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

Факультет Інформатики та обчислювальної техніки
Кафедра Автоматики та управління в технічних системах

«До захисту допущено»
Завідувач кафедрою
_____ Ролік О. І.
(підпис) (ініціали, прізвище)
«__»_____ 2019 р.

Магістерська дисертація

зі спеціальності 126 Інформаційні системи та технології

на тему: «Система аналізу продуктивності Kubernetes кластера»

Виконав: студент 6-го курсу, групи ІА-82мп
(шифр групи)

Білан Ольга Олександрівна
(прізвище, ім'я, по батькові) (підпис)

Науковий керівник к.т.н, доцент каф. АУТС, Букасов М. М.
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Рецензент к.т.н, доцент каф. ТК, Мелкумян К. Ю.
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент _____
(підпис)

Київ – 2019 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра автоматики та управління в технічних системах

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

Спеціальність – 126 Інформаційні системи та технології

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ С. Ф. Теленик

«___»_____20__ р.

ЗАВДАННЯ
на магістерську дисертацію студенту

Білан Ользі Олександрівні

1. Тема дисертації «Система аналізу продуктивності Kubernetes кластера», науковий керівник дисертації Букасов Максим Михайлович, доцент, к.т.н., затверджені наказом по університету від «___»_____ 20__ р. №_____

2. Термін подання студентом дисертації: _____

3. Об’єкт дослідження – продуктивність Kubernetes кластера.

4. Предмет дослідження – система аналізу продуктивності.

5. Перелік завдань, які потрібно розробити: аналіз існуючих рішень; аргументація вибору інструментів для побудови системи; дослідження теоретичного матеріалу з обраної теми; розробка системи для визначення продуктивності Kubernetes кластера; автоматизація розгортання системи; порівняння результатів системи для обраних хмарних провайдерів; порівняння з існуючими рішеннями.

6. Орієнтовний перелік графічного (ілюстративного) матеріалу: діаграма компонентів, діаграма використання, діаграма розгортання системи, діаграма послідовності, схема тестових кластерів у Google Cloud Platform та Amazon Web Services, структурна схема системи, діаграма використання, Helm для автоматизації, результати аналізу продуктивності кластерів за допомогою системи, діаграма компонентів Kubernetes кластера у хмарі.

7. Перелік публікацій: 1) «Тестування продуктивності Kubernetes кластера» – III Міжнародній науково-практичній конференції «Пріоритети сучасної науки».

– К.: МЦНіД, 2019. – Частина 2. – С.24. (наукове видання);
.

8. Дата видачі завдання: _____

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1	Проведення аналізу існуючих рішень	05.09.19-20.09.19	
2	Вибір інструментів для побудови системи	21.09.19-10.10.19	
3	Дослідження теоретичного матеріалу з обраної теми	11.10.19-01.11.19	
4	Розробка системи для визначення продуктивності кластера	02.11.19-17.11.19	
5	Автоматизація розгортання системи	18.11.19-25.11.19	
6	Порівняння результатів системи для обраних хмарних провайдерів	26.11.19-30.11.19	
7	Порівняння з існуючими рішеннями	01.12.19-03.12.19	

Студент О. О. Білан

Науковий керівник дисертації М. М. Букасов

РЕФЕРАТ

Магістерська дисертація на тему “Методи виявлення образ в коротких текстах”: 106 ст., 22 рис., 23 табл., 9 додатків, 27 джерел.

Об’єктом дослідження виступає продуктивність Kubernetes кластера. Предметом дослідження є система, що аналізує продуктивність кластера.

Метою дисертації являється створення системи для аналізу продуктивності Kubernetes кластера, порівняння різних хмарних середовищ для Kubernetes кластера на основі системи. У ній розглянуто питання тестування навантаженням кластера, дослідженя Kubernetes як оркестратора контейнерів. На практиці доведена дієздатність системи, що дозволяє адаптувати параметри для кожної інфраструктури та отримати інформативний звіт.

Розроблено мікросервісний застосунок на GO з автоматизацією запуска у різних середовищах всередині Kubernetes кластера.

Отримана базова система має потенціал для розвитку та може застосовуватись у сфері проектування систем, мікросервісної архітектури та навчанні. На основі результатів було розроблено базовий план для стартап-проекту.

Прогнозні припущення про розвиток дослідження – збільшення параметрів для конфігурації системи.

КУБЕРНЕТИС, МІКРОСЕРВІСНА АРХІТЕКТУРА, ТЕСТУВАННЯ НАВАНТАЖЕННЯМ, ХМАРНІ ПРОВАЙДЕРИ, КОНТЕЙНЕР

ABSTRACT

Master's dissertation on the theme "Methods of revealing image in short texts": 106 pages, 22 figures, 23 tables, 9 appendices, 27 sources.

The object of research is the Kubernetes cluster performance. The subject of the study is the analysis sytem of Kubernetes cluster performance.

The purpose of the dissertation is to create a system for analyzing the performance of a Kubernetes cluster, comparing different cloud environments for a Kubernetes cluster based on the system. It addresses the issue of cluster load testing, exploring Kubernetes as a container orchestrator. In practice, the system is capable of adapting the parameters for each infrastructure and receiving an informative report.

A microservice application was developed on GOlang with automation of launching in different environments within the Kubernetes cluster.

The resulting basic system has the potential for development and can be used in the field of systems design, microservice architecture and training. Based on the results, a basic plan for the startup project was developed.

Predictive assumptions about the development of the study - increasing the parameters for system configuration.

KUBERNETES, MICROSERVICE ARCHITECTURE, LOAD TESTING, CLOUD PROVIDERS, CONTAINER

Зміст

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ, ОДИНИЦЬ І ТЕРМІНІВ

API – Aplication Programming Interface

AWS – Amazon Web Services

CCM – Cloud Controller Manager

ELB – Elastic Load Balancer

GCE – Google Container Engine

GCP – Google Cloud Platform

GKE – Google Kubernetes Engine

HTTPS – HyperText Transfer Protocol Secure

k8s – Kubernetes

LB – Load Balancer

LXC – Linux Containers

URL – Uniform Resource Locator

Под – абстракція Kubernetes, що представляє один або групу контейнерів

ВСТУП

Швидкий розвиток технологій і величезна кількість користувачів мережі Інтернет змушують впроваджувати та розробляти нові методики щодо побудови складних інформаційних систем. Залишаючись неділимим в сучасному світі, важко змінюватись разом з потребами примхливого ринку. Тому требя поділяти сервіси систем за різними потребами та функціями. Створення програмного забезпечення у вигляді мікросервісів стає на передній план, порівнюючи з монолітом.

Мікросервісна архітектура – це рішення, яке може оптимізувати та полегшити процес розробки. Наразі найбільш популярною системою є Kubernetes. Kubernetes – це система з організацією контейнерів з відкритим кодом для автоматизації розгортання, масштабування та управління. Kubernetes це складний продукт відносно конкурентних, побудова кластера якого потребує хороших знань не тільки самої системи, а й інших фундаментальних речей, як операційні системи, тестування, проектування архітектури застосунків. Проте як зрозуміти, що кластер побудований буде відповідати прогнозованому навантаженню у майбутньому? Для цього потрібно мати систему для тестування та аналізу.

Дана магістерська дисертація присвячена створенню системи для аналізу продуктивності Kubernetes кластера. Її доцільність була обгрунтована прикладною задачею відносно планування інфраструктури кластера.

Об’єктом дослідження виступає продуктивність Kubernetes кластера. Предметом дослідження являються система аналізу продуктивності, що застосовується в даній сфері.

Зв’язок роботи з науковими програмами, планами, темами. Тематика роботи може бути включена в науково-технічні плани кафедри автоматики та управління в технічних системах Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського» оскільки Kubernetes є передовою технологією при побудові сучасних систем.

Метою розробки та аналізу, які представлені в даній магістерській дисертації є побудова системи для оцінки продуктивності Kubernetes кластера.

Основною задачею дослідження являється визначення продуктивності Kubernetes кластера у різних конфігураціях. Проміжними задачами можна визначити порівняльний аналіз результатів відносно різних конфігурацій кластера та огляд існуючих рішень, підходів для визначення продуктивності, а також вибір засобів для побудови системи. В даному дослідженні було використано методи емпіричного дослідження.

При створенні програмного продукту пройдено повний цикл його написання – від постановки завдання, написання технічного завдання і вимог до продукту, до написання програми та тестування.

Результати дослідження апробовані і опубліковані у вигляді тези «Тестування продуктивності Kubernetes кластера» на III Міжнародній науково-практичній конференції «Пріоритети сучасної науки» (додаток А).

1 МІКРОСЕРВІСНА АРХІТЕКТУРА ТА KUBERNETES

Очевидно, що тема даної роботи відноситься до розділу побудови інфраструктури та її тестування. Безпосередньо побудова мікросервісної архітектури є основою Kubernetes кластера та має великий спектр теоретичних відомостей.

У цьому розділі буде розглянуто теорії мікросервісної архітектури та Kubernetes кластера.

1.1 Мікросервісна архітектура

1.1.1 Актуальність роботи

З кожним днем технології швидко ростуть. Логіка програми не тільки заснована на кількості користувачів, але заздалегідь визначений спосіб і полегшує вашу розробку та розгортання. Великі корпоративні проекти підтримуються, а нові вимагають оптимальної економії архітектури. Мікросервісна архітектура – важається одним із найважливіших кроків у плануванні ресурсів та розвитку сучасної архітектури. Архітектура мікросервісу може дозволити додатку зареєструвати окрему службу обслуговування, з урахуванням існуючих тенденцій. Оскільки фундаментальним поняттям є Docker треба розглянути його розвиток. Docker – інструментарій для управління ізольованими Linux-контейнерами. Docker доповнює інструментарій Linux Containers (LXC) більш високорівневим API, що дозволяє керувати контейнерами на рівні ізоляції окремих процесів. Зокрема, Docker дозволяє не переймаючись вмістом контейнера запускати довільні процеси в режимі ізоляції і потім переносити і клонувати сформовані для даних процесів контейнери на інші сервери, беручи на себе всю роботу зі створення, обслуговування і підтримки контейнерів.

Datadog – хмарний сервіс моніторингу, який інтегрує дані з серверів, баз даних, служб та інструментів для забезпечення єдиного перегляду всього стека. (рис 1.1).

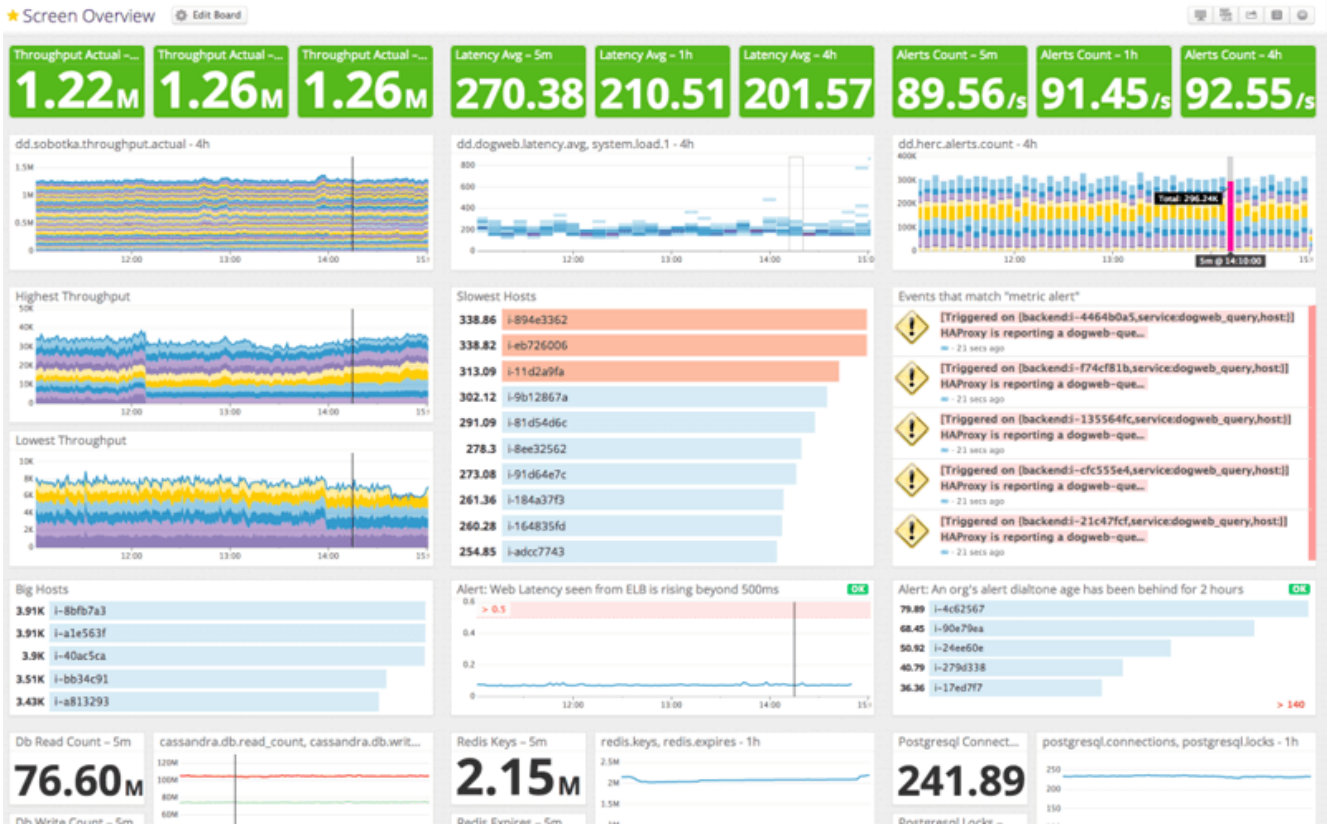

Рисунок 1.1 – Приклад панелі Datadog

Ці можливості надаються на платформі аналізу даних на основі SaaS. За даними ресурсу Datadog [1], з літа 2015 року 3 відсотки хокерів Docker до січня 2017 року цей показник збільшився до 15 відсотків. Ці цифри є досить вражаючими: за 1,5 року кількість господарів зросла в 5 разів (мал. 1.2). На початок березня 2016 року 13,6% клієнтів Datadog переїхали в Докер. Через рік ця цифра зросла до 18,8 відсотка. Це майже 40 відсотків зростання ринку за 12-місячний період.

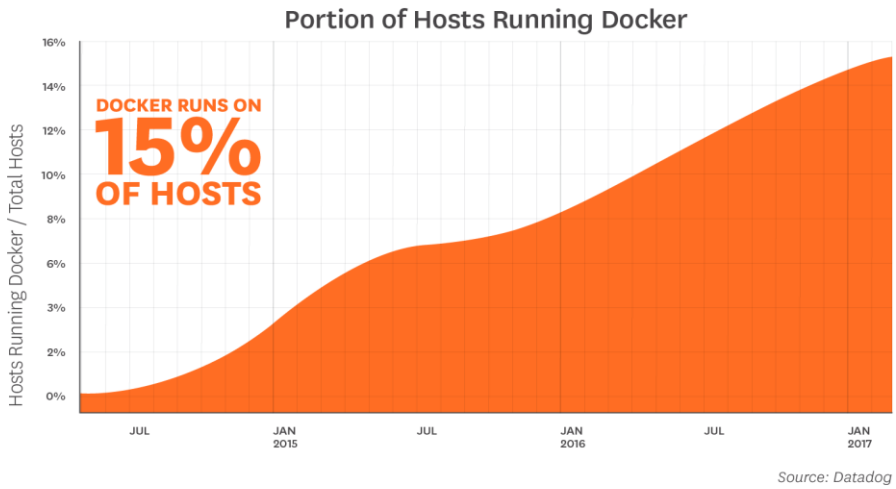

Рисунок 1.2 – Графік зміни кількості хостів на Docker

Найбільш технології на Docker це (рис 1.3):

– NGINX: Docker використовується, щоб містити багато серверів HTTP. NGINX – давній претендент на цей список;

– Redis: Цей популярний сховище даних ключових даних часто використовується як база пам'яті, черга повідомлень або кеш;

– Elasticsearch: повнотекстовий пошук продовжує збільшувати популярність;

– Registry: 18% компаній Docker використовують Registry, програму зберігання та розповсюдження Docker;

– Postgres: все більш популярна реляційна база даних з відкритим кодом видаляє MySQL вперше в цьому рейтингу;

– MySQL: Найбільш широко використовувана база даних з відкритим кодом у світі продовжує знаходити використання в інфраструктурі Docker. Додаючи номери MySQL та Postgres, здається, що використання Docker для запуску реляційних баз даних несподівано поширене;

– etcd: Розподілений магазин зберігання ключів використовується для забезпечення послідовної конфігурації через кластер Docker;

– Fluentd: це "уніфікований шаблон реєстрації з відкритим кодом", призначений для відокремлення джерел даних від фонових сховищ.

– MongoDB: розповсюджений Datastore NoSQL;

– RabbitMQ: агент повідомлень з відкритим кодом знаходить багато користі в середовищах Docker.

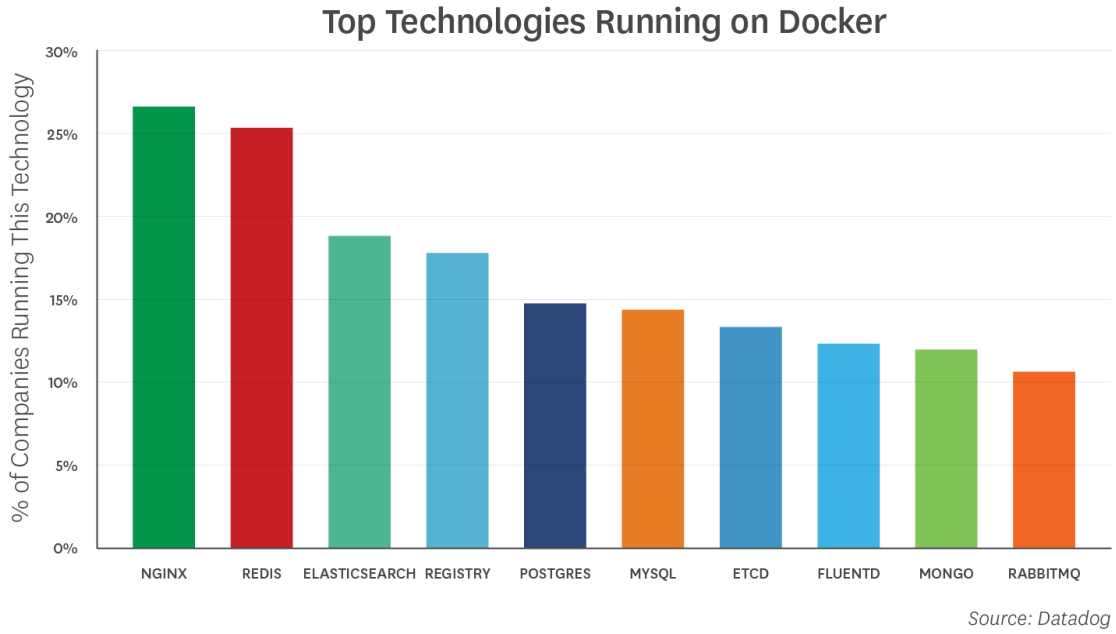

Рисунок 1.3 – Список найпоширеніших технологій реалізованих на Docker

На діаграмі видно, що використання архітектури мікросервісу при розробці сучасної архітектури додатків є дуже популярною сферою.

1.1.2 Визначення поняття мікросервісної архітектури

Мікросервіси: архітектурний стиль, за допомогою якого єдиний додаток будується як набір невеликих сервісів, кожен з яких функціонує у своєму процесі та спілкується з рештою за допомогою легких механізмів, як правило, HTTP (рис. 1.4). Ці послуги базуються на потребах бізнесу та реалізуються незалежно, використовуючи зазвичай повністю автоматизоване середовище. Існує абсолютний мінімум централізованого управління цими послугами. Самі по собі їх можна записати, використовуючи різні мови та технології зберігання. Архітектура мікросервісу добре адаптується до процесу безперервної доставки, на відміну від архітектури, орієнтованої на сервіс, мікросервіс призначений для створення єдиного

додатку, тоді як сервісно орієнтована система – це набір додатків, які взаємодіють між собою.

Основні властивості:

– високий рівень незалежності сервісів;

– простота заміни однієї реалізації сервісу іншою;

– сервіси організовані відносно бізнес-логіки, яку вони реалізують.

Кожна служба може бути реалізована за допомогою будь-якої мови програмування, OBD тощо. незалежно від інших.

Філософія мікросервісного підходу схожа на філософію Unix "Роби одну справу і роби це якісно":

– сервіси невеликі, розбиті на виконання єдиної функції;

– організаційна культура повинна охоплювати автоматизацію розгортання та тестування;

– культура і принципи проектування повинні охоплювати опрацювання відмов і дефектів;

– кожен сервіс гнучкий, стійкий до відмов, легко компонуємий з іншими сервісами, функціонально мінімальний та закінчений.

З точки зору якості існує техніка, яка описує основні характеристики, які повинні бути притаманні додатку з добре розробленою архітектурою: методика застосування дванадцяти факторів. Методика підходить для розробки додатків, включаючи мікросервіси, призначені для розміщення в хмарному середовищі.

Ця архітектура постійно критикується з моменту її створення. Окрім вирішення старих проблем та спрощення роботи, архітектура має ще кілька проблем:

– мережеві затримки;

– формати повідомлень;

– баланс навантаження і відмово стійкості;

– ускладнюється тестування і розробка.

Появився термін і, отже, архітектура, "наносервіси", а також піддавався критиці. Nanoservices – це архітектура, поділена на дуже малі модулі. Ця архітектура

була визнана неефективною через величезні проміжні витрати на інтеграцію багатьох наносервісів у повний сервіс.

Багато відомих компаній вирішили проблему моноліту, прийнявши архітектуру мікросервісу замість побудови єдиного жахливого моноліту. До них відносяться Amazon, eBay, Walmart, Netflix, SoundCloud, Spotify, Twitter, Stripe, PayPal, Uber та Medium.

Реалізація Netflix була настільки успішною, що вони виявили багато програмних засобів, що використовуються для розробки їх архітектури мікросервісів. Сьогодні Netflix можна вважати піонером розвитку мікросервісу, а його підхід став дослідницькою метою багатьох інших компаній у всьому світі.

Створення складних додатків по суті складно. Монолітна архітектура має сенс лише для простих і легких програм. Ви можете опинитися у світі болю, якщо використовувати його для складних програм. Архітектура мікросервісу, незважаючи на свої недоліки та проблеми із впровадженням, є найкращим варіантом для складних та розвиваються продуктів [6].

1.1.3 Порівняння монолітної архітектури та мікросервісів

Монолітна архітектура передбачає реалізацію всіх ресурсних служб як єдиної програмної системи. Тобто всі сервіси реалізуються за допомогою набору технологій (та мови програмування) та використовують загальні бібліотеки коду. Усі сервіси працюють з одним сервером баз даних, що дозволяє кожному сервісу отримати доступ безпосередньо до бази даних. Архітектура мікросервісу застосовує інший підхід. Кожна служба реалізується як окрема програмна система, часто кожна служба має свою базу даних. Оскільки служби не мають доступу до бази даних іншої служби, доступ до таких даних здійснюється за допомогою виклику інших служб ресурсу. Часто служби групуються, якщо вони реалізують подібні або тісно пов'язані з ними функції.

Важливою особливістю при оцінці процесу розвитку будь-якого ресурсу є Час на ринок. Ця функція оцінює швидкість використання функцій кінцевого

користувача. Коли використовується монолітна архітектура, встановлення нової версії навіть сервісу вимагає реалізації всієї системи. Оскільки все більше та більше ресурсів реалізуються у хмарі, постійно перевстановлювати всю програму дуже дорого. Крім того, установка всього продукту займає тривалий час, тому ресурс стає недоступним на момент встановлення, що, в свою чергу, може бути неприйнятним для деяких веб-ресурсів. Для мікросервісу процес впровадження нової версії незалежний для кожної частини. Це скорочує час перевстановлення та не блокує інших компонентів для користувачів.

Процес розробки програми монолітної архітектури є складнішим, оскільки будь-яка зміна може спричинити проблеми в іншій частині проекту. У архітектурі мікросервісу все простіше: кожен розробник відповідає за власну систему, яка не така громіздка, як ніби це моноліт.На рисунку 1.4 видно порівняння мікросервісної та монолітної архітектури

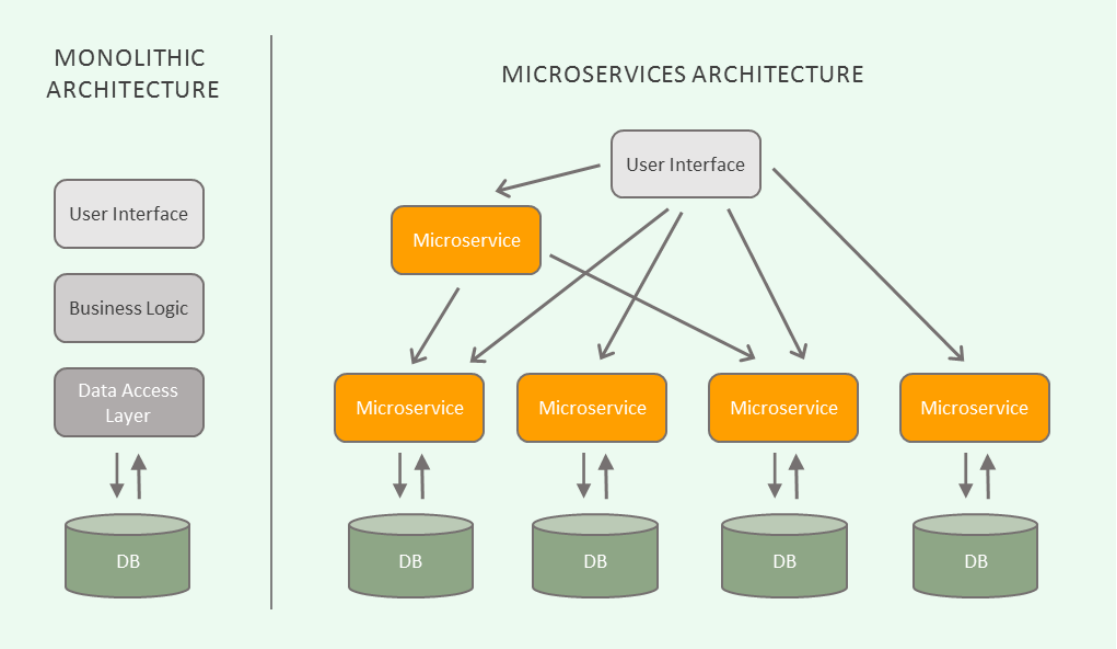

Рисунок 1.4 – Мікросервіси та моноліт порівняння

1.1.4 Одиниця віртуалізації контейнер

Віртуалізація на рівні операційної системи – це метод віртуалізації, при якому ядро операційної системи підтримує кілька ізольованих екземплярів простору користувача замість одного. Ці екземпляри (часто їх називають контейнерами або зонами) з точки зору користувача повністю ідентичні справжньому серверу. Для систем на базі UNIX ця технологія схожа на вдосконалену реалізацію механізму chroot. Ядро забезпечує повну ізоляцію контейнера, тому застосування різних контейнерів не може впливати один на одного.

Тобто віртуалізація на рівні операційної системи дозволяє віртуалізувати фізичні сервери на рівні ядра операційної системи. Шар віртуалізації операційної системи забезпечує ізоляцію та безпеку ресурсів між різними контейнерами. Шар віртуалізації робить кожен контейнер схожим на фізичний сервер. Кожен контейнер обслуговує додатки в ньому та навантаження.

Основні переваги контейнерної віртуалізації:

– контейнери виконуються на одному рівні з фізичними серверами. Відсутність віртуалізованого обладнання і використання реального обладнання і драйверів дозволяють отримати неперевершену продуктивність;

– кожен контейнер може масштабуватися до ресурсів цілого фізичного сервера;

– технологія віртуалізації на рівні ОС дозволяє домогтися високої щільності, серед доступних серед рішень віртуалізації. Можливе створення і запуск сотень контейнерів на одному звичайному фізичному сервері;

– контейнери використовують єдину ОС, що робить їх підтримку і оновлення дуже простим. Застосунки можуть бути також розгорнуті в окремому оточенні.

Віртуалізація на рівні операційної системи порівнюється з апаратною віртуалізацією на основі віртуальної машини (рис. 1.5). Віртуальна машина – це модель комп'ютерної машини, створена шляхом віртуалізації комп'ютерних ресурсів: процесора, оперативної пам’яті, зберігання та введення та виведення інформації. На

відміну від програми емуляції, характерної для пристрою, віртуальна машина забезпечує повну емуляцію фізичної машини або часу виконання (для програми). Якщо проект вимагає повного контролю над ядром операційної системи, ручним оновленням операційної системи, Windows як операційної системи, тоді необхідно використовувати апаратну віртуалізацію.

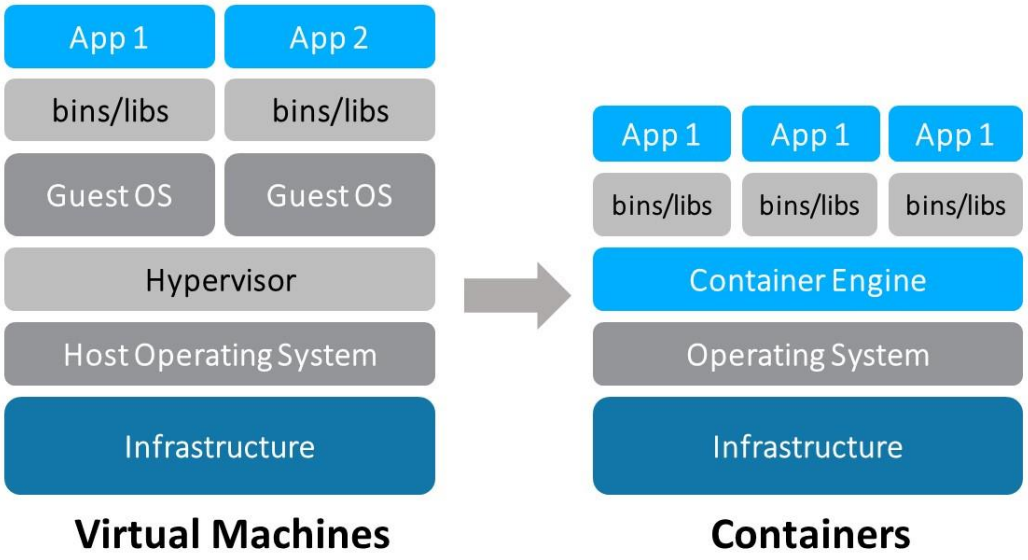

Рисунок 1.5 – Порівняння віртуалізації на рівні ОС та апаратної віртуалізації

Зазвичай віртуальні машини (ВМ) використовувались для уникнення несподіваної поведінки програми. Основна проблема полягає в тому, що VM – це додаткова операційна система, яка працює над хостом, і це додаткові гігабайти для проекту. Часто на вашому сервері розміщується кілька віртуальних машин, на кожній з яких достатньо місця. І відомо, що більшість хмарних платформ заряджають додатковий простір. Ще одна важлива незручність віртуальної машини – повільне завантаження. Docker вирішує всі ці проблеми, розділяючи ядро операційної системи між усіма контейнерами, які функціонують як окремі процеси хост-операційної системи.

Варто пам’ятати, що Docker – це не перша і не єдина платформа для контейнера. Однак зараз це найпопулярніший і найпотужніший варіант серед аналогів.

1.1.5 Оркестрація контейнерів

Оркестрація контейнерів – це процес організації декількох контейнерів на мережевому рівні, щоб програма, що складається з незалежних контейнерів (мікросервісів), працювала, як було заплановано. Оркестрація описує, як служби повинні взаємодіяти між собою за допомогою обміну даними, включаючи ділову логіку та послідовність дій.

Процес можна налагоджувати на всіх платформах. У той час як платформи, такі як Apache Mesos, Google Kubernetes та Docker Swarm, мають власні специфічні методології управління контейнерами, усі механізми оркестрації дозволяють користувачам контролювати, запускати та зупиняти контейнери, групувати їх у кластери та координувати всі процеси, необхідні для реалізація Засоби управління контейнерами дозволяють користувачам керувати розгортанням контейнерів та автоматизувати оновлення, контролюючи стан програми та її частин.

Переваги оркестрації контейнерів:

- координація та автоматизація усіх процесів;
- централізоване керування контейнерами, які складають єдиний застосунок;
- спрощення процесів оновлення та моніторингу;
- автоматизація процесу відмово стійкості застосунку (автоматичне переключення);
- простота налаштування.

1.2 Система оркестрації контейнерів Kubernetes

1.2.1 Архітектура Kubernetes

Архітектура Kubernetes забезпечує гнучкий, нещільно пов'язаний механізм для виявлення послуг. Як і більшість розподілених обчислювальних платформ, кластер

Kubernetes складається щонайменше з одного головного та декількох обчислювальних вузлів (рис 1.6). Майстер відповідає за розкриття інтерфейсу прикладної програми (API), планування розгортань та управління загальним кластером. Кожен вузол виконує час виконання контейнера, наприклад Docker або rkt, разом із агентом, який спілкується з майстром.

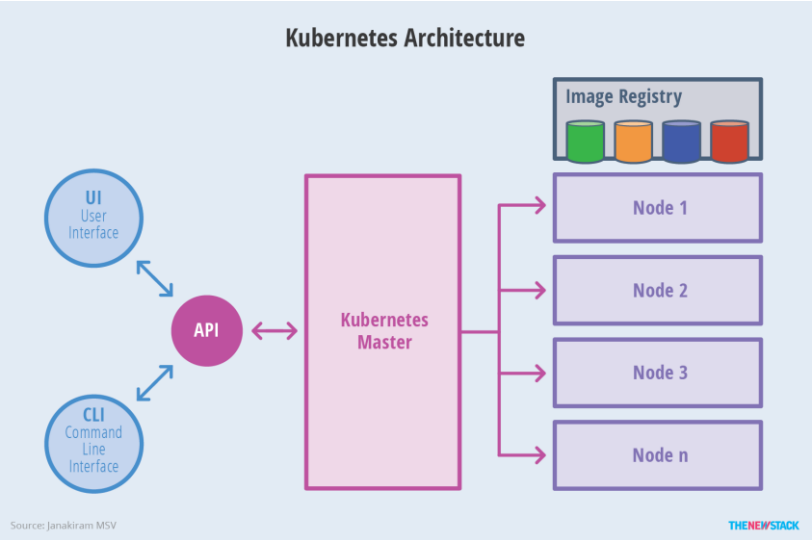

Рисунок 1.6 – Загальна схема архітектури

Вузол також працює з додатковими компонентами для ведення журналу, моніторингу, виявлення сервісу та додаткових застосунків. Вузли є робочими машинами кластеру Kubernetes. Вони виставляють програми, обчислювальну мережу та мережу зберігання. Вузли можуть бути віртуальними машинами (ВМ), що працюють у хмарі, або голі металеві сервери, що працюють в центрі обробки даних (рис 1.7).

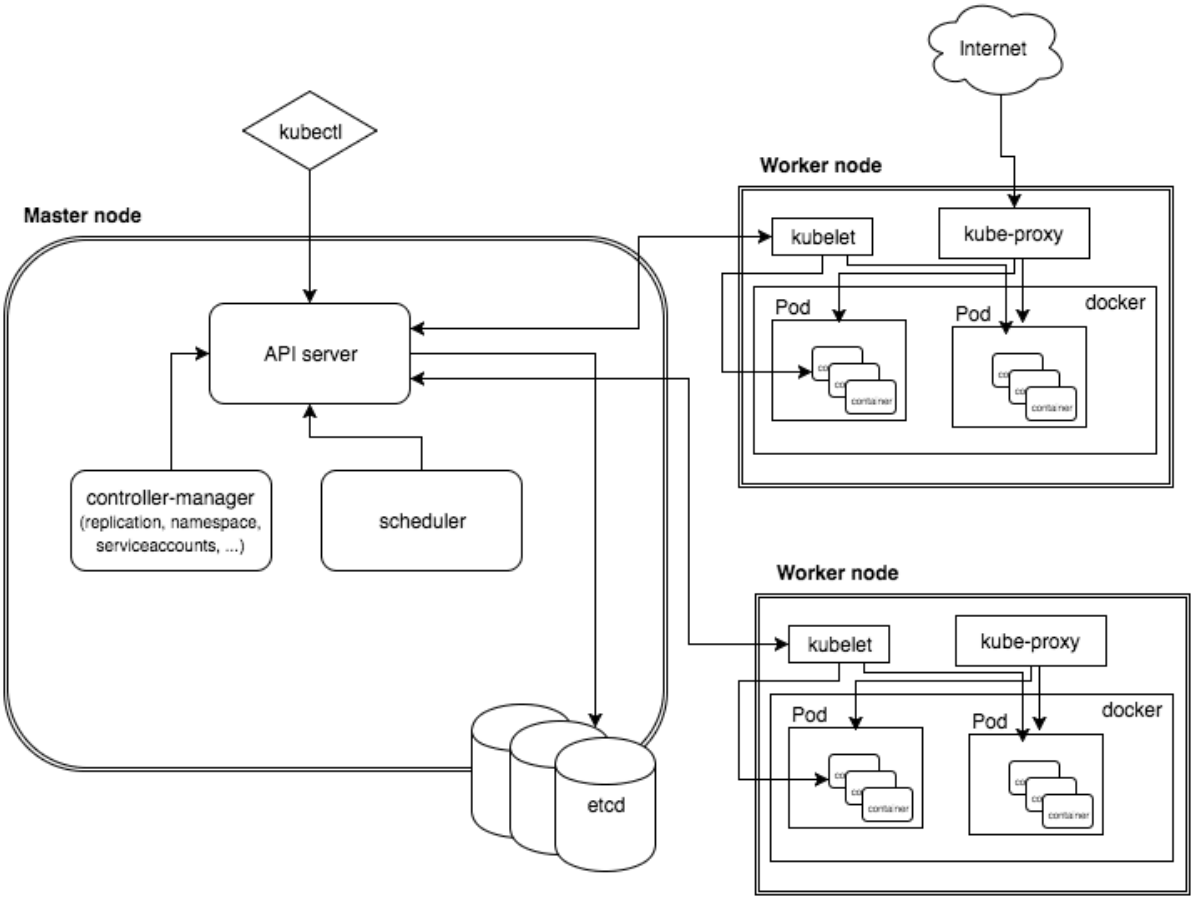

Рисунок 1.7 – Схема кластера з сервісами всередині вузлів

1.2.1.1 Вузли (Nodes)

Вузол – це робоча машина в Kubernetes, раніше відома як міньйон. Вузол може бути ВМ або фізичною машиною, залежно від кластера. Кожен вузол містить сервіси, необхідні для запуску подів і керуються основними компонентами. Послуги на вузлі включають в себе час виконання контейнерів, kubelet та kube-proxy.

Статус вузла містить таку інформацію:

– мережеа адреса;
– статус;
– ємність і розподілення;
– інформація.

Статус вузла та інші деталі щодо вузла можуть відображатися за допомогою команди:

```
kubectl describe node <insert-node-name-here>
```

Адреса. Використання цих полів залежить від постачальника послуг хмари або конфігурації серверу.

– HostName: ім'я хоста, як повідомляється ядром вузла. Можна змінити за допомогою параметра kubelet --hostname-override.

– ExternalIP: Зазвичай IP-адреса вузла, який є зовнішньо маршрутизованим (доступний за межами кластера).

– InternalIP: Зазвичай IP-адреса вузла, який може бути маршрутизований лише в кластері.

Таблиця 1.1 – Опис статусів вузлів

Статус вузла	Опис
Ready	True, якщо вузол здоровий і готовий приймати поди, False, якщо вузол не здоровий і не приймає поди, і Uknown, якщо контролер вузла не чув від вузла в останньому періоді node-monitor-grace-period (за замовчуванням 40 секунд)
MemoryPressure	True, якщо тиск на пам'ять вузла існує – тобто якщо пам’ять вузла є низькою; інакше False
PIDPressure	True, якщо на процеси існує тиск – тобто якщо на вузлі занадто багато процесів; інакше False
DiskPressure	True, якщо існує тиск на розмір диска – тобто якщо ємність диска низька; інакше False
NetworkUnavailable	True, якщо мережа для вузла неправильно налаштована, інакше помилково

Якщо статус залишається Uknown або False довше, ніж час-час витримки, то аргумент передається kube-controller-manager, і всі структури у вузлі планується видалити контролером вузла. Тривалість тайм-ауту виселення за замовчуванням становить п'ять хвилин. У деяких випадках, коли вузол недоступний, API сервер не в змозі спілкуватися з kubelet на вузлі. Рішення про видалення подів не може бути

передано kubelet доти, доки не буде відновлено зв'язок із аплікатором. Тим часом поди, призначені для видалення, можуть продовжувати працювати на розділеному вузлі.

Ємність та розподілення. Описує ресурси, доступні на вузлі: процесор, пам'ять та максимальну кількість подів, які можна запланувати на вузол. Поля в блоці ємності вказують загальну кількість ресурсів, які має вузол.

Інформація. Описує загальну інформацію про вузол, таку як версія ядра, версія Kubernetes (версія kubelet та kube-proxy), версія Docker (якщо використовується) та ім'я ОС. Цю інформацію kubelet збирає з вузла.

На відміну від подів і служб, вузол не притаманний Kubernetes: він створюється зовні хмарними постачальниками, такими як Google Compute Engine, або він існує у вашому пулі фізичних чи віртуальних машин. Отже, коли Kubernetes створює вузол, він створює об'єкт, який представляє вузол (рис. 1.8). Після створення Kubernetes перевіряє, чи дійсний вузол чи ні.

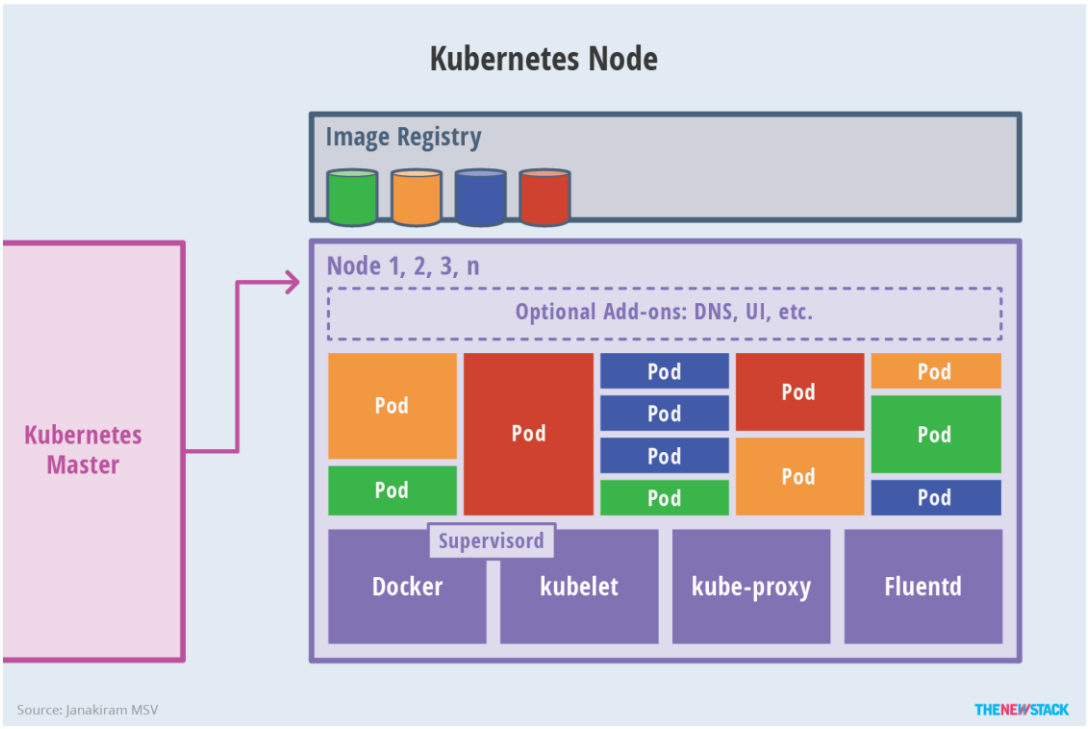

Рисунок 1.8 – Структура Kubernetes вузла

1.2.1.2 Комунікація між мастером та вузлами

З кластера до вузла. Всі шляхи зв’язку від кластера до головного завершуються в API сервері (жоден з інших основних компонентів не призначений для викриття віддалених служб). У типовому розгортанні API сервер налаштований для прослуховування віддалених з'єднань на захищеному порту HTTPS (443) з увімкненою однією або кількома формами аутентифікації клієнта. Потрібно ввімкнути одну або кілька форм авторизації, особливо якщо дозволено анонімні запити або жетони облікового запису послуги.

Вузли повинні бути забезпечені загальнодоступним кореневим сертифікатом для кластеру, щоб вони могли надійно підключитися до API сервера разом із дійсними обліковими записами клієнта. Наприклад, при розгортанні GKE за замовчуванням облікові дані клієнта, надані kubelet, мають форму клієнтського сертифіката.

Поди, які бажають підключитися до API servera, можуть зробити це надійно, використовуючи обліковий запис служби, щоб Kubernetes автоматично вводив публічний кореневий сертифікат і дійсний маркер носія в режим, коли він створений. Служба kubernetes (у всіх просторах імен) налаштована з віртуальною IP-адресою, яка переадресовується (через kube-proxy) на кінцеву точку HTTPS на API сервер. Основні компоненти також спілкуються з кластерним приладом через захищений порт.

В результаті режим роботи за замовчуванням для з'єднань від кластера (вузлів і подів, що працюють на вузлах) до мастера, захищений за замовчуванням і може працювати через ненадійні та / або загальнодоступні мережі.

З вузла до кластера. Є два шляхи зв'язку від мастера (API сервер) до кластера. Перший – від API сервера до процесу kubelet, який працює на кожному вузлі кластера. Другий – від API сервера до будь-якого вузла, поду або служби через kube-proxy API сервера.

API сервер to kubelet. З'єднання від API сервера до kubelet використовуються для:

– отримання логів з подів;
– прикріплення (через kubectl) до запущених подів;

– забезпечення функціональності переадресації kubelet.

Ці з'єднання припиняються в кінцевій точці HTTPS kubelet. За замовчуванням API сервер не перевіряє сертифікат обслуговування kubelet, що робить з'єднання предметом атаки "посеред" та небезпечно для запуску через ненадійні та / або загальнодоступні мережі. Щоб підтвердити це з'єднання, використовуйте прапор --kubelet-certificat-authority, щоб надати API сервер пакету кореневих сертифікатів, який слід використовувати для підтвердження сервісного сертифіката kubelet.

API сервер до вузлів, подів та служб. З'єднання від API сервера до вузла, поду або служби за замовчуванням до звичайних HTTP-з'єднань, тому не є ні автентифікованими, ні зашифрованими. Їх можна запустити через захищене HTTPS-з'єднання за допомогою префіксації https: до вузла, пода або імені служби в URL-адресі API, але вони не перевірять сертифікат, наданий кінцевою точкою HTTPS, і не надаватимуть облікові дані клієнта, тому підключення буде зашифровано, це не забезпечить гарантій цілісності. Наразі ці з'єднання не є безпечними для запуску через ненадійні та / або загальнодоступні мережі.

SSH тунелі. Kubernetes підтримує тунелі SSH для захисту шляхів зв'язку мастер -> кластер. У цій конфігурації API сервер ініціює тунель SSH до кожного вузла кластеру (підключення до ssh-сервера, який прослуховує порт 22) і передає весь трафік, призначений для kubelet, вузла, пода або послуги через тунель. Цей тунель забезпечує недоступність трафіку за межами мережі, в якій працюють вузли.Наразі тунелі SSH застаріли, тому не варто використовувати їх.

1.2.1.3 Контролери

У Kubernetes контролери – це петлі управління, які відстежують стан кластеру, а потім вносять або вимагають зміни, де це необхідно. Кожен контролер намагається перенести поточний стан кластера ближче до потрібного стану.

Керування через сервер AP. Контролер Job є прикладом вбудованого контролера Kubernetes. Вбудовані контролери керують станом, взаємодіючи з сервером API кластера. Job – це ресурс Kubernetes, який виконує поділ, або, можливо,

кілька Pods, щоб виконати завдання, а потім зупинитися. Коли контролер Job бачить нове завдання, він переконується, що десь у вашому кластері kubelet на наборі вузлів запускають потрібну кількість Pods, щоб виконати роботу. Контролер Job не запускає ні Pods, ні контейнери. Натомість контролер Job повідомляє сервер API створити або видалити Pods. Інші компоненти в площині управління діють на нову інформацію (є нові Pods для розкладу та запуску), і з часом робота виконується. Після того, як створиться нову роботу, бажаний стан буде виконати цю роботу. Контролер Job робить поточний стан для цієї роботи ближчим до бажаного стану: створює Pods, які виконують роботу. Контролери також оновлюють об'єкти, які їх налаштовують. Наприклад: як тільки робота виконується для роботи, контролер Job оновлює цей об'єкт Job, щоб позначити його завершеним.

Прямий контроль. На відміну від Job, деяким контролерам потрібно внести зміни до речей поза вашим кластером. Контролери, які взаємодіють із зовнішнім станом, знаходять потрібний стан із сервера API, а потім безпосередньо спілкуються із зовнішньою системою, щоб привести поточний стан у відповідність.

1.2.1.4 Поняття, що лежать в основі менеджера хмарних контролерів

Концепція менеджера хмарних контролерів (CCM) (спочатку була створена, щоб дозволити конкретному постачальника хмари та ядру Kubernetes розвиватися незалежно один від одного. Менеджер хмарних контролерів працює поряд з іншими основними компонентами, такими як менеджер контролерів Kubernetes, сервер API та планувальник (рис 1.9). Його також можна запустити як застосунок Kubernetes, і в цьому випадку він працює над Kubernetes.

Дизайн менеджера хмарного контролера заснований на механізмі плагінів, який дозволяє новим постачальникам послуг хмари легко інтегруватися з Kubernetes за допомогою плагінів. Існують плани щодо включення нових хмарних провайдерів на Kubernetes та для міграції хмарних провайдерів зі старої моделі на нову модель CCM.

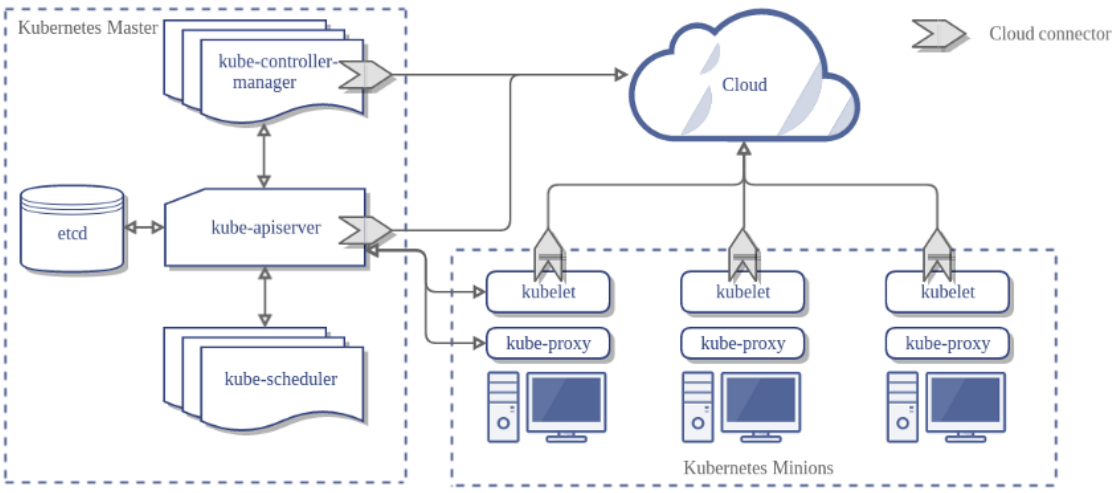

Рисунок 1.9 – Архітектура кластера Kubernetes без менеджера хмарного контролера

Компоненти CCM. CCM відриває частину функцій менеджера контролерів Kubernetes (KCM) і запускає його як окремий процес. Зокрема, він розбиває ті контролери в KCM, які залежать від хмарності. У KCM є такі петлі залежно від хмарних контролерів:

– контролер вузла;
– регулятор гучності;
– контролер маршруту;
– сервісний контролер.

У версії 1.9 CCM працює з наступними контролерами з попереднього списку:

– контролер вузла;
– контролер маршруту;
– сервісний контролер.

Функції CCM. CCM успадковує свої функції від компонентів Kubernetes, які залежать від хмарного постачальника. Цей розділ побудований на основі цих компонентів.

Менеджер контролерів Kubernetes. Більшість функцій CCM походять від KCM. KCM виконує наступні контури управління:

– контролер вузла;

– контролер маршруту;

– сервісний контролер;

– контролер вузла.

Контролер вузла відповідає за ініціалізацію вузла шляхом отримання інформації про вузли, що працюють у кластері від хмарного постачальника. Контролер вузла виконує такі функції:

– ініціалізація вузла з мітками, характерними для конкретних хмарних зон / регіонів;

– ініціалізація вузла з конкретними деталями екземпляра, наприклад, тип та розмір;

– отримання мережевих адрес та імені вузла;

– контролер маршруту.

Контролер маршрутів відповідає за правильне налаштування маршрутів у хмарі, щоб контейнери на різних вузлах кластеру Kubernetes могли спілкуватися один з одним. Контролер маршруту застосовується лише для кластерів Google Compute Engine.

Сервісний контролер. Контролер служби відповідає за прослуховування подій створення, оновлення та видалення служб. Виходячи з поточного стану сервісів у Kubernetes, він налаштовує балансири хмарних навантажень (наприклад, ELB, Google LB або Oracle Cloud Infrastructure LB), щоб відображати стан послуг у Kubernetes. Крім того, це забезпечує оновлення службових резервів для балансуючих хмарних навантажень.

Kubelet. Контролер Node містить функцію, що залежить від хмари kubelet. До введення CCM, kubelet відповідав за ініціалізацію вузла із специфічними для хмари деталями, такими як IP-адреси, мітки регіону / зони та інформація про тип екземпляра. Введення CCM перенесло цю операцію ініціалізації з kubelet в CCM.

У цій новій моделі kubelet ініціалізує вузол без конкретної хмарної інформації. Однак він додає до новоствореного вузла домішку, яка робить вузол

непередбачуваним, поки CCM не ініціалізує вузол інформацією, характерною для хмари. Потім він знімає цей відтінок.

Kubernetes постачається з набором вбудованих контролерів, які працюють всередині кубе-контролера-менеджера. Ці вбудовані контролери забезпечують важливу поведінку в основному.

Контролер розгортання та контролер завдань – приклади контролерів, які входять до складу самого Kubernetes ("вбудовані" контролери). Kubernetes дозволяє запустити пружну площину управління, так що якщо будь-який з вбудованих контролерів повинен вийти з ладу, інша частина площини управління візьме на себе роботу.

1.2.2 Типи об’єктів Kubernetes

Об'єкти Kubernetes – це сутності, які використовуються для представлення стану кластеру. Об'єкт – це "запис про наміри" – колись створений кластер робить все можливе, щоб він існував, як визначено [5]. Це відомо як "бажаний стан кластера". Kubernetes завжди працює над тим, щоб "поточний стан" об'єкта дорівнювати "бажаному стану" об'єкта. Бажаний стан може описувати:

- які поди (контейнери) працюють, а на яких вузли;
- кінцеві точки IP, які відображають в логічну групу контейнерів;
- скільки реплік контейнера працює.

1.2.2.1 Под (Pod)

Под – це група одного або декількох контейнерів (таких як Docker-контейнери), що мають спільне сховище / мережу та специфікацію способів запуску контейнерів. Вміст пода завжди розміщений за спільним розташуванням та спільним розкладом, і він працює у спільному контексті. Под моделює "логічний хост" для конкретного застосунка – він містить один або кілька контейнерів застосунків, які

відносно щільно з'єднані – у світі перед контейнерами, виконання на одній фізичній або віртуальній машині означало б виконання на одному і тому ж логічному хості .

Хоча Kubernetes підтримує більше часу виконання контейнерів, ніж просто Docker, Docker – це найбільш відомий час виконання, і він допомагає описати Pods в термінах Докера.

Спільний контекст Pod – це набір просторів імен Linux, груп та потенційно інших аспектів ізоляції – тих самих речей, які ізолюють контейнер Docker. У контексті Pod, окремі програми можуть застосовувати додаткові підрозділи.

Контейнери в межах Pod поділяють IP-адресу та простір порту, і можуть знаходити один одного через localhost. Вони також можуть спілкуватися між собою, використовуючи стандартні міжпроцесорні комунікації, такі як семафори SystemV або спільну пам'ять POSIX. Контейнери в різних подіумах мають чіткі IP-адреси і не можуть спілкуватися по IPC без спеціальної конфігурації. Ці контейнери зазвичай спілкуються між собою через IP-адреси Pod.

Програми в межах Pod також мають доступ до спільних томів, які визначені як частина Pod та доступні для монтажу у файлову систему кожної програми.

З точки зору конструкцій Docker, Pod моделюється як група контейнерів Docker із спільними просторами імен та спільними томами файлової системи (рис.1.10).

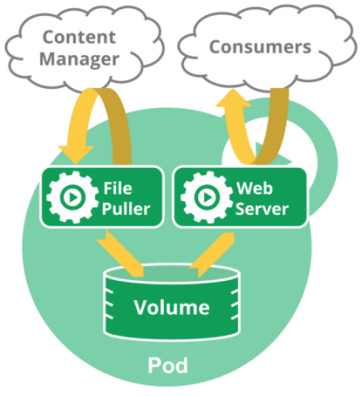

Рисунок 1.10 – Схема пода

Як і окремі контейнери для застосування, поди вважаються відносно ефемерними (а не міцними) сутностями. Як обговорювалося в життєвому циклі поділки, Струс створюється, присвоюється унікальний ідентифікатор (UID) та

планується до вузлів, де вони залишаються до припинення (відповідно до політики перезавантаження) або видалення. Якщо вузол гине, поділки, заплановані до цього вузла, плануються для видалення, після періоду таймауту. Даний Pod (як визначено UID) не "перенесено" на новий вузол; замість цього він може бути замінений на ідентичний Pod, з навіть тим же ім'ям, якщо бажано, але з новим UID (детальніше див. контролер реплікації).

Коли щось, як кажуть, має такий самий термін служби, як Pod, такий як об'єм, це означає, що він існує до тих пір, поки існує Pod (з цим UID). Якщо цей Pod видаляється з будь-якої причини, навіть якщо створена ідентична заміна, пов'язана річ (наприклад, том) також знищується та створюється заново.

1.2.2.2 ДемонСет (DaemonSet)

DaemonSet гарантує, що всі (або деякі) Вузли виконують копію Pod. Коли вузли додаються до кластеру, до них додаються Pods. Коли вузли видаляються з кластера, ці поди збирають сміття. Видалення DaemonSet очистить створені поди.

Деякі типові способи використання DaemonSet:

– запуск демона зберігання кластерів, таких як glusterd, ceph, на кожному вузлі;

– запуск демона колекції журналів на кожному вузлі, такому як вільне слово або logstash;

– запуск демона моніторингу вузлів на кожному вузлі, наприклад, експортер вузла Prometheus, агент Sysdig, colled, Dynatrace OneAgent, агент AppDynamics, агент Datadog, агент New Relic, Ganglia gmond або Instana Agent.

У простому випадку один DaemonSet, що охоплює всі вузли, буде використовуватися для кожного типу демона. Більш складна установка може використовувати декілька DaemonSets для одного типу демона, але з різними прапорцями та / або різними запитами на пам'ять і процесор для різних типів обладнання.

1.2.2.3 Розгортання (Deployment)

Розгортання забезпечує декларативні оновлення для Pods та ReplicaSets.

Ви описуєте бажаний стан у розгортанні, і контролер розгортання змінює фактичний стан на потрібний стан з контрольованою швидкістю. Можна визначити Deployments для створення нових ReplicaSets або для видалення існуючих Deployments та прийняття всіх їх ресурсів за допомогою нових Deployments.

Нижче наведено типові випадки використання для розгортання:

– розгортання для розгортання ReplicaSet. ReplicaSet створює Pods у фоновому режимі. Перевірте стан розгортання, щоб побачити, вдалося це чи ні;

– новий стан подів, оновлення PodTemplateSpec розгортання. Створюється новий ReplicaSet, і Deployment керує переміщенням Pods зі старого ReplicaSet до нового з контрольованою швидкістю. Кожна нова ReplicaSet оновлює перегляд Розгортання;

– відкат до попередньої версії розгортання, якщо поточний стан Розгортання не є стабільним. Кожен відкат оновлює версію розгортання;

– масштабування розгортання, щоб полегшити більше навантаження;

– призупинення Розгортання, щоб застосувати кілька виправлень до свого PodTemplateSpec, а потім відновіть його, щоб почати нову програму;

– статус розгортання як індикатор того, що випуск застряг.

1.2.2.4 РеплікаСет (ReplicaSet)

Завдання ReplicaSet – підтримувати стабільний набір реплік Pods, що працює в будь-який момент часу. Як такий, його часто використовують для гарантування наявності визначеної кількості однакових Pods.

ReplicaSet визначається з полями, включаючи селектор, який визначає, як ідентифікувати поди, який він може придбати, кількість реплік, які вказують, скільки Pods він повинен підтримувати, і шаблон поділу, що визначає дані нових Pods, які він

повинен створити, щоб відповідати номеру критеріїв реплік Потім ReplicaSet виконує своє призначення, створюючи та видаляючи поди за необхідності, щоб досягти потрібного числа. Коли ReplicaSet потребує створення нових подів, він використовує його PodTemplate.

Посилання, яке має ReplicaSet, до його подів здійснюється через поле metadata.ownerReferences подів, яке вказує, яким ресурсом належить поточний об'єкт. Усі поди, створені за подомогою ReplicaSet, мають власну ідентифікаційну інформацію ReplicaSet у полі свого власника. Саме через це посилання ReplicaSet знає про стан Pods, який він підтримує, і планує відповідно.

ReplicaSet ідентифікує нові Pods для придбання за допомогою свого селектора. Якщо є под, у якого немає OwnerReference або OwnerReference не є Контролером, і він відповідає селектору ReplicaSet, він буде негайно придбаний зазначеним ReplicaSet.

ReplicaSet гарантує, що визначена кількість реплік пода працює у будь-який момент часу. Однак, Deployment – це концепція вищого рівня, яка управляє ReplicaSets і надає декларативні оновлення для Pods разом із безліччю інших корисних функцій. Тому ми рекомендуємо використовувати розгортання замість прямого використання ReplicaSets, якщо не потрібна власна організація оновлення або не потрібна оновлення взагалі.

Це насправді означає, що ніколи не потрібно буде маніпулювати об'єктами ReplicaSet: замість цього краще використовувати Deployment та визначте свою програму в розділі специфікації.

1.2.2.5 Робота (Job)

Робота створює один або кілька Pods і забезпечує, щоб певна кількість з них успішно припинилася. Коли поди успішно завершені, робота відстежує успішні завершення. Коли буде досягнуто заданої кількості успішних завершень, завдання (тобто Робота) виконано. Якщо видалити роботу, то створені ресурси та сервіси роботи будуть видалені. Простий приклад – створити один об'єкт Job, щоб надійно

запустити один Pod до завершення. Об'єкт Job запустить новий Pod, якщо перший Pod не вдається або буде видалений (наприклад, через збій апаратного забезпечення вузла або перезавантаження вузла). Також можна використовувати Job для запуску декількох Pods паралельно.

1.2.2.6 Сервіс

Абстрактний спосіб розкрити застосунок, що працює на наборі Pods як мережеву послугу.

У Kubernetes не потрібно змінювати свою програму, щоб використовувати незнайомий механізм виявлення послуги. Kubernetes надає Pods власні IP-адреси та єдине DNS-ім’я для набору Pods, і може завантажувати баланс через них.

У Kubernetes Сервіс – це абстракція, яка визначає логічний набір Pods та політику, за допомогою якої можна отримати доступ до них (іноді ця схема називається мікропослугою). Набір Pods, орієнтований на Сервіс, зазвичай визначається селектором

Наприклад, розглянемо бекенд для обробки зображень без стану, який працює з 3-ма репліками. Ці репліки є мінливими – фронтенди не байдужі, який бекенд вони використовують. Незважаючи на те, що фактичні Pods, які складають набір резервних копій, можуть змінюватися, клієнти, що працюють за кордоном, не повинні знати про це, а також не повинні самі відслідковувати набір резервних копій.

У деяких частинах вашої програми (наприклад, на фронтальних сторінках) можете виставити Сервіс на зовнішню IP-адресу, яка знаходиться поза кластером.

Kubernetes ServiceTypes дозволяють вказувати, який тип послуги потрібен. За замовчуванням – ClusterIP.

Значення типу та їх поведінка:

ClusterIP: Розкриває Службу на внутрішній IP-кластер. Вибір цього значення робить Службу доступною лише з кластеру. Це ServiceType за замовчуванням.

NodePort: відкриває службу на IP-адресу кожного вузла на статичному порту (NodePort). Служба кластераIP, до якої маршрутизується служба NodePort,

автоматично створюється. Можливо зв’язатися із службою NodePort за межами кластера, подавши запит <NodeIP>: <NodePort>.

LoadBalancer: відкриває Службу зовні за допомогою балансира навантаження хмарного постачальника. Служби NodePort та ClusterIP, до яких спрямовуються зовнішні маршрутизатори балансування, автоматично створюються.

ExternalName: Зверніть Сервіс до вмісту поля externalName (наприклад, foo.bar.example.com), повернувши запис CNAME з його значенням. Ніякого проксингу не встановлюється.

Також можна використовувати Ingress для викриття своєї послуги. Ingress – це не тип сервісу, але він служить точкою входу для вашого кластеру. Це дозволяє консолідувати свої правила маршрутизації в єдиний ресурс, оскільки він може виставляти кілька сервісів під однією IP адресою.

1.2.2.7 Мітка (Label)

Мітки – це пара ключів / значень, які прикріплені до об'єктів, наприклад, поди. Мітки призначені для використання для визначення ідентифікаційних атрибутів об'єктів, які є значущими та актуальними для користувачів, але безпосередньо не передбачають семантику основної системи. Мітки можна використовувати для організації та вибору підмножини об'єктів. Мітки можуть бути прикріплені до об'єктів під час створення, а згодом додаватися та змінюватися в будь-який час. Кожен об’єкт може мати визначений набір міток ключа / значення. Кожен ключ повинен бути унікальним для певного об’єкта.

1.2.3 Застосування Kubernetes

Оскільки Kubernetes – це потужний інструмент управління контейнерами, який автоматизує розгортання та управління контейнерами, то треба визначити основні сфери застосування.

Оркестрація контейнерів. Контейнери надають простий спосіб упаковки та розгортання послуг, дозволяють ізолювати процеси, незмінність, ефективне використання ресурсів та легкі у створенніЦі контейнери потрібно розгорнути, керувати, підключити та оновити; якби це робити вручну, знадобиться ціла команда, присвячена цьому [6].

Чудово підходить для прийняття мульти хмар. Оскільки сьогоднішній бізнес працює на архітектурі мікросервісів, не дивно, що контейнери та інструменти, що використовуються для управління ними, стали настільки популярними. Архітектура мікросервісу дозволяє легко розділити застосунок на більш дрібні компоненти з контейнерами, які потім можна запустити в різних хмарних середовищах, надаючи можливість вибрати найкращого хоста для ваших потреб. Що найкраще в Kubernetes, це те, що він побудований для використання в будь-якому місці, щоб могли розгорнутись до публічних / приватних / гібридних хмар, що дозволяє охоплювати користувачів там, де вони перебувають, з більшою доступністю та безпекою.

Розгортання та оновлення програм в масштабі для швидшого виходу на ринок. Kubernetes дозволяє командам не відставати від вимог сучасної розробки програмного забезпечення. Без Kubernetes великим командам доведеться вручну складати сценарії власних робочих процесів розгортання. Контейнери в поєднанні з інструментом для оркестрування забезпечують управління машинами та послугами для вас – покращуючи надійність вашої програми, зменшуючи при цьому час і ресурси, витрачені на DevOps.

Kubernetes має кілька чудових функцій, які дозволяють швидше розгортати програми, маючи на увазі масштабованість:

– горизонтальне масштабування інфраструктури: нові сервери можна легко додавати або видаляти;

– автоматичне масштабування: автоматична зміна кількості запущених контейнерів на основі використання процесора або інших показників, що надаються застосунком;

– ручне масштабування: масштабування кількості запущених контейнерів вручну за допомогою команди або інтерфейсу.

Kubernetes пропонує такі можливості:

– використання контролера реплікації – він гарантує, що кластер має певну кількість запущених подів. Якщо подів занадто багато, контролер реплікації видаляє зайві поди. Якщо їх занадто мало, він запускає більше подів;

– перевірка здоров'я та самолікування. Kubernetes може перевірити стан вузлів і контейнерів, переконавшись, що програма не стикається з помилками. Kubernetes також пропонує самолікування та автоматичну заміну, тому не потрібно турбуватися, якщо контейнер чи под вийдуть з ладу;

– маршрутизація трафіку та балансування навантаження. Маршрутизація трафіку надсилає запити у відповідні контейнери. Kubernetes також вбудовані системи для балансування навантаження, щоб можна було збалансувати ресурси та швидко реагувати на перебої або періоди великого трафіку;

– автоматизовані розгортання та відкати. Kubernetes обробляє випуск нових версій чи оновлень без простоїв під час моніторингу стану контейнерів. Якщо випуск виконається з проблемами, він автоматично відкочується;

– канарські розгортання. Розгортання дозволяє тестувати нове розгортання у виробництві паралельно з попередньою версією.

Також контейнери дозволяють розбивати застосунки на більш дрібні частини, якими потім можна керувати за допомогою інструменту оркестрації, наприклад Kubernetes. Це дозволяє легко керувати базами кодів і тестувати конкретні входи та виходи. Як вже згадувалося раніше, Kubernetes має вбудовані функції, такі як самолікування та автоматизовані перекидання / відкати, ефективно керуючи контейнерами.

Kubernetes дозволяє декларативні вирази потрібного стану на відміну від виконання сценарію розгортання, тобто планувальник може контролювати кластер і виконувати дії, коли фактичний стан не відповідає бажаному.

Додаткові переваг:

– можна використовувати його для розгортання своїх служб, розгортання нових випусків без простоїв, а також для масштабування (або зменшення масштабу) цих служб;

– портативність, може працювати на публічній або приватній хмарі, в гібридному середовищі;

– можна перемістити кластер Kubernetes від одного постачальника хостингу до іншого, не змінюючи (майже) жодного з процесів розгортання та управління.

Kubernetes можна легко розширити, щоб задовольнити майже будь-які потреби. Можна вибрати, які модулі використовувати, і можна самостійно розробити додаткові функції та підключити їх. Kubernetes вирішить, де щось запустити і як підтримувати вказаний стан. Kubernetes можуть розміщувати репліки сервісу на найбільш підходящому сервері, перезапускати їх за потреби, копіювати їх та масштабувати. Самолікування – особливість, включена в його розробку з самого початку. З іншого боку, незабаром настає самоадаптація.

Коли треба використовувати його:

– якщо програма використовує архітектуру мікросервісу;

– якщо перейшли або переходять на архітектуру мікросервісу, то Kubernetes добре підійде, тому що, ймовірно, вже використовується програмне забезпечення, наприклад Docker, для контейнеру вашої програми;

– якщо потрібен швидкий розвиток та розгортання;

– якщо не можливо задовільнити потреби клієнтів через повільний час розробки, Kubernetes може допомогти. Замість того, щоб команда розробників витрачала свій час, обертаючи голови навколо життєвого циклу розробки та розгортання, Kubernetes (разом з Docker) може ефективно управляти цим для вас, щоб

команда могла витратити свій час на більш змістовну роботу, яка виводить продукти у двері.

Зниження витрат на інфраструктуру. Kubernetes використовує ефективну модель управління ресурсами на рівні контейнерів, подів і кластерів, допомагаючи знизити хмарні інфраструктурні витрати, забезпечуючи, щоб ваші кластери завжди мали доступні ресурси для запуску програм.

Коли не потрібно його використовувати:

– прості, легкі програми;

– якщо програма використовує монолітну архітектуру, може бути важко побачити реальні переваги контейнерів та інструменту, який використовується для їх оркестрування;

– не очікується, нові версії продукти будуть надходити.

Kubernetes – наступна велика хвиля хмарних обчислень, і легко зрозуміти, чому бізнес мігрує свою інфраструктуру та архітектуру, щоб відобразити епоху, керовану даними, керовану даними.

1.2.4 Порівняння Kubernetes та Docker Swarm

Docker swarm – це абстракція Docker, що прибирає кордони між різними машинами. Той же Docker engine, але працює в кластері. Декілька контейнерів-вузлів об'єднуються в сервіс, який обслуговує Swarm менеджер (рис 1.11). Swarm менеджер розміщує контейнери з застосунком на вільних хостах і приймає команди на управління кластером. Також Swarm працює як балансувальник навантаження між декількома «Воркерами», рівномірно розподіляючи запити, що приходять з будь-якої з сторін кластера.

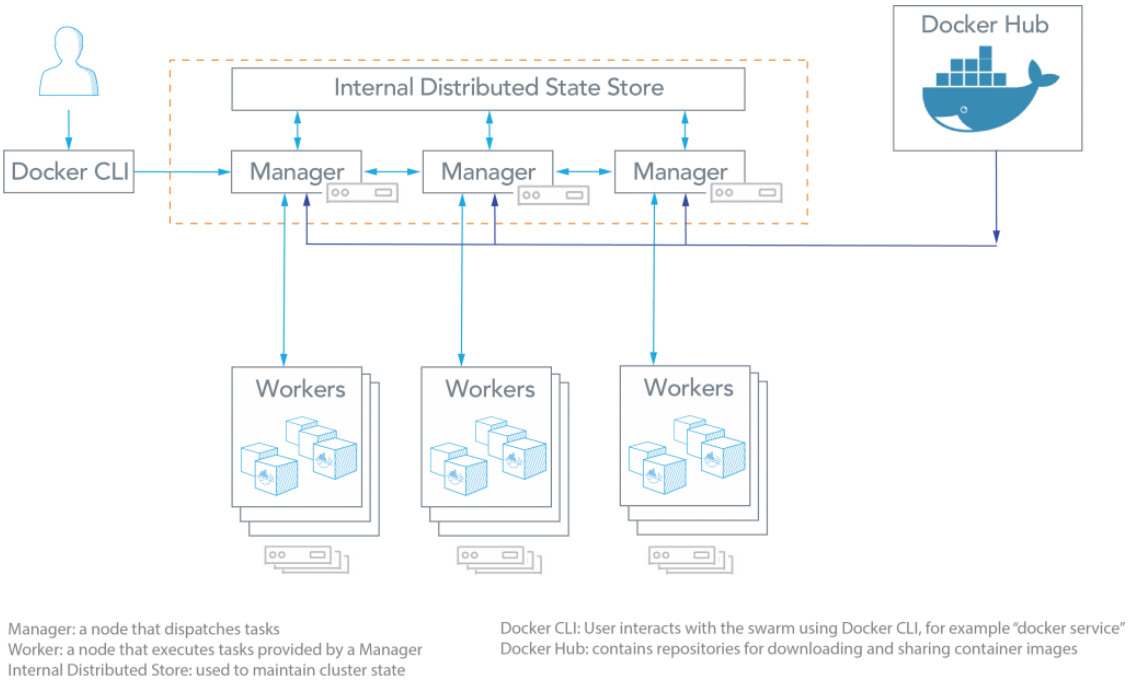

Рисунок 1.11 – Схема Docker Swarm кластера

Kubernetes є проектом з відкритим вихідним кодом, призначеним для управління кластером контейнерів Linux як єдиною системою.

Складові Kubernetes:

– Nodes (node.md): Нода це машина в кластері Kubernetes;

– Pods (pods.md): Pod це група контейнерів із загальними розділами, що запускаються як єдине ціле;

– Replication Controllers (replication-controller.md): replication controller гарантує, що певна кількість «реплік» pod'и будуть запущені в будь-який момент часу;

– Services (services.md): Сервіс в Kubernetes це абстракція яка визначає логічний об'єднаний набір pod і політику доступу до них;

– Volumes (volumes.md): Volume (розділ) це директорія, можливо, з даними в ній, яка доступна в контейнері;

– Labels (labels.md): Label'и це пари ключ / значення які прикріплюються до об'єктів, наприклад pod'ам. Label'и можуть бути використані для створення і вибору наборів об'єктів;

– Kubectl Command Line Interface (kubectl.md): kubectl інтерфейс командного рядка для управління Kubernetes.

Запущений кластер Kubernetes включає агент кубелету та допоміжні компоненти (API, планувальник тощо) для розподіленого рішення сховища. Наступна діаграма показує бажаний стан речей, хоча деякі речі все ще працюють, наприклад, як змусити кубелет (усі компоненти) працювати незалежно в контейнері, що зробить планувальник на 100% підключуваним.

Порівнюючи технічні можливості Kubernetes та Docker Swarm, ви можете скласти таблицю (таблиця 1.2):

Таблиця 1.2 – Порівняння технічних можливостей Kubernetes та Docker Swarm

Kubernetes	Docker Swarm
Визначення застосунку	
Додаток можна реалізувати, використовуючи комбінацію підлог, столів та послуг. Капсула – це група ємностей і атомна частина епіляції. Реалізація може мати сигнали на різних вузлах. Служба є зовнішньою частиною контейнера і інтегрована в DNS щодо вхідних запитів.	Додаток може бути реалізований як послуга кластера. Багато додатків контейнерів задаються за допомогою файлів синтаксису YAML. Смуги (службові екземпляри в кожній примітці кластера) можна розподілити в центрі обробки даних, випрямляючи мітки.
Масштабованість застосунків	
Кожен рівень програми визначається як підрозділ і його можна масштабувати, використовуючи управління реалізацією, вказане в YAML. Масштабування може бути ручним або автоматичним.	Послуги можна масштабувати за допомогою Docker Compose YAML-шаблонів. Послуги можуть бути глобальними або тиражуватися.

Продовження таблиці 1.2 – Порівняння технічних можливостей Kubernetes та Docker Swarm

Мережа	
Мережева модель – це єдина мережа, яка дозволяє всім компаніям спілкуватися один з одним. Мережева політика вказує, як стручки взаємодіють між собою. Модель вимагає двох CIDR, один з яких отримує IP-адресу, а другий для послуг.	Вставлення вузла в кластер Docker Swarm створює мережу накладання служб, яка охоплює всі хости Swarm і лише хост-контейнер Docker для контейнерів.
Health Checks (перевірка роботи застосунку)	
Існує два типи випробувань (техніко-економічне обгрунтування): доцільність (це відповідь на обслуговування) та підготовка (достатня для відповіді на обслуговування, але переживає і не може обслуговувати).	Немає вбудованих інструментів.
Зберігання (сховище)	
Два типи сховищ: Перший містить абстракції для окремих резервних копій (наприклад, NFS, AWS EBS, cef, flocker). Другий забезпечує абстракцію для запиту ресурсу пам'яті (наприклад, 8 Гб), який можна запустити з різними резервними копіями.	Встановлення контейнера для підтримки Docker Engine і Swarm. Системи обміну файлами, включаючи NFS, iSCSI і волокна, можуть бути налаштовані для вузлів.

2 АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ

Спочатку розглянемо основи тестування Kubernetes кластера. Аналіз ринку показав, що фактично потенційні конкуренти відсутні. Але є досить багато систем, які досліджують Kubernetes кластер з різним функціоналом та призначенням. Розглянемо деякі з них.

2.1 Тестування Kubernetes кластера

Моніторинг поточного стану програми – один із найефективніших способів передбачити проблеми та виявити вузькі місця у виробничому середовищі. Однак це наразі є однією з найбільших проблем, з якою стикаються майже всі організації з розробки програмного забезпечення. Терміни, недосвідченість, культура та управління – це лише деякі перешкоди, які можуть вплинути на успішність команд у подоланні цього завдання [7].

Зростаюче прийняття мікросервісів робить ведення журналів та моніторинг складнішими, оскільки велика кількість програм, що розповсюджуються та урізноманітнюються за своєю природою, спілкуються між собою. Один момент відмови може зупинити весь процес, але виявити його стає все складніше.

Моніторинг, звичайно, є лише одним із завдань, що ставлять мікросервіси. Обробка доступності, продуктивності та розгортання підштовхує команди створити або використовувати оркестрових інструментів для обробки всіх служб і серверів. Існує кілька інструментів оркестрації кластерів, але Kubernetes (K8S) стає все популярнішим порівняно зі своїми конкурентами. Інструмент для оркестровки контейнерів, такий як Kubernetes, обробляє контейнери в декількох комп'ютерах і усуває складність обробки розподіленої обробки.

Існує так багато змінних, щоб слідкувати за тим, що нам потрібні нові інструменти, нові методи та нові методи для ефективного фіксації важливих даних.

Оскільки Kubernetes – найпопулярніший в даний час контейнерний оркестратор, Він офіційно доступний у великих хмарах, наданих Google, Azure та, з

недавніх пір, AWS, і він може працювати в локальному, голому металевому центрі обробки даних. Навіть Докер прийняв Kubernetes і зараз пропонує його як частину своїх пакетів.

Існує кілька показників Kubernetes для моніторингу. Їх можна розділити на два основні компоненти: (1) моніторинг самого кластера та (2) моніторинг подів.

Моніторинг кластерів. Для моніторингу кластерів метою є моніторинг здоров'я всього кластеру Kubernetes. Як адміністратору, цікаво дізнатися, чи всі вузли кластера працюють належним чином і з якою потужністю, скільки застосунків працює на кожному вузлі та використання ресурсів усього кластеру.

Використання ресурсів вузлів – у цій галузі є багато показників, які стосуються використання ресурсів. Приклади цього є пропускна здатність мережі, використання диска, використання процесора та пам'яті. Використовуючи ці показники, можна з’ясувати, збільшувати чи зменшувати чи зменшувати кількість та розмір вузлів у кластері чи ні.

Кількість вузлів – кількість доступних вузлів є важливим показником, якого слід дотримуватися. Це дозволяє розібратися, за що платимо та виявити, для чого використовується кластер. Запуск подів – кількість запущених подів покаже, чи достатня кількість доступних вузлів і чи зможуть вони обробляти все робоче навантаження у випадку виходу з ладу [8].

2.2 Тестування навантаженням Kubernetes кластера

Тестування навантаження – це тестування продуктивності, яке визначає продуктивність системи в умовах реального навантаження. Це тестування допомагає визначити, як поводиться програма, коли кілька користувачів одночасно отримують доступ до неї.

Це тестування зазвичай ідентифікує – максимальна операційна ємність програми, чи достатня поточна інфраструктура для запуску програми, стійкість програми щодо пікового навантаження користувача, кількість одночасних

користувачів, які програма може підтримувати, та масштабованість, щоб дозволити більшій кількості користувачів отримати доступ до неї [8].

Це тип нефункціонального тестування. В інженерії програмного забезпечення тестування навантаження зазвичай використовується для клієнта / сервера, веб-додатків – як в Інтранеті, так і в Інтернеті.

Необхідність тестування навантаження – деякі надзвичайно популярні сайти зазнали серйозних простоїв, коли отримують величезний обсяг трафіку. Веб-сайти електронної комерції інвестують значні кошти в рекламні кампанії, але не в тестування навантаження, щоб забезпечити оптимальну роботу системи, коли цей маркетинг приносить трафік. Тестування навантаження дає впевненість у системі та її надійності та продуктивності.

Тестування навантаження допомагає виявити вузькі місця в системі за важких сценаріїв напруги для користувачів, перш ніж вони трапляться у виробничому середовищі. Тестування навантаження забезпечує чудовий захист від низької продуктивності та містить додаткові стратегії управління продуктивністю та моніторингу виробничого середовища.

Цілі тестування навантаження, завантаження тестування визначає такі проблеми перед переміщенням програми на ринок або виробництво:

– час відповіді на кожну транзакцію;
– продуктивність компонентів системи під різними навантаженнями;
– продуктивність компонентів бази даних при різних навантаженнях;
– затримка мережі між клієнтом і сервером;
– проблеми дизайну програмного забезпечення;
– питання налаштування сервера, такі як веб-сервер, сервер додатків, сервер баз даних тощо;
– проблеми з обмеженням обладнання, такі як максимізація процесора, обмеження пам'яті, вузьке місце тощо;

– тестування навантаження визначатиме, чи потребує вдосконалення системи або необхідна модифікація апаратного та програмного забезпечення для підвищення продуктивності.

Перед початком тестування навантаження потрібно створити середовище:

– конфігурація програмного забезпечення апаратної платформи;
– серверні машини;
– процесори;
– пам'ять;
– зберігання диска;
– конфігурація завантажувальних машин;
– конфігурація мережі;
– операційна система;
– серверне програмне забезпечення.

Головний показник для тестування навантаження – це час відгуку.

Стратегії тестування навантаження:

Існує велика кількість способів проведення тестування навантаження. Нижче наведено кілька стратегій тестування навантаження.

Тестування вручну навантаження: Це одна з стратегій виконання тестування навантаження, але вона не дає повторюваних результатів, не може забезпечити вимірюваного рівня напруги для програми та є неможливим координувати процес.

Засоби для перевірки навантаження з відкритим кодом: Є кілька інструментів для перевірки навантаження, які доступні у вигляді відкритого джерела, які є безкоштовними.

Інструменти перевірки навантаження для корпоративного класу: вони зазвичай постачаються із засобом захоплення / відтворення. Вони підтримують велику кількість протоколів. Вони можуть імітувати надзвичайно велику кількість користувачів.

Як зробити тестування навантаження

Процес тестування навантаження може бути коротко описаний нижче:

– створити спеціальне тестове середовище для тестування навантаження;
– завантажити тестові сценарії;
– визначити операції тестування навантаження для програми.

2.3 Дослідження існуючих систем

2.3.1 PowerfulSeal

Система хаотичного тестування кластеру, яка допомагає виявити різноманітні проблема інфраструктури якомога раніше. Цей застосунок заснований на основі принципів хаотичної інженерії (хаос рефлектує поведінку реального світу, орієнтація на результат системи, а не на внутрішні атрибути, запускати експерименти на production середовищі, автоматизація для запуску безперервно, мінімізація радіусу вибуху). Додаткові можливості програми дозволяють запускати систему у будь-якій варіації, як всередині кластеру, так і з зовні.
Проте у даної технології є недоліки – таке тестування є відносно некерованим. Також таке тестування потрібно запускати на production кластері, і якщо інженер обрав неправильний час тестування, що збігається з великим навантаженням з боку користувачів, то кластер може досить сповільнити обробку запитів за рахунок ще більшого навантаження. Дана система не може бути використана на етапі проектування кластера, тому що не дасть потрібних результатів [10]. Хаотичне тестування не відноситься до тестування до навантаження, воно допоможе знайти тільки недоліки конфігурації кластера, які не стосуються ресурсів на його вузлах. Наприклад, хаотичне тестування може знайти проблеми у цілісності побудови або у неправильній конфігурації системних об'єктів.

2.3.2 Kube-monkey

Це аналог попередньої системи, яка також базується на хаотичному тестуванні. Він випадковим чином видаляє Kubernetes (k8s) под в кластері,

заохочуючи та підтверджуючи розвиток сервісів, стійких до збоїв. Сервіс має такі ж недоліки, що і попередній. Проте порівнюючи з попереднім має ряд переваг – інформативна документація, більш кероване тестування, вибір планування запуску.

kube-monkey налаштовується змінними середовища або файлом toml, розміщеним у /etc/kube-monkey/config.toml і очікує існування конфігураційної карти для розгортання kube-monkey [12].

Ключі конфігурації та описи можна знайти в config/param/param.go

Приклад конфігураційного файла:

```
[kubemonkey]
dry_run = true       # Не видаляти поди, тільки вести журнал логів
run_hour = 8         # Запускати систему о 8 годині
start_hour = 10      # Не видаляти поди раніше 10 години
end_hour = 16        # Не видаляти поди пізніше 16 години
blacklisted_namespaces = ["kube-system"] # Critical apps live here
time_zone = "America/New_York"  #Задання часової зони
```

2.3.3 perf-tests/clusterloader2

Система запускає тестування навантаження Кубернетис кластера – тобто запускає всередині кластеру відповідну кількість сервісів, і виводить результат. Для того щоб запустити систему потрібно створити конфігураційні файли [13].

Ці параметри необхідні для виконання будь-якого тесту:

– kubeconfig – шлях до файлу kubeconfig;

– testconfig – шлях до тестового конфігураційного файла. Цей прапор можна використовувати декілька разів, якщо потрібно виконати більше одного тесту.

Параметри, які можуть бути вказані за бажанням:

– nodes – кількість вузлів у кластері. Якщо це не передбачено, тест призначить кількість вузлів кластера, які можна запланувати;

– report-dir – шлях до каталогу, де слід зберігати файли резюме. Якщо не вказано, резюме друкуються до стандартного журналу;

– provider – постачальник кластерів, варіанти: gce, gke, kubemark, aws, local, vsphere, скелет;

– masternаme – назва головного вузла;

– masterip – ім'я DNS / IP головного вузла;

– testoverrides – шлях до файлу з переопределеннями.

Запуск відбувається однією командою:

```
go run cmd/clusterloader.go --kubeconfig=kubeConfig.yaml --testconfig=config.yaml
```

Така система має не дуже інформативний звіт та не може бути запущена без додаткових конфігурацій, що ускладнює її запуск.

3 РОЗРОБЛЕННЯ СИСТЕМИ АНАЛІЗУ ПРОДУКТИВНОСТІ КЛАСТЕРА

Даний розділ присвячений розробленню програмного продукту – система аналізу продуктивності Kubernetes кластера. Далі наведені обгрунтування у виборі засобів розробки для системи, а також короткий опис роботи.

3.1 Аналіз вимог до Kubernetes кластера

Обмеження Kubernetes кластера наведені у даному розділі. На версії v1.16 Kubernetes підтримує кластери до 5000 вузлів. Більш конкретно, він підтримує конфігурації, які відповідають усім наступним критеріям:

- не більше 5000 вузлів;
- не більше 150000 загальних подів;
- не більше 300000 контейнерів;
- не більше 100 подів на вузол.

Кластер – це сукупність вузлів (фізичних чи віртуальних машин), що працюють агентами Kubernetes, якими керує «майстер» (площина управління рівнем кластера).

Вимоги до апаратного забезпечення – це одна або кілька машин, що працюють на одному з:

- Ubuntu 16.04+;
- Debian 9;
- CentOS 7;
- RHEL 7;
- Fedora 25/26;
- HypriotOS v1.0.1 +;
- Container Linux (тестовано з 1800.6.0).

Мінімальна потрібна пам'ять та процесор (ядра):

Мінімальна необхідна пам'ять основного вузла – 2 Гб, а робочий вузол – мінімум 1 Гб.

Головний вузол потребує щонайменше 1.5, а робочий вузол потребує щонайменше 0,7 ядер.

Налаштування кластерів. Для управління кластером потрібно встановити kubeadm, kubelet та kubectl:

– kubeadm: команда завантажувати кластер;

– kubelet: компонент, який працює на всіх машинах у вашому кластері та робить такі дії, як запуск подів і контейнерів;

– kubectl: інструмент командного рядка для спілкування з вашим кластером.

Перед встановленням цих пакетів необхідно виконати деякі передумови.

3.2 Аналіз вимог до системи

Розроблення будь-якої системи та продукту, в тому числі й будь-якого програмного продукту починається зі складання вимог до сутності, що розробляється.

Вимоги до проектованої системи являють собою сукупність тверджень про атрибути, властивості, або якості програмної системи, що підлягають реалізації в майбутньому.

Говорячи конкретно про систему, слід обговорити, що вона повинна вміти для того, щоб досягнути рішення поставлених проблем в даній дипломній роботі.

На вхід системи буде подавитись набір конфігурацій для тестування. На виході система повинна згідно результатів видавати звіт відносно кожного набору тестування

Для кращого формулювання вимог розділимо їх на функціональні та нефункціональні. До перших віднесемо ті, що стосуються конкретно набору функцій, власне алгоритмів роботи та поведінки системи. До останніх віднесемо ті, що стосуються характеру поведінки системи (бізнес-правила, атрибути якості, зовнішній вигляд і тд).

До категорії функціональних вимог віднесемо:

– можливість задання різних наборів тестування;

– можливість автоматизованого запуску;

– можливість відображати результат аналізу у вигляді заповненого файлу;

– можливість запуску без додаткової конфігурації.

Нефункціональні вимоги:

– можливість використання на різних Kubernetes кластерах незалежно від того, де кластер розгорнутий;

– програма повинна мати інформативний звіт після виконання аналізу.

3.3 Сценарії використання системи

У додатку И зображена діаграма використання. Зміст даної діаграми полягає в тому, що створена система подана у вигляді акторів, що взаємодіють зі системою за допомогою різних можливих варіантів використання. Варіант використання використовують для опису функції, які система пропонує. Іншими словами, кожен варіант використання визначає деякий набір дій, який виконує система при діалозі з актором. При цьому нічого не описується про те, яким чином буде реалізована взаємодія акторів із системою.

Розглянемо декілька сценаріїв використання системи:

– компанія переходить з монолітної архітектури на мікросервіси та було обрано використати Kubernetes як оркестратор контейнерів. Монолітний застосунок зараз має певне навантаження, яке є відомим для розробників. Застосунок розгорнутий на одному інстансі з певними ресурсами. Постає питання яка конфігурація Kubernetes сервера є оптимальною для мікросервісного застосунку. Адміністратори створюють тестовий кластер та для того щоб переконатися, що він буде відповідати майбутньому навнтаженню, можуть запустити розроблену систему;

– існує кластер і потрібно взнати пікове можливе навантаження кластера, тоді можна запустити систему з максимальним набором подів;

– існує кластер і потрібно зрозуміти, що буде відбуватися при пікових навантаженнях – чи буде кластер функціонувати, чи зламається;

– існує кластер та опція масштабування включена, потрібно перевірити як швидко буде відбуватися масштабування та чи працює воно як потрібно.

3.4 Вибір та обґрунтування засобів розробки

Для реалізації програми, яка може відповідати вашим вимогам, ви повинні визначити мову програмування, якою вона фактично написана. Йдеться про вибір найзручнішого, який у рамках цього проекту виявився досить важким та приніс певні труднощі через відсутність досвіду в галузі впровадження додатків, пов’язаних із машинним навчанням. Очевидно, найпростішим способом є використання існуючих бібліотек, які вже мають в собі методи та підходи, що має значно скоротити час, витрачений на написання програми.

У цьому розділі буде коротко проаналізовано та обґрунтовано вибір мови програмування, існуючих бібліотек з підготовленими рішеннями, вибір середовища розробки та інших додаткових інструментів. Усі висновки щодо вибору рішень ґрунтуватимуться на огляді популярності, зворотного зв’язку, певних кількісних показників та перспектив розвитку та додаткових потенційних удосконалень.

3.4.1 Вибір архітектури для системи

Враховуючи попереднє порівняння, ми створимо таблицю для порівняння монолітів та мікросервісів. Нехай кожен із критеріїв оцінки, необхідних для застосування дипломного проекту, зважується – 2 (0 – критерій взагалі не описує тип архітектури, 1 – критерій частково задоволений, 2 – критерій повністю відповідає типу архітектури). Тоді таблиця буде виглядати приблизно так (табл. 3.1):

Таблиця 3.1 – Порівняння типів архітектур

	Моноліт	Мікросервіси
Швидке внесення змін	0	2
Безпроблемне розгортання	1	2
Легкість підтримки/тестування	1	2
Різноманітність технологій	0	2
Зміни/проблеми однієї частини проекту не тягнуть за собою інші	0	2
Загальне	2	10

Отже, згідно з таблицею видно, що найкраще підходить мікросервісна архітектура для створення системи.

3.4.2 Вибір мови програмування

Дана система складається з трьох компонентів: веб-серверу, генератора навантаження та агрегатора, проте кожен компонент був побудований на Go з використанням різних бібліотек.

Go – компільована мова програмування із вбудованими засобами для паралельних обчислень і засобами віддаленого керування пакунками. Цю мову програмування розробив Google як частину проекту з розробки операційної системи Inferno.

Для цілей проекту Go з'явилося бажання отримати мову, яка поєднала швидкість розвитку з високою продуктивністю компільованих мов з легкістю писати коди та захищати від помилок, властивих мовам сценаріїв. Синтаксис Go базується на звичайних елементах C, де є деякі позики Python. Мова досить проста та коротка, але код все ще легко читати та читати. Спочатку проект був розроблений з упором на паралельне програмування та ефективну роботу багатоядерних систем, включаючи надання інструментів рівня оператора для паралельних обчислень та сумісності.

Мова надає інтегровані рішення для доступу до дозволених областей виділених блоків пам'яті та дозволяє використовувати сміття. Код мови Go зібраний в окремі бінарні файли, які функціонують спочатку без використання віртуальної машини (створення профілю, модулі налагодження та інші підсистеми усунення несправностей інтегровані як компоненти часу виконання), що дозволяє досягти продуктивності, порівнянної з програмами C.

Синтаксис Go досить схожий на синтаксис C: кодові блоки в дужках; Загальна структура управління виконанням програми включає if, для та перемикання. На відміну від C, крапка з комою в кінці рядка є необов’язковою; декларування змінних відрізняється; перетворення типів є суворим; Для підтримки паралельного програмування були введені нові інструкції переходу та вибору. Нові вбудовані типи включають хеш-таблиці, рядки Unicode, вирізані матриці та канали для передачі даних між процесами [14].

Go призначений для швидкої компіляції навіть на застарілому апаратному забезпеченні. Ця мова підтримує збирання сміття. Чітка структура Go, орієнтована паралельно (канали є альтернативою каналам введення даних) запозичена з послідовних комунікаційних процесів Тоні Гоар. На відміну від попередніх мов програмування, орієнтованих паралельно, таких як оккам чи лімбо, Go не надає будь-якої інтегрованої підтримки для попереджень про безпеку чи перевірки сумісності. На сьогодні Go не має вбудованої підтримки шаблонів, але її можна додати в майбутньому.

З функцій, доступних в C ++ або Java, Go не включає успадкування типів, загальне програмування (шаблон), твердження, арифметику вказівника. Письменники Go наголошують на відкритому програмуванні, прямо протиставляючи претензії та вказуючи на арифметику, прагнучи дозволити успадкування типів для зручності. Спочатку мова не включала обробку винятків, але в березні 2010 року було застосовано механізм, відомий як паніка / відновлення для обробки помилок та винятків, щоб уникнути проблем, що виникають у авторів.

Видимість функцій поза файлом, де вони визначені, неявно визначається великістю їхніх ідентифікаторів, на відміну від C ++, де використовується загальнодоступне ключове слово.

Незважаючи на це, golang є мовою програмування загального призначення. Це означає, що будь-яку програму можна написати.

Є декілька основних причин почати використовувати Go:

– він статичний, чудовий спосіб обробляти помилки, це робить Golang більш надійним і надійним;

– компілюється до одного бінарного файла;

– швидкий;

– простий, не треба писати величезні коментарі у коді, щоб пояснити іншим розробникам, що робить частину програми;

– легкий рівень входу, легко можна адаптуватися до написання програм.

3.4.3 Вибір середовища розробки

Як середовище розробки було обрано Visual Studio Code. Visual Studio Code – засіб для створення, редагування та зневадження сучасних веб-застосунків і програм для хмарних систем. Visual Studio Code розповсюджується безкоштовно і доступний у версіях для платформ Windows, Linux і OS X (рис. 3.1).

Корпорація Microsoft представила Visual Studio Code в квітні 2015 року в Build 2015. Це середовище розробки стало першим крос-платформним продуктом у лінійці Visual Studio.

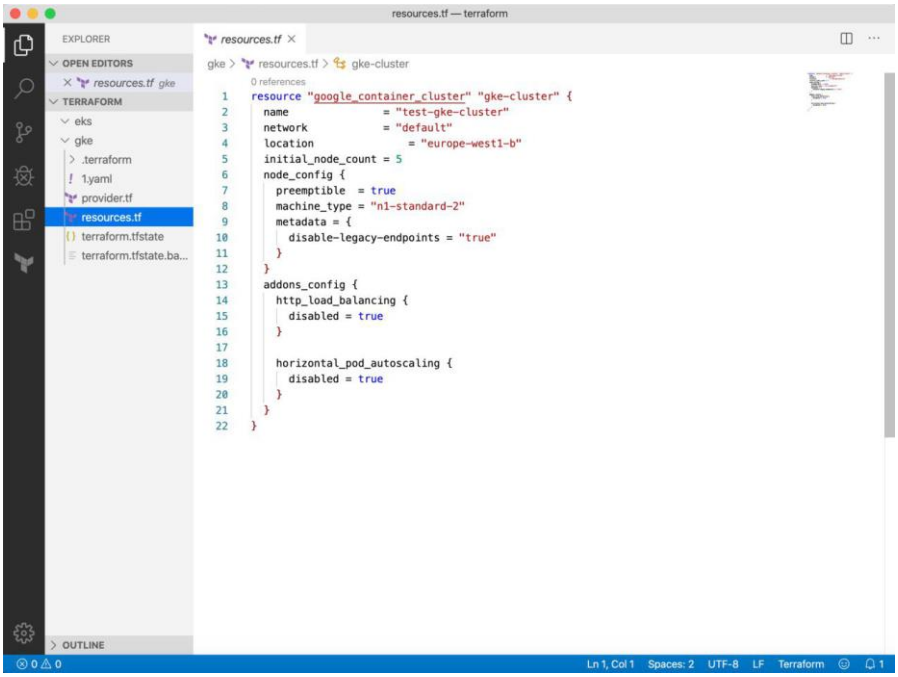

Рисунок 3.1 – середовище розробки

Основою Visual Studio Code є розробка безкоштовного проекту Atom, розробленого GitHub. Зокрема, Visual Studio Code є надбудовою для Atom Shell, який використовує двигун браузера Chromium та Node.js. Варто зазначити, що використання безкоштовного проекту Atom не згадується на веб-сайті Visual Studio Code, у прес-релізі чи в офіційному блозі.

Редактор включає вбудований осушувач повітря, інструменти для роботи з засобами Git та рефакторингу, навігацію по коду, автозаповнення типових конструкцій та контекстну допомогу. Продукт підтримує розробку для платформ ASP.NET і Node.js і позиціонується як полегшене рішення, що дозволяє позбутися

повністю інтегрованого середовища розробки. Серед підтримуваних мов та технологій: JavaScript, C ++, C #, TypeScript, jade, PHP, Python, XML, Batch, F #, DockerFile, Coffee Script, Java, HandleBars, R, Objective-C, PowerShell, Luna, Visual Basic, Markdown, JSON, HTML, CSS, LESS і SASS, Haxe та багато іншого.

3.4.4 Засоби для автоматизації додатку

Для автоматизація розгортання сервісів системи у кластері було використано Helm. Helm – це пакетний менеджер Kubernetes. Helm – це величезний зсув у способі визначення, зберігання та управління програмами на стороні сервера. Використання Helm цілком може стати ключем до масового прийняття мікросервісів, оскільки використання цього менеджера пакунків значно спрощує їх управління (рис 3.2).

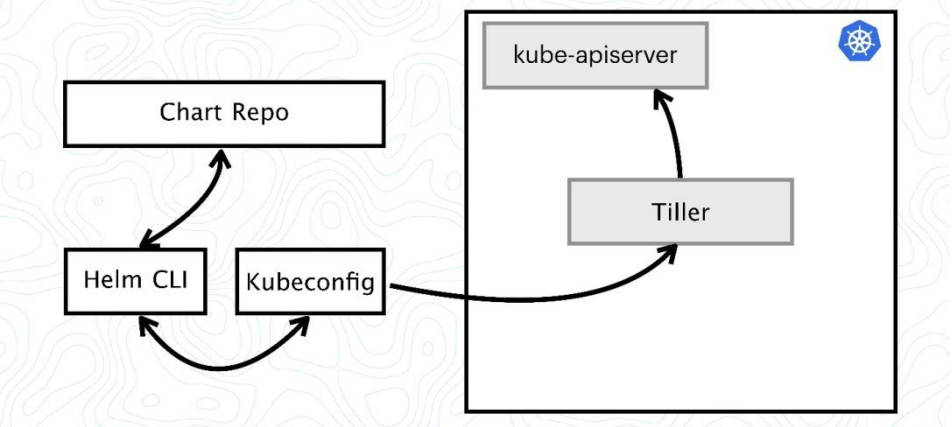

Рисунок 3.2 – Схема Helm

Наприклад, потрібно запустити веб-сайт, створений за допомогою WordPress, Joomla, Django або будь-якої іншої CMS. Веб-сайт буде отримувати мільйони щоденних відвідувачів з першого дня, і треба переконатися, що така величезна кількість з'єднань не призведе до заморожування або недоступності служби.

Використання можливостей віртуалізації забезпечує масштабування, так. Зауважте, що AMI, ARM (або контейнер Docker для цього питання), який використовується для запуску програми, буде залежати від віртуальної машини, на якій він зберігається, і зможе масштабувати лише те, як масштабуються віртуальні машини – додавши більше ресурсів у пул.

З Helm зовсім інша картина. Додаток може складатися з чітко визначених мікросервісів, і можна масштабувати лише ті, що нам потрібно масштабувати, додаючи в кластер більше вузлів і подів Kubernetes. Замість того, щоб працювати з цілісним зображенням і вирощувати всі ресурси, відбувається керування набором зображень і самостійне масштабування.

Проблеми починаються, коли треба запустити новий екземпляр програми, яка працює, скажімо, 50 мікросервісів. Запустити та поєднувати їх все буде складно. Однак, із Helm, все, що потрібно знати, – це назва діаграм відповідальних зображень. Запуск нового примірника – це питання до Helm.

Досі єдиною суттєвою проблемою щодо Helm було те, що коли два штурмових діаграми мають однакові мітки, вони заважають один одному та створюють проблеми у основних ресурсів. Це означає, що краще створити нове зображення для проекту, ніж додати до нього одну схему Helm, і це також вплине на відкати.

Результатом Helm є Helm charts (діаграми). Створюючи та розгортаючи програми, Helm Charts надає можливість використовувати пакети Kubernetes за допомогою натискання кнопки або однієї команди CLI.

"Helm" позначає як клієнтські, так і серверні компоненти Kubernetes: клієнт Helm та сервер Tiller / Helm. Клієнт взаємодіє з сервером для виконання змін у кластері Kubernetes.

Коли користувач виконує команду helm install, сервер tiller отримує вхідний запит і встановлює відповідний пакет – який містить визначення ресурсів для встановлення програми Kubernetes – у кластер Kubernetes. Позначені як "Діаграми", ці пакети схожі на пакети RPM та DEB для Linux. Вони надають зручний спосіб розробникам поширювати програми, а кінцевим користувачам встановлювати ці програми.

Серед понад 100 доступних в Інтернеті стабільних Helm Charts включають:

– MySQL та MariaDB: популярні сервери баз даних, якими користуються Вікіпедія, Facebook та Google;

– MongoDB: крос-платформа, орієнтована на документ, база даних NoSQL;

– WordPress: видавнича платформа для створення блогів та веб-сайтів.

Також розгортання системи може бути автоматизоване за рахунок створення пайплайну у будь-якому сервісі, наприклад – Jenkins.

3.4.5 Додаткові засоби розробки

Для простоти запуску системи було створено Makefile. Якщо запускати або оновлювати завдання, коли певні файли оновлюються, утиліта make може стати в нагоді. Утиліта make вимагає файл Makefile (або makefile), який визначає набір завдань, які потрібно виконати. Більшість проектів з відкритим кодом використовують для складання остаточного виконуваного двійкового файлу, який потім можна встановити за допомогою make install.

У даному випадку make використовується для полегшення запуску системи, отримання результатів, переривання виконная, отримання проміжного статусу виконання.

Приклад Makefile:

```
apply:
  helm --name kubeloader install --namespace kubeloader .
stop:
  helm delete --purge kubeloader
get:
  kubectl logs aggregator -n kubeloader | grep Results
```

У прикладі є три директиви:

- make – запуск системи у кластері;
- stop – видаляє систему повністю з кластера;
- get – отримання звіту з результатами.

3.4.6 Автоматизація розгортання Kubernetes кластера у різних середовищах

Для автоматизації розгортання було використано Terraform. Terraform – це інструмент для безпечної та ефективної побудови, зміни та модернізації

інфраструктури. Terraform може керувати існуючими та популярними постачальниками послуг, а також індивідуальними внутрішніми рішеннями.

Файли конфігурації описують Terraform компоненти, необхідні для запуску однієї програми або всього центру обробки даних. Terraform формує план виконання, описуючи, що він буде робити для досягнення бажаного стану, а потім виконує його для створення описаної інфраструктури. По мірі зміни конфігурації Terraform здатний визначити, що змінилося, та створити додаткові плани виконання, які можна застосувати.

Інфраструктура, яку Terraform може керувати, включає компоненти низького рівня, такі як обчислювальні екземпляри, сховище та мережу, а також компоненти високого рівня, такі як записи DNS, функції SaaS тощо.

Основними особливостями Terraform є:

1. Інфраструктура як код. Інфраструктура описується за допомогою синтаксису конфігурації високого рівня. Це дозволяє розробити план вашого центру обробки даних і поводитись так, як і з будь-яким іншим кодом. Крім того, інфраструктуру можна спільно використовувати та використовувати повторно.

2. Плани виконання. Terraform має крок "планування", де генерує план виконання. План виконання показує, що Terraform буде робити, коли ви запустите. Це дозволяє уникнути будь-яких сюрпризів, коли Terraform маніпулює інфраструктурою.

3. Графік ресурсів. Terraform створює графік усіх ваших ресурсів і паралелізує створення та модифікацію будь-яких незалежних ресурсів. Через це Terraform будує інфраструктуру максимально ефективно, і оператори отримують уявлення про залежності в їх інфраструктурі.

4. Автоматизація змін. Складні набори змін можуть бути застосовані до вашої інфраструктури при мінімальній взаємодії з людьми. Згаданий раніше план виконання та графік ресурсів допомагає зрозуміти, що Terraform зміниться та в якому порядку, уникаючи багатьох можливих людських помилок.

Приклад автоматизації створення архітектури за допомогою Terraform:

```
resource "google_container_cluster" "gke-cluster" {
```

```
  name               = "test-gke-cluster"
  network            = "default"
  location               = "europe-west1-b"
  initial_node_count = 5
  node_config {
    preemptible  = true
    machine_type = "n1-standard-2"
    metadata = {
      disable-legacy-endpoints = "true"
    }
  }
  addons_config {
    http_load_balancing {
      disabled = true
    }
    horizontal_pod_autoscaling {
      disabled = true
    }
  }
}
```

У прикладі створюється Kubernetes кластер у Google Cloud. Кластер складається з 5 вузлів, кожен з яких має n1-standard-2 тип інстансу та розміщений у europe-west1-b зоні.

3.5 Короткий опис програми

Система складається з трьох компонентів:

- агрегатор;
- генератор навантаження;
- веб-сервер.

Розглянемо кожен компонент окремо.

Агрегатор створює та керує сервісами для інших компонентів. Також він збирає та фіксує результати виконання та записує у лог.

У конфігурації агрегатора є стандартний набір сценаріїв:

```
var scenarios = []replicas{
        {
                title:     "No load",
                loadgenerator:  1,
                webserver: 1,
        },
        {
                title:     "Light load",
                loadgenerator:  1,
                webserver: 10,
        },
        {
                title:     "Basic load",
                loadgenerator:  5,
                webserver: 10,
        },
        {
                title:     "Normal load",
                loadgenerator:  10,
                webserver: 10,
        },
        {
                title:     "Heavy load",
                loadgenerator:  30,
                webserver: 10,
        },


        {
                title:     "Huge load",
                loadgenerator:  50,
                webserver: 20,
        },
}
```

Тобто агрегатор виступає оркестратором для тестових сценаріїв.

Генератор має задачу створювати запити за певний період часу. Він базується на зовнішній бібліотеці Vegeta, яка допомагає отримати деталізовану статистику щодо запитів. Vegeta реалізує інструмент для тестування навантажень HTTP.

Веб-сервер – простий веб-сервер, котрий налаштований на отримання запитів з генератора навантаження.

Конфігурація тест сценаріїв керується відповідно тільки з агрегатора. Ряд параметрів, які можуть бути змінені (таб. 3.2):

Таблиця 3.2 – Параметри конфігурації системи

Параметр	Значення за замовчуванням	Опис
agregator.maxReplicas	50	Максимальна кількість реплік для агрегатора
cpuRequests.loadbot	100м	Запит на пам'ять кожного генератора навантаження
cpuRequests.webserver	100м	Запит на пам'ять кожного веб-сервера
loadbot.rate	100	Query per second для кожного генератора навантаження
loadbot.workers	10	Початкова кількість подів генератора навантаження

Продовження таблиці 3.2 – Параметри конфігурації системи

loadbot.duration	1c	Тривалість кожної атаки генератором на веб-сервер
images.*Version	latest	Версія Зображення для завантажувача, веб-сервера та агрегатора
imagePullPolicy	IfNotPresent	Витягувати новий образ чи ні
rbac.create	true	Створити rbac правила для агрегатора

Приклад звіту після запуску системи:

```
obilan$ make get
kubectl logs aggregator -n kubeloader | grep Results
Results of the load scenario 'No load': Success: 100.00  % Query per second: 10035    Latency mean: 1.057385ms 99th:  8.431097ms
Results of the load scenario 'Light load': Success: 100.00  % Query per second: 10026    Latency mean: 698.808µs 99th:  3.561517ms
Results of the load scenario 'Basic load': Success: 100.00  % Query per second: 50188    Latency mean: 3.02228ms 99th:  23.029807ms
Results of the load scenario 'Normal load': Success: 100.00  % Query per second: 100248   Latency mean: 20.812261ms 99th: 125.260319ms
Results of the load scenario 'Heavy load': Success: 100.00  % Query per second: 299510   Latency mean: 824.097238ms 99th: 1.871545441s
Results of the load scenario 'Huge load': Success: 100.00  % Query per second: 494139   Latency mean: 748.192385ms 99th: 2.335936925s
```

У звіті можна побачити такі параметри:

- Success rate – процентне співвідношення успішності сценарія;
- QPS (query per second) – кількість запитів за секунду;
- Latency mean – середня затримка відповіді на запит;
- Latency 99^{th} – затримка кожного 1 реквеста з 100.

Отже, система показує не тільки, що кластер впорався з певною кількістю запитів, а агрегує дані про затримки у виконанні запитів.

Діаграма Автоматизації за домомогою Helm представлена у додатку

Ж та демонструє, що система працює у середині Kubernetes кластера, там запускається користувачем з його персонального комп’ютера за допомогою конфігурацій Helm. Helm використовує файли шаблонів та файл зі значеннями пареметрів. Далі він створює ресурс, який відповідає за створення об'єктів у Kubernetes кластері, які необхідні для системи. Також у додатку Д представлена діаграма розгортання системи.

У додатку Е представлена діаграма послідовності, яка відображає взаємодію між об’єктами за часом. Діаграма демонструє, які дії відбуваються всередині системи, коли користувач робить 3 запити:

- запуск системи;
- припинення системи;
- отримання кінцевого звіту.

3.6 Структурна схема системи

Спочатку складемо спрощену схему комунікації між сервісами всередині системи.

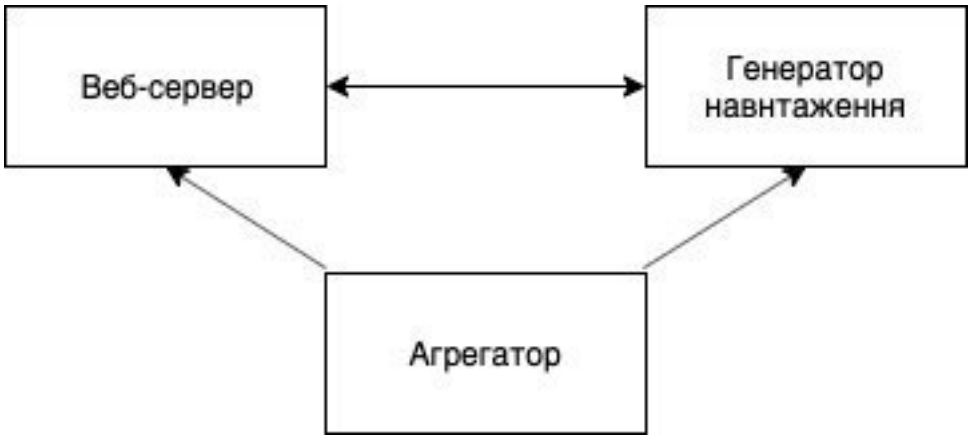

Рисунок 3.3 – Схема комунікації між мікросервісами всередині системи

З рисунка 3.3 видно, що агрегатор тільки створює веб-сервер та генератор навантаження, які безпосередньо обмінюються запитами між собою.

Для створення структурної схеми використаємо схему комунікації (рис. 3.3).

Структурна схема визначає основні функціональні частини системи, їх призначення та взаємозв'язок. Структурна схема зображена у додатку Б. Зі структурної схеми видно, що взаємодія користувача з системою відбувається через командний рядок. Усі підсистеми системи розміщені у Kubernetes кластері.

У підсистемі користувача розміщено:

– командний рядок – інтерфейс – це те, як комунікує користувач з системою. Замість командного рядка – може бути будь-яка система інтеграції з Kubernetes кластером, яка може запускати bash команду для того, щоб викликати функції системи.

– автоматизований скрипт, який отримє запити користувача з командного рядка та формує їх на запити до системи, яка встановлена у Kubernetes кластері.

У підсистемі Kubernetes кластера є три об'єкти, які можна побачити на рисунку 3.3:

1. Агрегатор:

– под – безпосередньо сервіс, який запускається у одному поді без можливості масштабування оскільки виступає контрольную панеллю системи;

– роль – роль, як об’єкт Kubernetes, яка дозволяю отримувати інформацію про поди, оновлювати та отримувати інформацію про контролери реплікації;

– сервісний акаунт – надає доступи для пода;

– пов'язання ролі з сервісним акаунтом – пов'язує сервісний акаунт та роль, за рахунок цього сервісний акаунт має усі доступи ролі.

2. Генератор навантаження:

– контролер реплікації – задає та підтримує певну кількість подів у кластері.

3. Веб-сервер:

– контролер реплікації – задає та підтримує певну кількість подів у кластері;

– сервіс – реалізує точку доступу для запитів з контролера реплікацій.

3.7 Проведення експериментів за сценаріями використання

Зробимо експеримент – запустимо стандартний набір сценаріїв для системи на не продуктивному кластері з одним інстансом.

Конфігурація кластера:

```
resource "google_container_cluster" "gke-cluster" {
  name               = "test-gke-cluster"
  network            = "default"
  location           = "europe-west1-b"
  initial_node_count = 1
  node_config {
    preemptible  = true
    machine_type = "n1-standard-2"
    metadata = {
      disable-legacy-endpoints = "true"
    }
  }
  addons_config {
    http_load_balancing {
      disabled = true
    }
```

```
        horizontal_pod_autoscaling {
          disabled = true
        }
      }
    }
```

Видно, що лише 1 інстанс у кластері з n1-standard-2 типом інстансу (2CPU, 7.5GB memory).

Кількість інстансів у створеному кластері:

```
obilan$ kubectl get nodes
NAME                                                STATUS   ROLES
AGE    VERSION
gke-test-gke-cluster-default-pool-e44a4da9-cz0l   Ready    <none>
5m3s   v1.13.11-gke.14
```

Після 15 хвилин запуску системи пройшли тільки 2 сценарії:

```
Olhas-Air:kubeloader obilan$ make get
kubectl logs aggregator -n kubeloader | grep Results
Results of the load scenario 'No load': Success: 100.00  % Query
per second: 10029    Latency mean: 656.728µs 99th:  3.856767ms
Results of the load scenario 'Light load': Success: 100.00  % Query
per second: 10038    Latency mean: 2.05063ms 99th:  22.951993ms
```

Створення подів у кластері заблоковано оскільки не вистачає ресурсів (інстансів). Більшість подів третього сценарія у статусі Pending:

```
loadgen-59s5h    0/1     Pending             0          7m9s
loadgen-dk2rp    0/1     Pending             0          7m9s
loadgen-dk2rp    0/1     Pending             0          8m9s
loadgen-59s5h    0/1     Pending             0          8m9s
loadgen-59s5h    0/1     Pending             0          9m9s
loadgen-dk2rp    0/1     Pending             0          9m9s
```

З експерименту видно, що кластер не може впоратися з навантаженням, проте великих збоїв у роботі не відбувається. Систему можна вимкнути:

```
obilan$ make stop
helm delete --purge kubeloader
release "kubeloader" deleted
```

Зробимо ще один експеримент – запустимо стандартний набір сценаріїв для системи на не продуктивному кластері з одним інстансом та включеної опцією автоматичного масштабування.

Конфігурація кластера:

```
resource "google_container_cluster" "gke-cluster" {
  name               = "test-gke-cluster"
  network            = "default"
  location           = "europe-west1-b"
  initial_node_count = 1
  node_config {
    preemptible  = true
    machine_type = "n1-standard-2"
    metadata = {
      disable-legacy-endpoints = "true"
    }
  }
  addons_config {
    http_load_balancing {
      disabled = true
    }

    horizontal_pod_autoscaling {
      disabled = false
    }
  }
}
```

Видно, що лише 1 інстанс у кластері з n1-standard-2 типом інстансу (2CPU, 7.5GB memory).

Кількість інстансів у створеному кластері на початку:

```
obilan$ kubectl get nodes
NAME                                          STATUS    ROLES     AGE
VERSION
gke-test10-default-pool-b9987e1f-1h4q    Ready      <none>    66s
v1.13.11-gke.14
```

Проміжне створення інстансів:

```
obilan$ kubectl get nodes
NAME                                          STATUS    ROLES     AGE
VERSION
gke-test10-default-pool-b9987e1f-1h4q    Ready      <none>    4m9s
v1.13.11-gke.14
gke-test10-default-pool-b9987e1f-94vk    Ready      <none>    18s
v1.13.11-gke.14
gke-test10-default-pool-b9987e1f-f04r    Ready      <none>    15s
v1.13.11-gke.14
```

Видно, що деякі поди в статусі поки чекають створення інстансів:

```
obilan$ kubectl get pods -n kubeloader
NAME              READY   STATUS    RESTARTS   AGE
aggregator        1/1     Running   0          45s
loadbots-nqtpl    1/1     Running   0          45s
webserver-5c25p   0/1     Pending   0          16s
webserver-5gjjp   1/1     Running   0          45s
webserver-6htql   0/1     Pending   0          16s
webserver-7cnv7   0/1     Pending   0          16s
webserver-b9csz   0/1     Pending   0          16s
webserver-drq9h   0/1     Pending   0          15s
webserver-fhvh8   0/1     Pending   0          16s
webserver-hbk6d   0/1     Pending   0          16s
webserver-jk755   0/1     Pending   0          16s
```

```
webserver-sfd5b   1/1     Running   0          16s
```

Фінальна масштабована кількість інстансів:

```
obilan$ kubectl get nodes
NAME                                          STATUS    ROLES     AGE
VERSION
gke-test10-default-pool-b9987e1f-1h4q    Ready      <none>    22m
v1.13.11-gke.14
gke-test10-default-pool-b9987e1f-1z74    Ready      <none>    11m
v1.13.11-gke.14
gke-test10-default-pool-b9987e1f-388d    Ready      <none>    13m
v1.13.11-gke.14
gke-test10-default-pool-b9987e1f-49p9    Ready      <none>    11m
v1.13.11-gke.14
gke-test10-default-pool-b9987e1f-94vk    Ready      <none>    18m
v1.13.11-gke.14
gke-test10-default-pool-b9987e1f-f04r    Ready      <none>    18m
v1.13.11-gke.14
gke-test10-default-pool-b9987e1f-hht3    Ready     <none>    4m25s
v1.13.11-gke.14
gke-test10-default-pool-b9987e1f-tcvn    Ready      <none>    11m
v1.13.11-gke.14
```

Під час роботи системи видно, що кластер масштабується та з'являються нові інстанси через те, що поди не має достатньо ресурсів для того, щоб бути створеними. Спочатку кластер складався з 1 інстанса, на останньому сценарії вже було 8 інстансів.

Результати:

```
kubectl logs aggregator -n kubeloader | grep Results
Results of the load scenario 'No load': Success: 100.00  % Query
per second: 10287    Latency mean: 5.292532ms 99th:  42.25233ms
Results of the load scenario 'Light load': Success: 100.00  % Query
per second: 10281    Latency mean: 4.379798ms 99th:  34.617922ms
Results of the load scenario 'Basic load': Success: 100.00  % Query
per second: 50766    Latency mean: 139.426705ms 99th:  574.784489ms
```

```
Results of the load scenario 'Normal load': Success: 100.00 % Query per second: 101156 Latency mean: 130.467265ms 99th: 717.447628ms
Results of the load scenario 'Heavy load': Success: 100.00 % Query per second: 305895 Latency mean: 793.094386ms 99th: 2.931237142s
```

Видно, що масштабування працює, та всі сценарії системи були виконані. Проте час затримки є досить високим через те, що час витрачається на очікування створення інстансів.

4 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ РОБОТИ СИСТЕМИ

Для порівняння було обрано два найбільш популярні провайдери хмарних обчислень – Amazon Web Services та Google Cloud Platform. У кожному з сервісів створимо по кластеру з однаковим набором вузлів та їх характеристик.

4.1 Elastic Kubernetes Service від Amazon Web services

Служба Amazon Elastic Kubernetes (Amazon EKS) – це керована послуга, яка дозволяє легко запускати Kubernetes на AWS, не потребуючи створення або обслуговування власної панелі управління Kubernetes. Kubernetes – це система з відкритим кодом для автоматизації розгортання, масштабування та управління контейнерними програмами.

Amazon EKS управляє екземплярами площини управління Kubernetes у кількох зонах доступності, щоб забезпечити високу доступність. Amazon EKS автоматично виявляє та замінює нездорові екземпляри площини управління, а також забезпечує автоматичне оновлення версій та виправлення для них.

Amazon EKS інтегрований з багатьма послугами AWS для забезпечення масштабованості та безпеки для програм, включаючи наступне (рис. 4.1):

- Amazon ECR для зображень контейнерів;
- Elastic Load Balancing для розподілу навантаження;
- IAM для аутентифікації;
- Amazon VPC для ізоляції.

Amazon EKS запускає сучасні версії програмного забезпечення Kubernetes з відкритим кодом, тому ви можете використовувати всі існуючі плагіни та інструменти від спільноти Kubernetes. Програми, що працюють на Amazon EKS, повністю сумісні з програмами, що працюють у будь-якому стандартному середовищі Kubernetes, незалежно від того, чи працюють у локальних центрах обробки даних чи у загальнодоступних хмарах. Це означає, що можна легко перемістити будь-який стандартний додаток Kubernetes до Amazon EKS без будь-якої зміни коду.

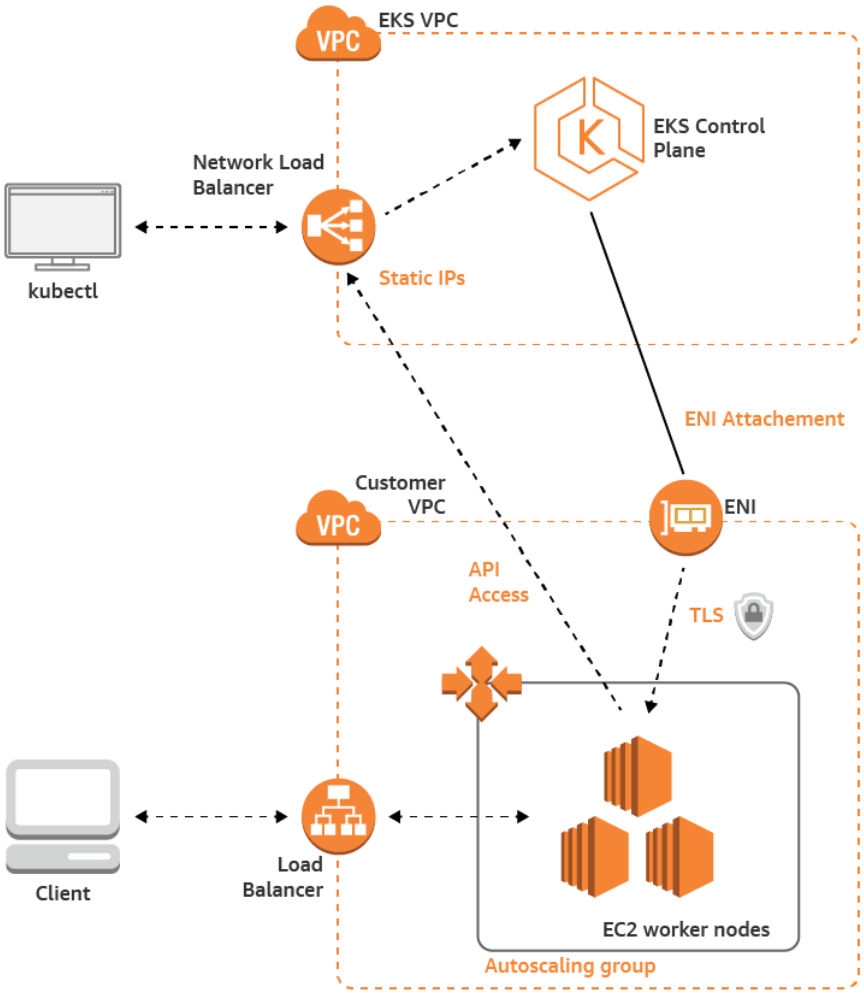

Рисунок 4.1 – Взаємодія EKS з іншими AWS сервісами

Amazon EKS управляє єдиною панелю управління Kubernetes для кожного кластеру, а інфраструктура контрольної площини не ділиться між кластерами чи обліковими записами AWS.

Ця контрольна панель складається щонайменше з двох вузлів сервера API та трьох вузлів etcd, які працюють через три зони доступності в регіоні. Amazon EKS автоматично виявляє та заміняє нездорові екземпляри площини управління, перезавантажуючи їх у зонах доступності в регіоні за потребою (рис 4.2). Amazon EKS використовує архітектуру AWS Regions, щоб підтримувати високу доступність.

Amazon EKS використовує мережеві політики Amazon VPC для обмеження трафіку між компонентами контрольної площини до одного кластеру. Компоненти площини управління для кластера не можуть переглядати або приймати зв’язок з

інших кластерів або інших облікових записів AWS, за винятком випадків, коли це дозволено політикою Kubernetes RBAC.

Ця безпечна та доступна конфігурація робить Amazon EKS надійною та рекомендованою для виробничих навантажень.

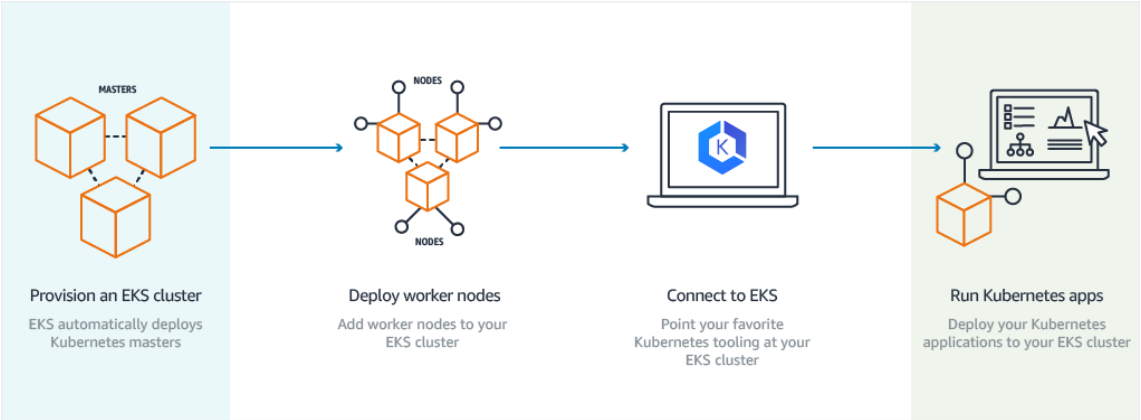

Рисунок 4.2 – Схема роботи з EKS

Для створення кластера EKS було використано Terraform модуль.

Конфігурація:

```
provider "aws" {
    region = "eu-central-1"
}

module "eks" {
  source          = "terraform-aws-modules/eks/aws"
  cluster_name    = "test-aws"
  cluster_version = "1.14"
  subnets         = ["subnet-97a00dfd", "subnet-0fbda572", "subnet-304fa07c"]
  vpc_id          = "vpc-97c207fd"
```

```
  worker_groups = [
    {
      instance_type = "m5.large"
      asg_max_size  = 5
      asg_max_size  = 5
    }
  ]
}
```

Окільки створені кластери у різних хмарних сервісах повинні бути однаковими за ресурсами, то було обрано m5.large тип для кожної з 5 інстансів, що відповідаю 2 CPU та 8 GB пам’яті. Також для створення кластеру в AWS треба виконати декілька передумов – створити VPC приватну хмару з усіма мережевими сервісами, як таблиця маршрутизації, сабнетами у різних зонах доступності. Також, якщо кластер повинен бути публічно доступним, він має мати публічну мережу. Усі попередньо створені ресурсі помічаються мітками при створенні кластера. Додаток Г демонструє деталізовану схему кластера.

Результат створеного кластера:

```
Olhas-Air:eks obilan$ kubectl get nodes
NAME      STATUS   ROLES    AGE   VERSION
ip-172-31-10-186.eu-central-1.compute.internal   Ready    <none>
76s   v1.14.7-eks-1861c5
ip-172-31-29-192.eu-central-1.compute.internal   Ready    <none>
81s   v1.14.7-eks-1861c5
ip-172-31-29-52.eu-central-1.compute.internal    Ready    <none>
82s   v1.14.7-eks-1861c5
ip-172-31-33-56.eu-central-1.compute.internal    Ready    <none>
81s   v1.14.7-eks-1861c5
ip-172-31-42-9.eu-central-1.compute.internal     Ready    <none>
81s   v1.14.7-eks-1861c5
```

Далі запускаємо систему для визначення продуктивності кластера та дивимося проміжні створені поди у тестовому неймспейсі:

```
Olhas-Air:eks obilan$ kubectl get pods -n kubeloader
NAME              READY   STATUS    RESTARTS   AGE
aggregator        1/1     Running   0          41s
loadgen-zctxm     1/1     Running   0          41s
webserver-5kcb7   1/1     Running   0          20s
webserver-8wkcw   1/1     Running   0          40s
webserver-czzlg   1/1     Running   0          20s
webserver-ddjs5   1/1     Running   0          20s
webserver-dwgg6   1/1     Running   0          20s
webserver-f2vlb   1/1     Running   0          20s
webserver-nwn8g   1/1     Running   0          20s
webserver-q4pkn   1/1     Running   0          20s
webserver-qcf2d   1/1     Running   0          20s
webserver-sgrpb   1/1     Running   0          20s
```

Отримані результати звіту:

```
Olhas-Air:kubeloader obilan$ make get
kubectl logs aggregator -n kubeloader | grep Results
Results of the load scenario 'No load': Success: 100.00  % Query per second: 10049    Latency mean: 798.977µs 99th:  3.159046ms
Results of the load scenario 'Light load': Success: 100.00  % Query per second: 10029    Latency mean: 763.06µs 99th:  1.345828ms
Results of the load scenario 'Basic load': Success: 100.00  % Query per second: 50076    Latency mean: 844.034µs 99th:  2.356106ms
Results of the load scenario 'Normal load': Success: 100.00  % Query per second: 100237   Latency mean: 1.112798ms 99th:  5.88042ms
Results of the load scenario 'Heavy load': Success: 100.00  % Query per second: 299452   Latency mean: 539.0202ms 99th:  1.450591162s
Results of the load scenario 'Huge load': Success: 100.00  % Query per second: 496600   Latency mean: 916.805635ms 99th:  1.847189119s
```

Порівняємо після запуску системи на іншому хмарному сервісі.

4.2 Google Kubernetes Engine від Google Cloud Engine

Google Kubernetes Engine забезпечує кероване середовище для розгортання, управління та масштабування ваших контейнерних програм за допомогою інфраструктури Google. Середовище GKE складається з декількох машин (зокрема, екземплярів Google Compute Engine), згрупованих разом для формування кластеру.

Оркестрація кластерів з GKE. Кластери GKE працюють за допомогою системи управління кластерами з відкритим кодом Kubernetes. Kubernetes пропонує механізми, за допомогою яких відбувається взаємодія з кластером. Команди та ресурси Kubernetes використовуються для розгортання та управління своїми програмами, виконання завдань адміністрування та встановлення політик та контролю стану здоров'я ваших розгорнутих навантажень.

Kubernetes базується на тих же принципах дизайну, які керують популярними службами Google і надає ті самі переваги: автоматичне управління, моніторинг та зонди життєздатності контейнерів додатків, автоматичне масштабування, прокручування оновлень тощо.

Версії та особливості Kubernetes. Майстри кластерів GKE автоматично удосконалюються для запуску нових версій Kubernetes, оскільки ці версії стають стабільними, тому ви можете скористатися новими можливостями від проекту Kubernetes з відкритим кодом.

Рисунок 4.3 – Схема GKE

Kubernetes на хмарній платформі Google. Запускаючи кластер GKE, ви також отримуєте переваги розширених функцій управління кластерами, які надає Google Cloud Platform. До них належать:

– збалансування навантаження Google Cloud Platform для екземплярів Compute Engine;

– набори вузлів для призначення підмножини вузлів в кластері для додаткової гнучкості;

– автоматичне масштабування кількості примірників вузла кластера;

– автоматичне оновлення програмного забезпечення вузла кластера;

– авторемонт вузла для підтримки здоров'я та доступності вузла;

– реєстрація та моніторинг за допомогою Stackdriver для наочності у вашому кластері.

– Google Kubernetes Engine (GKE), кластер складається щонайменше з одного головного кластера та декількох робочих машин, званих вузлами. Ці майстерні та вузлові машини керують системою оркестрації кластерів Kubernetes.

Кластер є основою GKE: об’єкти Kubernetes, які представляють ваші контейнерні програми, працюють над кластером.

Майстер кластерів. Мастер кластера запускає процеси управління площиною управління Kubernetes, включаючи сервер Kubernetes API, планувальник та контролери основних ресурсів. Життєвим циклом магістра керується GKE під час створення або видалення кластера. Це включає оновлення версії Kubernetes, що працює на майстрі кластера, яке GKE виконує автоматично, або вручну за вашим запитом, якщо ви бажаєте оновити раніше, ніж автоматичний графік.

Для створення кластера EKS було використано Terraform без зовнышнього модуля, оскільки конфігурація Kubernetes кластера в GCP є нативної відносно AWS. Додаток Г демонструє деталізовану схему кластера.

Конфігурація:

```
provider "google" {
  credentials = "${file("~/.config/gcloud/cred.json")}"
```

```
  project     = "diploma-256122"
  region      = "europe-west1"
}
resource "google_container_cluster" "gke-cluster" {
  name               = "test-gke-cluster"
  network            = "default"
  location               = "europe-west1-b"
  initial_node_count = 5
  node_config {
    preemptible  = true
    machine_type = "n1-standard-2"
    metadata = {
      disable-legacy-endpoints = "true"
    }
  }
  addons_config {
    http_load_balancing {
      disabled = true
    }

    horizontal_pod_autoscaling {
      disabled = true
    }
  }
}
```

Окільки створені кластери у різних хмарних сервісах повинні бути однаковими за ресурсами, то було обрано n1-standard-2 тип для кожної з 5 інстансів, що відповідаю 2 CPU та 7.5 GB пам’яті. Для створення кластера в GCP не потрібно створювати ніяких додаткових ресурсів.

Результат створеного кластера:

```
Olhas-Air:gke obilan$ kubectl get nodes
NAME                                                  STATUS   ROLES
AGE     VERSION
```

```
      gke-test-gke-cluster-default-pool-91daa092-7qfz   Ready    <none>
6h48m   v1.13.11-gke.14
      gke-test-gke-cluster-default-pool-91daa092-9g5m   Ready    <none>
6h48m   v1.13.11-gke.14
      gke-test-gke-cluster-default-pool-91daa092-h8gm   Ready    <none>
6h48m   v1.13.11-gke.14
      gke-test-gke-cluster-default-pool-91daa092-hhmm   Ready    <none>
4h59m   v1.13.11-gke.14
      gke-test-gke-cluster-default-pool-91daa092-tbkb   Ready    <none>
6h48m   v1.13.11-gke.14
```

Далі запускаємо систему для визначення продуктивності кластера та дивимося проміжні створені поди у тестовому неймспейсі:

```
      Olhas-Air:gke obilan$ kubectl get pods -n kubeloader
      NAME              READY   STATUS    RESTARTS   AGE
      aggregator        1/1     Running   0          22s
      loadgen-vzz65     1/1     Running   0          22s
      webserver-724rw   1/1     Running   0          5s
      webserver-75pz7   1/1     Running   0          22s
      webserver-bgx72   1/1     Running   0          5s
      webserver-bpmpj   1/1     Running   0          5s
      webserver-htzf9   1/1     Running   0          5s
      webserver-m69ln   1/1     Running   0          5s
      webserver-qk5bw   1/1     Running   0          5s
      webserver-s7sqw   1/1     Running   0          5s
      webserver-xzcds   1/1     Running   0          5s
      webserver-xzsfh   1/1     Running   0          5s
```

Отримані результати звіту:

```
      Olhas-Air:kubeloader obilan$ make get
      kubectl logs aggregator -n kubeloader | grep Results
      Results of the load scenario 'No load': Success: 100.00  % Query
per second: 10035    Latency mean: 1.057385ms 99th:  8.431097ms
```

```
Results of the load scenario 'Light load': Success: 100.00 % Query per second: 10026 Latency mean: 698.808µs 99th: 3.561517ms
Results of the load scenario 'Basic load': Success: 100.00 % Query per second: 50188 Latency mean: 3.02228ms 99th: 23.029807ms
Results of the load scenario 'Normal load': Success: 100.00 % Query per second: 100248 Latency mean: 20.812261ms 99th: 125.260319ms
Results of the load scenario 'Heavy load': Success: 100.00 % Query per second: 299510 Latency mean: 824.097238ms 99th: 1.871545441s
Results of the load scenario 'Huge load': Success: 100.00 % Query per second: 494139 Latency mean: 748.192385ms 99th: 2.335936925s
```

Порівняємо після запуску системи на іншому хмарному сервісі.

4.3 Аналіз результатів EKS та GKE

Для порівняння складемо таблицю:

Таблиця 4.1 – Порівняння результатів системи для хмарних провайдерів

Сценарій	Параметр	AWS EKS	GCP GKE
No load	Mean latency	798.977µs	1.057385ms
	99^{th} latency	3.159046ms	8.431097ms
Light load	Mean latency	763.06µs	698.808µs
	99^{th} latency	1.345828ms	3.561517ms
Basic load	Mean latency	844.034µs	3.02228ms
	99^{th} latency	2.356106ms	23.029807ms
Normal load	Mean latency	1.112798ms	20.812261ms
	99^{th} latency	5.88042ms	125.260319ms
Heavy load	Mean latency	539.0202ms	824.097238ms
	99^{th} latency	1.450591162s	1.871545441s

Продовження таблиці 4.1 – Порівняння результатів системи для хмарних провайдерів

Huge load	Mean latency	916.805635ms	748.192385ms
	99th latency	1.847189119s	2.335936925s

За таблицею складаємо 2 графіки:

– порівняння mean latency (рисунок 4.4);

– порівняння 99th latency (рисунок 4.5).

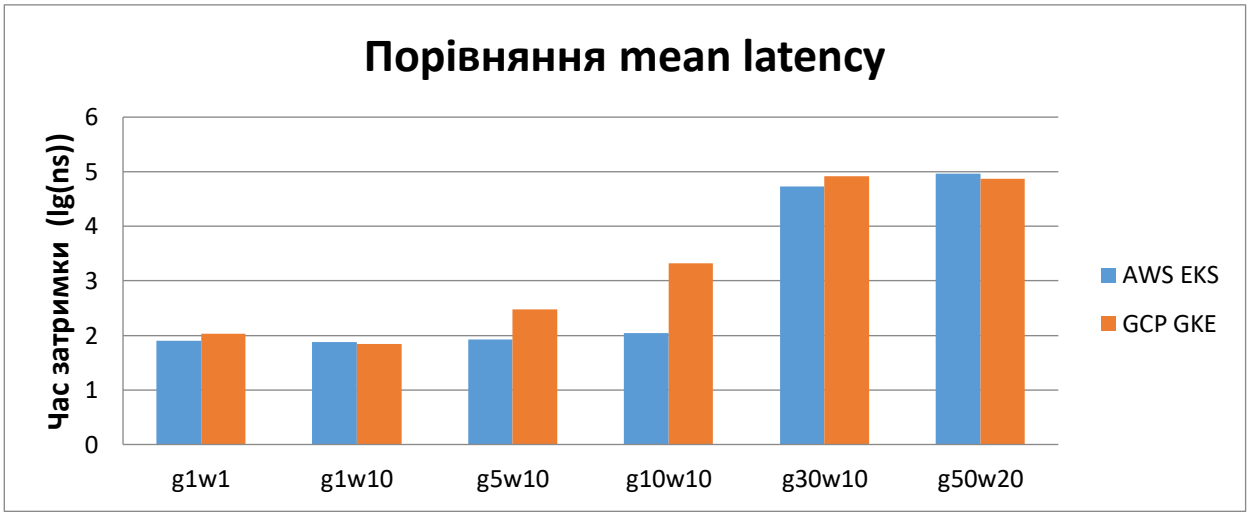

Рисунок 4.4 – Порівняння mean latency для хмар

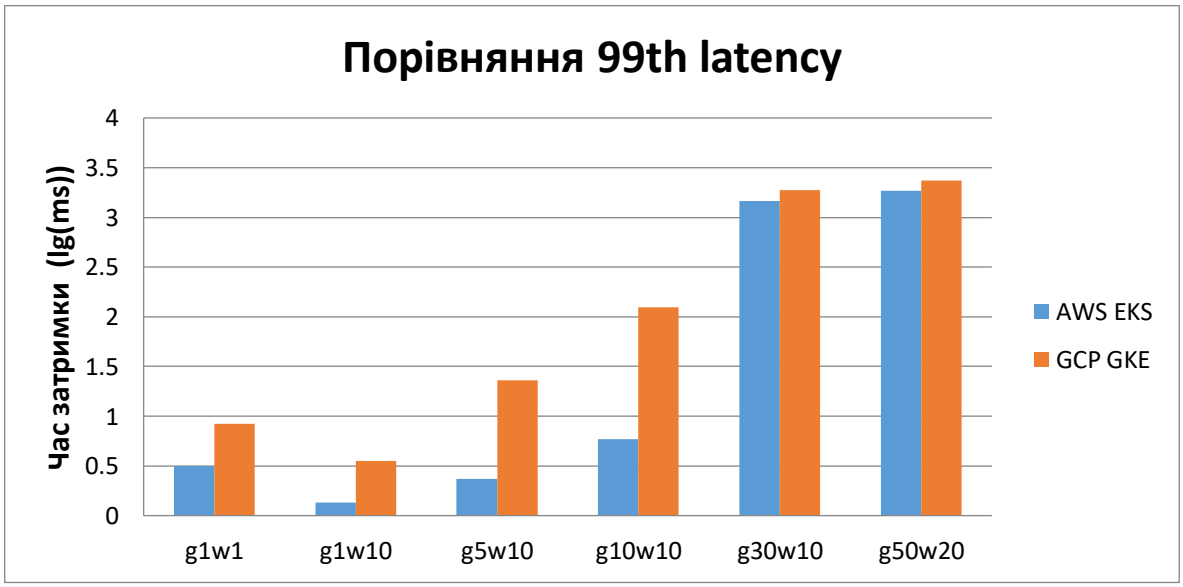

Рисунок 4.5 – Порівняння 99th latency для хмар

З таблиці та рисунку вище видно, що кластер AWS має меншу затримку відносно всіх сценаріїв, це обумовлено тим, що все таки тип інстансу не є однаковим с GCP. Також мережа AWS є відносно кращою та швидшою. Тільки при максимальному навантаження зі сценаріїв ми бачимо приблизно однакові результати.

Для обидвох кластерів можна включити автоматичне розширення ресурсів.

Не зважаючи на гірші показники, GKE має ряд переваг:

– автоматичне оновлення версії Kubernetes;

– надає свою інтегровану платформу моніторингу під назвою Stackdriver для моніторингу Kubernetes. Він може контролювати головний мастер і вузли, а також інтегрувати журнал і всі компоненти Kubernetes всередині платформи без додаткових кроків вручну;

– має найкращу доступність – High availability;

– має найбільш зріле рішення для автоматичного масштабування, доступне в інтерфейсі або через CLI. Користувачеві необхідно вказати бажаний розмір VM та мінімальну та максимальну кількість вузлів у пулі вузлів. Звідти Google Cloud обробляє всі інші кроки.

5 РОЗРОБКА СТАРТАП ПРОЕКТУ

У цьому розділі пропонується підхід до монетизації роботи, висвітлений у цьому документі. На основі аналізу ринку сформульовано основні вимоги, описано ідеї, виявлено сильні та слабкі сторони потенційного комерційного продукту.

Інформація буде представлена в таблицях.

5.1 Опис ідеї проекту

Основна мета стартап-проекту – розробити систему kubeloader, яка може бути використана призначеними замовниками (про це піде мова пізніше). Розглянемо зміст ідеї, можливі області застосування, основні переваги, які користувач може представити у таблиці 5.1.

Таблиця 5.1 – Опис ідеї стартап-проекту

Зміст ідеї	Напрямки застосування	Вигоди для користувача
Аналіз продуктивності Kubernetes кластера	Домашня мережа	Для вивчення Kubernetes кластера
	Робоча мережа	Для полегшення проектування Kubernetes кластера для компаній
	3. Інтернет (Навчальні сайти)	Порівняння різних конфігурацій кластерів, різних провайдерів хмарних послуг для Kubernetes.

Аналіз ринку показав, що фактично потенційні конкуренти відсутні. Існує схожа система, така як perf-test/clusterloader2. Проте звіт конкурента є малоінформативним, а налаштування є складним. Високий ступінь функціонального примітивізму та складність використання, обмеженість застосувань тощо. Вони більш докладно описані в таблиці 5.2.

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

Техніко-економічні характеристики ідеї	(потенційні) товари/концепції конкурентів			W (слабка сторона)	N (нейтральна сторона)	S (сильна сторона)
	Мій проект		Clusterloader			
Мова розробки	GO		GO			Новітні технології розробки
Тип	Мікросервіси		Мікросервіси		Мікросервіси	
Ліцензія	Умовно безкоштовна		Умовно безкоштовна		Безкоштовна	
Фінальний звіт	Включає в себе затримку при обробці		Немає результатів затримки, тільки кількість оброблених запитів			Затримка продемонстрвоана у звіті
Параметри, які можна змінити	Велика кількість параметрів		Невелика кількість параметрів			Більша кількість змінюваних параметрів
Додатковий функціонал	Можливість запуску без додаткових налаштувань		Необхідні додаткові налаштування			Запуск системи не потребує додаткового конфігурацій

Остаточний перелік слабких, сильних та нейтральних характеристик та властивостей ідеї потенційного продукту є основою для налаштування його конкурентоспроможності.

5.2 Технологічний аудит ідеї проекту

Далі йде аудит технології, за допомогою якої ідея проекту може бути реалізована.

Визначення технологічної життєздатності проектної ідеї передбачає аналіз наступних компонентів (табл. 5.3):

Таблиця 5.3 – Технологічна здійсненність ідеї проекту

№ п/п	Ідея проекту	Технології її реалізації	Наявність технологій	Доступність технологій
1	Командний інтерфейс	Make, bash	Наявні власні напрацювання	Так
2	Автоматизація розгортання	Helm, Terraform	Наявні безкоштовні бібліотеки для створення та використання, а також власні доопрацювання	Так
Обрана технологія реалізації ідеї проекту: Розробляється програма у вигляді мікросервісного за стосунку з трьох компонентів на мові GO. Автоматизація розгортання системи потребує make, Helm та Terraform. Усі інструменти є безкоштованими				

Тому, як видно з попередньої інформації, технічна доцільність проекту присутня. У використанні та підготовці бібліотек не повинно виникнути труднощів.

5.3 Аналіз ринкових можливостей запуску стартап-проекту

Зробимо аналіз попиту. Далі в таблиці 5.4 наводиться його аналіз, виходячи з наявного ринку.

Таблиця 5.4 – Попередня характеристика потенційного ринку стартап-проекту

№ п/п	Показники стану ринку (найменування)	Характеристика
1	Кількість головних гравців, од	Умовно 2
2	Загальний обсяг продаж, грн/ум.од	-\-
3	Динаміка ринку (якісна оцінка)	Зростає
4	Наявність обмежень для входу (вказати характер обмежень)	Відсутні
5	Специфічні вимоги до стандартизації та сертифікації	Практично відсутні
6	Середня норма рентабельності в галузі (або по ринку), %	90%

Ринок досить специфічний і вузький, але ніша практично не зайнята. Деякі статистичні дані недоступні.

Далі наведені групи визначених потенційних клієнтів, їх характеристики та орієнтовний перелік вимог до продукції для кожної групи.

У таблиці 5.5 наведені основні характеристики потенційних клієнтів стартап-проектів.

Таблиця 5.5 – Характеристика потенційних клієнтів стартап-проекту

№ п/п	Потреба, що формує ринок	Цільова аудиторія (цільові сегменти ринку)	Відмінності у поведінці різних потенційних цільових груп клієнтів	Вимоги споживачів до товару

Продовження таблиці 5.5 – Характеристика потенційних клієнтів стартап-проекту

1	Економія використання ресурсів	Компанії, які використовують Kubernetes кластер	Деякі покупці хочуть мати результати на діючому кластері	- інтуїтивно зрозумілий інтерфейс налаштування - висока інформативність звіту
2	Потреба у точному проектування кластера на ранніх етапах	Адміністратори та архітектори компанії	Кластери розгорнуті у різних середовищах	- інтуїтивно зрозумілий інтерфейс налаштування - висока інформативність звіту

Після виявлення груп потенційних клієнтів було проведено аналіз ринкового середовища: складені таблиці факторів, що сприяють реалізації проекту на ринку, та факторів, які його перешкоджають (таблиці 5.6 та 5.7). Фактори в таблиці представлені у порядку зменшення значення.

Таблиця 5.6 – Фактори загроз

№ п/п	Фактор	Зміст загрози	Можлива реакція компанії
1	Поява конкурентів	Подальша поява конкурентів після збільшення долі ринку	Розширення функціоналу системи

Продовження таблиці 5.6 – Фактори загроз

2	Ціна товару	Дорога розробка спричинює високу ціну товару, оскільки ціна виробляється з відновленням інвестицій протягом 3 років.	Дивіться цінову політику, можливо, зменшуючи її вартість, залучаючи більше користувачів. Просування за допомогою реклами.

Таблиця 5.7 – Фактори можливостей

№ п/п	Фактор	Зміст можливості	Можлива реакція компанії
1	Активний піар товару	Рекламуйте якісною рекламою.	Використання маркетингових послуг.

Далі наведено результат проведеного аналіз пропозиції де визначаються загальні риси конкуренції на ринку (таблиця 5.8).

Таблиця 5.8 – Ступеневий аналіз конкуренції на ринку

Особливості конкурентного середовища	В чому проявляється дана характеристика	Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентоспроможною)
1. Вказати тип конкуренції - монополія/олігополія/ монополістична/чиста	олігополія	Постійний розвиток та оновлення. Враховування побажань замовників.
2. За рівнем конкурентної боротьби - локальний/національний/…	національний	Максимальна можливість різноманітності локалізацій

Продовження таблиці 5.8 – Ступеневий аналіз конкуренції на ринку

3. За галузевою ознакою - міжгалузева/ внутрішньогалузева	міжгалузева	Позитивний вплив
4. Конкуренція за видами товарів: - товарно-родова - товарно-видова - між бажаннями	Товарно-родова. Конкуренція на рівні технології задоволення потреб.	Врахування побажань та їх задоволення приносить позитивний вплив
5. За характером конкурентних переваг - цінова / нецінова	нецінова	Головною конкурентною перевагою є унікальність, адже функціонал достатньо якісний та практичний.

Оскільки конкуренція досить слабка, ринок стає привабливим для розвитку. Також немає проблем з частково підготовленими рішеннями: запитуючі бібліотеки доступні для використання при розробці програми (модуля) під вільною ліцензією або ліцензією умовно-безкоштовного використання. Основними сильними сторонами продукту є функціональність, якість роботи та простота використання, що не стосується конкурентів.

На основі аналізу конкуренції, крім врахування особливостей проектної ідеї, вимог споживача до товару та факторів маркетингового середовища, визначається і підтверджується перелік конкурентних факторів. Аналіз проводиться згідно таблиці. 5.9.

Таблиця 5.9 – Обґрунтування факторів конкурентоспроможності

№ п/п	Фактор конкурентоспроможності	Обґрунтування (наведення чинників, що роблять фактор для порівняння конкурентних проектів значущим)
1	Задоволення конкретних потреб	На основі аналізу форумів програм аналогів можна дійти висновку, що потреби користувачів в той чи іншій функції не задоволені, або задоволені лише частково. Тому даний проект призваний задовольнити потреби.
2	Максимальна інтуїтивність взаємодії	Ці фактори зумовлені розвитком технології в першу чергу. І його доступність. Оскільки аналоги програмних аналогів були розроблені набагато раніше, то технології того часу не могли реалізувати вищезазначені фактори за розумною вартістю та якістю.
3	Інформативність звіту	
4	Точність формування звіту	

За визначеними факторами конкурентоспроможності (табл. 5.9) проводиться аналіз сильних та слабких сторін стартап-проекту (табл. 5.10). (С.П. – стартап проект, К.1 – конкурент 1).

Таблиця 5.10 – Порівняльний аналіз сильних та слабких сторін мого проекту

№ п/п	Фактор конкурентоспроможності	Бали 1-20	Рейтинг товарів-конкурентів у порівнянні						
			-3	-2	-1	0	1	2	3
1	Інтуїтивність налаштування	20				К.1		С.П.	
2	Достовірність результатів	20						К.1	С.П.
3	Можливість додаткового налаштування	20					К.1		С.П.

Продовження таблиці 5.10 – Порівняльний аналіз сильних та слабких сторін мого проекту

4	Інформативність звіту	20	К.1				С.П.		
5	Вартість	20				С.П. –К.1			

Фінальним етапом ринкового аналізу можливостей впровадження проекту є складання SWOT-аналізу (матриці аналізу сильних (Strength) та слабких (Weak) сторін, загроз (Troubles) та можливостей (Opportunities) (табл. 5.11) на основі виділених ринкових загроз та можливостей, та сильних і слабких сторін (табл. 5.10).

Таблиця 5.11 – SWOT- аналіз стартап-проекту

Сильні сторони: Задоволення потреб ринку: – висока інформативність звіту; – високий рівень кастомізації; – зручнсть використання.	Слабкі сторони: – ціна товару; – відносна складність розробки; – необхідність кастомізації під кожного замовника.
Можливості: – розширення сфер застосування.	Загрози: – малий обсяг продаж; – поява конкурентів.

На основі SWOT-аналізу розробляються альтернативи ринкової поведінки (перелік заходів) для виведення стартап-проекту на ринок та орієнтовний оптимальний час їх ринкової реалізації з огляду на потенційні проекти конкурентів.

Визначені альтернативи аналізуються з точки зору строків та ймовірності отримання ресурсів (табл. 5.12).

Таблиця 5.12 – Альтернативи ринкового впровадження стартап-проекту

№ п/п	Альтернатива (орієнтовний комплекс заходів) ринкової поведінки	Ймовірність отримання ресурсів	Строки реалізації
1	Поліпшення темпів розповсюдження завдяки роботі маркетологів	присутня	стислі
2	Зменшення витрат на розробку завдяки використанню безкоштовних рішень	проста	обширні
3	Короткочасне зменшення ціни на продукт для заманювання клієнтів. Розділення функцій програми на пакети для вибору	проста	стислі

Для початку доцільно обрати альтернативу номер 3.

5.4 Розроблення ринкової стратегії проекту

Розробка ринкової стратегії як перший крок включає визначення стратегії розкриття ринку.

Кожен профіль вашої цільової аудиторії буде відповідним. Важливим моментом для аналізу є готовність споживачів прийняти товар.

Ми також аналізуємо цільовий попит у межах цільової групи, інтенсивність конкуренції в сегменті, а також оцінюємо простоту входу в сегмент кожної цільової групи потенційних клієнтів.

Опис цільових груп потенційних споживачів (табл. 5.13).

Таблиця 5.13 – Вибір цільових груп потенційних споживачів

№ п/п	Опис профілю цільової групи потенційних клієнтів	Готовність споживачів сприйняти продукт	Орієнтовний попит в межах цільової групи (сегменту)	Інтенсивність конкуренції в сегменті	Простота входу у сегмент
1	Компанії інформаційних технологій	Висока зацікавленість	Середній попит	Практично відсутня на початку. Подальше збільшення	Середня складність
2	Адміністратори\архітектори	Ймовірна зацікавленість у груп, що націлені на наукові та офіційні теми.	Середній-вище середнього	Практично відсутня на початку. Подальше збільшення	Складність нижче середнього
3	Офіційні сайти для навчання	Зацікавленість вище середньої	Середній попит	Практично відсутня на початку. Подальше збільшення	Висока складність
Краще всього підійдуть групи 1 та 2 так як простота входу та рівень зацікавленості прийнятні.					

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

– якщо компанія зосереджується на одному сегменті – вона обирає стратегію концентрованого маркетингу;

– якщо працює із кількома сегментами, розробляючи для них окремо програми ринкового впливу – вона використовує стратегію диференційованого маркетингу;

– якщо компанія працює із всім ринком, пропонуючи стандартизовану програму (включно із характеристиками товару/послуги) – вона використовує масовий маркетинг.

Тож для роботи в обраних сегментах ринку необхідно сформувати базову стратегію розвитку (табл. 5.14).

Таблиця 5.15 – Визначення базової стратегії розвитку

№ п/п	Обрана альтернатива розвитку проекту	Стратегія охоплення ринку	Ключові конкуренти позиції відповідно до обраної альтернативи	Базова стратегія розвитку
1	Короткострокове зниження цін на продукцію для залучення клієнтів. Розділення програмних функцій у пакунках для вибору (різний залежно від клієнтів)	Завдяки поділу функцій на пакети можна дістатись до більшої частини ринку, адже замовник купує лише те, що йому потрібно.	Оскільки конкуренти, як і вони, не створюють значної конкуренції, цей елемент можна залишити порожнім.	Стратегія диференціації

Наступним кроком є вибір стратегії конкурентної поведінки (табл. 5.15).

Таблиця 5.15 – Визначення базової стратегії конкурентної поведінки

№ п/п	Чи є проект «першопрохідцем» на ринку?	Чи буде компанія шукати нових споживачів, або забирати існуючих у конкурентів?	Чи буде компанія копіювати основні характеристики товару конкурента, і які?	Стратегія конкурентної поведінки
1	Ні	Так, буде	Копіювання буде лише основних функцій, але із значним вдосконаленням	Стратегія лідера

Залежно від потреб споживачів вибраних сегментів для постачальника (новоствореної компанії) та для товару (див. Таблицю 5.5), а також залежно від обраної базової стратегії розвитку (таблиця 5.14) та стратегії поведінки конкурентно (таблиця 5.15) розробляється стратегія позиціонування, яка полягає у формуванні ринкової позиції (комплексу асоціацій), за допомогою якої споживачі повинні ідентифікувати бренд / проект.

У таблиці 5.16 наведена розроблена стратегія позиціонування продукту.

Таблиця 5.16 – Визначення стратегії позиціонування

№ п/п	Вимоги до товару цільової аудиторії	Базова стратегія розвитку	Ключові конкурентоспроможні позиції власного стартап-проекту	Вибір асоціацій, які мають сформувати комплексну позицію власного проекту (три ключових)
1	Базові знання Kubernetes	Стратегія диферен-ціації	Інформативність звіту Можливість додаткової конфігурації	Визначення продуктивності Kubernetes кластера Проектування Kubernetes кластера Порівняння різних середовищ для Kubernetes

Простіше кажучи: розробник вирішує розвиватися у напрямку основних функцій: функцій: 1) Визначення продуктивності Kubernetes кластера. 2) Проектування Kubernetes кластера. 3) Порівняння різних середовищ для Kubernetes.

5.5 Розроблення маркетингової програми стартап-проекту

Маркетингова програма стартап-проекту розроблена на основі потреб, заснованих на концепції потенційного продукту. Переваги та переваги, пропоновані продуктом, формулюються, особливо в порівнянні з конкурентами.

Перший крок – сформувати маркетингову концепцію товару, який отримає споживач. Для цього у таблиці 5.17 узагальнено результати попереднього аналізу товарної конкурентоспроможності.

Таблиця 5.17 – Визначення ключових переваг концепції потенційного товару

№ п/п	Потреба	Вигода, яку пропонує товар	Ключові переваги перед конкурентами (існуючі або такі, що потрібно створити
1	Інформативність та достовірність звіту	Найвищий рівень	Найвищій показник.
2	Зрозумілість та зручність	Простота налаштування та запуску	Завдяки використанню нових технологій – значна перевага перед конкурентами.

Крім того, розробляється трирівнева маркетингова модель товару: конкретизуються уявлення про товар та / або послугу, його фізичні компоненти та характеристики процесу його надання (табл. 5.18).

Таблиця 5.18 – Опис трьох рівнів моделі товару

Рівні товару	Сутність та складові
І. Товар за задумом	Мікросервісна система націлена на аналіз продуктивності Kubernetes кластера. Можливе використання для різних Kubernetes кластерів незалежно від середовища

Продовження таблиці 5.18 – Опис трьох рівнів моделі товару

II. Товар у реальному виконанні	Властивості/характеристики	М/Нм	Вр/Тх /Тл/Е/Ор
	1. Період обслуговування за ліцензією	Років	2
	2. Зручність використання	-	Висока
	3. Наявність документації	-	Так
	4. К-сть параметрів налаштування	Шт.	10+
	5. Формування звітів.	-	Так
	Програма пройшла усі можливі тестування та повністю відповідає визначеним головним вимогам ринку.		
	Постачається в електронному вигляді на сайті виробника.		
	Марка: Білан-лімітед. Назва: Проект-бета		
III. Товар із підкріпленням	Програмне забезпечення		
	Програмне забезпечення		
Проект буде захищено від копіювання регістрацією назви програми, створення заявки на отримання патенту на винахід, щоб уберегти алгоритм роботи від копіювання.			

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

ВИСНОВКИ ТА РЕКОМЕНДАЦІЇ

У даній магістерській дисертації розроблено систему аналізу продуктивності Kubernetes кластера при плануванні побудови мікросервісної інфраструктури. У рамках даного дослідження були проаналізовані існуючі системи. Наприклад, створена система на відміну від існуючих дозволяє аналізувати час затримки обробки пакетів та додавати його до фінального звіту при вимірювання продуктивності.

За рахунок використання шаблонів конфігурацій ресурсів вдалося суттєво зменшити час, який витрачається адміністратором системи для проведення досліджень. Система має параметри, які можна адаптувати під кластер за потребою. Автоматизація була виконана за допомогою сервісів make, Helm та Terraform. Структура системи базується на мікросервісах, що розгортаються всередині Kubernetes кластера.

Система була протестована на двох кластерах у різних хмарних провайдерів. Отримані результати свідчать про те, що система працює та виконує свої функції на обох. Для тестування було створено два Kubernetes кластери з відносно однаковим набором ресурсів у двох популярних хмарних провайдерів – Amazon Web Services та Google Cloud Platform. Результати експериментів показали, що час затримки обробки запитів AWS кластера є дещо кращим, проте GCP має вбудовані сервіси для моніторингу, які також використовують ресурси.

У якості майбутніх напрямків досліджень та вдосконалення системи потрібно вибрати наступне:

– можливість створення додаткових сценаріїв тестування за рахунок додавання нових вхідних параметрів;

– додавання до порівняння Azure як хмарного провайдера для того щоб переконатися, що результати залежать тільки від наданих ресурсів для Kubernetes;

– розширення методів тестування за рахунок додавання хаотичного тестування, який може бути використаний для робочого кластера.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Мікросервісна архітектура [Електронний ресурс] – Режим доступу: https://medium.com/@IvanZmerzlyi/microservices-architecture-461687045b3d.

2. Що таке Docker і як використовувати його з Python [Електронний ресурс] – Режим доступу: https://codeguida.com/post/1837

3. Kubernetes: An Overview [Електронний ресурс] – Режим доступу: https://thenewstack.io/kubernetes-an-overview/

4. Kubernetes documentation: Pods [Електронний ресурс] – Режим доступу: https://kubernetes.io/docs/concepts/workloads/pods/pod/

5. Amazon EKS Workshop [Електронний ресурс] – Режим доступу: https://eksworkshop.com/introduction/basics/concepts_objects/

6. Why (and when) you should use Kubernetes [Електронний ресурс] – Режим доступу: https://hackernoon.com/why-and-when-you-should-use-kubernetes-8b50915d97d8

7. Kubernetes Performance Measurements and Roadmap [Електронний ресурс] – Режим доступу: https://kubernetes.io/blog/2015/09/kubernetes-performance-measurements-and/

8. Kubernetes Monitoring: Best Practices, Methods, and Existing Solutions [Електронний ресурс] – Режим доступу: https://logz.io/blog/kubernetes-monitoring/

9. Which One Should You Prioritize? Kubernetes Performance, Cluster Utilization, or Cost Optimization? [Електронний ресурс] – Режим доступу: https://medium.com/@Mohamed.ahmed/which-one-should-you-prioritize-kubernetes-performance-cluster-utilization-or-cost-optimization-21469263b6a7

10. Load Testing Tutorial: What is? How to? (with Examples) [Електронний ресурс] – Режим доступу: https://www.guru99.com/load-testing-tutorial.html

11. PowerfulSeal [Електронний ресурс] – Режим доступу: https://github.com/bloomberg/powerfulseal#running-inside-of-the-cluster

12. Principles of chaos engineering [Електронний ресурс] – Режим доступу: http://principlesofchaos.org/?lang=ENcontent

13. Kube-monkey [Електронний ресурс] – Режим доступу: https://github.com/asobti/kube-monkey

14. Kubernetes perf-tests [Електронний ресурс] – Режим доступу: https://github.com/kubernetes/perf-tests

15. Go (мова програмування) [Електронний ресурс] – Режим доступу: https://uk.wikipedia.org/wiki/Go_(%D0%BC%D0%BE%D0%B2%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F)

16. Kubernetes: Production-Grade Container Orchestration [Електронний ресурс] – Режим доступу: https://kubernetes.io/

17. Основы Кубернетис [Електронний ресурс] – Режим доступу: https://habr.com/ru/post/258443/

18. Быстрое введение в Kubernetes [Електронний ресурс] – Режим доступу: https://eax.me/kubernetes/

19. Kubernetes на AWS [Електронний ресурс] – Режим доступу: https://aws.amazon.com/ru/kubernetes/

20. Kubernetes: Why It Matters for Performance Testing [Електронний ресурс] – Режим доступу: https://blog.gurock.com/kubernetes-performance-testing/

21. Kubernetes vs. Docker: A Primer [Електронний ресурс] – Режим доступу: https://containerjournal.com/topics/container-ecosystems/kubernetes-vs-docker-a-primer/

22. Distributed load testing with Gatling and Kubernetes [Електронний ресурс] – Режим доступу: https://medium.com/de-bijenkorf-techblog/https-medium-com-annashepeleva-distributed-load-testing-with-gatling-and-kubernetes-93ebce26edbe

23. Distributed load testing using Google Kubernetes Engine [Електронний ресурс] – Режим доступу: https://cloud.google.com/solutions/distributed-load-testing-using-gke

24. Kubernetes End-to-end Testing for Everyone [Електронний ресурс] – Режим доступу: https://kubernetes.io/blog/2019/03/22/kubernetes-end-to-end-testing-for-everyone/

25. Scale and Performance Testing of Kubernetes [Електронний ресурс] – Режим доступу: https://www.mirantis.com/blog/scale-performance-testing-kubernetes-answers-specific-applications/

26. 3 best practices for container performance testing [Електронний ресурс] – Режим доступу: https://techbeacon.com/app-dev-testing/3-best-practices-container-performance-testing

27. Vegeta [Електронний ресурс] – Режим доступу: https://github.com/tsenart/vegeta