а також ефективне створення та випуск додатків (додатків) для використання його клієнтами. Віртуальні системи були використані для полегшення створення додатків для центру обробки даних. Однак звичайні віртуальні системи, такі як VMware, занадто важкі.

Наприклад, звичайним віртуальним системам важко підтримувати великі програми, такі як програми планування ресурсів підприємства (ERP), програми керування взаємовідносинами з клієнтами (CRM) або програми баз даних, такі як HANA. Крім того, існуючі центри обробки даних вимагають

створення та встановлення програми, наприклад, на голому металі, щоразу, коли запитується додаток. Це неефективно за часом[26].

Це розкриття надає розподілену структуру керування для додатків у центрі обробки даних, яка є легкою та ефективною за допомогою контейнерів. На рис. 1.10 представлено зразкове середовище чи архітектуру;

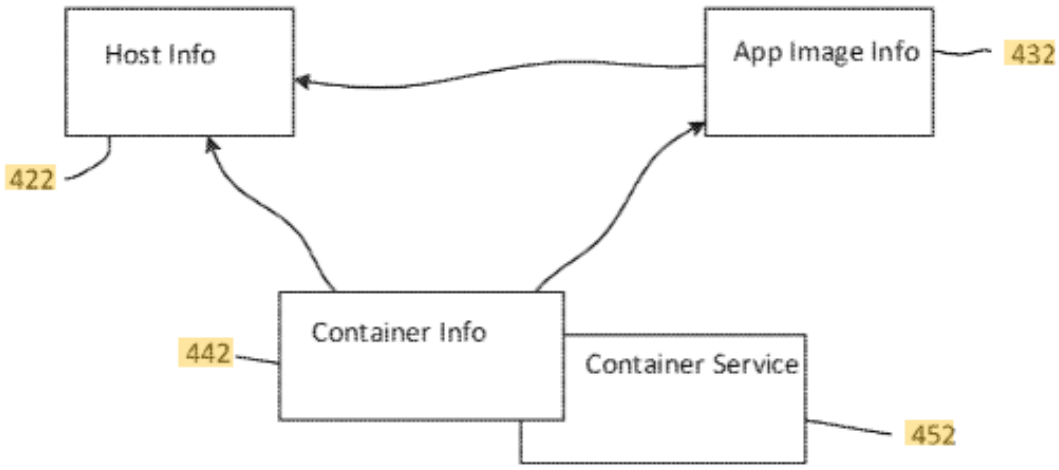


Рисунок 1.10 - Структуру керування додатків

В описанні даного патенту розкрито технологію для полегшення управління хмарним центром обробки даних і створення/розгортання додатків у хмарному центрі обробки даних. Відповідно до одного аспекту технології розкрита хмарна платформа з розподіленим контейнером.

В одному варіанті здійснення розкривається реалізований комп'ютером спосіб управління центром обробки даних. Спосіб включає надання центру обробки даних, в якому центр обробки даних містить хости для розміщення зображень додатків і менеджер хмари контейнерів для управління ресурсами центру обробки даних. Метод також включає в себе відповідь на користувача, який представляє нову програму для випуску в центр обробки даних, перевірку інформаційного файлу програми диспетчером хмари контейнерів, щоб визначити, чи нова програма вже існує в центрі обробки даних, і ініціювати збірку нової програми. якщо новий додаток не існує в

центрі обробки даних. Складання нової програми включає створення нового образу програми, налаштування контейнера нової програми, упаковку контейнера до нового образу програми, створення загалом x копій нового образу програми,

В іншому варіанті здійснення розкривається нетимчасовий машиночитаний носій, що має збережений на ньому програмний код. Збережений програмний код виконується комп’ютером для керування центром обробки даних. Виконаний метод управління включає надання центру обробки даних, в якому центр обробки даних містить хости для розміщення зображень додатків і менеджер хмари контейнерів для управління ресурсами центру обробки даних. Виконаний метод керування також включає відповідь на користувача, який представляє нову програму для випуску в центр обробки даних, перевірку інформаційного файлу програми диспетчером хмари контейнерів, щоб визначити, чи нова програма вже існує в центрі обробки даних, і ініціювати збірку новий додаток, якщо новий додаток не існує в центрі обробки даних. Складання нової програми включає створення нового образу програми, налаштування контейнера нової програми.

1.3.2 Патент US11120299B2

Назва патенту – «Встановлення та експлуатація різних процесів механізму штучного інтелекту, адаптованого до різних конфігурацій обладнання, розташованого локально та в гібридних середовищах».

У варіанті здійснення передбачено пристрій, що включає механізм штучного інтелекту (“AI”), який має декілька незалежних процесів на одній або кількох обчислювальних платформах. Механізм AI має користувальницький інтерфейс для одного або кількох користувачів в організації користувача[27].

Кілька незалежних процесів налаштовані на завантаження в одну або більше пам'яті однієї або кількох обчислювальних платформ. Одна або кілька

обчислювальних платформ розташовані на території організації користувача, що означає: і) одну або кілька обчислювальних платформ можна налаштувати для того, щоб один або кілька користувачів в організації користувача мали принаймні права адміністратора над одним або кількома обчислювальні платформи. Це дозволяє організації користувача налаштовувати апаратні компоненти однієї або кількох обчислювальних платформ, щоб діяти так, як організація користувача вибирає відповідати їхнім потребам для виконання та завантаження кількох незалежних процесів. іі) Один або кілька користувачів організації користувача можуть отримати фізичний доступ до однієї або кількох обчислювальних платформ. ііі)

Апаратні компоненти однієї або кількох обчислювальних платформ з’єднані один з одним через локальну мережу (LAN), і локальну мережу можна конфігурувати таким чином, що один або кілька користувачів в організації користувача мають право контролювати роботу локальну мережу. Кілька незалежних процесів налаштовані як набір незалежних процесів, кожен його незалежний процес загорнутий у власний програмний контейнер, так що кілька екземплярів одного незалежного процесу можуть виконуватися одночасно для масштабування для обробки дій, вибраних із групи, що складається з:

1) Виконання кількох навчальних сесій на двох або більше моделях ШІ одночасно,

2) Створення двох або більше моделей ШІ одночасно,

3) Проведення навчального сеансу на одній або кількох моделях ШІ, створюючи одну чи більше моделей ШІ одночасно,

4) Будь-яку комбінацію цих трьох, на той самий двигун AI. Перший і 4) будь-яку комбінацію цих трьох на одному механізмі AI. Перший і

У варіанті здійснення кожен незалежний процес із набору незалежних процесів загорнутий у свій власний програмний контейнер. Це включає принаймні процес інструктора та процес навчання. Процес інструктора

налаштований на виконання плану навчання, кодифікованого мовою програмування педагогічного програмного забезпечення. Процес учня налаштований на фактичне виконання базових алгоритмів навчання ШІ під час навчальної сесії. Процес інструктора і процес, що навчається, з набору незалежних процесів співпрацюють з одним або кількома джерелами даних для навчання нової моделі ШІ.

В описані даного патенту також надається спосіб встановлення кількох незалежних процесів, кожен з яких загорнутий у свій власний контейнер, на одну або кілька обчислювальних платформ. Одна або кілька обчислювальних платформ розташовані на території організації користувача.

На рис 1.11 блок-схему, що ілюструє ШІ системи і його локальну інфраструктуру обчислювальних платформ відповідно до варіанту здійснення.

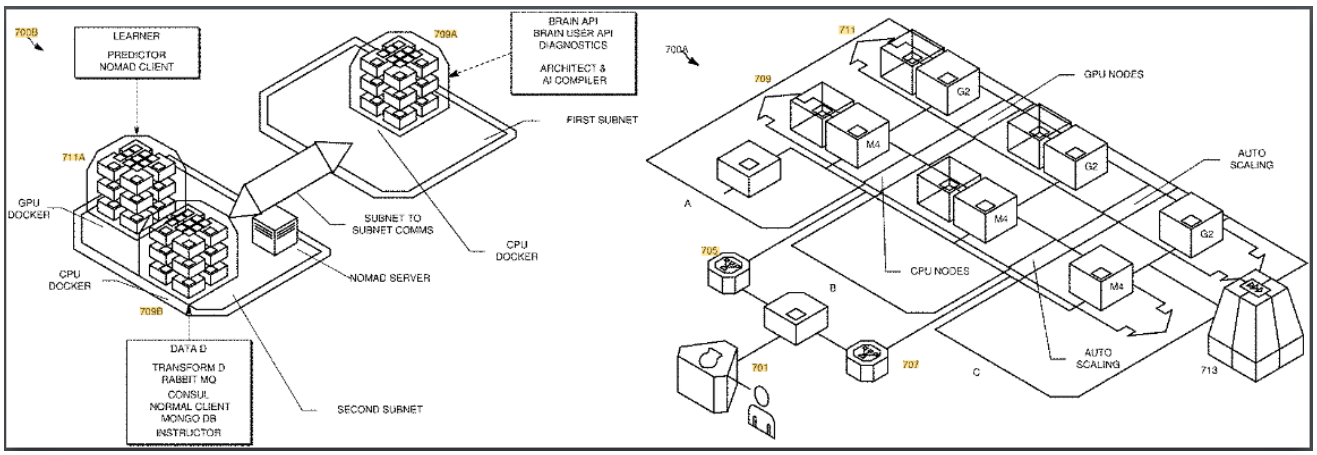


Рисунок 1.11 – Блок-схема локальної інфраструктуру обчислювальних платформ

Метод включає визначення кількості віртуальних машин, фізичних машин або як фізичних, так і віртуальних машин, доступних у кластері однієї чи кількох обчислювальних платформ, розташованих локально для створення та навчання моделей штучного інтелекту (“AI”) за допомогою однієї або більше обчислювальних платформ. Спосіб додатково включає в себе виділення мережевих адрес для машин і копіювання сценаріїв на машини для кількох незалежних процесів. Спосіб додатково включає виконання першого

сценарію, налаштованого для встановлення кількох незалежних процесів на машинах для кількох незалежних процесів. Крім того, метод додатково включає в себе виділення однієї або кількох ролей кожній машині або її вузлу в кластері, ролей для обмеження мікросервісів або незалежного процесу певним типам вузлів.

1.3.3 Наукова стаття

В науковій статті під назвою «Ін’єктивний метод отримання даних користувацького досвіду в ігрових симуляторах комп’ютерних мереж» Вінницького національного технічного університету, яку опублікували автори І. П. Малініч та В. І. Месюра було розглянуто способи організації роботи симуляторів комп’ютерних мереж, а також методи та підходи до отримання даних користувацького досвіду в них[28].

Наведено особливості застосування цієї технології в розважальних та навчальних цілях. Визначено основні цілі збору користувацьких даних в ігрових симуляторах комп’ютерних мереж, які зокрема застосовуються в олімпіадних та конкурсних змаганнях з програмування та застосування ІТ-технологій. В статті приділено увагу проблемі збору даних користувацького досвіду в додатках, для доступу до яких використовується віддалене термінальне підключення з використанням стороннього програмного забезпечення та розглянуто особливості застосування програмних інтерфейсів технологій, функціонал яких можливо використати як основу для розробки методу: технології контейнеризації користувацького середовища операційної системи LXC та User Mode Linux, технологію перехоплення пакетів на основі мережевого екрану NetFilter та віртуального комутатора Open vSwitch.

В основу нового методу покладено використання технології контейнерів операційних систем, де передбачена можливість запуску довільного коду кінцевого користувача. На відміну від ігрових симуляторів, де застосовується емуляція роботи системних команд, повнофункціональний

контейнер дає змогу організувати роботу справжнього випробувального середовища пісочниць. Для здійснення збору даних користувацького досвіду у віртуальному середовищі основний процес на хостовій машині запускає модулі-агенти всередині кожного контейнера.

Кожен модуль-агент при запуску отримує від основного процесу набір сигнатур процесів, відповідно з яких здійснюється збір даних користувацького досвіду. Сигнатури можуть бути таких типів: сигнатури файлів (хеш-суми або шлях у файловій системі), сигнатури UNIX-сокетів, сигнатури дій користувача в системі, маски повідомлень журналів та сигнатури перехоплення для мережевого екрану. Після отримання сигнатур внутрішній модуль відповідно до них запускає процеси монітору повідомлень системного журналу, монітору автентифікації користувачів, монітору процесів, а також монітору активності файлів та сокетів. Внутрішній модуль встановлює з’єднання з основним процесом на хостовій машині, через яке в режимі реального часу передаються та записуються в базу даних обліку активності (рис. 1.12) всі збіжності з сигнатурами, що виявляються моніторами.

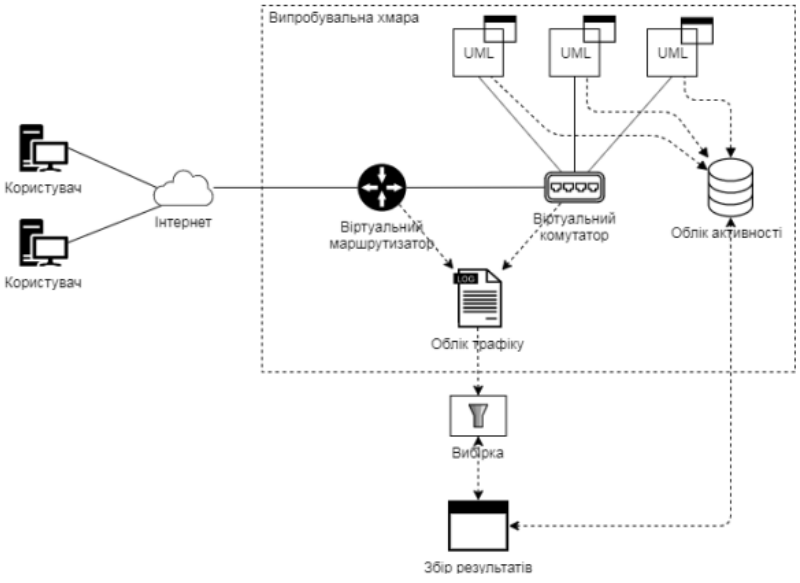


Рисунок 1.12 – Структурна схема випробувальної хмари, в якій здійснюється отримання даних користувацького досвіду

1.3.4 Наукова конференція

Згідно матеріалів наукової конференції Тернопільського національного технічного університету імені Івана Пулюя за авторством А. Луцків, Р. Луцишин було досліджено віртуалізацію на рівні операційної системи як метод віртуалізації, при якому ядро операційної системи підтримує декілька ізольованих примірників простору користувача, замість одного[27].

Ці примірники (часто звані контейнерами або зонами) з точки зору користувача повністю ідентичні реальному серверові. При організації навчального процесу доволі часто можна зустріти підхід до розгортання лабораторних стендів у вигляді образів готових віртуальних машин (Hadoop-кластери Cloudera CDH та Hortonworks HDP, MongoDB University). Проте, такі образи віртуальних машин є доволі об'ємними й їх використання є доволі ресурсоємним. У даному випадку використовується повна віртуалізація з використанням віртуальних машин Virtual Box, VM Ware. На думку авторів, доцільнішим є використання технологій контейнеризації, зокрема, використання платформи Docker.

Такий підхід передбачає наявність у студента деяких базових уявлень про контейнеризацію, проте, запуск та робота з таким лабораторним стендом є доволі простою (кількість необхідних попередніх налаштувань та кількість необхідних дій для запуску/зупинки/зберігання стану) та ефективною (з точки зору використання пам'яті й процесорного часу), швидшою у розгортанні (необхідність завантаження даних меншого об'єму), може бути використана як на робочому комп'ютері та і в хмарному сервісі.

Висновки до розділу 1

В даному розділі були проаналізовані основні методи для побудови веб-додатків, розглянуті переваги та недоліки монолітної архітектури, сервісо-орієнтованої архітектури (SOA), подійно-орієнтованої архітектури, мікросервісної архітектури. На основі цього аналізу були сформулювані наступні задачі, для подальшого дослідження оптимізації процесів

розгортання мікросервісів. основних методів використання неперервної інтеграції в мікросервісній архітектурі та аналізу способів розгортання сервісів та розгляд шаблонів проектування мікросервісів.

РОЗДІЛ 2
СТРУКТУРНО-ФУНКЦІОНАЛЬНЕ МОДЕЛЮВАННЯ СИСТЕМИ

2.1 Характеристика системи управління сервісами

Мікросервіси або архітектура мікросервісів — це архітектурний стиль, який визначає програму на набір невеликих автономних служб, змодельованих навколо бізнес-домену або бізнес-логіки програми.

Архітектура мікросервісів сьогодні стала одним із підходів до розробки передових додатків.

Принципи, на яких побудована архітектура мікросервісів:

- Масштабованість
- Доступність
- Стійкість до помилок
- Незалежність, самостійність
- Децентралізований контроль
- Несправність ізоляції
- Автоматичне виявлення послуг
- Безперервна доставка

Метою мікросервісів є збільшення швидкості випуску програми шляхом поділу програми на невеликі автономні служби, які ви можете реалізувати самостійно. Архітектура мікросервісів також створює деякі проблеми. Розглянемо основи архітектурних проблем і методи їх вирішення.

Мікросервісний підхід має кілька моделей архітектурного проектування, їх можна розділити на п’ять моделей (рис. 2.1).

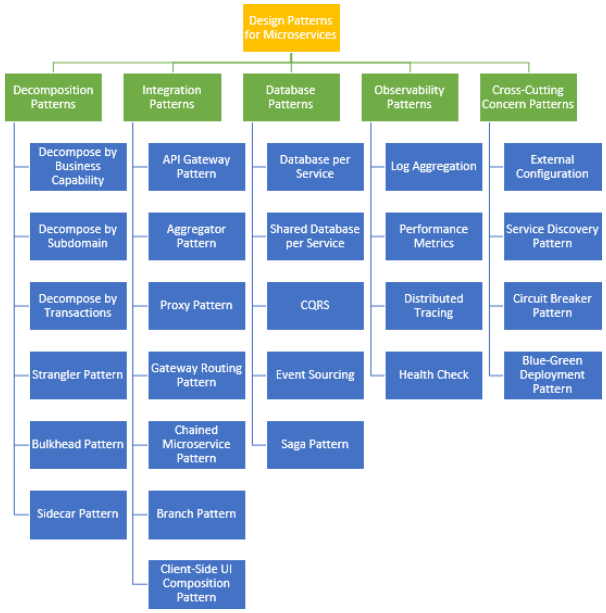


Рисунок 2.1. – Патерни мікросервісної архітектури

Слід розрізняти таке поняття, як «Бізнес-Вимога» (*Business Domain*, *бізнес-можливість*) – те що виконує організація, а це у свою чергу сприяє досягненню її бізнес-цілей. Наприклад, обробка кошика замовлень в інтернет-магазині – це бізнес-вимога, яка допомагає досягти більш широкої бізнес-цілі, що складається в забезпеченні можливості покупки товарів користувачами через інтернет. У кожногокомерційного підприємства є безліч бізнес-вимог, які разом утворюють його загальну бізнес-функцію.

Таким чином мікросервіс реалізує бізнес-можливість в цілому і повністю автоматизує її виконання. Проте можливо, що мікросервіс реалізує тільки частину бізнес-можливості і, таким чином, автоматизує її виконання лишечастково. В обох випадках область дії мікросервісу дорівнює бізнес-можливості.

Декомпонуючи систему таким чином, застосовують принцип єдиної відповідальності. Система розподіляється на основі бізнес-вимог та визначаються послуги, що задовольняють вимоги бізнесу (рис. 2.2).

Набір вимог для даного бізнесу залежить від типу бізнесу. Наприклад, робота страхової компанії зазвичай включає у себе продажі, маркетинг, обробка пропозицій і претензій, виставлення рахунків, відповідність. Отже

система, декомпозується на сервіси: сервіс продажу, сервіс маркетингу, сервіс рахунків і тд.

Рисунок 2.2 – Модель бізнес-вимоги

2.2 Метод декомпозиції субдомену

Предметно-орієнтоване проектування (*domain-driven design, DDD*) – підхід до проектування програмного забезпечення, заснований на моделюванні предметної області.

Обмеженим контекстом (*bounded context*) в предметно-орієнтованому проектуванні називається деяка частина предметної області, в межах якої терміни, поняття зберігають своє значення.

Обмежений контекст визначає частину предметної області, всередині якої контекст є однаковим. В межах одного обмеженого контексту підприємству може знадобитися виконувати кілька дій, кожна з яких, ймовірно, є бізнес-можливістю.

Декомпозиція програми з використанням бізнес-можливостей може стати хорошим початком, але можна зіткнутись із так званими «класами богів» (God Classes), розділити які буде непросто. Дизайн, на основі DDD,

визначає цей проблемний простір програми як бізнес-домен. Домен складається з декількох субдоменів (під-доменів), і кожен субдомен відповідає різній частині бізнесу.

Наприклад, бізнес-можливість включає у себе сервіс замовлень – бізнес- домен. Даний сервіс можна декомпонувати на декілька під-доменів:

– сервіс каталогу товарів;
– послуги з керування складом;
– послуги з управління замовленнями;
– послуги з управління доставкою.

Отже, декомпозиція субдомену використовується для дроблення бізнес-домену задля збереження автономності та атомарності кожного з модулів.

2.3 Шаблони проектування архітектури системи

2.3.1 Шаблон Strangler – міграція на мікросервіси

Шаблон Strangler (Ш*аблон придушення*) – покрокова міграція застарілої системи з поступовою заміною певних компонентів новими додатками і службами. Компоненти застарілої системи поступово видаляються, і з часом нова система повністю візьме на себе всі її функції, що дозволяє вивести стару систему з експлуатації.

Засоби розробки, технології розміщення і архітектури, на основі яких створюється будь-яка система, часто застарівають з плином часу. У міру додавання нових функцій і можливостей неухильно зростає складність старих додатків, що ще більше ускладнює обслуговування і додавання нових компонентів.

Але повна заміна складної системи часто пов'язана з величезними труднощами. У багатьох випадках краще переходити на нову систему поступово, зберігаючи стару систему для обробки тих можливостей, які ще не реалізовані в новій (рис. 2.3). Виконання двох різних версій однієї

програми означає, що клієнти повинні знати, в який з них знаходяться потрібні компоненти.

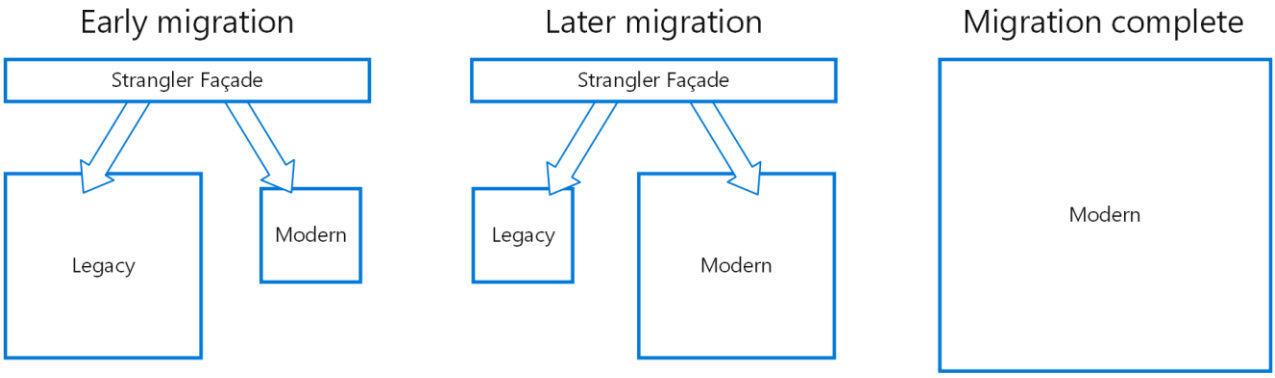


Рисунок 2.3 – Поетапний перехід на новий компонент

Згідно із шаблоном, потрібно поетапно замінювати компоненти новими додатками і службами. Створюється оболонка (фасад), яка перехоплює всі запити до застарілих систем серверної частини. Дана оболонка буде вибирати, куди направити такі запити: до застарілого компоненту додатку або до нової служби. Так можна перенести компоненти в нову систему поступово, не змінюючи інтерфейс для користувачів, які навіть не помітять, що відбувається поетапна міграція.

Дана модель дозволяє знизити ризики міграції та розподілити зусилля розробників по часу. Поки оболонка безпечно направляє користувачів до правильного додатку, можна додавати компоненти в нову систему в комфортному темпі, зберігаючи при цьому працездатність старого додатка. Через деякий час всі функції будуть перенесені в нову систему і застаріла система більше не буде потрібна. Коли цей процес завершиться, можна відключити і видалити стару систему.

Даний шаблон використовується у випадках переходу із монолітної системи до мікросервісної.

2.3.2 Шаблон Bulkhead – ізоляція відмов

Шаблон Bulkhead (Шаблон обкладинки) – це тип архітектури програми, яка стійка до відмов. Згідно із даною архітектурою, елементи програми

виділяються в групи, так що, якщо один не працює, інші продовжуватимуть функціонувати.

Рішенням є розділення екземплярів служб на декілька груп відповідно до характеру навантаження від користувачів і вимогами доступності. Такий підхід дозволяє ізолювати відмови (збої), зберігаючи працездатність служб хоча б для деяких користувачів навіть під час нештатної ситуації.

Аналогічно можна розділити ресурси для споживача, щоб виклики до однієї службі не впливали на доступність ресурсів для виклику інших служб.

Така схема надає наступні переваги:

– ізоляція споживачів і служб від каскадних відмов. Проблема, яка зачіпає споживача або службу, буде ізольована в одній конкретній групі, що не буде впливати на інші модулі;

– збереження функціональності навіть в разі збою служби. Інші служби та функції програми продовжать працювати;

– різна пріоритетність ресурсів для груп.

На рис. 2.4 показано клієнтів, які викликають один сервіс. Кожному клієнту призначений окремий екземпляр цієї служби. Клієнт №1 взаємодіє із пошкодженим сервісом. Так як кожен екземпляр служб ізольований від інших, інші клієнти спокійно продовжують працювати з цією службою.

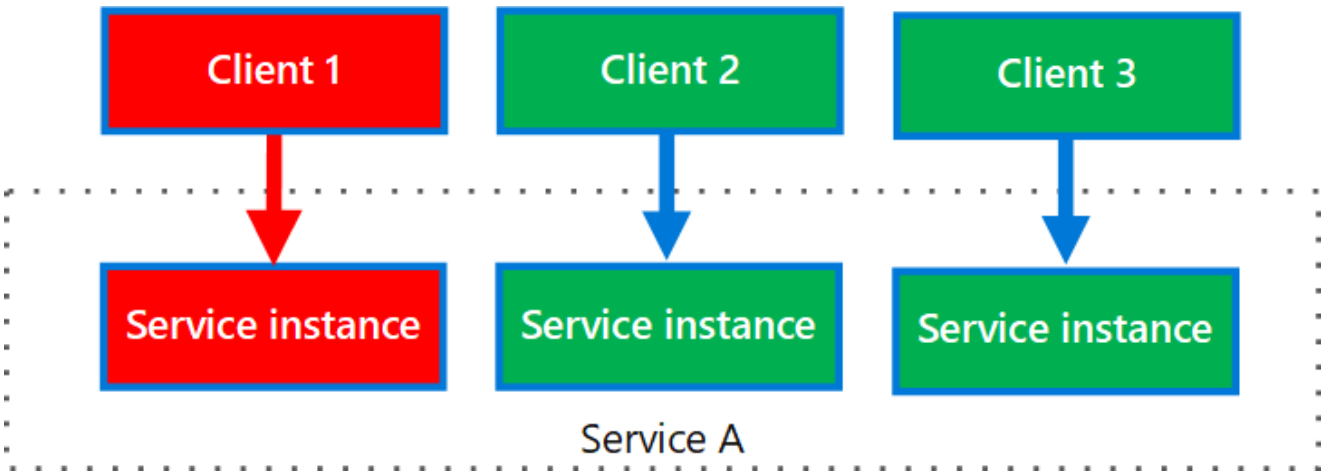


Рисунок 2.4. – Ізоляція відмов

2.3.3 Шаблон Sidecar – автономність сервісу

Шаблон Sidecar (*Шаблон розширення*) – для забезпечення ізоляції і інкапсуляції, компоненти програми розгортаються в окремому процесі або контейнері. Даний шаблон також може включати можливість створення додатків, що складаються з різнорідних компонентів і технологій.

Для додатків і служб часто потрібні пов'язані функції, такі як моніторинг, ведення журналів, конфігурація і мережеві служби. Ці периферійні завдання можна реалізувати в якості окремих компонентів або служб.

Якщо вони тісно інтегровані в додатку, то можуть виконуватися в одному процесі в якості додатку, за рахунок чого забезпечується ефективне використання загальних ресурсів. Однак це також означає, що вони неефективно ізольовані і збій в одному з цих компонентів може вплинути на інші або ж на кожну з програм. В результаті компонент і «батьківський» додаток тісно пов'язані один з одним.

Якщо додаток ділиться на служби, кожну з них можна створити з використанням різних мов і технологій. При цьому забезпечується додаткова гнучкість, але у кожного компонента є власні залежності, такі як доступ до базової платформі і ресурсів батьківського додатка. Крім того, розгортання цих функцій в якості окремих служб може призвести до затримок у додатку. Даний підхід продемонстровано на рис.2.5.

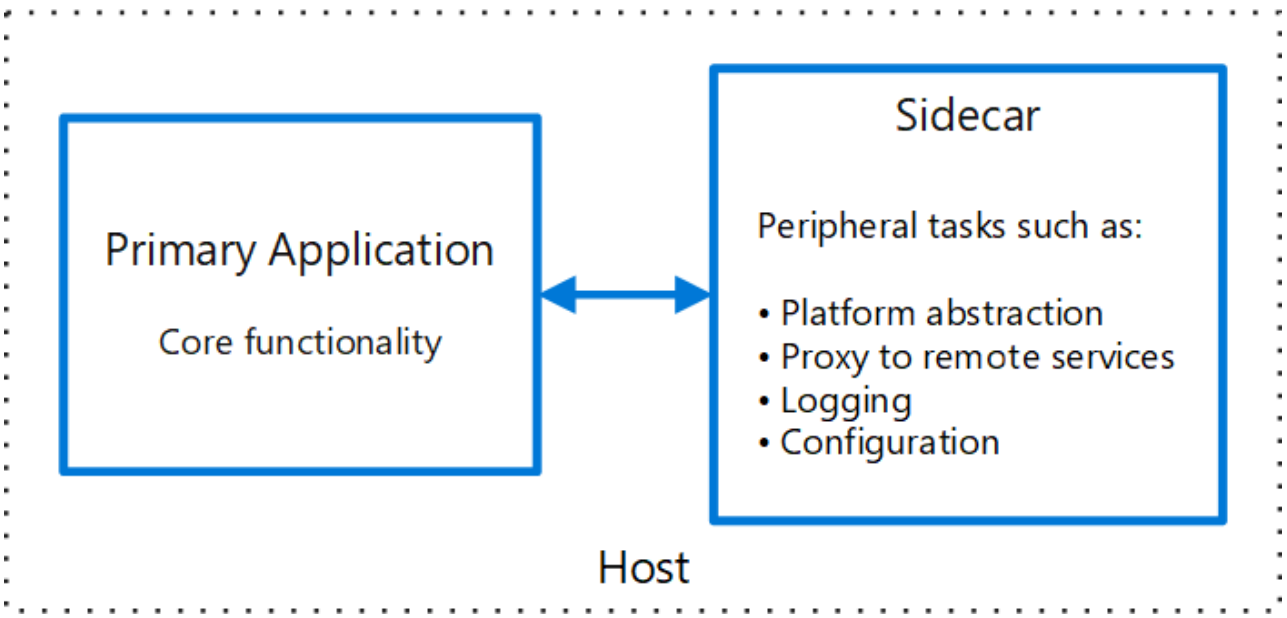


Рисунок 2.5. – Розділення служб по різним процесам

2.4 Опис міжпроцесорної взаємодії компонентів

Додаток на основі мікросервісів являє собою розподілену систему, що працює на кількох процесах або службах, іноді навіть не декількох серверах або вузлах. Зазвичай кожен екземпляр служби – це процес.

Таким чином, служби повинні взаємодіяти по протоколу внутрішьопроцесорної взаємодії, наприклад HTTP, AMQP або бінарного протоколу, такому як TCP, в залежності від характеру кожної служби.

Зазвичай використовуються два протоколи – запити і відповіді HTTP з вихідними API (в основному для запитів) і легкі асинхронні повідомлення при передачі оновлень у декілька мікросервісів.

Клієнт і служби можуть взаємодіяти через різні типи зв'язку в залежності від сценарію і цілей. Ці типи зв'язку можна розділити на два напрямки, які представлені на рис. 2.6.

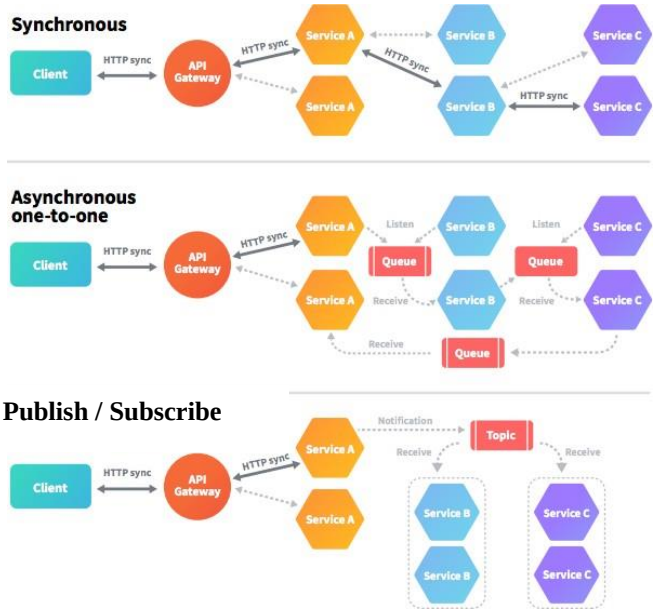


Рисунок 2.6. – Типи взаємодій мікросервісів

Перша група визначає, протокол синхронний чи асинхронний:

– синхронний протокол. HTTP – це синхронний протокол. Клієнт відправляє запит і чекає відповіді від служби. Тут важливо, що протокол (HTTP / HTTPS) є синхронним і код клієнта зможе продовжити виконання завдання тільки після отримання відповіді від HTTP-сервера;

– асинхронний протокол. Інші протоколи, наприклад AMQP (протокол, підтримуваний багатьма операційними системами і хмарними середовищами), використовують асинхронні повідомлення. Код клієнта або відправник повідомлення зазвичай не очікує відповіді. Він просто відправляє повідомлення, як при відправці повідомлення в чергу RabbitMQ або іншого брокера повідомлень.

Друга група визначає, має запит одного або кількох одержувачів:

– один одержувач – кожен запит повинен оброблятися тільки одним одержувачем або службою;

– декілька одержувачів – кожен запит може оброблятися різною кількістю одержувачів – від жодного до декількох. Такий тип взаємодії повинен бути асинхронним. Наприклад, механізм «Publish-Subscribe», який використовується в таких шаблонах, як архітектура, керована подіями. Він

заснований на інтерфейсі шини подій або брокера повідомлень, коли події оновлюють дані в декількох мікросервісах. Зазвичай це реалізується через службову шину або подібний об'єкт, наприклад службову шину Azure, за допомогою тем і підписок.

Додаток на базі мікросервісів часто використовує комбінацію цих стилів взаємодії. Найбільш поширений тип – взаємодія з одним одержувачем з синхронного протоколу, наприклад HTTP або HTTPS, при виклику звичайної служби веб-API HTTP. Для асинхронного взаємодії між мікросервісами зазвичай використовуються протоколи повідомлень.

2.5 Вибір стратегії для розгортання серверної частини та додатків

Кожен із мікросервісів представляє собою міні-додаток із своїми специфічними вимогами до розширення, ресурсу, масштабованості та моніторингу. Наприклад, потрібно запустити конкретну кількість екземплярів кожного сервісу в залежності від навантаження на цей сервіс. Крім того, 49 кожному екземпляру потрібно забезпечити відповідні ресурси процесора, пам'яті та I/O. Розгортання мікросервісу має бути швидким, надійним та економічно ефективним.

2.5.1. Розгортання на одному сервері

При використанні цього шаблону виділяється один або декілька фізичних або віртуальних серверів і запускається декілька екземплярів сервісів на кожному. У цілому це традиційний підхід до розгортання додатків.

Однією з основних переваг є ефективне використання ресурсів. Кілька екземплярів сервісів спільно використовують сервер і його операційну систему. Інша перевага цього шаблону полягає в тому, що розгортання екземпляру сервісу відбувається відносно швидко. Потрібно просто скопіювати вихідний код або скомпільований файл сервісу на сервер і запустити його.

Один з основних недоліків полягає в тому, що екземпляри сервісів практично не ізольовані, якщо тільки кожен екземпляр сервісу не є окремим процесом. Хоча ви можете точно здійснювати контроль за використанням ресурсів кожного примірника сервісу, ви не можете обмежувати ресурси, використовувані кожним екземпляром.

Неправильно працюючий екземпляр сервісу може використовувати всю пам'ять або ЦП сервера. Немає ніякої ізоляції сервісів, якщо кілька екземплярів виконуються в одному і тому ж процесі. Всі екземпляри можуть, наприклад, спільно використовувати одну і ту ж купу JVM (для Java додатків). Неправильно працюючий екземпляр сервісу може легко зламати інші сервіси, що працюють в тому ж процесі.

2.5.2. Віртуалізація на виділеному сервері

При використанні даного шаблону кожен екземпляр сервісу запускається ізольовано на своєму власному сервісі.

Потрібно упакувати кожен сервіс як образ віртуальної машини (VM), наприклад, такий як Amazon EC2 AMI. Кожен екземпляр сервісу – це віртуальна машина (наприклад, екземпляр EC2), яка запускається з використанням даного образу віртуальної машини.

На рис. 2.7 зображену схема розгортання сервісів на окремих серверах (хостах) із використанням VM.

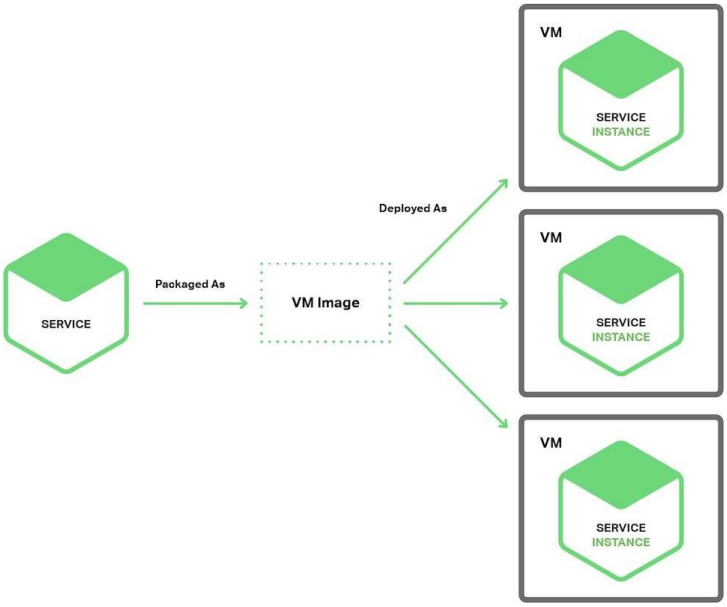


Рисунок 2.7 – Схема розгортання сервісів із використанням VM

Основною перевагою віртуальних машин є те, що кожен екземпляр сервісу працює в повній ізоляції. Він має фіксований обсяг ресурсів процесора і пам'яті і не може брати ресурси інших сервісів. Ще одна перевага розгортання мікросервісів як віртуальних машин полягає в тому, що можна використовувати розвинену хмарну інфраструктуру. Хмари, такі як AWS, Azure, Google Cloud, надають корисні функції, такі як балансування навантаження і автоматичне масштабування за потребою.

Недоліком є менш ефективне використання ресурсів. Кожен екземпляр сервісу має накладні витрати на віртуальну машину, включаючи операційну систему. Крім того, в типовому загальнодоступному IaaS (анг. «Infrastructure as a Service», Інфраструктура як послуга) віртуальні машини мають фіксовані розміри, і можливо, що віртуальна машина буде використовуватися недостатньо.

Іншим недоліком даного підходу є те, що розгортання нової версії сервісу зазвичай відбувається повільно. Образи віртуальних машин зазвичай створюються повільно через їх великі розміри. Крім того, для запуску операційної системи зазвичай потрібен деякий час.

2.5.3 Контейнеризація на виділеному сервері

При використанні даного підходу, кожен екземпляр сервісу запускається в своєму власному контейнері. Контейнери – це механізм віртуалізації на рівні операційної системи. Контейнер складається з одного або декількох процесів, що виконуються у власному обмеженому середовищі. З точки зору процесів, вони мають свій власний простір імен портів і кореневу файлову систему.

Існує можливість обмеження пам'яті контейнера і ресурсів процесора. Деякі реалізації контейнерів також мають обмеження швидкості I/O. Приклади контейнерних технологій включають «Docker» і «Solaris Zones».

Щоб використовувати дану стратегію, потрібно упакувати сервіс як «image» (образ) контейнера. Образ контейнера – це образ файлової системи, що складається з додатків і бібліотек, необхідних для запуску сервісу.

Можна використовувати менеджер кластерів, такий як «Kubernetes» або «Marathon», для управління контейнерами. Менеджер кластера розглядає хости як пул (групу) ресурсів. Менеджер вирішує, де розмістити кожний контейнер ґрунтуючись на ресурсах, необхідних для контейнера, і ресурсах, доступних на кожному хості.

Переваги контейнерів аналогічні перевагам віртуальних машин. Вони ізолюють сервісні екземпляри один від друга. Можна легко відстежувати ресурси, що споживаються кожним контейнером. Також, як і віртуальні машини, контейнери інкапсулюють технологію, використовувану для реалізації ваших послуг. API керування контейнерами також служить API для керування вашими сервісами.

Однак, на відміну від віртуальних машин, контейнери – це «легка» технологія. Контейнерні образи зазвичай дуже швидко створюються. Контейнери також запускаються дуже швидко, так як немає довгого механізму завантаження ОС. Коли контейнер запускається – запускається сервіс.

Хоча контейнерна інфраструктура стрімко розвивається, вона не настільки розвинена, як інфраструктура для віртуальних машин. Крім того, контейнери не так безпечні, як віртуальні машини, оскільки контейнери спільно використовують ядро операційної системи хоста.

2.5.4 Безсерверне розгортання

«Serverless» – безсерверна архітектура додатків. Основу архітектури складають мікросервіси, або функції (lambda), що виконують певне завдання і запускаються на контейнерах. Тобто кінцевому користувачеві дано тільки інтерфейс завантаження коду функції (сервісу) і можливість підключення до функції джерел подій (events).

Переваги даної архітектури:

– відсутність апаратної частини - серверів;

– відсутність прямого контакту і адміністрування серверної частини;

– практично необмежене горизонтальне масштабування;

– оплата тільки за використаний час CPU.

Недоліки:

– відсутність чіткого контролю контейнера – невідомо, де і як запускаються контейнери, хто має доступ;

– відсутність «цілісності» програми: кожна функція – це незалежний об'єкт, що може привести до поганої структурованості.

– «холодний старт» контейнера – перший запуск контейнера з лямбда функцією може становити 2-3 секунди, що не завжди добре сприймається користувачами.

2.6 Проектування архітектури системи.

Комплексна система управління сервісами – клієнт-серверний додаток, у якому клієнт взаємодіє з веб-серверами за допомогою браузера. Логіка веб-додатків розбита між серверами та клієнтами. Дані зберігаються та оброблюються на серверах, до яких можна отримати доступ за допомогою

браузера, мобільного додатку через протоколи HTTP або HTTPS (протокол із шифруванням інформації на базі TLS).

Данна система схожа на звичайний комп'ютерний застосунок, за винятком того, що він працює через Інтернет.

Архітектура веб-додатків описує взаємодію між додатками, базами даних та іншими проміжними компонентами.

Веб-архітектура – це концептуальна структура Всесвітньої павутини. Всесвітня павутина або інтернет дозволяє спілкуватися між різними користувачами та між різними системами та підсистемами.

Основою цього є різні компоненти та формати даних, які формують інфраструктуру Інтернету, що стає можливим завдяки трьом основним компонентам протоколів передачі даних (TCP / IP, HTTP, HTTPS), форматах представлення (HTML, CSS, XML, JSON, YAML) та стандартам адресації (URI, URL).

Термін веб-архітектура слід відрізняти від термінів архітектура веб-сайтів та архітектура інформації.

На рис. 2.8 представлено схематично представлено процес взаємодії користувача із системою.

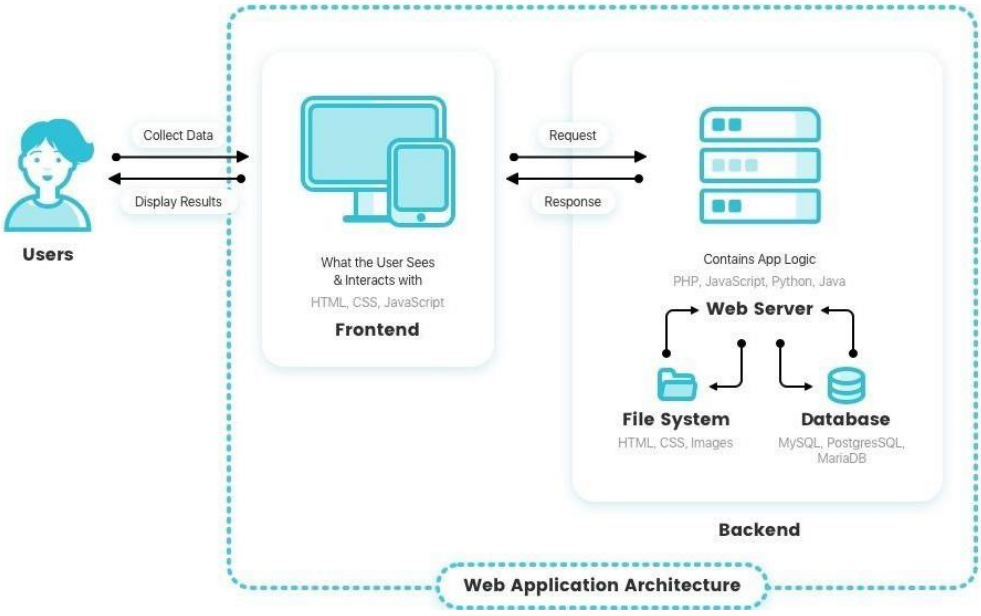


Рисунок 2.8 – Взаємодія користувача із системою

У будь-якому типовому веб-додатку (системі) є два компоненти:

– програмний код на стороні клієнта (Front-end складова)

– програмний код на стороні сервера (Back-end складова)

Програмний код на стороні клієнта реагує на події від користувача, такі як ввід, натискання на кнопку, наведення курсору на елемент. Комбінація CSS, HTML та JavaScript використовується для написання коду на стороні клієнта. Цей код аналізується веб-браузером. На відміну від серверного коду, клієнтський код можна переглянути, а також змінювати.

Клієнтський код зв’язується з веб-сервером лише через HTTP-запити і не може безпосередньо читати файли або дані з сервера.

Програмний код на стороні сервера виконує задачі по обробленню, зберіганню, агрегації даних. Для написання коду на стороні сервера використовуються такі високорівневі мови програмування як C#, Java, Python, PHP, Ruby. Серверна частина відповідає на HTTP-запити, відтворює сторінку, яку запитував користувач, обробляє дані, які надіслав користувач.

Висновки до розділу 2

Програмне забезпечення «Docker Swarm» дозволяє створювати кластерну інфраструктуру, яка масштабованою та стійкою до відмов. Було створено чотири мікросервіса і організовано взаємодію між ними при цьому було застосовано стандартизований підхід на основі REST-API та JSON API для взаємодії між сервісами.

Використовуючи платформу «Docker Hub» було реалізовано концепцію безперервної доставки, що значно спростило і пришвидшило розробку програмного забезпечення.

Було проаналізовано та досліджено відмово-стійкість системи і визначено що кластер відновлює свій стан після відмови деякого вузла. За допомогою «Prometheus» було створено моніторинг системи, що дає зможу своєчасно виявляти потенційні проблеми у роботі серверу або сервісів.

РОЗДІЛ 3
РОЗРОБКА АПАРАТНОЇ ЧАСТИНИ

3.1 Опис концепції контейнеризації

Один з принципів базової організації мікросервісної архітектури говорить про те, що сервіс повинен бути ізольованим і автономним, повністю інкапсулюючи оточення виконання. Для дотримання цього принципу всі компоненти, такі як операційна система, середовище виконання та виконуваний код мікросервіс повинні бути автономними і ізольованими. Єдиний спосіб добитися цього - один мікросервіс на одну віртуальну машину. Однак це призведе до недостатньої утилізації ресурсів віртуальної машини. Також у багатьох випадках через додаткові накладні витрати можуть звести всі переваги мікросервісів нанівець [29].

Технологія контейнеризації далеко не нова та не новаторська технологія. Вона використовується досить тривалий час. Проте, дана технологія стала набирати значної популярності з приходом хмарних технологій. Недоліки традиційних віртуальних машин стали каталізатором зростання популярності контейнерів. Постачальники інструментів для роботи з контейнерами, наприклад, «Docker», значною мірою спростили технологію контейнеризації, що сприяло впровадженню даної технології у широкі маси. Популярність «DevOps» та мікросервісної архітектури також прискорила переродження технології контейнеризації.

Технологія контейнеризації надає приватне оточення операційну систему. Ця технологія також називається віртуалізація операційної системи [30]. У цьому підході ядро операційної системи надає ізольований віртуальний простір. Кожне з Віртуальний простір називається контейнером. Контейнери дозволяють процесам створювати ізольоване оточення на операційній системі містить ці контейнери.

На рис.3.1 зображені різні шари, залучені до процесу контейнеризації.

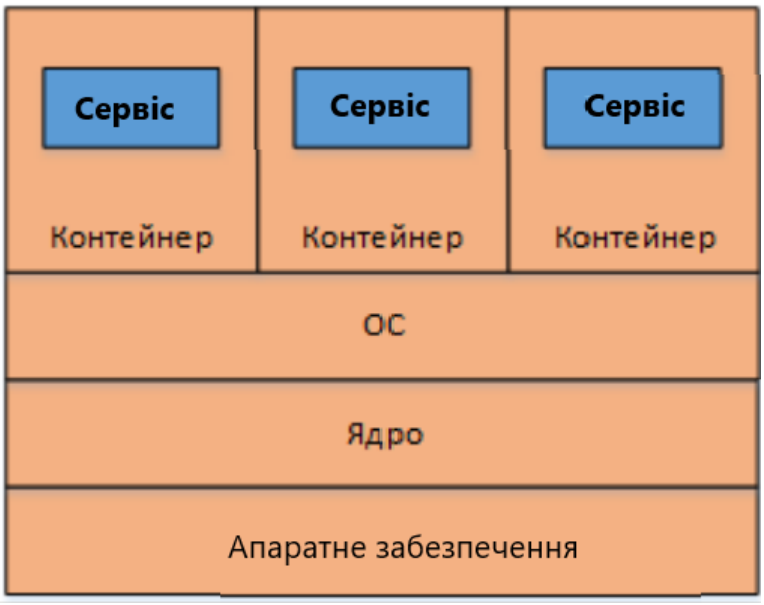


Рисунок 3.1 – Шари контейнеризації

Контейнери, це простий механізм для збирання та постачання слабопов'язаних компонентів програмного забезпечення. У загальному випадку, контейнери упаковують всі виконувані файли та бібліотеки, які необхідні для запуску програми.

Контейнери повністю ізолюють такі елементи:

– файлову систему;
– IP адреса;
– мережеві інтерфейси
– ;внутрішні процеси;
– простору імен;
– бібліотеки операційної системи;
– бінарні файли програми;
– залежності;
– файли конфігурації програми.

Усі інструменти контейнеризації засновані на функціональності ядра Linux. Основні компоненти ядра Linux для контейнеризації перераховані нижче:

• простору імен;

• керуючі групи.

Простір імен (анг. Namespaces) використовується для створення ізольованого оточення, яке називається контейнер. Коли запускається контейнер, докер створює нові простори імен для цього контейнера. Простір імен надають із себе шар ізоляції. Кожен аспект роботи контейнера виконується окремому просторі

.Нижче перераховані групи просторів імен:

– PID;

– мережна взаємодія;

– міжпроцесна взаємодія

– точки монтування файлової системи;

– ізольоване ядро та ідентифікатори версій.

Керуючі групи (анг. Control groups) так само, як і простору імен містяться в ядрі Linux. Керуючі групи обмежують ресурси, споживані додатком. Також вони дозволяють движку докеру ділити доступні апаратні ресурси між контейнерами і обмежувати або розширювати ці ресурси за потребою. Наприклад, можна обмежити використання пам'яті якогось певного контейнера. Різні організації використовують мільярди контейнерів. Крім цього, багато великих компаній інвестують у технологію контейнеризації. «Docker» значно випереджає конкурентів та підтримується багатьма великими постачальниками операційних систем, а також хмарними провайдерів. Також зараз розробляється відкрита специфікація контейнерів.

3.2 Відмінність віртуалізації та контейнерів

Віртуальні машини такі як «Hyper-V», «VMWare» та «Zen» були найпопулярнішим вибором для віртуалізації дата-центрів кілька років тому. Промислові компанії значно заощадили свої кошти завдяки впровадження віртуалізації у порівнянні з використанням сервером без віртуалізації [31]. Віртуалізація також допомогла багатьом підприємствам оптимізувати використання існуючих інфраструктур. Оскільки віртуальні машини

дозволяють автоматизувати процес побудови інфраструктури, багато компаній зіткнулися з необхідністю зменшення зусиль з управління цією автоматизацією. Віртуальні машини також допомогли компаніям отримати ізольовану середовище для запуску програм.

На перший погляд віртуалізація та контейнеризація мають однакові показники. Проте, контейнери та віртуальні машини –це не одне і те ж. Таким чином твердження, віртуальні машини та контейнери працюють однаково, некоректно. Віртуальні машини та контейнери зовсім різні за своєю природою технології, також вони вирішують різні проблеми віртуалізації.

Ці відмінності помітні на рис. 3.2.

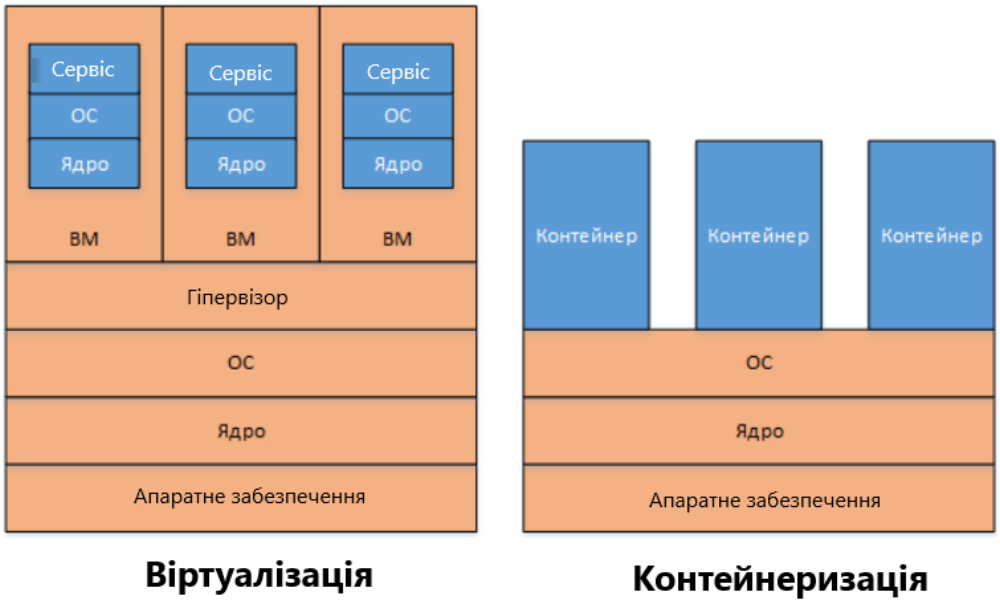


Рисунок 3.2 – Відмінності віртуалізації та контейнеризації

Віртуальні машини працюю на набагато нижчому рівні, ніж контейнери. Віртуальні машини надають віртуалізацію програмного забезпечення, наприклад, CPU, систем введення-виводу, пам'яті. Віртуальна машина ізольований компонент з вбудованою операційною системою, яка називається гостьовою операційною системою. Віртуальна машина містить у собі операційну систему цілком і виконує її без будь-якої залежності оточення операційної системи, де запущено образ віртуальної машини.

Оскільки віртуальна машина містить оточення операційної системи цілком, вона досить важка. З одного боку, цей аспект можна вважати перевагою, а з іншого недоліком. Перевага полягає у повній ізоляції процесів, що виконуються на віртуальній машині. Нестача полягає в обмеженій кількості віртуальних машин, які можуть виконуватися на сервері, оскільки віртуальні машини резервують ресурси необхідні їхньої роботи.

Розмір віртуальної машини безпосередньо впливає на час запуску та зупинки. Запуск віртуальної машини починає завантаження операційної системи, тому загальний час запуску віртуальної машини як правило, високе. Віртуальні машини більш привабливі для співробітників компаній, які керують інфраструктурою, оскільки не потребують воскового рівня кваліфікації світі контейнерів, контейнери не емулюють апарат незабезпечення чи операційну систему цілком.

На відміну від віртуальних машин, контейнери використовують певні частини ядра як гостьовий операційної системи, так і хостової операційної системи. Контейнери використовують концепцію гостьової операційної системи. Технологія контейнеризації надає ізольоване виконання оточення безпосередньо на хостовій операційній системі. У свою чергу, цей аспект також є одночасно перевагою та недоліком. Перевага полягає в легковажності та швидкості [30]. Оскільки контейнери спільно використовують хостову операційну систему. Використання ресурсів контейнерами досить мало. В результаті на одній машині може бути запущено безліч невеликих контейнерів, чого не можна досягти при використанні віртуальних машин. Оскільки контейнери запущені на одній хостовій операційній системі існують та обмеження. Наприклад, неможливо встановити налаштування між мережевого екран (анг. ip tables) всередині контейнера. Процеси всередині контейнера повністю ізольовані та незалежні від процесів в інших контейнерах, що виконуються на одній хостовій системі.

На відміну від віртуальних машин, образи контейнерів публічно-доступні на різних порталах. Це значною мірою спрощує роботу розробників, оскільки вони не витрачають свого часу на побудову образів із нуля. Можна скористатися базовими образами з сертифікованих джерел та додати додаткові шар програмних компонентів на базовий образ.

Легковажна природа контейнерів також відкриває безліч можливостей для їх автоматичного створення, публікації, завантаження та копіювання. Можливість завантажити, зібрати, доставити та запустити контейнер лише з використанням декількох команд або з використанням REST API більш підходящим вибором під час розробки систем. Складання нового контейнера займає кілька секунд. Таким чином складання контейнера може стати етапом процесу безперервного постачання.

Таким чином контейнери мають величезну перевагу перед віртуальними машинами, але віртуальні машини мають свої сильні сторони. Більшість компаній використовують як контейнери, так і віртуальні машини, наприклад, для запуску контейнерів у віртуальній машині.

3.3 Система керування контейнерами Docker

«Docker» є найпопулярнішою платформою, що використовує технологію контейнеризації [33].

Платформа «Docker» вирішує три основні проблеми розгортання сервісів:

- доставка коду на сервер;
- запуск коду;
- одноманітність оточення.

«Docker» дозволяє ізолювати послуги від інфраструктури, таким Таким чином досягається можливість доставляти їх набагато швидше. «Docker» дозволяє керувати інфраструктурою за допомогою тих самих принципів, які використовуються при керуванні програмами. При використанні

інструментів «Docker» для доставки, тестування та розгортання, значно скорочується затримка між фіксацією нового коду у системі контролю версій та запуском його на сервері промислової експлуатації.

«Docker» надає можливість упаковувати та запускати сервіси в ізольованому оточенні, яке називається контейнером. Дана ізольованість та безпека дозволяє запускати кілька контейнерів на одному хості одночасно. Контейнери легковажні, оскільки не вимагають роботи гіпервізора. Вони запускаються безпосередньо на ядрі хост машини. Таким чином досягається можливість запуску більшої кількості контейнерів, ніж віртуальних машин, на тому самому фізичному обладнання. Також, docker-контейнери можна запускати у самих віртуальні машини.

Він надає наступні інструменти та платформу для керування життєвим циклом контейнерів. За допомогою нього відбувається упаковка програми та їх компонентів у контейнер. В екосистемі «Docker» ,контейнер стає атомарним одиницею для поширення та тестування програми. Після розробки, програма може бути розгорнуто на сервері промислової експлуатації вручну або за допомогою оркеструвальника контейнерів. Дана техніка розгортання однакова незалежно від того, де розгортається додаток на промислову експлуатацію, будь то в локальному центрі, хмарному провайдері або в гібридному оточенні.

«Docker» оптимізує життєвий цикл розробки, дозволяючи розробникам працювати у стандартизованому оточенні. При використання докер, оточення при розробці нічим не відрізняється від оточення у промисловій експлуатації.

Контейнери ідеально підходять для безперервної інтеграції або безперервного постачання [32]. Звичайний сценарій розробки програмного забезпечення зводиться до отримання якогось артефакту, що будується після будь-якої зміни вихідного коду програми. При використанні Докер, цим артефактом може бути контейнер. Розробники чи інженери з тестування використовують докер контейнери для розгортання додатків на тестове

оточення та запускають автоматичні або ручні випробування. Якщо під час перевірок виявляється баг, цей контейнер може бути розгорнутий в оточенні розробки, виправлений і перенесений назад тестове оточення для повторної перевірки. Якщо всі перевірки успішно завершено достатньо доставити контейнер з виправленнями в промислове (реальне) оточення.

Платформа легковага та швидка. Вона надає життєздатну, економічно вигідну альтернативу віртуальним машинам на основі гіпервізора. З її допомогою можна використовувати більше обчислювальних потужностей, оскільки ці ресурси не витрачаються на обслуговування гіпервізора.

3.4 Архітектура Docker

«Docker» використовує клієнт-серверну архітектуру, схематичне зображення якої представлено на рис. 3.3.

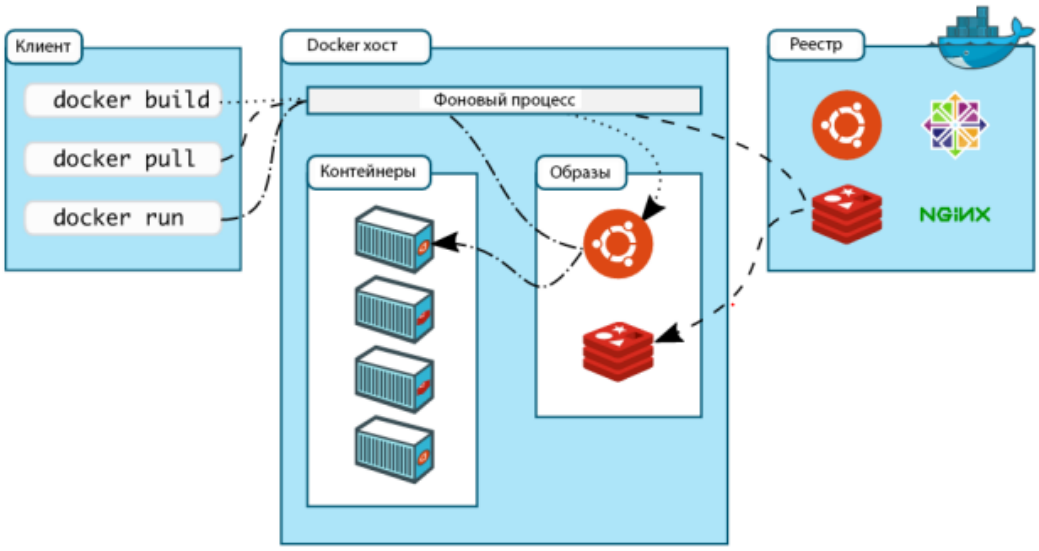


Рисунок 3.3 – Архітектура Docker

Docker-клієнт взаємодіє з фоновим процесом – сервером Docker, який, у свою чергу, запускає контейнери. Клієнт та фоновий процес можуть виконуватися в одній операційній системі, також є можливість підключити клієнта до віддаленого фонового процесу. Docker-клієнт та фоновий процес взаємодіють через REST API поверх сокетів або через мережевий інтерфейс. Фоновий процес «Docker» (dockerd) приймає запити та керує його об’єктами.

Фоновий процес також може взаємодіяти з іншими фоновими процесами. Він керує такими об'єктами як:

- образи;
- контейнери;
- мережна взаємодія;
- томи.

Docker-клієнт є основним способом взаємодії з сервером. При виконання команди клієнт надсилає цю команду фоновому процесу, який у свою чергу її виконує. Docker-клієнт може взаємодіяти з безліччю різних серверів.

3.5 Робота з образами.

Реєстр контейнерів зберігає образи. Цей реєстр може бути, як громадським, і приватним [31]. «Docker Hub» є офіційним відкритим реєстром, в якому зберігаються офіційні образи від різних провайдерів технологій, наприклад «Ubuntu» або «Nginx». Додавати образи до публічний реєстр може будь-хто. За замовчуванням Docker налаштований на пошук образів у цьому публічному реєстрі. Також є можливість створювати приватні реєстри образів.

При використанні команд «docker pull» або «docker run» необхідні образи будуть завантажені зі вже сконфігурованого реєстру. При використанні команди «docker push», вказаний образ буде розміщено в реєстрі.

При використанні докер створюються та використовуються образи, контейнери, мережева взаємодія, томи, плагіни та інші компоненти [32].

Образ – це незмінний шаблон, що містить інструкції для створення контейнера. У більшості випадків образ базується на іншому образі, доповнюючи його новою функціональністю. Наприклад, можна збудувати образ, що базується на образі ОС Linux «Ubuntu», розширивши його

встановивши веб-сервер «Apache» з розробленою програмою, а також настроївши сервер для правильної роботи програми.

Є можливість створювати нові образи або ж пере використовувати вже створені та розміщені у реєстрі. Щоб створити новий образ, потрібно створити конфігураційний файл, який називається «Docker File». Цей файл містить інструкції для створення та запуску зображення. Кожна інструкція конфігураційного файлу створює шар в образі. Коли конфігураційний файл змінюється та запущено процес повторного складання образу, тільки ті шари, які змінилися, будуть зібрані знову. Ця технологія робить контейнери легковагими та швидкими в порівнянні з іншими технологіями віртуалізації.

Контейнер – це запущений екземпляр образу. Контейнери можна створювати, запускати, зупиняти та видаляти, використовуючи «Docker API» або інтерфейс командного рядка. Контейнер може бути підключений до одного або кілька мереж. До нього може бути змонтована частина файлової системи ост машини. Також є можливість створити новий образ із поточного стан контейнера.

За замовчуванням контейнери відносно добре ізольовані від інших контейнерів та хост-машини [32]. Є можливість керувати ізоляцією мережевої взаємодії контейнерів та підключеними томами між іншими контейнерами та хостовою системою.

Контейнер створюється з образу, а також містить усі конфігураційні параметри, що надаються при його створенні та запуску. При зупинці контейнера, всі зміни в його стані, якісно зафіксовані у постійному сховищі пропадають.

Кожен образ посилається на список незмінних шарів, які представляють розбіжності у файловій системі. Шари складені один на одного, щоб сформувати базу для кореневої файлової системи контейнера, як показано на рис.3.4.

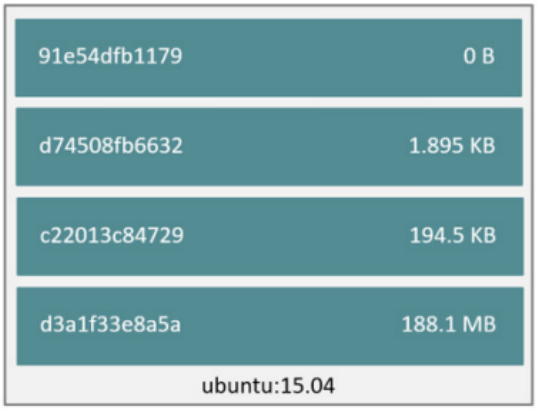


Рисунок 3.4 – Рівні образу «Docker»

Коли створюється контейнер, до образу додається додатковий рівень, який називається контейнерним шаром [33]. Всі зміни зроблений цьому запущеному контейнері, такі як створення, зміна, видалення файлів, записуються на цей шар.

Рис. 3.5 зображено шари вже запущеного контейнера.

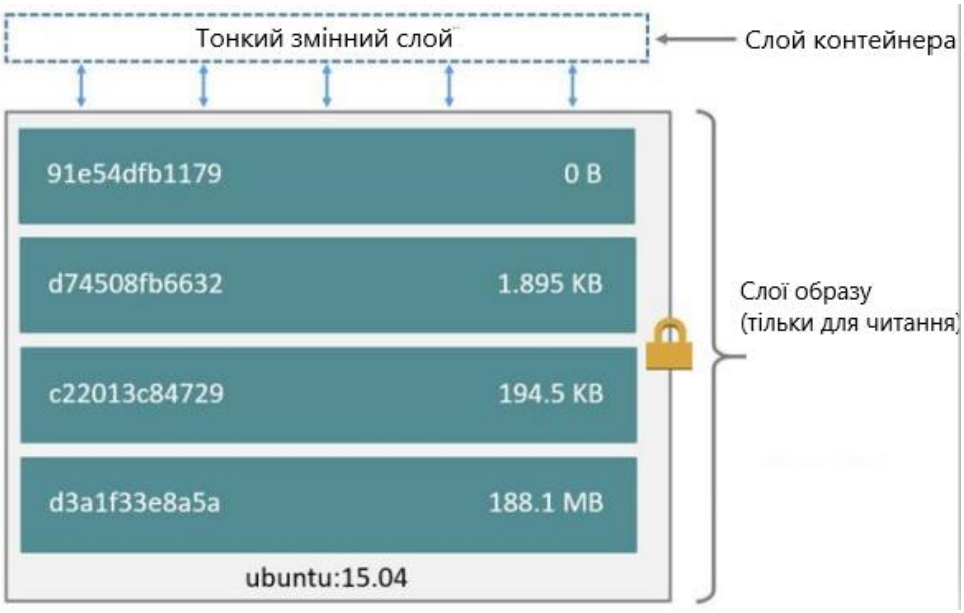


Рисунок 3.5 – Шари запущеного контейнера

Основною відмінністю між контейнером та образом є верхній шар, що змінюється. Усі зміни контейнера, які додають або змінюють дані зберігаються в цьому шарі. Коли контейнер видаляється, шар, що змінюється, видаляється разом з ним, а шар, що ієрархічно розташований нижче, залишається недоторканим. Оскільки кожен контейнер має власний шар, що

змінюється, і всі зміни зберігаються в ньому, безліч контейнерів можуть спільно використовувати його образ та мати різний стан (рис.3.6).

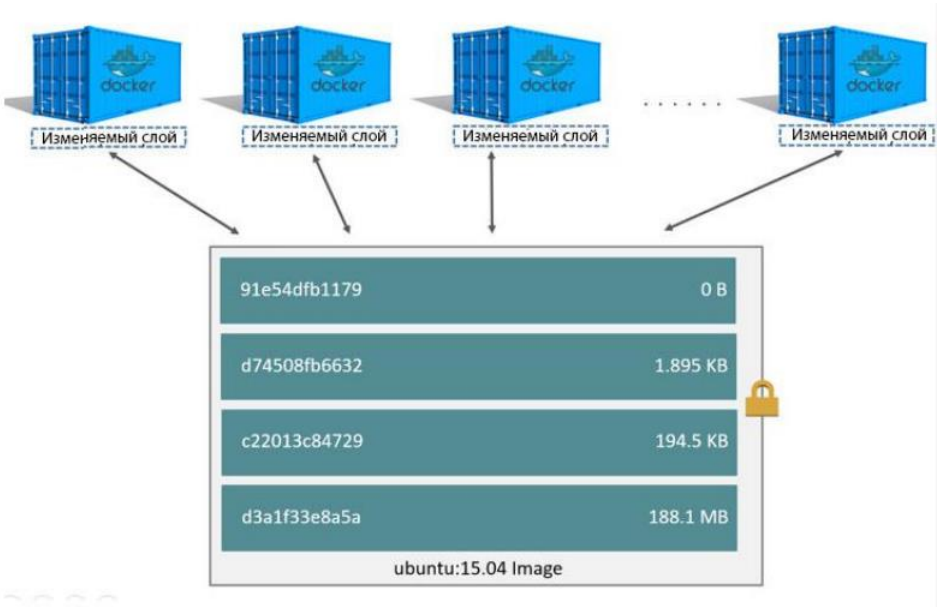


Рисунок 3.6 – Використання одного образу безліччю контейнерів

Іншою концепцією, що дозволяє збільшити продуктивність полягає у копіюванні даних при зміні (Copy-on-Write). При використання цієї стратегії системні процеси, яким потрібні однакові дані, використовують їх спільно, та не мають власну копію. Якщо одному процесу потрібно змінити дані, лише операційна система зробить копію даних цього процесу. Тільки той процес, що модифікує дані, має копію цих даних.

«Docker» використовує стратегію копіювання під час запису як для образів, і для контейнерів [34]. За допомогою цієї стратегії оптимізується як дисковий простір, що займається як образами, так і час запуску контейнерів.

3.6. Вибір вимог до технічних і програмних засобів

Слід звернути увагу на використання наступних технічних і програмних засобів:

1. «Docker» – це програмна платформа для швидкої розробки, тестування і розгортання додатків. Вона упаковує програмні додатки в стандартизовані блоки, які називаються контейнерами.

В процесі виконання проекту планується використання версії «Docker» під порядковим номером 19.03.5.

2. «PHP» – скриптова мова загального призначення, що інтенсивно застосовується для розробки веб-додатків.

В процесі виконання проекту планується використання версії PHP під порядковим номером 7.2.26.

3. «Laravel» – безкоштовний веб-фреймворк з відкритим кодом, призначений для розробки веб-додатків з використанням архітектурної моделі MVC (анг. Model View Controller, модель-уявлення-контролер). Laravel має вбудовані модулі для роботи із базою даних, HTTP-запитами та сесіями.

В процесі виконання проекту планується використання версії Laravel під порядковим номером 6.10.1.

4. «PhpStorm» – комерційне крос-платформне інтегроване середовище розробки для PHP.

PhpStorm включає у себе інтелектуальний редактор для PHP, HTML і JavaScript з можливостями аналізу коду на льоту, запобігання помилок в коді і автоматизованими засобами рефакторингу для PHP і JavaScript. Є повноцінний SQL-редактор з можливістю редагування отриманих результатів запитів.

В процесі виконання проекту планується використання версії PhpStorm під порядковим номером 2019.3.

5. «MySQL» – це система керування базами даних, а саме реляційними БД. У реляційній базі даних дані зберігаються в окремих таблицях, завдяки чому досягається виграш в швидкості і гнучкості.

«SQL» як частина системи MySQL можна охарактеризувати як мову структурованих запитів плюс найбільш поширений стандартний мова, яка використовується для доступу до баз даних.

В процесі виконання проекту планується використання MySQL під порядковим номером 5.7.21.

6. «VirtualBox» – це програмний продукт віртуалізації для різних операційних систем. Тобто, це програмне забезпечення, яке імітує справжній комп'ютер, що дає можливість користувачеві встановлювати, запускати і використовувати інші операційні системи, як звичайні додатки. Такий собі комп'ютер в комп'ютері.

В процесі виконання проекту планується використання версії VirtualBox під порядковим номером 6.1.2.

7. «Prometheus» – програмне забезпечення для збору і аналізу показників стану веб-додатків, а саме RAM, CPU, кількість помилок і інші користувацькі метрики.

В процесі виконання проекту планується використання версії Prometheus під порядковим номером 2.15.2.

8. «Grafana» – інструмент для візуалізації зібраних метрик (показників), що дозволяє будувати гнучкі графіки, діаграми, проводити обчислення над даними метрик.

В процесі виконання проекту планується використання версії Grafana під порядковим номером 6.5.3.

Висновки до розділу 3

«Docker» розроблений для більш швидкого розгортання додатків. З його допомогою можна відокремити додаток від інфраструктури і розгортати додаток на будь-якому хості, який підтримує віртуалізацію. Цей сервіс допомагає розгортати, швидко тестувати програмні додатки та зменшити час між написанням коду і його запуску. Docker робить це за допомогою «легкої» платформи контейнерної віртуалізації. За допомогою цієї платформи можна виконувати будь-який програмний додаток, який безпечно ізольований у контейнері.

РОЗДІЛ 4
РОЗРОБКА ПРОГРАМНОЇ ЧАСТИНИ

4.1 Процес контейнеризації системи

«Docker» дозволяє запаковувати сервіси з усіма залежностями в образи, що дає можливість передавати дані образи через мережу і запускати на будь-якому сервері, де встановлена ця програмна платформа. Для реалізації постановки задачі створимо декілька мікросервісів: «api-gateway», «users-api», «documents-api», «statistics-api».

Сервіс «apigateway» надає зовнішнім клієнтам загальний інтерфейс взаємодії з системою. Також даний шлюз нерідко виступає в якості єдиної точки входу для зовнішніх запитів. В такому випадку на шлюз може бути покладена відповідальність за забезпечення безпеки транспортного рівня з використанням різних каналів безпеки.

До переваг використання «api-gateway» підходу також відносять: ізоляція клієнтів від структури сервісів, уніфікований API для клієнтів, автоматичне розпізнавання місцезнаходження потрібного сервісу.

– задача мікросервісу «users-api» – зберігання інформації про користувачів.

– задача мікросервісу «documents-api» –інформація про документи.

– задача мікросервісу «statistics-api» – агрегація статистики відвідувань.

Кожен мікросервіс має свою власну БД «MySQL».

На рис. 4.1. зображена схема архітектури проектованого додатку

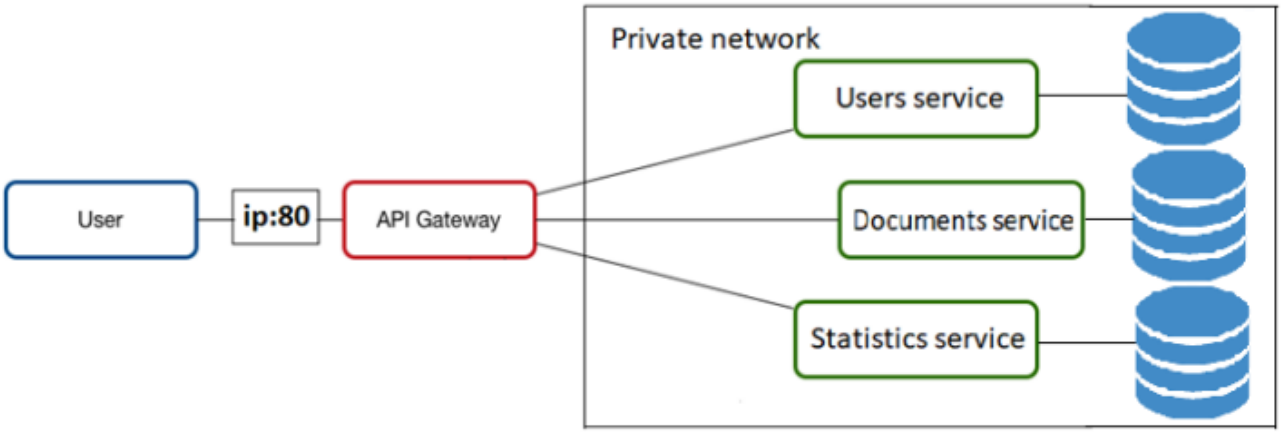


Рис. 4.1. – Архітектура додатку

4.1.1 Мікросервіс «api-gateway»

В основі сервісу «api-gateway» знаходиться веб-сервер «Nginx», який прослуховує порт 80 (порт HTTP) і приймає запити від користувачів (веб-браузер, мобільний додаток та інші). На основі URL вирішується, до якого мікросервісу направити запит.

URL, що починається із «api/v1/users» – перенаправляється до мікросервісу «users-api», «api/v1/documents» – перенаправляється до мікросервісу «documents-api», «api/v1/statistics» – перенаправляється до мікросервісу «statistics-api».

Конфігурація веб-сервера знаходить у файлі «default.conf», який аналізується сервером «Nginx» при запуску служби. Інструкції конфігурації веб-сервера приведено у додатку С.

Структурна схема представлена на рис. 4.1.

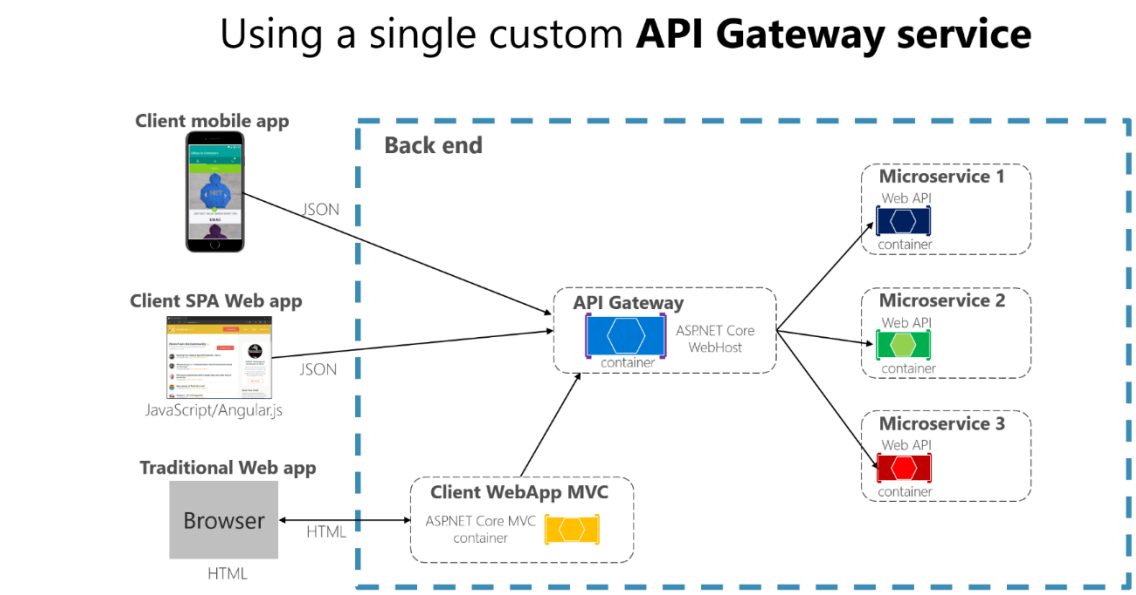


Рисунок 4.1 – Мікросервіс api-gateway

4.1.2 Мікросервіс «users-api»

Мікросервіс *user-api* включає у себе веб-сервер «Nginx» та службу «*php-fpm*» для обробки PHP-скриптів.

На рис.4.2 зображений REST-сумісний інтерфейс для взаємодії із сервісом. Тобто, GET запит на URL адресу «`[app_ip]:80/api/v1/users`» дає змогу отримати користувачів додатку.

На рис. 4.2 зображено приклад відповідіза даним запитом.

```
+--------+----------+------------------+------+-----------------------------------------------+------------+
| Domain | Method   | URI              | Name | Action                                        | Middleware |
+--------+----------+------------------+------+-----------------------------------------------+------------+
|        | GET|HEAD | api/v1/users     |      | App\Http\Controllers\UsersController@index    |            |
|        | GET|HEAD | api/v1/users/{id} |      | App\Http\Controllers\UsersController@getById |            |
+--------+----------+------------------+------+-----------------------------------------------+------------+
```

Рисунок 4.2 – Інтерфейс взаємодії із сервісом users-api

Приклад відповіді на API запит представлено нижче:

```
{
    "data": [
        {
            "id": 1,
            "name": "magnus.bashirian",
            "email": "gislason.loy@gmail.com",
            "created_at": "2020-01-19 13:22:17",
            "updated_at": null
        },
    ],
    "meta": {
        "container_id": "6ae6484f505a",
        "service_name": "users-api"
    }
}
```


Збірка образу виконується за допомогою «Dockerfile», який містить у собі усі інструкції по тому, як збирати кожен шар образу. Кожна інструкція додає до образу новий шар, тобто варто мінімізувати їх кількість, щоб зменшити розмір образу.

У додатку В приведено вміст «Dockerfile» файлу для сервісів: *users-api*, *documents-api*, *statistics-api*.

Інструкція FROM вказує, який базовий образ використовувати, у даному випадку це образ із встановленим PHP. Нижче даної інструкції йде встановлення усіх необхідних пакетів.

Інструкція CMD вказує на запуск служб додатку: *nginx* та *php-fpm*.

4.1.3 Мікросервіс «documents-api»

Мікросервіс *documents-api* включає у себе веб-сервер *nginx* і службу *php-fpm* для обробки PHP-скриптів.

На рисунку 4.3. зображено REST-сумісний інтерфейс взаємодії із сервісом «*documents-api»*. Тобто, за допомогою визначених GET запитів можна дізнатись інформацію про документи.

```
+--------+----------+--------------------------+------+-----------------------------------------------------+------------+
| Domain | Method   | URI                      | Name | Action                                              | Middleware |
+--------+----------+--------------------------+------+-----------------------------------------------------+------------+
|        | GET|HEAD | api/v1/documents         |      | App\Http\Controllers\DocumentsController@index      |            |
|        | GET|HEAD | api/v1/documents/user/{id} |    | App\Http\Controllers\DocumentsController@getByUserId |            |
|        | GET|HEAD | api/v1/documents/{id}    |      | App\Http\Controllers\DocumentsController@getById    |            |
+--------+----------+--------------------------+------+-----------------------------------------------------+------------+
```

Рисунок 4.3 – Інтерфейс взаємодії із сервісом documents-api

4.1.4 Мікросервіс «statistics-api»

По аналогії із іншими сервісами, мікросервіс *statistics-api* включає у себевеб-сервер *nginx* і службу *php-fpm* для обробки php-скриптів.

На рис. 4.4. зображено REST-сумісний інтерфейс взаємодії із сервісом statistics-api. Тобто, за допомогою GET запитів можна отримати статистику відвідувань по кожному користувачу.

```
+--------+----------+---------------------------+------+-----------------------------------------------------+------------+
| Domain | Method   | URI                       | Name | Action                                              | Middleware |
+--------+----------+---------------------------+------+-----------------------------------------------------+------------+
|        | GET|HEAD | api/v1/statistics         |      | App\Http\Controllers\StatisticsController@index     |            |
|        | GET|HEAD | api/v1/statistics/user/{id} |      | App\Http\Controllers\StatisticsController@getByUserId |            |
+--------+----------+---------------------------+------+-----------------------------------------------------+------------+
```

Рисунок – 4.4. Інтерфейс взаємодії із сервісом statistics-api

4.2.Стандартизація формату повідомлень

За допомогою специфікації JSON API було стандартизовано формат повідомлень для обміну між мікросервісами.

Для усіх API відповідей від сервісів, слідуючи стандарту, було додано заголовок: «Content-Type: application/json; charset=UTF-8».

Була створена бібліотека «JsonApi» на мові PHP для уніфікації формату повідомлень, що дало змогу спростити підтримку сервісів і покращило модульність коду. Усі повідомлення були стандартизовані у формат виду:

```
[
'data' => [],
'meta' => [],
'errors' => [],
]
```

Де, в ключ «data» записується інформація про запитуваний ресурс, «meta» – додаткова, довідкова інформація про ресурс, «errors» – інформація у разі виникнення помилки.

Приклад використання бібліотеки приведено нижче.

```
public function index(): JsonResponse
{
```

```
$data = new DataObject( AppUserModel::all()->toArray()
);
$meta = new MetaObject([
'container_id' => getenv('HOSTNAME'), 'service_name' => self::SERVICE_NAME,
]);
$jsonApi = new JsonApi($data, $meta);return response()->json(
$jsonApi->toArray(), 200, [], JSON_PRETTY_PRINT
);
}
```

Отже, в процесі виконання магістерскої работи було проаналізовано тривалість збірки кожного мікросервісного образу. Із табл. 4.1 видно, що сервіс *api-gateway*, має мінімальне значення, це пояснюється тим, що кодова база даного образу мінімальна. Таким чином, можна зробити висновок, що тривалість збірки є прямопропорційною величиною до розміру образу, тому для швидкого розгортання та збірки потрібен мінімальний розмір образу.

Таблиця 4.1 – Аналіз тривалості збірки образів

Сервіс	Кількість вимірів	Розмір образу (мегабайт)	Середня тривалість збірки (секунд)
users-api	20	204,7	195,4
documents-api	20	201,4	191,8
statistics-api	20	203,1	194,3
api-gateway	20	48,5	54,2

4.3 Опис процесу розробки програмного забезпечення

Система реєстру образів «Docker Hub» може автоматично створювати образ із вихідного коду у зовнішньому сховищі (наприклад, «GitHub») та автоматично пересилати упакований образ у репозиторій.

Під час налаштування автоматизованих збірок, потрібно створити список гілок та тегів, які потрібно упакувати. Після того як нова версія коду потрапляє у систему контролю версій, «Docker Hub» автоматично запускає

збірку образу на базі вихідного коду. Після того як збірка завершена, образ поміщується у репозиторій із присвоєним тегом.

Також існує можливість встановити значення змінних середовища, які використовуються у процесах збірки, що дає змогу динамічно виконувати ті чи інші інструкції при збірці. У випадку, якщо збірка завершилась із помилкою, існує можливість запустити процес знову.

На рис. 4.5 зображено налаштовану конфігурацію автоматизованих збірок образів.

Рисунок 4.5 – Конфігурація автоматизованих збірок

Завдяки опції кешування (анг. *Build Caching*) досягається зменшення тривалості збірки за рахунок збереження проміжних шарів образу. Кожен шар образу поміщається сховище «Docker Hub» і при наступній збірці, якщо хеш-сума поточного шару не змінилась, використовується шар із сховища, замість того щоб створювати новий.

Підхід на основі безперервної доставки дав змогу автоматизовано створювати версії Docker-образів без особистого запуску збірки.

Задля забезпечення відмово-стійкості додатку так звані кластери широко використовуються для підтримки важливих БД, для зберігання файлів у мережах, бізнес-пропозицій та систем обслуговування клієнтів, таких як веб-сайти електронної комерції.

Наведемо основі поняття, які застосовуються при проектуванні відмово-стійкої інфраструктури.

Відмово-стійкість (анг. *Fault Tolerance, FT*) – здатність системи до подальшої роботи після виходу з ладу будь-якого її елемента.

Кластер – група серверів (обчислювальних одиниць), об'єднаних каналами зв'язку.

Відмово-стійкий кластер (анг. *Fault Tolerant Cluster, FTC*) – кластер, відмова сервера у якому, не приводить до повної непрацездатності всього кластеру. Задачі виведеної з ладу машини розподіляються між однією або декількома іншими екземплярами додатку в автоматичному режимі.

Неперервна доступність (анг. *Continuous Availability, CA*) – концепція при якій, користувач може в будь-який момент використовувати службу, без переривань, навіть у разі відмови одного або декількох вузлів.

Висока доступність (анг. *High Availability, HA*) – у випадку виходу з ладу служби, система автоматично відновить працездатний стан, час простою мінімізується.

Перевага кластеризації для підвищення доступності стає очевидною у разі збою будь-якого вузла, при чому інший вузол кластера може взяти на себе навантаження несправного вузла, і, таким чином, користувачі не помітять переривання в доступі до сервісу. Кластеризація може бути здійснена на різних рівнях комп'ютерної системи, включаючи апаратне забезпечення, операційні системи, програми-утиліти, системи управління і додатки. Чим більше рівнів системи об'єднані кластерної технологією, тим вище надійність, масштабованість і керованість кластеру.

На рис. 4.6 зображено проектовану кластерну архітектуру. Єдиною точкою входу у додаток є сервіс «*api-gateway»*: веб-сервер « *nginx»* із відкритим портом 80. Тобто додаток знаходиться у приватній мережі, недоступній ззовні, крім порту 80 (HTTP). У свою чергу, екземпляри мікросервісів, у кількості трьох на кожен сервіс, теж знаходяться у приватній мережі.

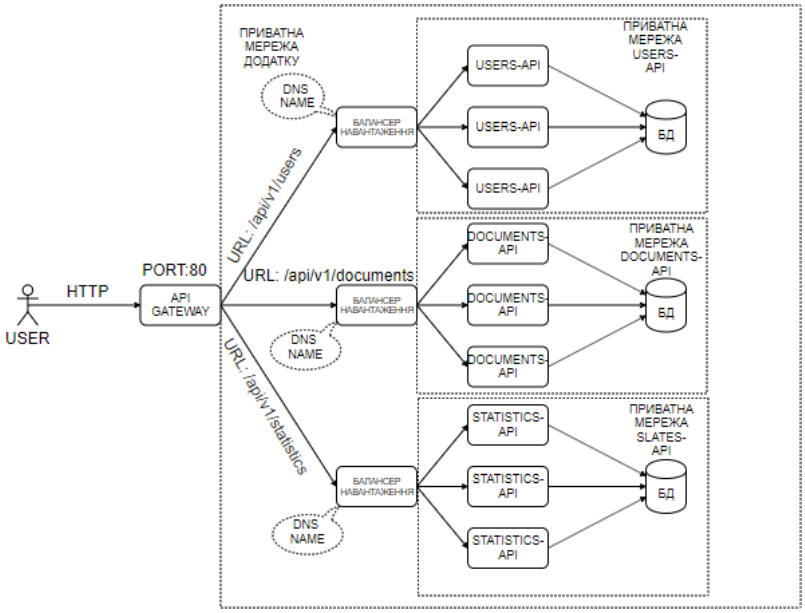


Рисунок 4.6 – Проектована кластерна інфраструктура

Три балансери навантаження виступають єдиною точкою входу до мікросервісів: запит потрапляє на балансер навантаження, після чого перенаправляється на потрібний, зазвичай менш навантажений у даний момент, екземпляр.

Зв'язок між мікросервісами досягається за рахунок DNS імен, які присвоєні кожному балансеру навантаження, тобто, якщо із сервіса *users-api* потрібно відправити запит до сервіса «*documents-api»*, потрібно використовуватиDNS ім’я балансера навантаження сервіса *documents-api*.

Використовуючи технологію «Docker Swarm», яка вбудована в службу «Docker», можна створити масштабований кластер на базі декількох фізичних машин (хостів).

Використовуючи «VirtualBox», було створено чотири віртуальні машині (рис. 4.7), які об’єднані в один кластер. Машина із назвою «*default»* буде виступати у якості менеджера, тобто головного экземпляру, який буде керувати кластером і збалансовувати навантаження між іншими машинами.

```
NAME             ACTIVE   DRIVER       STATE     URL                         SWARM   DOCKER     ERRORS
default          *        virtualbox   Running   tcp://192.168.99.100:2376           v19.03.5
swarm-worker     -        virtualbox   Stopped                                       Unknown
swarm-worker-2   -        virtualbox   Stopped                                       Unknown
swarm-worker-3   -        virtualbox   Stopped                                       Unknown
```

Рисунок 4.7 – Список віртуальних машин

Запуск кластера відбувається за допомогою команди (наведено нижче), де «swarm-docker» та «compose.yml» файли це конфігураційні файли кластеру:

«docker stack deploy --compose-file swarm-docker-compose.yml app»

Тривалість розгортання кластеру: 7.31 секунди.

У додатку А приведений вміст конфігураційного файлу кластера.

Режим виконання «Docker Swarm» автоматично займається розподіленням сервісів, і їх екземплярів у вигляді контейнерів, по чотирьом запущеним віртуальним машинам, які є вузлами кластеру.

Із результату виконання команди (рис. 4.8) видно, що сервіси *users-api, documents-api, statistics-api* були створені у кількості по 3 контейнери на кожен сервіс. Сервіс api-gateway та БД кожного сервіса були створені одиничними контейнерами.

```
docker@default: $ docker service ls
ID                  NAME                MODE                REPLICAS
r8mazuiasx87        app_api-gateway     replicated          1/1
qk93dfs6qpps        app_documents-api   replicated          3/3
ooapjb9mdzbj        app_documents-db    replicated          1/1
cidkfcovzlnp        app_statistics-api  replicated          3/3
y65uludclat8        app_statistics-db   replicated          1/1
nxrpk81hhzcl        app_users-api       replicated          3/3
biiqm82dej2h        app_users-db        replicated          1/1
```

Рисунок 4.8 – Список сервісів кластеру

Масштабування кластеру відбувається за допомогою команди:

«docker service scale app_users-api=5»

Тривалість масштабування сервісу: 3.89 секунд.

У даному випадку відбувалось масштабування сервісу *users-api* із трьох контейнерів до п’яти.

Коли необхідно зупинити виконання робочої машини (машини з роллю «Worker»), наприклад, для обслуговування або для її видалення з кластеру, треба використовувати режим «Swarm draining». Перехід фізичної машини зі стану «Active» в стан «Drain» призводить до того, що керуюча машина переміщує всі запущені завдання або контейнери з drain-варіанту на інші активні машини шляхом зупинки таких контейнерів на першій та запуску їх на активних машинах кластеру

Наприклад, щоб перевести хост-машину з іменем «swarm-worker» у режим обслуговування, потрібно ввести команду:

«docker node update --availability drain swarm-worker»

Статус хоста «керуюча» або «робоча» можна змінювати. «Docker Swarm» дозволяє використовувати дві і більше керуючих машини. Для досягнення більшої відмово-стійкості в разі виходу зі строю однієї керуючої машини рекомендується використовувати дві або більше керуючих хоста.

Наприклад, наступна команда переодить хост-машину «*swarm-worker»* із «робочої»у «керуючу»: *docker node promote swarm-worker.*

4.4 Забезпечення відмово-стійкості системи.

Відмово-стійкість кластеру гарантується самим «Docker». Це досягається в тому числі за рахунок того, що в кластері можуть одночасно працювати декілька керуючих хост-машин, які можуть в будь-який момент замінити лідера, що вийшов з ладу. Використовується так званий алгоритм підтримки розподіленого консенсусу – «Raft».

«Raft» реалізується поверх кластера одноманітних слабо-пов’язаних машин, на кожній з яких працює машина станів, таким чином, кожна хост-машина гарантовано стає в згоду з іншими машинами.

Відмово-стійкість сервісів, а саме екземплярів контейнерів даних сервісів, гарантується за допомогою реплікації: контейнери одного сервісу можуть розміщуватись на декількох хост-машинах (робітників з роллю «Worker»).

Наприклад, відключимо усі машини крім головної (*default*) за допомогою команди «*docker-machine.exe stop [node_name]»*. Після зупинки машин, усі контейнери мігрували на хост що залишився – *default*, що видно на рис. 4.9.

Тривалість міграції на доступну ноду: 12.67 секунд.

ID	NAME	NODE	DESIRED STATE	CURRENT STATE
intzhwvqw282	app_users-api.1	default	Running	Starting less than a second ago
tdof9stcggkb	app_api-gateway.1	default	Running	Running 4 seconds ago
u0h3dkli6dsy	app_statistics-db.1	default	Running	Running 9 seconds ago
7lofn2xgqu85	app_documents-db.1	default	Running	Running 11 seconds ago
bspknm7f99sd	app_users-db.1	default	Running	Running 12 seconds ago
5tfhl1v184ua	app_statistics-api.1	default	Running	Running 13 seconds ago
kvrfbl51ixai	app_documents-api.1	default	Running	Running 16 seconds ago
sfsxngk9dlkx	app_users-api.2	default	Running	Starting less than a second ago
m1v8ccxxyoh6	app_statistics-api.2	default	Running	Running 13 seconds ago
jbq8m2tmeca8	app_documents-api.2	default	Running	Running 16 seconds ago
mzin8tkt7111	app_users-api.3	default	Running	Starting less than a second ago
w2d82isp1g9s	app_statistics-api.3	default	Running	Running 13 seconds ago
lphdcl3rx1sv	app_documents-api.3	default	Running	Running 16 seconds ago

Рисунок 4.9 – Забезпечення відмово-стійкості сервісів

Для перевірки відмово-стійкості окремо взятого контейнера будемо використовувати симуляцію нештатної зупинки контейнера. Для зупинки контейнера використовується команда «*stop»*, наприклад: «*docker stop 783ee96f6500»*, де, *«783ee96f6500»* – унікальний ідентифікатор контейнера.

На рис. 4.10 видно, що сервіс «*users-api»*, включає тільки дві копії сервісу з трьох.

Тривалість запуску нового контейнера: 4.3 с.

```
docker@default: $ docker service ls
ID              NAME                MODE          REPLICAS
5411m9xiyhk2    app_api-gateway     replicated    1/1
3sg6xr1anyou    app_documents-api   replicated    3/3
bkhjfsgtw3bp    app_documents-db    replicated    1/1
wd7zn195n52k    app_statistics-api  replicated    3/3
49rhwychjzij    app_statistics-db   replicated    1/1
bvs54deujqy7    app_users-api       replicated    2/3
qh519471dqub    app_users-db        replicated    1/1
```

Рисунок 4.10 – Аварійна зупинка контейнера

Одним із недоліків мікросервісної архітектури є мережеві затримки при взаємодії сервісів один з одним.

Для мінімізації даного недоліку найчастіше використовуються принцип за яким контейнери, які часто взаємодіють між собою, розміщуються в одній мережі або у географічно близьких дата-центрах. Це дає змогу зменшити час очікування відповіді від іншого сервісу при синхронній взаємодії між

сервісами. У випадку використання асинхронної моделі взаємодії – сервіс не чекає відповіді і продовжує виконання інструкцій.

Проведемо дослідження тривалості затримки відповіді для REST-ресурсу: «api/v1/users/[user_id]».

У даному випадку запит перенаправляється на сервіс «users-api», який взаємодіє і отримує дані із сервісів «documents-api» і «statistics-api».

У табл. 4.2 приведено аналіз часу очікування відповіді від додатку. Загалом час очікування склав 1,09 с. При чому, найшвидше запит пройшов через сервіс «api-gateway», а найдовше – «users-api».

Отже, можна зробити висновок, що на маршрутизацію запиту витрачається найменше часу.

Таблиця 4.2 – Аналіз часу очікування відповіді.

Сервіс	Тривалість, секунди
Api Gateway	0,19
Users-api	0,38
Documents-api	0,22
Statistics-api	0,3
Усього	1,09

4.5 Моніторинг системи.

Моніторинг додатків і серверів додатків – важливий компонент кожної інфраструктури. Існує потреба у постійному моніторингу стану контейнерів, серверів, завантаження центрального процесора, споживання пам'яті, дискову утилізацію і т.д. Також є додаткова необхідність у сповіщені розробників, якщо у сервера закінчується доступна пам'ять або додаток перестає відповідати на запити, що дозволить запобігти проблемам, які можуть у перспективі призвести до відмови сервера.

«Prometheus» – система моніторингу різних систем і сервісів, за допомогою якої системні адміністратори можуть збирати інформацію про поточні параметри, налаштовувати конфігурацію для отримання інших повідомлень (наприклад, попередження, коли завантаження CPU перейшло за граничну межу).

Використовуючи «Prometheus», опишемо необхідні групи метрик:

– метрики хост-машини;

– метрики контейнерів;

– метрики «api-gateway» (єдина точка входу у додаток).

Вміст конфігураційного файлу «Prometheus» наведено нижче. Блок «*targets»* описує звідки брати метрики, у даному випадку це контейнери експортерів метрик. Інструкція «*scrape_interval»* вказує з яким інтервалом опитувати експортерів.

```
global: scrape_interval: 5s
evaluation_interval: 5sexternal_labels:
monitor: 'prometheus-grafana-exporter'

scrape_configs:
job_name: 'node-exporter'scrape_interval: 10s static_configs:
targets: ['node-exporter:9100']

job_name: 'nginx-exporter'scrape_interval: 10s static_configs:
targets: ['nginx-exporter:9113']

job_name: 'containers-exporter' scrape_interval: 10s
static_configs:
targets: ['cadvisor:8080']
```

Експортер – частина програмного забезпечення, яка отримує існуючі метрики від сторонньої системи і експортує їх в формат, зрозумілий для сервера «Prometheus».

На рис. 4.11 зображено графік використання CPU хост-машини «default». Ізграфіку видно, що CPU навантаження залишається стабільним на рівні ~12%. Це пояснюється тим, що система більшу частину часу ненавантажена, так як немає трафіку користувачів.

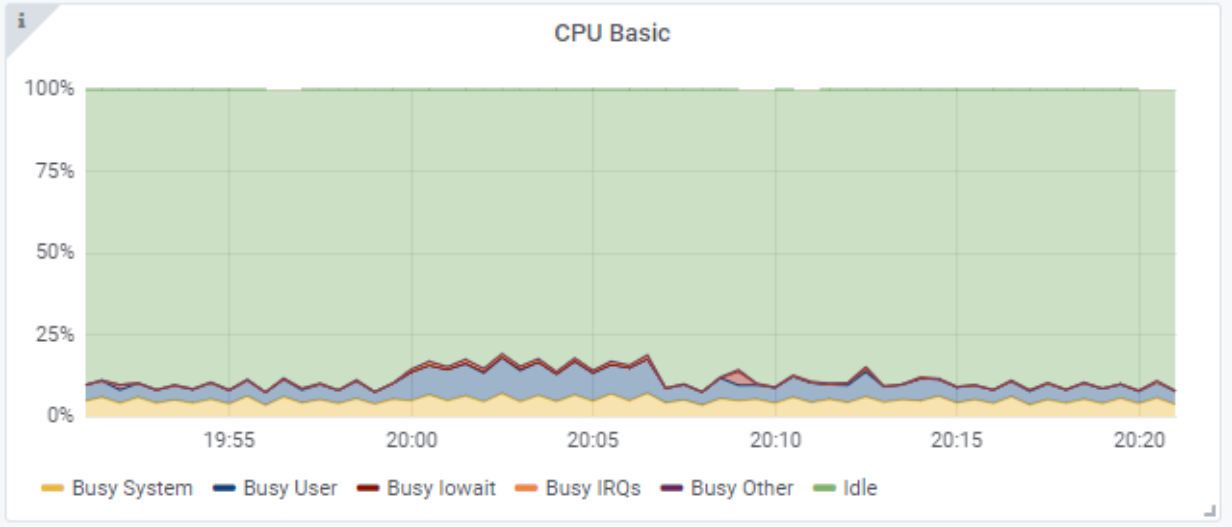


Рисунок 4.11 – Графік використання CPU хост-машиною

На рис. 4.12 зображено графік використання CPU кожним запущеним контейнером.

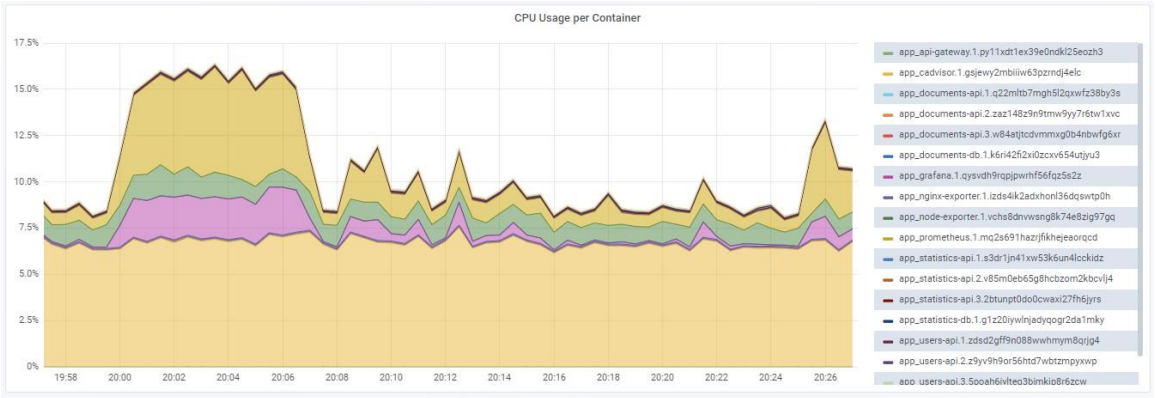
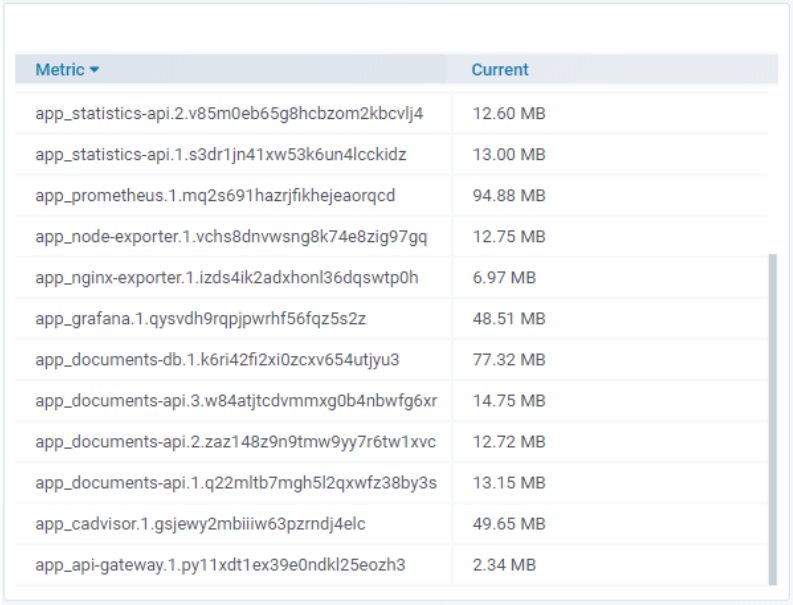


Рисунок 4.12 – Графік використання CPU контейнерами

Використовуючи даний графік можна визначити, який контейнер найбільше навантажує процесор і прийняти міри: наприклад, додати декілька екземплярів сервісу на інший хост.

Запит на отримання метрик для рис. 4.13 наведений нижче:

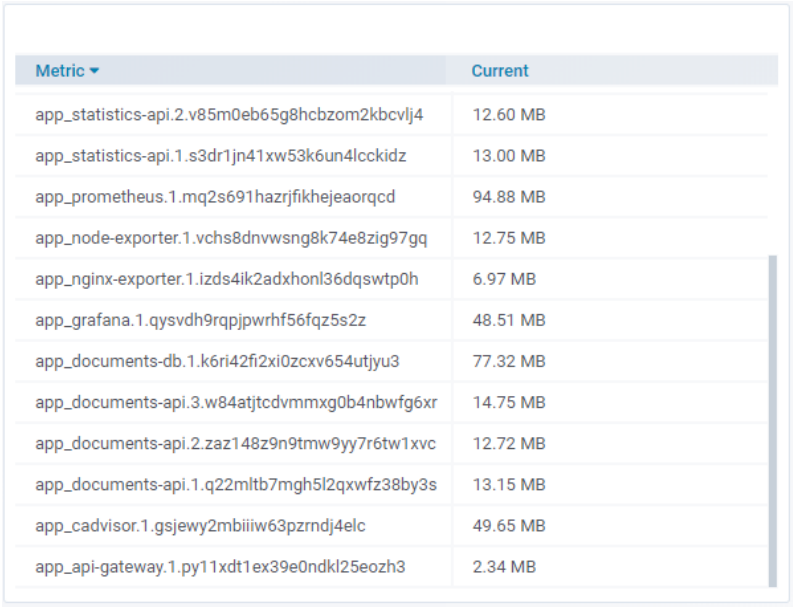


Metric	Current
app_statistics-api.2.v85m0eb65g8hcbzom2kbcvlj4	12.60 MB
app_statistics-api.1.s3dr1jn41xw53k6un4lcckidz	13.00 MB
app_prometheus.1.mq2s691hazrjfikhejeaorqcd	94.88 MB
app_node-exporter.1.vchs8dnvwsng8k74e8zig97gq	12.75 MB
app_nginx-exporter.1.izds4ik2adxhonl36dqswtp0h	6.97 MB
app_grafana.1.qysvdh9rqpjpwrhf56fqz5s2z	48.51 MB
app_documents-db.1.k6ri42fi2xi0zcxv654utjyu3	77.32 MB
app_documents-api.3.w84atjtcdvmmxg0b4nbwfg6xr	14.75 MB
app_documents-api.2.zaz148z9n9tmw9yy7r6tw1xvc	12.72 MB
app_documents-api.1.q22mltb7mgh5l2qxwfz38by3s	13.15 MB
app_cadvisor.1.gsjewy2mbiiiw63pzrndj4elc	49.65 MB
app_api-gateway.1.py11xdt1ex39e0ndkl25eozh3	2.34 MB

Рисунок 4.13 – Використання RAM контейнерами

На рис. 4.14 зображено таблицю використання RAM контейнерами із якої видно що система найбільше навантажена моніторинговими сервісами, які сканують метрики хоста, а інші сервіси простоюють так як мережевого трафіку немає.

Запит для отримання метрик використання RAM контейнерами наведенонижче:



Metric	Current
app_statistics-api.2.v85m0eb65g8hcbzom2kbcvlj4	12.60 MB
app_statistics-api.1.s3dr1jn41xw53k6un4lcckidz	13.00 MB
app_prometheus.1.mq2s691hazrjfikhejeaorqcd	94.88 MB
app_node-exporter.1.vchs8dnvwsng8k74e8zig97gq	12.75 MB
app_nginx-exporter.1.izds4ik2adxhonl36dqswtp0h	6.97 MB
app_grafana.1.qysvdh9rqpjpwrhf56fqz5s2z	48.51 MB
app_documents-db.1.k6ri42fi2xi0zcxv654utjyu3	77.32 MB
app_documents-api.3.w84atjtcdvmmxg0b4nbwfg6xr	14.75 MB
app_documents-api.2.zaz148z9n9tmw9yy7r6tw1xvc	12.72 MB
app_documents-api.1.q22mltb7mgh5l2qxwfz38by3s	13.15 MB
app_cadvisor.1.gsjewy2mbiiiw63pzrndj4elc	49.65 MB
app_api-gateway.1.py11xdt1ex39e0ndkl25eozh3	2.34 MB

Рисунок 4.14 – Використання RAM контейнерами

На рис. 4.15 зображено графік із інформацією про кількість прийнятих з’єднань сервісом «*api-gateway»* за визначений період часу.

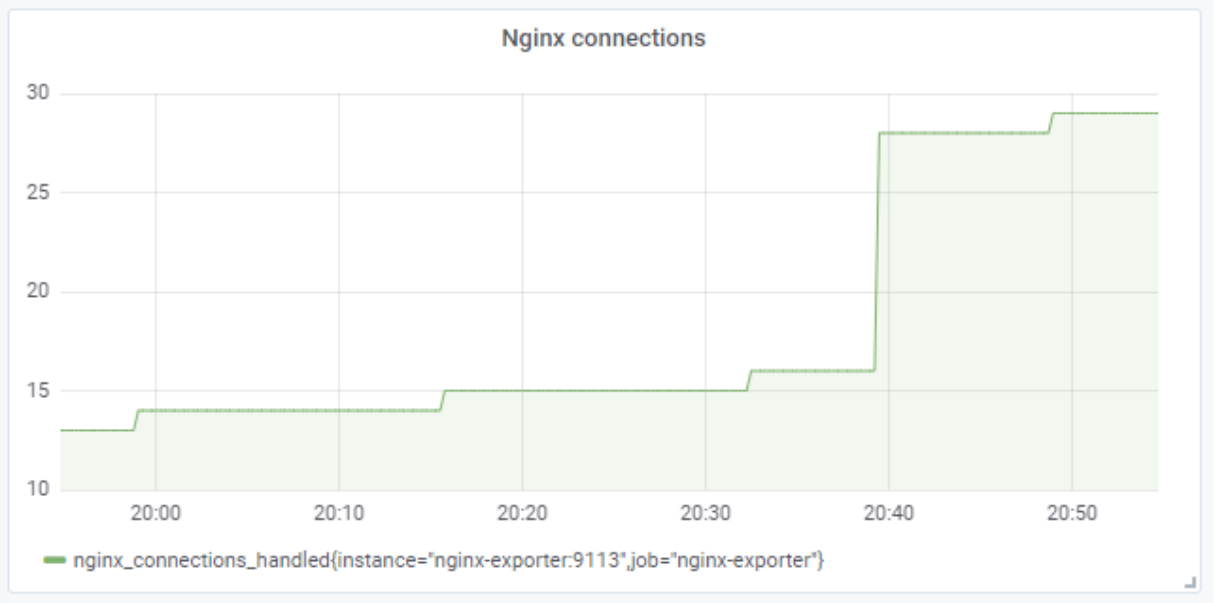


Рисунок 4.15 – Кількість прийнятих з’єднань «Nginx»

4.6 Тестування програмного забезпечення.

З метою тестування проведено дослід, де вивчили залежність використання CPU та диску від того, який з цих способів ми використовуємо. Ми отримали наступні результати:

Таблиця 4.3 – Використання «CPU and Disk» під час різних підходів до контейнеризації

№ п/п	*Підхід контейнеризації*	*1 мікросервіс*	*5 мікросервісів*	*10 мікросервісів*
1.	Один мікросервіс – один контейнер	5% CPU	22% CPU	31% CPU
		691 MB	2.5 GB	4.1 GB
2.	Усі мікросервіси в одному контейнері	5% CPU	27%CPU	64% CPU
		689 MB	3.1 GB	7 GB

На рис. 4.16 представлено залежності між кількістю мікросервісів та використанням CPU.

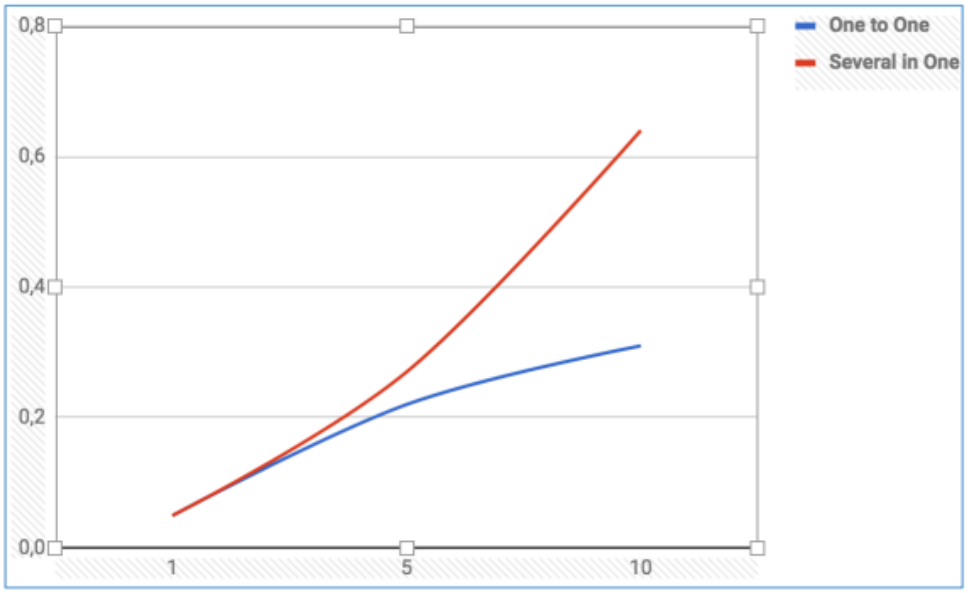


Рисунок 4.16 – Графік залежності між кількістю мікросервісів та використанням CPU

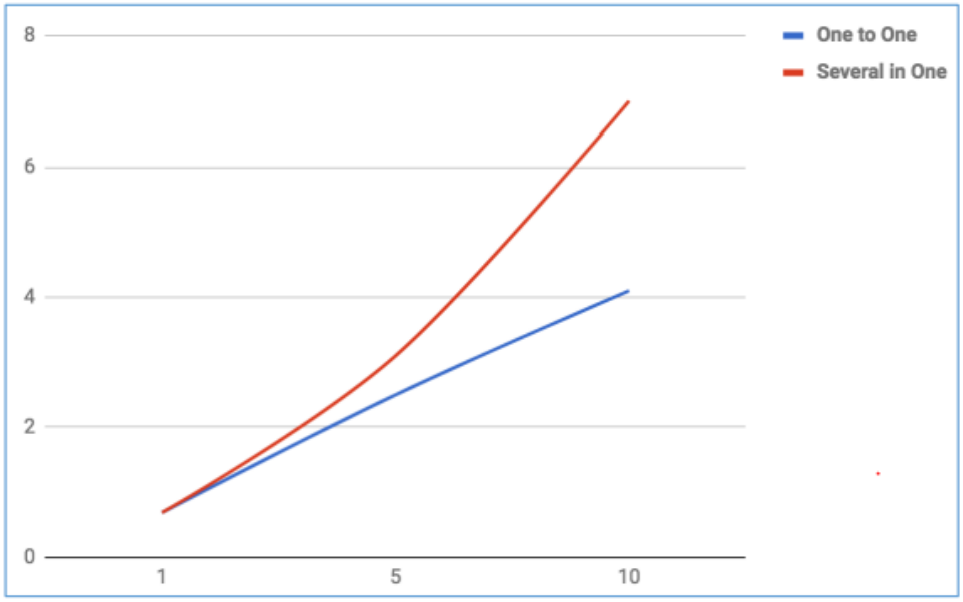


Рисунок 4.17 – Графік залежності між кількістю мікросервісів та використанням диску

З результатів досліду, можна зробити висновок, що шаблон “Один мікросервіс – один контейнер” виграє за критерієм економії ресурсів. У даному випадку ми бачимо, що є економія CPU та ресурсу диску.

ВИСНОВКИ

З метою досягнення поставленої задачі, а саме розробки комплексної системи управління сервісами з використанням засобів контейнеризації було проведено детальний аналіз предметної області і визначення поняття «сервіс» як динамічна система дій, спрямованих на отримання заданих результатів у багатокритеріальному полі протягом встановленого строку та в рамках виділених ресурсів із залученням виконавців, які володіють необхідними навичками та знаннями.

Проведено дослідження моделей управління сервісами та огляд існуючих аналогів. Аналіз дає можливість зробити висновок про те, більшість існуючих на ринку систем використовують застарілі методи управління системою, а програмний продукт часто є не повністю функціональним та мають незручне керування.

Також було здійснено аналітичний огляд літератури та патентної інформації, що включала в себе 2 патенти, результати досліджень наукової статті, книжки, матеріали конференцій

Зібрані данні дали можливість сформувати вимог до системи, зо дає можливість забезпечення продукту наступними компонентами: модульність, скомпонованість, повторюваність, відкритість, абстракція та синхронність, а також клієнтоорієнтованість.

Теоретичне обґрунтування та аналіз процесів управління дозволили розробити модель автоматизованої системи управління сервісами. Була проведена якісна оцінка розробленої моделі із застосуванням таких методів експертних оцінок та порівняльного аналізу рівня автоматизації бізнес-процесів підприємства при використанні різних моделей управління.

Було охарактеризовано та описано принципи побудови системи управління сервісами. В результаті наукового пошуку було оглянуто метод декомпозиції суб-домену, а також розглянуто шаблони проектування архітектури системи.

Отримані данні дають можливість здійснити вибір стратегії для розгортання серверної частини та додатків, а також провести попереднє проектування архітектури система. Всю інформацію будо представлено у відповідних структурних схемах.

На основі проведеного аналізу було здійснено вибір технічних та програмних засобів для реалізації системи. Систему було вирішено спроектування на базі мікросервісної архітектури .

Задачу стандартизації взаємодії мікросервісів один з одним було вирішена за допомогою специфікації «JSON API».

В рамках проведеного опису концепції контейнеризації та дослідження базових принципів та відмінностей технологій віртуалізації та контейнеризації саме систему «Docker» було обрано ля забезпечення відповідного програмного середовища, а також проаналізовано основні підходи до налаштування процесів постійної інтеграції. Також було наведено основні питання які стосуються проблем безпеки міжсервісного спілкування в системі, а також підходів до масштабування додатку.

Було детально описано процес встановлення та налаштування ПО «Docker». Для уникнення проблем при компіляції було проведено стандартне встановлення ПО «Docker Toolbox».

Аналіз архітектури віртуальної платформи «Docker» та дослідження процесів роботи з образами також мали місце в процесі написання магістерської роботи та надали можливість сформувати програмні вимоги до системи, серед яких є використання наступних технічних і програмних засобів: «Docker», «PHP», «Laravel», «PhpStorm», «MySQL», «VirtualBox», «Prometheus», «Grafana».

Було також розглянуто основні питання які стосуються проблем безпеки міжсервісного спілкування та забезпечення відмово-стійкості системи системі з відповідним тестування програмного забезпечення.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

[1] В. Б. Важоха, Я. М. Крайник «Інформаційні технології та інженерія» в Методи і засоби комп’ютерної інженерії. *Комплексна система управління сервісами з використанням засобів контейнеризації*, Миколаїв, 2022.

[2] Поняття "проект" [Онлайновий]. Available: http://www.pandia.ru/365896/. [Дата звернення: 21 грудня 2021].

[3] Бег'юлі, Ф. Управління проектом: пров. з англ. / Ф. Бегьюлі - М.: Гранд ФАІР-ПРЕС, 2002. - 202 с. [Дата звернення: 21 грудня 2021].

[4] 10. Івасенко, А. Г. Управління проектами / А. Г. Івасенко, Я. І. Ніконова, М. В. Каркавін - Ростов-на-Дону: Фенікс, 2009. - 327 с. [Дата звернення: 21 грудня 2021].

[5] 25. Фунтов, В. Н. Основи управління проектами у компанії. / В. Н. Фунтов - СПб.: Пітер, 2011. - 393 с.. [Дата звернення: 21 грудня 2021].

[6] Преображенська, Т.В. Управління проектами: навч. посібник/Т.В. Преображенська, М.Ш. Муртазіна, А.А. Алетдінова. - Новосибірськ: Вид-во НДТУ, 2018 - 123 с. [Дата звернення: 21 грудня 2021].

[7] Сербська, О.В. Сучасні методи управління проектами // Матеріали Афанасьєвських читань, №2, 2016 [Дата звернення: 21 грудня 2021].

[8] "Software Engineering - Guide to the software engineering body ofknowledge". ISO/IEC TR 19759:2015. [Дата звернення: 21 грудня 2021].

[9] Основи програмної інженерії з SWEBOK [Онлайновий]. Available: https://ligurio.github.io/swebok-ru/. [Дата звернення: 21 грудня 2021].

[10] Managing the Development of Large Software Systems: concepts and techniques. Rouce Winston, 1970, ICSE '87 Proceedings of the 9th International conferense on Software Engineering [Дата звернення: 21 грудня 2021].

[11] A Spiral Model of Software Development and Enhancement. Barry W.

Boehm,
TRW Defense System Group, 1988 [Дата звернення: 21 грудня 2021].

[12] Скотт, А. Гнучкі технології: екстремальне програмування та уніфікований процес розробки. Wiley (ISBN 0-471-20282-7), 2002 - Scott W. Ambler, Agile Modeling: Effective Practices for Extreme Programming, 2002; переклад та видання російською мовою: ЗАТ Видавничий дім "Пітер" (ISBN 5-94723-545-5) [Дата звернення: 21 грудня 2021].

[13] Шайхулова, А.Ф. Автоматизація та управління інноваційними проектами технічного переозброєння авіадвигунебудівного виробництва на основі каскадного методу оптимізації: Автореф ... дис. кан. тих. наук. - Уфа, 2018. -20 с. [Дата звернення: 21 грудня 2021].

[14] Норенков, І.П. Автоматизовані інформаційні системи: навч. посібник/І.П. Норенків. - М.: Вид-во МДТУ ім. н.е. Баумана, 2011. – 342 [2] с.: іл. – (Інформатика у технічному університеті [Дата звернення: 21 грудня 2021].

[15] Чуланова, О. Л. Технологія управління проектами та проектними командами на основі методології гнучкого управління проектами Agile // Вісник євразійської науки. – 2018. – №1.-З 31-35 [Дата звернення: 21 грудня 2021].

[16] PMBOK® Guide – Sixth Edition // Project Management Institute [Онлайновий]. Available: https://www.pmi.org/pmbok-guide-standards/foundational/pmbok [Дата звернення: 21 грудня 2021].

[17] Advantages and Disadvantages of Trello // SoftwareDeveloperIndia [Онлайновий]. Available:https://www.software-developer-india.com/advantages-anddisadvantages-of-trello . [Дата звернення: 21 грудня 2021].

[18] What is Asana? CompareCamp [Онлайновий]. Available: http://comparecamp.com/asana-reviews-pricing-benefits-and-features-analysis/

[Дата звернення: 21 грудня 2021].

[19] Trello vs Asana: The Best Project Management App in 2017? [Онлайновий]. Available: https://www.process.st/trello-vs-asana/. [Дата звернення: 21 грудня 2021].

[20] Managing the Development of Large Software Systems: concepts and techniques [Онлайновий]. Available: https://smartercommunities.media/pioneering-management. [Дата звернення: 21 грудня 2021].

[21] Макашов, П. Л., Романенко, Н. А. Сервіс-орієнтований підхід до управління IT проектами на прикладі використання програмного продукту "Jira" // Сучасні інформаційні технології та IT- освіта. – 2015. – №11. - С.12-16. [Дата звернення: 21 грудня 2021].

[22] Product Guides & Tutorials // Atlassian [Онлайновий]. Available: https://www.atlassian.com/software/jira/guides/getting-started/overview. [Дата звернення: 21 грудня 2021].

[23] Benefits of Using JIRA [Онлайновий]. Available: https://www.dumbfunded.co.uk/guides/6-benefits-of-using-jira/. [Дата звернення: 21 грудня 2021].

[24] Життєвий цикл проектного завдання [Онлайновий]. Available: http://projectimo.ru/upravlenie-proektami/zhiznennyj-cikl-proekta.html . [Дата звернення: 21 грудня 2021].

[25] Патент US10237118B2. Efficient application build/deployment for distributed container cloud platform [Онлайновий]. Available: https://patentimages.storage.googleapis.com/1b/96/8f/8f3a10eb000362/US10237118.pdf [Дата звернення: 21 грудня 2021].

[26] Патент US11120299B2. Installation and operation of different processes of an AI engine adapted to different configurations of hardware located on-premises and in hybrid environments [Онлайновий]. Available:

https://patents.google.com/patent/US11120299B2/en?q=docker+service+system&oq=docker+service+system. [Дата звернення: 21 грудня 2021].

[27] ІН’ЄКТИВНИЙ МЕТОД ОТРИМАННЯ ДАНИХ КОРИСТУВАЦЬКОГО ДОСВІДУ В ІГРОВИХ СИМУЛЯТОРАХ КОМП’ЮТЕРНИХ МЕРЕЖ [Онлайновий]. Available: https://doi.org/10.31649/1997-9266-2019-146-5-49-54 . [Дата звернення: 21 грудня 2021].

[28] ВИКОРИСТАННЯ КОНТЕЙНЕРИЗАЦІЇ ПРИ РОЗРОБЛЕННІ ЛАБОРАТОРНИХ ПРАКТИКУМІВ СТУДЕНТІВ [Онлайновий]. Available http://elartu.tntu.edu.ua/bitstream/lib/27290/2/IMST_2018_Lutskiv_A_M-Vykorystannia_konteineryzatsii_97.pdf . [Дата звернення: 21 грудня 2021].

[29] Martin Fowler, Microservices. [Онлайновий]. Available: https://martinfowler.com/articles/microservices.html [Дата звернення: 21 грудня 2021].

[30] Pethuru Raj. Docker: Creating Structured Containers. - Packt Publishing, 2016. - 320 с. [Дата звернення: 21 грудня 2021].

[31] Rajesh RV, Spring Microservices. - Packt Publishing, 2016. - 436 с [Дата звернення: 21 грудня 2021].

[32] Randall Smith. Docker Orchestration. - Packt Publishing, 2017. - 284 с. [Дата звернення: 21 грудня 2021].

[33] The Italian town pioneering blockchain for waste management [Онлайновий]. Available: https://smartercommunities.media/the-italian-town-pioneering-blockchain-for-waste-management. [Дата звернення: 21 грудня 2021].

[34] Kubernetes, Kubernetes Documentation [Онлайновий]. Available: https://kubernetes.io/docs/home/ [Дата звернення: 21 грудня 2021].

ДОДАТОК А

ВИХІДНИЙ КОД КОНФІГУРАЦІЙНОГО ФАЙЛУ «DOCKER SWARM»

```
version: "3.7"
services:
 api-gateway:
 image: api-gateway:latest
 deploy:
 replicas: 1
 ports:
 - 80:80
 depends_on:
 - users-api
 - documents-api
 - statistics-api

 users-api:
 image: zhenia97chap/users-api:latest
 deploy:
 replicas: 3
 depends_on:
 - users-db
 documents-api:
 image: documents-api:latest
 deploy:
 replicas: 3
 depends_on:
 - documents-db

 statistics-api:
 image: statistics-api:latest
 deploy:
 replicas: 3
 depends_on:
 - statistics-db
 users-db:
 image: mysql:5.7.21
 deploy:
 replicas: 1
 environment:
 MYSQL_USER: root
 MYSQL_ROOT_PASSWORD: password
 MYSQL_DATABASE: users-api
 volumes:
```

```
- db_users_data:/var/lib/mysql
 documents-db:
 image: mysql:5.7.21
 deploy:
 replicas: 1
 environment:
 MYSQL_USER: root
 MYSQL_ROOT_PASSWORD: password
 MYSQL_DATABASE: documents-api
 volumes:
 - db_documents_data:/var/lib/mysql
 statistics-db:
 image: mysql:5.7.21
 deploy:
 replicas: 1
 environment:
 MYSQL_USER: root
 MYSQL_ROOT_PASSWORD: password
 MYSQL_DATABASE: statistics-api
 volumes:
 - db_statistics_data:/var/lib/mysql

prometheus:
 image: prom/prometheus:v2.15.2
 deploy:
 placement:
 constraints:
 - node.role == manager
 volumes:
 - ./prometheus/:/etc/prometheus/
 - prometheus_data:/prometheus
 command:
 - '--config.file=/etc/prometheus/prometheus.yml'
 - '--storage.tsdb.path=/prometheus'
 - '--web.console.libraries=/etc/prometheus/console_libraries'
 - '--web.console.templates=/etc/prometheus/consoles'
 - '--storage.tsdb.retention=200h'
 - '--web.enable-lifecycle'
 ports:
 - 9090:9090
 node-exporter:
 image: prom/node-exporter:v0.18.1
 user: root
 volumes:
 - /proc:/host/proc:ro
- /sys:/host/sys:ro
 - /:/rootfs:ro
 command:
```

```
 - '--path.procfs=/host/proc'
 - '--path.sysfs=/host/sys'
 - '--collector.filesystem.ignored-mount-points=^/(sys|proc|dev|host|etc)($$|/)'

 nginx-exporter:
 image: nginx/nginx-prometheus-exporter:0.5.0
 environment:
 - SCRAPE_URI=http://192.168.99.100/nginx_status
 - TELEMETRY_PATH=/metrics
 - NGINX_RETRIES=10
 logging:
 driver: "json-file"
 options:
 max-size: "5m"

 cadvisor:
 image: google/cadvisor:latest
 volumes:
 - /:/rootfs:ro
 - /var/run:/var/run:rw
 - /sys:/sys:ro
 - /var/lib/docker/:/var/lib/docker:ro
 grafana:
 image: grafana/grafana:latest
 volumes:
 - grafana_data:/var/lib/grafana
 environment:
 - GF_SECURITY_ADMIN_USER=admin
 - GF_SECURITY_ADMIN_PASSWORD=admin
 - GF_USERS_ALLOW_SIGN_UP=false
 ports:
 - 3000:3000
volumes:
 db_users_data:
 driver: local
 db_documents_data:
 driver: local
 db_statistics_data:
 driver: local
 prometheus_data:
 driver: local
 grafana_data:
 driver: local
```

ДОДАТОК Б

DOCKERFILE МІКРОСЕРВІСІВ «USERS-API», «DOCUMENTS-API», «STATISTICS-API»

```
FROM php:7.2-fpm
RUN apt-get update \
 && apt-get -y install git \
 && apt-get -y install zip \
 && apt-get -y install procps \
 && apt-get -y install htop \
 && apt-get -y install nano \
 && apt-get -y install supervisor \
 && apt-get -y install net-tools \
 && apt-get -y install nginx
RUN docker-php-ext-install pdo_mysql
COPY . /var/www
COPY docker/supervisor/conf.d /etc/supervisor/conf.d/
COPY docker/nginx/conf.d/default.conf /etc/nginx/conf.d
RUN chown -R www-data:www-data /var/www
RUN rm /etc/nginx/sites-available/default /etc/nginx/sites-enabled/default
WORKDIR /var/www
RUN curl -sS https://getcomposer.org/installer | php -- --
installdir=/usr/local/bin --filename=composer
RUN composer install -n --prefer-dist --ignore-platform-reqs
RUN mv .env.example .env
RUN php artisan key:generate
CMD ["/usr/bin/supervisord", "-n"]
```

ДОДАТОК В

КОНФІГУРАЦІЯ ВЕБ-СЕРВЕРА СЕРВІСУ «API-GATEWAY»

```
server {
 listen 80;
 index index.php index.html;
 root /var/www/public;
 access_log /var/log/nginx/access.log;
 error_log /var/log/nginx/error.log;
 fastcgi_split_path_info ^(.+\.php)(/.+)$;
 fastcgi_index index.php;
 include fastcgi_params;
 fastcgi_param SCRIPT_FILENAME ${document_root}/index.php;
 fastcgi_param PATH_INFO $fastcgi_path_info;

 location / {
 return 200 'App is alive';
 add_header Content-Type text/plain;
 }
 location /api/v1/users {
 rewrite ^(.*)/$ $1 permanent;
 fastcgi_pass users-api:9000;
 }
 location /api/v1/documents {
 rewrite ^(.*)/$ $1 permanent;
 fastcgi_pass documents-api:9000;
 }
 location /api/v1/statistics {
 rewrite ^(.*)/$ $1 permanent;
 fastcgi_pass statistics-api:9000;
 }
}
```