НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

Навчально-науковий комплекс "Інститут прикладного системного аналізу"
(повна назва інституту/факультету)

Кафедра Системного проектування
(повна назва кафедри)

«На правах рукопису»
УДК 004.03

«До захисту допущено»

Завідувач кафедри
_____ А.І. Петренко
(підпис) (ініціали, прізвище)
"___"_____2017 р.

Магістерська дисертація

на здобуття ступеня магістра

зі спеціальності 8.05010102 Інформаційні Технології
Проектування
(код і назва)

на тему: «Використання мікросервісів при розробці локальної комп'ютерної мережі»

Виконав: студент 6 курсу, групи ДА-51м
(шифр групи)

Мироненко Сергій Сергійович _____
(прізвище, ім'я, по батькові) (підпис)

Науковий керівник доцент, к.т.н., Кисельов Г.Д. _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант Розробка
стартап-проекту доцент, к.т.н., Кисельов Г. Д. _____
(назва розділу) (науковий ступінь, вчене звання, , прізвище, ініціали) (підпис)

Рецензент професор Кафедри автоматизації проектування _____
енергетичних процесів та систем
д.т.н., професор, Аушева Н.М. _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

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

Київ – 2017 року

Національний технічний університет України

«Київський політехнічний інститут»

Інститут (факультет) ННК "Інститут прикладного системного аналізу"
(повна назва)

Кафедра Системного проектування
(повна назва)

Рівень вищої освіти – другий (магістерський)

Спеціальність 8.05010102 Інформаційні технології проектування
(код і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ А.І. Петренко
(підпис) (ініціали, прізвище)
«___»_____2017 р.

ЗАВДАННЯ
на магістерську дисертацію студенту
Мироненку Сергію Сергійовичу
(прізвище, ім'я, по батькові)

1. Тема дисертації Використання мікросервісів при розробці локальної компютерної мережі,
науковий керівник дисертації Кисельов Генадій Дмитрович, к.т.н., доц. ,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «20» березня 2017 р. № 32-ст

2. Термін подання студентом дисертації _____

3. Об'єкт дослідження Мікросервіси у складних компонованих мережах

4. Предмет дослідження Методи реалізації та побудова складної мережі на мікросервісній архітектурі кафедри СП ННК «ІПСА»

5. Перелік завдань, які потрібно розробити

1. Провести систематизацію, порівняння поширених технік побудови мереж на мікроервісах
2. Провести аналіз популярних оркестраторів та обрати центральний сервіс
3. Побудувати конфігурацію оркестртора та кластера
4. Розробити зразок кластера
5. Надати практичні рекомендації щодо наповнення кластеру функціоналом
6. Орієнтовний перелік ілюстративного матеріалу презентація на тему:
« Використання мікросервісів при розробці локальної компютерної мережі»
7. Орієнтовний перелік публікацій

Мироненко С.С. “Development and building process of orchestration for microservices network on top of the AWS” - Закритий журнал EPAM systems Ukraine 2016.
Мироненко С.С. “Orchestration, CM and IAC in AWS and in MaaS” - Закритий журнал EPAM systems 2017

8. Консультанти розділів дисертації

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Розробка стартап-проекту	Кисельов Г.Д., к.т.н., доцент		

9. Дата видачі завдання 06.02.2017

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1	2	3	4
1	Отримання завдання	06.02.2017	
2	Збір інформації	20.02.2017	
3	Аналіз вимог завдання, вибір методів і засобів розв’язання поставленої задачі	05.03.2017	
4	Порівняння характеристик оркестраторів	15.03.2017	
5	Написання конфігурацій оркестраторів	01.04.2017	
6	Написання та пошук відповідних контейнерів	08.04.2017	
7	Розгортання системи на тестовому аккаунті AWS	15.04.2017	
1	2	3	4
10	Оформлення дипломної роботи	31.05.2017	
11	Отримання допуску до захисту та подача роботи в ДЕК	09.06.2017	

Студент _____ С.С. Мироненко
(підпис) (ініціали, прізвище)

Науковий керівник дисертації _____ Г.Д. Кисельов
(підпис) (ініціали, прізвище)

РЕФЕРАТ

Магістерська дисертація: 114 сторінок загального тексту, 14 рисунків, 25 таблиць, 35 використаних джерел

Актуальність даної роботи полягає у дослідженні сучасної архітектури мікросервісів та перебудови вже існуючої комплексної логічної мережі на існуючих ресурсах без втрати потужності. Мікросервіси – відносно нова технологія, що дозволяє з легкістю, маючи доволі обмежені ресурси розгортати комплексну логіку з генерації, накопичення та обробки різномнітної інформації.

Наприклад на дану інфраструктуру перейшли вже багато компаній, як то Twitter, Facebook, Netflix – усі ці компанії об’єднує великий обсяг даних та необхідність забезпечення доступ до ресурсів. Кафедра, звичайно ж, не має таких обсягів даних, проте є доцільним модернізувати структуру та мати засоби побудування та оновлення процесів.

Об’єкт дослідження – покращення використання інфраструктури кафедри, та створення інформаційного середовища для обміну та керування інформаційними ресурсами.

Предмет дослідження – створення рекомендацій, конфігурацій та налаштувань кластера мікросервісів та його логічного наповнення для кафедри СП ННК «ІПСА».

Метою досліджень є створення конфігурації, на основі досліджень для поліпшення та модернізації процесів навчання та обробки інформації на кафедрі СП ННК «ІПСА»

Практичною цінністю є розгортання вказаного у цій праці варіанту реалізації кластеру мікросервісів.

Ключові слова:

Kubernetes, Mesos, Docker, кластер, мікросервіс, оркестартор

ABSTRACT

Master’s thesis consists of 114 pages of general text, 14 pictures, 25 tables, 35 sources.

The relevance of this work lies in the field of studies of modern microservices architecture for rebuilding and optimizing for maximum utilization of “IASA” CAD Department’s computer infrastructure. Microservices is rather new technology, which allow user to achieve, with relevant ease, and having limited resources, to build complex application logic and infrastructure, which will generate, stack and process various information.

There are multiple example, which will prove success of this technology. Known users are Twitter, Facebook, Netflix – all of this respective companies share one thing in common, which is – big amounts of information to process, and need in all time working and performance service,

The object of study – modernization and improvement of currently existing CAD department’s infrastructure.

Subject of research – creating the design and configurations for developing own cluster on premise in terms of mentioned department.

The aim of research is to create optimal design, based on the results of the research to redesign the current infrastructure or/and to optimize it’s usability.

Practical use – lies in the creation and providing mentioned microservices cluster design in real life on premises of CAD department.

Key words:

Kubernets, Mesos, Docker, cluster, microservice, orchestrator

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ПЗ	**Програмне забезпечення**
БД	База даних
ОС	Операційна система
YAML	Yet another markup language
IT	Інформаційні технології
K8ts	Kubernetes
***Nix**	Unix, Linux
ПК	Персональний ком’пютер
MQ	Message Queue
API	Application Program Interface
[0-9]k	Тисяч
LMS	Learning Management System
IAC	Infrastructure As Code
CF	Configuration Management
AWS	Amazon Web Service
CD	Continuous Delivery
Vs.	Versus
MaaS	Metal as a Service

Вступ

На даний момент у світі IT існує тенденція до перебудови архітектури додатків на мікросервіси. Дана тенденція виникла із інженерної потреби пришвидшити процес створення додатків та поліпшення процесу розгортання їх. Передусім цей процес можна покращити двома методами:

1) Зробити усі компоненти додатків незалежними
2) У архітектурному плані – зменшити вплив взаємодії між сервісами

Власне кажучи цей підхід є дуже успішним саме із-за того, що будь-якої складності архітектуру можна декомпозувати та скласти таким чином, щоб логічні блоки були максимально відокремленими і незалежними, а канали передачі між ними – прозорими. У рамках мережі це надає позитивного впливу через наступні фактори:

1) Логічна прозорість архітектури
2) Мінімальний вплив між компонентами
3) Похибкостійкість
4) Легкість керування.

Саме через те було прийнято рішення дослідити цю тему і запропонувати інтегрувати кластер мікросервісів у логічну мережу кафедри Системного Проектування ННК «ІПСА».

Надалі розглянемо поняття мікросервісів, а також розробимо загальну архітектуру бажаного кластеру, сформуємо потребу, а також сформуємо та вирішимо інженерну задачу, що буде покладена у інженерну пропозицію, що до створення такого кластеру у мережі кафедри.

1 Розгляд поняття мікросервісу

1.1 Мікросервіси.

У широкому сенсі – мікросервіс описує стиль розробки ПЗ, а також організацію компонентів великої комплексної системи у стилі малих оркестрованих компонентів. З кожним роком все більше і більше спеціалістів у ІТ починають використовувати даний підхід до розробки ПЗ, архітектури чи то інших процесів.

Якщо коротко, то архітектурний стиль мікросервісів – підхід, при якому ПЗ, чи то архітектура будується, як набір невеликих сервісів, кожен з яких працює у власному процесі і комунікує з іншими сервісами, використовуючи легковісні механізми, як – то до прикладу – HTTP, AJP, RMI чи то інші. Ці мікросервіси побудовані навколо необхідностей інженерної потреби та розгортаються незалежно, з використанням автоматизованого середовища. Власне кажучи, ці сервіси можуть бути написані на будь-якій мові програмування, а з точки зору архітектури та коду – виконувати будь-який функціонал.

Для того, щоб розглянути тему детальніше, краще за все порівняти підхід мікросервісів із монолітним класичним стилем на прикладі програмного забезпечення, що збудоване як єдине ціле. Великі бізнес додатки, наприклад, часто включають в себе три основних частини: користувацький інтерфейс, базу даних та власне сервер. Серверна частина оброблює HTTP запити, виконуючи доменну логіку, робить запити у базу даних, заповнює HTTP сторінки, що потім надсилаються клієнту. Будь-які зміни до монолітного потребують повторної компіляції та розгортки додатку.

Монолітний сервер – доволі очевидний спосіб побудови таких систем. Уся логіка обробки запитів виконується у одному процесі, при цьому можна використовувати можливості мови програмування. При цьому можна запускати і тестувати додаток на ПК програміста, використовуючи стандартний процес розгортання для перевірки змін перед розгорткою додатку у середовищі

продуктиву. Монолітний додаток можна горизонтально масштабувати, шляхом запуску декількох фізичних серверів за балансувальником навантаження.
Монолітні додатки – доволі успішні та стабільні, але все більше людей відмолюються від них, через те, що все більше додатків розгортаються у хмарних провайдерах, а логіка обробки інформації у додатку зростая. Будь-які зміни, навіть найменші, потребують перекомпіляції усього моноліту, та рестарту всього стеку. З плином часу стає все важче зберігати гарну модульну структуру, зміни одного модулю мають тенденцію впливати на код інших модулів. Масштабувати доводиться увесь додаток загалом, навіть, якщо це потребує всього один модуль. Ці незручності призвели до архітектурного стилю мікросервісів – побудові додатків та архітектури у вигляді набору сервісів. На додачу до можливості незалежного масштабування та розгортки кожен сервіс також отримує чітку фізичну межу, що дозволяє гнучку ізоляцію та незалежне розгортання сервісу.

1.2 Властивості архітектури мікросервісів

Не можна сказати, що існує формальне визначення стилю мікросервісів, але можна спробувати описати те, що вважається загальними характеристиками додатків, що використовують цей стиль. Не завжди вони зустрічаються в одному додатку всі відразу, але, як правило, кожне подібний додаток включає в себе більшість цих характеристик.

Архітектура мікросервісів використовує бібліотеки, але їх основний спосіб розбиття додатку - шляхом розділення його на сервіси. Бібліотеки визначаються як компоненти, які підключаються до програми і викликаються нею в тому ж процесі, в той час як сервіси - це компоненти, що виконуються в окремому процесі і взаємодіють між собою через веб-запити або RPC(remote procedure call) – процедури відкладеного виклику.

Головна причина використання сервісів замість бібліотек - це незалежне розгортання. Якщо розробляти додаток, що складається з декількох бібліотек, які

працюють в одному процесі, будь-яка зміна, чи оновлення цих бібліотеках призводить до перерозгортання всього додатку, або ж його рестарту. Але якщо додаток розбитий на кілька сервісів, то зміни, що зачіпають будь-який з них, потребуватимуть перерозгортання тільки зміненого сервісу. Звичайно, деякі зміни будуть зачіпати інтерфейси, що, в свою чергу, потребують певної координації між різними сервісами, але мета хорошої архітектури мікросервісів - мінімізувати необхідність в такої координації шляхом установки правильних кордонів між мікросервісами, а також механізму еволюції взаємозв'язків сервісів.

Інший наслідок використання сервісів як компонентів – не явний інтерфейс взаємодії між ними. Більшість мов програмування не мають хорошого механізму для оголошення публічного інтерфейсу, чи то пак endpoint'у (точки виходу програми). В основному тільки документація запобігає порушення інкапсуляції компонентів. Сервіси дозволяють уникнути цього через використання явного механізму віддалених викликів.

Тим не менше, використання сервісів подібним чином має свої недоліки. Дистанційні виклики працюють повільніше, ніж виклики в рамках процесу, і тому API повинен бути менш деталізованим (coarser-grained), що часто призводить до незручності у використанні. Якщо ж потрібно змінити набір взаємодії між компонентами, зробити це складніше через те, що вам потрібно перетинати кордони процесів.

1.3 Обробка даних мікросервісами

При вибудовуванні комунікацій між процесами час від часу споглядається те , як в механізми передачі даних містилася значна частина логіки. Хорошим прикладом тут є Enterprise Service Bus (ESB). ESB-продукти часто включають в себе можливості по передачі, оркестровці і трансформації повідомлень, а також застосування бізнес-логіки[1].

Проте бажаним зараз є протилежне – сам севіс має обробляти інформацію, натомість передавачі просто мають передавати інформацію, але обробкою мають займатись сервіси. Додатки, побудовані з використанням мікросервісної архітектури, мають бути настільки незалежними і сфокусовані, наскільки можливо: вони містять власну доменну логіку і виступають більше як фільтри в класичному Unix сенсі - отримують запити, застосовують логіку і генерують відповідь . Замість складних протоколів, таких як WS або BPEL, вони використовують прості REST-ові протоколи через HTTP.

Команди, що практикують мікросервісну архітектуру, використовують ті ж принципи і протоколи, на яких побудована всесвітня павутина (і, по суті, Unix). Часто використовувані ресурси можуть бути закешовані з дуже невеликими зусиллями з боку розробників або IT-адміністраторів.

Другий часто використовуваний інструмент комунікації - легка шина повідомлень. Така інфраструктура як правило не містить доменної логіки - прості реалізації типу RabbitMQ або ZeroMQ не роблять нічого крім надання асинхронної фабрики передачі повідомлень. Логіка при цьому існує на кінцях цієї шини - в сервісах, які відправляють і приймають повідомлення.

У монолітному додатку компоненти працюють в одному процесі і комунікують між собою через виклик методів. Найбільша проблема в зміні моноліту на мікросервіси лежить в зміні шаблону комунікації. Непродумане розбиття моноліту додатку, чи переносу архітектури на мікросерісну віртуалізацію – призводить до великої кількості комунікацій, що знижує швидкість роботи сервісу, а також призводить до витрати зайвих ресурсів.

1.4 Автоматизація інфраструктури

Техніки автоматизації інфраструктури сильно еволюціонували за останні кілька років. Еволюція хмари в цілому та зокрема AWS зменшила операційну складність побудови, розгортання та функціонування мікросервісів.

Безліч продуктів і систем, що використовують мікросервісну архітектуру, були побудовані командами з великим досвідом в Continuous Delivery і Continuous Integration. Команди, що будують додатки подібним чином, інтенсивно використовують техніки автоматизації інфраструктури. Це проілюстровано на рисунку нижче.

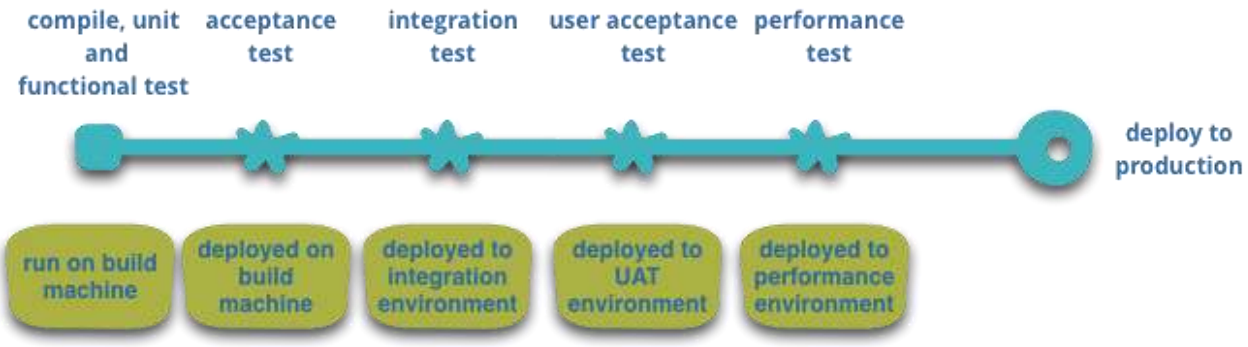

Рис. 1.1- Пайплайн запуску мікросервісів

Інша область, де команди використовують інтенсивну автоматизацію інфраструктури, - це управління мікросервісами в продакшн. На відміну від процесу розгортання, який, як описано вище, у монолітних додатків не сильно відрізняється від такого у мікросервісів, їх спосіб функціонування може мати відчутні відмінності.

1.5 Проектування під відмову

Наслідком використання сервісів як компонентів є необхідність проектування додатків так, щоб вони могли працювати при відмові окремих сервісів. Будь-звернення до сервісу може не спрацювати через його недоступність. Це є недоліком мікросервісів в порівнянні з монолітом, тому що це вносить додаткову складність в додаток.

Так як сервіси можуть відмовити в будь-який час, дуже важливо мати можливість швидко виявити неполадки і, якщо можливо, автоматично відновити працездатність сервісу. Мікросервісна архітектура робить великий акцент на моніторингу додатків в режимі реального часу, як то технічних елементів, наприклад, як багато запитів в секунду отримує база даних. Семантичний моніторинг може надати систему раннього попередження проблемних ситуацій, дозволяючи командам підтримки/розробки розробці підключитися до дослідження проблеми на самих ранніх стадіях.

Це особливо важливо у випадку з мікросервісною архітектурою, тому що розбиття на окремі процеси і комунікація через події може призвести до несподіваної поведінки. Моніторинг украй важливий для виявлення небажаних випадків такої поведінки і швидкого їх усунення.

Моноліти можуть бути побудовані так само прозоро, як і мікросервіси. Насправді, так вони і повинні будуватися[2]. Різниця в тому, що знати, коли сервіси, що працюють в різних процесах, перестали правильно взаємодіяти між собою, це більш критично. У випадку з бібліотеками, розташованими в одному процесі, такий вид прозорості швидше за все буде не так корисний.

Команди, що працюють з мікросервісною архітектурою, як правило, створюють системи моніторингу та логування для кожного індивідуального сервісу. Прикладом може служити консоль, що показує статус (онлайн / офлайн) сервісу

і різні технічні та бізнес-метрики: поточна пропускна здатність, час обробки запиту і т.п.

1.6 Еволюційний дизайн

Ті, хто практикує мікросервісну архітектуру, зазвичай багато працювали з еволюційним дизайном і розглядають декомпозицію сервісів як подальшу можливість дати розробникам контроль над змінами (рефакторингом) їх застосування без уповільнення самого процесу розробки. Контроль над змінами не обов'язково означає зменшення змін: з правильним підходом і набором інструментів можна робити часті, швидкі, добре контрольовані зміни.

У плані інфраструктури еволюційна декомпозиція дозволяє оновлювати, чи перезавантажувати сервіси без впливу на інші компоненти.

Кожен раз при розбитті додатку на компоненти, спеціаліст стикаєтеся з необхідністю прийняти рішення, як саме ділити додаток. Ключова властивість компонента - це незалежність його заміни або поновлення, що має на увазі наявність ситуацій коли його можна переписати з нуля без порушення взаємодіючих з ним компонентів – це і є межею розділення. Після розіделення дуже важливо перенести інфраструктуру таким чином, щоб не виникали проблеми ізоляції, взаємодії, чи ж то втрати даних[3,4].

Гарним прикладом переносу дуже виликої інфраструктури є веб-сайт Guardian - хороший приклад програми, яка була спроектована і побудована як моноліт, але потім еволюціонувала в бік мікросервісів. Ядро сайту все ще залишається монолітом, але новий функціонал додається шляхом побудови мікросервісів, які використовують API моноліту. Такий підхід особливо корисний для функціональності, яка по суті своїй є тимчасовою. Приклад такої функціональності - спеціалізовані сторінки для освітлення спортивних подій. Такі частини сайту можуть бути швидко зібрані разом з використанням швидких мов програмування і видалені як тільки подія закінчиться. Ми бачили схожий

підхід в фінансових системах, де нові сервіси додавалися під відкрилися ринкові можливості і віддалялися через кілька місяців або навіть тижнів після створення.

Такий упор на замінюваності - окремий випадок більш загального принципу модульного дизайну, який полягає в тому, що модульність визначається швидкістю зміни функціоналу. Речі, які змінюються разом, повинні зберігатися в одному модулі. Частини системи, що змінюються зрідка, не повинні перебувати разом з швидко змінюваними сервісами. Якщо регулярно змінюються два сервісу разом, вартує подумати над тим, що можливо їх слід об'єднати логічно, чи ж то у один контейнер.

Приміщення компонент в сервіси додає можливість більш точного планування релізу. З монолітом будь-які зміни вимагають перекомпіляції і розгортання всього програми. З мікросервісами потрібно перерозгорнути тільки ті сервіси, що змінилися. Це дозволяє спростити і прискорити процес релізу. Недолік такого підходу в тому, що доводиться хвилюватися щодо того, що зміни в одному сервісі зламають сервіси, які звертаються до нього. Традиційний підхід до інтеграції полягає в тому, щоб вирішувати такі проблеми шляхом версійності, але мікросервіси воліють використовувати версійність тільки в разі крайньої необхідності. Можна уникнути версійності шляхом проектування сервісів так, щоб вони були настільки толерантні до змін сусідніх сервісів, наскільки можливо.

1.7 Висновки за розділом

У даному розіді були надані відомості про поняття мікросервісів із точки зору коду та архітектури. Були визначені поняття та переваги підходу мікросервів у побудові сучасних сервісів, згідно з цих відомостей було прийнято рішення побудови логічної мережі мікросервісів на кафедрі СП ННК ІПСА через наступні пункти:

1) Еволюційність дизайну

Для кафедри, де начальний план передбачає велику кількість виконання наукових лабораторних робіт велике значення має швидке розгортання сервісів для, що забезпечать середовище для виконання лабораторних робіт та її перестворення для запобігання плагіату між курсами та групами.

2) Ізоляція

Конце важливо ізолювати сервіси для забезпечення безпеки інформаційних ресурсів кафедри.

3) Незалежність елементів інфраструктури

При забезпеченні незалежності можна легко перестворювати великі елементи інфраструктури без впливу на користувача.

2 Побудова мережі

У данному розділі ми приступимо до аналізу та вибору механізму оркестрації (основни кластеру), що буде запускати контейнери із мікросервісами, у які буде власне кажучи, покладений логічний функціонал мережі. Враховуючи, що метою проекту є модернізація мержі кафедри – одразу приберемо такі оркестратори, як то AWS EC2, або ж Azure Container service через те, що вони є вендорними та пропрієтарними для вказаних у назві клауд провайдерів.

На даний момент існує декілька основних (популярних та підтримуваних) оркестраторів для мікросервісних контейнерів, у яких будуть працювати згодом кафедральні додатки та сервіси це:

1) Docker swarm
2) Kubernetes
3) Mesos

Таблиця 2.1 Коротке інформація згідно сайтів оркестраторів

Оркестратор	Розробник	Рік випуску	Статус	Підтримка	Розмір спільноти
Docker Swarm	Docker	2015	Активний, але перенесений у основний Docker Engine	+	4k+ Stars, 800+ Forks, 150+ Contributors

Таблиця 2.1 Коротке інформація згідно сайтів оркестраторів

Оркестратор	Розробник	Рік випуску	Статус	Підтримка	Розмір спільноти
Docker Swarm Mode	Docker	2016	Активний	+	1k+ Stars, 200+ Forks, 70+ Contributors
Kubernetes	CNCF	2015	Активний	Від третіх сервісів	20k+ Stars, 7k+ Forks, 1k+ Contributors
Mesos	Apache Software Foundation	2016	Активний	+	2k+ Stars, 1k+ Forks, 200+ Contributors

Таблиця створена згідно наступних джерел

[34]-[35] включно

Таблиця 2.2 Порівняння Обраних оркестраторів

Features	**Docker Swarm**	**Docker Swarm mode**	**Kubernetes**	**Mesos**
Архітектура планувальника	Монолітний	Розподілений Стан	Розподілений Стан	Двох рівневий
Агностичність	-	-	Docker RKT	Docker RKT Інші
Service Discovery	Відсутня, потрібні треті сервіси	Нативна підтримка	Навтивна підтримка за допомогою вбудованого DNS кластера чи за	Так, за допомогою Mesos-DNS

Таблиця 2.2 Порівняння Обраних оркестраторів

Features	**Docker Swarm**	**Docker Swarm mode**	**Kubernetes**	**Mesos**
			допомогою енв. Змін.	
Secret management	-	Присутня, за допомогою Docker Secret Management	Навтивна підтримка	Нативно відсутня, підтримка можлива, якщо присутня у фреймворку
Configuration management	Через змінні середовища у compose файлі	Через змінні середовища у compose файлі files	Нативна підтримка через ConfigMap	Нативно відсутня, підтримка можлива, якщо

Таблиця 2.2 Порівняння Обраних оркестраторів

Features	**Docker Swarm**	**Docker Swarm mode**	**Kubernetes**	**Mesos**
			or Інжекція за допомогою Через змінних середовища у	присутня у фреймворку
Logging	Потребує налаштування дрейвера логування , що буде передавати логи на треті	Потребує налаштування дрейвера логування , що буде передавати логи на треті	Можн тільки відпраляти логи на третичні сервіси	За доромогою ContainerLogger or або ж якщо його надає фреймворк

Таблиця 2.2 Порівняння Обраних оркестраторів

Features	**Docker Swarm**	**Docker Swarm mode**	**Kubernetes**	**Mesos**
	сервіси, як то ELK	сервіси, як то ELK		
Monitoring	Потребує треті сервіси для моніторингу усіх контейнерів	Потребує треті сервіси для моніторингу усіх контейнерів	За допомогою Heapsters може проводити базовий моніторинг	Надсилає метрики на моніторинг третіх сервісів
High-Availability	Нативна підтримка створення	Навтивна підтримка	Нативна підтримка завдяки	Нативна підтримка, завдяки багатьом

Таблиця 2.2 Порівняння Обраних оркестраторів

Features	**Docker Swarm**	**Docker Swarm mode**	**Kubernetes**	**Mesos**
	багатьох менеджерів		реплікації мастерів	масйстрам із вибором через ZooKeeper
Load balancing	-	Можлива, шляхом балнсування портів сервісами балансування	За потреб зовнішній балансувальник створюється перед сервісом	Надається фреймворком, наприклад Marathon
Networking	+	+	Потребує інших сервісів для створення	Потребує третіх сервісів для

Таблиця 2.2 Порівняння Обраних оркестраторів

Features	**Docker Swarm**	**Docker Swarm mode**	**Kubernetes**	**Mesos**
			оверлей мережі	мережевої взаємодії
Application definition	Docker Compose файли	Docker Compose стеки та файли	Використовує yaml для створення будь-яких об’єктів	Залежить від фреймворку
Deployment	Docker Compose	Підтримує роллбек та апдейт стратегії	Нативна підтримка розгортки, якщо вказана стратегія	Залежить від фреймворку

Таблиця 2.2 Порівняння Обраних оркестраторів

Features	**Docker Swarm**	**Docker Swarm mode**	**Kubernetes**	**Mesos**
		Docker Compose		
Auto-scaling	-	-, але має простий механізм ручного масштабування	Нативна підтримка у рамках, вказаних для пода	Залежить від фреймворку
Self-healing	-	+	+	Залежить від фреймворку
Stateful support	Використання томів даних	Використання томів даних	За допомогою StatefulSets	Використання томів даних

Таблиця 2.2 Порівняння Обраних оркестраторів

Features	**Docker Swarm**	**Docker Swarm mode**	**Kubernetes**	**Mesos**
			чи Використання томів даних	
Documentations	Плутана, через перенiс Swarm у основний Docker Engine	Плутана, через перенiс Swarm у основний Docker Engine	Плутана через перенiс документації із github.io на kubernetes.io	Детальна, централізована

Таблиця створена за наступними посиланнями:

[11]-[32] включно

На основі таблиці порівнянь оберемо основних претендентів, це:

1) Kubernetes
2) Docker Swarm
3) Mesos DCOS

Трохи детальніше розглянемо Kubernetes та Mesos, як основні, оскільки Docker Swarm, хоч і є доволі простим та оегким до опанування, але не надасть усього необхідного функціоналу, в основному нас більше цікавить можливість розгортати поди Кубернетес (тут і надалі будемо використовувати укаїнську назву продукту), а також гнучкість керування DCOS.

Перед детальним розгляданням продуктів, давайте сформуємо належним чином потреби мережі для кафедри СП ННК ІПСА:

1) Гнучкість розгортання
2) Надійність сервісу
3) Балансування навантаження
4) Ізольованість ресурсів
5) Гранулярність доступу
6) Легкість опанування
7) Розвинутий ком’юніті
8) Сервіс має бути безкоштовним

Сформовані вимоги є мінімальними і можуть не відображати усіх потреб мережі, особливо в рамках enterprise, проте є необхідними та достатніми для навчальної мережі. Що стосовно вимог, то розглянемо кожну з них детальніше та обґрунтуємо їх :

1) Гнучкість розгортання:

   Сама система має пропагувати максимальну доступну легкість розгортання контейнерів із сервісом за мінімальний час з максимальною прозорістю. Це забезпечить:

a. Швидкодію оновлення
b. Швидке опанування системою навчальними майстрами кафедри
c. Можливість оновлення будь-якої кількості елементів мережі без впливу на іншіі компоненти

2) Надійність:

   Враховуючи важливість даного сервісу у загальній логічній топології мережі – конче необхідно щоб сервіс був надійним, підтримував перерозгортання контейнерів у разі відмови якогось з них

3) Балансування навнтаження:

   Оптимальним буде, якщо сервіс може «із коробки» підтримувати балансування навантаження між контейнерами. Це матиме одразу два позитивних моменти:

a. Зниження кількості різноманітності сервісів, що позитивно повпливає на «поріг входження» у систему для новачка
b. Знизить рівень абстракції, що збільшить швидкодію per se знизить оверхед (overhead – накладні витрати, у контексті Information Technologies це додатковий, адитивний параметр, що означає час Δt який випливає із проходження інформації через конвеєр абстракції, виражається у відсотках. Тут і надалі буде використовуватись у вигляді транслітерації).

4) Ізольованість ресурсів:

   Матиме величезну роль для безпеки інформаційних ресурсів кафедри. Основним тут буде легкість створення віртуальних мереж, та взаємодія трафіку між ними, також велике значення має легкість керування цим ресурсом.

5) Гранулярність доступу:

Для компоненту, який матиме глобальний вплив на мережу абсолютною потребою має бути розділення доступу до ресурсу(ів) таким чином, щоб недосвідчений працівник не зміг своєю діяльністю повпливати на працездатність системи загалом. У рамках кафедри буде дуже легко реалізувати завдяки підключенню до домену Active Directory, що вже існує.

6) Легкість опанування:

Легкість опанування технологією матиме ключовий момент у діяльності підтримки сервісу це стосується:

a. Написання конфігурації, що дозволить швидко розгортати нові ресурси, як то мержі і контейнери
b. Легкість опанування API (application programming interface), за наявності такого, що дозволить створення сервісів студентами/працівниками кафдери, як то до прикладу WEB інтерфейс створення ресурсу контейнеру для виконання курсу лабораторних робіт.

7) Розвинена Спільнота:

Має не останню роль, оскільки підтримка на таких ресурсах, як stackoverflow та serverfault допоможе швидше вирішувати задачі, а регулярні оновлення – «зашивати дирки» у системі безпеки мережі та сервісу. Також це позитивно вплине на можливість міграції

8) Безкоштовність:

Мало що можна сказати, але на превеликий жаль у реаліях України безкоштовність сервісу має велику роль.

Повернемось до описання обраних «перетендентів»

1) Docker Swarm:

Розглянемо основні моменти:

a. Не агностичний – підтримка тільки контейнерів докер

b. Доволі стабільна робота, через постійну підтримку та механізм самопоновлення (перезапуск контейнерів, що зупинилися)
c. Внутрішній балансувальник – відсутній
d. Внутрішнє нативне логування відстунє
e. Моніторинг – нативний відстуній
f. Пристуній нативний драйвер мережі
g. Автоматичне масштабування – відсутнє
h. Документація - доволі заплутана, через переніс сервісу у ядро іншого

2) Kubernetes
a. Агностичний – підтримка dokcer/rkt
b. Стабільний
c. Присутній внтурішній балансувальник подів.
d. Логування – пристунє, проте так само найкращим виходом буде використання ELK
e. Моніторинг – базовий присутній, що надає запис базових метрик у бекенд, вказаний користувачем
f. Нативний драйвер мережі – відстутній
g. Автоматичне масштабування – присутнє
h. Документація- доволі струтурована

3) Mesos
a. Агностичний – підтримка dokcer/rkt/інших драверів
b. Стабільний
c. Присутній внтурішній балансувальник подів.
d. Логування – пристунє, проте так само найкращим виходом буде використання ELK

e. Моніторинг – базовий присутній, що надає запис базових метрик у бекенд, вказаний користувачем
f. Нативний драйвер мережі – відстутній
g. Автоматичне масштабування – залежить від фреймворку
h. Документація- дуже детальна та структурована

Номад був відкинутий через недостатню розвиненість, та відмовою самого вендора від продукту через неконкурентність.

2.1 Порівняння основних конкурентів

Проведемо більш детальне порівняння між обраними трьома варіантами, щоб запропонувати один

2.1.1 Порівняння 1

Мезос – Кубернетес

Кубернетес

Кубернетес, згідно офіційної інформації є системою із відкритим кодом для керування автоматичним розгортанням, масштабуванням та керуванням контейнеризованих додатків. Розробник – Google, який вклав туди свій досвід роботи з контейнерами за останні роки. Розглянемо архітектуру кубернетес на рисунку нижче

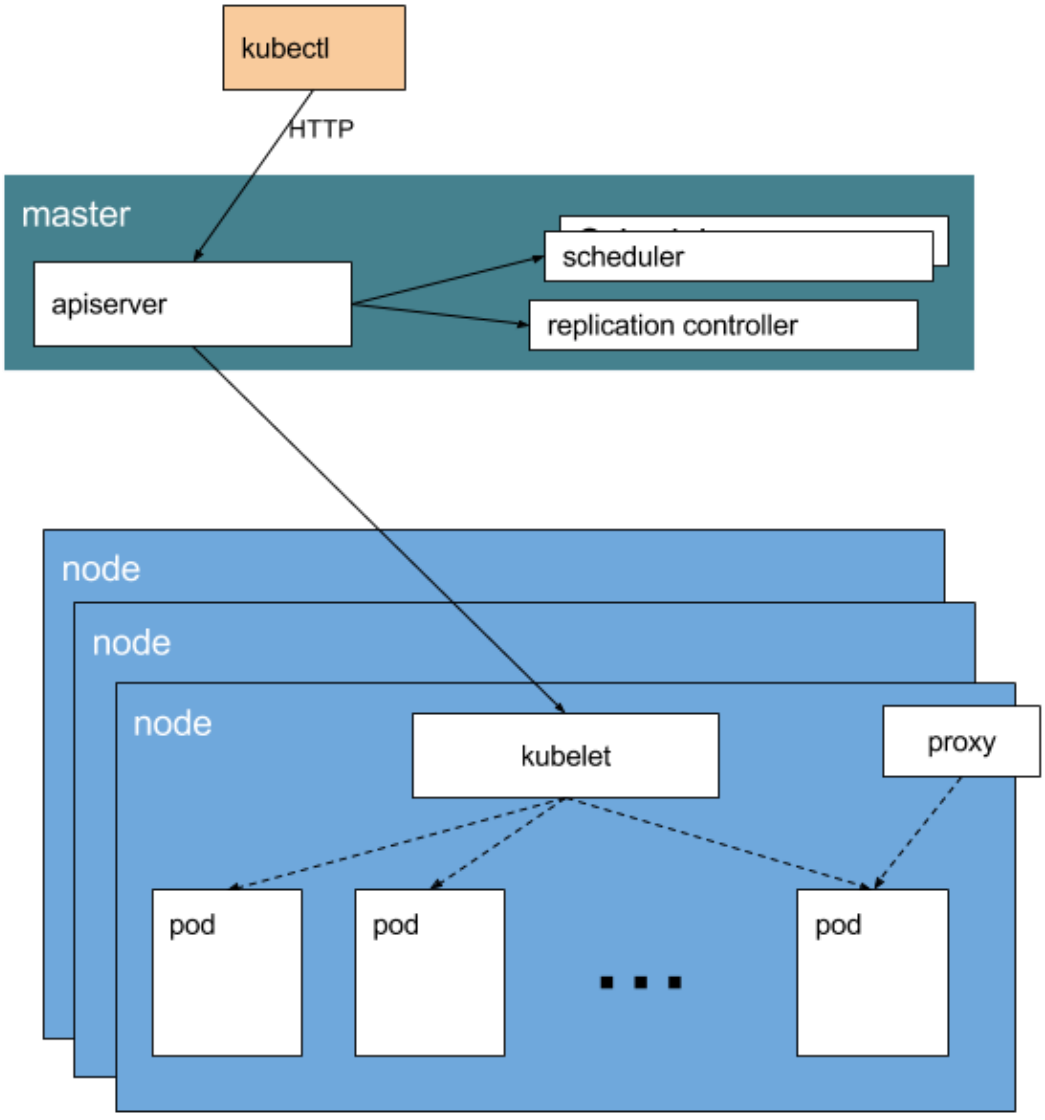

Рис. 2.1 – Архітектура Kubernetes

Головні компоненти

- PODS – поди. Кубернетес розгортає і планує контейнери у групах, що називаються подами. Зазвичай под містить 1-5 контейнерів, що співпрацюючи утворюють сервіс.
- Flat Networking Space – однорангова плоска мережа. Стандартна модель мережі Кубернетес довзоляє спілкуватись усім подам між собою. Контейнери у тому

самому поді поділяють один IP та комунікують, використовуючи порти на локалхості.

- Labels – мітки. Мітки є парами ключ-значення, що прикріплені до об”єктів та можуть бути використані для пошуку та оновлення багатьох об”єктів в одному наборі.
- Services – сервіси є кінцевими точками, до яких можна звертатись за ім”ям та можуть бути пов”язані із подами, використовуючи селектори міток. Сервіси будуть автоматично розсилати запити за алгоритмом round-robin між подами
- Replication Controllers – Контролери реплікації. Абстракція подів, та їх представлення у системі. Відповідають за контроль та моніторинг над кількістю працюючих подів для сервісу, покращуючи стійкість до помилок.

2.1.2 Порівняння 2

Мезос - Сварм

Apache Mesos – кластерний менеджер із відкритим кодом, розроблений для масштабування до дуже великих розмірів із великою кількістю хостів (наприклад Twitter). Мезоз підтримує велику кількість корисних навантажень, як то до прикладу завдання Hadoop, підтримка клауд-нативних додатків. Архітектура Мезоз розроблялась навколо понять надійності, еластичності та високої доступності.

Розглянемо його архітектуру

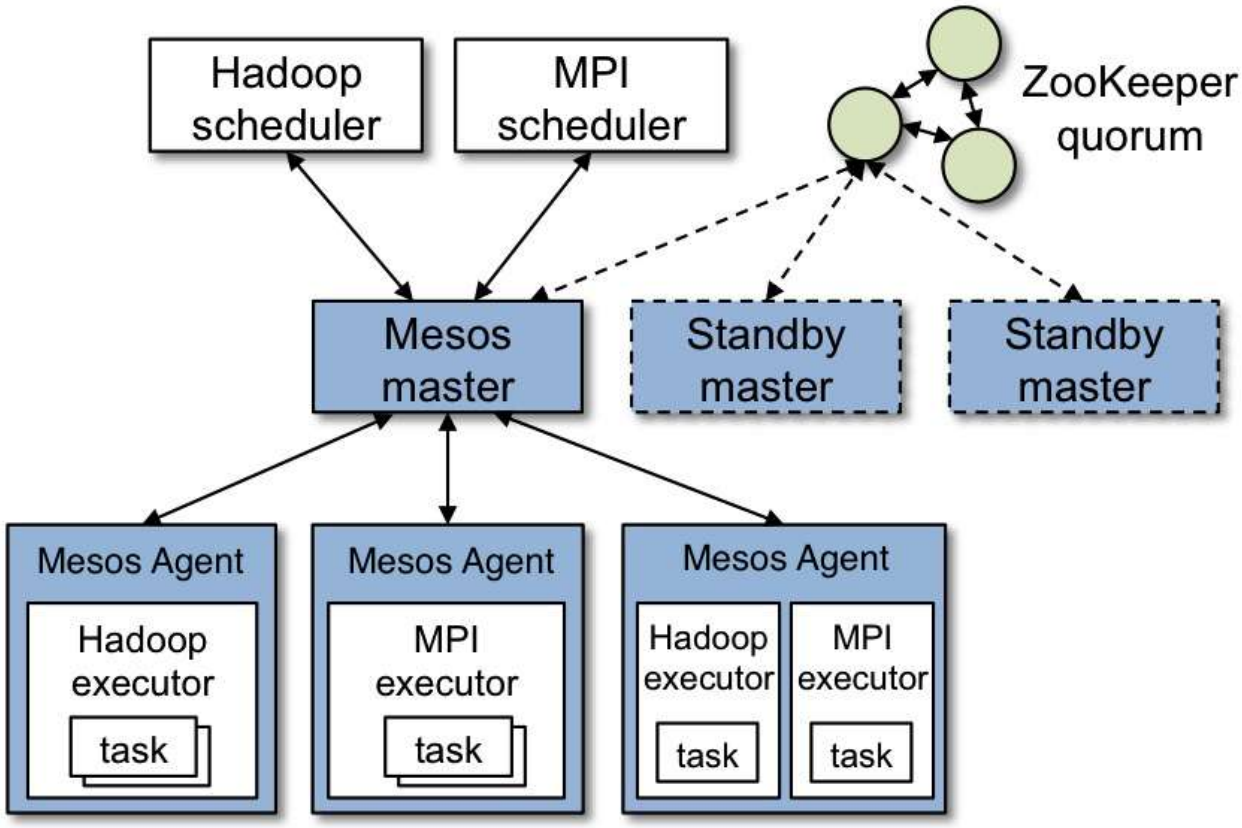

Рис 2.2 – Архітектура MESOS

Основні компоненти

- Mesos Agent Nodes – відповідає за роботу завдань. Усі агенти надають мастеру список доступних ресурсів
- Mesos Master – мастер відповідає за розсилання завдань агентам. Мастер зберігає список ресурсів та надає їх фреймворкам, як то Hadoop, до прикладу. Мастер вирішує скільки ресурсів необхідно виділити, основуючись на стратегії виділення. Завжди має бути дублюючий майстер, що використовується, як «гаряча» заміна, в разі падіння системи.
- ZooKeeper – використовується для обирання майстра. Завжди запущено декілька майстрів, що забезпечують стійкість до сбою системи

- Frameworks – Фреймоврки координують із мастером для планування задачі на агентних нодах. Фреймворк складається з двох частин
- Виконуючий процес, що працює на агентах та відповідає за роботу завдань
- Планувальник, що працює із мастером, та на основі списку мастера вибирає ресурси, на яких будуть підняті контейнери

Може існувати безліч фреймворків, що працюють на основі кластера Мезос. Користувачі воліють працювати із фрейморком, а не з мезосом напряму, що є цілком логічно, через складність взаємодії із планувальником мезос, але про це згодом.

На рисунку вище вказана схема, де мезоз працює із фрейворком Marathon, що працює як планувальник. Планувальник Marathon використовує ZooKeeper для визначення мастера, якому власне і буде відправляти завдання. Обидва і майстер, і планувальник мають копії, що страхують сервіс від відмови у разі падіння одного з них.

Марафон, створений Мезосферою розроблений для запуску, моніторингу та масштабування довго працюючих додатків, враховуючи запуск додатків орієнтованих на клауд. Клієнти взаємодіють із Мезос через REST API. Інші функції включають в себе підтримку функціоналу моніторингу та інтеграції із балансувальниками.

Порівняльна таблиця запропонована нижче:

Таблиця 2.3 – Порівняльна таблиця Kubernetes vs. Mesos		
	Kubernetes	**Mesos**
Типи навантажень	Нативні клауд додаки	Нативні клауд додакт та усілякі інші
Визначення додатків	Комбінація Подів, Сетів, Контролерів.	Модель "Application Group" - дерево залежностей, що запускається в якомусь порядку.
Масштабованість	Кожний додаток визначений власним подом, що може бути масштабований вручну, або автоматично коли керується Деплойментом, чи Контролером Реплікації.	Може масштабувати окрему групу, усі залежні у дереві теж замасштабуються
Високодоступність	Поди дистрибуються серед Робочих Нод. Сервіси також підлягають високодоступності завдяки пошуку	Додатки дистрибуються серед Робочих нод

Таблиця 2.3 – Порівняльна таблиця Kubernetes vs. Mesos		
	Kubernetes	**Mesos**
	непрацюючих нод та їх рестарту.	
Баласнування Навантаження	Поди балансуються через сервіси балансувальник навантаження	Через Mesos DNS, що може працювати як рудаментарний балансувальник.
Автомасштабування Додатку	Автоматичне Масштабування використовуючи деякий набір подів задається через API Replication Controllers. Модель кількості використання процесорного часу зявилась у версії 1.1	Доступний Rate-sensitive autoscaling для версії ентерпрайз Mesosphere
Оновлення додаку Та відкат у разі провалу	Модель Деплойменту підтримує різноманітні стратегії Тести на	"Rolling restarts" model uses application-defined minimumHealthCapacity (ratio of nodes serving new/old

Таблиця 2.3 – Порівняльна таблиця Kubernetes vs. Mesos		
	Kubernetes	**Mesos**
	робочездатність додатків	application) "Health check" hooks consume a "health" API provided by the application itself
Логування та моніторинг	Перевірки двох типів 1) Працює/Ні 2) Готовий/Ні Перевірка Ресурсу Heapster/Grafana/Influx	Логування: ELK Моніторинг: Використовуються треті сервіси
Сховище	Два API для сховища 1) Перший надає абстракції для бекенду (NFS, AWSm EBS, ceph, flocker) 2) Другий надає абстракції на виділення самого ресурсу, наприклад у гігабайтах із будь якої абстракції першого рівня	Контейнери Марафон можуть використовувати постійні сховища, проте сховища постійно інсуюють тільки на тій ноді, де створені. Експериментальна інтеграція flocker підтримує існування

Таблиця 2.3 – Порівняльна таблиця Kubernetes vs. Mesos

	Kubernetes	**Mesos**
	Модифікація ресурсів, що використовується Докер Демоном потребує тимчасоого виключення ноди із кластеру	не тільки локальних нод.
Мережа	Мережева модель надає будьякому контейнеру дає будь якому контейнеру зв'язуватись з іншими подами чи сервісами Та потребує дві мережі: 1) Для сервісу 2) Для Пода У стоці обидві мережі не мають бути доступні з назовні	Інтеграція порт-хост За замовчуванням контейнер не має своєї адреси, але якщо є інтеграція із Calico – то матиме. Проте навіть по при це контейнери не можуть поділяти один неймспейс

Таблиця 2.3 – Порівняльна таблиця Kubernetes vs. Mesos		
	Kubernetes	**Mesos**
Занходження сервісів	Можуть занходити одне одного за допомогою intra-DNS	Треті сервіси автзнаходження
Робота та масштабування	Реліз 1.2 – одночасно 1000 нод API відповідає 99% часу 99% подів запускаються за 5 секунд, якщо образ вже є	Може працювати за 50000 нод, також може запускати LXC, Docker, Kubernetes Та бути гіпервізором

Докер Сварм

Згідно офіційного сайту Docker Swarm надає нативні можливості кластеризації для уможливлення об'єднання групи самостійно працюючих докерів у один віртуальний докер. Swarm використовує стандартний Docker API, таким чином стандартна команда docker run може бути використана для запуску контейнерів а Swarm подбає про вибір підходящого хоста для запуску гостьового контейнеру. Інші інструменти, що використовують
Docker API, наприклад Docker Compose, використовуватиме Swarm без змін.

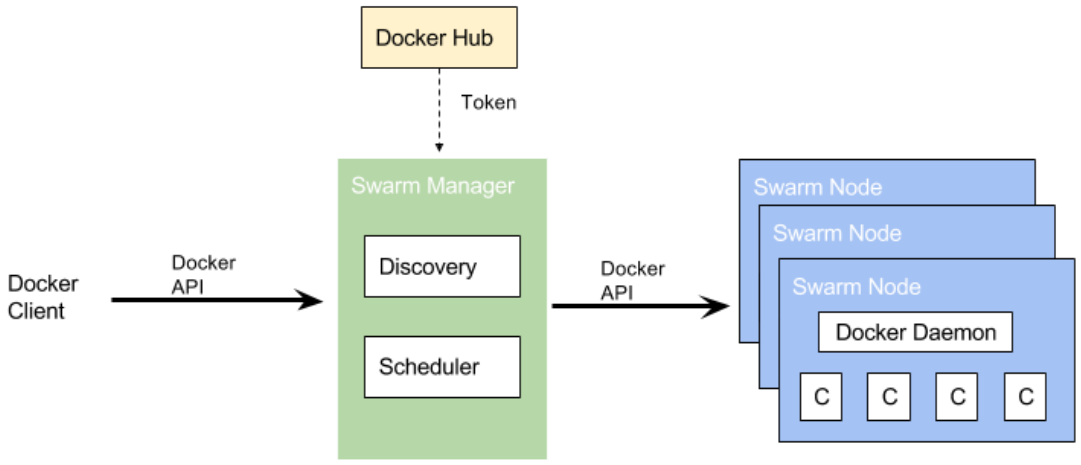

Рис 2.3 – Архітектура Swarm

Кожен хост у кластері, окрім мастера має запущений процес агента, мастер в свою чергу грає роль менеджера і володіє процесом Docker Swarm Manager. Менеджер оркеструє та планує запуск контейнерів на хості. Так само, як і інші оркестратори сервіс знаходження (Discovery) буде знаходити і додавати машини у кластер. Інструменти, такі як Consul, ZooKeeper, etcd потребуються для забезпечення відмовостійкості за рахунок переключення на запасний менеджер.

Таблиця 2.4 Порівняльна характеристика Kubernets vs. Swarm Mode

	Kubernetes	**Swarm**
Типи навантажень	Нативні клауд додаки	Нативні клауд додаки

Таблиця 2.4 Порівняльна характеристика Kubernets vs. Swarm Mode		
	Kubernetes	**Swarm**
Визначення додатків	Комбінація Подів, Сетів, Контролерів.	Запускаються у кластері. Фільтр прискорює запуск за рахунок оптимізації
Масштабованість	Кожний додаток визначений власним подом, що може бути масштабований вручну, або автоматично коли керується Деплойментом, чи Контролером Реплікації.	Може окремо балансувати додатки у Docker Compose
Високодоступність	Поди дистрибуються серед Робочих Нод. Сервіси також підлягають високодоступності завдяки пошуку непрацюючих нод та їх рестарту.	Розприділяються у кластері. Менеджер, що відповідає за весь кластер присутній має репліки. Запити на репліки

Таблиця 2.4 Порівняльна характеристика Kubernets vs. Swarm Mode		
	Kubernetes	**Swarm**
		автоматично передаються на головну ноду. Якщо ж основний менеджер став не робочим, то ZooKeeper,чи etcd обирути новий.
Баласнування Навантаження	Поди балансуються через сервіси балансувальник навантаження	Зазвичай балансувальник – ще один додаток у Docker Compose
Автомасштабування Додатку	Автоматичне Масштабування використовуючи деякий набір подів задається через API Replication Controllers.	Не доступне напряму.

Таблиця 2.4 Порівняльна характеристика Kubernets vs. Swarm Mode		
	Kubernetes	**Swarm**
	Модель кількості використання процесорного часу зявилась у версії 1.1	
Оновлення додаку Та відкат у разі провалу	Модель Деплойменту підтримує різноманітні стратегії Тести на робочездатність додатків	Під час розгортки можна ставити паузи між різними групами додатків, надаючи час на функціональний старт, у разі падіння можна відкатитсь до попередньої версії.
Логування та моніторинг	Перевірки двох типів 3) Працює/Ні 4) Готовий/Ні Перевірка Ресурсу Heapster/Grafana/Influx	Логування: ELK Моніторинг: Треті сервіси

Таблиця 2.4 Порівняльна характеристика Kubernets vs. Swarm Mode		
	Kubernetes	**Swarm**
Сховище	Два API для сховища 3) Перший надає абстракції для бекенду (NFS, AWSm EBS, ceph, flocker) 4) Другий надає абстракції на виділення самого ресурсу, наприклад у гігабайтах із будь якої абстракції першого рівня Модифікація ресурсів, що використовується Докер Демоном потребує тимчасоого виключення ноди із кластеру	Дозволяє монтувати директорії на контейнер, та томи За замовчуванням зберігається локально, проте підтримує зберігання на різних бекендах.
Мережа	Мережева модель надає будь-якому контейнеру дає будь якому контейнеру зв’язуватись з іншими подами чи сервісами Та потребує дві мережі: 3) Для сервісу	Може створювати однорангову мережу як на ноді, так і на кластері, за замовчуванням трафік –

Таблиця 2.4 Порівняльна характеристика Kubernets vs. Swarm Mode		
	Kubernetes	**Swarm**
	4) Для Пода За замовчуванням обидві мережі не мають бути доступні з назовні	шифрований. Якщо використовується одна і та ж Мережа то контейнери завжди комунікують
Занходження сервісів	Можуть занходити одне одного за допомогою intra-DNS	Intra-DNS Рекомендовано: Docker libkv project. 1) Consul 0.5.1 чи вище, 2) Etcd 2.0 чи вище, 3) ZooKeeper 3.4.5 чи вище .
Робота та масштабування	Реліз 1.2 – одночасно 1000 нод API відповідає 99% часу 99% подів запускаються за 5 секунд, якщо образ вже є	Від 1000 до 50000 нод із інкрементальним наватнаженням

Проаналізувавши потреби кафедри та порівнявши запропоновані варіанти – оберемо Kubernetes, за наступними критеріями:

1) Популярність – інструмент постійно підтримується
2) Найкращий із пердставлених механізм самовідновлення – контейнери, що впали відновляться одразу ж після падіння, на відміну від того, що інші оркестратори просто відмічають, контейнер як не функціонуючий.
3) Легкість користування, на відміну від Mesos – писати планування сервісів значно простіше
4) Найстабільніший API серед претендентів – 99% відповідей гарантовані, найдовший час відповіді – 1 секунда.

Обравши преетендента на роль оркестратора, ознайомимось з системою віртуалізації, що зараз успішно працює на кафедрі – Citrix XenServer.

Citrix Xen працює на кафедрі як DomO система – тобто MaaS, як гіпервізор поверх «заліза» із мінімальною абстракцією, надаючи послуги розміщення віртуальних машин. На даний час там позгорнуті декілька серверів, що забезпечують функціонал обслуговування мережі.
Враховуючи абсолютно різні технологію вртуалзіації – ми не будемо порівнювати дві технології, проте вони цілком можуть співіснувати – Kubernetes у даному проекті буде встановлений у якості хоста на сервер віртуалізації Xen, таким чином буде забезпечна ізоляція ресурсів, та зручність розгортання.

2.2 Висновки за розділом

У цьому розділі було обрано оптимальний оркестратор на роль центрального сервісу мережі для мікросервісів кафедри згідно потреб, що були визначені вище. Вартує зазначити, що можна використовувати декілька оркестраторів, ускладнивши рівень абстракції, але на даному етапі це не є необхідним, оскільки значне поскладення абстракції погіршить швидксть роботи та продуктивність. Іншою стороною цього рішення є моральна застарілість фізичних серверів кафедри та зношеність жорстких дисків – сервери не мають великої кількості логічних ядер, отже мікросервісні машини можуть вийти доволі слабкими, а велике навантаження вводу-виводу скоріше за все почне швидко виводити жорсткі диски з робочого стану

2.3 Практична частина

До початку практичної частини, визначимо потреби інсталяції.

2.3.1 Загальний вигляд технічного завдання:

Кластер Kubernetes буде встановлений, як група віртруальних машин на Citrix XenServer. У якості мережі та балансувальника навантаження буде використанний Ingress у звязці із Cintrix Netscaler. Навчальні сервіси будуть розглянуті нижче. Сервіс документообігу буде розгорнутий у середовищі Kubernets, але для надійності продубльований, як віртуальна машина рівня абстракії Xen –для надійності. Kubernetes буде встановлений на операційній системі CoreOS, з огляду через офіційну підтримку та легкість конфігурації.

2.3.2 Обгрунтування

У даній роботі ми знехтуємо описанням процесу виділення та створення віртуальних машин, через його примітивність, вартує зауважити, що це займатиме дві-три хвилини на кожну, а отже перейдемо до інсталяції кластеру.

Коментарий: Дл початку Обгрунтуемо вибір ОС

CoreOS.

На початку розробки системи було виявлено, що CoreOS є альтернативним вибором, у порівнянні із тредиційним *nix, як то CentOS, RedHat, Fedora, debian і т.п. Тому було не дуже зрозуміло, чому багато ресурсів та статей про інсталяцію кластеру уникали використання класичного лінуксу, який надає систему, де може бути встановлено одночансо багато не «клауд»/«мікросервісних» систем одночасно. У той же час CoreOs розроблена спеціально для роботи з контейнерами. Система не пристосована для роботи із багатьма високорівневими системами на одному хості, як то бази даних, поштові сервери, чи комплексні додатки – тому є дуже легкою. Кожен сервіс працює у окремому контейнері, у CoreOS, що чудово підходить під філософію kubernetes та мікросервісів загалом. Окрім того великим плюсом є те, що CoreOS є одним із найважливіших контрибютерів kubernetes, тому ця ОС є “bliding-edge” (найсучасніша, у IT bleeding edge є філософським підходом, де киористовується найновіші розробки), а тому гарно підходить, через вчасне виявлення системних помилок та їх регулярне виправлення.

Більш того – CoreOs це вдалий пыдхыд для надання універсального розгортання нових машин – так званого “cloud-config”. Технологія себе гарно зарекламувала у такому сервісі, як EC2 від AWS, де необхідні команди, у тестовому вигляді, передаються через “User Data” – поле у формі створення машини. Ці команди одразу виконуються після завантаження системи. Перевагою існування цього поля є те, що за допомогою різноманітних інструментів Configuration Management, наприклад Ansible, чи то Infrastructure As a Code, наприклад

Terraform, можна одразу підняти велику кількість віртуальних хостів, не витрачаючи часу на індивідуальне налаштування.

Ingress – балансування on premise

Балансування у данній побудові буде грати роль як gateway, так і власне балансувальника загалом. На даному етапі зупинимось на двох варіантах побудови

1) Kubernetes Ingress + Citrix Netscaler
2) Nginx + DNS balancing

Обидва варіанти побудови мають свої переваги та недоліки, порівняння буде запропоновано після описання обох. Вибір буде за тим, який надасть більше можливостей, при обмежених ресурсах кафедри у плані baremetal.

2.4 Виконання ТЗ з встановлення кластеру.

2.4.1 Варіант 1

Описання системи[3,4]

1) Hypervisor –Xen
2) Dom0 – CentOS (Xen включений в неї)
3) CoreOS – хост система для Kubernetes

a. Master Node
b. Node 1
c. Node 2
d. Node 3

Інсталяція кубернетес

```
sudo su
mkdir -p /var/lib/libvirt/images/
```

```
cd /var/lib/libvirt/images/
git clone -b blog_post_v2
https://github.com/andrewmichaelsmith/xen-coreos-
kube.git coreos
cd coreos
wget https://beta.release.core-os.net/amd64-
usr/current/coreos_production_xen_image.bin.bz2 -O -
| bzcat > coreos_production_xen_image.bin
```

Створення дискового простору для мастера та трьох нод

```
#To pad out extra space in the image.
#TODO: Undoubtedly a better way than this
dd if=/dev/zero of=tempfile bs=1G count=2
cat tempfile >> coreos_production_xen_image.bin

cp coreos_production_xen_image.bin master1.bin
cp coreos_production_xen_image.bin node1.bin
cp coreos_production_xen_image.bin node2.bin
cp coreos_production_xen_image.bin node3.bin
```

Генерація SSH ключа

ssh-keygen -t rsa -b 4096 -C "$USER@$HOSTNAME"

та інсталяція на машини

```
KEY=$(cat ~/.ssh/id_rsa.pub)
sed "s#SSH_KEY#$KEY#g" < master1/openstack/latest/user_data.tmpl >
master1/openstack/latest/user_data
sed "s#SSH_KEY#$KEY#g" < node1/openstack/latest/user_data.tmpl >
node1/openstack/latest/user_data
sed "s#SSH_KEY#$KEY#g" < node2/openstack/latest/user_data.tmpl >
node2/openstack/latest/user_data
```

```
sed "s#SSH_KEY#$KEY#g" < node3/openstack/latest/user_data.tmpl >
node3/openstack/latest/user_data
```

Генерація сертифікатів

```
cd certs
openssl genrsa -out ca-key.pem 2048
openssl req -x509 -new -nodes -key ca-key.pem -days
10000 -out ca.pem -subj "/CN=kube-ca"
openssl genrsa -out apiserver-key.pem 2048
openssl req -new -key apiserver-key.pem -out
apiserver.csr -subj "/CN=kube-apiserver" -config
openssl.cnf
openssl x509 -req -in apiserver.csr -CA ca.pem -
CAkey ca-key.pem -CAcreateserial -out apiserver.pem
-days 365 -extensions v3_req -extfile openssl.cnf
cd ..
```

Та їх встановлення на мастер ноди

```
#Total hack, so it's indented correctly when we move
it in to .yml
sed -i 's/^/        /' certs/*.pem
sed -i $'/CA.PEM/ {r certs/ca.pem\n d}'
master1/openstack/latest/user_data
sed -i $'/APISERVER.PEM/ {r certs/apiserver.pem\n
d}' master1/openstack/latest/user_data
sed -i $'/APISERVER-KEY.PEM/ {r certs/apiserver-
key.pem\n d}' master1/openstack/latest/user_data
```

Перевіримо налаштування

```
curl 'https://validate.core-os.net/validate' -X PUT
--data-binary '@master1/openstack/latest/user_data'
| python -mjson.tool
curl 'https://validate.core-os.net/validate' -X PUT
--data-binary '@node1/openstack/latest/user_data' |
python -mjson.tool
```

```
curl 'https://validate.core-os.net/validate' -X PUT
--data-binary '@node2/openstack/latest/user_data' |
python -mjson.tool
curl 'https://validate.core-os.net/validate' -X PUT
--data-binary '@node3/openstack/latest/user_data' |
python -mjson.tool
```

Якщо усі команди передали “null” – результат є успішним.

Тоді:

1) Створюємо iso образ для того, щоб завнатажити конфігурацію до віртуальної машини xen

```
mkisofs -R -V config-2 -o master1-config.iso
master1/
mkisofs -R -V config-2 -o node1-config.iso node1/
mkisofs -R -V config-2 -o node2-config.iso node2/
mkisofs -R -V config-2 -o node3-config.iso node3/
```

2) Створюємо віртульні машини CoreOS

```
xl create master1.cfg
  xl create node1.cfg
  xl create node2.cfg
  xl create node3.cfg
```

Описаний вище процес запустить 4 віртуальні машини із cloud-config.

Сам процес

- Завантаження образу flannel
- Kubelete запуститься і завнтажить hypercube

- Запустяться контейнери для
  - Api server
  - Controller manager
  - Scheduler на мастері
  - Контейнери для kube-proxy стартують на нодах

Для моніторингу процесу – підключимось до консолі та промоніторимо завантаження ноди

```
xl console master1
```

А також journalctl на мастері

```
ssh  core@master
journalctl  -f
```

Встановимо kubectl

```
curl -O https://storage.googleapis.com/kubernetes-
release/release/v1.2.3/bin/linux/amd64/kubectl
chmod +x kubectl
mv kubectl /usr/local/bin/kubectl
```

Перевіримо статус кластера із dom0 системи

```
root@xen# kubectl -s http://192.168.122.254:8080
get nodes
NAME              STATUS    AGE
192.168.122.2     Ready     1m
192.168.122.254   Ready     1m
192.168.122.3     Ready     1m
192.168.122.4     Ready     1m
```

Тепер створимо іменний простір, без якого, власне кажучи не буде працювати резолв, та назвемо його kube-system.

```
curl -H "Content-Type: application/json" -XPOST -
d'{"apiVersion":"v1","kind":"Namespace","metadata":
{"name":"kube-system"}}'
"http://192.168.122.254:8080/api/v1/namespaces"
```

тепер перевірка надасть результат із зрозумілим розбиттям

```
NAME                                          READY
STATUS              RESTARTS   AGE
kube-apiserver-192.168.122.254                1/1
Running             0          3m
kube-controller-manager-192.168.122.254       1/1
Running             1          4m
kube-proxy-192.168.122.2                      1/1
Running             1          4m
kube-proxy-192.168.122.254                    1/1
Running             0          3m
kube-proxy-192.168.122.3                      1/1
Running             0          3m
kube-proxy-192.168.122.4                      1/1
Running             0          3m
kube-scheduler-192.168.122.254                1/1
Running             0          3m
```

Таким чином на гіпервізорі встановлені 4 віртуальних машини із CoreOs у якості операційної системи. На них у свою чергу встановлений кластер, що складаєтсья з одного матсеру та трьох нод.

Функціонально кластер виглядає наступним чином:

- Flannel service - обслуговує однорангову мережу. Дозволяє контейнерам обмінюватись трафіком.
- Etcd service – Відповідає за збереження стану kubernetes
- Docker service – власне кажучи сервіс, за допомогою якого kubernetes запускає контейнери

- Kubelet service – обєднує ноди у кластер, дозволяє запуск інших додатків куберенетес

2.4.2 Варіант2

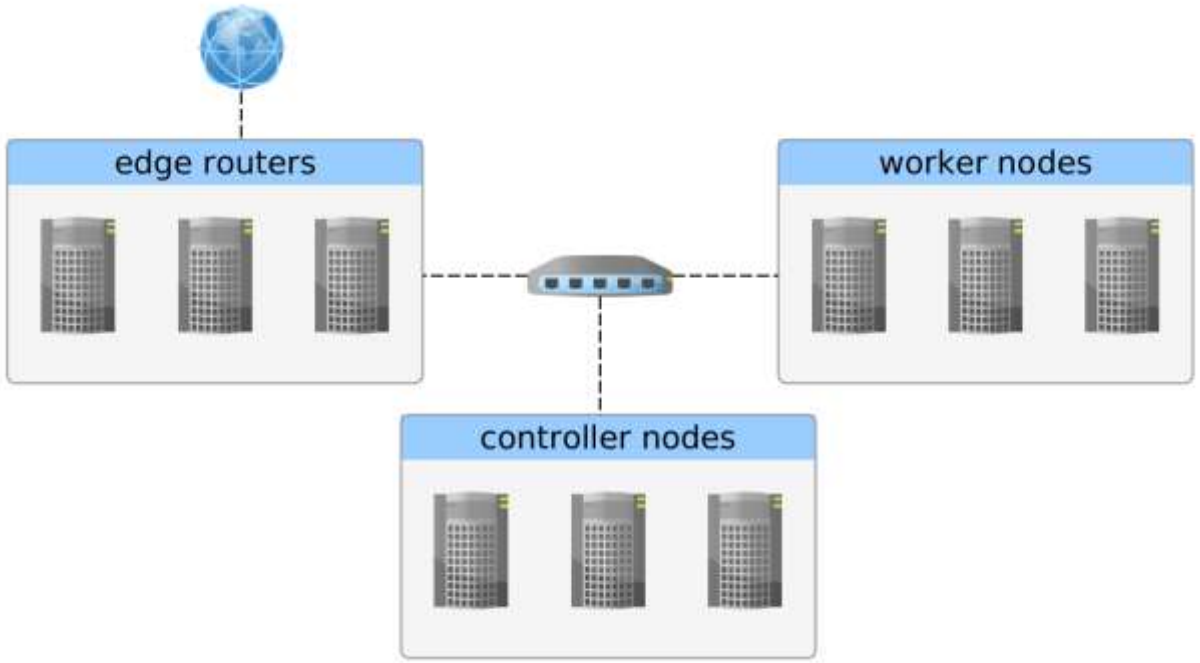

Рис 2.4 – Структура кластеру, за варіантом 2

Описання даної побудови:

Виділимо основні наступні віхи

1) Три види нод, що у даній побудові виконують наступне:[3,4]

- Контролери (мастери) – проводять оркестрацію контейнерів, якщо майстер не один, то проводяться вибори і обирається один активний
- Агентні – ноди, на яких працюють поди
- Роутери - ноди із публічними адресами та публічними доменами

2) Віртуальна машина на Xen
3) pfSense – виконуюча роль роутера віртуальна машина

Інсталяція

Так само, як і у першому пункті згенеруємо

1) образи iso
2) генерація сертифікатів

a. Контролера
b. Робочих нод
c. CLI kubectl

3) Генерація конфігурацій для сервісу cloud config, що встановить потрібні нам ноди

```
git clone https://github.com/xetys/kubernetes-coreos-baremetal
$ cd kubernetes-coreos-baremetal
$ ls
-rwxrwxr-x 1 magos magos 3,6K Jan 26 00:25 build-cloud-config.sh
-rwxrwxr-x 1 magos magos   84 Jan 26 00:25 build-image.sh
-rw-rw-r-- 1 magos magos 1,6K Jan 26 00:25 certonly-tpl.yaml
-rwxrwxr-x 1 magos magos  489 Jan 26 00:25 configure-kubectl.sh
-rw-rw-r-- 1 magos magos  408 Jan 26 00:25 master-openssl.cnf
-rw-rw-r-- 1 magos magos  272 Jan 26 00:25 worker-openssl.cnf
```

Таким чином у обох конфігураціях (цій і попередній) основною дією є генерація образів ISO та монтування їх у операційну систему CloudOS перед завантаженням, що нададуться гостьовим хостам гіпервізором XEN.

Тепер необхідно підготувати самі машини кластера:

1) Контролер 10.10.10.1
2) Нода 1 10.10.10.2
3) Нода 2 10.10.10.3
4) EDGE роутер 123.234.234.123 (<= «чесна» адреса, тут вона для прикладу)

Треба зауважити, що для роботи кластеру необхідно замаршрутувати трафік таким чином, щоб Ноди мали доступ до EDGE, для того, щоб була можливість виставити вміст подів для доступу з назовні.

Після створення і завантаження машин - згенеруємо cloud config для кожного члена кластеру, лістинг коду bash скриптів наданий у лістингу.

```
$ ./build-cloud-config.sh controller 10.10.10.1
...
$ ./build-cloud-config.sh worker1 10.10.10.2
10.10.10.1
...
$ ./build-cloud-config.sh worker2 10.10.10.3
10.10.10.1
...
$ ./build-cloud-config.sh example.com 123.234.234.123
10.10.10.1
...
```

Після цих дій маємо наступне:

1) Теку із ssl, що містить:

a. TLS пару, що необхідна для створення та верифікації інших TLS сертифікатів для цього кластеру

b. Пару ключів для адмністратора, що використовуватиметься kubectl

2) Інвентарій для створення кожної ноди

```
tree inventory
inventory
├── node-controller
│   ├── cloud-config
│   │   └── openstack
│   │       └── latest
│   │           └── user_data
│   ├── config.iso
│   ├── install.sh
│   └── ssl
│       ├── apiserver.csr
│       ├── apiserver-key.pem
│       └── apiserver.pem
├── node-example.com
│   ├── cloud-config
│   │   └── openstack
│   │       └── latest
│   │           └── user_data
│   ├── config.iso
│   ├── install.sh
│   └── ssl
│       ├── worker.csr
│       ├── worker-key.pem
│       └── worker.pem
├── node-worker1
│   ├── cloud-config
│   │   └── openstack
│   │       └── latest
│   │           └── user_data
│   ├── config.iso
│   ├── install.sh
│   └── ssl
│       ├── worker.csr
│       ├── worker-key.pem
│       └── worker.pem
└── node-worker2
    ├── cloud-config
    │   └── openstack
    │       └── latest
    │           └── user_data
    ├── config.iso
    ├── install.sh
    └── ssl
        ├── worker.csr
        ├── worker-key.pem
        └── worker.pem
```

Тепер розглянемо cloud config. Він містить конфігурацію etc2/system.d/ для flannel та calico, та офіційні скрипти інсталяцій для CoreOS для MaaS.

Далі монтуємо образ та запускаємо машини, та запускаємо kubectl

```
$ ./configure-kubectl.sh 10.10.10.1
```

Перевіримо установку

```
$ kubectl get nodes
NAME            STATUS                        AGE
10.10.10.1      Ready,SchedulingDisabled      3m
10.10.10.2      Ready                         3m
10.10.10.3      Ready                         3m
123.234.234.123 Ready                         3m
```

```
curl -s localhost:10255/pods | jq -r
'.items[].metadata.name' # controller only, show
running pods, should be apiserver, controller-
manager, scheduler and proxy
```

Перевірка показала, що все готово. Тепер встановимо балансувальники nginx-ingress. Враховуючи, що тепер є встановлений кластер, проте з мінімальною кількістю інструментів. Тепер необхідно забезпечити доступ із мережі до кластеру, для чого використовується механізм Ingress, що надає послуги маршрутизації трафіку kubernetes. Для кафедри найкращим способом підійде легковісний nginx ingress контролер, що підтримує TCP, UDP, websocket, шифрування, тощо.

Ingress – сервіс, що у цьому кластері виконує роль балансувальника сьомого рівня OSI, а також маршрутизатора і брадмауера; Ingress - набір правил, що дозволяють пройти вхідному трафіку до кластеру Може бути налаштований таким чином, щоб надавати сервісам зовнішній URL, балансування трафіку, термінацію SSL, віртуальний хостинг, тощо. Для того, щоб цей сервіс працював

необхіно мати працюючу ноду у кластері, таким чином мастер зможе контролювати контролер Ingress та його динамічнічний функціонал шляхом моніторингу ендпойнту (від англ. Endpoint – точнка виходу api сервісу).

Надалі для забезпечення безпеки введемо розідлення на інфрастурктурному рівні Edge нод від мастера і робочих нод, таким чином кластер kubernets із High Availablility буде виглядати як на рисунку, що запропонований нижче.

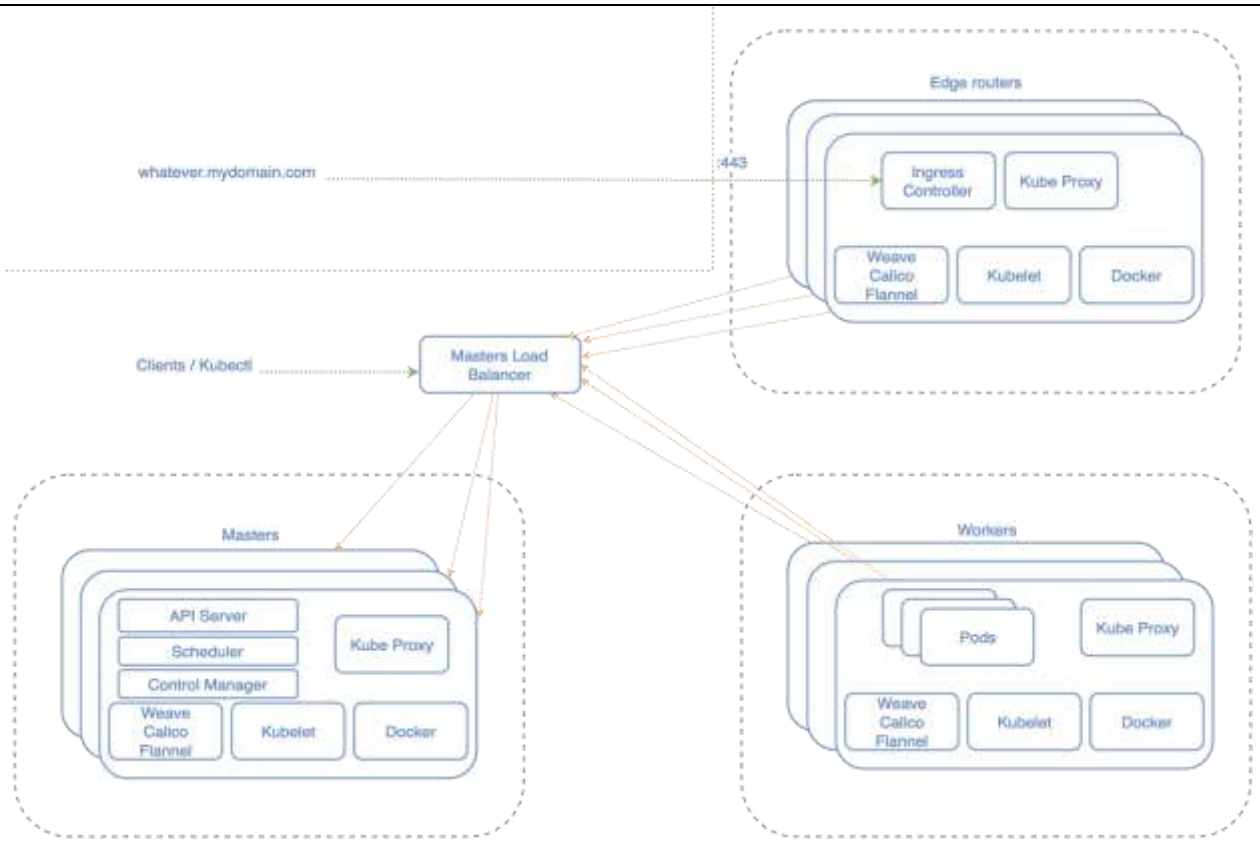

Рис 2.5 - Архітектура Кластера Кубернетес у режимі вискодоступності

Для реалізації цього більш скаладного варіанту – використаємо Traefik[7]. Traefik є сучасною реалізацією зворотнього проксі і балансувальника, що був створений для підтримки і обслуговування мікросервісів. Автоматично він підтримує усі три претенденти, через що був і обраний на випадок того, якщо буде розгорнуто

додатковий гіпервізор, що не буде kubernetes. Загальний вигляд алгоритму роботи Traefik зпропонований на рисунку нижче.

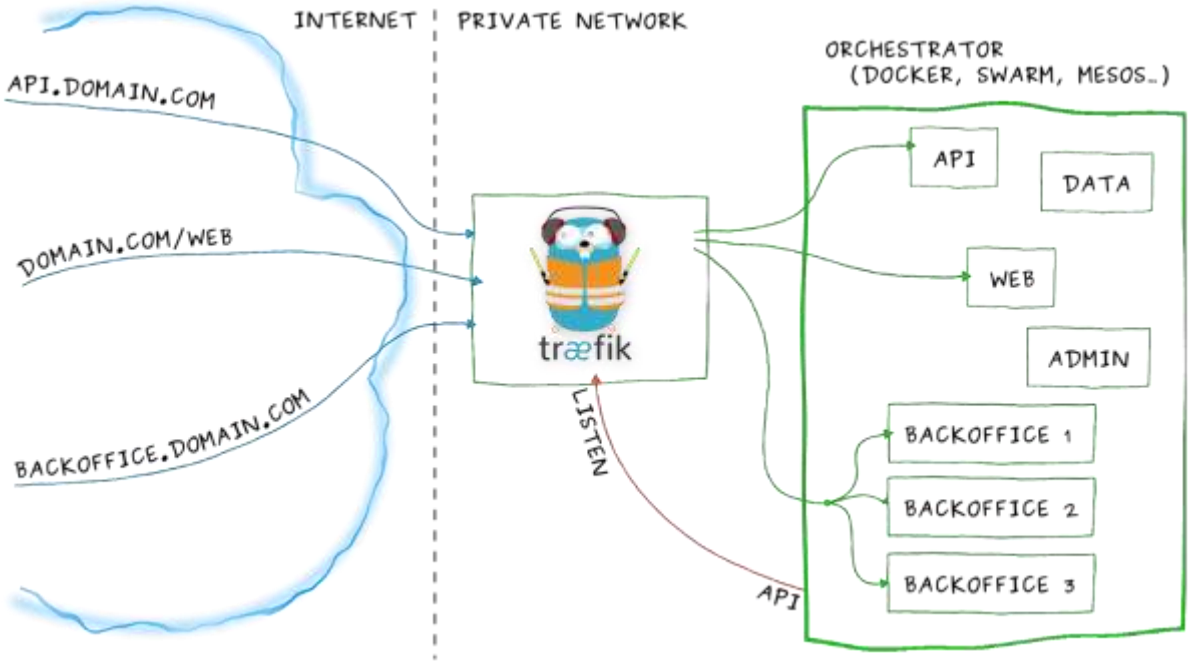

Рис. 2.6 – Схема доступ трафіку за допомогою Traefik

Слід зауважити, що для найоптимальнішої розгортки кластер буде запущений із правилами DaemonSet, що відповідають за те, що деякі, чи всі ноди мають як мінімум один набір подів, що були визначені у конфігурації, в працюючому стані.[6] Перевага підходу у тому, що поди, які додаються у кластер автоматично розподіляються у ноди, які були додані, коли ж прибираються, то garbage collector (з англ. Прибиральник сміття – механізм, що вивільняє ресурси із памяті) автоматично видаляються. Видалення DaemonSet'у автоматично прибере поди, щ обули створені першим.

Обравши механізм популяції – відключимо Edge ноди від загальної розгортки наступними командами:[5]

1) `-register-schedulable=false`
2) `--node-labels=edge-router`

Команда 1 не дозволить подам популювати та бути запалнованими на роутерах, а команда 2 надасть цим нодам зручний лейбл, для зручного і прозорого написання планувальника, що буде необхідно для DaemonSet. Kuberenetes буду запускати ДемонСети на кожній ноді, навіть, якщо вона позначена, як не-для-планування (non-schedulable), враховуючи, що DaemonSet має запускатись тільки на edge-router'ах, встановимо «nodeSelector» у положення перевірки ролі «edge-router», лейбл, щоб був створений вище.

Таким чином команду, що буде виконано для створення лейби edge роутера буде виглядати наступним чином:

```
$ kubectl label node "role=edge-router" -l
"cube.cad.ntu-kpi.kiev.ua/hostname=123.234.234.123"
```

Мережевий стек готовий, тепер визначимо обробку помилок:

Коли користувач звертається до не існуючого сервісу, будемо віддавати помилку 404 "Not Found", як і належить, для цього визначемо наступний конфігураційний файл:

Default-backend.yaml

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: default-http-backend
  namespace: kube-system
spec:
  replicas: 1
  selector:
    app: default-http-backend
  template:
    metadata:
      labels:
        app: default-http-backend
    spec:
      terminationGracePeriodSeconds: 60
      containers:
      - name: default-http-backend
        # Any image is permissable as long as:
```

```
            # 1. It serves a 404 page at /
            # 2. It serves 200 on a /healthz endpoint
            image: gcr.io/google_containers/defaultbackend:1.0
            livenessProbe:
              httpGet:
                path: /healthz
                port: 8080
                scheme: HTTP
              initialDelaySeconds: 30
              timeoutSeconds: 5
            ports:
            - containerPort: 8080
            re s:
              limits:
                cpu: 10m
                memory: 20Mi
              requests:
                cpu: 10m
                memory: 20Mi

    ---

    apiVersion: v1
    kind: Service
    metadata:
      labels:
        app: default-http-backend
      name: default-http-backend
      namespace: kube-system
    spec:
      ports:
      - port: 80
        protocol: TCP
        targetPort: 8080
      selector:
        app: default-http-backend
      sessionAffinity: None
      type: ClusterIP
```

Після чого одразу примінимо зміни

```
kubectl create -f default-backend.yaml -n kube-system
```

Та визначимо вигляд ingress контролера.

Ingress-controller.yaml

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: nginx-ingress-controller-v1
  namespace: kube-system
  labels:
    k8s-app: nginx-ingress-lb
    kubernetes.io/cluster-service: "true"
spec:
  template:
    metadata:
      labels:
        k8s-app: nginx-ingress-lb
        name: nginx-ingress-lb
    spec:
      hostNetwork: true
   terminationGracePeriodSeconds: 60
      nodeSelector:
        role: edge-router
      containers:
      - image: gcr.io/google_containers/nginx-ingress-controller:0.8.3
        name: nginx-ingress-lb
        imagePullPolicy: Always
        readinessProbe:
          httpGet:
            path: /healthz
            port: 10254
            scheme: HTTP
        livenessProbe:
          httpGet:
            path: /healthz
            port: 10254
            scheme: HTTP
          initialDelaySeconds: 10
          timeoutSeconds: 1
        # use downward API
        env:
          - name: POD_NAME
            valueFrom:
              fieldRef:
                fieldPath: metadata.name
          - name: POD_NAMESPACE
            valueFrom:
              fieldRef:
                fieldPath: metadata.namespace
        ports:
        - containerPort: 80
          hostPort: 80
        - containerPort: 443
          hostPort: 443
        args:
        - /nginx-ingress-controller
        - --default-backend-service=$(POD_NAMESPACE)/default-http-backend
```

Примінимо конфігурацію

```
kubectl create -f ingress-controller.yaml
```

Тепер, якщо, до приклду користувач зверенться до example.com та субдоменів, то отримає 404.

Тепер перевіримо створений ingress контролер – щапустимо простий под із додатком, що друкує запит, який приходить

```
kubectl run echoheaders --
image=gcr.io/google_containers/echoserver:1.4 --
replicas=1 --port=8080
```

тепепер зробимо його достпуним для доступу через порт 80

```
kubectl expose deployment echoheaders --port=80 --
target-port=8080 --name=echoheaders
```

додамо правило для ingress

Echoheader-ingress.yaml

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: echomap
spec:
  rules:
  - host: echo.example.com
    http:
      paths:
      - path: /
        backend:
          serviceName: echoheaders
          servicePort: 80
```

Та примінимо це правило

```
$ kubectl create -f echoheader-ingress.yaml
```

```
[root@k8s-master ingress-test]# kubectl describe ing
Name:                   echomap
Namespace:              default
Address:                10.39.1.45
Default backend:        default-http-backend:80 (10.244.3.28:8080)
Rules:
  Host          Path    Backends
  ----          ----    --------
  foo.bar.com
                /foo    echoheaders-x:80 (<none>)
  bar.baz.com
                /bar    echoheaders-y:80 (<none>)
                /foo    echoheaders-x:80 (<none>)
Annotations:
Events: <none>
[root@k8s-master ingress-test]#
```

Рис. 2.6 Перевірка роботи кластеру

Результат перевірки – задовільний. Кластер налаштований.

2.5 Висновки за розділом:

Враховуючи простоту та досконаліть другого варіанту, оберемо його як доцільний та найпростіший для розгортання через наступні фактори:

1) Простота розгортки
2) Низький рівень абстракції
3) Використання DaemonSet
4) Можливість емулювати скалдну архіткетуру для навчання студентів
5) Просте обслуговування

Із фінансової точки зору – кластер варіанту 2 виходить дешевшим, у районі 100 Евро на місяць, якщо розгортати його у клауд провайдері, також він дозволяє гнучко поділяти ядра (12 на сервері, що перетворюються у 40 логічних у кластері).

Після налаштування сервісу – наступним кроком буде створення та обрання мікросервісів (докерів), для створення інфраструктури у мережі кафедри.

3 Логічна мережа. Наповнення Кластеру Функціоналом

Визначившись із налаштуваннями кластеру – наповнимо його логічним змістом, встановивши кістяк сервісів, що будуть обслуговувати кафедру.

3.1 Планування:

Для функціонування установи буде мінімально необхідне наступне:

1) Сервіс Автентифікації
2) Сервіс Документообігу
3) Сервіс, що надаватиме студентам обмежені можливості із старту обмеженої кількості контейнерів.
4) Система моніторингу

Також доцільним буде перенести веб ресурс сайту на мікросервісну, оскільки сайт кафедри абсолютно stateless, тому розглянемо це, як п’ятий пункт.
Також додатково у рамках курсів програмування можна запропонувати інсталяцію сервісу код аналізу, на кшталт Moodle.

Розглянувши мінімальні потреби, опишемо технічне завдання для побудови інфраструктури:

1) Сервіс автентифікації – SAMBA, чи ж то OpenLDAP, що буде реплікувати базу даних, що вже міститься у Active Directory Windows Domain. Через нього буде забезпечений логін в систему kubernetes, а також у докери та суміжні сервіси
2) Система моніторингу. Будуть присутні одразу дві

a. Вже існуюча на кафедрі Nagios буде наглядати за роботою смаого кластеру
b. Prometheus – буде відповідати за моніторинг піднятих контейнерів мікросервісів

3) Сервіс документообігу. Найбільш доцільним тут буде зауважити, що це буде більше як сервіс обміну самими файлами, як то для студентів, так і для персоналу кафедри, проте із розширеними можливостями редагування файлів. Критерії обрання будуть описані нижче.
4) Додаткові сервіси, що можуть бути запропоновані. Цей пункт плану продемонструє можливості встановлення різноманітних сервісів для навчання та функціонування кафедри, у цій праці до розгляду пропонується наступне:

a. LMS Moodle

3.2 Виконання

3.2.1 Автентифікація

У сам kubernetes та сервіси, що запущені на ньому буде необхідно забезпечити доступ. Цей пункт реалізації доволі простий, оскільки все що буде потрібно – це підключення до вже існуючого домену, що є абсолютно тривіальним:
Для виконання цього пункту – виконаємо наступне

1) Встановимо на віртуальній машині Xen keystone service, що надасть доступ до дерева домену куберенетес із наступними налаштуваннями

Згенеруємо секретний токен SSL

openssl rand -hex 10, що у випадку цієї праці дорівнює
«eb18653b1907c4c0e97f»

/etc/keystone/keystone.conf

```
[DEFAULT]
admin_token = eb18653b1907c4c0e97f
public_bind_host = kauth.cad.ntu-kpi.kiev.ua
admin_bind_host = kauth.cad.ntu-kpi.kiev.ua

[identity]
driver = ldap

[ldap]
chase_referrals = false
page_size = 1000
query_scope = sub
suffix = CN=Administrator, DC=cs-lab,DC=cad,DC=ntu-
kpi,DC=kiev,DC=ua
url = ldap://dragon2:389
use_tls = false
#If use_tls = true then the following parameters
will need to be uncommented
#tls_req_cert = demand
#tls_cacertfile =<path-to-cert-file>
user = Administrator@cs-lab.cad.ntu-kpi.kiev.ua
password = “користувацький_пароль”
user_allow_create = false
user_allow_update = false
user_attribute_ignore = enabled
user_id_attribute = CN
user_mail_attribute = mail
user_name_attribute = sAMAccountName
user_objectclass = organizationalPerson
user_tree_dn = OU=Users,DC=cs-lab,DC=cad,DC=ntu-
kpi,DC=kiev,DC=ua
[eventlet_server_ssl]
enable = True
certfile = /etc/keystone/ssl/certs/keystone.pem
keyfile = /etc/keystone/ssl/private/keystonekey.pem
ca_certs = /etc/keystone/ssl/certs/ca.pem
```

```
cert_required = False

[ssl]
ca_key = /etc/keystone/ssl/private/cakey.pem
key_size = 2048
valid_days = 3650
cert_subject=/C=IN/ST=Administrator/O=cad.ntu-
kpi.kiev.ua/CN=kauth.cad.ntu-kpi.kiev.ua
```

Тепер встановимо SSL

keystone-manage ssl_setup --keystone-user keystone --keystone-group keystone –rebuild

Та перезапустимо keystone:

service keystone restart

2) Налаштування Кубернетес

На операційній системі

copy the /etc/keystone/ssl/certs/ca.pem file /etc/pki/ca-trust/ /anchors/kauth.cad.ntu-kpi.kiev.ua.crt

&& run update-ca-trust

3) Перевірка

На мастері кубуернетес перевіримо доступність користувацьких записів:

```
keystone --os-cacert /etc/keystone/ssl/certs/ca.pem
--os-auth-url "https://kauth.cad.ntu-
kpi.kiev.ua:5000/" --os-token
"eb18653b1907c4c0e97f" --os-endpoint "https://
kauth.cad.ntu-kpi.kiev.ua:35357/v2.0/" user-list |
grep Administrator
```

Рис 3.1 – Результат перевірки імпорту користувачів

4) Налаштування Кубернетес

Налаштування кубернетес для авторизації

```
v1
kind: Config
preferences: {}
clusters:
- cluster:
    certificate-authority-data:
LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUR1VENDQXFHZ0F3SUJBZ0lKQUw4NFMx
TGUwUTNjTUEwR0NTcUdTSWIzRFFFQkN3VUFNRUl4UURBK0JnTlYKQkFNVU56RTVNaTR4Tmpn
dU1USXlMakUzTXl4SlVEb3hOekl1TVRZdU1DNHlMRWxRT2pFM01pNHhOaTQxT0M0dwpRREUw
TmprNE9ERTFOelV3SGhjTk1UWXdOek13TVRJeU5qRTFXaGNOTWpZd056STRNVEl5TmpFMVdq
QkNNVUF3ClBnWURWUVFERkRjeE9USXVNVFk0TGpFU1pNHhOek1zU1ZBNk1UY3lMakUyTGpB
dU1peEpVRG94TnpJdU1UWXUKTlRndU1FQXhORF
    server: https://kauth.cad.ntu-kpi.kiev.ua:443
  name: cube.cluster
contexts:
- context:
    cluster: cube.cluster
    namespace: administrator
    user: administrator
  name: administrator.cube.cluster
users:
- name: administrator
  user:
    username: administrator
    password: password
current-context: administrator.cube.cluster
```

Таким чином налаштований кластер для автентифікації.

3.2.2 Моніторинг

Враховуючи, що на кафедрі вже існує система моніторингу Nagios – не будем обрати до уваги її налаштування. Проте розглянемо систему Prometheus, та аргументацію до вибору саме її.

Сучасні кластерні мікросервісні архітектури, що використовує Kubernetes, де слабкозв'язані контейнери створюються та знищуються, в залежності від того, коли і де вони необхідні, вимагає нового підходу до ефективного моніторингу. Вже недостатньо, моніорити навантаження і активність на конкретній машині, коли робота, що призначається цій машини може змінюватися згідно вимог кластеру і додатку в цілому. Якщо ж архітектура розроблена таким чином, що процес може зникнути, не можна сказати, чи є відмова критичною, спостерігаючи за одним тільки процесом.

Так само, як Kubernetes змушує архітекторів більше думати про додатки та навантаження, аніж про процеси, так само і процес моніторингу. Кластеро-центричний моніторинг надає перевагу у тому, що користувач може реагувати на тренди поведінки на рівні додатків, реагувати швидше та отримувати менше інформаційного шуму.

Prometheus – відкритий продукт, що був спеціально створений задля моніторингу кластерних систем, таких як Kubernetes, зокрема модель назначень та керування нодами. На разі Promethtus підтримує Api Kubernetes, що дуже зручно, оскільки запит напряму до кластера від системи моніторингу буде швидшим, аніж використання третіх інструментів, також слід зауважити, що Kubernetes має набір заздалегідь підготовлених метрик, що ними може скористатись Prometheus.

3.2.2.1 Процес встановлення

1) Скористаймось офіційним образом від CoreOS

   kubectl create -f https://raw.githubusercontent.com/coreos/blog-examples/master/monitoring-kubernetes-with-prometheus/prometheus.yml

2) За необхіності - сконфігуруємо сховище, куди прометеус зберігає дані, для цього необхіно замінити emptyDir на абсолютний шлях у конфігурації.

3.2.2.2 Перевірка

1) У стоковій конфігурації скористаймося прокидуванням портів

```
$ kubectl get pods -l app=prometheus -o name | \ sed
's/^.*\///' | \ xargs -I{} kubectl port-forward {}
9090:9090
```

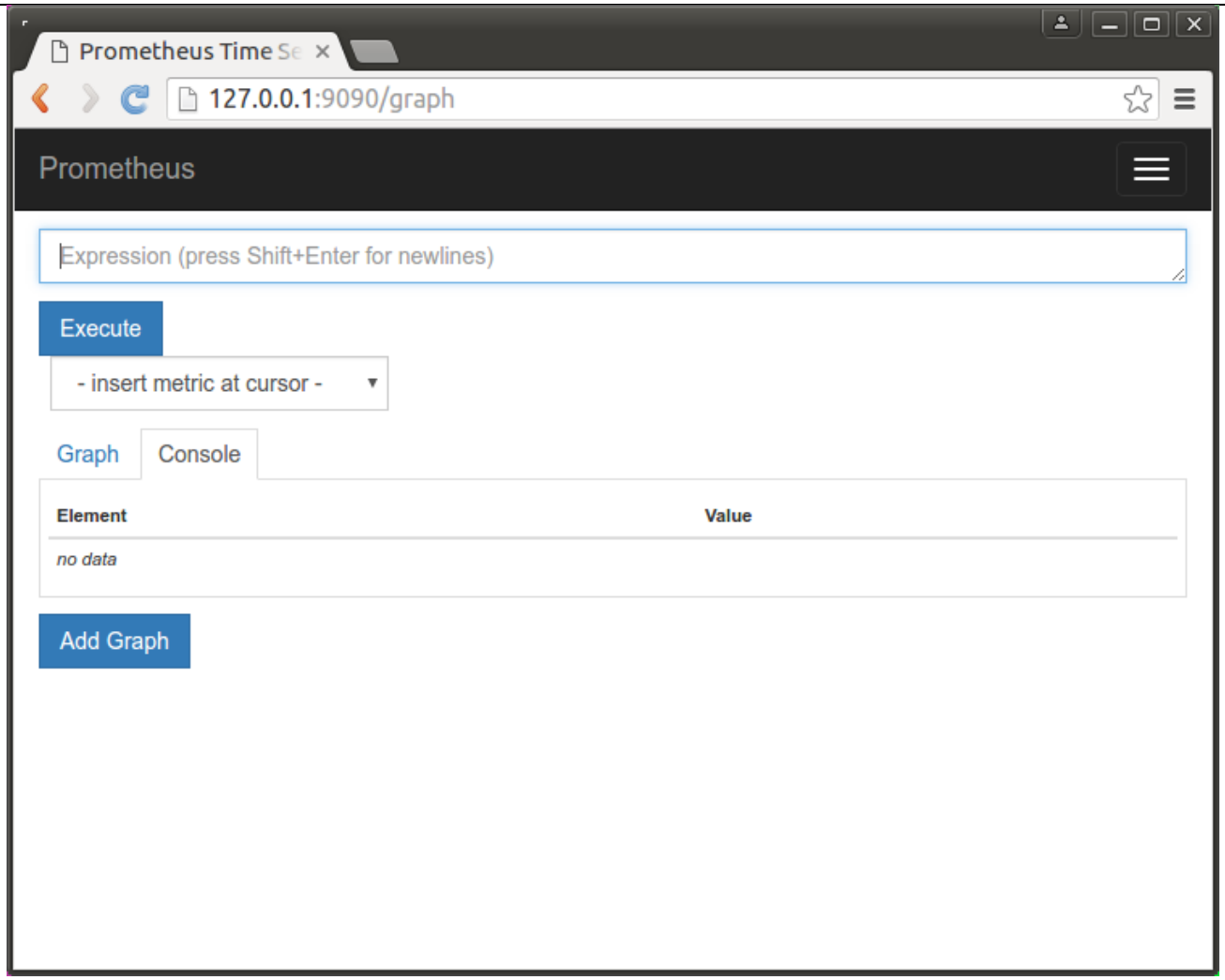

Рис. 3.2 – Користувацький інтерфейс Prometheus

Система піднята і готова до налаштування
Тепер створимо декілька запитів для ілюстрації ідеї. Можна відобразити зайнятісь CPU, RAM, вводу-виводу, та інше, за допомогою запиту

```
container_memory_usage_bytes{image="CONTAINER:VERSION"}
```

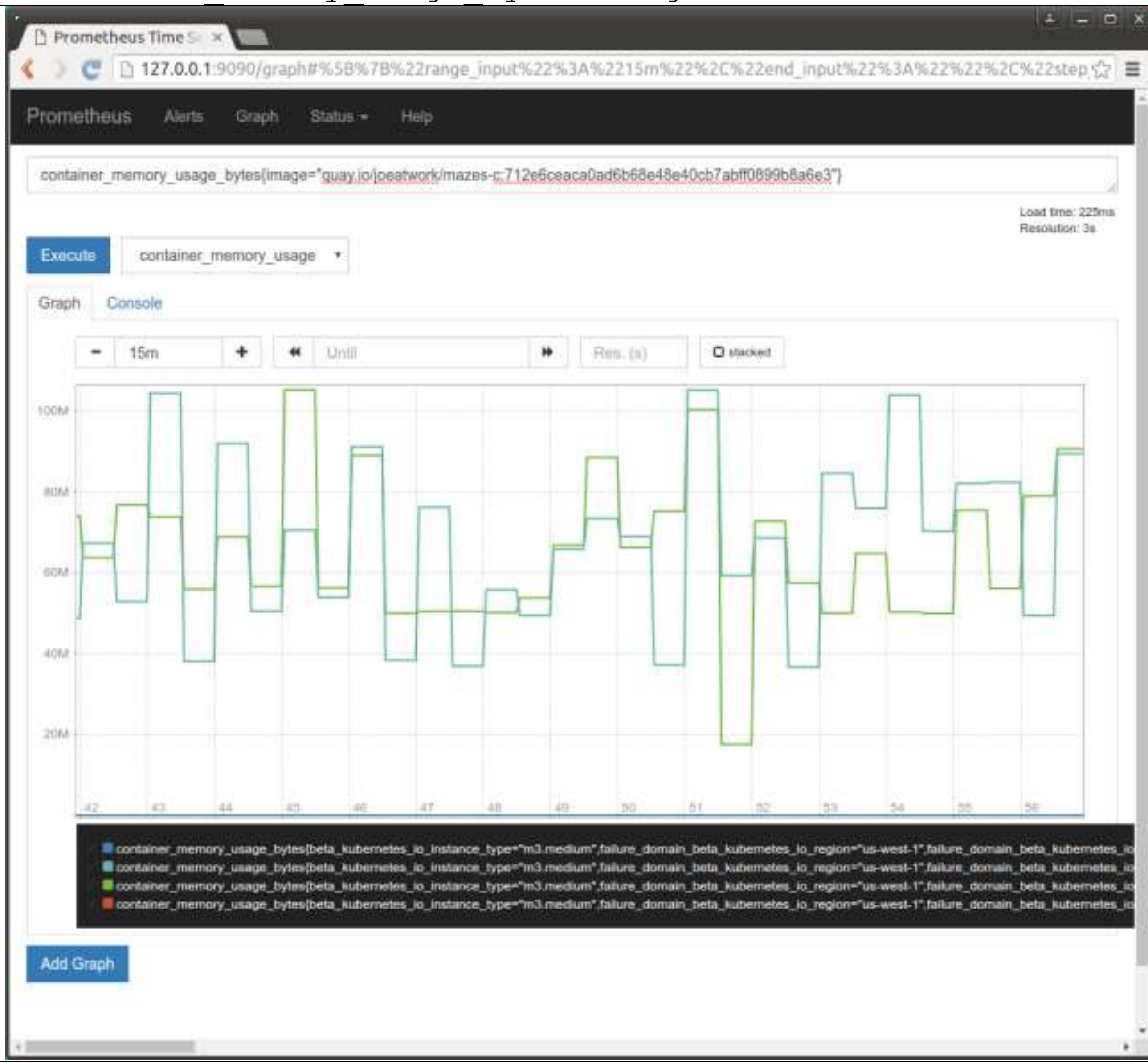

Рис 3.3 – Графіки роботи контейнера

Також можна моніторити сам kubernetes, наприклад:

```
sum(container_memory_usage_bytes{kubernetes_namespace="kube-system"})
sum(container_memory_usage_bytes{kubernetes_namespace="kube-system",
kubernetes_pod_name=~"kube-dns-v11.*"})
```

Дані запити відобразять кількість ресурсів, що використовує неймспейс kube-system у першому випадку, та скільки використовує окремий под у другому

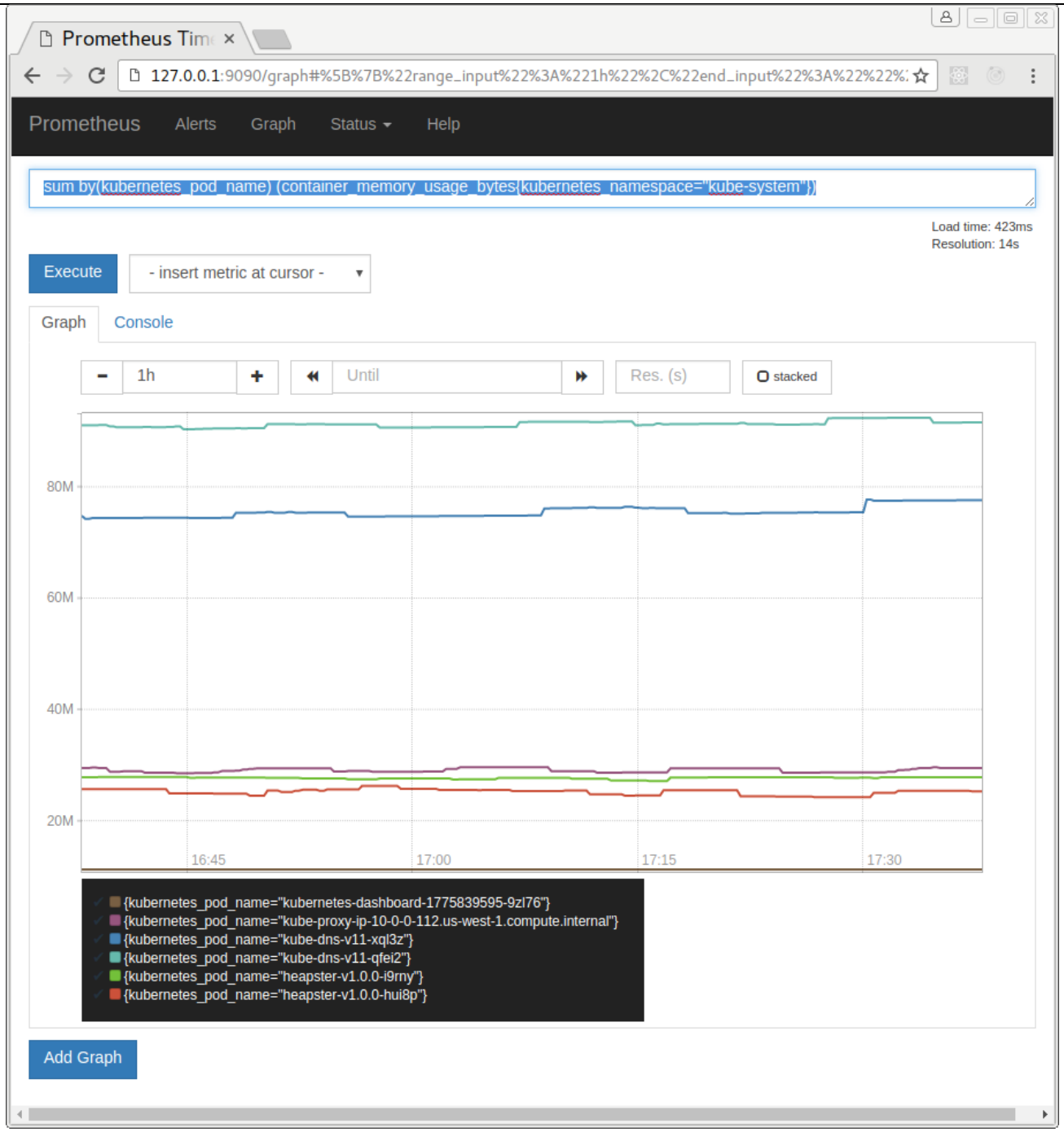

Рис 3.4 – Графік Роботи неймспейсу kube-system

Таким чином у нас є налаштована система приклад системи моніторингу, що можна використовувати. Вона легковісна, може сповіщувати користувача через канали пошти, чи інші зручні мессенжери. Дозволяє гнучке налаштування.

3.2.3 Сервіс Документообігу

У якості сервісу документообігу буде запропонований ownCloud, через швидкість налаштування та легкість використання. Також сервіс можна встановити таким чином, щоб він був вебформою до вже інсуючого сховища. Скористаємось для демонстрації скористаємось наступними параметрами:

1) Mysql, є реляційною базою даних, що буде містити користувацькі логіни та паролі, вразі відмови доступу до Active Directory
2) Owncloud – власне система розпрділеного доступу.

Налаштування MYSQL

Mysql.yaml

```
kind: Service
apiVersion: v1
metadata:
  name: mysql
spec:
  ports:
    - name: mysql
      port: 3306
      protocol: TCP
  selector:
    app: mysql

---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: mysql
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - name: mysql
        image: mysql:5.7.12
        ports:
        - containerPort: 3306
        env:
          - name: MYSQL_ROOT_PASSWORD
            value: rootpwd
          - name: MYSQL_DATABASE
            value: owncloud
          - name: MYSQL_USER
            value: owncloud
          - name: MYSQL_PASSWORD
            value: owncloud
```

Налаштування ownCloud

Owncloud.yaml

```
kind:
Service
              apiVersion: v1
              metadata:
                name: owncloud
              spec:
                ports:
                  - name: owncloud
                    port: 80
```

```
      protocol: TCP
  selector:
    app: owncloud
  type: LoadBalancer


---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: owncloud
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: owncloud
    spec:
      containers:
      - name: owncloud
        image: gurvin/owncloud-dp
        ports:
        - containerPort: 80
        re s:
          requests:
            cpu: '1024m'
            memory: '1G'
        volumeMounts:
          - name: owncloud
            mountPath: /var/www/html/data
      volumes:
        - name: owncloud
          persistentVolumeClaim:
            claimName: owncloud-dp
```

Створивши конфігураційні файли запускаємо контейнери:

kubectl create –f mysql.yaml owncloud.yaml

таким чином маємо запущений owncloud, перевіримо його:

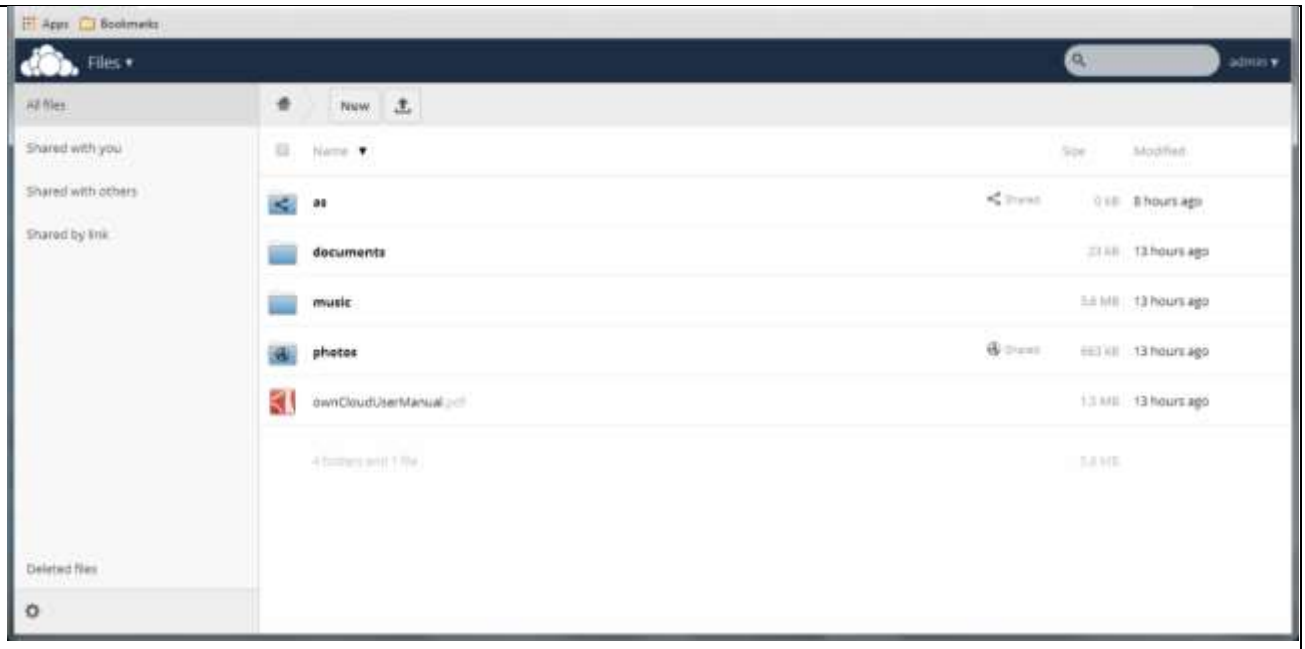

Рис 3.5 – Користувацький інтерфейс OwnCloud

Сервіс працює успішно, тепер перейдемо до розширення функціоналу, а саме встановлення інтеграції з онлайн офісом.

Налаштування Collaborative tools.

1) Перейдемо до налаштувань Apps
2) Активуймо потрібний нам модуль підтримки MS Word

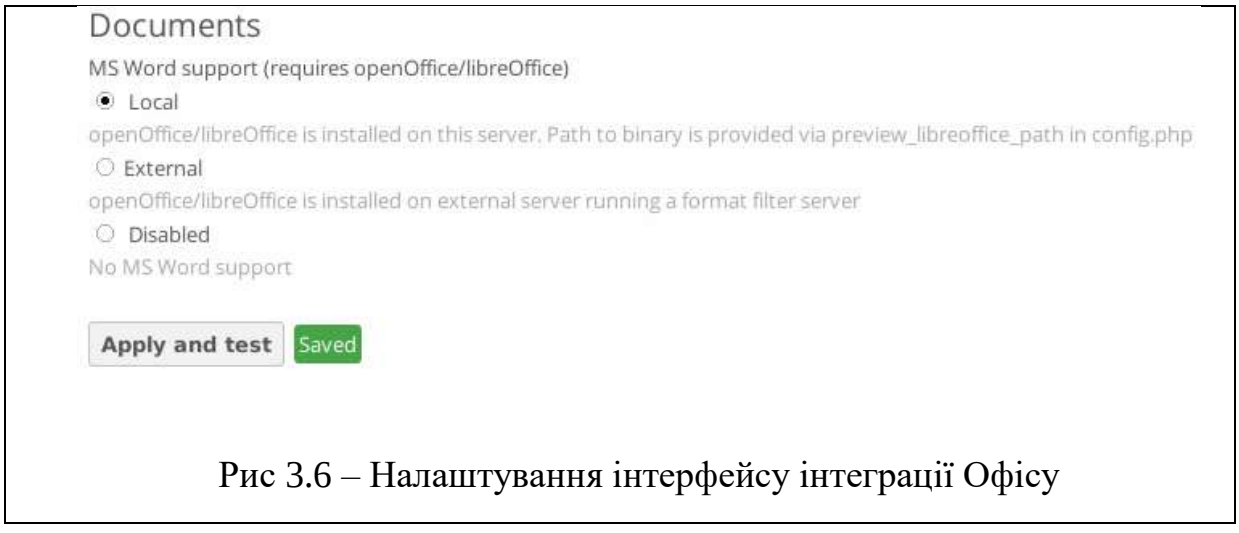

Рис 3.6 – Налаштування інтерфейсу інтеграції Офісу

3) Перевіримо, відкривши базовий документ для прикладу:

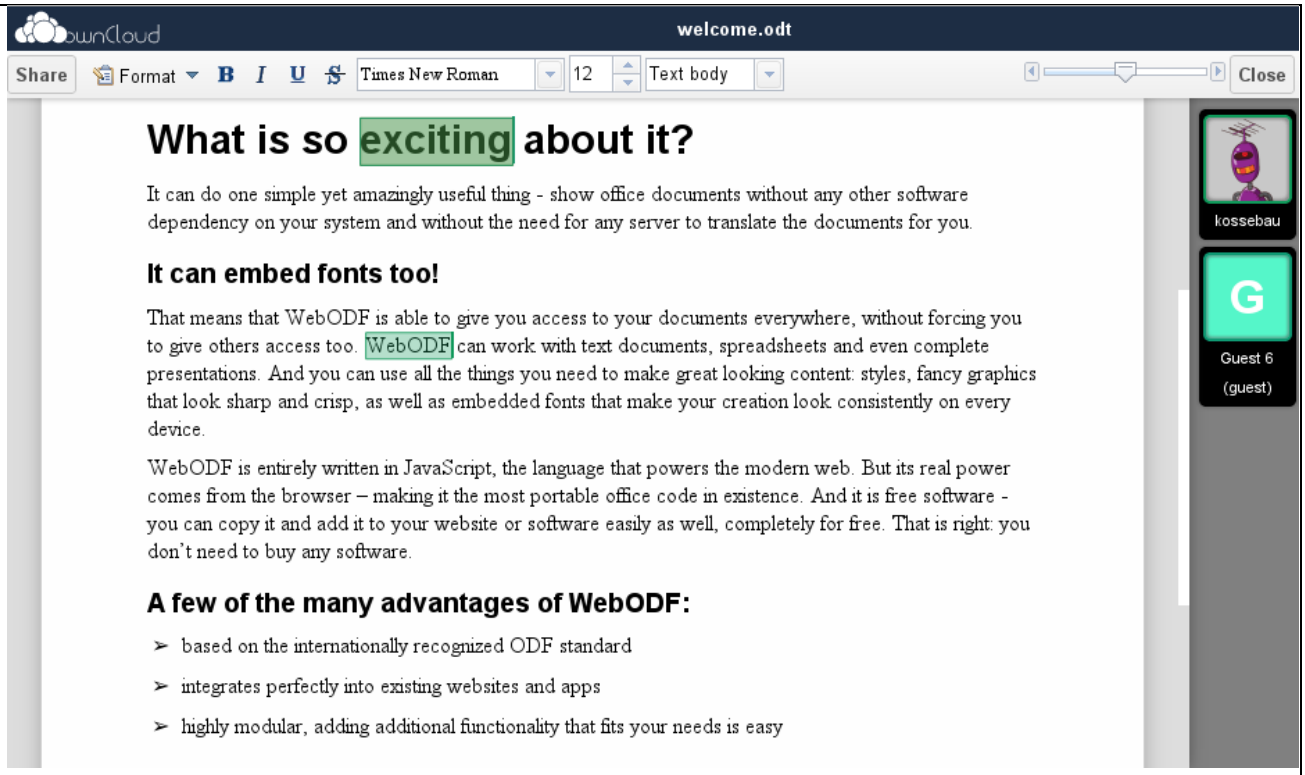

Рис 3.7 – Користувацький інтерфейс опен офісу

Таким чином, маємо інструмент, що дозволяє базове редагування файлів, із виділеним місцем на жорсткому диску, що виділяється груповими політиками owncloud (за замовчуванням 1 ГБ), де студенти можуть передавати свої лабораторні роботи, а викладачі вести журнали успішності.

3.2.4 Система Дистанційного Навчання

Маючи основні необхідні системи, продемонструємо корисність кластеру на прикладі швидкого розгортання такої системи, як Learning Management System Moodle.

Налаштування конфігурації для кубернетес:

Moodle.yaml

```
apiVersion: v1
items:
- apiVersion: v1
  kind: Service
  metadata:
    creationTimestamp: null
    labels:
      io.kompose.service: mariadb
    name: mariadb
  spec:
    clusterIP: None
    ports:
    - name: headless
      port: 55555
      targetPort: 0
    selector:
      io.kompose.service: mariadb
  status:
    loadBalancer: {}
- apiVersion: v1
  kind: Service
  metadata:
    annotations:
      kompose.service.type: nodeport
    creationTimestamp: null
    labels:
      io.kompose.service: moodle
    name: moodle
  spec:
    ports:
    - name: "80"
      port: 80
      targetPort: 80
    - name: "443"
      port: 443
      targetPort: 443
    selector:
      io.kompose.service: moodle
    type: NodePort
  status:
    loadBalancer: {}
- apiVersion: extensions/v1beta1
  kind: Deployment
  metadata:
    creationTimestamp: null
    name: mariadb
  spec:
    replicas: 1
    strategy:
      type: Recreate
    template:
```

```
        metadata:
          creationTimestamp: null
          labels:
            io.kompose.service: mariadb
        spec:
          containers:
          - env:
            - name: ALLOW_EMPTY_PASSWORD
              value: "yes"
            image: bitnami/mariadb:latest
            name: mariadb
            re s: {}
            volumeMounts:
            - mountPath: /bitnami/mariadb
              name: moodle-mariadb-data
          restartPolicy: Always
          volumes:
          - name: moodle-mariadb-data
            persistentVolumeClaim:
              claimName: moodle-mariadb-data
    status: {}
  - apiVersion: v1
    kind: PersistentVolumeClaim
    metadata:
      creationTimestamp: null
      labels:
        io.kompose.service: moodle-mariadb-data
      name: moodle-mariadb-data
    spec:
      accessModes:
      - ReadWriteOnce
      re s:
        requests:
          storage: 100Mi
    status: {}
  - apiVersion: extensions/v1beta1
    kind: Deployment
    metadata:
      annotations:
        kompose.service.type: nodeport
      creationTimestamp: null
      name: moodle
    spec:
      replicas: 1
      strategy:
        type: Recreate
      template:
        metadata:
          creationTimestamp: null
          labels:
            io.kompose.service: moodle
        spec:
          containers:
          - image: bitnami/moodle:latest
            name: moodle
            ports:
            - containerPort: 80
            - containerPort: 443
            re s: {}
```

```
            volumeMounts:
            - mountPath: /bitnami/moodle
              name: moodle-moodle-data
            - mountPath: /bitnami/apache
              name: moodle-apache-data
            - mountPath: /bitnami/php
              name: moodle-php-data
          restartPolicy: Always
          volumes:
          - name: moodle-moodle-data
            persistentVolumeClaim:
              claimName: moodle-moodle-data
          - name: moodle-apache-data
            persistentVolumeClaim:
              claimName: moodle-apache-data
          - name: moodle-php-data
            persistentVolumeClaim:
              claimName: moodle-php-data
    status: {}
  - apiVersion: v1
    kind: PersistentVolumeClaim
    metadata:
      creationTimestamp: null
      labels:
        io.kompose.service: moodle-moodle-data
      name: moodle-moodle-data
    spec:
      accessModes:
      - ReadWriteOnce
      re s:
        requests:
          storage: 100Mi
    status: {}
  - apiVersion: v1
    kind: PersistentVolumeClaim
    metadata:
      creationTimestamp: null
      labels:
        io.kompose.service: moodle-apache-data
      name: moodle-apache-data
    spec:
      accessModes:
      - ReadWriteOnce
      re s:
        requests:
          storage: 100Mi
    status: {}
  - apiVersion: v1
    kind: PersistentVolumeClaim
    metadata:
      creationTimestamp: null
      labels:
        io.kompose.service: moodle-php-data
      name: moodle-php-data
    spec:
      accessModes:
      - ReadWriteOnce
      re s:
        requests:
```

```
          storage: 100Mi
    status: {}
kind: List
metadata: {}
```

Конфігурація стартує сервіси:

a) Mariadb

b) Moodle

Перевіримо інсталяцію LMS:

Рис. 3.8 – Інсталяційне вікно LMS Moodle

На рисунку вище видно старт самої системи, докер контейнер на разі встановлений на ручну інсталяцію. Результат успішний.

3.3 Висновки за розділом.

Кластер був наповнений мінімальним логічним функціоналом для функціонування навчальної установи. Був встановлений сервіс автентифікації, за допомогою якого можна розподіляти доступ до кластера, а також через неї можливо роздавати доступ до функціоналу інших контейнерів. Була встановлена та швидко налаштована система документообігу, та обміну файлами, а також система дистанційного навчання. Всі системи швидко розгортаються, а також стійкі до похибок, оскільки налаштовування Kubernetes такі, що при відмові контейнера буде одразу створений новий. Таким чином було продемонстрована гнучкість кластеру мікросервісів, швидкість налаштування нових рішень, та гнучкість розробки логіки мережі.

Висновки

У цій праці було досліджене поняття сучасної архітектури мікросервісів, а ні основі дослідження було запропоновано побудувати кластер мікросервісів на кафедрі СП ННК «ІПСА». Було обрано оркестратор Kubernetes у якості основної системи кластеру, через якості, що дозволять її легке розготрання та контроль сервісів, що вона запускає. Побудова мікросервісного кластеру на кафедрі є цілком доцільним через зручність керування та оновлення інфраструктури, а також через створення динамічного середовища для створення процесів для навчання студентів та виконання лабораторного практикуму, а також пайплайну для опрацювання різноманітної інформації. Згідно вище описаного можна зробити висновок, що мікросервісна архітектура буде панувати на ринку найблищих декілька років, а при виході її з моди, чи розробці нового архітектурного стилю буде доволі легко мігрувати дану інфраструктуру на новий лад.

З екномічної точки зору дана архітектурна парадигма є дуже вигідною через утілізацію усєї потужності мережі клауд, чи заліза на 90-95% без втрати функціоналу чи потужності.

З цього можна зробити висновки, що побудова кластеру є цілком доцільно та бажаною перспективою, що забезпечить наступне:

1) Утилізацію доступних потужностей кафедри
2) Легке оновлення структури кластеру
3) Високу доступність сервісу
4) Надання корисних навчиок студентам
5) Захист інформаційних ресурсів кафедри

Кафедрі буде запропоновано на ресурсах, що доступні створити кластер Kubernetes, що згодом можна буде розширити. Це надасть кафедрі гнучкий інструмент із розгортання інфраструктури для навчального процесу та внутрішніх процесів кафедри. Для цього буде мінімально оновити парк серверів

(додати 1, 2 середньої потужності серверів) та закупити нові жорсткі диски. Побудований кластер після розширення дозволить перенести функціонал баз даних, файлового обміну та моніторингових систем цілком на себе. Не є доцільним відмовлятись від існуючого Xen, оскільки він працює із іншим рівнем віртуалізації, а також сервери Windows із їх функцоналом поки що не можливо перенести у функціонал контейнерів, тому технології Xen та Kubernetes будуть працювати у синергії, доповнючи одне одного.

4 РОЗРОБЛЕННЯ СТАРТАП-ПРОЕКТУ

Метою даного розділу є проведення маркетингового аналізу стартап проекту для визначення принципової можливості його ринкового впровадження та можливих напрямів реалізації цього впровадження.

4.1 Опис ідеї проекту

Опис ідеї стартап-проекту наведено у таблиці 5.1.

Таблиця 5.1 – Опис ідеї стартап-проекту

Зміст ідеї	**Напрямки застосування**	**Вигоди для користувача**
Дослідження інфраструктури кафедри, мікросервісів та оркетсраторів	1. Дослідження парадигми мікросервісів	Дозволяє зразу визначити доцільність побудови
	2. Дослідження оркестраторів	Дозволяє обрати оркестартор, що максимально утилізує доступні ресурси кафедри
	3. Наповнення кластеру доцільними сервісами	Дозволяє побудувати пайплайни для обробки інформації, збереженя інформації, створення ресурсів

Таблиця 5.1 – Опис ідеї стартап-проекту		
Зміст ідеї	**Напрямки застосування**	**Вигоди для користувача**
		для навчання студентів

У таблиці 5.2 наведено сильні, слабкі та нейтральні характеристики ідеї проекту.

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	**Конкурент1**	**Конкурент2**	**Конкурент3**			
	Форма реалізаціх	Kubernetes	Mesos	Swarm	Nomad			+

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	**Конкурент1**	**Конкурент2**	**Конкурент3**			
	Собівартість	Низька	Низька	Низька	Висока		+	
	Утилізація ресурсів	+	+	+	+			+
	Швидкість	+/-	-	+	+	+		

4.2 Технологічний аудит ідеї проекту

Технологічну здійсненність ідей проекту наведено у таблиці 5.3.

Таблиця 5.3 – Технологічна здійсненність ідеї проекту

№ п/п	Ідея проекту	Технології її реалізації	Наявність технологій	Доступність технологій
1	Створення кластеру на мікросервісній архітектурі	Kuberenets	Наявна	Безкоштовна, доступна
		CoreOS	Наявна	Безкоштовна, доступна
		Ingress, pfSense	Наявна	Безкоштовна, доступна
2	Створення Необхідних контейнерів	LDAP, ownCloud, Prometheus, Moodle	Наявні	Безкоштовні, доступні

Обрана технологія реалізації ідеї проекту: Створення кластеру – Kubernetes тому що члени команди мають досвід роботи а також через поширеність технологій та простоті розробки образів Docker - бо безкоштовний а також багато прикладів, легкість контролю та керування

4.3 Аналіз ринкових можливостей запуску стартап-проекту

Попередню характеристику потенційного ринку стартап-проекту наведено у таблиці 5.4.

Таблиця 5.4 – Попередня характеристика потенційного ринку стартап-проекту

	Показники стану ринку (найменування)	**Характеристика**
	Кількість головних гравців, од	9
	Загальний вартість побудови, грн/ум.од	5000 грн./ум.од
	Динаміка ринку (якісна оцінка)	Зростає
	Наявність обмежень для входу (вказати характер обмежень)	Немає
	Специфічні вимоги до стандартизації та сертифікації	Немає
	Середня норма рентабельності в галузі (або по ринку), %	R = (3000000 * 100) / (1000000 * 12) = 25%

Характеристику потенційних клієнтів стартап-проекту наведено у таблиці 5.5.

Таблиця 5.5 – Характеристика потенційних клієнтів стартап-проекту

№ п/п	**Потреба, що формує ринок**	**Цільова аудиторія (цільові сегменти ринку)**	**Відмінності у поведінці різних потенційних цільових груп клієнтів**	**Вимоги споживачів до товару**
1.	Необхідність обудови, чи переносу інфраструктури на мікросервісну архітектуру	Установи,. Що бажають оптимізувати свою архітектуру	Різні розміри устнов, статут установ, розмір інфраструктури	Наявність ресурсів, складність інфраструктури

Фактори загроз наведено у таблиці 5.6.

Таблиця 5.6 – Фактори загроз

№ п/п	**Фактор**	**Зміст загрози**	**Можлива реакція компанії**
1.	Конкуренція	Складність реалізації, Обмеженість ресурсів, Обмеження логіки сервісів	1) Відмова від проекту
2.	Зміна потреб користувачів	Користувачам необхідне	1) Передбачити можливість

		програмне забезпечення з іншим функціоналом	додавання нового функціоналу до створюваного кластеру

Фактори можливостей наведено у таблиці 5.7.

Таблиця 5.7 – Фактори можливостей

№ п/п	**Фактор**	**Зміст можливості**	**Можлива реакція компанії**
1.	Зростання можливостей потенційних покупців	Ріст зацікавленості до продукту серед інших груп користувачів з різним рівнем технічної грамотності	Додати підказки, інструкції та демонстрації роботи системи
2.	Зниження довіри до конкурента 3	У ПЗ конкурента №3 нещодавно була знайдена помилка, завдяки чому вдалося отримати контроль над системою третій особі	При виході на ринок звертати увагу покупців на безпеку нашого ПЗ

У таблиці 5.8 наведено ступеневий аналіз конкуренції на ринку.

Таблиця 5.8 – Ступеневий аналіз конкуренції на ринку

Особливості конкурентного середовища	**В чому проявляється дана характеристика**	**Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентоспроможною)**
1. Вказати тип конкуренції - досконала	Існує 3 фірми-конкурентки на ринку	Врахувати ціни конкурентних компаній на початкових етапах ,вказати на конкретні переваги перед конкурентами
2. За галузевою ознакою - внутрішньогалузева	Конкуренти мають ПЗ, яке використовується лише всередині даної галузі	Створити основу інфраструктури таким чином, щоб можна було легко переробити дану інфраструктуру для використання у інших галузях
4. Конкуренція за видами товарів: - товарно-видова	Види товарів є однаковими, а саме – послуга створення	Створити інфраструктуру, враховуючи недоліки конкурентів
5. За характером конкурентних переваг - нецінова	Вдосконалення технології створення ПЗ, щоб собівартість була нижчою	Використання менш дорогих технологій для розробки, ніж використовують конкуренти

У таблиці 5.9 наведено аналіз конкуренції в галузі за М. Портером.

Таблиця 5.9 – Аналіз конкуренції в галузі за М. Портером

Складові аналізу	**Прямі конкуренти в галузі**	**Потенційні конкуренти**	**Постачальники**	**Клієнти**	**Товари-замінники**
	Навести перелік прямих конкурентів	**Визначити бар'єри входження в ринок**	**Визначити фактори сили постачальників**	**Визначити фактори сили споживачів**	**Фактори загроз з боку замінників**
Висновки:	Існує 3 конкуренти на ринку. Найбільш схожим за виконанням є конкурент 2, так як його рішення також швидко працююче	Є конкуренти, є можливість виходу на ринок	Постачальників багато, тому можна вважати, що вони не диктують умови на ринку	Важливим для користувача є стабільна робота інфраструктури	Товари-замінники можуть використати більш швидку технологію створення кластеру та зменшити собівартість послуги за рахунок

Таблиця 5.9 – Аналіз конкуренції в галузі за М. Портером					
Складові аналізу	**Прямі конкуренти в галузі**	**Потенційні конкуренти**	**Постачальники**	**Клієнти**	**Товари-замінники**
	Навести перелік прямих конкурентів	**Визначити бар'єри входження в ринок**	**Визначити фактори сили постачальників**	**Визначити фактори сили споживачів**	**Фактори загроз з боку замінників**
					швикості реалізації.

У таблиці 5.10 наведено обґрунтування факторів конкурентоспроможності.

Таблиця 5.10 – Обґрунтування факторів конкурентоспроможності		
№ п/п	**Фактор конкурентоспроможності**	**Обґрунтування (наведення чинників, що роблять фактор для порівняння конкурентних проектів значущим)**
1.	Простота архітектури	Простота роботи побудови та програми оркестратора спрощує роботу користувачеві, що робить її більш комфортною та зручною
2.	Утилізація ресурсів	Дозволяє уилізувати максимум ресурсів за мінімальну ціну

У таблиці 5.11 наведено порівняльний аналіз сильних та слабких сторін

Таблиця 5.11 – Порівняльний аналіз сильних та слабких сторін

	Фактор конкурентоспроможності	**Бали 1-20**	**Рейтинг товарів-конкурентів у порівнянні**						
	Простота архітектури	20							
	Утилізація ресурсів	15							

У таблиці 5.12 наведено SWOT- аналіз стартап-проекту.

Таблиця 5.12 – SWOT- аналіз стартап-проекту

Сильні сторони: Простота архітектуриу	Слабкі сторони: Високий рівень абстракції, потреби до заліза
Можливості: у конкурента 2 виявлена складність із керуванням архітектуриз; ацікавленість продуктом ширших груп споживачів	Загрози: конкуренція, зміна потреб користувачів

У таблиці 5.13 наведено альтернативи ринкового впровадження стартап-проекту.

Таблиця 5.13 – Альтернативи ринкового впровадження стартап-проекту

№ п/п	Альтернатива (орієнтовний комплекс заходів) ринкової поведінки	Ймовірність отримання ресурсів	Строки реалізації
1.	Використання Kubernetes з кластером Ingress без overlay	85%	4 місяці
2.	Використання Kubernetes з кластером Ingress з overlay та nginx-ingress мережею	90	4,5 місяці

Обираємо альтернативу 2, тому що вона має більшу ймовірність отримання ресурсів.

Після аналізу зазначити обрану альтернативу.

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

4.4 Розроблення ринкової стратегії проекту

У таблиці 5.14 наведено вибір цільових груп потенційних споживачів.

Таблиця 5.14 – 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.	Наукові установи	Критичним є низька ціна послуги та підписання контракту на підтримку	Інтеграція існуючих сервісів	Існує 3 конкуренти, які надають схожі рішення.	Важко зайти у сегмент, через низьке фінансування та технічну застарілість ресурсу
2.	Комерційні установи	Критичним є якість побудови, що буде виконувати потреби бізнес логіки	Контроль параметрів Стабільність Дешева ціна		Маючи стабільний кластер, вискоку швидкість виконання та не високу ціну легко знайти замовника

Які цільові групи обрано: групи 1 та 2.

У таблиці 5.15 наведено визначення базової стратегії розвитку.

Таблиця 5.15 – Визначення базової стратегії розвитку

	Обрана альтернатива розвитку проекту	**Стратегія охоплення ринку**	**Ключові конкурентоспроможні позиції відповідно до обраної альтернативи**	**Базова стратегія розвитку***
	Використання Kubernetes з кластером Ingress з overlay та nginx-ingress мережею	Ринкове позиціювання	Простота використання, утилізація ресурсу	Диференціації

У таблиці 5.16 наведено визначення базової стратегії конкурентної поведінки.

Таблиця 5.16 – Визначення базової стратегії конкурентної поведінки

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

			більшу швидкіть та продуктивність	

У таблиці 5.17 наведено визначення стратегії позиціонування.

Таблиця 5.17 – Визначення стратегії позиціонування

	Вимоги до товару цільової аудиторії	**Базова стратегія розвитку**	**Ключові конкурентоспроможні позиції власного стартап-проекту**	**Вибір асоціацій, які мають сформувати комплексну позицію власного проекту (три ключових)**
	Простота налаштування та утилізація ресурсу	Диференціації	Простота користувацького інтерфейсу, що дозволяє спростити роботу з системою, можливість легкого поновлення та розгортання сервісу	Кастомізація, простота, гнучкість

4.5 Розроблення маркетингової програми стартап-проекту

У таблиці 5.18 наведено визначення ключових переваг концепції потенційного товару.

Таблиця 5.18 – Визначення ключових переваг концепції потенційного товару

№ п/п	Потреба	Вигода, яку пропонує товар	Ключові переваги перед конкурентами (існуючі або такі, що потрібно створити
1.	Утилізація	Утилізувати усю потужність ресурсів та вижати із них максимум	Легковісність, простота інфраструктури
2.	Простота інтерфейсу	Простота та зручність ПЗ	Користувачам не потрібно замислюватися над тим як робити тонке налаштування, живучість сервісу – сервіс має алгоритм «самолікування», що з мінімальними втратами в часі сам відновить роботу

У таблиці 5.19 наведено визначення меж встановлення ціни.

Таблиця 5.19 – Визначення меж встановлення ціни

№ п/п	Рівень цін на товари-замінники	Рівень цін на товари-аналоги	Рівень доходів цільової групи споживачів	Верхня та нижня межі встановлення ціни на товар/послугу
1	6000	7000	20000	5000

У таблиці 5.20 наведено формування системи збуту.

Таблиця 5.20 – Формування системи збуту

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

У таблиці 5.21 наведено концепція маркетингових комунікацій

Таблиця 5.21 – Концепція маркетингових комунікацій

№ п/п	Специфіка поведінки цільових клієнтів	Канали комунікацій, якими користуються цільові клієнти	Ключові позиції, обрані для позиціонування	Завдання рекламного повідомлення	Концепція рекламного звернення
1.	Замовлення через інтернет, або у компанії посередників	Інтернет	Утилізація ресурсу, швидкість налашування, інтеграція існуючих процесів	Показати переваги рішення, у тому числі і перед конкурентами	Демо-ролик із використання

5 Перелік Посилань

1. Загальна інформація про мікросервіси - Режим доступу - https://habrahabr.ru/company/latera/blog/279105/
   Дата доступу : 20.02.2017.
2. Блог учаснику проекту Mesos - Режим доступу - https://www.quora.com/profile/Florian-Leibert
   Дата доступу 05.06.2017
3. Репозиторії учасника проекту Мезос - Режим доступу - https://github.com/ FlorianLeibert /
   Дата доступу 05.06.2017
4. Блог із інформацією про кластер кубернетес - Режим доступу - http://stytex.de/
   Дата доступу 05.06.2017
5. Блог із ілюстартивним прикладом кластеру Кубернетес - Режим доступу - https://andrewmichaelsmith.com/2016/05/my-kubernetes-setup/
   Дата доступу 05.06.2017
6. Репозиторій блогера, що працює з кубернетес - Режим доступу - https://github.com/stytexde
   Дата доступу 05.06.2017
7. Матеріал дослідження мережі для кластеру кубернетес, блог компанії Цитрікс - Режим доступу - https://www.citrix.com/blogs/2017/04/17/netscaler-and-kubernetes-an-enabler-for-digital-
   Дата доступу 05.06.2017
8. Матеріал дослідження мережі для кластеру кубернетес, репозиторій компанії Цитрікс - Режим доступу - https://github.com/citrix/kube-ingress-citrix-netscaler
   Дата доступу 05.06.2017
9. Матеріал дослідження мережі для кластеру кубернетес, приклад мережево налаштування, резпозиторій компанії Кубернетес - Режим доступу - https://github.com/kubernetes/contrib/tree/master/ingress/controllers/nginx/examples
   Дата доступу 05.06.2017
10. Блог із матеріалами для налаштуваннями кластеру кубернетес - Режим доступу - https://andrewmichaelsmith.com/2016/05/my-kubernetes-setup/
    Дата доступу 05.06.2017
11. Стаття на feed stream, Матеріал порівняння оркестраторів - Режим доступу - https://medium.com/@mustwin/a-handy-guide-to-the-mesos-kubernetes-swarm-jungle-ad6bc086c736#.b9s4qvu66
    Дата доступу 13.04.2017
12. Матеріал порівняння оркестраторів, документація проекту Докер - Режим доступу - https://docs.docker.com/swarm/discovery/
    Дата доступу 13.04.2017

13. Матеріал порівняння оркестраторів, документація проекту Докер - Режим доступу - https://docs.docker.com/engine/swarm/
Дата доступу 13.04.2017
14. Матеріал порівняння оркестраторів, офіційний репозиторій проекту Мезос - Режим доступу - https://mesosphere.github.io/mesos-dns/
Дата доступу 13.04.2017
15. Матеріал порівняння оркестраторів, документація проекту Кубернетес - Режим доступу - https://kubernetes.io/docs/user-guide/connecting-applications/
Дата доступу 13.04.2017
16. Матеріал порівняння оркестраторів, документація проекту Кубернетес - Режим доступу - https://kubernetes.io/docs/user-guide/services/#environment-variables
Дата доступу 13.04.2017
17. Матеріал порівняння оркестраторів, блог компанії Докер - Режим доступу - https://blog.docker.com/2017/02/docker-secrets-management/
Дата доступу 13.04.2017
18. Матеріал порівняння оркестраторів, документація проекту Кубернетес - Режим доступу - https://kubernetes.io/docs/user-guide/secrets/
Дата доступу 13.04.2017
19. Матеріал порівняння оркестраторів, репозиторій проекту Мезос - Режим доступу - https://github.com/mesosphere/marathon/issues/2295
Дата доступу 05.06.2017
20. Матеріал порівняння оркестраторів, документація проекту Докер - Режим доступу - https://docs.docker.com/compose/environment-variables/
Дата доступу 13.04.2017
21. Матеріал порівняння оркестраторів, документація проекту Кубернетес - Режим доступу - https://kubernetes.io/docs/user-guide/configmap/
Дата доступу 13.04.2017
22. Матеріал порівняння оркестраторів, документаціяя проекту Кубернетес - Режим доступу - https://kubernetes.io/docs/user-guide/environment-guide/
Дата доступу 13.04.2017
23. Матеріал порівняння оркестраторів, документація проекту Мезос - Режим доступу - https://mesos.apache.org/documentation/latest/logging/
Дата доступу 13.04.2017
24. Матеріал порівняння оркестраторів, стаття-диспут моніторнигу кластерів - Режим доступу - https://technologyconversations.com/2016/11/07/collecting-metrics-and-monitoring-the-cluster/
Дата доступу 13.04.2017

25. Матеріал порівняння оркестраторів, документація проекту Мезос - Режим доступу - http://mesos.apache.org/documentation/latest/monitoring/
Дата доступу 13.04.2017
26. Матеріал порівняння оркестраторів, документація проекту Докер - Режим доступу - https://docs.docker.com/swarm/multi-manager-setup/
Дата доступу 13.04.2017
27. Матеріал порівняння оркестраторів, документація фреймворку Марафон - Режим доступу - https://mesosphere.github.io/marathon/docs/service-discovery-load-balancing.html
Дата доступу 13.04.2017
28. Матеріал порівняння оркестраторів, документація проекту Докер - Режим доступу - https://docs.docker.com/swarm/networking/
Дата доступу 13.04.2017
29. Матеріал порівняння оркестраторів, документація проекту Докер - Режим доступу - https://docs.docker.com/engine/reference/commandline/stack_deploy/
Дата доступу 13.04.2017
30. Матеріал порівняння оркестраторів, документація проекту Докер - Режим доступу - https://docs.docker.com/docker-cloud/apps/service-scaling/
Дата доступу 13.04.2017
31. Офіційний сайт документації Кубернетес - Режим доступу - https://kubernetes.io/docs/user-guide/horizontal-pod-autoscaling/
Дата доступу 13.04.2017
32. Документація фреймворку Марафон - Режим доступу https://mesosphere.github.io/marathon/docs/application-basics.html
Дата доступу 13.04.2017
33. Офіційний сайт компанії Докер - Режим доступу - Docker.io
Дата доступу 13.04.2017
34. Офіційний сайт проекту Кубернетес - Режим доступу - Kubernetes.io
Дата доступу 13.04.2017
35. Офіційний сайт проекту Мезос - Режим доступу - mesos.apache.org
Дата доступу 13.04.2017