НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Навчально-науковий інститут телекомунікаційних систем

Кафедра телекомунікацій

«На правах рукопису»
УДК _____

«До захисту допущено»
Завідувач кафедри
_____ Сергій КРАВЧУК
«___»_____2021 р.

Магістерська дисертація
на здобуття ступеня магістра
за освітньо-професійною програмою «Інженерія та програмування інфокомунікацій»
зі спеціальності 172 «Телекомунікації та радіотехніка»

на тему: « Метод управління навантаженням в мережі кластера віртуалізації Kubernetes »

Виконав:
студент VI курсу, групи ТЗ-01мп
Педько Андрій Дмитрович _____

Керівник:
Ст.викладач кафедри ТК, к.т.н.,
Маньківський В.Б. _____

Рецензент:
Доцент кафедри ІКТС НН ІТС, к.т.н., доцент,
Созонник Г.Д. _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2021 року

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

Інститут телекомунікаційних систем
Кафедра телекомунікацій

Рівень вищої освіти – другий (магістерський)

Спеціальність – 172 «Телекомунікації та радіотехніка»

Освітньо-професійна програма «Інженерія та програмування інфокомунікацій»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Сергій КРАВЧУК
«___»_____2021 р.

ЗАВДАННЯ
на магістерську дисертацію студенту

Педьку Андрію Дмитровичу

1. Тема дисертації « Метод управління навантаженням в мережі кластера віртуалізації Kubernetes », науковий керівник дисертації старший викладач кафедри ТК Маньківський Володимир Броніславович, к.т.н., затверджені наказом по університету від «04» листопада 2021 р. № 3673-с.

2. Термін подання студентом дисертації 10.12.2021 р.

3. Об’єктом дослідження - додаток розвернутий в мережі кластера Kubernates.

4. Предмет дослідження - uptime, response time, та одночасна кількість користувачів підключенна до сервісу.

5. Перелік завдань, які потрібно розробити:

1. Пошук тестового додатку для розгортання.

2. Підготовка Docker контейнерів та файлів конфігурації для pod та deployments.

3.Налаштування Apache JMeter та додавання плагінів для побудови графіків та Stepping Thread Group.

4. Розгортання додатків в 3 різних конфігурацій та тестування.

5. Розробка рекомендацій що до балансування навантаження в мережі кластеру Kubernetes.

6. Розробити стартап-проект

6. Орієнтовний перелік ілюстративного матеріалу:

1. Графічне зображення принципу роботи додатку.

2. Графічне зображення pods та deployments в розгорнутому додатку.

3. Графічне зображення результатів Get запитів.

4. Графічне зображення результатів Post запитів та кількості підключених користувачів до додатку.

5. Графічне зображення площини управління Kubernetes.

7. Орієнтовний перелік публікацій

8. Дата видачі завдання “ 1 ” липня 2021 р.

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1	Ознайомлення з темою, складання спільно з керівником практики робочого плану проходження практики, пошук мікросервісного додатку для подольшого його використання.	01.07.2021– 16.07.2021	виконано
2	Збір інформації про технологію оркестрації контейнерів Kubernetes. Порівняння з Docker (Docker Swarm).	22.07.2021 – 13.08.2021	виконано
3	Створення контейнерів та розміщення їх на Docker hub. Збір інформації для правильного написання маніфестів Kubernetes.	24.08.2021 – 29.08.2021	виконано

4	Вибір параметрів для порівняння ефективності додатку реалізованому на Kubernetes та Docker.	01.09.2021 – 15.09.2021	виконано
5	Реалізація додатку в кластері Kubernetes та вимірювання параметрів. Порівняння параметрів з розверненням цього ж додатку з використанням інших технологій.	19.09.2021 – 05.10.2021	виконано
6	Написання першого та другого розділів.	08.10.2021 – 16.10.2021	виконано
7	Налаштування інструменту для тестування та написання третього та четвертого розділів.	22.10.2021 – 14.11.2021	виконано
8	Розробка стартап проекту.	16.11.2021 – 02.12.2021	виконано

Студент Андрій ПЕДЬКО

Науковий керівник дисертації Володимир МАНЬКІВСЬКИЙ

РЕФЕРАТ

Магістерська дисертація містить: 87 сторінок, 89 рисунків, 16 таблиць, 13 посилань.

В роботі описана технологія Kubernetes та описано як розвивалися методи розгортання додатків. В теоретичній частині описані особливості цієї технології, її переваги та недоліки. Також розглянуті передумови розвитку цієї технології та порівняння її з конкурентами.

В практичній частині створено Docker контейнери та manifest файли для мікросервісів. Після чого розгорнуто веб-додаток в 3 різних конфігураціях та перевірено такі показники як: response time для GET і POST запитів та кількість підключених користувачів до додатку. Було налаштовано інструмент для тестування на стійкість до навантажень Apache JMeter. За результатами досліджень були розроблені рекомендації, котрі підвищують ефективність використання системних ресурсів та стійкість сервісу до навантажень.

Ключові слова: Kubernetes кластер, response time, відмовостійкість, Docker, pod, deployment, ефективність.

ABSTRACT

Master's dissertation contains: 87 pages, 89 figures, 16 tables, 13 lines.

The dissertation describes Kubernetes technology and describes how application development methods have developed. The theoretical part describes the features of this technology, its advantages and disadvantages. The prerequisites for the development of this technology and its comparison with competitors are also considered.

In the practical part, Docker containers and manifest files for microservices have been created. Then the web application was deployed in 3 different configurations and checked such indicators as: response time for GET and POST requests and the number of users connected to the application. The Apache JMeter Load Test Tool has been configured. Based on the results of research, recommendations have been developed that increase the efficiency of using system resources and the resilience of the service to loads.

Keywords: Kubernetes cluster, response time, fault tolerance, Docker, pod, deployment, efficiency.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

API	application programming interface
K8S	Kubernetes
YAML	Yet Another Markup Language
SSH	Secure Shell
DNS	Domain Name System
IP	Internet Protocol
RPM	RPM Package Manager
HDFS	Hadoop Distributed File System
DevOps	Development Operations
OCI	Open Container Initiative
CI/CD	Continuous Integration/Continuous Delivery
PyP	Primary Years Programme
URL	Uniform Resource Locator
CRI-O	Container Runtime Interface
CLI	Command Line Interface
JSON	JavaScript Object Notation
UID	User identifier
FTP	File Transfer Protocol
LDAP	Lightweight Directory Access Protocol
SOAP	Simple Object Access Protocol
JMS	Java Message Service
POP3	Post Office Protocol Version 3
IMAP	Internet Message Access Protocol
HTTP	HyperText Transfer Protocol
ОЗУ	Оперативна пам’ять
ОС	Операційна система
ВМ	Віртуальна машина

ВСТУП

З розвитком індустрії інформаційних технологій попит на високодоступні, відмовостійкі та ефективні сервіси постійно зростає. Велика кількість користувачів мережі Інтернет змушує впроваджувати та розробляти нові методики для побудови складних інформаційних систем. Потреба в цьому може бути знайдена в будь-яких галузях — від розваг до наукових досліджень. Однак є кілька проблем, які перешкоджають можливостям мережевих додатків. Одним з них є використання застарілих технологій та архітектурних рішень.

Використання застарілих методик та технологій призводить до поганого масштабування додатку, оскільки на тей час Інтернет не був настільки важливим в повсякденному житті та була значно менша кількість користувачів. Тому додатки котрі погано масштабуються не задовольняють потреби сучасного інформаційного суспільства. Таким чином, розробка програмного забезпечення у вигляді мікросервісів стає на передній план, порівнюючи з застрілим монолітом.

В сучасному житті широкого розповсюдження набули додатки написані на основі мікросервісної архітектури. Мікросервісна архітектура – це рішення, яке може оптимізувати процес розробки програмного забезпечення та полегшити його підтримку[1]. На сьогоднішній день, однією з найбільш популярних систем для розгортання мікросервісних додатків є Kubernetes. Kubernetes – це система з організацією контейнерів з відкритим кодом для автоматизації розгортання, масштабування та управління[1].

Дана магістерська дисертація присвячена створенню рекомендацій що до балансування навантаження в кластері Kubernetes.

Областю дослідження є процес управління навантаженням в мережі кластеру Kubernetes. Об’єктом дослідження виступає додаток розвернутий в мережі кластера Kubernates. Предметом дослідження являються uptime, response time, та одночасна кількість користувачів підключенна до сервісу.

Метою роботи є підвищення ефективності використання системних ресурсів та відмовостійкості сервісу.

Основною задачею дослідження являється розробка рекомендацій та їх практична перевірка спрямованих на підвищення відмовостійкості та

ефективності сервісу розгорнутого на Kubernetes кластері. Проміжними задачами можна визначити порівняльний аналіз результатів відносно різних конфігурацій кластера, підходів для визначення продуктивності, а також вибір засобів для побудови системи.

РОЗДІЛ 1
ХАРАКТЕРИСТИКА ТЕХНОЛОГІЇ KUBERNETES

1.1 Особливості технології

Kubernetes — це платформа з відкритим кодом, яка автоматизує операції з контейнерами. Вона спрощує багато ручних процесів, пов'язаних з розгортанням та масштабуванням упакованих у контейнери додатків. Таким чином, можна об'єднувати групи хостів, які працюють під керуванням контейнерів Linux, і Kubernetes допоможе ефективно керувати цими кластерами, які можуть охоплювати вузли у загальнодоступних, приватних чи гібридних хмарах. Тому Kubernetes є ідеальною платформою для розміщення хмарних програм, що потребують швидкого масштабування, таких як потокова передача даних у реальному часі через Apache Kafka.

Давайте повернемося назад і подивимося, чому Kubernetes так корисний.

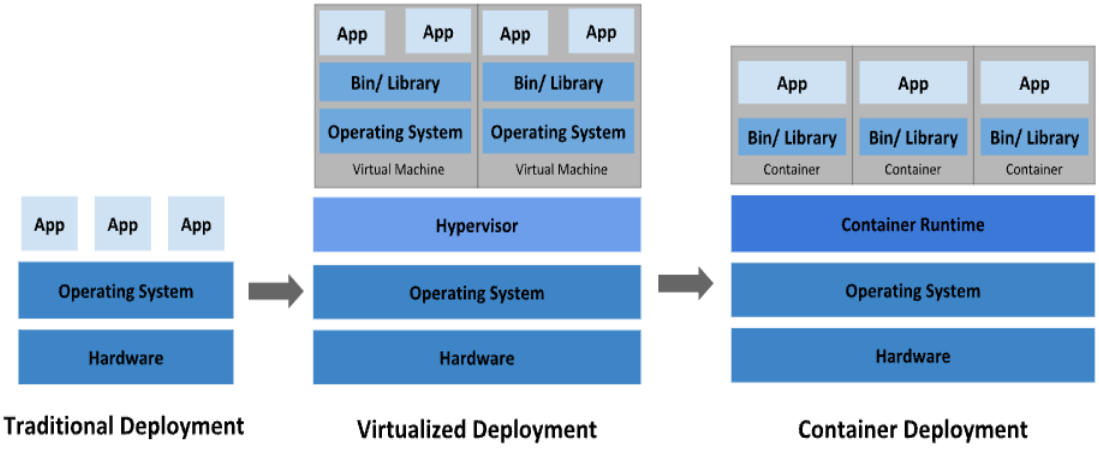

Рис. 1.1 Розвиток методів розгортання мережевих додатків

Традиційна ера розгортання: Раніше організації запускали додатки на фізичних серверах (Рис. 1.1). Не було ніякого способу визначити межі ресурсів для додатків на фізичному сервері, і це викликало проблеми з розподілом ресурсів. Наприклад, якщо кілька додатків виконуються на фізичному сервері, можуть бути випадки, коли один додаток буде займати більшу частину ресурсів, і в результаті чого інші додатки будуть працювати гірше. Рішенням цього було запустити кожен додаток на іншому фізичному сервері. Але це не масштабувалося, оскільки ресурси використовувалися не повністю, через що організаціям було складно підтримувати безліч фізичних серверів[2].

Ера віртуального розгортання: Для вирішення була представлена віртуалізація. Вона дозволила запускати кілька віртуальних машин (ВМ) на одному фізичному сервері (Рис. 1.1). Віртуалізація ізолює додатки між віртуальними машинами і забезпечує певний рівень безпеки, оскільки інформація одного додатка не може бути вільно доступна іншому додатку.

Віртуалізація дозволяє краще використовувати ресурси на фізичному сервері і забезпечує кращу масштабованість, оскільки додаток можна легко додати або оновити, крім цього знижуються витрати на обладнання і багато іншого. За допомогою віртуалізації можна перетворити набір фізичних ресурсів в кластер одноразових віртуальних машин[2]. Але, оскільки кожна віртуальна має свою ОС поверх хостової розвиток методів розгортання на цьому не зупинився і продовжувались пошуки більш ефективних. Оскільки встановлення операційної системи використовує велику кількість системних ресурсів, що робить такий метод не раціональним з економічної точки зору.

Ера контейнерів: Контейнери схожі на віртуальні машини, але у них є властивості ізоляції для спільного використання операційної системи (ОС) між додатками (Рис 1.1). Тому контейнери вважаються легкими. Подібно віртуальній машині, контейнер має свою власну файлову систему, процесор, пам'ять, простір процесу і багато іншого[2]. Також контейнери Docker можна з легкістю переноси на сервери при цьому не налаштовую змінні середовища та не встановлючи залежності на хостову ОС.

За допомогою використання контейнерів можна з легкістю розгорнути додатки на будь-якому сервері. Для компаній з власним продуктом необхідно управляти додатками контейнерами, на яких розгорнутий додаток та гарантувати відсутність простоїв та стійкість до високих навантажень, оскільки з економічної точки зору, такі простої можуть завдати компанії великих збитків та втрачених клієнтів. Тому, якщо контейнер виходить з ладу, необхідно його перезапустити або швидко створити новий.

В такому випадку, Kubernetes і приходить на допомогу! Kubernetes дає вам фреймворк для гнучкої роботи розподілених систем. Він займається масштабуванням і обробкою помилок в додатку, надає шаблони розгортання і

багато іншого. Також має можливість автоматично перезапускати контейнери та створювати нові, що мінімізує час простою додатку.

Kubernetes потужний інструмент, якщо необхідно оптимізувати робочу версію програми для хмари, полягає в отриманні платформи для планування та запуску контейнерів на кластерах фізичних або віртуальних машин. У ширшому сенсі це допомагає повністю реалізувати надійну інфраструктуру на основі контейнерів у виробничих середовищах.

З Kubernetes можна:

- організувати контейнери на кількох хостах;
- максимально ефективно використовувати ресурси, необхідні для запуску корпоративних програм;
- керувати та автоматизувати розгортання та оновлення додатків;
- монтувати та додавати сховища для запуску додатків із відстеженням стану;
- масштабувати контейнерні програми та їх ресурси «на льоту»;
- перевіряти працездатність та самовідновлення додатків за допомогою автозаповнення, автозапуску, автореплікації та автомасштабування.

1.2 Переваги технології

Kubernetes (часто скорочують до K8s) - відкрита система оркестрації контейнерів, представлена компанією Google в 2014 році. K8s дозволяє управляти серверами, на яких запущені контейнери, і самими контейнерами на них, а також підвищує утилізацію заліза, запускаючи безліч контейнерів на одному сервері[3].

З точки зору архітектури в кластер Kubernetes входять головні вузли, які надають базові служби Kubernetes і оркеструються робочі навантаження додатки, і підлеглі вузли, які виконують робочі навантаження додатки, їх називають нодамі. Система може складатися з декількох кластерів.

Програмне управління релізами

У Kubernetes кілька переваг для розробників.

У бойовому середовищі (prod) використовуються рівно ті ж контейнери, які тестувалися в dev-середовищі. Так що якщо додаток працює в тесті, то,

швидше за все, воно буде працювати і в бою. Організувати таке з менеджером пакетів RPM складніше, оскільки доводиться пам'ятати про залежності. А при поставці коду в контейнері всі залежності вже в ньому містяться.

Kubernetes стежить, щоб стан кластера відповідав бажаному, при необхідності перезапускає потрібні компоненти. Kubernetes дозволяє программно управляти релізами на серверах під своїм управлінням, він може:

- випустити реліз однією кнопкою;
- швидко повернутися до попередньої версії релізу;
- проводити a / b тестування;
- виконувати оновлення додатку поступово (за відсотками), стежачи за показниками моніторингу, щоб швидко виявити можливі помилки в новій версії;
- автоматично збільшити або зменшити розмір кластера, тобто додати або прибрати Ноди в залежності від навантаження (якщо використовувати хмарні провайдери такі як: Amazon, GCP, Microsoft Azure);
- стежити, щоб була рівно задана кількість інстансів додатку, наприклад - швидко доводити їх до потрібного числа при втраті частини інстансів через будь-який збій.

Відмовостійкість

Для додатків на фізичному сервері потрібно продумувати схему його відновлення при збоях:

- якщо додаток працює на одному сервері і він виходить з ладу - треба думати, як організувати його автоматичне розгортання в новому місці;
- якщо додаток працює в декількох копіях паралельно, щоб при падінні однієї завжди залишалася робоча копія, - потрібно налаштовувати автоматичне балансування навантаження.

У Kubernetes ці проблеми вирішені автоматично: він сам розподіляє додатки так, щоб вони продовжили роботу в разі збою на конкретному фізичному сервері, а також з урахуванням навантаження на них. Наприклад, якщо зона відмови - дата-центр, то він буде розподіляти додаток за різними

дата-центрам. Таким чином, втрата дата-центру може ніяк не вплинути на сервіс.

Kubernetes - це автоматизація процесів

Додатки в k8s розгортаются і тестуються без участі адміністраторів. Так що в Kubernetes вся операційна підтримка роботи софта лежить на плечах програмістів, а адміністратори стежать, щоб стабільно працював шар хмарної інфраструктури - тобто, сам Kubernetes[4].

Тому компанії орендують кластери Kubernetes в хмарі, якщо хочуть повністю зняти з себе рутину по їх адмініструванння і займатися тільки розробкою.

Якщо написані тести (liveness / readiness probes), то розміщення софта в Kubernetes дозволить безпечно розгортатися хоч 50 раз за день без участі когось ще.

Також варто відзначити автоматичне вертикальне та горизонтальне масштабування (автоскейлінг): в залежності від актуальних показників навантаження, Kubernetes автоматично збільшує або зменшити розмір кластера, перерозподіляє його ресурси, додає і видаляє інстанси додатків.

Плюси Kubernetes

Kubernetes вважається системою або навіть платформою оркестрації контейнерів з найширшими функціональними можливостями. З точки зору використання K8s в Big Data системах найбільш значимі наступні переваги Kubernetes :

· автоматичне балансування навантаження за допомогою постійного моніторингу відомостей про продуктивність та використання ресурсів та відповідний розподіл працюючих додатків по всьому віртуальному кластеру;

· наявність API і інтерфейсу командної консолі для ручного управління розподілом контейнерів і навантаженням між вузлами системи;

· забезпечення інформаційної безпеки за рахунок механізму мережевих політик і рольової авторизації користувачів по ключу;

· наявність засобів логічної ізоляції додатків, команд і робочих навантажень в одному кластері з метою надання запущеним застосункам і командам мінімально необхідного набору привілегій для використання ресурсів;

· підтримка розподілених і мережевих файлових систем, таких як, наприклад, HDFS для Apache Hadoop, завдяки початковій орієнтації на кластери Big Data;

· наявність дешбордів - наочних вітрин з важливими показниками (dashboard).

1.3 Недоліки технології

При всіх перевагах K8s ця система управління контейнерними кластерами має такі недоліки, які значно ускладнюють її використання на практиці:

• велика кількість специфічних понять і взаємозалежних сутностей (под, сервіс, кубелет, контролер і т.д.);

• документація, яка недостатньо докладно описує систему;

• додавання додаткового рівня абстракції збільшує складність і крихкість системи;

• недолік і висока вартість досвідчених фахівців, які досконало володіють цією DevOps-технологією.

Висновки:

Таким чином, можна зробити висновок, що Kubernetes потужний інструмент для розгортаняя додатків написаних на основі мікросервісної архітектури. Чудово підходить для розгортання Big Data систем, може автоматично моніторити стан кластеру та відновлювати пошкоджені вузли.

РОЗДІЛ 2
ПЕРЕДУМОВИ РОЗВИТКУ KUBERNETES

Ранній залізний вік. Були часи, коли адміністратори давали легкозапам’ятовувавні імена для швидкого доступу до них. Час з ухвалення рішення про покупку сервера до його введення в продакшн займало кілька місяців.

Потім в корпораціях подібних до Google і Amazon зробили API, щоб піднімати потрібний сервер за секунди. Пізніше було зрозуміло, що так можна повністю автоматизувати цикл життя сервера, і це було величезним кроком вперед.

Також з'явилися системи типу Puppet, Saltstack і Chef, які дозволяли вказати, яка конфігурація додатків на сервері потрібна, і привести її до нього. Це сильно відрізнялося від традиційного підходу системних адміністраторів - зайти на сервер і запускати команди, поки він не опиниться налаштований.

Кордон між адміністратором і програмістом став розмиватися, так як щоб використовувати фреймворки з налаштування серверів, треба було трохи програмувати.

Контейнери. Потім на стику технологій Linux з'явився Docker - комплекс, що дозволяє зручно упаковувати додатки в контейнери і запускати їх на будь-якому linux-хості. Контейнер, зібраний на робочому комп'ютері програміста, точно так же запускався і на бойовому сервері, що знімало чимало проблем при запуску софта в бойовому середовищі.

Прийшло розуміння, що добре, коли розробник, який пише софт, відповідає за нього і в продакшині, а не просто віддає адміністраторам. Інакше виходить так, що програмісти хочуть викочувати швидко, а адміністраторам потрібен стабільний продакшн, вони чинять опір частим релізам. Якщо програміст буде відповідати за весь цикл життя софта «від і до», часті викочування перестануть заважати адміністраторам.

Народження DevOps. Для згладжування цього протиріччя виникла методологія DevOps - взаємне проникнення Dev- (розробка) і Ops- (адміністрування) команд в усі цикли розробки софта (Рис 2.1): від

проектування до роботи на продакшині і лагодження силами тих, хто писав код.

Рис. 2.1 Ланцюг безперервної інтеграції, розгортання і доставки (Continuous Integration, Continuous Delivery / Deployment, CI / CD)

Мікросервіси в контейнерах. Слідом прийшло розуміння, що якщо хочеться швидко викочувати в продакшн, треба розділяти великий софт на дрібні частини, щоб можна було їх розробляти та оновлювати незалежно. Так виникли мікросервіси.

Хмарні API і соціально-економічні передумови Kubernetes. З появою API публічних хмарних платформ і контейнерів з мікросервісами весь цикл розробки і постачання софта на продакшн змогли контролювати самі розробники. Це дозволило їм розробляти фічі з потрібною швидкістю без опору адмінів.

Так як контейнери треба якось автоматично зводити на бойових серверах, виникла необхідність в софті, який буде керувати всіма контейнерами через своє API - і так виник Kubernetes.

2.1 Контейнеризація, переваги та недоліки

Сьогодні контейнери масово використовують для розробки і розгортання додатків, а саме поняття стало таким же поширеним, як «віртуальна машина», «хмара» або «сервер».

Хоча контейнери і технології контейнеризації були відомі досить давно, справжню популярність вони отримали після появи в 2013 р платформи Docker, пов'язаного з нею активної спільноти користувачів і старту ініціативи OCI (Open Container Initiative). Крім ряду явних переваг, у контейнерів є недоліки і уразливості, якими можуть скористатися в своїх цілях кіберзлочинці[5].

Переваги контейнерів

Програмісти люблять контейнери в першу чергу за гнучке середовище, яке дозволяє створювати їх набагато швидше, ніж віртуальні машини (з якими у контейнерів, до речі, чимало спільного). Так само швидше відбувається їх запуск і зупинка. Крім цього, контейнери набагато менш вимогливі до ресурсів системи. Як правило, їх «вага» вимірюється не в гігабайтах, а в мегабайтах, оскільки при контейнерній віртуалізації немає потреби «упаковувати» всю ОС, щоб додаток нормально функціонував. Замість цього всі контейнери використовують ядро системи хоста, вивільняючи величезну кількість ресурсів і дозволяючи встановити набагато більше таких контейнерів на один хост.

Незважаючи на спільне використання ядра, всі програми ізольовані один від одної. Вони не можуть один на одного впливати, навіть якщо в якомусь з них відбудеться помилка. Це означає, що розробники вільно можуть змінювати параметри своїх додатків і не боятися, що їх програма «покладе» весь сервер. Ця ж ізольованість і використання ядра без потреби в гостьовій ОС, як в віртуальних машинах, дозволяють без проблем переносити контейнери з додатками з сервера на сервер. Крім того, застосування контейнерів спрощує управління версіями і оновлення додатків, так як оновлення софта в одному контейнері не вплине на інші частини системи.

До інших переваг технології контейнеризації варто віднести активну спільноту. Внаслідок популярності цього напрямку практично будь-яка проблема, з якою зіткнувся розробник, або вже вирішена кимось іншим і описана на форумах, або швидко знайде своє рішення. З тієї ж причини в мережі Інтернет вистачає публічних бібліотек-репозиторіїв з контейнерами, а над вдосконаленням екосистеми контейнерної віртуалізації і інструментами для розробки і управління працює величезна кількість фахівців з усього світу. Причому багато рішень стосуються автоматизації процесів, які дозволяють побудувати ланцюжки безперервної інтеграції, розгортання і доставки (Continuous Integration, Continuous Delivery / Deployment, CI / CD) практично без участі користувачів.

Недоліки контейнерів і загрози безпеці

Серед основних недоліків контейнерів слід назвати підвищену складність управління великим їх числом в системі (якого легко досягти через малу вагу самих контейнерів). Для вирішення цього завдання часто застосовуються додаткові інструменти, наприклад Kubernetes. Також в рамках однієї системи можна використовувати тільки контейнери, побудовані для конкретної архітектури, наприклад для ОС Windows, оскільки всі вони задіють ядро самої системи, а не гостьові ОС, як в віртуальних машинах. При цьому для ресурсоємних додатків буде ефективніше використовувати не контейнери або ті ж віртуальні машини, а традиційний сервер з прямим доступом до його ресурсів.

До явних загроз безпеки можна віднести публічні репозиторії, які допомагають розвивати технології контейнеризації. Команда дослідників Cisco Talos в травні нинішнього року виявила, що на порталі Docker Hub вже три роки контейнери Alpine Linux поширювалися з порожнім паролем суперкористувача, а подальші дослідження Kenna Security підтвердили, що ця ж проблема стосується приблизно 20% всіх контейнерів на тому ж ресурсі. Крім того, взлом публічного реєстру образів і додавання зловмисниками власних бібліотек дозволять заразити всі сервери, в кеші яких зберігається цей образ, оскільки за замовчуванням сервери таких реєстрів є довіреними ресурсами.

З уже згадуваного вище Docker Hub в червні 2018 року був видалено 17 таких контейнерів, але кількість їх завантажень за цей час перевалило за 1 млн. Більш того, в кінці квітня 2019 року репозиторій був зламаний і в руки хакерів потрапили дані понад 190 тис. користувачів , що показує неослабний інтерес зловмисників до цієї сфери. А в кінці травня 2019 пройшла чергова хвиля поширення контейнерів з кріптомайнером, що використовують API Docker. Восени 2018 року через каталогу PyPi також були видалені заражені контейнери, причому шкідливий код в них міг навіть підміняти адреси Bitcoin-гаманців жертв на потрібні хакерам.

2.2 Docker, мінімальний незалежний об'єкт Kubernetes

Що таке Kubernetes і порівняння його з Docker

Kubernetes - це платформа для оркестрації контейнерів, яка дозволяє побудувати розподілену відмовостійку систему. Kubernetes автоматично управляє життєвим циклом контейнерів, підтримує стабільний стан системи і розподіляє навантаження по різним вузлам.

Kubernetes і Docker не можна порівнювати напряму. Docker - інструмент для створення і запуску контейнерів, а Kubernetes - оркестратор, інструмент для управління контейнерами. Kubernetes дозволяє побудувати кластер (Cluster) - розподілену відмовостійку систему, в той час як Docker працює на окремому вузлі[6].

Ще є Docker Swarm - це вбудований в докер інструмент оркестровки контейнерів. Його як саме і можна повноцінно порівнювати з Kubernetes.

Основна перевага Kubernetes перед простими Docker-контейнерами можна описати одним словом - автоматизація. Але під цією автоматизацією розуміється багато різних можливостей.

Потрібно відзначити, що це все можна автоматизувати і без Kubernetes, але для цього доведеться використовувати багато окремих інструментів, налаштовувати їх і писати скрипти. Наприклад, для автоматизації викочування оновлень додатків можна використовувати Ansible - систему управління конфігураціями. Така система може оновити всі інстанси додатку і переконатися, що вони працюють. Але для цього система повинна знати, які хости зараз працюють, як до них звернутися, на яких саме хостах працюють інстанси потрібного додатка. Для цього доведеться писати скрипти, які будуть за цим стежити.

Займатися підтримкою цих скриптів та інструментів доведеться самостійно. Іноді можуть виникати проблеми, наприклад, якщо у одного з інструментів вийде нова версія, яка несумісна з іншими інструментами. Або для нової версії інструменту доведеться переписувати скрипти, тому що зміниться логіка роботи цього інструменту.

У Kubernetes всі ці автоматизації вже включені, вони розробляються і підтримуються великим співтовариством розробників. Коли виходить нова версія Kubernetes, розробники тестують і перевіряють все самі, а значить, всі внутрішні інтеграції та автоматизації точно будуть працювати.

Коли в інфраструктурі стає багато вузлів (серверів, на яких працюють контейнери), ними потрібно централізовано керувати. Зручніше за все мати центральну точку управління, яка може прийняти команди і виконати дії незалежно від того, на якому вузлі знаходяться контейнери.

Docker працює в рамках окремих вузлів. Якщо у вас кілька вузлів, то на кожному з них запущений докер-демон, який нічого не знає про існування інших вузлів. Кожен демон знає лише про те, що відбувається на його вузлі. І тому звичайними докер-контейнерами складно управляти централізовано.

У Kubernetes є центральна точка управління - API-сервер. У кластері може бути багато вузлів, контейнерів і налаштувань, але всіма ними можна управляти через єдиний сервер. У кластер можна відправити команду (наприклад, щоб оновити наш додаток), і Kubernetes сам знайде, на яких вузлах він знаходиться, і оновить його.

Управління конфігураціями і конфіденційними даними

Додаткам часто потрібні файли налаштувань, змінні оточення або конфіденційні дані (логіни, паролі, токени і т. П.). Зазвичай такі файли зберігаються окремо від додатків. Якщо додаток працює на декількох вузлах, потрібно поширити ці файли налаштувань на всі вузли. Причому якщо настройки зміняться, їх знову потрібно оновити на всіх вузлах.

Docker. Так як докер-демон працює окремо на кожному вузлі, всі маніпуляції з доставки і синхронізації цих файлів між вузлами потрібно проводити вручну. До того ж у Docker-контейнерів немає можливості зберігання конфіденційних даних в зашифрованому вигляді. Докер може використовувати тільки змінні оточення.

У Kubernetes є засіб для центрального управління конфігураціями - ConfigsMaps для налаштувань і Secrets для конфіденційних даних. Якщо

розмістити в них свої настройки, то додатки зможуть отримати до них доступ з будь-якого вузла.

Автомасштабування вузлів

Технологія контейнеризації дозволяє легко масштабувати додаток. Якщо навантаження на додаток зростає, можна запустити додаткові екземпляри, щоб програма не зависала. Але відслідковувати зміну навантаження вручну неправильно, набагато зручніше це автоматизувати. Тут під масштабуванням ми маємо на увазі як збільшення кількості контейнерів, так і підключення додаткових вузлів, якщо ресурсів існуючих вузлів стає недостатньо.

Docker контейнери не можуть автомасштабуватися. Так як кожен докер-демон працює тільки в рамках свого вузла, він не знає, що відбувається на інших. Докер-демон не може стежити за навантаженням, створювати додаткові контейнери або підключати нові вузли.

Kubernetes вміє автомасштабуватися. «З коробки» він вміє масштабувати кількість контейнерів, і додатково можна налаштувати підключення нових вузлів. Коли навантаження на додаток зросте, Kubernetes додасть нові інстанси додатку, а при необхідності і підключить нові вузли. Коли навантаження спаде, Kubernetes завершить зайві контейнери і звільнить вузли.

Оновлення додатків і повернення до попередніх версій

Додатки часто допрацьовуються і змінюються. При оновленні контейнерного додатку потрібно оновити інстанси на всіх вузлах. При цьому на час поновлення потрібно зробити так, щоб оновлюваний контейнер не отримував запити, тобто його потрібно вивести з балансування навантаження. Також часто потрібно виконувати оновлення без простоїв за методом Rolling-Update, тобто щоб всі інстанси оновлювалися послідовно і додаток був доступний в будь-який момент часу.

Інше завдання - відкат оновлень. Іноді для швидкого усунення помилки потрібно відкотити оновлення, щоб потім спокійно розбиратися і усувати помилки. По суті, ця процедура аналогічна оновленню, просто в зворотному напрямку.

В Docker потрібно оновлювати окремо кожен контейнер. У контейнерів немає засобу, щоб однією або декількома командами оновити всі інстанси. Потрібно знайти всі вузли, на яких працює додаток, і оновлювати їх по черзі.

Kubernetes. Щоб викотити нову версію програми в Kubernetes, досить в Deployment-файлі вказати нову версію в імені образу. При цьому якщо потрібно оновити програмне забезпечення по типу Rolling-Update, це теж досить вказати в Deployment-файлі. Якщо після поновлення потрібно відкотитися, досить виконати команду rollout, і Kubernetes сам відкотить всі контейнери.

Система зберігання данних.

Багатьом додаткам потрібен доступ до сховища постійних даних. Так як контейнери за своєю природою ізольовані, зазвичай налаштувати доступ до сховища не сама тривіальна задача.

У Docker є томи (Volumes), які потрібні для спрощення роботи з постійними даними. Але в будь-якому випадку томи потрібно додатково налаштовувати, причому робити це доведеться для кожного контейнера окремо. Наприклад, у нас є мережеве сховище Ceph. Щоб підключити його до контейнера, спочатку потрібно створити те, підключити його до всіх вузлів з контейнерами і потім змонтувати цей том в кожен окремий контейнер.

Kubernetes набагато спрощує роботу з дисками. Повністю «з коробки» підключити диски не вийде, тут все одно буде потрібно деяка настройка і додаткові інструменти. Але зробити це набагато простіше за рахунок об'єктів PersistentVolume і PersistentVolumeClaim. Спочатку потрібно створити те і налаштувати його один раз, а потім кожен контейнер може самостійно запитувати ресурси, без необхідності додаткових маніпуляцій. Детальніше про цей механізм читайте в нашій статті про основи Kubernetes.

Розподіл навантаження і маршрутизація трафіку з зовнішньої мережі.

Якщо у програми є кілька інстансів, то, щоб звернутися до додатка, потрібно знати його адреси. Причому зовнішні сервіси, які звертаються до додатка, навіть не повинні знати, скільки інстансів у додатку, де вони запущені

і так далі. Вони повинні взаємодіяти з програмою як з окремимою сутністю, не замислюючись про його інстанси.

Інше завдання - маршрутизація трафіку з зовнішньої мережі. Якщо додаток має бути доступним з інтернету або будь-якої іншої зовнішньої мережі, необхідно налаштувати маршрутизацію. Щоб зовнішній трафік правильно розподілявся по потрібним сервісам і додаткам.

Docker. У простих докер-контейнерах немає можливості розподіляти навантаження між інстансами додатку. Наприклад, додаток хоче звернутися до сервісу. Щоб це зробити, він повинен знати IP-адресу цього сервісу. А якщо у сервісу кілька інстансів, то програма має знати всі адреси і сама вирішувати, в які саме інстанси звернутися. Якщо один з контейнерів сервісу увімкнеться знову і отримає нову адресу, додаток повинен якось про це дізнатися. Це незручний і неправильний підхід до розробки.

Проблема посилюється, якщо до сервісу потрібно звертатися із зовнішньої мережі. Для цього доведеться підтримувати в актуальному стані і ще й таблицю маршрутизації для зовнішнього трафіку.

Kubernetes бере розподіл трафіку на себе. Якщо розглядати приклад вище, то Kubernetes дозволяє над усіма інстансами сервісу створити єдиний об'єкт типу Service. Цей об'єкт сам стежить за всіма інстансами, і, якщо контейнер перезапускається, він сам оновить інформацію про його нову адресу. І щоб додаток міг звернутися до сервісу, досить знати одне доменне ім'я, яке буде завжди постійним.

Також Kubernetes вміє перенаправляти зовнішній трафік в контейнери. Для цього потрібно встановити і налаштувати Ingress Controller, але це набагато простіше, ніж самостійно налаштовувати і підтримувати маршрутизацію. Kubernetes дозволяє зробити так, щоб до додатків можна було звертатися по доменних іменах або URL-шляхах. Ingress Controller сам буде перенаправляти трафік на потрібний інстанси контейнера.

Перш ніж почати працювати з Kubernetes, його потрібно вивчити. Тому що це більше, ніж просто Docker і контейнери. Необхідно розгорнути кластер, налаштувати його і підтримувати. Ці завдання можна спростити, якщо

використовувати хмарні рішення, наприклад Kubenetes as a Service від Mail.ru Cloud Solutions.

2.3 Docker swarm як передумова винекнення Kubernetes

При побудові великої продакшн-системи в неї обов'язково закладають вимоги по відмовостійкості, продуктивності і масштабованості. Тобто система повинна бути захищена від збоїв, не гальмувати і мати можливість для збільшення потужності.

Зазвичай для цього створюють кластер - окремо стоячі хости (сервери) об'єднують під загальним управлінням, з боку це виглядає як єдина система. При цьому вона набагато стійкіша до збоїв і продуктивніша:

• Відмовостійкість досягається завдяки надмірності хостів (в рамках кластера вони називаються нодами). Система працює відразу на кількох нодах, якщо одна з них вийде з ладу, інші спокійно продовжать роботу.

• Балансування навантаження дозволяє рівномірно навантажити кожну ноду. Кластер стежить за навантаженням і сам розподіляє всередині себе завдання: одну програму запустить на одній ноді, іншу програму - на інший.

• Масштабованість допомагає підлаштовувати продуктивність кластера під навантаження. Якщо у додатків і сервісів не вистачає ресурсів, можна швидко підключити додаткові Ноди.

При роботі з контейнерами ці завдання вирішують системи оркестровки. Оркестровка - це управління і координація взаємодії між контейнерами. Контейнери запускаються на хостах, а хости об'єднують в кластер.

У Docker є стандартний інструмент оркестровки - Docker Swarm Mode, або просто Docker Swarm. Він поставляється «з коробки», досить простий в налаштуванні і дозволяє створити простий кластер буквально за хвилину.
Кластер Swarm (Docker Swarm Cluster) складається з нод, які ділять на два типи(Рис. 2.2):

• Керуюча нода (Manager). Це нода, яка приймає запити і розподіляє завдання між усіма нодами в кластері. Менеджерів може (і повинно) бути кілька, але серед них обов'язково є одна найголовніша нода - лідер, який керує всім кластером.

• Робоча нода (Worker). Підпорядкована нода, яка не приймає рішень, а просто виконує відправляються їй завдання.

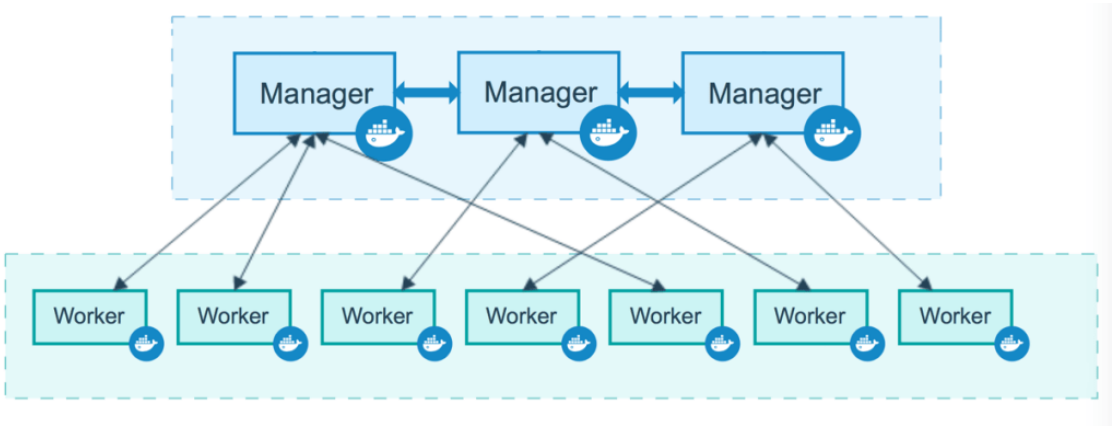

Рис. 2.2 Схематичний пристрій кластера Docker Swarm: три керуючих Ноди і сім робітників.

В Docker Swarm замість прямого використання контейнерів використовуються сервіси (Docker Swarm Service). Вони схожі на контейнери, але все ж це трохи інше поняття. Сервіс - це щось на зразок рівня абстракції над контейнерами. У Swarm ми не запускаємо контейнери явно - цим займаються сервіси. Для досягнення відмовостійкості ми лише вказуємо сервісу кількість реплік - нод, на яких він повинен запустити контейнери. А Swarm вже сам простежить за тим, щоб ця вимога виконувалося: знайде відповідні хости, запустить контейнери і буде стежити за ними. Якщо один з вузлів відвалиться - створить нову репліку на іншому хості.

Отже, Docker Swarm - вбудоване в Докер рішення для оркестровки контейнерів, яке виконує базові функції і яке легко налаштувати.

Що таке Kubernetes і в чому його переваги перед Swarm.

Крім стандартного Docker Swarm є й інші інструменти оркестровки, наприклад Kubernetes. Це складна система, яка дозволяє побудувати відмостійку і легко масштабовану платформу для управління контейнерами. Він вміє працювати не тільки з контейнерами Docker, але і з іншими контейнерами: rkt, CRI-O.

У Kubernetes досить багато можливостей, які дозволяють будувати масштабні розподілені системи. Через це поріг входження в технологію

набагато вище, ніж в Swarm. Потрібно мати відповідний рівень знань, а на первинну установку і настройку може піти кілька днів.

Якщо дивитися глобально, то пристрій Kubernetes схоже на Swarm. Кластер складається з двох типів нод: головної (Master) і робочих (Worker):

• Master-нода стежить за станом свого кластера, розподіляє навантаження і розгортає контейнери на нодах.

• Робочі Ноди обробляють запити, що надходять.

Але якщо дивитися глибше, то пристрій Kubernetes набагато складніше. У ньому окремі модулі, наприклад: proxy-балансувальник, etcd для зберігання стану кластера та інші компоненти. Не будемо детально все це описувати. Досить зрозуміти, що Kubernetes влаштований набагато складніше, ніж Docker Swarm.

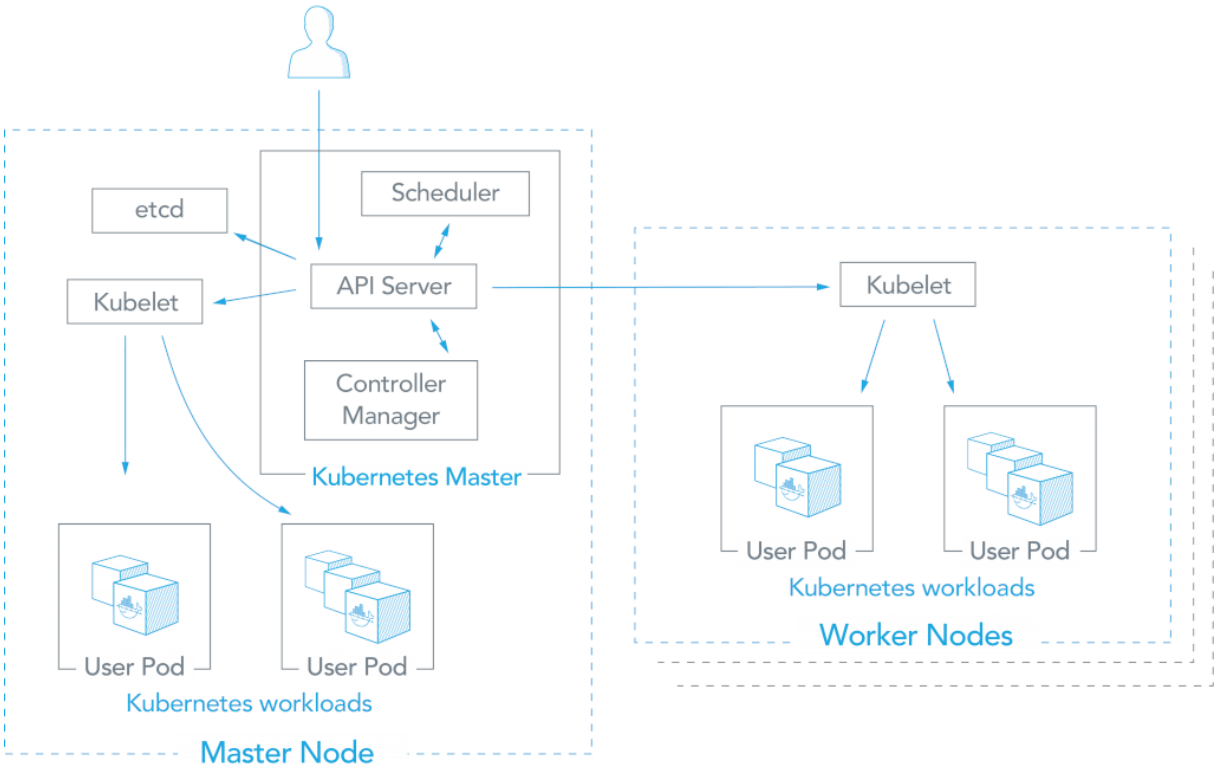

Рис. 2.3 Схема роботи шару управління Kubernetes

Глобально Kubernetes схожий на Swarm. Але всередині все набагато складніше(див. Рис 2.3).

Так навіщо потрібен Kubernetes зі своїми труднощами, коли вже є Docker Swarm?

Справа в тому, що Kubernetes дозволяє вирішувати завдання, які не під силу Docker Swarm. Для прикладу візьмемо автомасштабування: це коли система сама підлаштовує свою потужність під навантаження. Для цього в кластер автоматично додаються / видаляються Ноди, або в існуючих нодах для «важких» завдань буде виділятися більше / менше ресурсів. Наприклад, в кінці місяця бухгалтерія веде особливо активну роботу: рахує зарплату, звіряє внески на оплату і готує звіти. У такі періоди навантаження на систему зростає. Якщо ресурсів не буде вистачати, то у бухгалтерів почнуть гальмувати програми і вони не зможуть нормально працювати.

Але якщо система вміє масштабуватися, вона відреагує на зростання навантаження і збільшить ресурси для цих завдань. А коли навантаження спаде, знову звільнить ці потужності. Якщо кластер розміщений в хмарі, автомасштабування сильно економить гроші. У моменти простою невикористовувані ресурси звільняються, і за них не потрібно переплачувати.

Так ось, в Kubernetes можна налаштувати автомасштабування. Так, доведеться написати конфігураційний файл і виконати інші настройки, але в результаті ви отримаєте робочу і стабільну систему. А якщо розгорнути кластер в хмарі, яке підтримує автомасштабування, то на настройку піде всього кілька хвилин.

Docker Swarm не вміє робити цього «з коробки». Можна побудувати автомасштабовану систему з використанням Swarm. Але для цього доведеться вручну писати скрипти або програми, які будуть стежити за навантаженням, приймати рішення і посилати команди в Docker Swarm. Або можна використовувати сторонні розробки, на кшталт Orbiter, але його можливості теж обмежені, і в будь-якому випадку це ще одна додаткова надбудова над Swarm.

Тепер уявіть, що крім автомасштабування у вас є інші завдання, для яких доводиться городити купу інструментів над Swarm. Все це потрібно підтримувати, розуміти як воно працює і ретельно тестувати при оновленнях. У Kubernetes такі складності заховані усередині, і вони стабільно працюють.

Docker Swarm vs Kubernetes: що вибрати?

Наведемо коротке порівняння двох технологій: які у них плюси і мінуси, в яких випадках їх краще використовувати.

Docker Swarm - це вбудований інструмент кластеризації, який працює «з коробки». Він простий у використанні і налаштуванні, але не дуже гнучкий.

Плюси простота і швидкість настройки Docker Swarm: робочий кластер Swarm можна підняти за хвилину [7].

Інструменти, сумісні з Docker: більшість команд Docker CLI буде працювати в Swarm.

Мінуси вузька функціональність: можливості обмежені Docker API. Це означає, що Swarm здатний зробити лише те, що дозволяють можливості Docker.

Він підійде для невеликих компаній або проектів, де не потрібна гнучкість Kubernetes.

Kubernetes - це універсальний засіб для створення розподілених систем. Це складна система з великою кількістю можливостей, яку не так просто налаштувати самостійно. Але якщо розмістити Kubernetes як сервіс в хмарі, це значно спрощує його настройку і використання.

Плюси потужний інструмент. У Kubernetes багато можливостей, які дозволяють будувати дійсно складні розподілені системи.

Kubernetes вміє працювати з різними системами контейнеризації, хоча найчастіше використовується Докер і його контейнери.

Мінуси складність налаштування: часто потрібні глибші знання і більше часу на установку і настройку.

Для управління використовується окремий набір команд і інструментів, несумісних з Docker CLI.

Для кого підійде: для великих компаній або проектів, де потрібна гнучка настройка всієї інфраструктури.

Висновки:

Таким чином, можна зробити висновок що: Docker Swarm - це стандартна система оркестровки Docker, яка вирішує базові завдання. Він простий в установці та налаштування, але не дуже гнучкий. Kubernetes навпаки - потужна система оркестровки, яка дозволяє будувати масштабні розподілені системи. Але він не такий простий в установці та налаштування, не кожен інженер відразу зможе розібратися в ньому. Хмарні провайдери дозволяють усунути головний недолік Kubernetes - складність. З їх допомогою можна розгорнути готовий кластер за 10 хвилин.

РОЗДІЛ 3
ОСОБЛИВОСТІ ПЛАТФОРМИ KUBERNETES

3.1 Основні поняття платформи

При розгортанні Kubernetes ви маєте справу з кластером. Кластер Kubernetes складається з набору машин, так звані вузли, які запускають контейнеризовані додатки. Кластер має як мінімум один робочий вузол. У робочих вузлах розміщені поди, що є компонентами програми. Площина управління керує робочими вузлами і подами в кластері. У промислових середовищах площину управління зазвичай запускається на декількох комп'ютерах, а кластер, як правило, розгортається на кількох вузлах, гарантуючи відмовостійкість і високу надійність. На цій сторінці у загальних рисах описується різні компоненти, необхідні для роботи кластера Kubernetes. Нижче показана діаграма кластера Kubernetes з усіма пов'язаними компонентами[8].

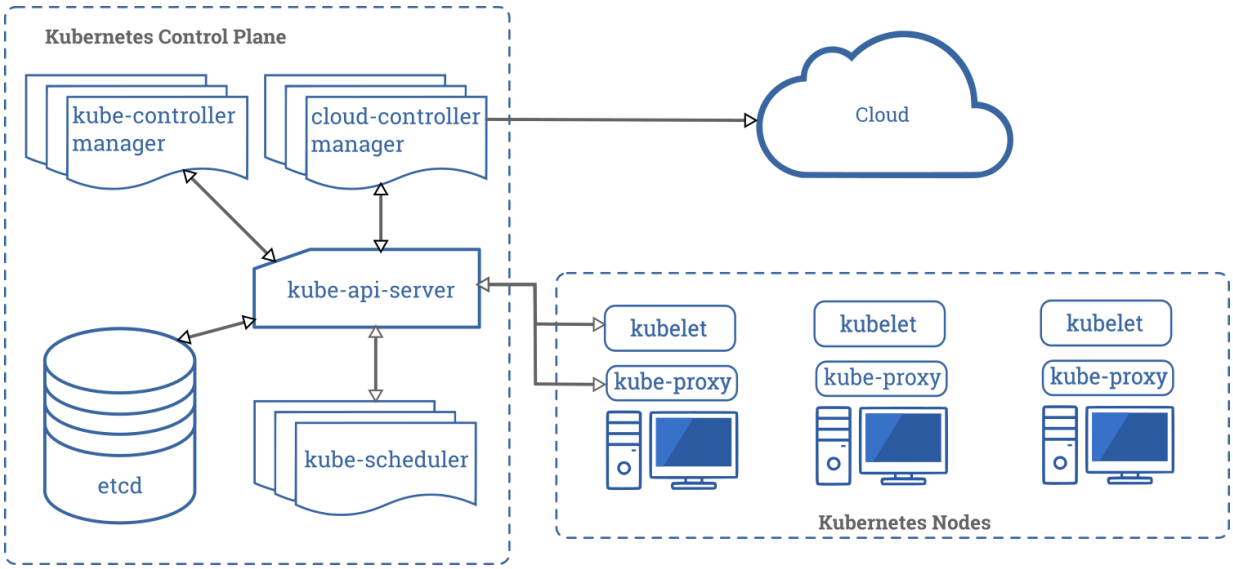

Рис. 3.1 Площина управління компонентами

Компоненти панелі управління відповідають за основні операції кластера (наприклад, планування), а також обробляють події кластера (наприклад, запускають новий під, коли поле replicas розгортання не відповідає необхідному кількості реплік).

Компоненти панелі управління можуть бути запущені на будь-якій машині в кластері. Однак для простоти сценарію налаштування зазвичай

запускають всі компоненти панелі управління на одному комп'ютері і в той же час не дозволяють запускати призначені для користувача контейнери на цьому комп'ютері.

kube-apiserver

Сервер API - компонент Kubernetes панелі управління, який представляє API Kubernetes. API-сервер - це клієнтська частина панелі управління Kubernetes[8].

Основний реалізацією API-сервера Kubernetes є kube-apiserver. kube-apiserver призначений для горизонтального масштабування, тобто розгортання на кілька примірників. Ви можете запустити кілька екземплярів kube-apiserver і збалансувати трафік між цими примірниками.

etcd

Розподілене і високонадійні сховище даних в форматі "ключ-значення", яке використовується як основне сховище всіх даних кластера в Kubernetes[8]. Перед ти як використовувати etcd в вашому кластері Kubernetes, переконайтеся, що у вас налаштоване резервне копіювання даних.

kube-scheduler

Компонент площини управління, який відстежує створені поди без прив'язаного вузла і вибирає вузол, на якому вони повинні працювати. При плануванні розгортання подів на вузлах враховуються безліч чинників, включаючи вимоги до ресурсів, обмеження, пов'язані з апаратними / програмними політиками, приналежності (affinity) і неналежність (anti-affinity) вузлів / подів, місцезнаходження даних, граничних термінів[8].

kube-controller-manager

Компонент Control Plane запускає процеси контролера.

Цілком логічно, що кожен контролер в свою чергу є окремим процес, і для спрощення всі такі процеси скомпільовані в один двійковий файл і виконуються в одному процесі.

Ці контролери включають[8]:

- Контролер вузла (Node Controller): повідомляє і реагує на збої вузла.

• Контролер реплікації (Replication Controller): підтримує правильну кількість подів для кожного об'єкта контролера реплікації в системі.

• Контролер кінцевих точок (Endpoints Controller): заповнює об'єкт кінцевих точок (Endpoints), тобто пов'язує сервіси (Services) і поди (Pods).

• Контролери облікових записів і токенів (Account & Token Controllers): створюють стандартні облікові записи і маркери доступу API для нових просторів імен.

cloud-controller-manager

Cloud-controller-manager запускає контролери, які взаємодіють з основними хмарними провайдерами. Двійковий файл cloud-controller-manager - це альфа-функціональність, що з'явилися в Kubernetes 1.6.

За допомогою cloud-controller-manager код як хмарних провайдерів, так і самого Kubernetes може розроблятися незалежно один від одного. У попередніх версіях код ядра Kubernetes залежав від коду, призначеного для функціональності хмарних провайдерів. У майбутніх випусках код, специфічний для хмарних провайдерів, повинен підтримуватися самим хмарним провайдером і компонуватись з cloud-controller-manager під час запуску Kubernetes[8].

kubelet

Агент, який працює на кожному вузлі в кластері. Він стежить за тим, щоб контейнери були запущені в поді. Утиліта kubelet приймає набір PodSpecs, і гарантує працездатність і справність визначених у них контейнерів. Агент kubelet не відповідає за контейнери, які не створені Kubernetes[8].

kube-proxy

kube-proxy - мережевий проксі, який працює на кожному вузлі в кластері, і який реалізує частину концепції сервіс. Він конфігурує правила мережі на вузлах. За допомогою них вирішуються мережеві підключення до ваших под з середини і зовні кластера.

kube-proxy використовує рівень фільтрації пакетів операційної системи, якщо він доступний. В іншому випадку, kube-proxy сам обробляє передачу мережевого трафіку.

Середовище виконання контейнера

Середовище виконання контейнера - це програма, призначена для виконання контейнерів. Kubernetes підтримує кілька середовищ для запуску контейнерів: Docker, containerd, CRI-O, і будь-яка реалізація Kubernetes CRI (Container Runtime Interface) [8].

3.2 Об’єкти платформи Kubernetes

Kubernetes використовує об’єкти для представлення стану кластера. Зокрема, вони описують таку інформацію:

- Які контейнерні програми запущені (і на яких вузлах).
- Доступні ресурси для цих програм.
- Стратегії керування додатками, які стосуються, наприклад, перезапуску, оновлення або відмовостійкості.

Після створення об’єкта Kubernetes відстежуватиме існування об’єкта. Створюючи об’єкт, ви вказуєте системі Kubernetes, яким має бути робоче навантаження кластера; це бажаний стан кластера.

Щоб працювати з об’єктами Kubernetes – створювати, змінювати чи видаляти – потрібно використовувати Kubernetes API. Наприклад, під час використання інструмента CLI kubectl він отримує доступ до Kubernetes API. Ви також можете використовувати Kubernetes API у своїх програмах, використовуючи одну з клієнтських бібліотек.

Специфікація і статус об'єкта

Майже кожен об’єкт Kubernetes має два вкладених поля об’єкта, які контролюють конфігурацію об’єкта: специфікацію та статус. Коли об’єкт створюється, у полі специфікації вказується необхідний стан (опис характеристик, які повинен мати об’єкт).

Поле статусу описує поточний стан об’єкта, який створюється та оновлюється самим Kubernetes та його компонентами. Площина керування Kubernetes постійно керує фактичним станом кожного об’єкта, щоб відповідати бажаному стану, визначеному користувачем.

Наприклад: deployment — це об’єкт Kubernetes, який представляє запущену програму в кластері. Коли ви створюєте об’єкт Deployment, ви

можете вказати в його полі специфікації, що вам потрібно три репліки програми. Система Kubernetes отримає специфікацію об’єкта Deployment і запустить три екземпляри програми, таким чином оновивши статус (стан) об’єкта відповідно до вказаної специфікації. У разі збою одного з екземплярів (це тягне за собою зміну стану) Kubernetes виявить невідповідність між специфікацією та статусом і виправить її, тобто активує новий екземпляр замість того, що вийшов з ладу.

Опис об'єкта Kubernetes

При створенні об’єкта в Kubernetes потрібно передати специфікацію об’єкта, яка містить необхідний стан, а також основну інформацію про об’єкт (наприклад, його назву). Коли ви використовуєте Kubernetes API для створення об’єкта (безпосередньо або через kubectl), відповідний запит API має містити всю вказану інформацію у форматі JSON у тілі запиту. У більшості випадків ви передасте ці дані до kubectl, як написано у файлі .yaml. Потім інструмент kubectl перетворює їх у JSON під час виконання запиту до API.

Обов'язкові поля

У файлі .yaml створюваного об’єкта Kubernetes необхідно вказати значення для таких полів:

- apiVersion - версія Kubernetes API, яка використовується для створення об'єкта
- kind - тип створюваного об'єкта
- metadata - дані, що дозволяють ідентифікувати об'єкт (ім'я, UID і необов'язкове поле простору імен)
- spec - необхідний стан об'єкта

Конкретний формат поля об’єкта специфікації залежить від типу об’єкта Kubernetes і містить вкладені поля, специфічні для використовуваного об’єкта. У довідці Kubernetes API ви можете знайти формат специфікації для будь-якого об’єкта Kubernetes. Наприклад, специфікація Pod знаходиться в ядрі PodSpec v1, а специфікація для Deployment знаходиться в програмах DeploymentSpec v1.

3.3 Створення файлів конфігурації Kubernetes

Основи YAML

Важко уникнути YAML, якщо ви робите щось, що стосується багатьох областей програмного забезпечення, зокрема Kubernetes, SDN і OpenStack. YAML, що розшифровується як «Yet Another Markup Language» - це текстовий формат, який можна читати людиною для вказівки інформації типу конфігурації. Наприклад, розберемо визначення YAML для створення Pod.

Визначаючи маніфест Kubernetes, YAML дає вам ряд переваг, зокрема:

- Зручність: вам більше не доведеться додавати всі свої параметри до командного рядка
- Обслуговування: файли YAML можна додавати в систему управління джерелами, наприклад, репозиторій Github, щоб ви могли відстежувати зміни
- Гнучкість: ви зможете створювати набагато складніші структури за допомогою YAML, ніж у командному рядку

YAML є наднабором JSON, що означає, що будь-який дійсний файл JSON також є дійсним файлом YAML[9]. Навіть якщо ви тільки намагаєтеся знайти приклади в Інтернеті, вони, швидше за все, в YAML (не JSON), тому ми можемо також звикнути до цього. Проте можуть бути ситуації, коли формат JSON є більш зручним, тому добре знати, що він доступний для вас.

YAML відносно легко вивчити. Є лише два типи структур, про які вам потрібно знати в YAML:

- списки
- карти

Створення файлу Kubernetes Pod за допомогою YAML

Розібравшись з основами, давайте подивимося, як це використовувати. Можна почати створювати файл конфігурацій для Pod, використовуючи YAML.

```
---
apiVersion: v1
kind: Pod
metadata:
  name: rss-site
  labels:
    app: web
spec:
  containers:
    - name: front-end
      image: nginx
      ports:
        - containerPort: 80
    - name: rss-reader
      image: nickchase/rss-php-nginx:v1
      ports:
        - containerPort: 88
```

Рис. 3.2 Приклад файлу конфігурації pod

Розбираючи його по частинах, починаємо з версії API; тут просто v1(Рис 3.2).

Вказуємо, що ми хочемо створити Pod для зберігання вашої програми або хмарного сервісу; замість цього ми можемо вказати Deployments, Job, Service тощо, залежно від того, чого ми намагаємося досягти.

Далі вказуємо метадані. Тут ми вказуємо назву Pod, а також мітку, яку ми будемо використовувати, щоб ідентифікувати pod для Kubernetes.

Нарешті, налаштуємо фактичні об’єкти, які складають модуль. Властивість spec включає в себе будь-які контейнери, вимоги до пам’яті, обсяги сховища, мережу чи інші деталі, про які Kubernetes має знати, а також такі властивості, як перезапуск контейнера. Повний список властивостей Kubernetes Pod можна знайти в специфікації Kubernetes API, але давайте детальніше розглянемо типове визначення контейнера:

```
...
 spec:
   containers:
     - name: front-end
       image: nginx
       ports:
         - containerPort: 80
     - name: rss-reader
 ...
```

Рис. 3.3 Специфікації pod

У цьому випадку ми маємо просте, досить мінімальне визначення: ім’я (front-end), образ, на якому воно засноване (nginx), і один порт, на якому контейнер буде прослуховувати внутрішньо (80). З них обов’зкове лише ім’я, але загалом, якщо ви хочете, щоб воно робило щось корисне, вам знадобиться більше інформації.

Ви також можете вказати більш складні властивості, такі як команда, яка запускатиметься під час запуску контейнера, аргументи, які він повинен використовувати, робочий каталог або те, чи потрібно витягувати нову копію зображення щоразу, коли його екземпляр створюється. Ви також можете вказати ще більш глибоку інформацію, наприклад, розташування журналу виходу контейнера. Ось властивості, які ви можете встановити для контейнера, які ви можете знайти в довіднику Kubernetes YAML[10]:

- name
- image
- command
- args
- workingDir
- ports
- env
- resources
- volumeMounts
- livenessProbe

- readinessProbe
- lifecycle
- terminationMessagePath
- imagePullPolicy
- securityContext
- stdin
- stdinOnce
- tty

Для створення Pod необхідно скористатися наступною командою:

```
kubectl create -f pod.yaml

pod "rss-site" created
```

Рис. 3.4 Команда для створення pod та результат її виконання

Висновки:

Таким чином, можна зробити висновок, що кластер Kubernetes має як мінімум один робочий вузол. У робочих вузлах розміщені поди, що є компонентами програми. Площина управління керує робочими вузлами і подами в кластері. Для розвернення додатку в Kubernetes кластері, написаного на основі мікросервісної архітектури, необхідно вміти створювати файли конфігурації, а також створювати Docker контейнери для подальшого розміщення їх на pods та deployments.

РОЗДІЛ4
ГІПОТЕЗА ТА ЇЇ ЕКСПЕРЕМЕНТАЛЬНА ПЕРЕВІРКА

4.1 Опис експерементального додатку та гіпотеза дослідження

Додаток буде виконувати лише одну функцію. Воно приймає, як вхідні дані, одну пропозицію, після чого, використовуючи засоби аналізу текстів, здійснює аналіз тональності (sentiment analysis) цієї пропозиції, отримуючи оцінку емоційного ставлення автора пропозиції до якогось об'єкта.

Ось як виглядає головне вікно цієї програми (Рис 4.1):

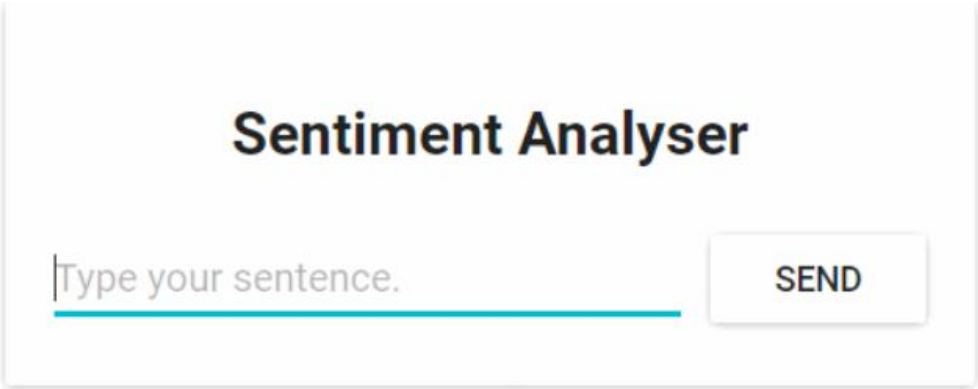

Рис. 4.1 Інтерфейс додатку для взаємодії з користувачем

З технічної точки зору програма складається з трьох мікросервісів, кожен з яких вирішує певний набір завдань:

SA-Frontend – веб-сервер Nginx, який обслуговує статичні файли React.

SA-WebApp - веб-додаток, написаний на Java, який обробляє запити від фронтенду.

SA-Logic - Python-додаток, який виконує аналіз тональності тексту.

Мікросервіси існують не в ізоляції. Вони реалізують ідею «поділу обов'язків», але їм, у своїй, необхідно взаємодіяти друг з одним.

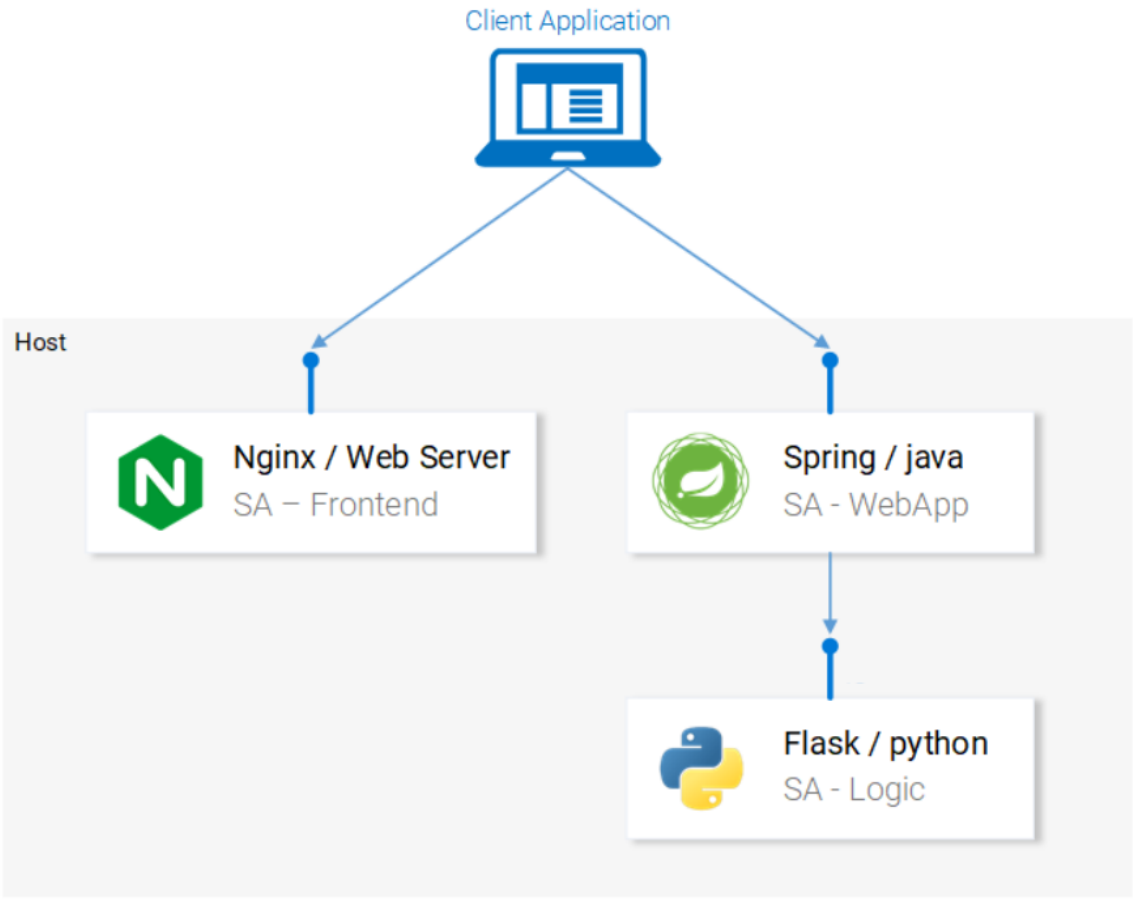

Рис. 4.2 Схема роботи експерементального додатку

На наведеній вище схемі можна побачити етапи роботи системи, що ілюструють потоки даних у додатку. Розберемо їх:

1. Браузер запитує у сервера файл index.html (який, своєю чергою, робить завантаження пакета React-программы).
2. Користувач взаємодіє з додатком, це викликає звернення до веб-застосунку, заснованому на Spring.
3. Веб-програма перенаправляє запит на виконання аналізу тексту Python-додатку.
4. Python-додаток проводить аналіз тональності тексту та повертає результат у вигляді відповіді на запит.
5. Spring-програма відправляє відповідь React-програмі (а вона, у свою чергу, показує результат аналізу тексту користувачеві).

Гіпотеза дослідження полягає у тому, що використання кластеру Kubernetes може бути ефективнішим для розгортання додатків побудованих на основі мікросервісної архітектури, ніж з використанням звичайних Docker контейнерів у рамках ВМ, з точки зору відмовостійкості та використання системних ресурсів. Не дивлячись на те, що докер контейнери використовують хостову ОС, додаток розгорнутий на ВМ буде менш ефективним, оскільки хостова ОС це повноцінна операційна система, котра може мати в собі запущені програми котрі не виконують ніякої корисної ролі для додатку розміщенему в ній. Також є можливість того, якщо не має ізоляції докер контейнерів в ресурсах (тоб-то не вказані системні ресурси для кожного з контейнерів) вони можуть заважати роботі один одного, хоча й виконуватимуть різні функції.

Згідно гіпотизі дослідження, Kubernetes cluster побудований лише з використанням Pods повинен мати майже такіж показники як і на докер контейнерах. Але використовуючи, конфігурацію з deployments та loadbalancers можна значно підвищити показники системи як з точки зору використання системних ресурсів так і з відмовостійкості. Використання системних ресурсів можна буде оптимізувати шляхом кількості pods в deployments (тоб-то якщо на сервіс немає великого навантаження працюватиме 1 pod, якщо він не буде справлятися з навантаженням буде підніматися більша кількість pod в залежності від кількості системних ресурсів). Load balancer буде рівномірно розподіляти запити між pods, що повинно зменшити вірогідність крашу pods.

Таким чином, використання K8s має бути ефективнішим з точки зору доступності додатку та ефективності використання системних ресурсів (оперативної пам’яті, процесорів). А також будуть перевірені показники кластеру Kubernetes в різних конфігураціях і методів розгортання та обрано оптимальні.

4.2 Конфігурація тестового оточення

Для конфігурації тестового отчення буде використовуватись:

1. Virtual Box (програма віртуалізації для операційних систем)
2. Docker (інструмент для управління ізольованими Linux-контейнерами)

3. Kubectl (процес, що забезпечує обмін даними між **Kubernetes** master і робочим вузлом)
4. Minikube (інструмент, що дозволяє легко запускати Kubernetes на локальній машині)

Спочатку необхідно встановити Virtual Box на хостову ОС, після чого в ньому розгорнути Minikube та створити ВМ на ОС (Ubuntu 20.04 server) на Minikube та ВМ виділити по 4гб ОЗУ та 4 ЦП.

Також встановити Kubectl для взяємодії з Minikube. Таки чином, хостова ОС буде в ролі Master node та Minikube worker node.

Після цього необхідно створити Docker контейнери для фронтенду з використанням серверу Nginx, webapp написаного на мові програмування Java та Logic, котра реалізована на Python3. Для створення докер контейнерів необхідно встановити Docker на локальну машину, після чого можна приступити до написання Dockerfile для кожного з 3 мікро-додатків, але перед цим необхідно створити білди з усіма залежностями. Так для **sa-frontend** необхідно:

1. Встановити платформу Node.js
2. Втановити npm (скориставшись командою npm install)

Після цього необхідно перейти в папка sa-frontend та виконати команду npm run build, що створить папку build з всіма необхідними статичними файлами для React додатку фронтенду. Запустивши додаток на localhost, він буде виглядати наступним чином (Рис 4.3).

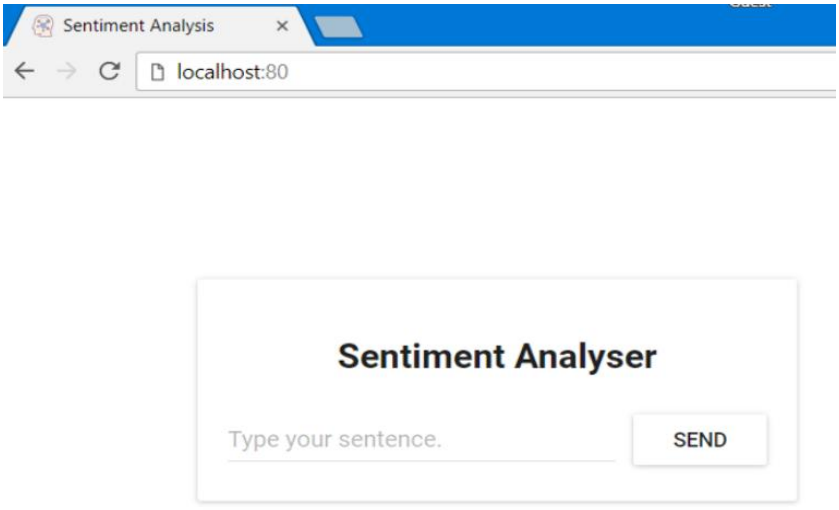

Рис. 4.3 Графічний інтрфейс експерементального додатку

При цьому він не буде виконувати ніякого функціоналу, оскільки sa-webapp та sa-logic ще не запущені.

Налаштування sa-webapp

Для того, щоб розгорнути Spring-додаток, знадобиться JDK8 і Maven і правильно налаштовані змінні середовища. Після цього необхідно упакувати додаток в jar файли, в моєму випадку для цього треба перейти в папку sa-webapp і виконати команду nvm install. Після виконання цієї команди у папці sa-webapp буде створено директорію target. Тут буде знаходитись Java-додаток, упакований у jar-файл, представлений файлом sentiment-analysis-web-0.0.1-SNAPSHOT.jar. Для запуску java додатку необхідно перейти в папку target та виконати наступну команду:

```
java -jar sentiment-analysis-web-0.0.1-SNAPSHOT.jar
```

Налаштування sa-logic

Для того, щоб запустити Python-додаток, у вас повинні бути встановлені Python 3 і Pip, і потрібно, щоб були правильно налаштовані відповідні змінні середовища. Для встановлення залежностей в моєму випадку необхідно перейти в папку sa-logic та виконати наступну команду:

- `python -m pip install -r requirements.txt`

Після створення білдів можна описати вище перечислені налаштування в докер файли, в випадку з sa-frontent та sa-webapp можна використати білди котрі були створення для підняття веб додатку без використання Docker та Kubernetes.

Таким чином, ми матимемо 3 Dockerfile:

```
1  FROM nginx
2  COPY build /usr/share/nginx/html
```

Рисунок 4.4 Dockerfile для фронтенду sa-frontend

```
1  FROM openjdk:8-jdk-alpine
2  ENV SA_LOGIC_API_URL http://localhost:5000
3  ADD target/sentiment-analysis-web-0.0.1-SNAPSHOT.jar /
4  EXPOSE 8080
5  CMD ["java", "-jar", "sentiment-analysis-web-0.0.1-SNAPSHOT.jar",
   "--sa.logic.api.url=${SA_LOGIC_API_URL}"]
```

Рис. 4.5 Dockerfile для вебдодатку sa-webapp

```
FROM python:3.6-slim
COPY sa /app
WORKDIR /app
RUN pip3 install -r requirements.txt && \
    python3 -m textblob.download_corpora
EXPOSE 5000
ENTRYPOINT ["python3"]
CMD ["sentiment_analysis.py"]
```

Рис. 4.6 Dockerfile для аналізатора тексту sa-logic

Після написання Docker файлів необхідно запушити їх на Docker Hub для подальшого їх використання в кластері Kubernetes.

Після цього, можна починати створювати manifest файли для розгортання цього додатку на Kubernetes кластері. Файли конфігурації для pods матимуть наступний вигляд:

```
apiVersion: v1
kind: Pod
metadata:
  name: sa-frontend
  labels:
    app: sa-frontend
spec:
  containers:
    - image: Mortys3/sentiment-analysis-frontend
      name: sa-frontend
      ports:
        - containerPort: 80
```

Рис. 4.7 Файл конфігурацій sa-frontend pod

```
apiVersion: v1
kind: Pod
metadata:
  name: sa-web-app
  labels:
    app: sa-web-app
spec:
  containers:
  - image: Mortys3/sentiment-analysis-web-app
    imagePullPolicy: Always
    name: sa-web-app
    env:
      - name: SA_LOGIC_API_URL
        value: "http://sa-logic"
    ports:
      - containerPort: 8080
```

Рис. 4.8 Файл конфігурацій sa-webapp pod

```yaml
apiVersion: v1
kind: pod
metadata:
  name: sa-logic
  labels:
    app: sa-logic
spec:
  containers:
    - image: Mortys3/sentiment-analysis-logic
      imagePullPolicy: Always
      name: sa-logic
      ports:
        - containerPort: 5000
```

Рис. 4.9 Файл конфігурацій sa-logic pod

Показники будуть зніматися не в єдиному типі налаштування кластеру Kubernetes, тому в іншому налаштуванні будемо використовувати Deployments, файли конфігурації яких, матимуть наступний вигляд:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sa-frontend
  labels:
    app: sa-frontend
spec:
  selector:
    matchLabels:
      app: sa-frontend
  replicas: 4
  minReadySeconds: 15
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 2
      maxSurge: 4
  template:
    metadata:
      labels:
        app: sa-frontend
    spec:
      containers:
        - image: mortys3/sentiment-analysis-frontend:minikube
          imagePullPolicy: Always
          name: sa-frontend
          ports:
            - containerPort: 80
```

Рис. 4.10 Файл конфігурації типу Deployment для sa-frontend

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sa-web-app
  labels:
    app: sa-web-app
spec:
  selector:
    matchLabels:
      app: sa-web-app
  replicas: 2
  minReadySeconds: 15
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
  template:
    metadata:
      labels:
        app: sa-web-app
    spec:
      containers:
      - image: rinormaloku/sentiment-analysis-web-app
        imagePullPolicy: Always
        name: sa-web-app
        env:
          - name: SA_LOGIC_API_URL
            value: "http://sa-logic"
        ports:
          - containerPort: 8080
```

Рис. 4.11 Файл конфігурації типу Deployments sa-webapp

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sa-logic
  labels:
    app: sa-logic
spec:
  selector:
    matchLabels:
      app: sa-logic
  replicas: 2
  minReadySeconds: 15
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
  template:
    metadata:
      labels:
        app: sa-logic
    spec:
      containers:
        - image: rinormaloku/sentiment-analysis-logic
          imagePullPolicy: Always
          name: sa-logic
          ports:
            - containerPort: 5000
```

Рис. 4.12 Файл конфігурації типу Deployment для sa-logic

Оскільки, Deployments це об’єдняння Pods необхідно створити Service типу loadbalncer для кожного з мікросервісів. В моєму випадку 3 мікросервіси тому необхідно створити 3 loadbalancers відповідно:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: sa-frontend-lb
spec:
  type: LoadBalancer
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: sa-frontend
```

Рис. 4.13 Файл конігурації типу LoadBalancer(Service) для sa-frontend

```yaml
apiVersion: v1
kind: Service
metadata:
  name: sa-web-app-lb
spec:
  type: LoadBalancer
  ports:
  - port: 80
    protocol: TCP
    targetPort: 8080
  selector:
    app: sa-web-app
```

Рис. 4.14 Файл конігурації типу LoadBalancer(Service) для sa-webapp

```yaml
apiVersion: v1
kind: Service
metadata:
  name: sa-logic
spec:
  type: LoadBalancer
  ports:
    - port: 80
      protocol: TCP
      targetPort: 5000
  selector:
    app: sa-logic
```

Рис. 4.15 Файл конігурації типу LoadBalancer(Service) для sa-logic

Додавання loadbalancers буде розподіляти навантаження між pods в кожному з deployments. Після запущення всіх deployments та сервісів типу loadbalancer додаток матиме наступний вигляд:

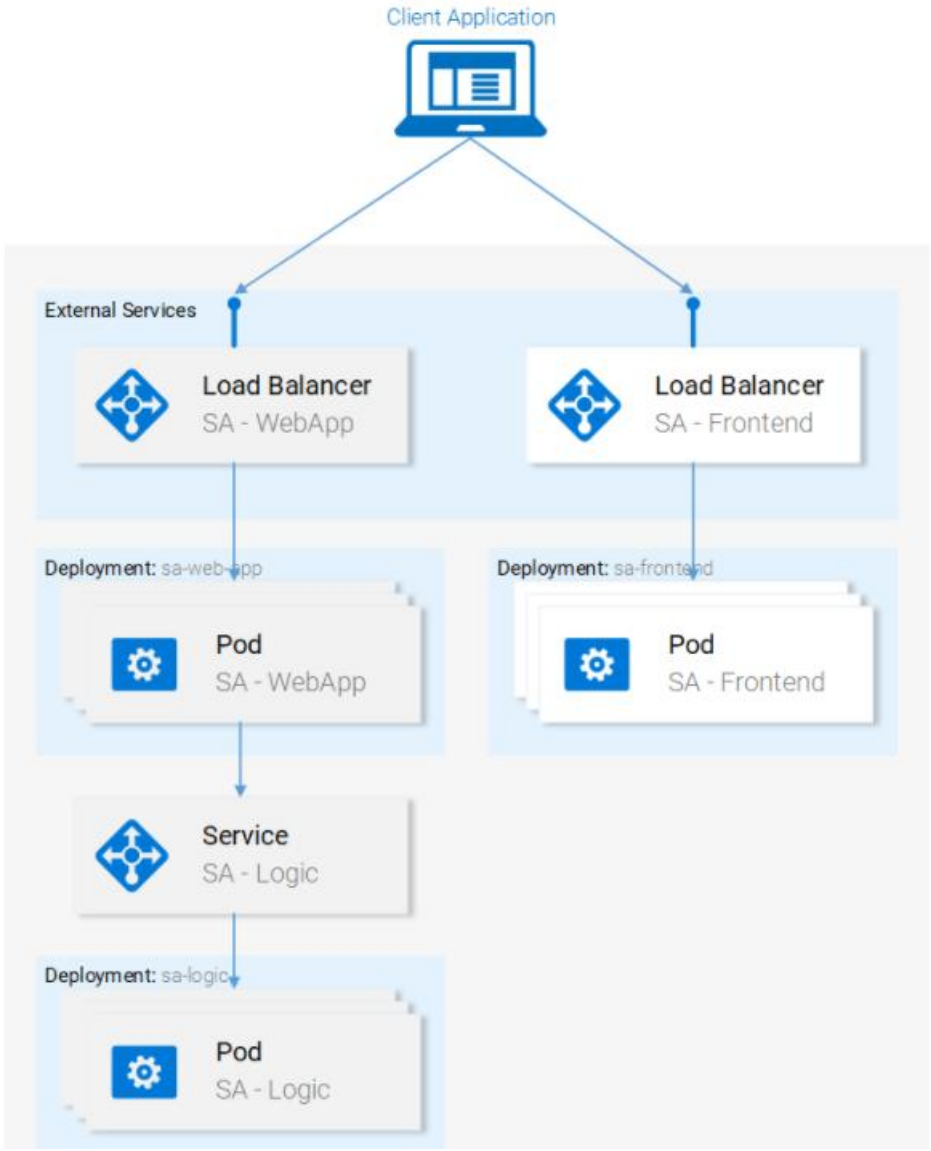

Рис. 4.16 Схема конфігурації кластеру Kubernetes з використанням deployments

4.3 Налаштування Apache JMeter для тестування навантаження

Apache JMeter — інструмент для проведення тестування навантаження, що розробляється Apache Software Foundation.[12]

Хоча спочатку JMeter розроблявся як засіб тестування web-додатків, в даний час він здатний проводити тести навантаження для FTP, LDAP, SOAP, JMS, POP3, IMAP, HTTP і TCP.

Цікавою є можливість створення великої кількості запитів за допомогою кількох комп'ютерів при керуванні цим процесом з одного з них. Архітектура, що підтримує плагіни сторонніх розробників, дозволяє доповнювати інструмент новими функціями, деякі з них будуть використані в моєму випадку.

Для роботи Jmeter необхідно встановити Java після чого зайти в папка bin та запустити файл ApacheJMeter. Оскільки в моєму випадку функціоналу програми не достатньо необхідно встановити plugin manager. Для цього необхідно скачати файл jmeter-plugin-manager-1.6.jar та перемістити в папку lib/ext. Після цього перезапустивши програму з’явится нове вікно з самим менеджером додатку. Необхідно встановити наступні доповнення:

1. 3 Basic Graphs
2. Additional Graphs

Після встановлення необхідних додатків, матимемо змогу використовувати такі корисні графіки як:

1. Active Threads Over Time
2. Bytes Throughput Over Time
3. Connect Times Over Time
4. Hits per Second
5. Response Codes per Second
6. Response Latencies Over Time
7. Response Times Distribution
8. Response Tmes Over Time
9. Response Times Percentlies
10. Transactions per Second

Для того щоб приступити до тестування додатку необхідно створити запити. В нашому випадку це POST та Get запити. Get запит буде спрямований на sa-frontend, а POST на сумісну роботу sa-webapp та sa-logic. Для побудови запиту необхідно дізнатися IP адресу та порт по якому треба надсилати запит, для sa-frontend в даному випадку це 192.168.99.101: 31806. З використанням JMeter досить обрати GET та вибрати протокол HTTP.

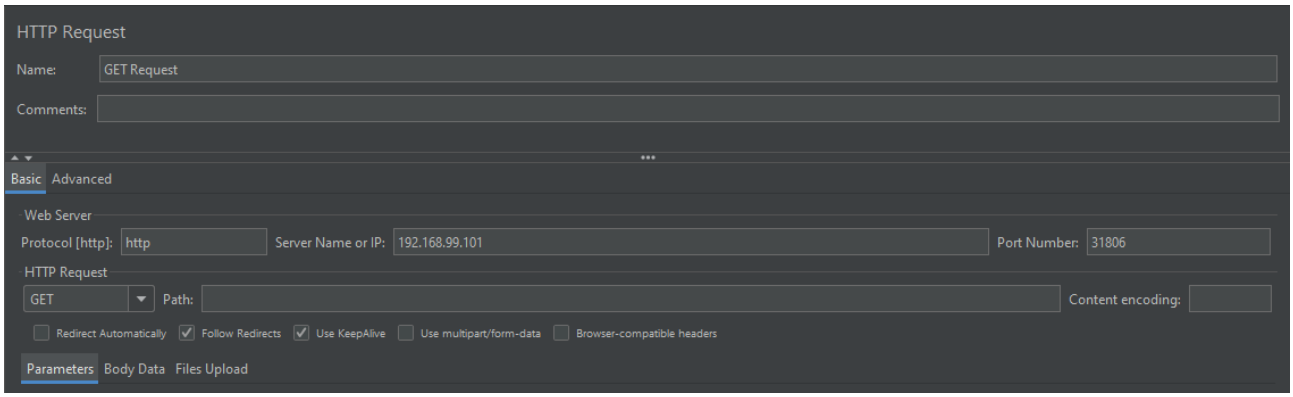

Рис. 4.17 Налаштування GET запиту для проведення тестування

Аналогічним способом необхідно налаштувати POST, але необхідно додати деякі параметри:

1. В Header manager (Content-type:application/json)
2. В Body Data ({"sentence":"Будь-яке речення"})

Також змінится тип запиту на POST і порт з 31806 на 31661

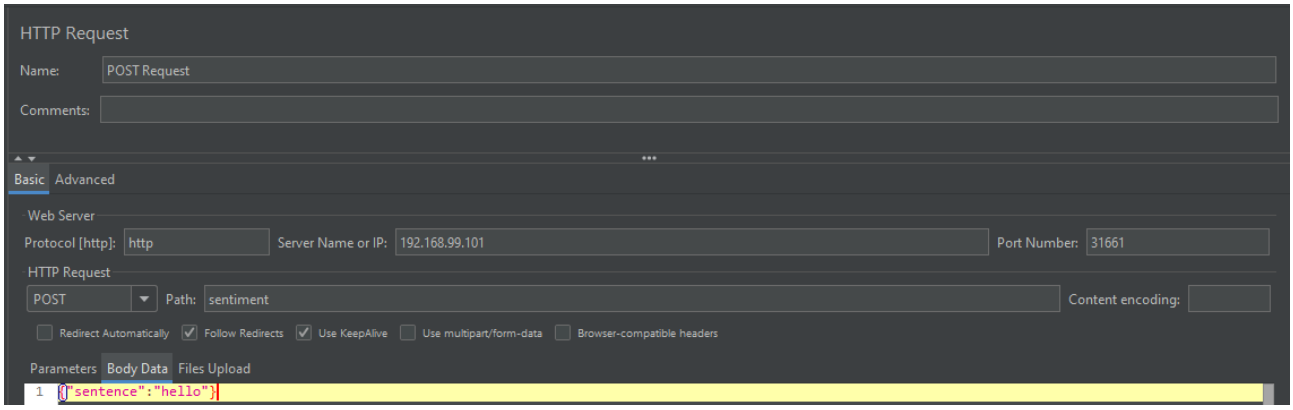

Рис. 4.18 Налаштування POST запиту

4.4 Результати дослідження

Після налаштування Apache JMeter можна приступати до тестування. Загалом буде протестовано 3 типи розгортання мікросервісного додатку:

1. Додаток розгорнутий на докер контейнерах (в рамках ВМ)
2. Додаток розгорнутий в кластері Kubernetes побудований лише з використанням Pods.
3. Додаток з використанням loadbalancers та deployments.

Після запуску додатку розпочнемо проводити тестування з використанням Apache JMeter. Тестування будемо проводити 500 thread (користувачі). Кожен користувач буде робити 1, 3, 6, 9 запитів з інтервалом в 2сек. Буде вимірюватись response time.

Розпочнему з 500 користувачів котрі роблять по 1 запиту POST та GET. Результати тестування додатку розгорнутого на Docker контейнерах для 500 запитів.

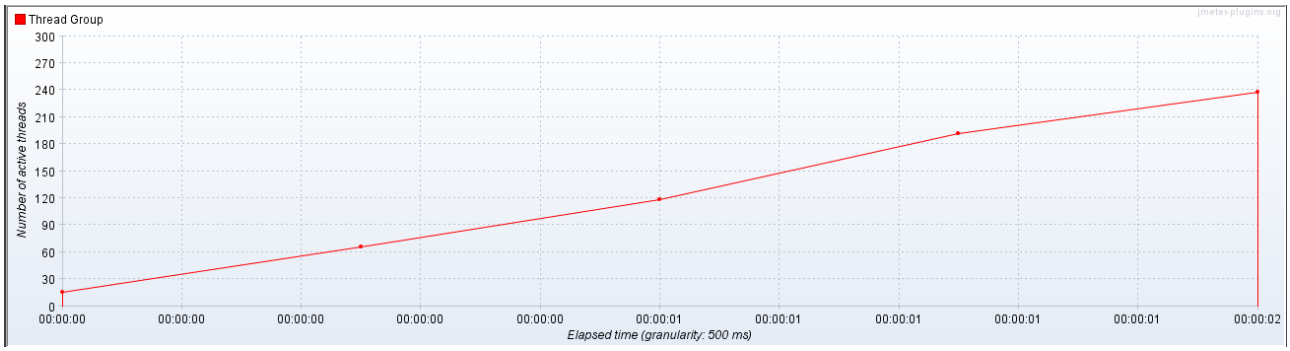

Рис. 4.19 Підключення користувачів до додатку

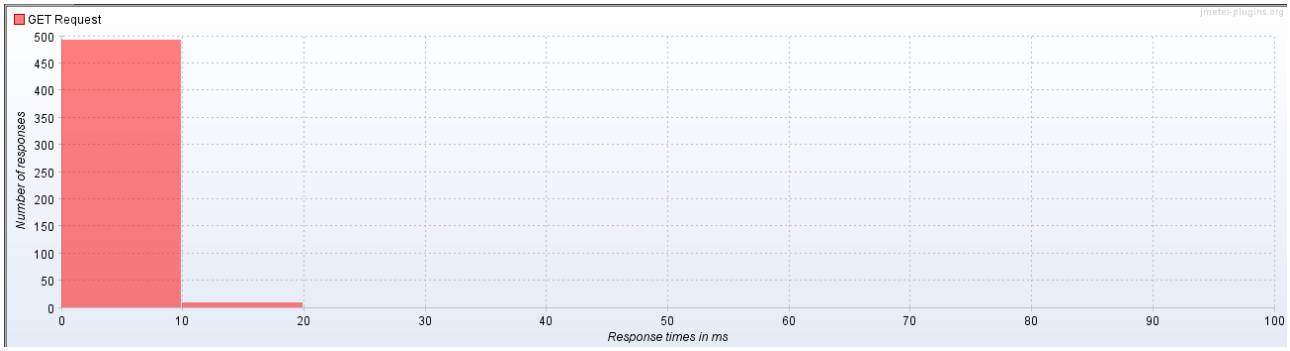

Рис. 4.20 Графік часу затримки для GET запитів

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
GET Request	500	2	0	15	1,92	0,00%	250,5/sec	194,48	30,58	795,0
TOTAL	500	2	0	15	1,92	0,00%	250,5/sec	194,48	30,58	795,0

Рис. 4.21 Середні результати тестування для GET запитів

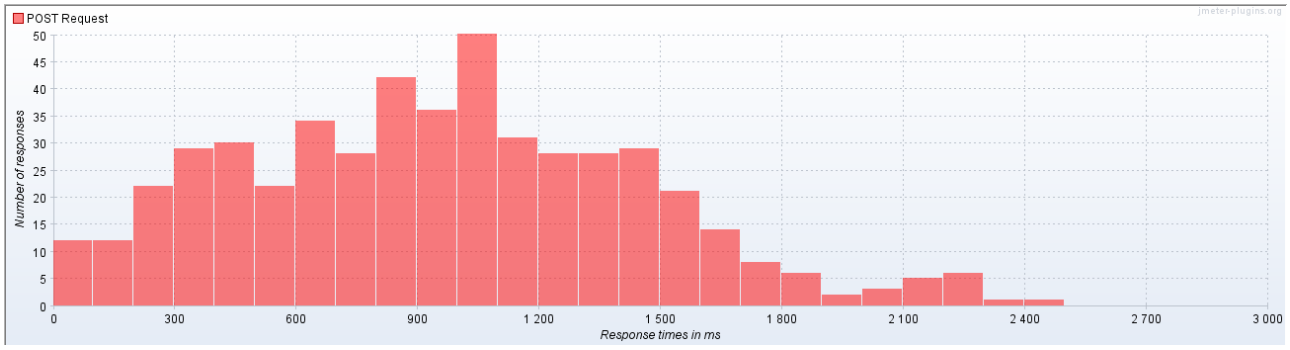

Рисунок 4.22 Графік часу затримки для POST запитів

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
POST Request	500	958	17	2459	492,85	0,00%	133,1/sec	24,70	28,99	190,0
TOTAL	500	958	17	2459	492,85	0,00%	133,1/sec	24,70	28,99	190,0

Рис. 4.23 Середні результати тестування POST запитів

Проведемо аналогічне тестування для для такої ж кількості користувачів, але кожен з них буде робити по 1 запиту кожні 2 секунди, тоб-то в цілому повинно бути 1500 запитів POST та GET.

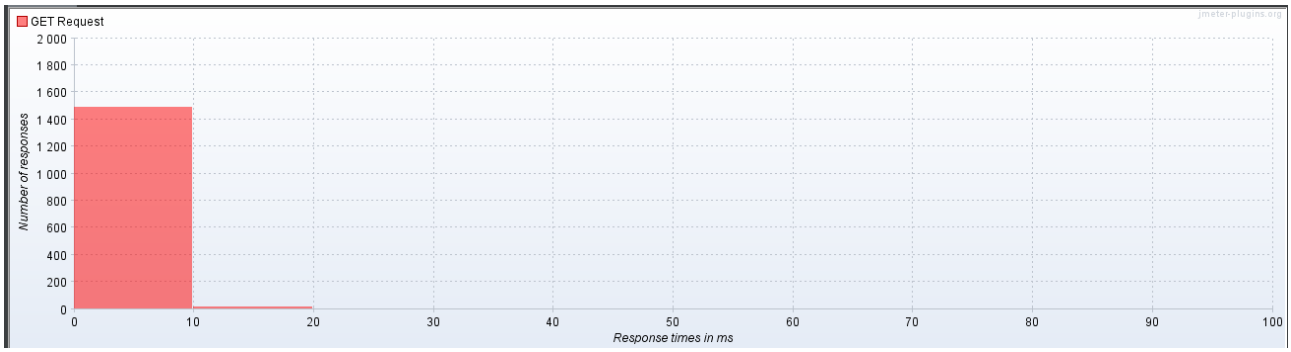

Рис. 3.23 Графік часу затримки для GET запитів

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
GET Request	1500	1	0	16	1,67	0,00%	216,3/sec	167,92	26,40	795,0
TOTAL	1500	1	0	16	1,67	0,00%	216,3/sec	167,92	26,40	795,0

Рис. 4.24 Середні результати тестування для GET запитів

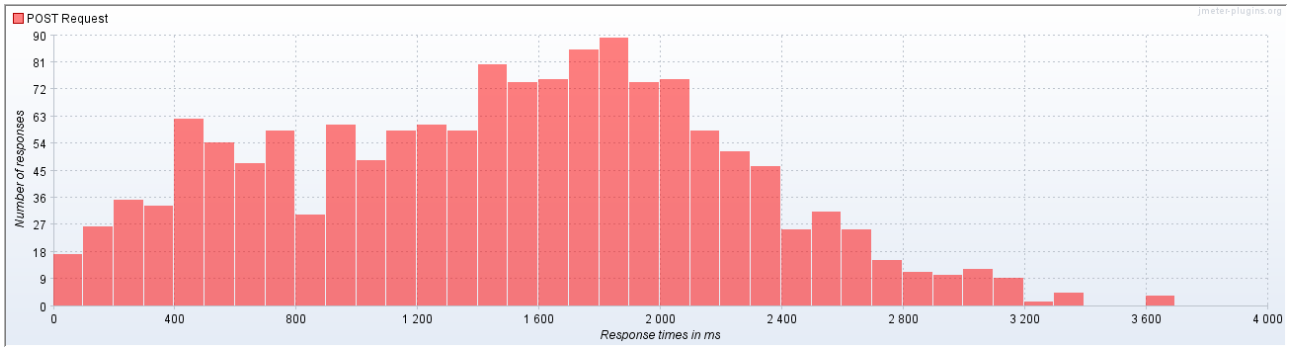

Рисунок 4.25 Графік часу затримки для POST запитів

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
POST Request	1500	1484	31	3684	731,80	0,00%	211,8/sec	39,30	46,13	190,0
TOTAL	1500	1484	31	3684	731,80	0,00%	211,8/sec	39,30	46,13	190,0

Рис. 4.26 Середні результати тестування POST запитів

Проведемо аналогічні тести для 3000 запитів:

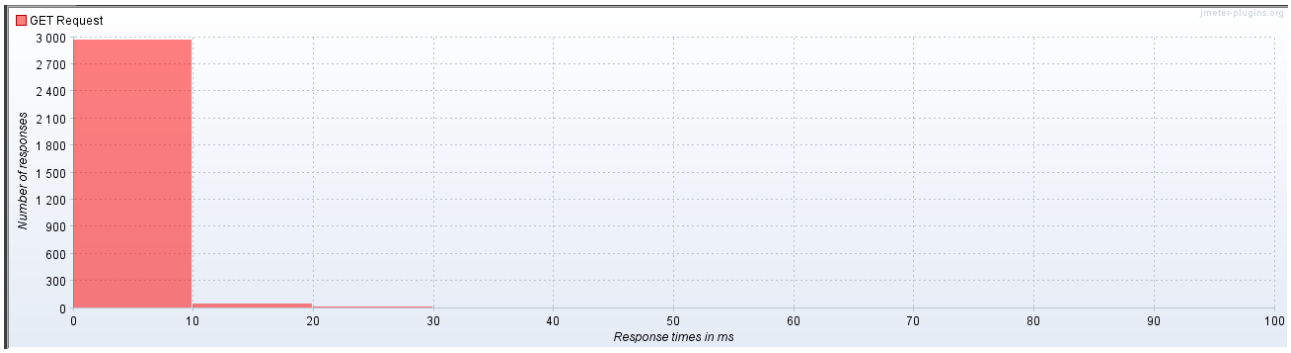

Рис. 4.27 Графік часу затримки для GET запитів

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
GET Request	3000	1	0	22	1,89	0,00%	234,5/sec	182,05	28,62	795,0
TOTAL	3000	1	0	22	1,89	0,00%	234,5/sec	182,05	28,62	795,0

Рис. 4.28 Середні результати тестування для GET запитів

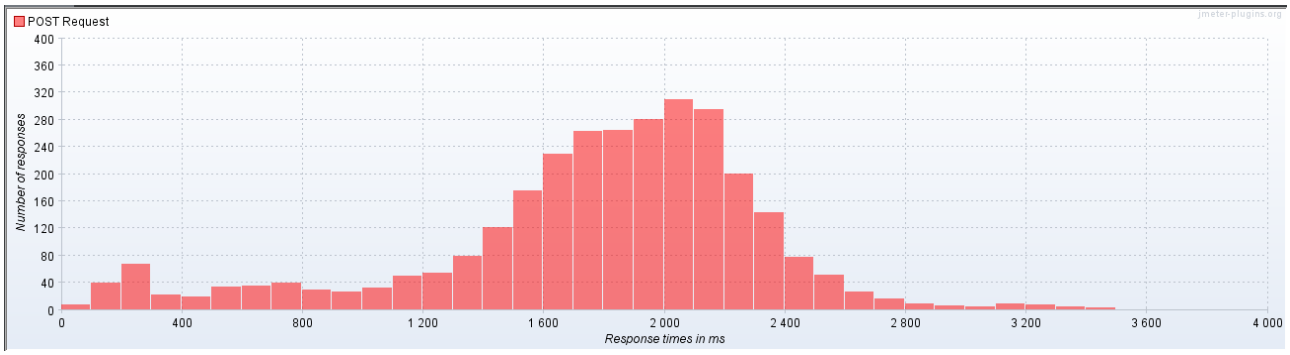

Рис. 4.29 Графік часу затримки для POST запитів

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
POST Request	3000	1772	61	3407	560,22	0,00%	219,4/sec	40,72	47,79	190,0
TOTAL	3000	1772	61	3407	560,22	0,00%	219,4/sec	40,72	47,79	190,0

Рис. 4.30 Середні результати тестування POST запитів

Проведемо аналогічне тестування для 4500 POST та GET запитів:

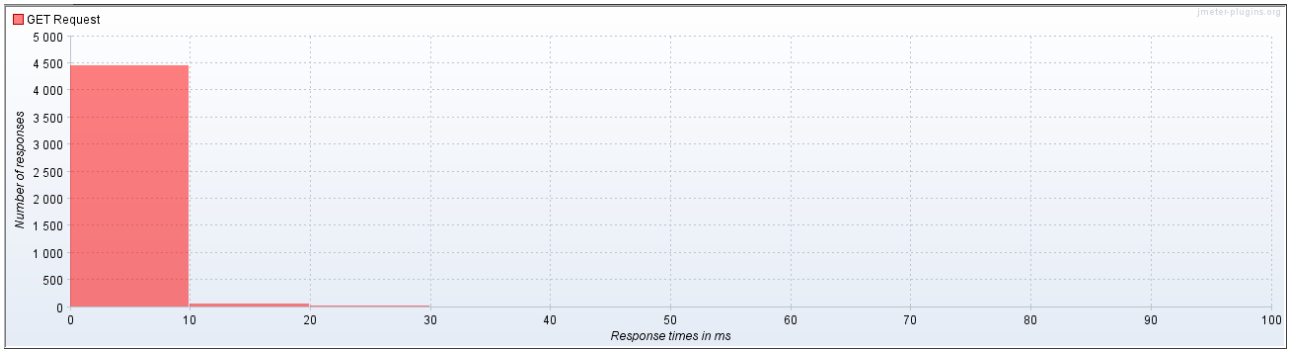

Рис. 4.31 Графік часу затримки для GET запитів

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
GET Request	4500	1	0	27	1,82	0,00%	236,0/sec	183,25	28,81	795,0
TOTAL	4500	1	0	27	1,82	0,00%	236,0/sec	183,25	28,81	795,0

Рис. 4.32 Середні результати тестування для GET запитів

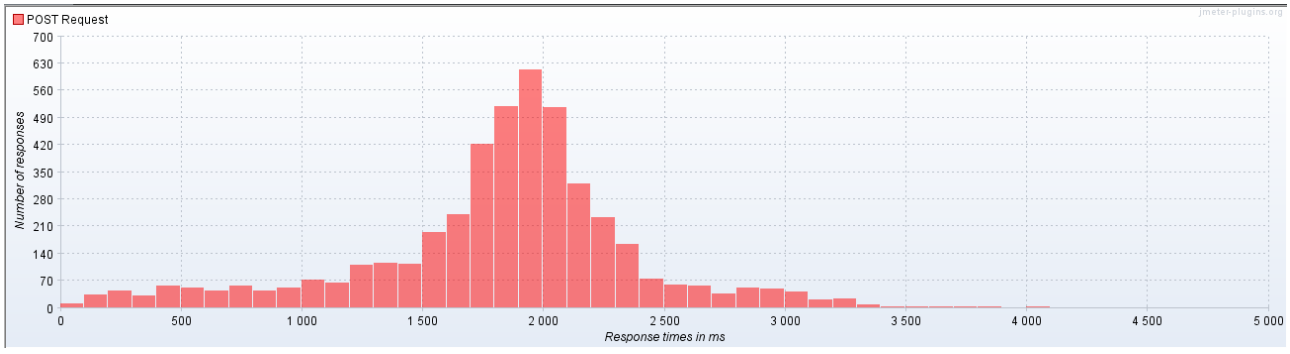

Рис. 4.33 Графік часу затримки для POST запитів

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
POST Request	4500	1823	39	4025	562,23	0,00%	234,8/sec	43,56	51,13	190,0
TOTAL	4500	1823	39	4025	562,23	0,00%	234,8/sec	43,56	51,13	190,0

Рис. 4.34 Середні результати тестування POST запитів

Проведемо тест на стійкість до навантажень. Кожні 2 секунди будемо збільшувати кількість користувачів на 100, котрі будуть робити по 1 запиту POST та GET.

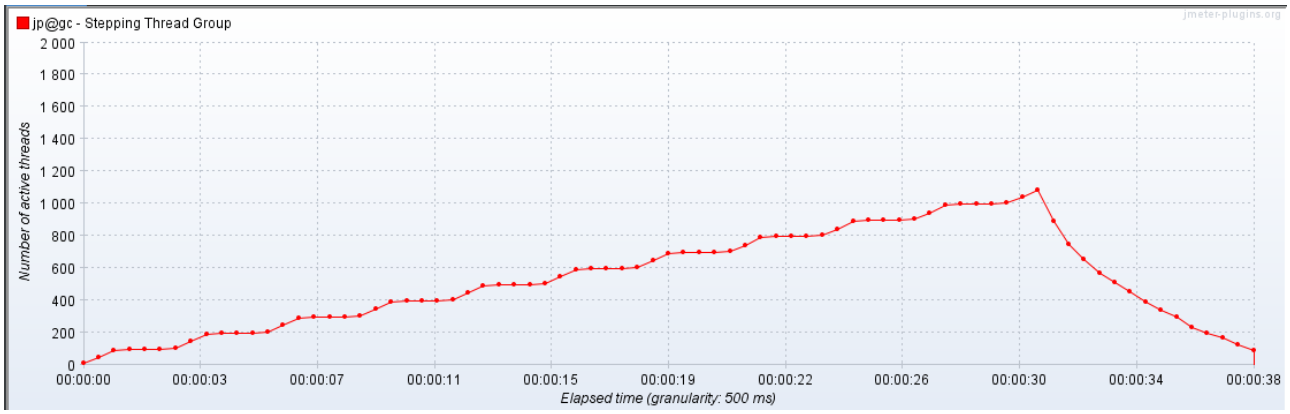

Рис. 4.35 Підключення користувачів до додатку

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
GET Request	7113	19	0	3045	182,50	0,62%	192,6/sec	151,49	23,37	805,4
TOTAL	7113	19	0	3045	182,50	0,62%	192,6/sec	151,49	23,37	805,4

Рис. 4.36 Середні результати тестування для GET запитів

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
POST Request	7069	2497	54	7209	1325,04	16,17%	186,6/sec	97,23	31,92	533,6
TOTAL	7069	2497	54	7209	1325,04	16,17%	186,6/sec	97,23	31,92	533,6

Рис. 4.37 Середні результати тестування POST запитів

Таким чином, додаток розгорнутий на Docker контейнерах має наступні результати.

Таблиця 4.1

Показники додатку побудованого на Docker контейнерах

Samples	Response Time, ms	
	GET	POST
500	2	958
1500	1	1484
3000	1	1772
4500	1	1823

Середня затримка GET запиту при роботі з 500 користувачів 1.25 мс, при тестуванні на стійкість до навантажень 19 мс.

Середня затримка на POST запити 1509 мс, при тестування на стійкість до навантажень 2497 мс.

Загалом при тестуванні на максимальну кількість користувачів котрі роблять POST та GET запити додаток зміг обслуговувати одночасно 1080 користувачів.

2. Результати додатку розгорнутого в кластері Kubernetes побудованого з використанням pods та та сервісу взаємодії з ними.

Розпочнемо знімати показники для 500 запитів GET та POST.

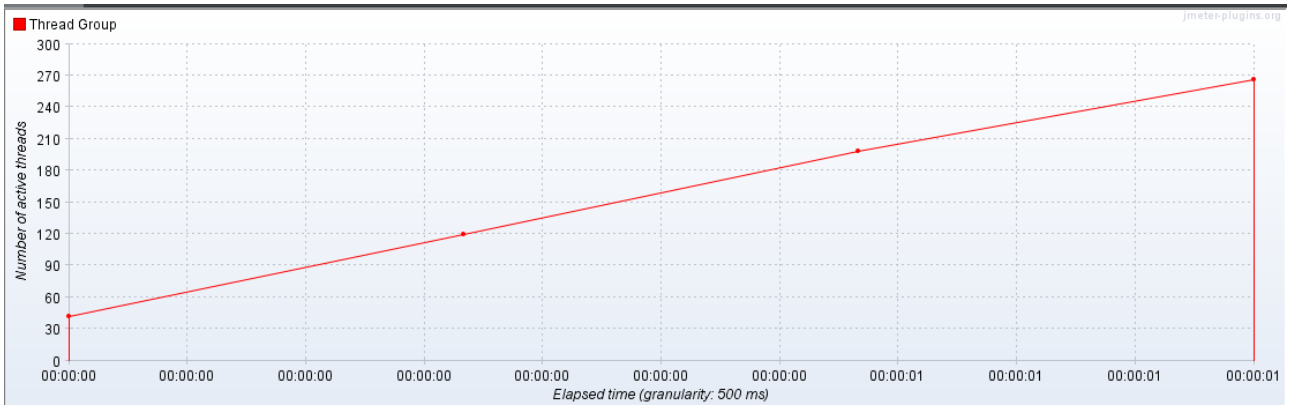

Рис. 4.38 Підключення користувачів до додатку

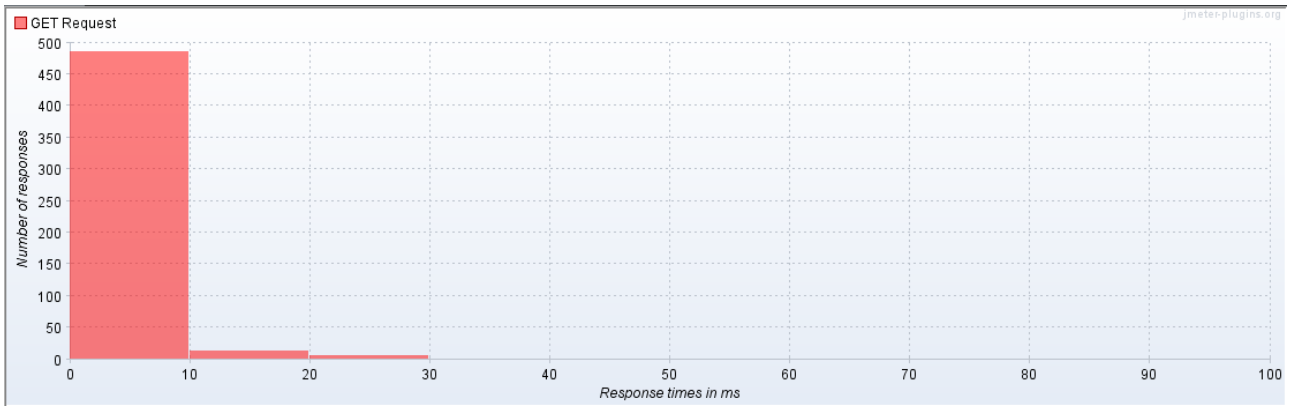

Рис. 4.39 Графік часу затримки для GET запитів

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
GET Request	500	2	0	35	2,88	0,00%	250,9/sec	194,77	30,62	795,0
TOTAL	500	2	0	35	2,88	0,00%	250,9/sec	194,77	30,62	795,0

Рис. 4.40 Середні результати тестування для GET запитів

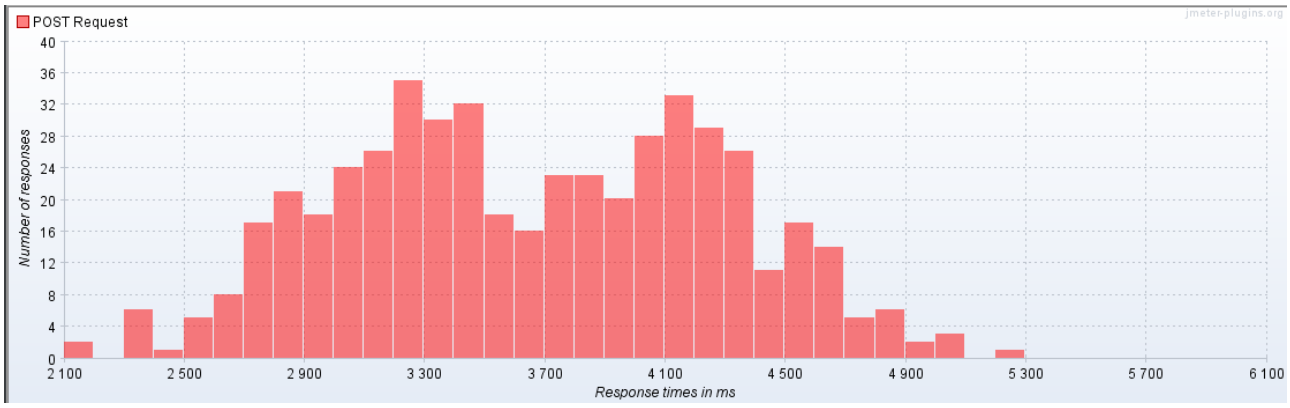

Рисунок 4.41 Графік часу затримки для POST запитів

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
POST Request	500	1821	2138	5244	617,68	0,00%	85,2/sec	14,48	17,22	174,0
TOTAL	500	1821	2138	5244	617,68	0,00%	85,2/sec	14,48	17,22	174,0

Рис. 4.42 Середні результати тестування POST запитів

Проведемо аналогічне тестування для 1500 запитів POST та GET:

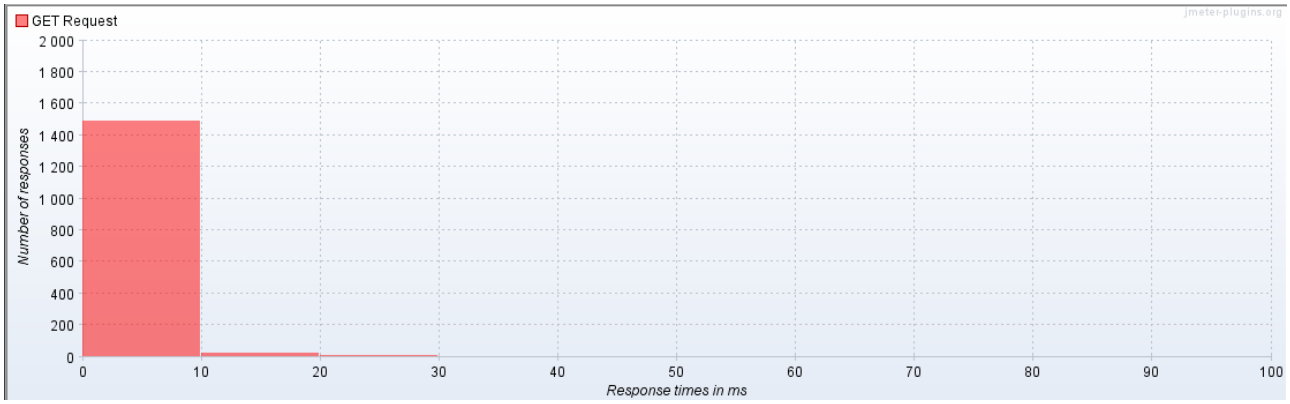

Рис. 4.43 Графік часу затримки для GET запитів

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
GET Request	1500	2	0	20	1,99	0,00%	170,1/sec	132,05	20,76	795,0
TOTAL	1500	2	0	20	1,99	0,00%	170,1/sec	132,05	20,76	795,0

Рис. 4.44 Середні результати тестування для GET запитів

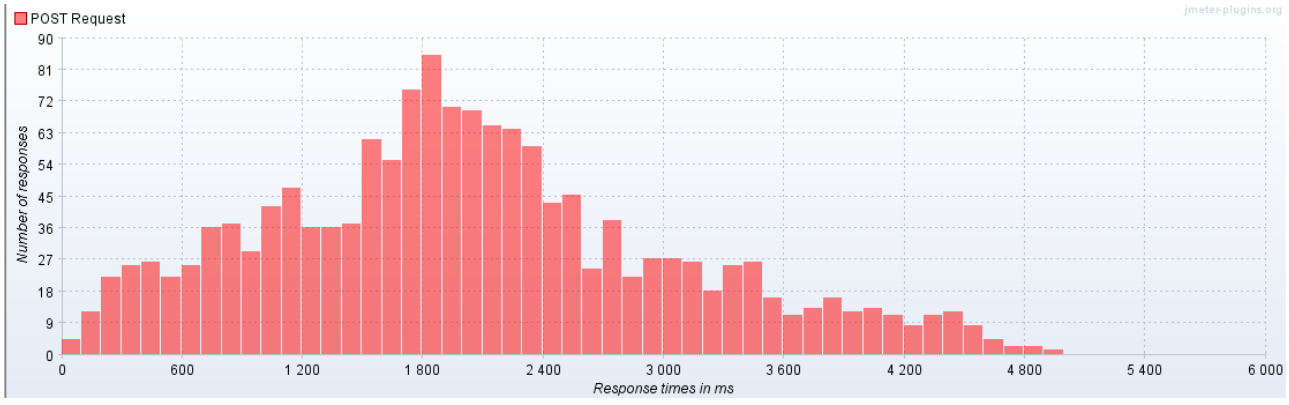

Рис. 4.45 Графік часу затримки для POST запитів

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
POST Request	1500	2036	67	4960	1009,08	0,00%	168,4/sec	28,62	34,05	174,0
TOTAL	1500	2036	67	4960	1009,08	0,00%	168,4/sec	28,62	34,05	174,0

Рис. 4.46 Середні результати тестування POST запитів

Проведемо аналогічне тестування для 3000 запитів POST та GET:

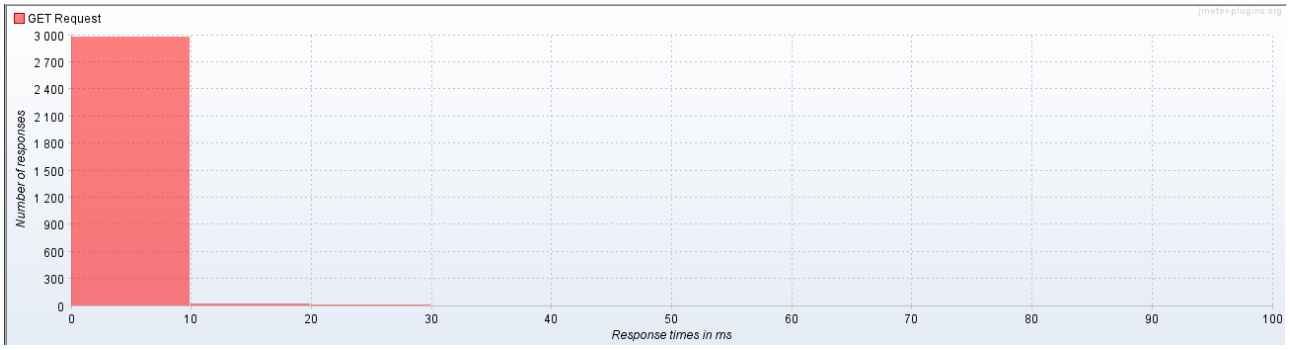

Рис. 4.47 Графік часу затримки для GET запитів

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
GET Request	3000	1	0	28	1,64	0,00%	195,3/sec	151,64	23,84	795,0
TOTAL	3000	1	0	28	1,64	0,00%	195,3/sec	151,64	23,84	795,0

Рисунок 4.48 Середні результати тестування для GET запитів

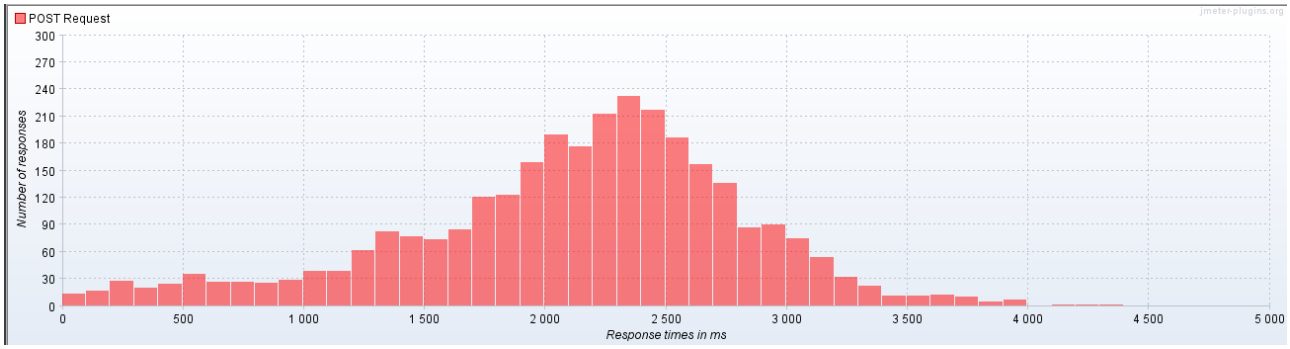

Рис. 4.49 Графік часу затримки для POST запитів

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
POST Request	3000	2117	29	4363	704,17	0,00%	194,9/sec	33,11	39,39	174,0
TOTAL	3000	2117	29	4363	704,17	0,00%	194,9/sec	33,11	39,39	174,0

Рис. 4.50 Середні результати тестування POST запитів

Проведемо аналогічне тестування для 4500 запитів POST та GET:

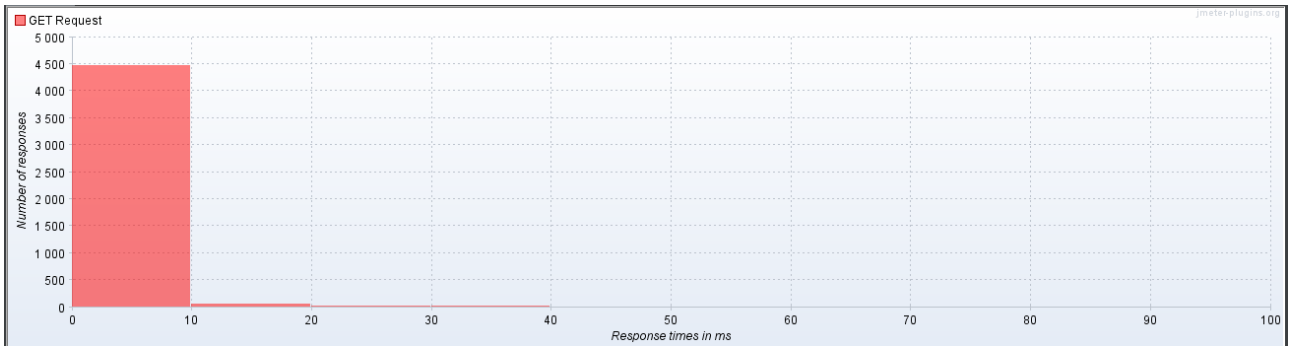

Рис. 4.51 Графік часу затримки для GET запитів

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
GET Request	4500	1	0	30	1,90	0,00%	198,1/sec	153,80	24,18	795,0
TOTAL	4500	1	0	30	1,90	0,00%	198,1/sec	153,80	24,18	795,0

Рис. 4.52 Середні результати тестування для GET запитів

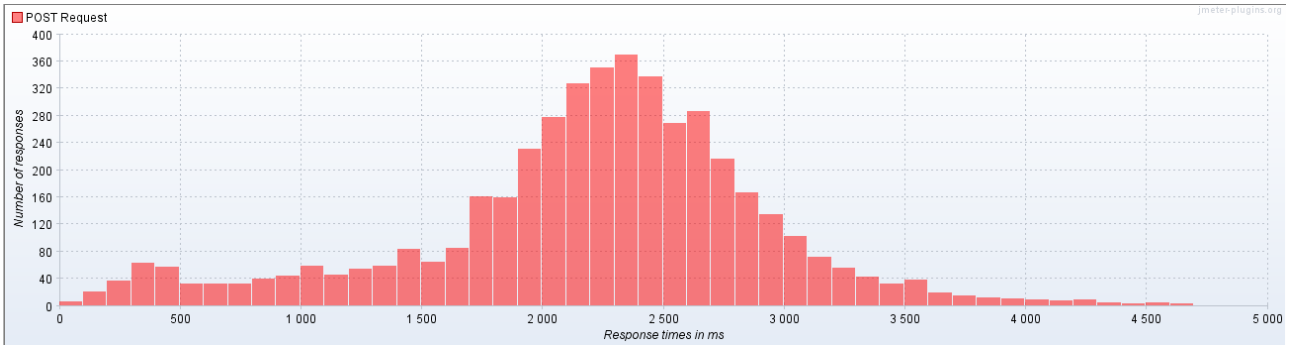

Рис. 4.53 Графік часу затримки для POST запитів

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
POST Request	4500	2203	19	4692	722,47	0,00%	198,0/sec	33,64	40,02	174,0
TOTAL	4500	2203	19	4692	722,47	0,00%	198,0/sec	33,64	40,02	174,0

Рис. 4.54 Середні результати тестування POST запитів

Проведемо тест на стійкість до високих навантажень з використанням Stepping Thread Group в Apache JMeter, тоб-то збільшуючи кожні 2 секунди на 100 користувачі, кожен з яких буде робити запит кожні 2 секунди, показники будемо вимірювати до появи першої помилки.

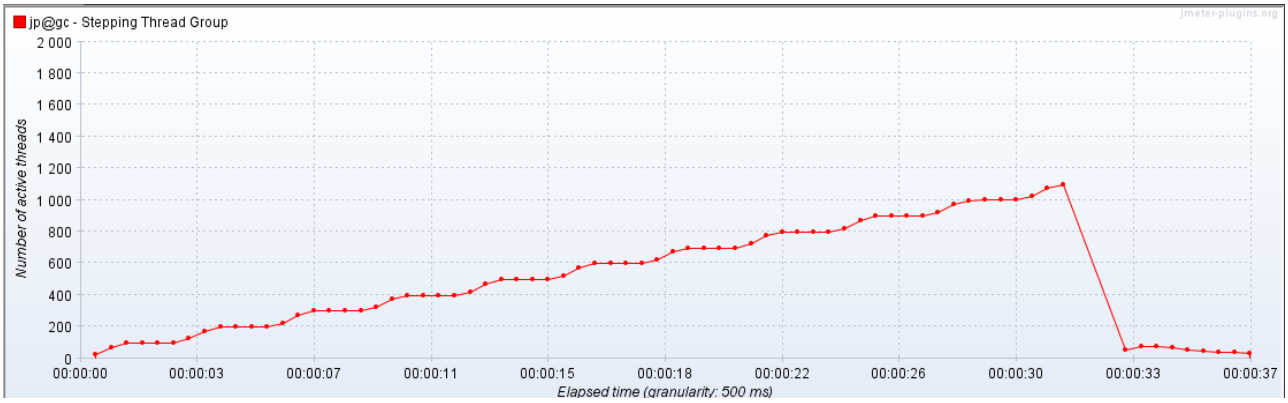

Рис. 4.55 Підключення користувачів до додатку

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
GET Request	4638	2	0	43	2,82	0,00%	173,2/sec	134,48	21,14	795,0
TOTAL	4638	2	0	43	2,82	0,00%	173,2/sec	134,48	21,14	795,0

Рис. 4.56 Середні результати тестування для GET запитів

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
POST Request	4638	2206	4	8457	1188,55	17,92%	172,4/sec	101,38	28,88	602,2
TOTAL	4638	2206	4	8457	1188,55	17,92%	172,4/sec	101,38	28,88	602,2

Рис. 4.57 Середні результати тестування POST запитів

Таким чином, додаток розгорнутий на Kubernetes кластері з використанням pods матиме наступні результати.

Таблиця 4.2

Показники додатку побудованого на Kubernetes кластері з використанням тільки pods

Samples	Response	Time, ms
	GET	POST
500	2	1821
1500	2	2036
3000	1	2117
4500	1	2203

Середня затримка GET запиту при роботі з 500 користувачів 1.5 мс, при тестуванні на стійкість до навантажень 2 мс.

Середня затримка на POST запити 2044 мс, при тестування на стійкість до навантажень 2206 мс.

Загалом при тестуванні на максимальну кількість користувачів котрі роблять POST та GET запити додаток зміг обслуговувати одночасно 1100 користувачів.

3 Результати додатку розгорнутого в кластері Kubernetes побудованого з використанням Deployments та loadbalancers.

Для запуску кластеру скористаємося командами:

1. minikube start
2. minikube service sa-frontend

Після запуску кластера почнемо проводити тестування з використанням Тестування будемо проводити 500 thread (користувачі). Кожен користувач буде робити 1, 3, 6, 9 запитів з інтервалом в 2сек. Буде вимірюватись response time.

Розпочнему з 500 користувачів котрі роблять по 1 запиту POST та GET.

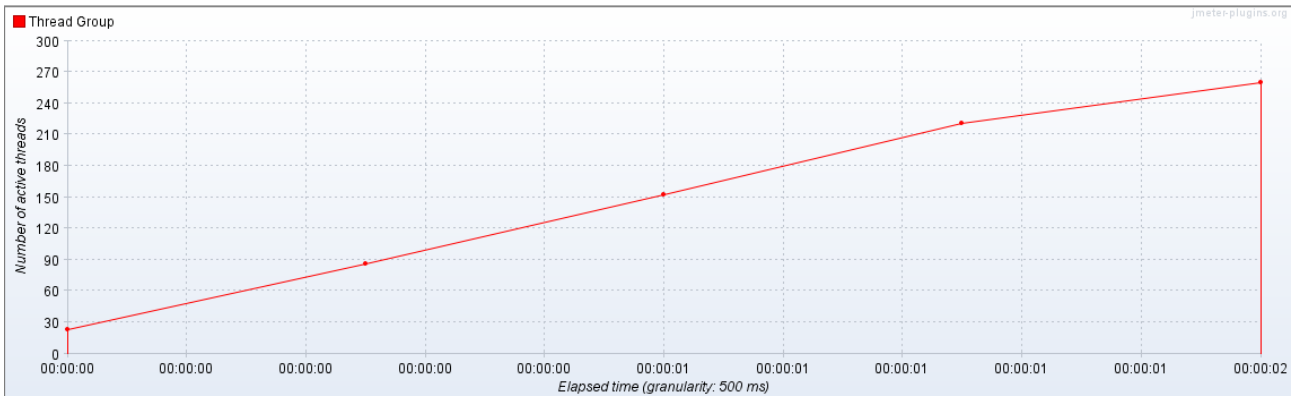

Рис. 4.58 Підключення користувачів до фронтенду

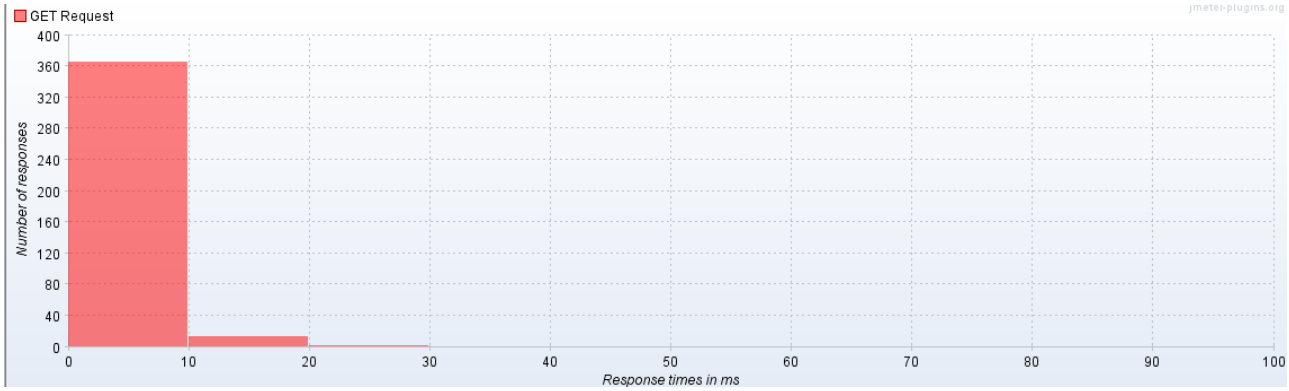

Рис. 4.59 Графік часу затримки для GET запитів

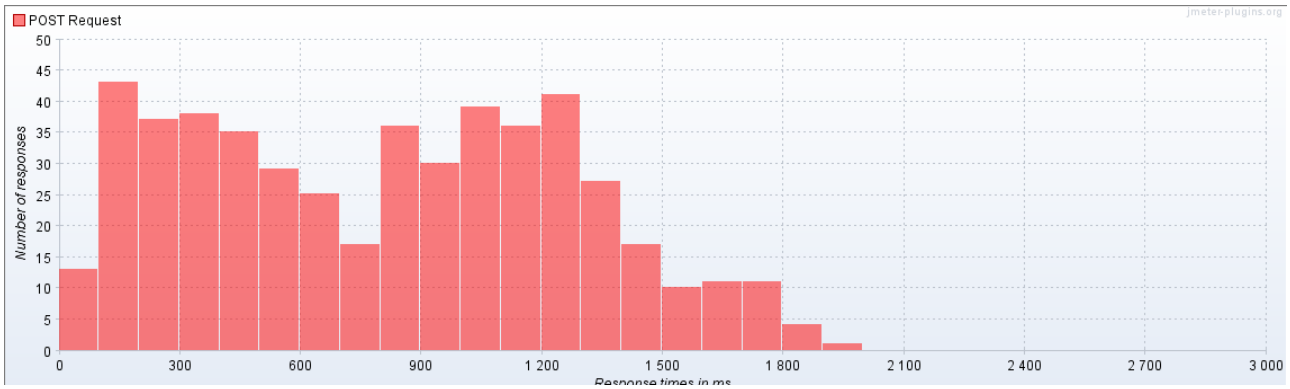

Рис. 4.60 Графік часу затримки для POST запитів

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Received KB/...	Sent KB/sec
GET Request	500	3	2	6	9	18	0	41	0,00%	250,3/sec	194,29	30,55
TOTAL	500	3	2	6	9	18	0	41	0,00%	250,3/sec	194,29	30,55

Рис. 4.61 Середні результати тестування для GET запитів

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
POST Request	500	813	57	1996	471,92	0,00%	161,6/sec	27,45	32,66	174,0
TOTAL	500	813	57	1996	471,92	0,00%	161,6/sec	27,45	32,66	174,0

Рис. 4.62 Середні результати тестування POST запитів

Буде проведено аналогічні експеремент, але з 1500, 3000 та 4500 запитів від 500 користувачів з інтервалом в 2 секунди.

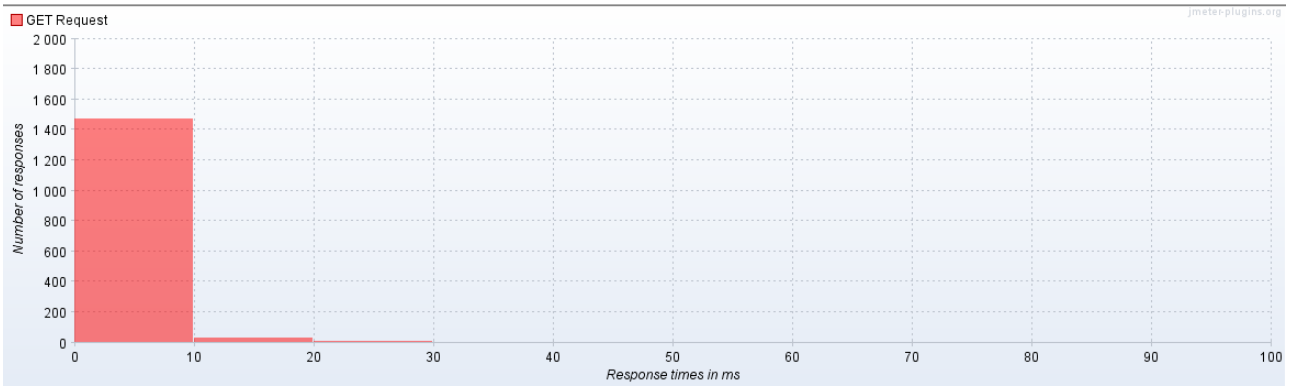

Рис. 4.63 Графік часу затримки для GET запитів

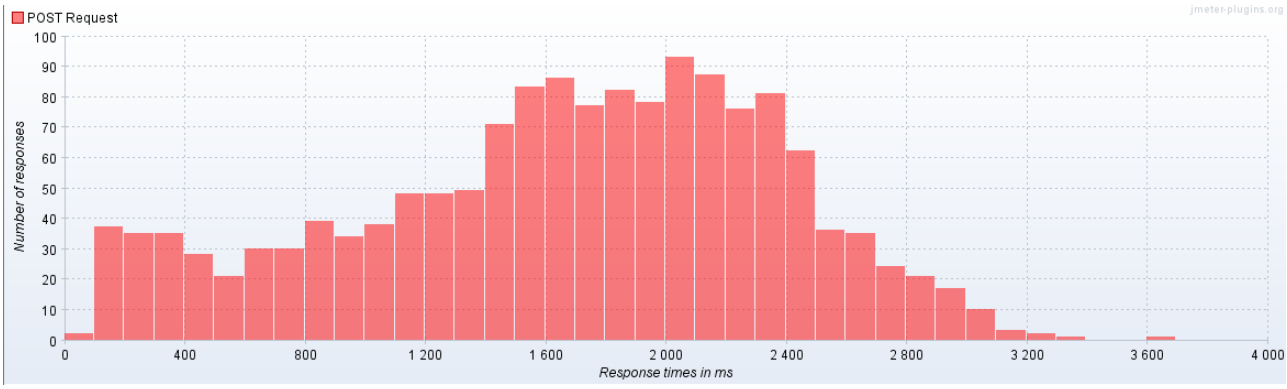

Рис. 4.64 Графік часу затримки для POST запитів

GET Request	1500	1	0	28	2,25	0,00%	207,1/sec	160,78	25,28	795,0
TOTAL	1500	1	0	28	2,25	0,00%	207,1/sec	160,78	25,28	795,0

Рис. 4.65 Середні результати тестування для GET запитів

POST Request	1500	1663	87	3665	713,12	0,00%	195,6/sec	33,24	39,54	174,0
TOTAL	1500	1663	87	3665	713,12	0,00%	195,6/sec	33,24	39,54	174,0

Рис. 4.66 Середні результати тестування POST запитів

Повторимо для 3000 запитів:

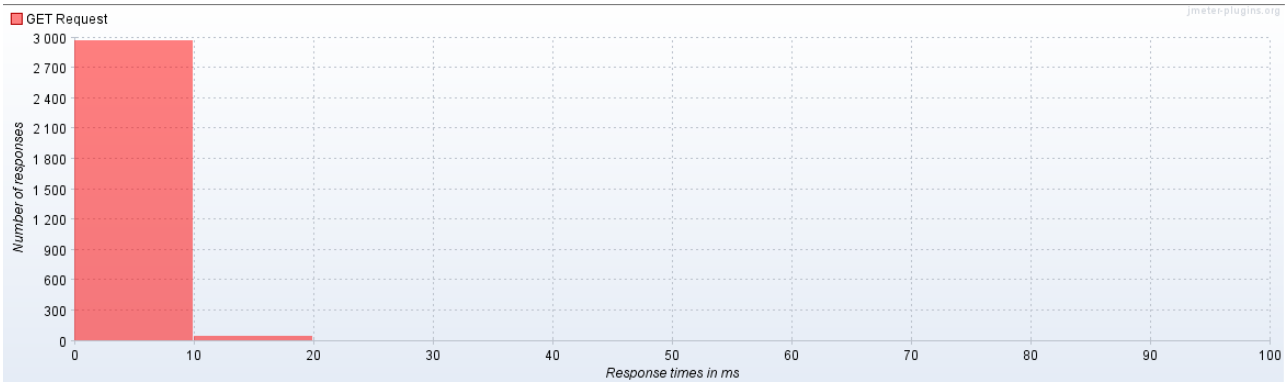

Рис. 4.67 Графік часу затримки для GET запитів

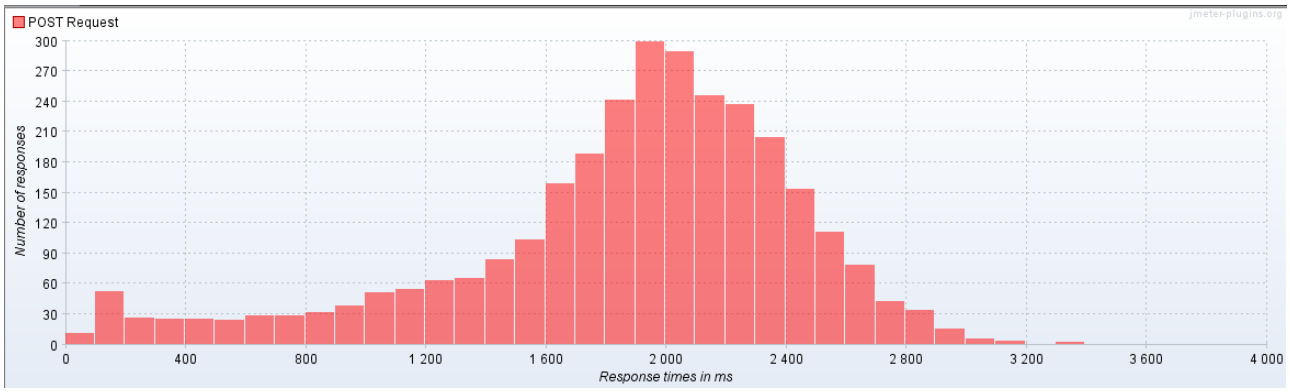

Рис. 4.68 Графік часу затримки для POST запитів

GET Request	3000	1	0	19	1,83	0,00%	221,7/sec	172,09	27,06	795,0
TOTAL	3000	1	0	19	1,83	0,00%	221,7/sec	172,09	27,06	795,0

Рисунок 4.69 Середні результати тестування для GET запитів

POST Request	3000	1867	64	3360	587,52	0,00%	209,2/sec	35,55	42,29	174,0
TOTAL	3000	1867	64	3360	587,52	0,00%	209,2/sec	35,55	42,29	174,0

Рис. 4.70 Середні результати тестування POST запитів

Повторимо для 4500 запитів POST та GET:

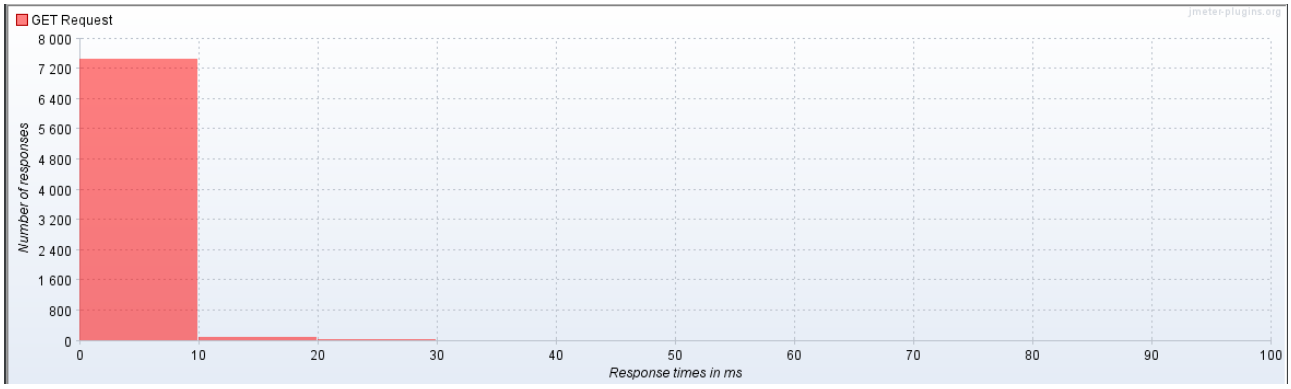

Рис. 4.71 Графік часу затримки для GET запитів

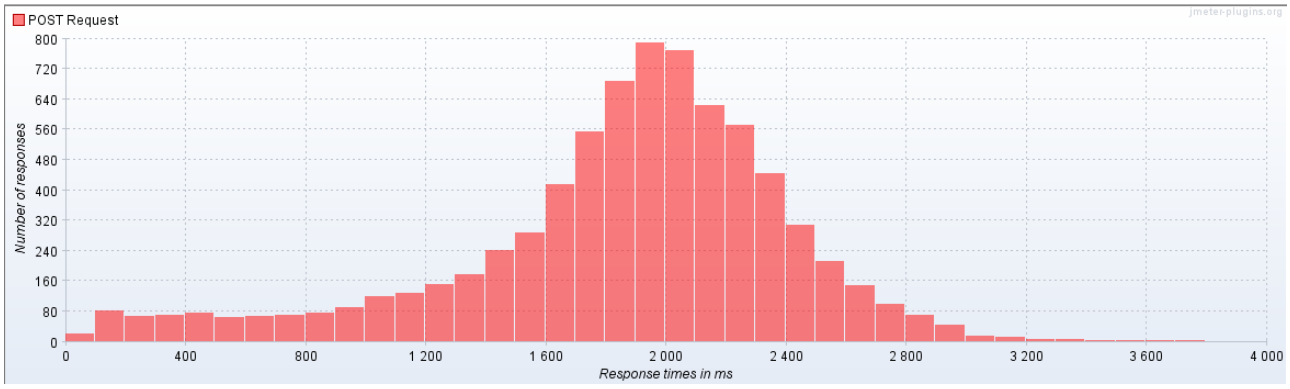

Рис. 4.72 Графік часу затримки для POST запитів

GET Request	7500	1	0	28	1,74	0,00%	41,5/sec	32,24	5,07	795,0
TOTAL	7500	1	0	28	1,74	0,00%	41,5/sec	32,24	5,07	795,0

Рис. 4.73 Середні результати тестування для GET запитів

POST Request	7500	1853	25	3752	561,78	0,00%	41,5/sec	7,06	8,39	174,0
TOTAL	7500	1853	25	3752	561,78	0,00%	41,5/sec	7,06	8,39	174,0

Рис. 4.74 Середні результати тестування POST запитів

Проведемо тест на стійкість до високих навантажень шляхом, показники будемо вимірювати до появи першої помилки.

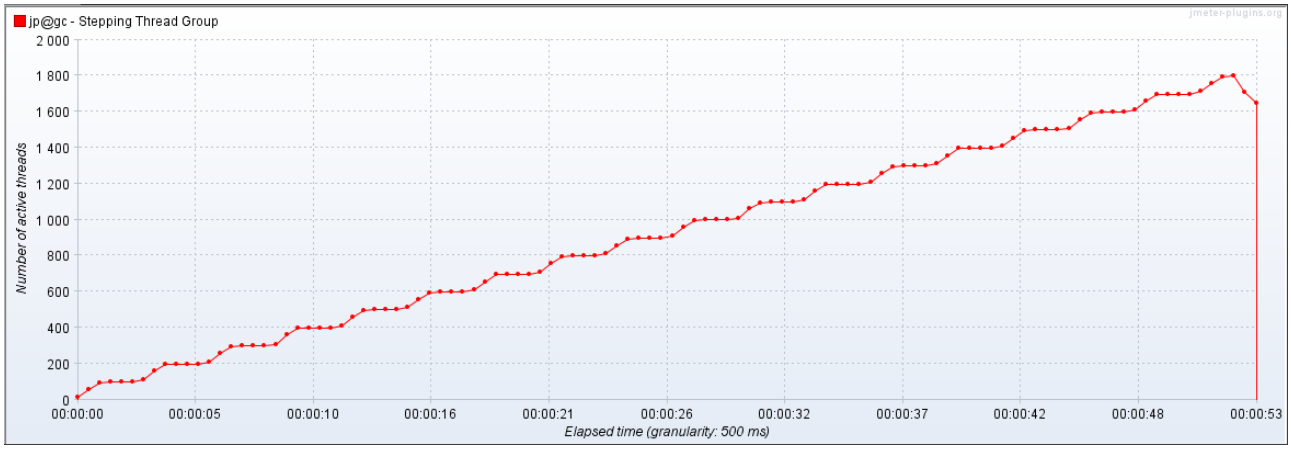

Рис. 4.75 Підключення користувачів до додатку

Фронтенд витримав навантаження в 1800 користувачів, але веб додаток почав відповідати response error.

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Received KB/...	Sent KB/sec
GET Request	9707	4	2	6	10	19	0	1123	0,00%	181,0/sec	140,52	22,10
TOTAL	9707	4	2	6	10	19	0	1123	0,00%	181,0/sec	140,52	22,10

Рис. 4.76 Середні результати тестування для GET запитів

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Received KB/...	Sent KB/sec
POST Request	9700	6992	5504	9786	10821	45310	3	46961	13,16%	103,4/sec	47,60	18,32
TOTAL	9700	6992	5504	9786	10821	45310	3	46961	13,16%	103,4/sec	47,60	18,32

Рис. 4.77 Середні результати тестування POST запитів

Таким чином, додаток розгорнутий на Kubernetes кластері з використанням pods матиме наступні результати.

Таблиця 4.3

Показники додатку побудованого на Kubernetes кластері з використанням deployments та loadbalancers

Samples	Response Time, ms	
	GET	POST
500	2	813
1500	1	1663
3000	1	1867
4500	1	1853

Середня затримка GET запиту при роботі з 500 користувачів 1.25 мс, при тестуванні на стійкість до навантажень 4 мс.

Середня затримка на POST запити 1549 мс, при тестування на стійкість до навантажень 6992 мс.

Загалом при тестуванні на максимальну кількість користувачів котрі роблять POST та GET запити додаток зміг обслуговувати одночасно 1800 користувачів.

Також pod в кластері kubernetes має властивість заново створюватись після того, як він вимкнувся або крашнувся. На рисунку 4.78 зображено затримку відповідей на POST запити, response latencies = 0 означає, що под котрий крашнувся не відповідає, про цьому через 15 секунд він знову працює, але оскільки навантаження було надвисоким для такої кількості системних ресурсів він не може справитися з кількістю запитів і знову крашится.

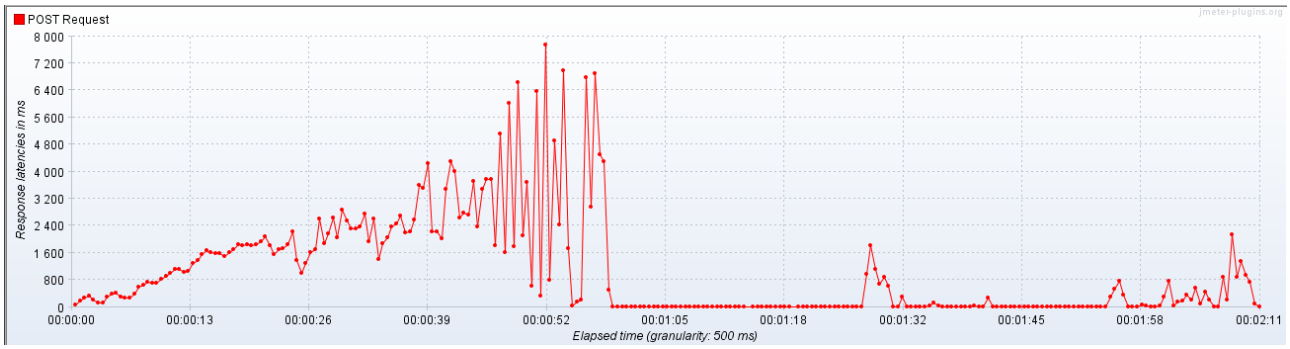

Рис. 4.78 Графік росту затримки з збільшенням кількості користувачів

Висновки:

В даному розділі було описано експерементальний додаток та принцип його роботи. Було описано налаштування тестового оточення, збірку білдів та розміщення мікросервісного додатку на кластері Kubernetes. Також було налаштовано інструмент для здійснення тестування на стійкість до навантажень та зняття показників системи. Після тестування, маємо наступні показники кожної з 3 систем:

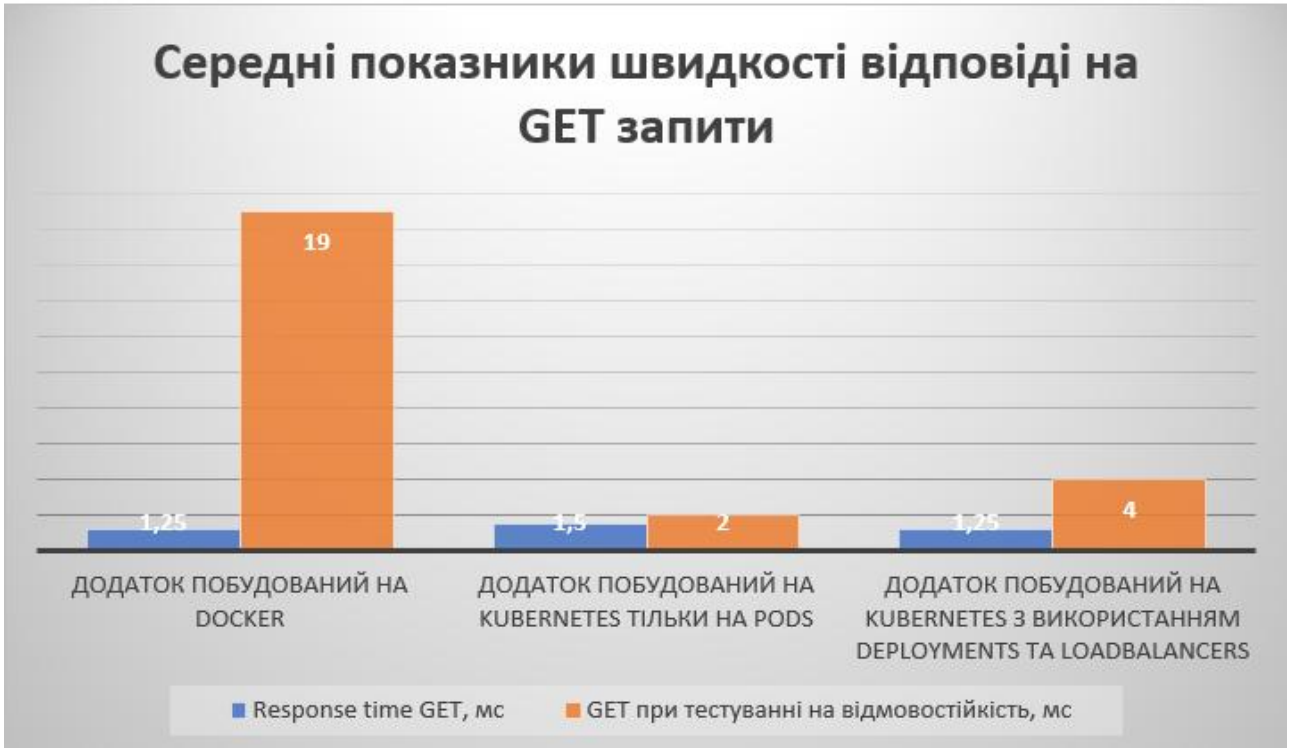

Рис. 4.79 Середні показники швидкості відповіді на GET запити

Рис. 4.80 Середні показники швидкості відповіді на POST запити

За результатами проведеного тестування, можна побачити, що при роботі в звичайних умовах response time значно перевищує нормативне значення при всіх 3 методах розгортання мікросервісного додатку. Оскільки, нормальним часом відповіді для frontend > 200мс[11].

При роботі з великими навантаженнями, можна побачити, як виріс час відповіді на GET запити в додатку розгорнутому за допомогою Docker контейнерів. Це відбувається тому що не має ізоляції системних ресурсів, таким чином контейнери з Java мікросервісом потребують більшої кількості оперативної пам’яті, цим самим забираючи ресурси у nginx серверу.

При POST запитах можна побачити, що час відповіді значно вище, оскільки необхідно опрацьовувати кожен з запитів. При всіх 3 методах розгортання додатку використовувалась однакова кількість системних ресурсів, а саме 4гб оперативної пам’яті та 4 процесори. Але додаток побудований на Docker контейнерах та кластері Kubernetes з використанням тільки pods витримали навантаження лише в 1100 користувачів. При цьому додаток побудований на Kubernetes кластері з використанням deployments та loadbalancers (при використанні однакової кількості системних ресурсів) зміг показати себе краще на 63.63% по одночасному підключенні користувачів, оскільки такий тип розгортання (за допомогою використання loadbalancers) ріномірно ділить запити між pods, що недає перевантажити їх. Також при використанні хмарного провайдера (Amazo, GCP, Microsoft Azure) можна налаштувати автоматичне створення pods в разі необхідності, що дасть змогу обслуговувати більшу кількість клієнтів (але в нашому випадку в цьому немає необхідності, оскільки ми маємо сталі системні ресурси). При цьому response time на POST запити значно збільшився, такий метод розгортання показав значно вижчі показники з точки зору відмовостійкості.

Також при використання такого методу, в разі збою на якійсь з pod вона буде автоматично розгорнута заново, що позитивно впливає на uptime сервісу в цілому. Використання deployments дає змогу неперервно оновлювати додаток новими релізами, а також при збоях повертатися до попередньої версії даже швидко, що також підвищує uptime додатку.

Таким, чином можна зробити виснок, що для побудови сервісу з високим показником uptime та відмовостійкості необхідно:

- Створити Docker контейнери з середовищем, яке відповідає наступним характеристикам: кількість зовнішніх залежностей, котрі несуть

функції generic (загального або обслуговуючого призначення) повинна бути нижча співвідношенню (4.1):

$$R = \frac{K_{generic}}{N_{common}} < 0.1 \quad (4.1)$$

Дотримання співвідношення (4.1) дає змогу ефективніше використовувати системні ресурси, оскільки вони будуть виконувати тільки корисну роботу. При цьому ефективність використання системних ресурсів, таких як операційна пам'ять буде визначено (4.2):

$$K_{mem} = \frac{\sum_1^{K_{generic}} M_{generic}}{\sum_1^{K_{generic}} M_{common}} \quad (4.2)$$

- Кількість системних ресурсів, необхідних для коректної роботи сервісів (так званий розмір footprint), повинна відповідати в півтора рази вище від базового (4.3):

$$F_{footprint} = \left(F_{core} + F_{system} + F_{lib}\right) * 1.5 \quad (4.3)$$

Перед тим як розгорнути додаток, провести тестування кожного з мікросервісів окремо та визначити розмір системних ресурів за допомогою відповідного програмного комплексу, наприклад, Apache JMeter, який використаний в даній роботі.

- Ізолювати використання системних ресурсів для кожного з pod (тоб-то заздалегіть вказати скільки ОЗУ може використовувати кожен тип pod). Краще встановити мінімальне та максимальне значення.
- Необхідно використовувати deployment для кожного мікросервісу, а не тільки pods, це дасть змогу контролювати кількість pod для кожного з мікросервісу, а також спростить їх контроль.
- В кожний з Deployments вказати стратегію типу Rolling update та правильно налаштувати значення maxUnavailable та maxSurge в залежності від кількості реплік. (це дасть змогу оновлювати додаток без перезавантаження, що підвищує uptime сервісу)

- Використовувати loadbalncers для того щоб розподіляти навантаження між кожним pod в deployments, що зменшить вірогідність збою на pods. При цьому коефіцієнт завантаженності балансера визначається (4.4):

$$p = \frac{\lambda}{m*\mu} < 1 \quad (4.4)$$

де:

λ – інтенсивність надходження заявок в одиницю часу

μ – інтенсивність обслуговування заявок в одиницю часу

m – кількість таких сервісів від load balancer

Відповідно, значення виразу (4.4) не повинно перевищувати 1, що визначає нормальні умови експлутації.

РОЗДІЛ 5
РОЗРОБЛЕННЯ СТАРТАП ПРОЕКТУ

В даному розділі викладено маркетинговий аналіз перспектив реалізації високодоступних систем в мережі кластера віртуалізації Kubernates, а також оцінено можливості її ринкового впровадження.

5.1 Опис ідеї проекту

З розвитком індустрії інформаційних технологій попит на високодоступні, відмовостійкі та ефективні сервіси постійно зростає. Велика кількість користувачів мережі Інтернет змушує впроваджувати та розробляти нові методики для побудови складних інформаційних систем. Потреба в цьому може бути знайдена в будь-яких галузях — від розваг до наукових досліджень. Однак є кілька проблем, які перешкоджають можливостям мережевих додатків. Одним з них є використання застарілих технологій та архітектурних рішень.

Сервіси побудовані на застарілих методиках та технологіях дуже складно масштабувати, що не задовольняє потреби сучасного інформаційного суспільства.

Ідея проекту полягає у надаванні послуг таких як: розгортання додатків на будь-якому дата центрі (AWS, GCP, Microsoft Azure) або локальні сервіси клієнтів, його підтримка та аудит (ефективності використання системних ресурсів та відмовостійкості сервісу).

Таблиця 5.1

Опис ідеї стартап - проекту

Зміст ідеї	**Напрямки застосування**	**Вигоди для користувача**
Підвищити доступність сервісів за рахунок побудови їх в вигляді мікросервісної архітектури та запуск сервісу в кластері віртуалізації Kubernates	1. Телекомунікації	Високодоступні та надійні у використанні сервіси.

5.2 Технологічний аудит ідеї проекту

Проводиться аудит технологій, за допомогою якої може бути реалізована ідея проекту, тобто технології створення послуги.

Таблиця 5.2

Результати огляду основних видів технологій

Ідея проекту	Технології реалізації	Наявність технологій	Доступність технологій
Побудова сервісу на базі мікросервісної архітектури з використанням кластеру віртуалізації Kubernates	Модифікація проекту відповідно до мікросервісної архітектури	Наявні	Доступні
	Контейнеризація окремих мікросервісів		
	Запуск сервісу під управлінням кластеру Kubernates		

У таблиці Таблиця 5.2 результати огляду основних видів технологій, які можуть або будуть використовуватися з метою розвернення сервісів з мікросервісною архітектурою, їх підтримки та надання пов’язаних послуг. В результаті було обрано одну головну технологію (кластер віртуалізації Kubernates) та декілька допоміжних (Ansible, Jenkins, Terraform, Docker).

Обрана технологія реалізації ідеї проекту: Створення інстансів у відповідності до розміщення потенційного клієнту для сервісу та розвернення його в кластері Kubernetes, автоматизація виходу нових релізів та аудит уже існуючих розгортань з використанням рекомендацій описаних в магістерські дисертації. При необхідності надавання рекомендацій що до модифікація коду проекту відповідно до стандартів мікросервісної архітектури, після чого

контейнеризація окремих мікросервісів та запуск проекту з використанням кластеру K8s.

5.3 Аналіз ринкових можливостей запуску стартап-проекту

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

Спочатку було проведено аналіз попиту: наявність попиту, обсяг, динаміка розвитку ринку (табл. 5.3).

Таблиця 5.3

Попередня характеристика потенційного ринку

	Показники стану ринку (найменування)	**Характеристика**
1	Кількість головних гравців, од	Немає підтвердженої інформації
2	Загальний обсяг продаж, грн/ум.од	Немає підтвердженої інформації
3	Динаміка ринку (якісна оцінка)	Зростає
4	Наявність обмежень для входу (вказати характер обмежень)	Структурні обмеження
5	Специфічні вимоги до стандартизації та сертифікації	Немає
6	Середня норма рентабельності в галузі (або по ринку), %	Немає підтвердженої інформації

За результатами аналізу таблиці було зроблено висновок, що ринок є привабливим для входження. Надалі були визначені потенційні групи клієнтів, їх характеристики, та сформовано орієнтовний перелік вимог до товару для кожної групи.

Таблиця 5.4

Характеристика потенційних клієнтів стартап-проекту

Потреба, що формує ринок	Цільова аудиторія (цільові сегменти ринку)	Відмінності у поведінці різних потенційних цільових груп клієнтів	Вимоги споживачів до товару
Стійкі до навантажень, високодоступні сервіси	Приватні підприємці з власним продуктом	Очікується висока зацікавленість в стартапі саме людей, які пов’язані з web-сервісами	Висока доступність, стійкість до навантажень та великої кількості запитів.

Після визначення потенційних груп клієнтів було проведено аналіз ринкового середовища: складено таблиці факторів, що сприяють ринковому впровадженню проекту, та факторів, що йому перешкоджають (табл. 5.5-5.6)

Таблиця 5.5

Фактори загроз

	Фактор	Зміст загрози	Можлива реакція компанії
1	Конкуренція	Вихід на ринок одного з гігантів сумісних областей з комплексним програмним рішенням, що міститиме у собі аналог нашого продукту.	1. Передбачити додаткові переваги власного проекту для того, щоб повідомити про них саме після виходу міжнародної компанії на ринок. 2. Обрати нову цільову аудиторію і зосередитися на ній. 3. Об’єднання з компанією- конкурентом.
2	Економічний	Подорожчення вартості та обслуговування обладнання, необхідного для роботи системи.	Оптимізація програмного продукту,дляможливості його запуску на більш бюджетних пристроях

В таблиці вище були наведені фактори загроз та способи зменшення ризиків. Найбільшою загрозою є конкуренція. Для боротьби з конкуренцією нам необхідно передбачити найкращий набір функціоналу. Також необхідно передбачити можливість додавання нового функціоналу, або зробити фокус на більш вузькій цільовій аудиторій, надаючи їм більше уваги. Також, при виході на ринок дуже потужного конкурента, можна розглянути можливість об'єднання або поглинання з метою збереження вкладених в даний проект коштів.

Таблиця 5.6

Фактори можливостей компанії

	Фактор	**Зміст можливості**	**Можлива реакція компанії**
1	Відсутність вітчизняних компаній з аналогічними послугами	Вільний ринок	Можливість швидкого розвитку
2	Попит	Більш широке розповсюдження технології	Постійна підтримка продукту
3	Науково-технічний	Тенденція до випуску покращеного спеціалізованого обладнання та розробка більш ефективних алгоритмів	Адаптація існуючого рішення і алгоритмів під нову технологію

В цій таблиці було розглянуто фактори можливостей. Найбільш цікавим для нас звичайно є відсутність вітчизняних компаній котрі надають аналогічні послуги, адже це автоматично робить наш продукт унікальним на внутрішньому ринку.

Після цього був проведений аналіз пропозиції: визначено загальні риси конкуренції на ринку (табл. 5.7).

Таблиця 5.7

Ступеневий аналіз конкуренції на ринку

Особливості конкурентного середовища	**В чому проявляється дана характеристика**	**Вплив на діяльність підприємства**
За рівнем конкурентної боротьби: міжнародний.	Конкуренти з різних країн світу.	Розвиток на українських підприємствах та вихід на ринок.
За галузевою ознакою: внутрішньогалузева.	Продукт використовується лише всередині даної галузі.	Постійне вдосконалення продукту.
Конкуренція за видами товарів: товарно-видова конкуренція.	Конкуренція між товарами одного виду.	Створення кращої і якіснішої продукції.
За характером конкурентних переваг: нецінова.	Збільшення функціональності в межах однієї системи та збільшення якості її роботи.	Зниження ціни на продукт та підтримка його якості

В цій таблиці наведено аналіз конкуренції на ринку. Визначено, що наш сервіс працює в середовищі нецінової, товарно-видової конкуренції. Конкуренція відбувається не лише в середині країни, а й на міжнародному ринку. В таблиці 4.8 наведено SWOT-аналіз стартап-проекту, базуючись на характеристиках проекту, що були надані в попередніх таблицях.

Таблиця 5.8

SWOT-аналіз стартап-проекту

Сильні сторони: 1. Підвищення якості та доступності сервісів 2. Послуга відповідає потребам споживачів	Слабкі сторони 1. Низька репутація підприємства на початку впровадження проекту. 2. Велика вартість модифікації масштабних проектів
Можливості: 1. Вихід на міжнародний ринок.	Загрози: 1. Зниження доходів потенційних клієнтів. 2. Вихід на ринок нових технологій

В наступній таблиці буде описано базові стратегії розвитку.

Таблиця 5.9

Визначення базової стратегії розвитку

Обрана альтернатива розвитку	**Стратегія охоплення ринку**	**Ключові конкурентоспроможні позиції**	**Базова стратегія розвитку**
Проведення конференції для зацікавлених користувачів	Ексклюзивний розподіл	Висока надійність та стійкість до високих навантажень	Стратегія диференціації

Для обраної альтернативи розвитку проекту було обрано не ексклюзивний розподіл, а стратегію диференціації, як базову стратегію розвитку. Тому що, така стратегія полягає в орієнтації діяльності підприємства на надавання унікальних послуг, які визначаються важливими достатньою кількістю споживачів.

Таблиця 5.10

Визначення базової стратегії конкурентної поведінки

Чи є проект першопрохідцем на ринку	Чи буде компанія шукати нових клієнтів, або забирати існуючих у конкурентів	Чи буде компанія копіювати основні характеристики	Стратегія конкурентної поведінки
Проект не є першопрохідцем на міжнародному ринку	Компаня буде забирати існуючих та шукати нових клієнтів	Буде	Стратегія лідера

В таблиці 5.10 було визначено базову стратегію конкурентної поведінки. Стартап-проект не є першопрохідцем на міжнародному ринку , але є всі передумови для заняття свої ніші на внутрішньому ринку. Даний проект буде намагатись перетягнути частину користувачів від конкурентів та з збільшенням клієнтської бази масштабуватись. Також планується взяти найкращі практики з компаній конкурентів за для покращення послуг котрі буде надавати стартап-проект.

Таблиця 5.11

Визначення стратегії позиціонування

Вимоги до товару цільової аудиторії	Базова стратегія розвитку	Ключові конкурентоспроможні позиції власного стартап-проекту	Вибір асоціацій, які мають сформувати комплексну позицію власного проекту
Висока доступність та стійкість до навантажень	Створення сервісів високої доступності	Використання рекомендацій розроблених в цій роботі	Надійність, стійкість до навантажень якість, ефективність

В таблиці вище описано старатегію позиціонування даного стартап-проекту. Описано основні вимоги цільової аудиторії до товару. Визначено базову стратегію розвитку. Сформовано перелік ключових позицій конкурентоспроможності даного

5.4 Розроблення маркетингової програми стартап – проекту

Маркетингова програма стартап-проекту розроблена на основі потреб, заснованих на концепції потенційного продукту.

Першим кроком є формування маркетингової концепції послуг, які отримає споживач. Для цього у (табл. 5.12) потрібно підсумувати результати попереднього аналізу конкурентоспроможності послуг.

Таблиця 5.12

Визначення ключових переваг концепції потенційнмх послуг

	Потреба	**Вигода, яку пропонує товар**	**Ключові переваги перед конкурентом (існуючі або такі, що потрібно створити)**
1	Доступність	Найвищий рівень	Автоматичне підстроювання параметрів Системи можливість розгортання сервісу на їх обладнанні
2	Стійкість до навантажень	Найвищий рівень стійкості	Побудова сервісів З використанням рекомендацій Розроблених в цій роботі

Таблиця 5.14

Концепція маркетингових комунікацій

Специфіка поведінки цільових клієнтів	Канали комунікацій	Ключові позиції, обрані для позиціонування	Завдання рекламного повідомлення
Клієнти дізнаються про нові продукти з реклами в інтернеті або по рекомендаціям інших людей.	Інтернет	Висока доступність, стійкість до великої кількості запитів	Демо ролик з використанням, реклами.

Висновки:

В даному розділі було проведено аналіз компанії яка надає послуги та рекомендації з аудиту уже існуючих сервісів клієнтів та побудови сервісів на кластері Kubernates, а також підвищення ефективності та відмовостійкості уже існуючого у якості стартап проекту.

Було проведено аналіз ризиків та можливостей які можуть виникнути. Основними загрозами, очікувано, виявились конкуренція та подорожчення вартості на обслуговування обладнання та програмного забезпечення, необхідного для роботи системи. Найбільш вдалою можливістю для нас, звичайно ж, є відсутність вітчизняних конкурентів, адже це автоматично робить наш стартап-проект унікальним на внутрішньому ринку.

Для впровадження ринкової реалізації проекту слід обрати альтернативу, яка передбачає проведення конференції для зацікавлених користувачів, а потім якісну рекламу та PR, сконцентровану навколо позитивних характеристиках нашого продукту.

З огляду на проведений аналіз, можна чітко сказати, що подальша імплементація проекту є доцільною, адже він може знайти свою цільову аудиторію та зайняти місце на ринку.

ЗАГАЛЬНІ ВИСНОВКИ ПО РОБОТІ

Отже, в даній магістерський дисертації була розглянута технологія Kubernetes. В теоретичній частині описані особливості цієї технології, її переваги та недоліки. Також розглянуті передумови розвитку цієї технології та порівняння її з конкурентами. Було встановлено, що Kubernetes потужний інструмент для розгортаняя додатків написаних на основі мікросервісної архітектури, а також підходить для розгортання Big Data систем, може автоматично моніторити стан кластеру та відновлювати пошкоджені вузли.

В практичній частині було розгорнуто стенд для тестування, котрий включає в себе три типи розгортання веб додатку:

- Розгорунутого на Docker контейнерах
- В Kubernetes кластері з використанням pods
- В Kubernetes кластері з використанням deployments та loadbalncers

Тестування проводилось за допомогою інструменту для тестування Apache JMeter, а також описано його налаштування.

За результатами проведеного тестування, було виявлено, що при роботі в звичайних умовах response time має високі показники при всіх 3 методах розгортання мікросервісного додатку.

При роботі з великими навантаженнями, встановлено, що виріс час відповіді на GET запити в додатку розгорнутому за допомогою Docker контейнерів та описано чому це відбувається.

При POST запитах можна побачити, що час відповіді значно вище, оскільки необхідно опрацьовувати кожен з запитів. При всіх 3 методах розгортання додатку використовувалась однакова кількість системних ресурсів, а саме 4гб оперативної пам’яті та 4 процесори. Але додаток побудований на Docker контейнерах та кластері Kubernetes з використанням тільки pods витримали навантаження лише в 1100 користувачів. При цьому додаток побудований на Kubernetes кластері з використанням deployments та loadbalancers (при використанні однакової кількості системних ресурсів) зміг

показати себе краще на 63.(63)% по одночасному підключенні користувачів. При цьому response time на POST запити значно збільшився, такий метод розгортання показав значно вижчі показники з точки зору відмовостійкості та ефективності використання системних ресурсів. Після проведеного тестування були розроблені рекомендації, щодо управління навантаженням в мережі кластеру Kubernetes.

Робота має велику практичну цінність, оскільки значну її частину становить саме програмне вирішення проблем з ефективністю та відмовостійкістю додатків, а не економічне. Рішення полягає у дотриманні рекомендацій котрі розроблені в даній магістерській роботі.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Hands-On Microservices with Kubernetes: Build, deploy, and manage scalable microservices on Kubernetes 1st Edition, by Gigi Sayfan (2019) p.45
2. [Електронний ресурс] – Режим доступу: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/
3. [Електронний ресурс] – Режим доступу: https://mcs.mail.ru/blog/put-k-kubernetes-i-ego-preimushchestva-dlya-razrabotki
4. [Електронний ресурс] – Режим доступу: https://ela.kpi.ua/bitstream/123456789/29960/1/Khmarni_ta_gridtekhnolohii_Konspekt_lektsii1.pdf
5. The Docker Book: Containerization is the new virtualization Kindle Edition by James Turnbull (2014) pp.32-35
6. Kubernetes and Docker - An Enterprise Guide: Effectively containerize applications, integrate enterprise systems, and scale applications in your enterprise 1st Edition, Kindle Edition by Scott Surovich (2020) pp. 110-128
7. Using Docker, by (author) Adrian Mouat (2016)
8. [Електронний ресурс] – Режим доступу: https://kubernetes.io/uk/docs/_print/
9. [Електронний ресурс] – Режим доступу: https://uk.wikipedia.org/wiki/YAML
10. [Електронний ресурс] – Режим доступу: https://www.mirantis.com/blog/introduction-to-yaml-creating-a-kubernetes-deployment/
11. [Електронний ресурс] – Режим доступу: https://seoportal.net/baza/skorost-zagruzki/vremya-otveta-servera
12. [Електронний ресурс] – Режим доступу: https://uk.wikipedia.org/wiki/JMeter
13. [Електронний ресурс] – Режим доступу:

https://github.com/rinormaloku/k8s-mastery