Міністерство освіти і науки України
Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій
Кафедра кібербезпеки та захисту інформації

ДОПУСТИТИ ДО ЗАХИСТУ:
В.о. завідувача кафедри
кібербезпеки
та захисту інформації
_____ Іван ПАРХОМЕНКО
«__» _____ 2025 р.

ПОЯСНЮВАЛЬНА ЗАПИСКА

кваліфікаційної роботи

галузь знань *12 Інформаційні технології*
(шифр і назва галузі знань)

спеціальність *125 Кібербезпека та захист інформації*
(код і назва спеціальності)

освітній ступень *магістр*

освітньо-наукова програма *Кібербезпека*
(назва освітньої програми)

на тему: «Багаторівнева модель захисту високонавантажених систем у хмарі»

Виконавець: студент ІІ курсу, групи КБм-22

_____ **Олексій БУРЯТОВ**
(підпис) (Ім'я, ПРІЗВИЩЕ)

	Ім'я, ПРІЗВИЩЕ	**Підпис**
Науковий керівник	Володимир НАКОНЕЧНИЙ	
Нормоконтроль	Іван БІЛОКОНЬ	

Київ 2025

Міністерство освіти і науки України
Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій
Кафедра кібербезпеки та захисту інформації

ЗАТВЕРДЖЕНО:
В.о. завідувача кафедри
кібербезпеки
та захисту інформації
_____ Іван ПАРХОМЕНКО
«25» жовтня 2024 р.

ЗАВДАННЯ
на виконання кваліфікаційної роботи

спеціальності *125 Кібербезпека та захист інформації*
(код і назва спеціальності)

освітній ступень *магістр*

Здобувача(ки) КБм-22 (група) Бурятова Олексія Олексійовича (прізвище ім'я по-батькові)

Тема кваліфікаційної роботи Багаторівнева модель захисту високонавантажених систем у хмарі

1. ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Рішення засідання кафедри кібербезпеки та захисту інформації факультету інформаційних технологій протокол № 4 від 24.10.2024 р.

2. МЕТА ТА ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Об'єкт досліджень	Процес взаємодії з даними у високонавантажених системах у хмарному середовищі.
Предмет досліджень	Методи захисту високонавантажених систем від несанкціонованого доступу.
Мета	Розробка методу захисту високонавантажених систем у хмарному середовищі.
Вихідні дані для проведення роботи	Методи захисту високонавантажених систем у публічних та приватних хмарних середовищах.

3. ОЧІКУВАНІ НАУКОВІ РЕЗУЛЬТАТИ

Наукова новизна Удосконалення методу захисту високонавантажених систем у хмарному середовищі за рахунок впровадження автоматизації розгортання оточення та GitOps підходу.

Практична цінність Покращення безпеки інформаційних систем у хмарних середовищах.

4. ЕТАПИ ВИКОНАННЯ РОБОТИ

Найменування етапів робіт	Строки виконання робіт (початок-кінець)
Уточнення постановки задачі	25.10.2024 – 29.12.2024
Аналіз літературних джерел	30.12.2024 – 12.02.2025
Ознайомлення з провайдерами хмарних платформ	13.02.2025 – 21.02.2025
Дослідження сервісів, що надають провайдери хмарних платформ	22.02.2025 – 26.02.2025
Дослідження існуючих загроз, що впливають на працездатність хмарних компонентів	27.02.2025 – 04.03.2025
Аналіз архітектури високонавантаженої системи в хмарному середовищі	05.03.2025 – 10.03.2025
Аналіз автоматизації розгортання та підтримки інформаційної системи в хмарному середовищі	11.03.2025 – 17.03.2025
Аналіз GitOps стратегії по розгортанню та керуванню хмарних компонентів	18.03.2025 – 19.03.2025
Налаштування CI/CD, збір логів та моніторингу інформаційної системи в хмарному середовищі	20.03.2025 – 17.04.2025
Розробка рекомендацій по реагуванню та усуненню загроз у хмарному середовищі	18.04.2025 – 25.04.2025
Оформлення пояснювальної записки згідно методичних рекомендацій	26.04.2025 – 15.05.2025
Подача пакету документів на розгляд ЕК	15.05.2025 – 19.05.2025

Завдання видав _____ Володимир НАКОНЕЧНИЙ
(підпис) (Ім'я, ПРІЗВИЩЕ)

Завдання прийняв до виконання _____ Олексій БУРЯТОВ
(підпис) (Ім'я, ПРІЗВИЩЕ)

Дата видачі завдання: 25.10.2024 р.

Термін подання кваліфікаційної роботи до ЕК 19.05.2025 р.

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи «Багаторівнева модель захисту високонавантажених систем у хмарі»: 68 сторінок, 34 рисунки та 4 таблиці. 23 літературних джерела.

Об’єкт дослідження – процес взаємодії з даними у високонавантажених системах у хмарному середовищі.

Мета роботи – розробка методу захисту високонавантажених систем у хмарному середовищі.

Методи дослідження – метод автоматизації процесу розгортання та оновлення інформаційних систем, метод GitOps.

У роботі проаналізовано сучасні методи побудови інформаційної системи в хмарному середовищі. Проведено аналіз компонент архітектури високонавантаженої системи. Запропоновано метод захисту інформаційної системи шляхом впровадження GitOps підходу, DevOps практик та безперервної інтеграції коду.

Наукова новизна: запропоновано удосконалення методу захисту високонавантажених систем у хмарному середовищі за рахунок впровадження автоматизації розгортання оточення та GitOps підходу.

Актуальність теми:

Із поширенням використання штучного інтелекту все більше необхідно тримати ресурсів, щоб бути відповідати на сьогоденні тенденції. Сучасний підхід розгортання високонавантажених систем спростився з появою провайдерів хмарних послуг. Для того, щоб ефективно та безпечно користуватися запропонованими інструментами, необхідно запевнити клієнтів та замовників у захищеності оточення.

Також за останні роки рівень загрози кібератак нечувано зріс. Виконання базових налаштувань хмарного середовища, побудова багаторівневої моделі захисту з поєднанням автоматизації розгортання й підтримки інформаційної системи з GitOps підходом є доцільним рішенням аби збільшити захист інформаційних систем у хмарному середовищі.

Ключові слова: GitOps, Microsoft Azure, Infrastructure as Code, логування, моніторинг.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ

GitOps – Git Operations
DevOps – Development and IT Operations
IaC – Infrastructure as Code
AD – Active Directory
IdM – Identity Management
SSO – Single Sign On
IaaS – Infrastructure as a service
CRM – Штучний Інтелект
PaaS – Platform as a service
FaaS – Function as a service
SaaS – Software as a service
TLS – Transport Layer Security
VPN – Virtual Private Network
MTTR – Mean Time To Recovery
SaC – Security as Code
SIEM – Security Information and Event Management
ETL – Extract, Transform, Load
GUI – Graphical User Interface
MD – Markdown
CLI – Command Line Interface

ЗМІСТ

ВСТУП

Із всесвітнім прогресом розвитку нових технологій, людство має на меті полегшити життя та замінити однотипну й монотонну роботу, яку може виконати за неї «розумний дім». Усе частіше можна зустріти в житті людей автономні прилади, роботи-пилососи тощо. Нещодавно взагалі був прорив штучного інтелекту. Тому постає питання в тому, щоб звичайну роботу, результат якої можна спрогнозувати, віддати автоматизованим системам, уникаючи можливого людського фактору.

Загальна теорія систем вивчає можливі аспекти дослідження систем, в тому числі й прийняття рішень у них. Системний аналіз є складовою частиною теорії систем разом із такими дисциплінами, як: кібернетика, інформатика, дослідження операцій та системотехніка.

Програмні системи проходять свій життєвий цикл, починаючи з невеликих розроблених рішень, повністю зрозумілих одній людині, але через швидке зростання технічного боргу в монолітній архітектурі сервісів було обрано мікросервісний тип архітектури, здатної надійно масштабуватися як: scale out – вширшки (більше користувачів), так і scale up – вгору (більше функцій).

Високонавантажені системи – це системи, здатні обробляти величезну кількість запитів від користувачів, працювати з великими масивами даних у реальному часі та забезпечувати стабільну роботу навіть у критичні періоди навантаження. Такі системи мають забезпечувати не лише високу продуктивність, але й мати змогу адаптуватися до швидкозмінних умов, володіти надійністю та масштабованістю. Наприклад, системи онлайн-банкінгу, транспортної логістики або медичних досліджень активно використовують хмарне середовище для аналізу даних і формування рекомендацій у реальному часі.

Постає питання в тому як правильно організувати інформаційну систему, щоб вона була:

- спроектована та впроваджена згідно з вимогами замовника;
- оптимізована та економічно окупною;

- захищеною від кіберзагроз;
- масштабованою та відновлюваною (у разі збоїв, зламу техніки тощо).

Усе це можливо досягти за умови ретельної підготовки, досліджень та компетентної команди розробників, системних адміністраторів, спеціалістів із кібербезпеки тощо. Наразі тенденції показують попит на використання хмарних ресурсів. Це дозволяє:

- легко масштабуватися;
- платити лише за використаний час того чи іншого сервісу без необхідності тримати купу обладнання та оренди дата-центрів;
- використовувати новітні технології, включно з ШІ інструментами.

Тим не менше, використання публічних та/або приватних інфраструктур у хмарних середовищах не позбавляє від відповідальності за захист та забезпечення надійності компонентів системи. Визначивши методи та спрямувавши увагу на додатковому захисті інфраструктури, це допоможе мінімізувати реальні наслідки у випадку кібератаки.

РОЗДІЛ 1
ТЕОРЕТИЧНІ ОСНОВИ ПОБУДОВИ СИСТЕМ У ХМАРІ ТА МЕТОДИ ЇХ ВИКОРИСТАННЯ

1.1 Технічні особливості хмарних сервісів

Хмарні технології – це парадигма обчислень, яка дозволяє отримувати доступ до обчислювальних ресурсів (наприклад, обчислювальної потужності, програмного забезпечення, сховища даних) через Інтернет за принципом Pay as you go – «плати за використання». Замість того, щоб купувати та обслуговувати власне обладнання, користувачі орендують необхідні ресурси у провайдера хмарних послуг [1].

Хмара – це абстрактне представлення великої кількості обчислювальних ресурсів, об’єднаних в єдину працюючу систему. Користувачем хмарних послуг є організація або приватна особа, яка використовує хмарні ресурси для своїх потреб. Провайдером хмарних послуг є компанія, яка володіє та управляє хмарною інфраструктурою і надає користувачам доступ до своїх ресурсів. Кожне хмарне середовище надає доступ до певних сервісів. Сервіс – це конкретна послуга, яку надає провайдер (наприклад, віртуальні машини, бази даних, програмне забезпечення тощо).

Існує три основні моделі розгортання хмарних послуг[2]:

- IaaS (Infrastructure as a Service) – інфраструктура як послуга: Провайдер надає користувачам базову обчислювальну інфраструктуру (сервери, мережі, сховища), яку користувачі можуть конфігурувати під свої потреби. Це найнижчий рівень абстракції в хмарних обчисленнях;
- PaaS (Platform as a Service) – платформа як послуга: провайдер надає платформу для розробки і запуску додатків, включаючи операційну систему, середовище розробки, бази даних та інші інструменти. При використанні цього підходу користувачам не потрібно турбуватися про управління інфраструктурою;

- SaaS (Software as a Service) – програмне забезпечення як послуга: провайдер надає готові програмні продукти через веб-інтерфейс (наприклад, електронна пошта або CRM-системи). Користувачам не потрібно встановлювати і обслуговувати програмне забезпечення.

Типова архітектура хмарної платформи включає різні компоненти. API інтерфейс – це точка доступу користувачів до хмарних послуг через веб-інтерфейс або API. API (Application Programming Interface) - це набір інструментів, протоколів і визначень, які дозволяють програмам взаємодіяти один з одним. У контексті хмарних обчислень API відіграє ключову роль у наданні користувачам і розробникам доступу до всіх можливостей хмарної платформи. API служить точкою доступу до хмарних сервісів, дозволяючи управляти ресурсами (серверами, сховищами, базами даних, мережевими налаштуваннями тощо) через програмний інтерфейс, а не тільки через GUI – графічний веб-інтерфейс.

Контролер – це компонент, який керує доступом користувачів до ресурсів, розподіляє ресурси між користувачами і забезпечує безпеку. Контролер в хмарних обчисленнях діє як центральний елемент управління, який надає користувачам доступ до обчислювальних ресурсів, контролює їх розподіл і відповідає за безпеку всієї інфраструктури. Коли користувач відправляє запит на створення або використання певного ресурсу, контролер обробляє запит, перевіряючи токени авторизації, права доступу та автентичність користувача відповідно до встановлених політик безпеки. Таким чином, він виконує функції автентифікації та авторизації, дозволяючи лише авторизованим користувачам взаємодіяти з певними ресурсами.

Разом з контролем доступу контролер також розподіляє ресурси між користувачами або додатками. Це означає, що він розглядає як буде використовуватися час процесора, оперативна пам'ять або сховище, щоб уникнути перезавантаження системи та ефективно підтримувати роботу процесів. Коли навантаження збільшується, процес може ініціювати масштабування - додавання нових ресурсів або перерозподіл існуючих для підтримки стабільності.

Контролер приділяє особливу увагу безпеці – він реалізує механізми шифрування даних, логує все в журнал подій, забезпечує доступ до конфіденційної

інформації та запобігає деяким загрозам. Ще однією важливою функцією є постійний моніторинг стану системи, що дозволяє швидко реагувати на проблеми або критичні зміни в операційній інфраструктурі.

Мережа – це віртуальна мережа, яка з'єднує всі компоненти хмарної платформи і забезпечує передачу даних між ними. Мережа в хмарній інфраструктурі виступає як віртуальне середовище, яке об'єднує всі компоненти хмарної платформи і забезпечує безперервну і надійну передачу даних між ними. По суті це не фізична, а логічно сформована структура, яка дозволяє різним службам, віртуальним машинам, базам даних, сховищам та іншим елементам хмарної екосистеми взаємодіяти один з одним незалежно від їх фізичного розташування.

Віртуальна мережа створюється за допомогою спеціального програмного забезпечення, яке повторює роботу звичайної комп'ютерної мережі, але вже всередині хмари. Такий підхід забезпечує гнучкість в конфігурації, масштабуванні та ізоляції ресурсів. Це дає можливість групувати середовища для різних завдань або користувачів, що значно підвищує рівень безпеки і продуктивності.

Передача даних в такій мережі здійснюється через віртуальні маршрутизатори, комутатори і брандмауери, які виконують ті ж функції, що і фізичні пристрої, але у віртуальному форматі. Це дозволяє точно контролювати трафік, застосовувати політики доступу, фільтрувати пакети і забезпечувати захист від несанкціонованих перешкод.

Мережева інфраструктура є критично важливою для ефективного функціонування хмарної платформи, оскільки вона забезпечує зв'язок між усіма її частинами, забезпечує обмін даними в реальному часі та гарантує швидкість, безпеку та стабільність взаємодії з сервісом.

Обчислювальні ресурси в хмарній інфраструктурі є ключовими елементами, які надаються користувачам для виконання різних обчислювальних завдань. Вони включають віртуальні машини, контейнери, lambda функції та інші компоненти, які забезпечують обробку, зберігання та виконання даних у хмарному середовищі.

Віртуальні машини (VM) є одним з найпоширеніших видів обчислювальних ресурсів. Вони працюють по принципу роботи фізичних серверів, надаючи

користувачам ізольоване середовище з власною операційною системою, оперативною пам'яттю, дисковим простором та іншими параметрами. Це дозволяє запускати будь-яке програмне забезпечення, яке нібито працює на окремому комп'ютері, з можливістю масштабуватися і швидко переміщуватися між фізичними нодами.

Контейнери, на відміну від віртуальних машин, є легшими і швидше розгортаються. Вони ізольовані один від одного, але ділять ядро операційної системи, що значно скорочує споживання ресурсів. Контейнери особливо корисні для розгортання мікросервісних архітектур, де кожна служба працює незалежно і може масштабуватися незалежно.

Lambda функції - це ще одна форма обчислення, яка дозволяє запускати окремі функції або фрагменти коду без необхідності керувати інфраструктурою. Користувач просто завантажує функцію, а хмара автоматично виділяє ресурси для її виконання і зупиняє їх після її закінчення. Це дуже зручно для завдань, які керуються подіями або вимагають високої масштабованості при мінімальних витратах.

Усі ці обчислювальні ресурси надаються користувачам відповідно до їхніх потреб - як послуга (IaaS, PaaS або FaaS) - і можуть бути швидко масштабовані, автоматизовані та інтегровані з іншими хмарними службами. Вони є основою будь-якої хмарної платформи, оскільки дозволяють користувачам ефективно виконувати розрахунки, обробляти дані та запускати свої додатки без необхідності фізичного обладнання.

Сховище даних – це системи зберігання даних, які забезпечують надійне зберігання даних користувачів. Саме в них зберігатимуться всі дані, включно з чутливими. Варто пам’ятати, що безпека за сховищами даних повністю перекладається на тих, хто розгортає та використовує сервіси.

Хмарне середовище має свої переваги. Переваги хмарних технологій:

- Гнучкість. Можливість швидко масштабувати ресурси в залежності від потреб є одним із фактором вибору провайдерів хмарних послуг. У разі, якщо збільшується навантаження на поточну інфраструктуру – наприклад, зростання

кількості користувачів, то збільшення кількості реплік допоможе збалансувати навантаження.

- Економія. Є різні типи підписок у хмарних провайдерах – від Pay as you go (оплата тільки за використані ресурси) до місячних або річних контрактів.
- Доступність. Можливість доступу до ресурсів з будь-якого місця з підключенням до Інтернету є значною перевагою після того, як були карантинні заходи й єдиною точкою виходу в світ був лише інтернет.
- Надійність. Високий рівень доступності завдяки відмовостійкості та автоматичному відновленню.

1.2 Контейнеризація ресурсів у хмарі

Окремим напрямом у сучасному використанні хмарних ресурсів є контейнеризація та оркестрація.

Контейнеризація — це метод віртуалізації на рівні операційної системи, який дозволяє запускати та ізолювати застосунки разом з усіма необхідними залежностями у контейнерах. На відміну від традиційних віртуальних машин, контейнери є легкими, швидкими в запуску та не вимагають окремої операційної системи.

Контейнеризація має свої характеристики. Ізольованість контейнеру – це коли кожен контейнер працює незалежно. Ізоляція контейнера передбачає ізоляцію середовища виконання контейнерної програми від операційної системи сервер хоста та інших процесів, що виконуються на хості.

Ця ізоляція має кілька форм, включаючи ізоляцію файлової системи, ізоляцію мережі, ізоляцію системних викликів та ізоляцію використання ресурсів, таких як процесор та пам'ять.

Технічні деталі роботи ізоляції контейнерів залежать від контейнера. Кожен тип контейнера підходить до ізоляції контейнера по-різному, ізолюючи різні частини системи.

Контейнери Linux, такі як Docker, використовують cgroups, seccomp фільтри. Cgroups дозволяють встановлювати обмеження використання ресурсів на групу

процесів. Наприклад, cgroups може обмежувати використання різних ресурсів, таких як I/O диска, пам'ять, мережа, час процесора і навіть окремі процесори в багатоядерній системі. Seccomp дозволяє фільтрувати всі системні виклики, зроблені процесом. Ця фільтрація працює з технологією ядра, що дозволяє функціям у контейнері мати ізольований вигляд системи.

Також особливістю контейнеризації є невеликий розмір (менший порівняно з віртуальними машинами). У Linux контейнеризація використовує функції віртуалізації на рівні операційної системи, такі як простори імен та контрольні групи (cgroups). На відміну від традиційних віртуальних машин (VM), які вимагають повної гостьової операційної системи для кожного екземпляра, контейнери поділяють ядро ОС хост-системи. Це робить контейнери дуже легкими, швидшими для запуску та менш затратними в ресурсах, ніж віртуальні машини. Це дозволяє розробникам розгортати кілька контейнерів на одній VM для більш високої щільності та більш ефективного використання основних апаратних ресурсів.

Портативність – це можливсість один і той самий контейнер можна запускати в будь-якому середовищі: локальному, хмарному, тестовому, продакшн. Контейнери працюють в будь-якому місці, від локальної машини до хмари, без проблем сумісності. Контейнери забезпечують стабільне середовище виконання для додатків, що дозволяє їм надійно працювати в різних системах і платформах. Ця портативність усуває проблему локального розгортання й спрощує розгортання додатків, будь то локальне, в хмарі або в гібридних середовищах.

Автоматизація – це налаштування автоматичних процесій, які дозволяють контейнерам легко інтегруються в CI/CD-процеси.

Основний інструмент контейнеризації — це Docker. Docker - це платформа з відкритим кодом, яка дозволяє розробникам створювати, розгортати, запускати, оновлювати та керувати контейнерами. Контейнери спрощують розробку і доставку розподілених додатків. Вони стають все більш популярними, оскільки організації переходять до хмарної розробки та гібридних багатошарових середовищ. Розробники можуть створювати контейнери без Docker, працюючи безпосередньо з можливостями, вбудованими в Linux та інші операційні системи, але Docker робить

контейнеризацію швидше і простіше. За даними Docker, понад 20 мільйонів розробників використовують щомісяця. Як і інші технології контейнеризації, включаючи Kubernetes, Docker відіграє вирішальну роль у сучасній розробці програмного забезпечення, зокрема в архітектурі мікросервісів. Компонентами Docker є:

- Docker Engine – рушій, який запускає контейнери;
- Docker Image – шаблон з усім необхідним ПЗ;
- Docker Container – запущений екземпляр образуl
- Dockerfile – файл з інструкціями для створення образу.

Перевагами використання контейнеризації у хмарі є економічні та функціональні. Масштабованість: можна легко запустити декілька екземплярів одного застосунку. Це потрібно, щоб керувати збільшеним навантаженням або скорочуватися, щоб зберегти ресурси під час зменшення навантаження.

Гнучкість надає просте оновлення застосунків, зміна конфігурацій, відмовостійкість. На додаток до підтримки CLI для управління моделями, Docker Desktop тепер включає в себе спеціальний розділ в графічному інтерфейсі [3]. Це дає розробникам більше гнучкості для візуального перегляду, запуску та керування моделями поряд зі своїми контейнерами, обсягами та зображеннями.

Ефективне використання ресурсів контейнерами зосереджують уваги на використання ресурсів ОС більш економічно, ніж віртуальні машини. Це дозволяє створювати більше застосунків із на менших ресурсах.

Простота розгортання – це ще одна перевага використання Docker. Один образ – одна поведінка на будь-якому середовищі є запорукою цілісності даних.

1.3 Оркестрація ресурсів у хмарі

Оркестрація – це процес автоматичного управління життєвим циклом контейнерів: їх створенням, розгортанням, масштабуванням, балансуванням навантаження, оновленням та моніторингом тощо[4].

Одна з найвідоміших систем оркестрації – Kubernetes. Завданням оркестрації є:

- Автоматичне масштабування контейнерів при зміні навантаження;
- Перезапуск контейнерів у випадку помилки коду (використання Health checks);
- Розподіл навантаження між нодами кластеру;
- Безперервне оновлення застосунків без необхідності вимкнення роботи сервісів (rolling updates);
- Керування чутливими даними та конфігураціями – централізоване управління змінними середовища [4].

Kubernetes — це платформа з відкритим кодом для управління контейнеризованими застосунками в кластерному середовищі.

Основні компоненти Kubernetes наведені в таблиці 1.1.

Таблиця 1.1

Компоненти Kubernetes

Компонент	**Опис**
Container	найменша одиниця Kubernetes
Pod	одиниця, що містить один або декілька контейнерів
Node	фізичний або віртуальний сервер, на якому розгортаються Pod'и
Cluster	сукупність вузлів, які керуються централізовано
Deployment	об'єкт (сутність), який описує бажаний стан сервісу з необхідними ресурсами, включаючи змінні оточення, секрети тощо
Service	об'єкт, який описує мережевий доступ до застосунку
Ingress	об'єкт, який описує позакластерний доступ до сервісу
Persistent volume	об'єкт, що виділяє дисковий простір для застосунків

Варто провести порівняння між Kubernetes та Docker у цілому. При виборі між Kubernetes і Docker, розуміння їхніх переваг та недоліків допоможе з наслідками в майбутньому. Docker пропонує простий підхід до контейнеризації з мінімальним

налаштуванням. Розгортання додатків з використанням файлів YAML дозволяє здійснювати автоматичне управління, балансування навантаження серед контейнерів, а також вбудовану безпеку та контроль доступу.

Для менш складних робочих навантажень або коли простота є пріоритетом, Docker Swarm, інструмент для оркестрування контейнерів від Docker, може бути відповідним варіантом, навіть якщо Docker Swarm став застарілим, а Kubernetes став основним сервісом для оркестрування контейнерів.

Основна перевага, яку Kubernetes пропонує, - це можливість керувати кількома контейнерними додатками, поширеними на кількох VM. Незважаючи на те, що він вимагає більш просунутого початкового налаштування, він пропонує безліч потужних функцій. Він краще показує себе в керуванні, забезпечуючи і організовуючи складні програми.

Docker добре підходить для простих потреб контейнеризації та більш легких робочих навантажень, в той час як Kubernetes в управлінні складними додатками, забезпечуючи мережеві можливості та можливості масштабування. У таблиці 1.2 наведено порівняння функціоналу Kubernetes та Docker.

Таблиця 1.2

Порівняльна таблиця Kubernetes з Docker

Функціонал	**Можливості Docker.**	**Можливості Kubernetes**
Ізоляція ресурсів	Дозволяють.	Дозволяють.
Контроль мережевих доступів	Не дозволяють. Необхідно обмежувати доступ на створення контейнерів зайвим користувачам [5].	Дозволяють.
Розділення доступів за допомогою RBAC	Не дозволяють. Усі користувачі, які мають	Дозволяють

Продовження таблиці 1.2

	доступ до комп'ютеру з Docker, мають повний доступ до всіх компонент.	
Розділення доступів за допомогою RBAC	Не дозволяють. Усі користувачі, які мають доступ до комп'ютеру з Docker, мають повний доступ до всіх компонент.	Дозволяють
Резервне копіювання	Не дозволяють.	Дозволяють. Є можливість у створенні мульти-кластерів, які будуть дублювати дані на безпечне сховище
Безпечне збереження чутливих даних	Не дозволяють. Необхідно обмежувати права на файл, де будуть зберігатися sensitive дані.	Дозволяють. Додатково можливо примонтувати secret-об'єкти у вигляді файлу всередині контейнеру.
Масштабування ресурсів	Не дозволяють. Необхідно виконувати вручну або через стороннє ПЗ – Docker Swarm.	Дозволяють. Необхідні дії можна виконати на сторінці Microsoft Azure або через Terraform [4].

1.4 Контейнеризація та оркестрація в Microsoft Azure

Оскільки контейнеризація набула популярності, стала очевидною необхідність ефективного управління кількома контейнерами. Це призвело до розвитку платформ для оркестрування контейнерів, причому Kubernetes був одним із головних.

Розуміючи важливість контейнеризації в сучасній розробці програмного забезпечення, Microsoft Azure пропонує дві основні послуги: Azure Container Instances (ACI) і Azure Kubernetes Service (AKS).

Azure Container Instances (ACI) - це керований сервіс, який дозволяє запускати контейнери безпосередньо в публічній хмарі Microsoft Azure, не вимагаючи використання віртуальних машин (VM). Використовуючи Azure Container Instances, не потрібно надавати базову інфраструктуру або використовувати послуги вищого рівня для управління контейнерами. ACI надає основні можливості для управління групою контейнерів на хост-машині. Він підтримує використання повних контейнерних оркестрів, таких як Kubernetes, а також для більш складних завдань, таких як скоординовані оновлення та автоматизоване масштабування.

При розгортанні контейнерів у великих масштабах прийнято використовувати контейнерні оркестратори, такі як: Kubernetes, Nomad і Docker Swarm. Ці інструменти допомагають автоматизувати та керувати взаємодією між контейнерами та проблемами, такими як виділення ресурсів, мережа та управління сховищами. Екземпляри контейнерів Azure забезпечують деякі основні функції оркестрації контейнерів. За своєю складовою він не призначений для повноцінної платформи для оркестрування. У таблиці 1.3 наведено переваги між ACI та AKS.

Таблиця 1.3

Таблиця переваг Azure Container Instances та Azure Kubernetes Service

Інструмент	**Перевага**
Azure Container Instances	Простий сервіс для запуску контейнерів без управління інфраструктурою

Продовження таблиці 1.3

	Підходить для одноразових або легких задач (наприклад, обробка зображень, запуск скриптів)
	Швидкий запуск і оплата лише за фактичний час виконання. Azure Kubernetes Service (AKS)
Azure Kubernetes Service	Повністю керована платформа для розгортання Kubernetes
	Забезпечує автоматичне масштабування, оновлення кластерів, моніторинг та безпеку
	Інтегрується з Azure Active Directory, Azure Monitor, Azure Policy

Приклади використання AKS:

- Побудова мікросервісних застосунків;
- CI/CD з GitHub Actions або Azure DevOps [6];
- Створення resilient-архітектури з autoscaling та rolling updates.

Висновки до першого розділу

У цьому розділі було знайомство з хмарним середовищем. Було проаналізовано технічні особливості хмарних сервісів. Проведено порівняння між оркестрацією та контейнеризацією. Визначено, що в залежності від потреб проєкту, над яким працює команда розробників та DevOps інженерів, є різні варіанти сервісів, які пропонує провайдер хмарних послуг.

Було проаналізовано інструменти від Microsoft Azure, які в подальшому стануть основною платформою майбутньої інформаційної системи.

РОЗДІЛ 2
ТЕХНОЛОГІЇ БЕЗПЕРЕРВНОЇ ІНТЕГРАЦІЇ ТА ДОСТАВКИ ЯК ОСНОВА НАДІЙНОГО ФУНКЦІОНУВАННЯ ХМАРНИХ СЕРВІСІВ

2.1 Опис технології безперервної інтеграції та доставки

Більшість організацій не в змозі розгорнути зміни за лічені хвилини або години, замість цього потребують декілька тижнів або місяців. Вони також не можуть розгорнути сотні або тисячі змін за короткий проміжок часу.

В епоху, коли конкурентна перевага вимагає швидкої реакції, високого рівня обслуговування, і невпинних експериментів, ці організації знаходяться на значній відстаній, ніж конкуренти. Це значною мірою пояснюється їхня нездатність вирішити основні задачі та проблеми, з якими стикається компанія, включно з кіберінцидентами.

Сучасні хмарні сервіси, особливо високонавантажені системи, потребують не лише надійної інфраструктури, але й ефективних підходів до розробки, тестування та впровадження змін. Однією з ключових концепцій, що забезпечує стабільність і безперервність функціонування таких систем, є підхід CI/CD — Continuous Integration (безперервна інтеграція) та Continuous Delivery/Deployment (безперервна доставка або розгортання).

CI/CD — це автоматизований процес, який охоплює всі етапи життєвого циклу програмного забезпечення: від створення та перевірки коду до його впровадження в робоче середовище. Такий підхід дозволяє зменшити час між розробкою нових функцій та їх впровадженням у продакшн, знижує ризик помилок, підвищує якість програмного забезпечення та спрощує управління складними розподіленими системами у хмарному середовищі. На рисунку 2.1 наведено етапи CI/CD.

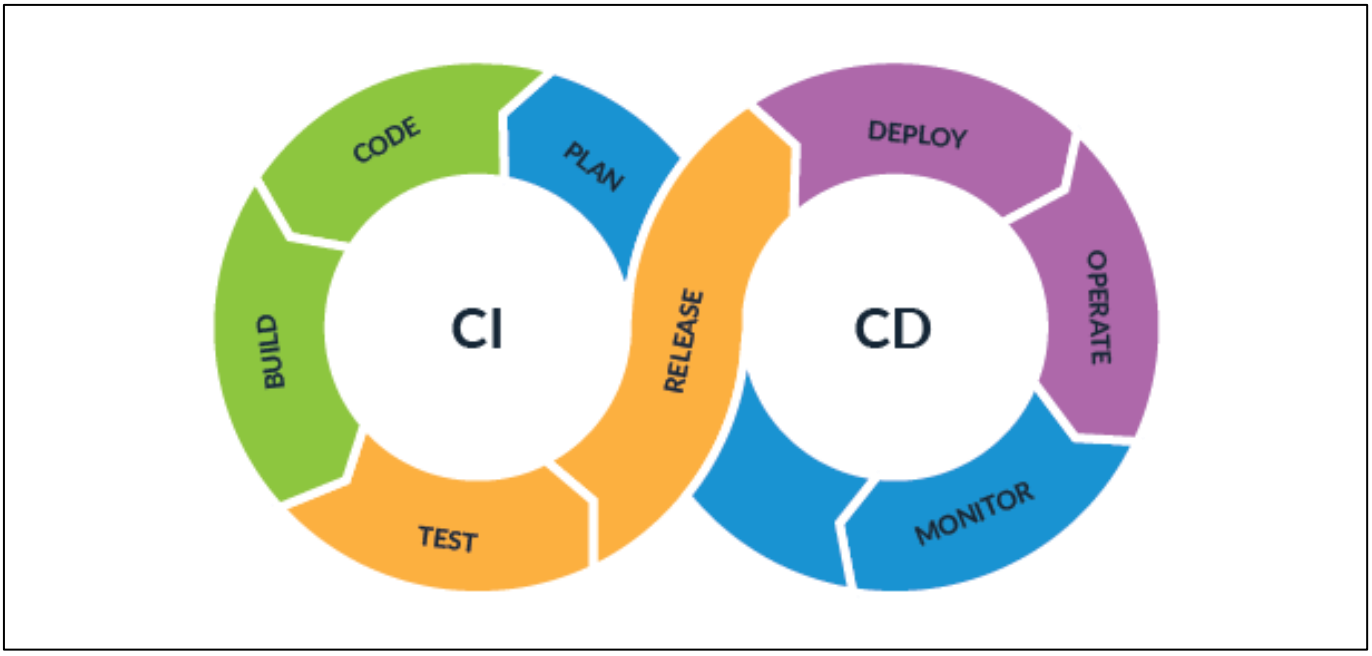

Рисунок 2.1 – Етапи CI/CD

CI - це практика розробки програмного забезпечення, яку члени команди часто інтегрують свою роботу. Вони зазвичай роблять це, об'єднуючи свої зміни коду в загальну основну гілку з великою частотою. Кожна інтеграція перевіряється автоматизованим процесом складання, який запускає тести для виявлення помилок інтеграції якомога швидше. Багато команд використовують сервер CI або хмарний сервіс для автоматизації цього процесу тестування та збирання. Основними цілями безперервної інтеграції є:

- виявлення помилки та проблем з інтеграцією на стаді збору коду. Це робить простішим визначити їх та швидше виправити.
- забезпечення атомарності – усі члени команди працюють над актуальним кодом, а не використовують застарілу версію;
- уникнення ситуацій, коли різні члени команди працювали над одним функціоналом, який не працює в сукупності;
- тестування коду на вразливість, що забезпечує рівень захисту хмарної системи в цілому.

У контексті Microsoft Azure реалізація CI/CD забезпечується за допомогою таких інструментів, як Azure DevOps, GitHub Actions, Azure Pipelines, Azure Kubernetes Service, а також вбудованих засобів моніторингу, тестування та безпеки.

Microsoft Azure Pipelines надає потужні можливості, як от автоматичне оновлення коду. Тестування коду координується автоматично.

На рисунку 2.2. можна побачити процес створення нового проєкту на платформі Azure DevOps.

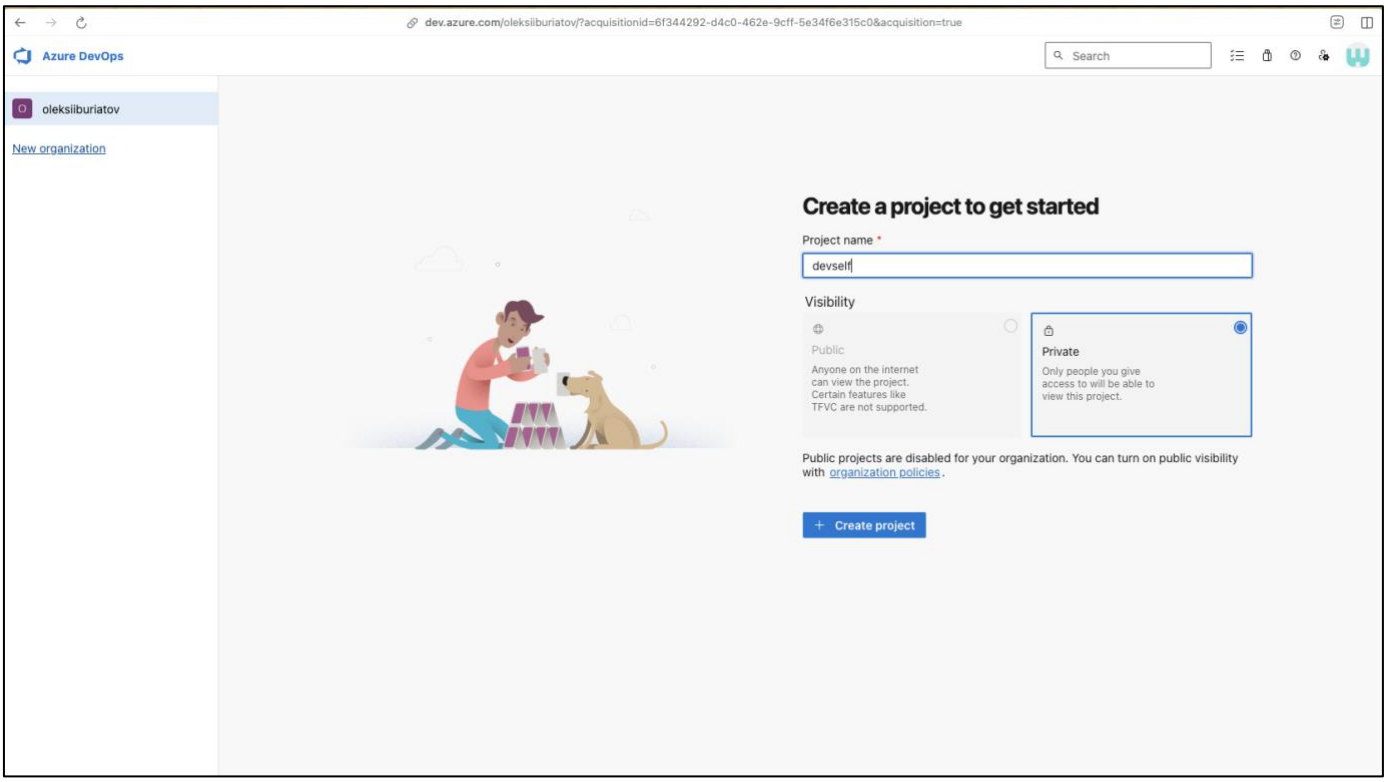

Рисунок 2.2 – Створення нового проєкту в Azure DevOps

Потрібно вказати ім'я та вибрати систему контролю версій (наприклад, Git). Там також можна вибрати методологію ведення проєкту, такі як Agile, Scrum або Basic, залежно від підходу ведення проєктів команди. Після створення з’являється доступ до інструментів для планування роботи, запуску збірок та розгортання додатків, управління вихідним кодом,.

Для того, щоб переглянути доступність цього сервісу та його використання, можна скористатися вкладкою “Usage” (рисунок 2.3):

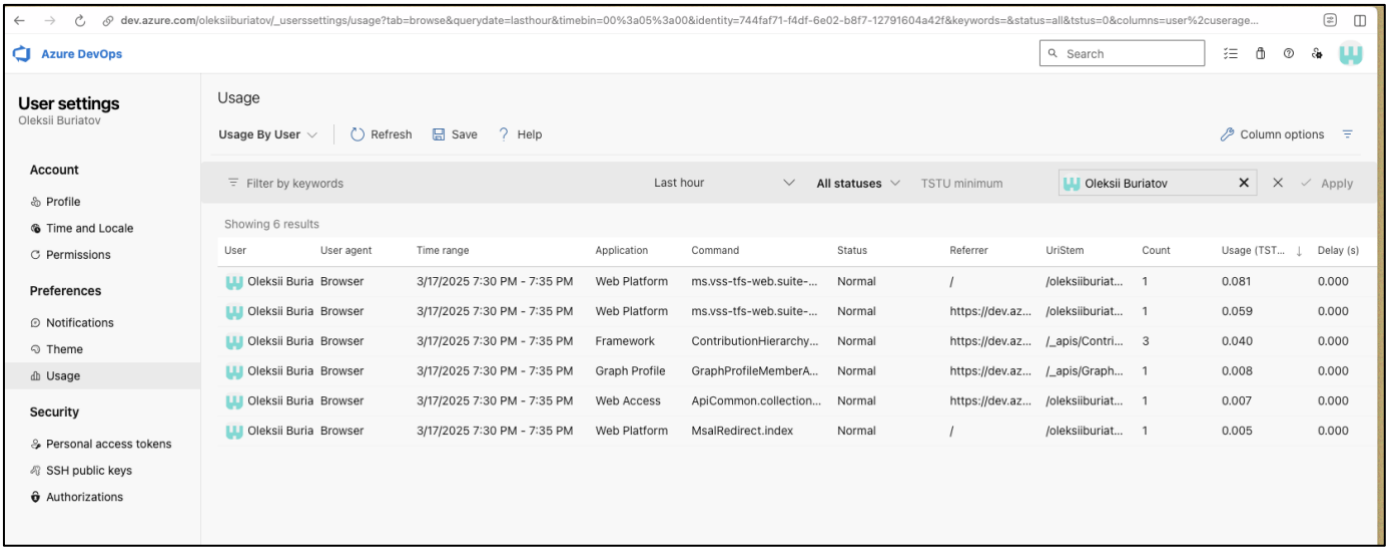

Рисунок 2.3 – Використання та доступність Azure DevOps

2.2Етапи реалізації CI/CD

Реалізація CI/CD-процесу умовно поділяється на кілька послідовних етапів, які можуть бути адаптовані залежно від особливостей системи, архітектури та вимог до безпеки.

2.2.1 Безперервна інтеграція (continious integration)

- Після кожного внесення змін до репозиторію спрацьовує тригер на збірку образа. На цьому етапі перевіряються доступи до ключів автентифікації до приватних репозиторіїв, залежності бібліотек, актуальність версій коду тощо. У разі якщо все було вказано правильно, то збирається образ (найчастіше docker образ);
- Запускаються юніт-тести, лінтинг коду;
- Запускається аналізатор коду. Цей етап важливий для захисту як компоненти інфраструктури, так і сама інфраструктура, включаючи дані, які зберігаються в сховищах;
- У разі успішного проходження перевірок створюється артефакт та зберігається в сховищах образів. Частіше за все такі сховища є приватними.

Найпопулярніші це: DockerHub, GitLab Registry, Nexus, OpenShift Container Registry тощо;

2.2.2 Безперервна доставка (continious delivery)

- Створений артефакт автоматично передається у середовище для подальшого тестування або попереднього перегляду (стейджинг). На даному етапі образ розгортається в BETA середовищі, яке має бути ідентичним до LIVE середовища, проте без реальних даних та реальних користувачів. Відбувається тестування QA інженерами, які дають заключну оцінку щодо нового функціонала. Тестування може бути успішним або неуспішним. Якщо були знайдені помилки та недоліки, то такий функціонал іде на доопрацювання;
- Результати тестування дозволяють прийняти рішення щодо ручного або автоматичного впровадження у продакшн.

2.2.3 Безперервне розгортання (continious deployment)

- Усі успішно протестовані зміни можуть автоматично розгортаються у робоче середовище без додаткового схвалення, або ж додатково перевіряються відповідними фахівцями, які надають результати ручного тестування;
- Застосовуються техніки rolling updates, blue-green deployment, canary releases для зменшення ризику збоїв. У багаторівневому розгортанні всі компоненті в цільовому середовищі одночасно оновлюються кількома новими процесами (рисунок 2.4). Ця стратегія використовується для додатків, які мають сервісні або версійні залежності, або якщо розгортання відбувається в неробочий час. Rolling updates – це стратегія розгортання, яка оновлює запущені екземпляри програми з новою версією. Усі компоненти в цільовому середовищі поступово оновлюються артефактною версією. Blue-green розгортання - це стратегія розгортання, яка використовує два однакових середовища: «синє» (BETA середовище) та «зелене» (LIVE середовище) з різними версіями програми (рисунок 2.5). Забезпечення якості та тестування

користувача, як правило, проводяться в «синьому» середовищі, де розміщуються нові версії або зміни [7]. Користувальницький трафік переміщується з «зеленого» середовища в синє середовище після того, як нові зміни були перевірені і прийняті в «синьому» середовищі. Після успішного розгортання можна переходити до нового середовища.

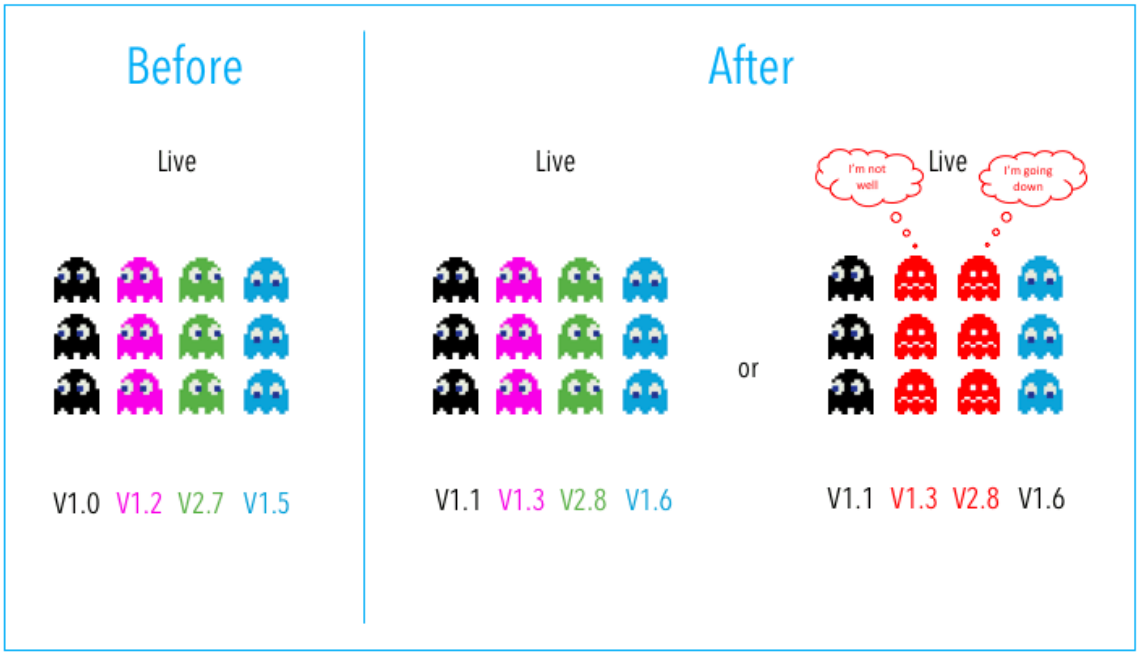

Рисунок 2.4 – Багаторівнева модель розгортання

Рисунок 2.5 – Blue-green модель розгортання

• Кожен із цих етапів супроводжується налаштуванням сповіщень, моніторингом та автоматизованим відкатом змін у разі помилок [8].

2.3 Переваги використання CI/CD у високонавантажених хмарних системах

Використання технологій безперервної інтеграції та доставки (CI/CD) у високонавантажених хмарних системах має суттєві переваги, які охоплюють як технічні аспекти розгортання програмного забезпечення, так і стратегічні цілі щодо стабільності, масштабованості та інформаційної безпеки. У контексті динамічного середовища хмарних інфраструктур, де ресурси та навантаження можуть швидко змінюватися, ефективна автоматизація процесів розробки й впровадження набуває вирішального значення.

2.3.1 Підвищення швидкості оновлення

CI/CD дозволяє впроваджувати нові функціональні можливості, виправлення помилок і оновлення без затримок, пов'язаних із ручним втручанням. Завдяки автоматизації тестування, побудови та розгортання є суттєве скорочення часу між написанням коду та його впровадженням у продакшн. Це особливо важливо у високонавантажених системах.

2.3.2 Забезпечення стабільності та надійності застосунків

Постійне автоматизоване тестування змін гарантує, що в LIVE-середовище потрапляє лише перевірений код. CI/CD дозволяє виявляти помилки, включно з вразливостями, ще на етапі інтеграції, тим самим знижуючи ймовірність критичних збоїв у роботі системи. У хмарних середовищах із великим потоком користувачів це особливо актуально, оскільки навіть короткочасні проблеми можуть призвести до втрати доступності сервісу.

2.3.3 Масштабованість процесів у відповідь на зміну навантаження

CI/CD-процеси інтегруються з інструментами автоматичного масштабування та контейнеризації (наприклад, Azure Kubernetes Service), що дає змогу гнучко адаптувати систему до пікових навантажень. Це забезпечує не лише технічну гнучкість, а й зменшення часу на адаптацію архітектури до нових умов.

2.3.4 Зниження ризику людського фактору

Автоматизація ключових етапів розгортання мінімізує участь людини в рутинних або критичних операціях. Це, своєю чергою, зменшує ризики помилок, пов'язаних із ручним внесенням змін у конфігурацію або інфраструктуру. Для систем, що обслуговують велику кількість користувачів у реальному часі, це є важливим чинником забезпечення безпеки та надійності.

2.3.5 Відповідність політикам безпеки

CI/CD дозволяє впровадити автоматизовану перевірку відповідності внутрішнім стандартам, політикам безпеки та регуляторним вимогам. Це включає в себе сканування коду на наявність вразливостей, перевірку залежностей, контроль за дотриманням стилістичних і структурних стандартів. Особливу увагу приділяють доступ до репозиторіїв співробітників, адже деякі дані можуть бути чутливими. У високонавантажених системах, де навіть незначні порушення можуть мати масштабні наслідки, такий контроль є критично важливим.

2.3.6 Можливість швидкого відновлення після збоїв (low MTTR)

CI/CD забезпечує впровадження технік, які дозволяють швидко відкотити зміни у випадку виявлення помилок після релізу (наприклад, blue-green deployment, canary releases). Завдяки цьому середній час на відновлення (Mean Time to Recovery, MTTR)

значно зменшується, що сприяє високій доступності сервісу навіть у разі непередбачених ситуацій.

2.3.6 Безшовна інтеграція з DevSecOps підходами

Інтеграція безпеки у всі етапи CI/CD-пайплайну (так званий підхід Security as Code) дозволяє забезпечити проактивний контроль за кіберризиками. Сканування вразливостей, перевірка конфігурацій, аналіз доступів і політик здійснюються автоматично, ще до того, як зміни потрапляють у робоче середовище. Це особливо актуально для високонавантажених систем, які є потенційно привабливими об'єктами для атак.

2.4 Використання підходу Infrastructure as Code (IaC) у процесах CI/CD

Одним з ключових елементів реалізації CI/CD у хмарному середовищі є підхід Infrastructure as Code (IaC) — інфраструктура як код. Ця концепція передбачає управління інфраструктурою (сервери, мережі, сховища, сервіси тощо) через декларативні або скриптові конфігураційні файли, аналогічно до програмного коду. Замість ручного налаштування ресурсів через інтерфейс користувача, IaC дає змогу описувати всі компоненти інфраструктури у вигляді текстових інструкцій, що дозволяє автоматизувати процес розгортання, масштабування та оновлення середовища.

2.4.1 Основні принципи IaC

- Автоматизація — створення, оновлення та знищення інфраструктури відбувається автоматизовано, що усуває людський фактор і помилки конфігурації;
- Версійність — всі зміни інфраструктури відстежуються у системах контролю версій (Git), що забезпечує історію змін і можливість відкату;

- Узгодженість — інфраструктура в усіх середовищах (dev/test/prod) будується з одних і тих самих шаблонів, що гарантує однакову поведінку;

У середовищі Microsoft Azure доступні різноманітні інструменти для реалізації принципу IaC:

- Azure Resource Manager (ARM) Templates — JSON-шаблони для декларативного опису інфраструктури в Azure. Дають змогу автоматично створювати ресурси з чітким визначенням параметрів (рисунок 2.6);

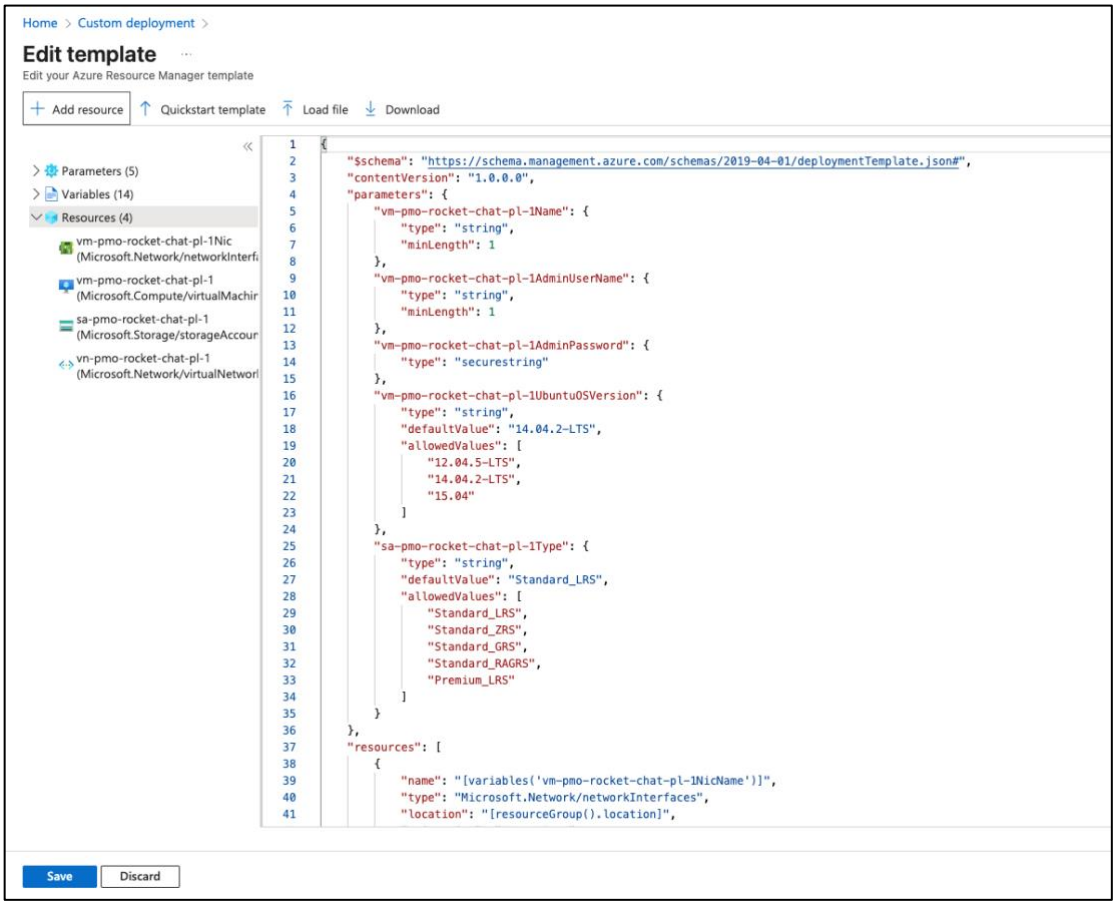

Рисунок 2.6 – Створення ARM-шаблону для розгортання віртуальної машини в Microsoft Azure

- Bicep — нова декларативна мова, розроблена Microsoft як спрощення для ARM-шаблонів. Має компактний синтаксис, інтеграцію з Visual Studio Code і підтримує всі можливості ARM (рисунок 2.7);

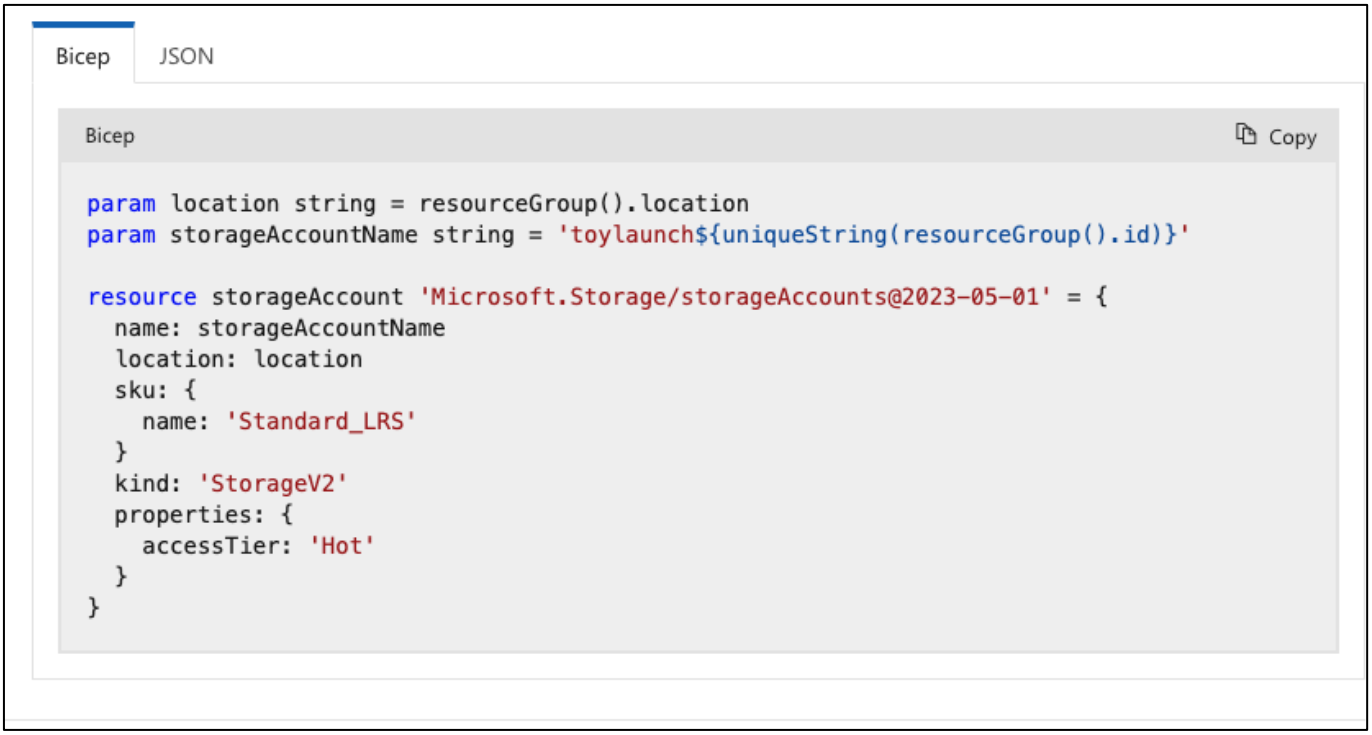

Рисунок 2.7 – Створення storageAccount за допомогою bicep у Microsoft Azure

- Terraform — кросплатформений IaC-інструмент із підтримкою Azure (рисунок 2.8). Він дозволяє описувати інфраструктуру мовою HCL (HashiCorp Configuration Language) і керувати нею з використанням однієї кодової бази для кількох середовищ.

Він розроблений таким чином, щоб бути читабельним як для людини, так і для машини, що дозволяє легко описувати та читати конфігураційні файли. HCL найчастіше використовується в таких інструментах, як Terraform, де він дозволяє описувати хмарні ресурси, залежності та конфігурації об’єктів структурованим і декларативним способом (див. додаток В). Його синтаксис схожий на JSON, але з більшою гнучкістю та читабельністю.

Terragrunt – це обгортка для Terraform, яка допомагає керувати та спрощувати конфігурації у великих або складних налаштуваннях інфраструктури Terraform. Він наділяє додатковими функціональними можливості, такі як: збереження конфігурацій DRY (Don't Repeat Yourself), управління станом *.tfstate* та обробкою залежностей між модулями.

```
resource "azurerm_resource_group" "example" {
  name     = "example-resources"
  location = "West Europe"
}

resource "azurerm_network_security_group" "example" {
  name                = "acceptanceTestSecurityGroup1"
  location            = azurerm_resource_group.example.location
  resource_group_name = azurerm_resource_group.example.name
}

resource "azurerm_network_security_rule" "example" {
  name                        = "test123"
  priority                    = 100
  direction                   = "Outbound"
  access                      = "Allow"
  protocol                    = "Tcp"
  source_port_range           = "*"
  destination_port_range      = "*"
  source_address_prefix       = "*"
  destination_address_prefix  = "*"
  resource_group_name         = azurerm_resource_group.example.name
  network_security_group_name = azurerm_network_security_group.example.name
}
```

Рисунок 2.8. – Створення security group за допомогою Terraform у Microsoft Azure

- Ansible —це інструмент автоматизації з відкритим вихідним кодом, який використовується для управління конфігурацією, розгортання додатків та автоматизації завдань [9]. Він використовує просту, зрозумілу людині мову YAML (YAML Ain't Markup Language) для визначення завдань автоматизації у файлах – playbooks (плейбуках). Ansible достатньо доступу до SSH, щоб розпочати працювати з сервером або віртуальною машиною. Ansible також може бути як автоматизатор конфігурацій, який також може використовуватись у зв’язці з Azure для налаштування середовищ після створення ресурсів (рисунок 2.9);

```yaml
- name: Create a security group
  azure_rm_securitygroup:
    resource_group: myResourceGroup
    name: mysecgroup
    purge_rules: true
    rules:
      - name: DenySSH
        protocol: Tcp
        destination_port_range: 22
        access: Deny
        priority: 100
        direction: Inbound
      - name: 'AllowSSH'
        protocol: Tcp
        source_address_prefix:
          - '174.109.158.0/24'
          - '174.109.159.0/24'
        destination_port_range: 22
        access: Allow
        priority: 101
        direction: Inbound
      - name: 'AllowMultiplePorts'
        protocol: Tcp
        source_address_prefix:
          - '174.109.158.0/24'
          - '174.109.159.0/24'
        destination_port_range:
          - 80
          - 443
        access: Allow
        priority: 102

- name: Update rules on existing security group
  azure_rm_securitygroup:
    resource_group: myResourceGroup
    name: mysecgroup
    rules:
      - name: DenySSH
        protocol: Tcp
        destination_port_range: 22-23
        access: Deny
        priority: 100
        direction: Inbound
      - name: AllowSSHFromHome
        protocol: Tcp
        source_address_prefix: '174.109.158.0/24'
        destination_port_range: 22-23
        access: Allow
        priority: 102
        direction: Inbound
    tags:
      testing: testing
      delete: on-exit
```

Рисунок 2.9. – Створення security group за допомогою Ansible у Microsoft Azure

2.4.2 Переваги використання Infrastructure as Code у високонавантажених хмарних системах

Перехід до автоматизації аварійного відновлення дає організаціям можливість перейти від реактивного відновлення до проактивної стійкості. ControlMonkey запустив рішення Automated Disaster Recovery для відновлення всієї хмарної інфраструктури на відміну від даних [10]. Автоматизація істотно скорочує час

відновлення - аж на 90% у деяких сценаріях - тим самим мінімізуючи час простою бізнесу та збої в роботі.

Практично кажучи, якщо значні частини хмарної інфраструктури не захоплені в IaC, будь-яке видалення або втрата ресурсів може призвести до великих зусиль з ручного відновлення. Автоматизація дозволяє відновити за лічені хвилини замість годин, значно зменшуючи час простою та зменшуючи тиск від замовників та клієнтів послуг.

- Швидке масштабування — автоматичне створення додаткових віртуальних машин або сервісів за заздалегідь описаними шаблонами;
- Відновлення після збоїв — у випадку втрати конфігурацій або помилок система може бути відновлена в мінімальні строки за допомогою існуючих IaC-скриптів [10];
- Безперервне тестування інфраструктури — шаблони можна перевіряти на помилки і відповідність політикам безпеки ще до застосування змін;
- Уніфікація середовищ — однакові шаблони використовуються для різних етапів життєвого циклу (розробка, тестування, продакшн), що мінімізує різницю в поведінці застосунку.
- Статичний аналіз – ізольований аналіз коду, виявляючи ризики, неправильні конфігурації та помилки відповідності, що стосуються лише самого IaC. Такі інструменти, як kubescan, Snyk, Coverity тощо, можуть бути використані для статичного аналізу IaC.
- Перевірка залежностей з відкритим кодом - аналізу залежностей з відкритим кодом, такі як пакети ОС, бібліотеки тощо для виявлення потенційних ризиків. Такі інструменти, як BlackDuck, Snyk, WhiteSource Bolt для GitHub та подібні можуть бути використані для аналізу залежностей з відкритим кодом IaC.

Висновки до другого розділу

У цьому розділі було розглянуто взаємодію з хмарною платформою через призму безперервної інтеграції (CI процеси) та безперервної доставки (CD процеси).

Аналіз проводився з метою покращення розгортання, автоматизації, а також додатковому захисті інфраструктури.

Цей розділ спрямований на зміну підходів у вигляді «ручного» керування до підходу автоматичного. Використовуючи Terraform, досягається узгоджений стан, який може легко відтворитися, адже IaC – інфраструктура як код – позбавляє будь-яких ручних маніпуляцій, що унеможливлює помилку зі сторони виконавця.

РОЗДІЛ 3
РОЗГОРТАННЯ БАГАТОРІВНЕВОЇ МОДЕЛІ ЗАХИСТУ ІНФОРМАЦІЙНОЇ СИСТЕМИ В MICROSOFT AZURE

Для розгортання моделі захисту інформаційної системи в Microsoft Azure було взято за основу мікросервісну архітектуру з вуличного освітлення.

3.1 Архітектура інформаційної системи

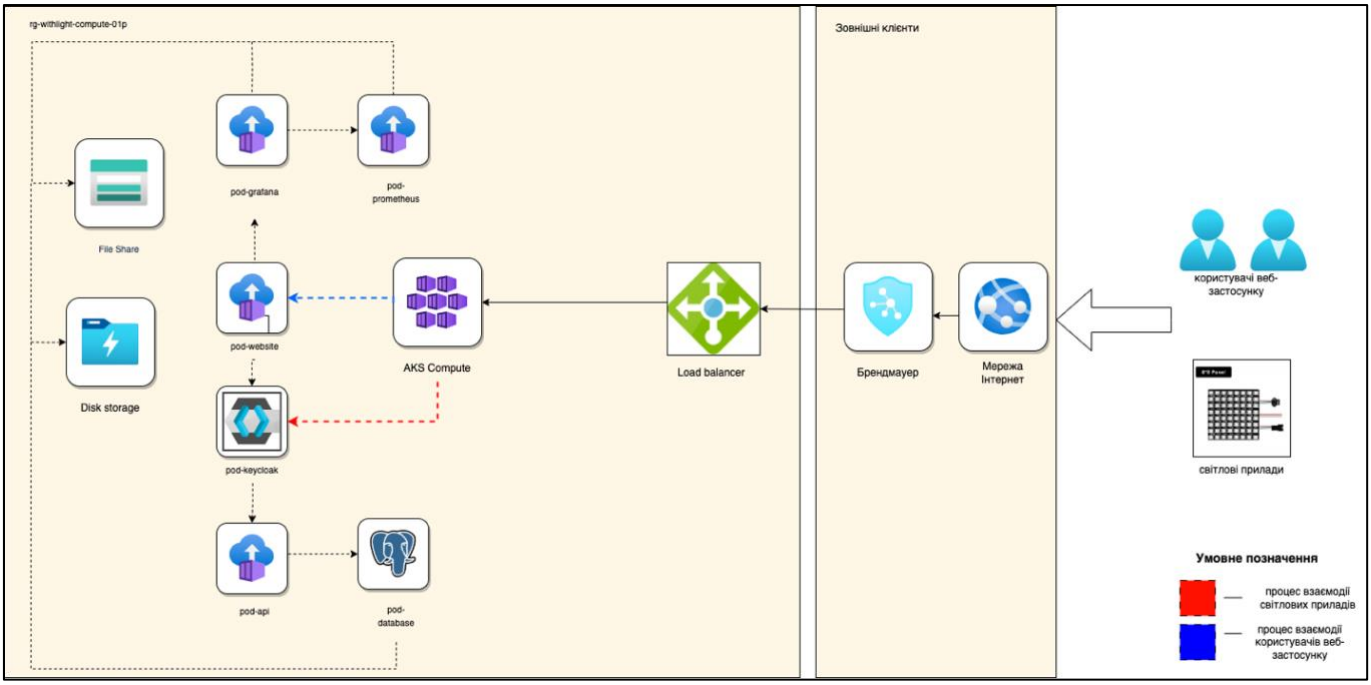

Рисунок 3.1 – Архітектура інформаційної системи в Microsoft Azure

Дана архітектура складається з наступних компонентів (рисунок 3.1):

- Azure Kubernetes Service
- Azure Files
- Azure Disk
- Load Balancer
- Firewall
- Database

Кожен із цих компонентів є PaaS рішенням, що дозволяє гарантувати відмовостійкість системи. SLA складає 99,9(9)% від усього часу [11].

3.2 Розгортання оточення

Реалізація високонавантаженої системи в хмарі полягала в тому, щоб забезпечити максимальний рівень захисту на кожному етапі розгортання.

3.1.1 Доступ до хмари

Робота з хмарним середовищем розпочинається зі створення користувача (рисунок 3.2).

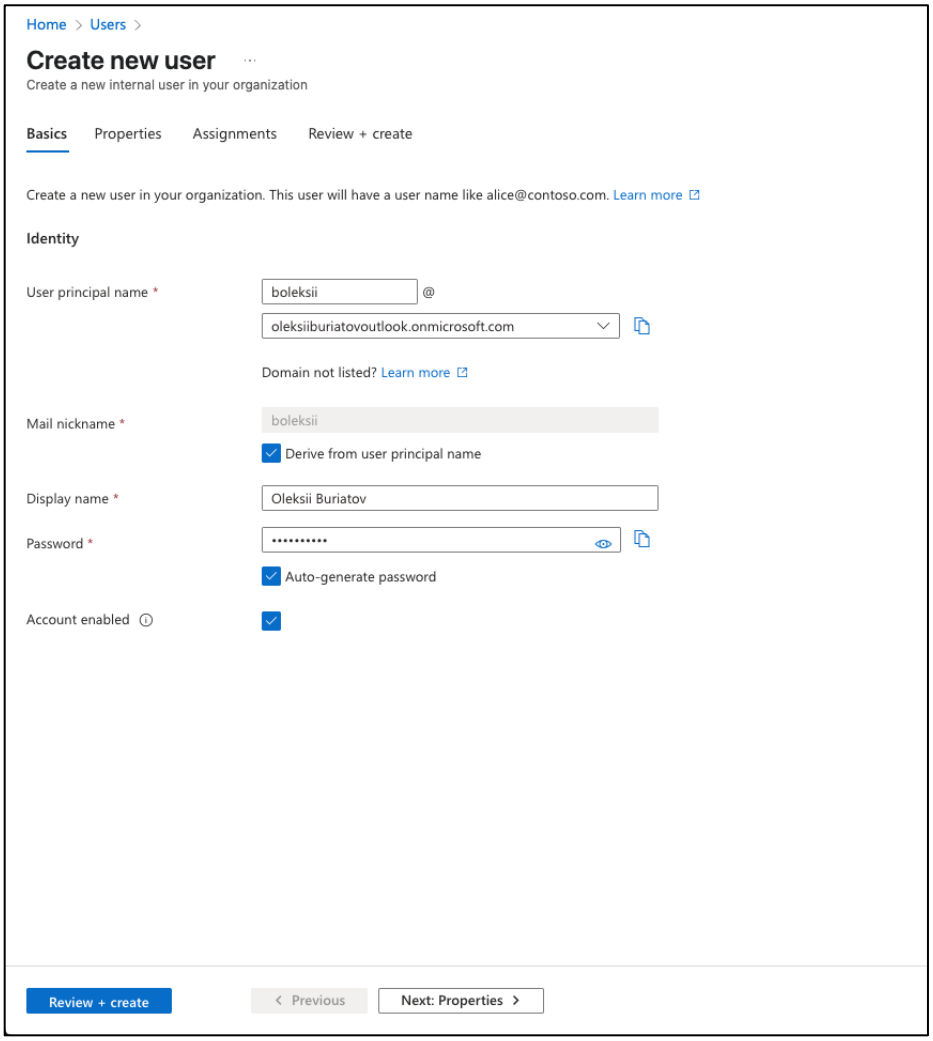

Рисунок 3.2 – Створення користувача в Microsoft Azure

На цьому кроці користувачу додаються ролі, які визначатимуть права та доступи до тих чи інших ресурсів (рисунок 3.3).

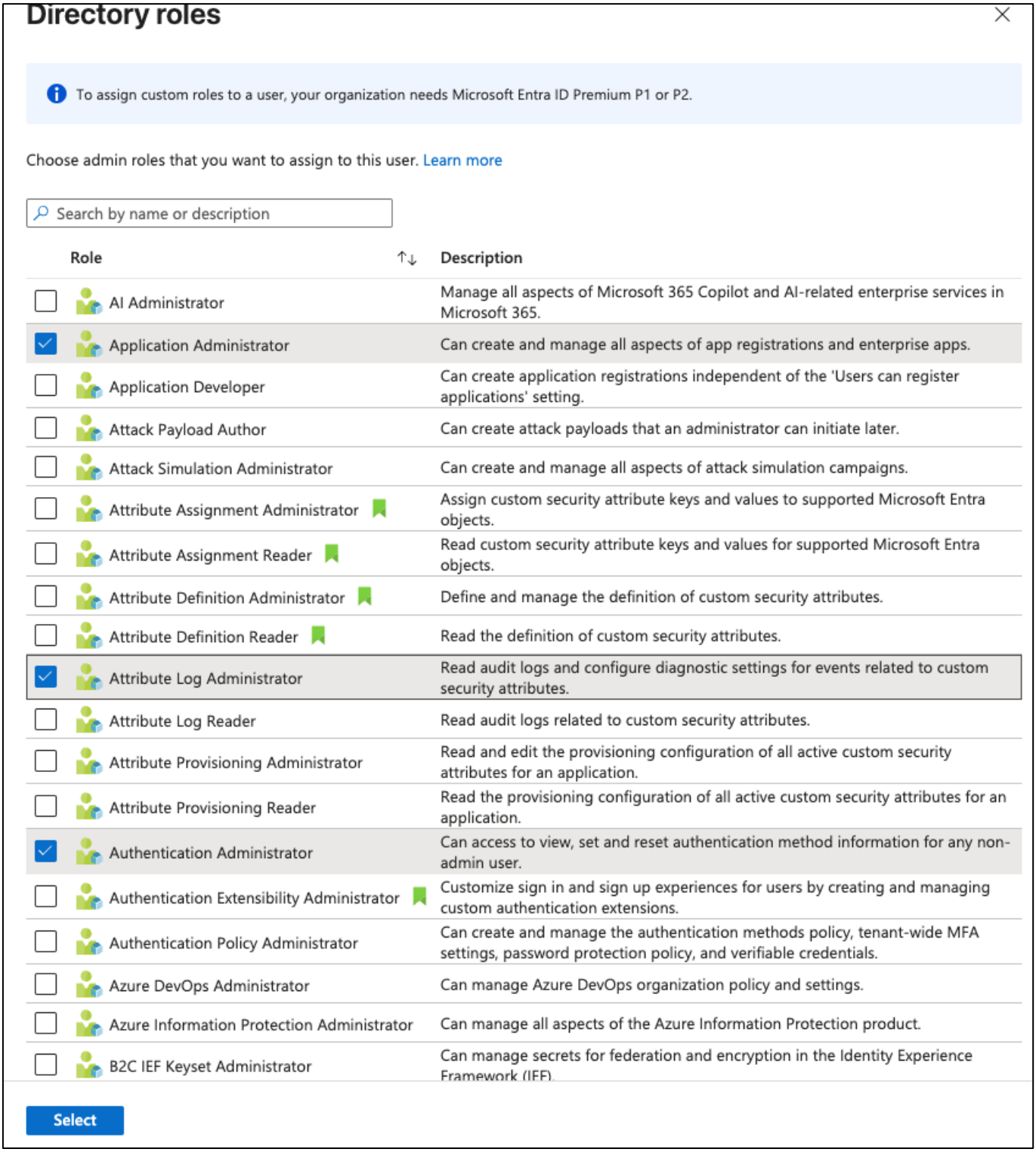

Рисунок 3.3 – Видача ролей користувачу для виконання задач

Надання правильних ролей доступу є першим рівнем захисту інформаційної системи в хмарі. Це мінімізує фактор несанкціонованого доступу до ресурсів, які користувач не має права переглядати, додавати або редагувати згідно з посадовими обов'язками.

3.1.2 Опис інфраструктури за допомогою IaC

Наступним кроком для забезпечення цілісності оточенні необхідно підготувати описані конфігураційні файли – ресурси, які будуть розгорнуті, використовуючи Terraform. На рисунку 3.4 зображено фрагмент коду по розгортанню Azure Kubernetes Service.

```
2   resource "tls_private_key" "aks_ssh" {
3     algorithm = "RSA"
4   }
5
6
7   resource "azurerm_kubernetes_cluster" "aks" {
8     name                            = "aks-${var.project}-compute-${substr(var.location, 0, 6)}-${var.suffix}${var.enviroment}"
9     location                        = azurerm_resource_group.rg.location
10    resource_group_name             = azurerm_resource_group.rg.name
11    dns_prefix                      = "aks${var.project}${substr(var.location, 0, 6)}${var.suffix}${var.enviroment}"
12    private_cluster_enabled         = false
13    public_network_access_enabled = true
14    node_resource_group             = "rg-${var.project}-compute-aks-${substr(var.location, 0, 6)}-${var.suffix}${var.enviroment}"
15
16    key_vault_secrets_provider {
17      secret_rotation_enabled = false
18    }
19
20    default_node_pool {
21      name = "default"
22      node_labels = {
23        label = "main"
24      }
25      node_count            = 2
26      max_pods              = 110
27      vm_size               = "Standard_D2_v5" # 4 core, 8 RAM, OS temp storage
28      enable_auto_scaling   = false
29      enable_node_public_ip = true
30    }
31
32    network_profile {
33      load_balancer_sku  = "standard"
34      network_plugin     = "azure"
35      network_policy     = "azure"
36      service_cidr   = "172.19.0.0/16"
37      dns_service_ip = "172.19.0.10"
38      load_balancer_profile {
39        outbound_ip_address_ids = [azurerm_public_ip.outbound_pip_aks.id]
40      }
41    }
```

Рисунок 3.4 – Фрагмент коду для розгортання Azure Kubernetes Service

Також необхідно налаштувати балансувальник даних, який буде приймати на вхід та вихід трафік (рисунок 3.5).

```
resource "azurerm_public_ip" "outbound_pip_aks" {
  name                = "pip-outbound-aks-${var.project}-compute-${substr(var.location, 0, 6)}-${var.suffix}${var.enviroment}"
  resource_group_name = azurerm_resource_group.rg.name
  location            = azurerm_resource_group.rg.location
  allocation_method   = "Static"
  sku                 = "Standard"

  tags = {
    Enviroment = var.enviroment
    Project    = var.project
    Location   = var.location
    Subproject = var.subproject
  }
}

resource "azurerm_public_ip" "load_balancer_pip_aks" {
  name                = "pip-load-balancer-aks-${var.project}-compute-${substr(var.location, 0, 6)}-${var.suffix}${var.enviroment}"
  resource_group_name = "rg-${var.project}-compute-aks-${substr(var.location, 0, 6)}-${var.suffix}${var.enviroment}"
  location            = azurerm_resource_group.rg.location
  allocation_method   = "Static"
  sku                 = "Standard"

  tags = {
    Enviroment = var.enviroment
    Project    = var.project
    Location   = var.location
    Subproject = var.subproject
  }

  depends_on = [
    azurerm_kubernetes_cluster.aks
  ]
}

output "outbound_pip_aks" {
  value = azurerm_public_ip.outbound_pip_aks.*.ip_address
}

output "load_balancer_pip_aks" {
  value = azurerm_public_ip.load_balancer_pip_aks.*.ip_address
}
```

Рисунок 3.5 – Опис балансувальника даних у Microsoft Azure

Після того, як конфігураційні файли були підготовлені, необхідно активувати оточення за допомогою команди *terraform init* . Одразу після потрібно виконати *terraform apply* – ця команда потрібна для розгортання інфраструктури – підтвердження своїх намірів, ввівши в командному рядку *yes*. На порталі https://portal.azure.com можна переглянути чи з’явилася ресурсна група з необхідними компонентами. На рисунку 3.6 можна побачити групу з назвою «devself».

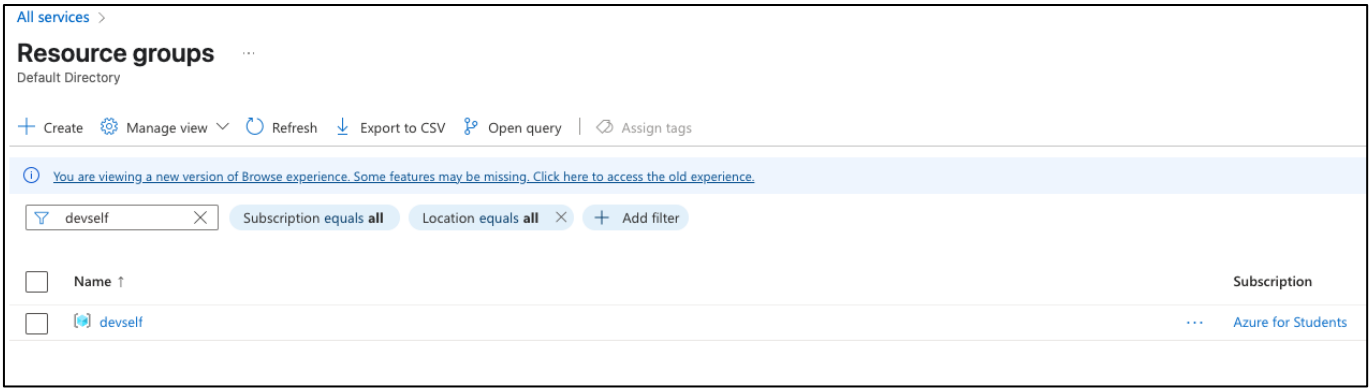

Рисунок 3.6 – Створені ресурси в Microsoft Azure

3.1.3 Налаштування мережевих груп безпеки

Azure Kubernetes Service або Virtual Machine маю бути доступні з мережі Інтернет, так і в мережу Інтернет. Load balancer, який був розгорнутий з прикладу вище, отримує IP, за яким буде здійснюватися доступ до додатків із Azure Kubernetes Service. На даному етапі необхідно налаштувати мережеві групи, в яких зазначається перелік портів, до/або з яких можна підключатися до об'єктів інфраструктури (рисунок 3.7).

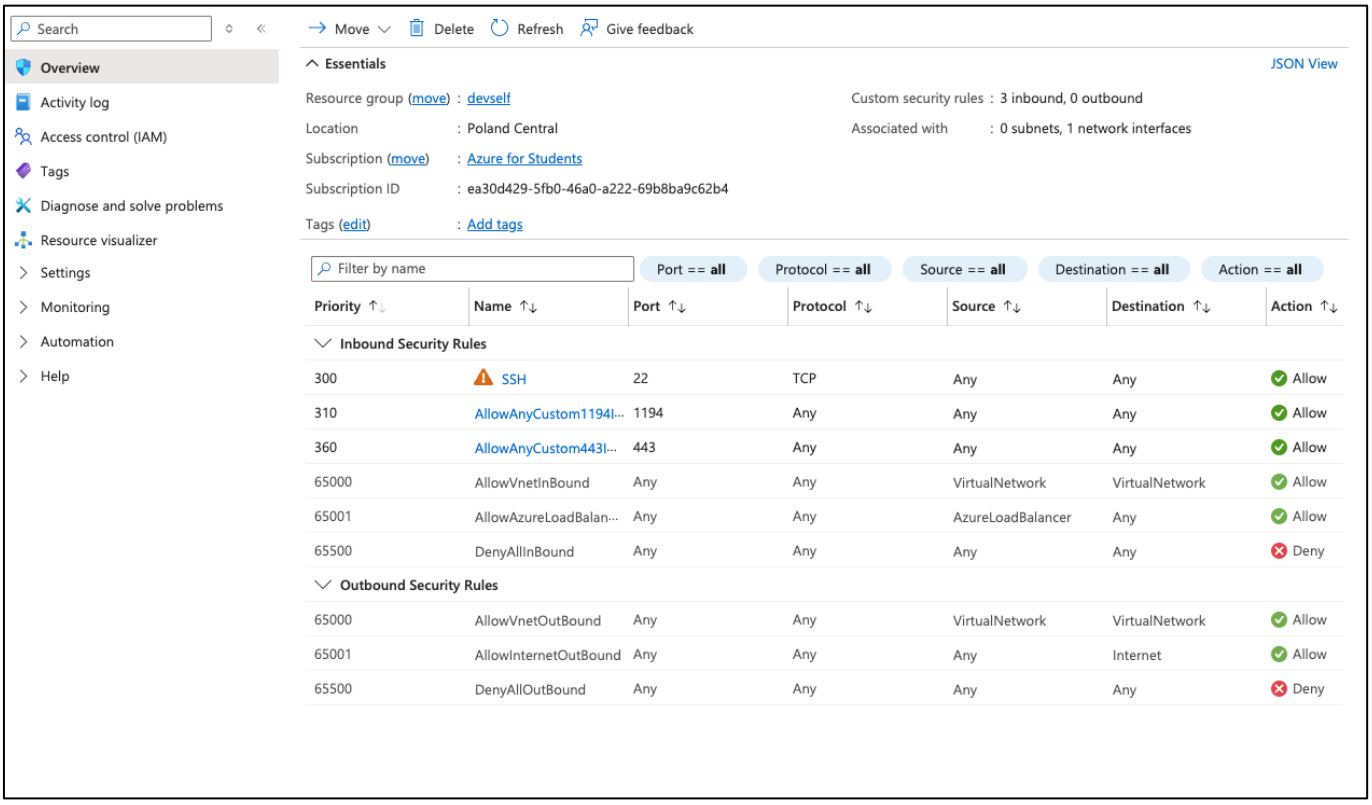

Рисунок 3.7 – Налаштування мережевих груп безпеки в Microsoft Azure

Даний рисунок показує, що доступ до ресурсів, які мають в собі цю мережеву групу можуть підключатися по портах 22, 443 та 1194, а вихідний трафік для самих віртуальних машин можливий будь-куди. Необхідно дотримуватися підходу видачі доступів від найменшого – тобто, не варто одразу відкривати весь перелік IP адрес, а потрібно проаналізувати які саме потрібні й ті відкривати в світ.

3.1.4 Налаштування безперервної інтеграції та доставки коду

Так як оточення готове до розгортання коду, необхідно налаштувати CI/CD пайплайни. У даному прикладі використовується GitLab як інструмент CI/CD (також можна було обрати Azure DevOps, але для цього прикладу економічно доцільніше було обрати саме платформу GitLab).

На рисунку 3.8 зображено пайплайни, які успішно зібрали код, протестували та розгорнули в Kubernetes

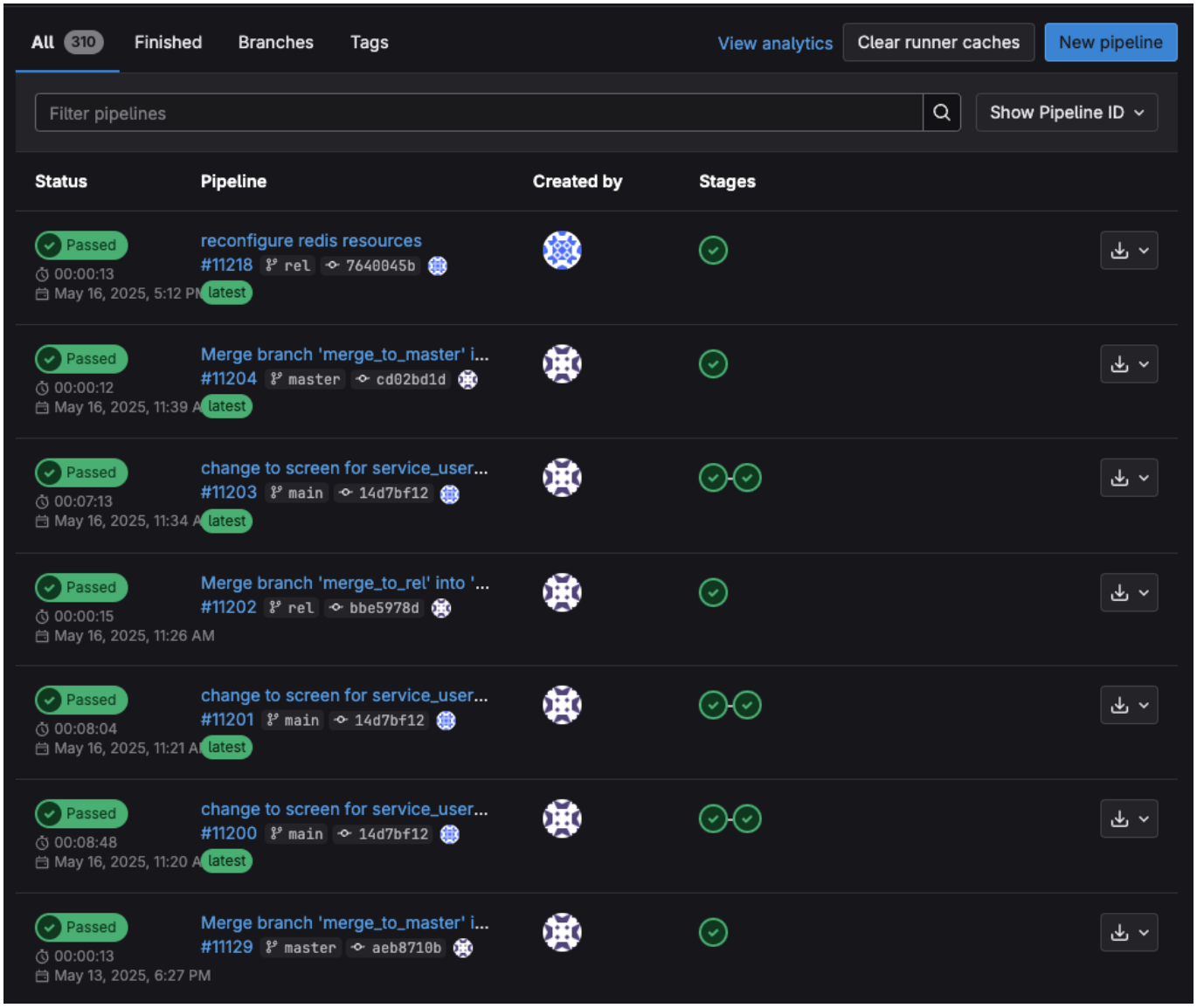

Рисунок 3.8 – CI/CD пайплайни, які успішно зібрали код

Для цієї інтеграції було описано .gitlabci.yml конфіг (рисунок 3.9). Він:

- збирає образ із кодом;
- тестує код, використовуючи Unit тести;
- об’єднує гілки dev із prod, щоб зміни потрапили в Kubernetes (додаток Д).

```
service_user_search_light_build:
  stage: build
  image: gitlab.azure.net:5000/okd/conf/gitrunner-s2i-podman-push:1.1
  allow_failure: false
  rules:
    - if: '($CI_PIPELINE_SOURCE == "trigger" || $CI_PIPELINE_SOURCE == "web") && ($PROJECT_ENV=="rel" || $PROJECT_ENV=="master") && ($REVERT_DEPLOY == "no") &&
    $BUILD_USER_SEARCH_LIGHT == "yes" && $CI_COMMIT_REF_NAME == "main"'
  id_tokens:
    VAULT_ID_TOKEN:
      aud: https://gitlab.azure.net
  secrets:
    SSH_KEY:
      vault: okd/to/${VAULT_AUTH_ROLE}/ssh-bitbucket/ssh-key.priv@admins
      file: false
      token: $VAULT_ID_TOKEN
  before_script:
      - echo "${SSH_KEY}" > ~/.ssh/id_rsa
      - echo "${SSH_CONFIG}" > ~/.ssh/config
      - chmod 600 ~/.ssh/id_rsa ~/.ssh/config
      - podman login ${PRIVATE_REGISTRY_URL} -u ${CI_REGISTRY_USER} -p ${CI_REGISTRY_PASSWORD}
      - podman pull ${S2I_SOURCE_IMAGE_PYTHON}:${S2I_SOURCE_TAG_PYTHON} --authfile ${S2I_SOURCE_IMAGE_AUTH_FILE}
  script:
    - s2i build --ref=${PROJECT_ENV} ${PROJECT_URL} --context-dir=${CONTEXT_DIR}/${SERVICE_USER_SEARCH_LIGHT} ${S2I_SOURCE_IMAGE_PYTHON} --as-dockerfile Dockerfile $
    {SERVICE_USER_SEARCH_LIGHT}
    - echo $CRM_PIPSERVER_NETRC > netrc
    - cp -r ${SERVICE_USER_SEARCH_LIGHT_DOCKERFILE_PATH} Dockerfile
    - podman build --platform linux/amd64 --build-arg PIP_INDEX_URL=$PIP_INDEX_URL --build-arg MICROPIPENV_DEFAULT_INDEX_URLS=$PIP_INDEX_URL  -t ${SERVICE_USER_SEARCH_LIGHT}_$
    {PROJECT_ENV}:pipeline-${CI_JOB_ID} -f Dockerfile
    - podman run -it --rm ${SERVICE_USER_SEARCH_LIGHT}_${PROJECT_ENV}:pipeline-${CI_JOB_ID} /bin/bash -c "python3.9 -m unittest /opt/app-root/src/${SERVICE_USER_SEARCH_LIGHT}/tests/
    test_runner.py"; echo $?
    - podman tag ${SERVICE_USER_SEARCH_LIGHT}_${PROJECT_ENV}:pipeline-${CI_JOB_ID} ${PRIVATE_REGISTRY_URL}/${TARGET_REGISTRY_PATH}/${SERVICE_USER_SEARCH_LIGHT}_${PROJECT_ENV}
    :pipeline-${CI_JOB_ID}
    - podman push --authfile /run/containers/0/auth.json ${PRIVATE_REGISTRY_URL}/${TARGET_REGISTRY_PATH}/${SERVICE_USER_SEARCH_LIGHT}_${PROJECT_ENV}:pipeline-${CI_JOB_ID}
    - echo "SERVICE_USER_SEARCH_LIGHT_REPOSITORY=${PRIVATE_REGISTRY_URL}/${TARGET_REGISTRY_PATH}/${SERVICE_USER_SEARCH_LIGHT}_${PROJECT_ENV}" >> build.env
    - echo "BASE_IMAGE_TAG=pipeline-${CI_JOB_ID}" >> build.env
  artifacts:
    reports:
      dotenv: build.env

service_user_search_light_merge_request:
  stage: merge_request
  image: gitlab.ropot.net:5000/okd/conf/gitpusher:1.2
  allow_failure: false
  when: on_success
  rules:
    - if: '($CI_PIPELINE_SOURCE == "trigger" || $CI_PIPELINE_SOURCE == "web") && ($PROJECT_ENV=="rel" || $PROJECT_ENV=="master") && ($REVERT_DEPLOY == "no") &&
    $BUILD_USER_SEARCH_LIGHT == "yes" && $CI_COMMIT_REF_NAME == "main"'
```

Рисунок 3.9 –.gitlabci.yml конфіг для збірки образу

На рисунку 3.10 можна побачити поди, які нещодавно оновилися.

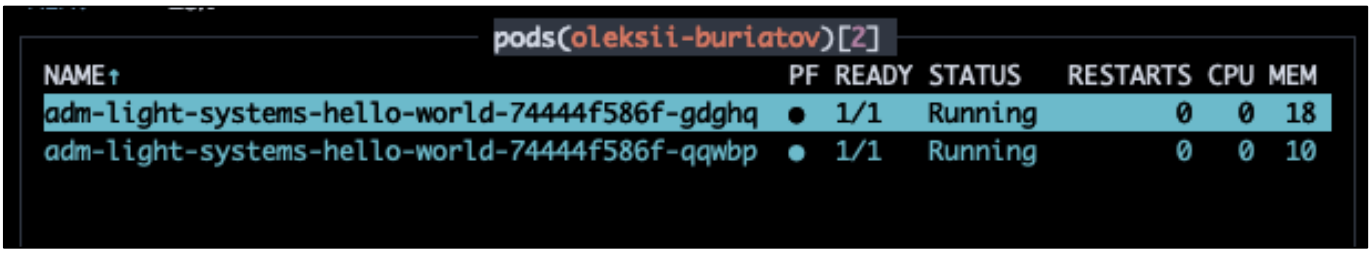

Рисунок 3.10 – Поди Kubernetes

3.3 Збір логів

Основна мета DevOps інженерів, системних адміністраторів та спеціалістів із кібербезпеки полягає в тому, щоб усе працювало. У разі якщо, щось не працює, необхідно аналізувати від середовища розробки до логів усередині розгорнутих сервісів. Для цього варто звернути увагу Graylog. Graylog - це потужна платформа для управління інформацією та подіями безпеки (SIEM) та аналітика журналів, яка централізує, захищає та контролює дані, створені машинами, за допомогою різних джерел. Незалежно від того, чи використовується для кібербезпеки, IT-операцій або відповідності, Graylog надає командам дієві ідеї за допомогою швидкого пошуку, оповіщення та можливостей візуалізації [13].

3.3.1 Логування на рівні Kubernetes (DaemonSet)

Логування на рівні Kubernetes дозволяє централізовано збирати, обробляти, фільтрувати та передавати в сховище. Це дозволяє тримати єдину логіку збору та обробку логів, їх пошук та логічно групувати. На рисунку 3.11 зображено приклад централізованого збору логів із оточення.

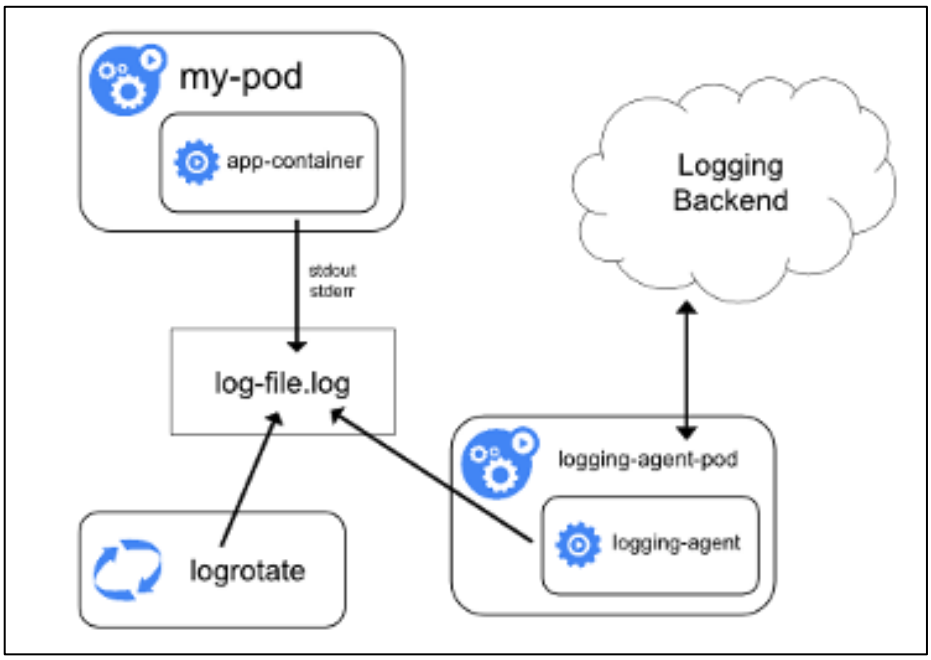

Рисунок 3.11 – Централізований збір логів

Логування на рівні поди (додатковий контейнер у кожному сервісі) – інший варіант збору логів із різних сервісів. Цей варіант надає можливість обирати конкретний агент по збору логів, який потрібен під специфічні задачі . На рисунку 3.12 зображено приклад логування на рівні поди.

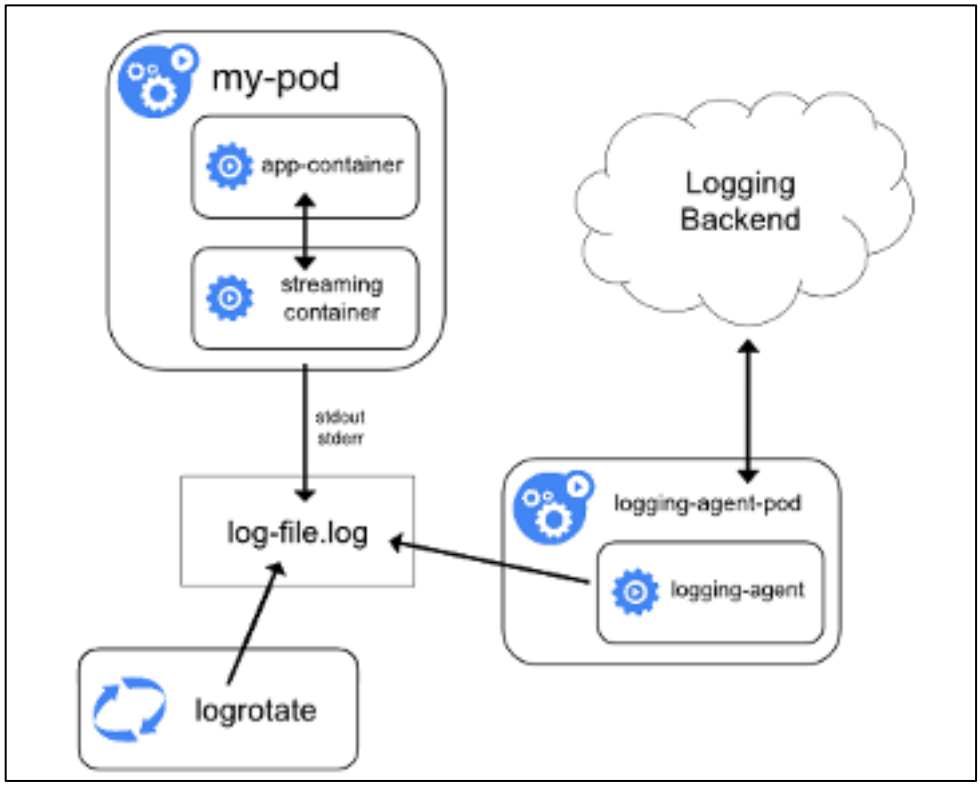

Рисунок 3.12 – Логування на рівні поди

Логування на рівні додатку – це третій можливий варіант по збору логів. У даному випадку вся логіка по зборку, обробці, фільтруванню та передача в сховище перекладається на розробників. Вони проєктують все на боці коду та несуть за це відповідальність.

Цей підхід є виправданим, коли потрібна специфічна логіка збору та взаємодії з логами. На рисунку 3.13 зображено приклад централізованого збору логів на рівні додатку.

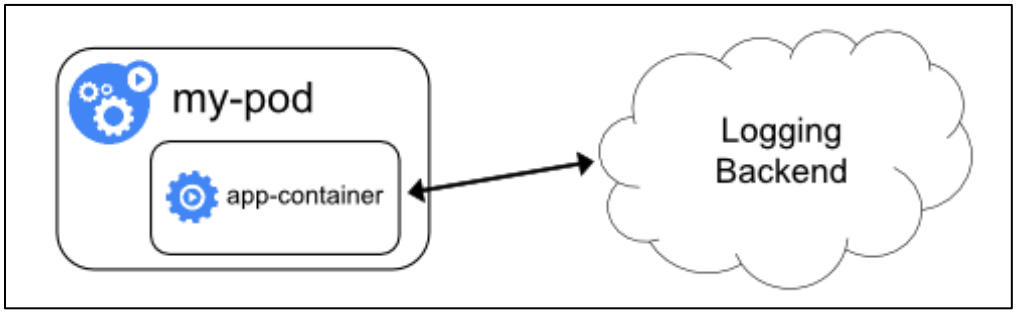

Рисунок 3.13 – Логування на рівні додатку

Для того, щоб логи почали збиратися, необхідно встановити агенти в Azure Kubernetes Service.

Fluentd – це агент-колектор зі збору логів із відкритим вихідним кодом, призначений для централізованої обробки даних у різних системах, що дозволяє агрегувати, трансформувати та пересилати логи в різні місця призначення. Він відслідковує всі лог-файли, збираючи журнали з різних джерел, таких як: програми, сервери та хмарні служби, а потім обробляючи їх через серію фільтрів, перш ніж відправляти їх на системи зберігання або аналізу, такі як Elasticsearch, Kafka, Graylog тощо. Архітектура Fluentd дозволяє користувачам налаштовувати його за допомогою плагінів для введення, виведення та обробки даних [12].

Використання Fluentd зосереджено навколо централізованої агрегації логів (додаток Б). Він широко використовується в управлінні журналами для збору з декількох джерел, таких як контейнери, сервери та мікросервіси, і пересилає їх до централізованої системи реєстрації для аналізу та усунення недоліків. Fluentd часто використовується в хмарних середовищах для обробки журналів в Kubernetes або контейнерних додатках, гарантуючи, що журнали ефективно збираються та маршрутизуються незалежно від динамічної інфраструктури.

У реальному часі Fluentd використовується для обробки даних журналу та подій. Fluentd також корисний для моніторингу безпеки, агрегування та пересилання журналів, пов'язаних із безпекою. Крім того, гнучкість Fluentd робить його придатним для побудови ETL (Extract, Transform, Load) процесів, де дані обробляються і передаються в різні вихідні системи, що дозволяє організаціям оптимізувати свої робочі процеси обробки даних. Його масштабованість, простота використання і

велика екосистема плагінів роблять універсальним інструментом для управління логами.

Встановлення агентів відбувається за допомогою DaemonSet об’єкта Kubernetes. На рисунку 3.14 зображено 18 реплік fluentd, по кожній на ноду кластера Kubernetes. Цей об’єкт встановлюється на всіх нодах AKS та починає вичитувати в режимі ReadOnly директорію хоста */var/log/*.

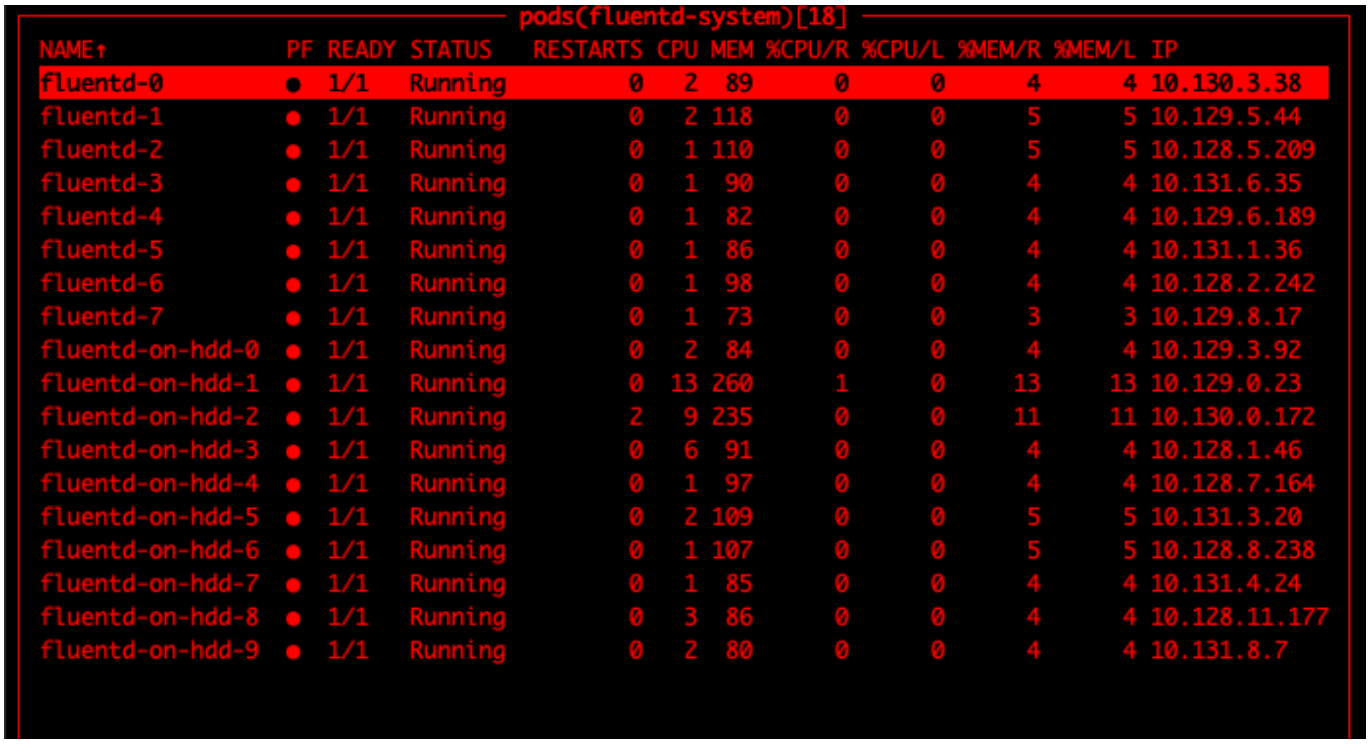

Рисунок 3.14 – Встановлені агенти Fluentd в Kubernetes

Сама SIEM система Graylog може бути встановлена як on-premises, так і використана хмарна версія [12][14].

На рисунку 3.15 зображено GUI Graylog, який відображає логи сервісів. На ньому можна побачити:

- графіки в розрізі часу;
- поле для фільтрів;
- повідомлення.

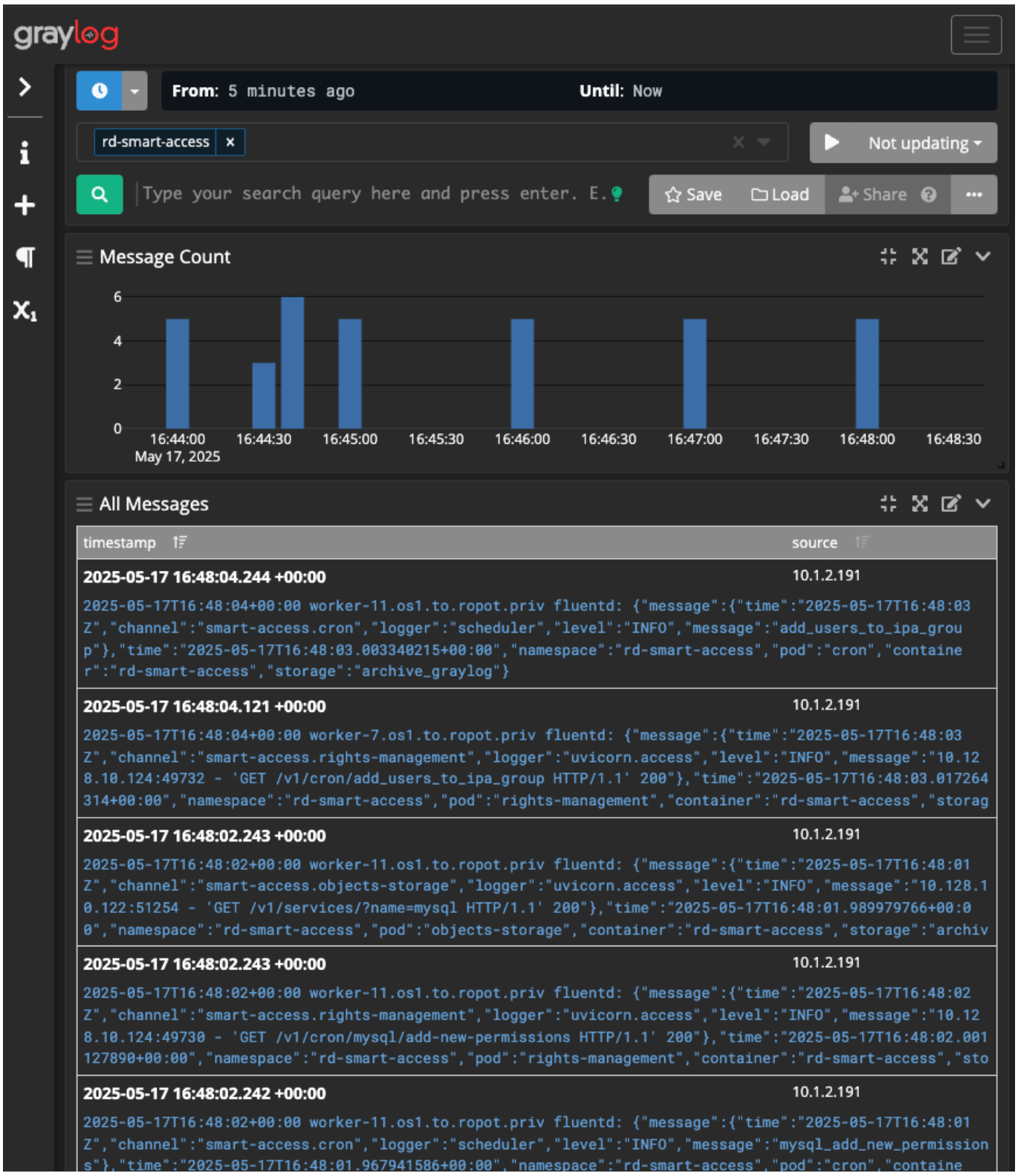

Рисунок 3.15 – Відображення логів у SIEM системі Graylog

Так як основна задача побудови комплексного рішення по побудові багаторівневої моделі захисту системи є надійність та доступність, то, використовуючи GitOps підхід, усі конфігураційні файли варто зберігати в

репозиторіях Git. На рисунку 3.16 зображено репозиторій з структурою зберігання таких конфігураційних файлів [15].

Рисунок 3.16 – Конфігураційні файли fluentd в репозиторії Git

3.4 Зберігання паролей та чутливих даних

Кожний сервісів, який працює, повинен підключатися до баз даних, мати SSL сертифікати, авторизовуватися в Graylog для передачі логів тощо. Усі ці процеси потребують чутливих даних – паролей або секретів (з англ. secrets). Якщо їх зберігати в незахищеному вигляді в Git або у вигляді файлів на хості, то це може призвести до:

- втрати паролей;
- неактуальності паролей;
- витоку чутливої інформації.

Як додатковий рівень захисту інформаційної системи як on-premises, так і в хмарі, необхідно потурбуватися про керування паролями.

У кінцевому рахунку, мета полягає в безпеці послуг, які кожен сучасний додаток або середовище вимагає, наприклад, управління паролем, шифрування даних тощо.

3.4.1 Hashicorp Key Vault

HashiCorp Vault – це платформа з відкритим кодом для керування секретами, створена для динамічних та гібридних середовищ. Цей інструмент підтримує тимчасові облікові дані, шифрування як послугу та контроль доступу на основі ідентифікації. Завдяки плагінам для Kubernetes, Terraform та хмарних провайдерів, вона є популярною серед команд DevOps, які керують складними розподіленими інфраструктурами [16]. На рисунку 3.17 зображено графічний інтерфейс Hashicorp Vault Secrets, де створюється новий секрет.

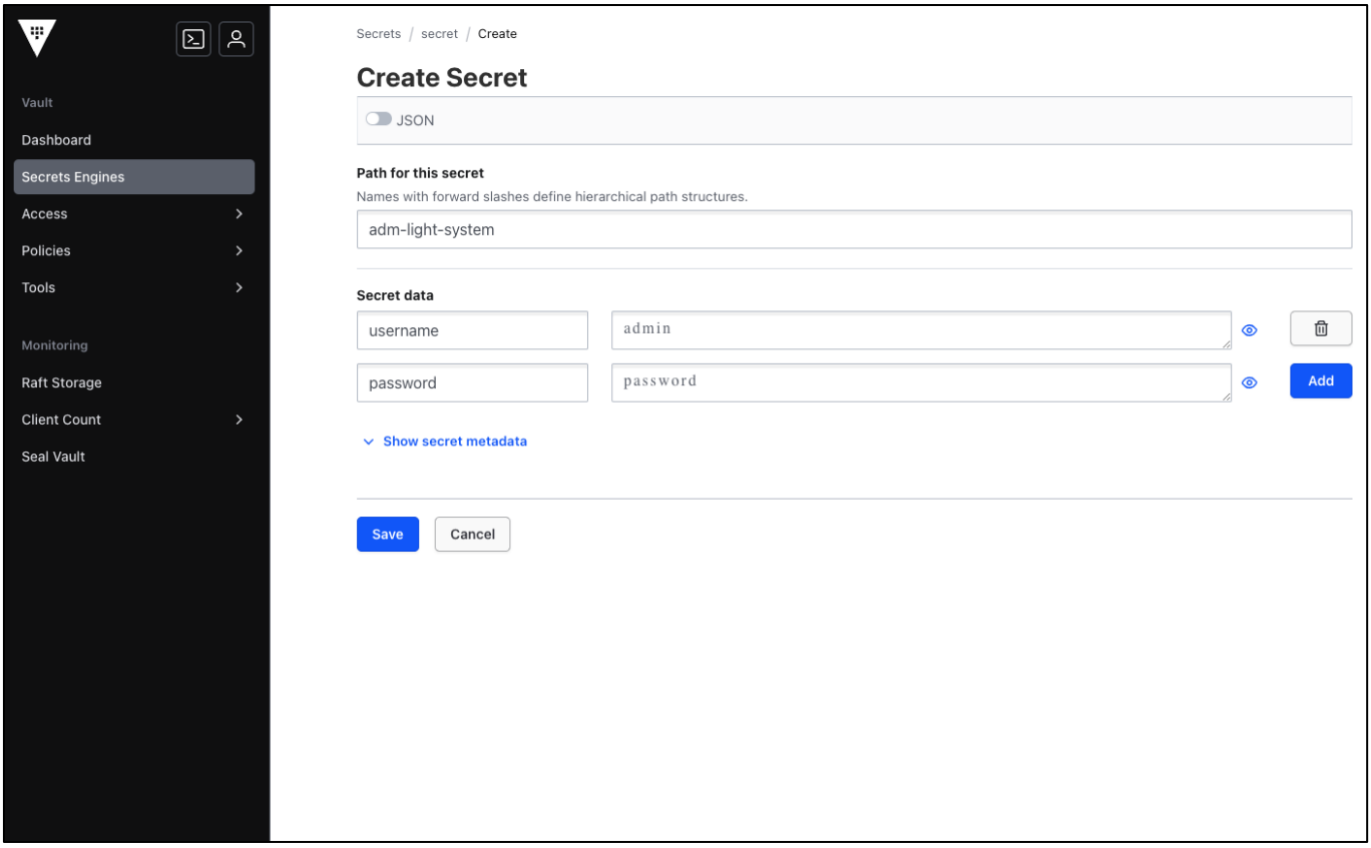

Рисунок 3.17 – Створення секрету в Hashicorp Key Vault

Обираючи цей інструмент для використання в хмарному середовищі Microsoft Azure, варто пам’ятати, що все управління та підтримка буде в зоні відповідальності системних адміністраторів. Microsoft Azure не пропонує цей сервіс як PaaS. Тобто потрібно розуміти для яких цілей потрібен цей інструмент. Наприклад, Hashicorp Key Vault має інтеграцію з GitLab, яка дозволяє інтегрувати та керувати секретами, які

потрібні для CI/CD пайплайнів одразу з GUI Hashicorp Key Vault. Також є офіційна інтеграція з Kubernetes за допомогою Vault Secrets Operator [17]. Цей інструмент дозволяє створювати Secrets об’єкти всередині Kubernetes з Hashicorp Key Vault. При зміні секрету, цей оператор помітить зміни та оновить секрет в Secrets об’єктах. Також є можливість автоматичного перезапуску Deployment об’єктів після зміни секретів. Це дозволяє одразу оновити секрети всередині сервісу, що працює [18].

3.4.2 Azure Key Vault

Azure Key Vault – це хмарна служба керування секретами від Microsoft, розроблена для безпечного зберігання та контролю доступу до криптографічних ключів, сертифікатів і конфіденційних даних конфігурації. Як PaaS сервіс Azure, вона безперешкодно інтегрується з іншими хмарними компонентами від Microsoft, забезпечуючи автоматичну ротацію, модулі апаратної безпеки (HSM) та сертифікати відповідності.

Спочатку необхідно створити сховище на порталі Microsoft Azure. (рисунок 3.18).

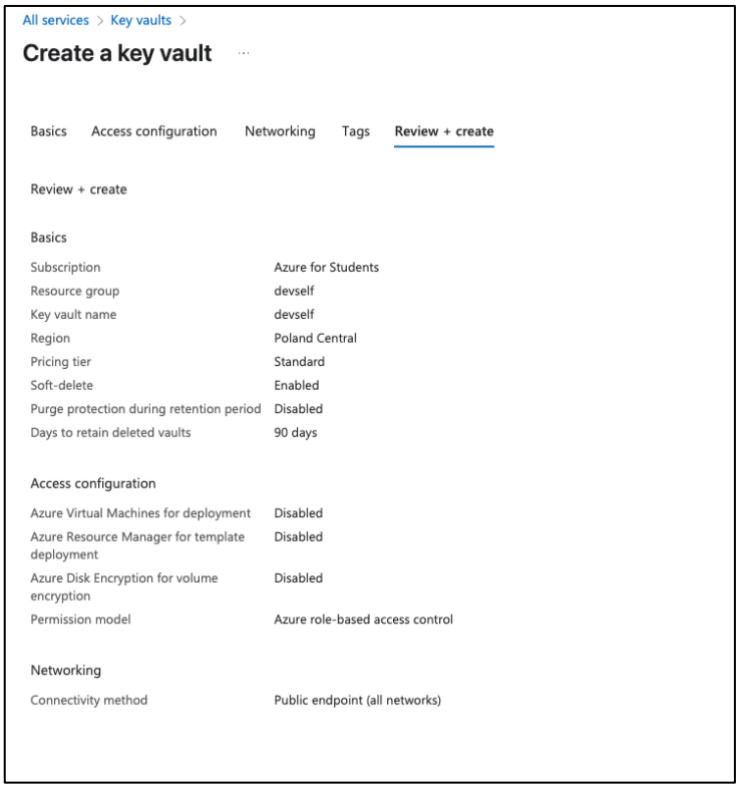

Рисунок 3.18 – Створення сховища в Microsoft Azure Key Vault

Створення сховища може займати деякий час, адже в цей момент розгортається спеціальне програмне забезпечення під інфраструктуру. Приклад сховища зображено на рисунку 3.19.

Рисунок 3.19 – Ініціалізація Azure Key Vault

Після цього можна створювати секрет. Для цього потрібно заповнити поля з назвою секрету, значенням секрету, строк дії секрету (можна обрати безстроковий) та дату активації, якщо це необхідно (рисунок 3.20).

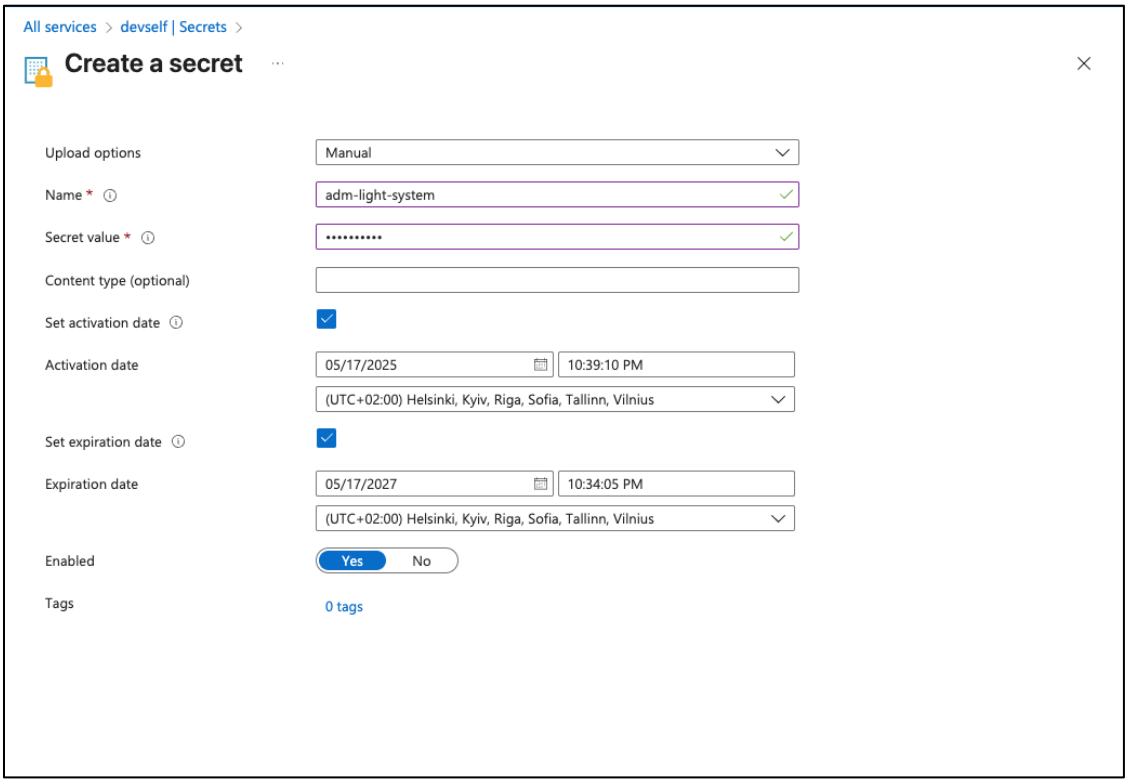

Рисунок 3.20 – Створення секрету в Azure Key Vault

Нижче в таблиці 3.1 наведено порівняння між Azure Key Vault та Hashicorp Key Vault.

Таблиця 3.1

Порівняння Hahicorp Key Vault з Azure Key Vault

Особливості	**Azure Key Vault**	**Hashicorp Key Vault**
Ціна	0,03–0,03–1,10/місяць за секрет/ключ.	Безкоштовно (з відкритим кодом) / корпоративні плани починаються від $0,50/годину.
Хмарне оточння	Тільки Azure	AWS, GCP, Azure, локально, гібридно.
Типи секретів	Статичні секрети, ключі, сертифікати.	Динамічні секрети, PKI, OIDC.
Контроль доступу	Azure RBAC та IAM.	ACL та policy
Масштабованість	Автоматичне масштабування (у межах обмежень Azure).	Необмежене масштабування.

Висновки до третього розділу

Даний розділ описує практичне знайомство з хмарною платформою Microsoft Azure та її налаштуванням від створення користувача до збору логів.

Було проведено комплекс дій по розгортанню інформаційної системи, яка може вважатися високонавантаженою, адже передбачається, що в реальних умовах кількість реплік буде збільшуватися. Для цього було створено та налаштовано балансувальник даних, який надсилатиме трафік до Azure Kubernetes Service.

Особливу увагу було приділено захисту безпеки даних, а саме: використання хмарних рішень по зберіганню секретів, логування сервісів, обмеження трафіку виключно з конкретних портів тощо.

РОЗДІЛ 4
РЕКОМЕНДАЦІЇ З ПІДТРИМКИ ВИСКОНАВАНТАЖЕНОЇ СИСТЕМИ ПІСЛЯ РОЗГОРТАННЯ В ХМАРНОМУ СЕРЕДОВИЩІ

Тепер, коли все налаштовано, варто потурбуватися про підтримку інфраструктури. Мається на увазі, що потрібно приділяти увагу наступним питанням:

- оновлення;
- документація;
- автоматизація.

4.1 Оновлення інфраструктури

Azure періодично оновлює свою платформу, щоб покращити надійність, продуктивність та безпеку інфраструктури хоста для віртуальних машин. Мета цих оновлень варіюється від виправлення програмних помилок у компонентах у середовищі до оновлення мережевих компонентів або виведення з експлуатації обладнання.

Оновлення рідко впливають на віртуальні машини. Коли оновлення має ефект, Azure обирає найменш впливовий метод для оновлень [19]:

Якщо оновлення не потребує перезавантаження, віртуальна машина призупиняється, поки хост оновлюється, або віртуальна машина переноситься на вже оновлений хост.

Якщо технічне обслуговування вимагає перезавантаження, системні адміністратори повинні отримати сповіщення про планове технічне обслуговування. Azure також надає часовий проміжок, протягом якого можна самостійно розпочати технічне обслуговування. Період самостійного оновлення зазвичай становить 35 днів (якщо технічне обслуговування не є терміновим). Azure підтримує технології, щоб зменшити кількість випадків, коли планове технічне обслуговування платформи вимагає перезавантаження віртуальних машин.

4.2 Документація

Досліджуючи проблеми керування високонавантаженою інфраструктурою, важливим фактором є неактуальна документація або взагалі її відсутність. Із зростанням кількості сервісів, допоміжних інструментів, залежностей тощо усе складніше тримати в голові зв’язаність між сервісами. Особливо критично постає питання в документації тоді, коли з’являється новий співробітник, якого потрібно ввести в курс справ та задати напрям роботи з проєктом.

Оформлення документації також важлива для безпеки інфраструктури. Наприклад, спеціаліст із кібербезпеки повинен знати мережеві правила безпеки до всіх віртуальних машин із метою дослідження та аналізу можливих загроз і кібератак. Відсутність або неактуальність цієї документації наражає всю інформаційну систему на невиправдані ризики.

Для полегшення роботи з Microsoft Azure компанія розробила ARI – Azure Resource Inventory. Azure Resource Inventory (ARI) - це комплексний модуль PowerShell, який генерує докладні звіти Excel про будь-яке середовище Azure, до якого є доступ. Він призначений для адміністраторів і технічних фахівців, яким потрібен простий і швидкий спосіб документування своїх середовищ Azure. ARI доступний як модуль PowerShell, який можна встановити безпосередньо виконавши команду в PowerShell [20]. PowerShell – це інструмент для створення автоматизації, який був розроблений Microsoft. Він призначений в основному для системних адміністрування. Він поєднує CLI – command line interface (інтерфейс командного рядка) зі скриптовою мовою, побудованого на базі фреймворку .NET, що дозволяє користувачам автоматизувати завдання, керувати налаштуваннями системи та різними службами Microsoft. PowerShell забезпечує доступ до системних компонентів, підтримує автоматизацію як у Windows, так і в крос-платформних середовищах, включаючи інтеграцію з Azure.

На рисунку 4.1 та 4.2 зображено приклад використання Azure Resource Inventory.

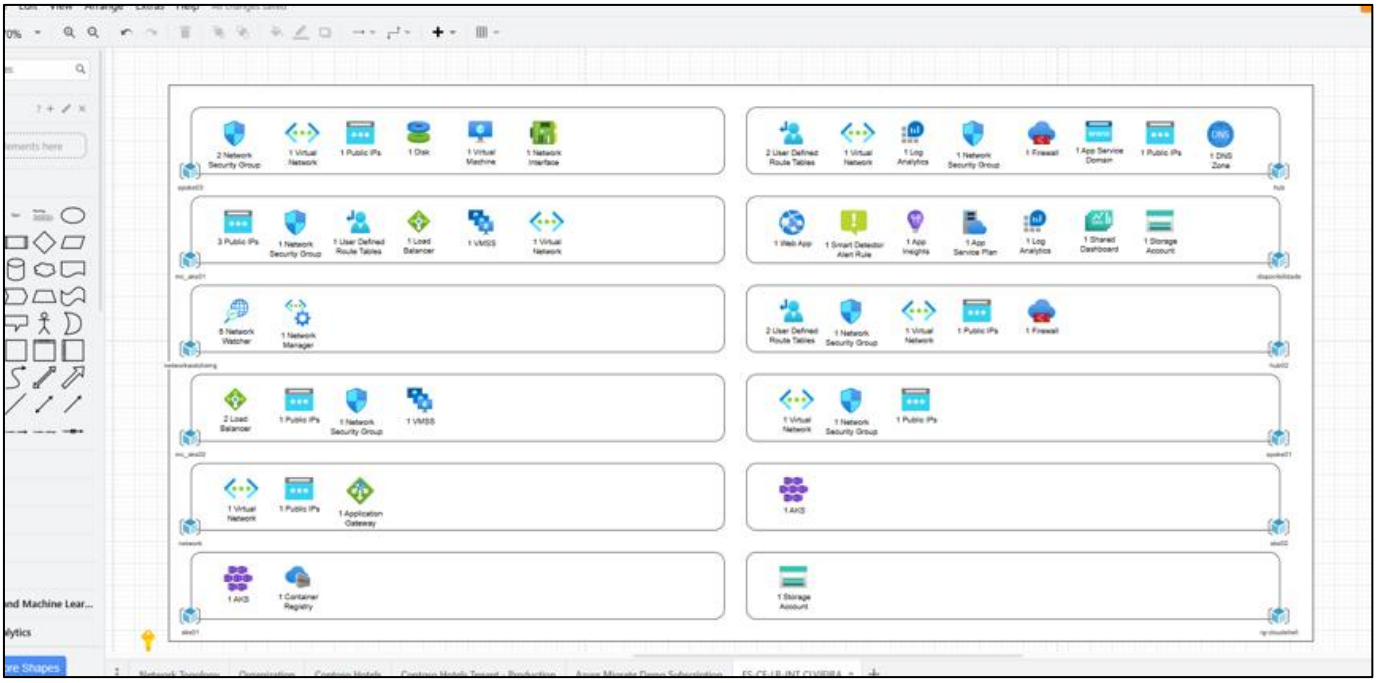

Рисунок 4.1 – Відображення всіх ресурсів у Microsoft Azure

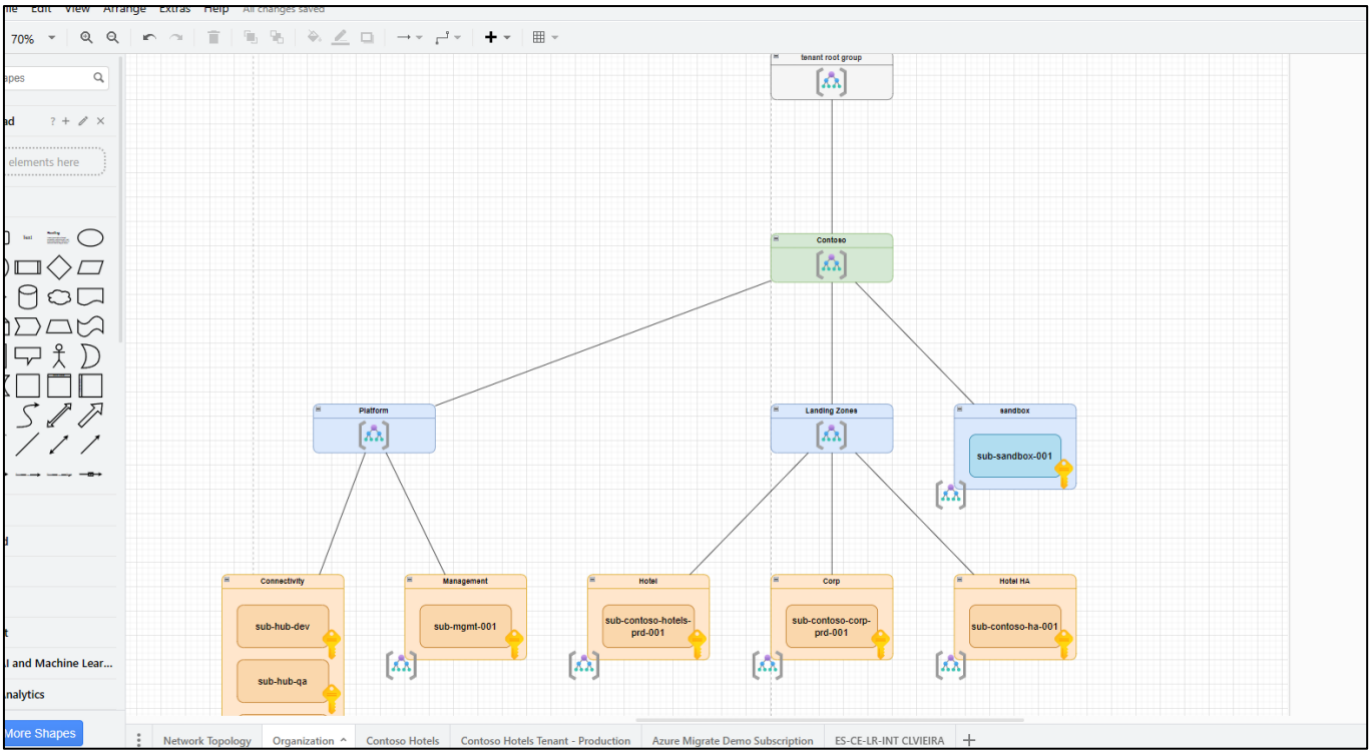

Рисунок 4.2 – Відображення дерева залежності організації у Microsoft Azure

Розглянувши базову документацію на рівні інфраструктури в хмарі, варто пам’ятати про документацію кожного сервісу, який був розроблений та має власну кодову базу. У GitLab існує готове рішення – Wiki [21]. Це розділ, у якому

зберігаються .md файли. Цей тип файлів має свій синтаксис, який робить документацію чіткою й структурованою. На рисунку 4.3 можна побачити приклад Wiki у GitLab із .md файлами.

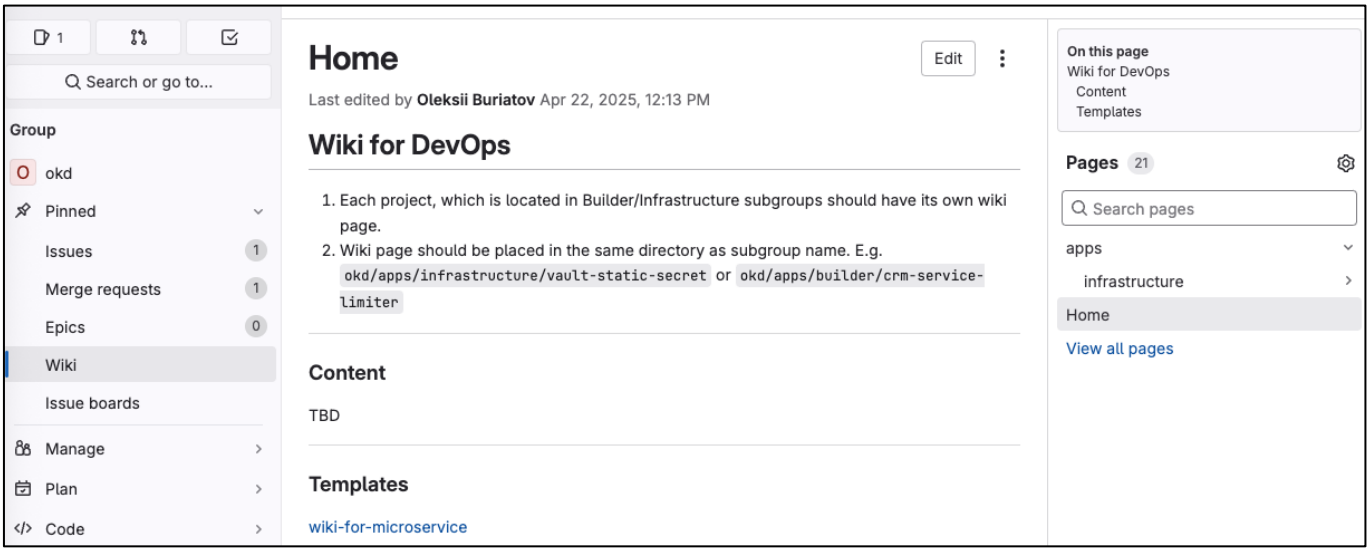

Рисунок 4.3 – Wiki в GitLab

Перевагою використання Wiki від Gitlab є те, що є можливість створення шаблонів, які легко заповнювати після того як сервіс був запущений в роботу (рисунок 4.4).

Рисунок 4.4 – Wiki шаблон для мікросервісів

На рисунку 4.4 зображено посилання на шаблон, який використовується при розгортанні мікросервісів.

4.3 Автоматизація процесів

Коли мікросервісів стає більше – з'являється необхідність у додаткових ресурсах. Коли з'являється потреба в додаткових ресурсах – збільшується інфраструктура: сервери, віртуальні машини, віртуальні мережі, групи мережевих безпек тощо. Чим більше всього, тим складніше керувати все це вручну [22].

Критично важливо максимально автоматизувати процеси. Наприклад, щоб установити пакет *python3.12* на 150 віртуальних машин займе цілий день роботи. На щастя, є інструменти, які дозволяють виконувати подібні операції зі встановлення пакетів, бібліотек, оновлень, файлів тощо. Наприклад, puppet має Enterprise рішення для керування Microsoft Azure інфраструктурою [22][23] (рисунок 4.5).

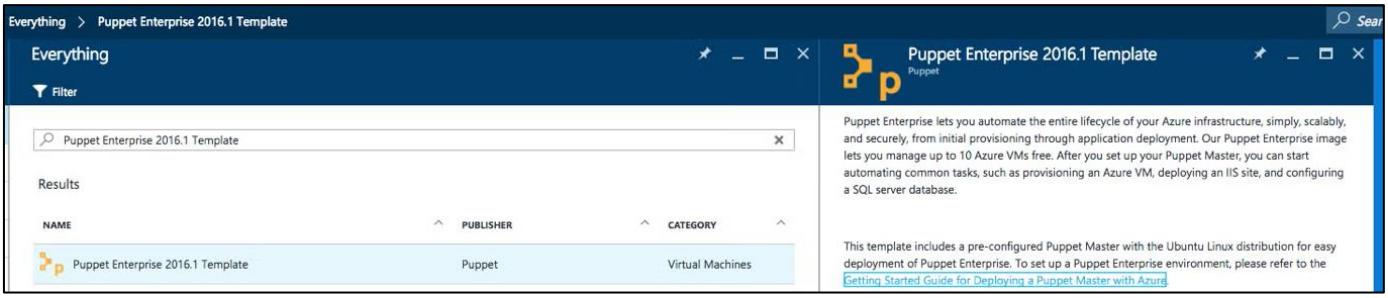

Рисунок 4.5 – Puppet Enterprise для Microsoft Azure

Висновки до четвертого розділу

У цьому розділі було проаналізовано та надано рекомендації з підтримки інфраструктури як всередині хмарної платформи так і поза нею. Було розглянуто основні питання, які потребують уваги, а саме:

- оновлення;
- документація;
- автоматизація інформаційної системи та оточення.

інформаційної системи та оточення. Було наведено приклади оформлення документації та використання інструментів щодо автоматизації виконання однотипних задач по всій інфраструктурі.

ВИСНОВКИ

У результаті проведеного дослідження багаторівневого захисту високонавантаженої системи в хмарному середовищі Microsoft Azure було проаналізовано ключові компоненти, що забезпечують безпеку, стабільність та масштабованість сервісів. Основна увага була зосереджена на порівнянні інструментів розгортання контейнеризованих додатків — Azure Container Instances (ACI) та Azure Kubernetes Service (AKS).

Дослідження показало, що AKS є більш придатним для високонавантажених систем, оскільки надає розширені можливості для масштабування, автоматизації оновлень, управління безпекою та інтеграції з CI/CD процесами.

Також було підтверджено, що впровадження безперервної інтеграції та доставки (CI/CD) безпосередньо впливає на цілісність та захист сервісів, знижуючи ризики людської помилки, пришвидшуючи оновлення систем та забезпечуючи контроль над змінами.

У рамках дослідження надано низку рекомендацій, які мають критичне значення для підтримки актуального та захищеного стану хмарної інфраструктури:

Регулярне оновлення компонентів (кластерів, контейнерів, бібліотек) як один із основних методів захисту від відомих вразливостей.

Ведення актуальної документації щодо архітектури, політик безпеки та процесів оновлення з метою забезпечення прозорості та узгодженості дій.

Одним із критичних аспектів забезпечення кібербезпеки в хмарних середовищах є захист конфіденційної інформації, зокрема облікових даних, ключів доступу, токенів авторизації, з'єднань до баз даних та API. У високонавантажених системах, де велика кількість компонентів взаємодіє між собою, небезпечно зберігати такі секрети в коді або у відкритому доступі. Для цього використовуються спеціалізовані сервіси — сховища із секретами (Secrets Vaults), наприклад, Azure Key Vault.

У сучасних хмарних платформах відсутність захищеного сховища секретів створює одну з найбільших загроз безпеці. Використання рішень, таких як Azure Key Vault або Hashicorp Key Vault, є обов’язковим елементом багаторівневого захисту, який дозволяє забезпечити конфіденційність, цілісність та контроль над критичною інформацією у високонавантажених і розподілених системах.

Автоматизація процесів розгортання, моніторингу та реагування, що дозволяє зменшити час відгуку на інциденти та забезпечити стабільність у роботі з високонавантаженими системами.

Отже, багаторівневий підхід до безпеки в Azure у поєднанні з сучасними практиками DevOps створює надійну основу для захисту критичних сервісів та забезпечення їх безперебійної роботи в умовах високого навантаження.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. COMPARATIVE STUDY OF CLOUD PLATFORMS - MICROSOFT AZURE, GOOGLE CLOUD PLATFORM AND AMAZON EC2 [Electronic resource] // JREAS. – 2020. – Access: https://pdfs.semanticscholar.org/10bf/f6d5dee2c2dd62f85eac3ea1900045cae820.pdf.

2. Dlnya A. A. Webserver Based Smart Monitoring System Using ESP8266 Node MCU Module [Electronic resource] / Abdulahad Aziz Dlnya // International Journal Of Scientific & Engineering Research. – 2018. – Access: https://www.researchgate.net/profile/Dlnya-Aziz-4/publication/326672970_Webserver_Based_Smart_Monitoring_System_Using_ESP8266_Node_MCU_Module/links/5b5cae8e458515c4b2501919/Webserver-Based-Smart-Monitoring-System-Using-ESP8266-Node-MCU-Module.pdf.

3. Yiwen Xu. Docker Desktop 4.41: Docker Model Runner supports Windows, Compose, and Testcontainers integrations, Docker Desktop on the Microsoft Store. [Electronic resource] // Docker. – Access: https://www.docker.com/blog/docker-desktop-4-41/.

4. Kubernetes [Electronic resource] // The Kubernetes. – Access: https://kubernetes.io/.

5. Yevgeniy Brikman. Terraform: Up & Running. – United States of America : O'Reilly Media, Inc., 2022. – 43–44 с. – ISBN 978-1-098-11674-3.

6. THE DEVOPS HANDBOOK / Gene Kim et al. – Portland : IT Revolution Press, LLC, 2016. – 467–468 с. – ISBN 978-1-942788-07-2.

7. Harness. Intro to Deployment Strategies: Blue-Green, Canary, and More. [Electronic resource] // Harness. – Access: https://www.harness.io/blog/blue-green-canary-deployment-strategies.

8. Prometheus [Electronic resource] // Prometheus Authors. – Access: https://prometheus.io/.

9. Chris Houseknecht (@chouseknecht), Matt Davis (@nitzmahone). Manage Azure

network security groups. [Electronic resource] // Ansible. – 15.05.2025. – Access: https://docs.ansible.com/ansible/latest/collections/azure/azcollection/azure_rm_securitygroup_module.html.

10. Press Release Globenewswire. ControlMonkey Redefines Disaster Recovery Cloudscape. [Electronic resource] // Markets Insider. – 25.03.2025. – Access: https://markets.businessinsider.com/news/stocks/controlmonkey-redefines-disaster-recovery-cloudscape-1034511546.

11. Microsoft. Service Level Agreements (SLA) for Online Services. [Electronic resource] // Microsoft. – Access: https://www.microsoft.com/licensing/docs/view/Service-Level-Agreements-SLA-for-Online-Services?assetType=285&year=2025.

12. Vijay K. What is Fluentd and use cases of Fluentd? [Electronic resource] // Theaiops. – 24.12.2024. – Access: https://www.theaiops.com/what-is-fluentd-and-use-cases-of-fluentd/.

13. Graylog. What is Graylog? [Electronic resource] // Graylog. – Access: https://go2docs.graylog.org/current/what_is_graylog/what_is_graylog.htm.

14. Graylog Cloud Platform. [Electronic resource] // Graylog. – Access: https://graylog.org/products/cloud/.

15. Scott Chacon, Ben Straub. Pro Git. – 2019. – 104–106 c.

16. Priya Mervana. Azure Key Vault vs HashiCorp Vault (2025): AI, Pricing & Multi-Cloud Tests. [Electronic resource] // Sslinsights. – 08.04.2025. – Access: https://sslinsights.com/azure-key-vault-vs-hashicorp-vault/.

17. Hashicorp. Vault-secrets-operator. [Electronic resource] // GitHub. – Access: https://github.com/hashicorp/vault-secrets-operator.

18. Warley’s CatOps. How to with Hashicorp Vault, a comprehensive guide. [Electronic resource] // Medium. – 23.03.2023. – Access: https://medium.com/@williamwarley/how-to-with-hashicorp-vault-a-comprehensive-guide-1e15c0afb37e.

19. Microsoft. Maintenance for virtual machines in Azure. [Electronic resource] // Microsoft. – 22.08.2024. – Access: https://learn.microsoft.com/en-us/azure/virtual-machines/maintenance-and-updates.

20. Microsoft. Azure Resource Inventory User Guide. [Electronic resource] // GitHub. – 06.03.2025. – Access: https://github.com/microsoft/ARI/blob/main/HowTo.md.

21. GitLab. Wiki. [Electronic resource] // GitLab. – Access: https://docs.gitlab.com/user/project/wiki/.

22. Microsoft. Infrastructure as code on Azure with Puppet & Chef. [Electronic resource] // Microsoft. – Access: https://info.microsoft.com/rs/157-GQE-382/images/Infrastructure-as-Code-guide-EN-v6_299129.pdf.

23. Lee Atchison. Architecting for Scale 2nd Edition. How to Maintain High Availability and Manage Risk in the Cloud. – New Relic, 2016. – 117–118 c. – ISBN 978-1-491-94339-7.

ДОДАТОК А

СПИСОК ОПУБЛІКОВАНИХ ПРАЦЬ ЗА ТЕМОЮ РОБОТИ

Тези наукових доповідей:

1. Oleksii Buriatov and Volodymyr Nakonechnyi. Cyber resilient cloud infrastructure for high-load systems. X INTERNATIONAL CONFERENCE. Information Technology and Implementation (Satellite), IT&Is-2023., Kyiv – P. 81.

2. Oleksii Buriatov and Volodymyr Nakonechnyi. Centralized application log collection and processing model for auditing and securing high-load systems. XI INTERNATIONAL CONFERENCE. Information Technology and Implementation (Satellite), IT&Is-2024., Kyiv – P. 202.

ДОДАТОК Б

fluentd.yml

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: fluentd
  namespace: fluentd-system
spec:
  serviceName: collector-logs
  replicas: 8  # Number of nodes or pods (1 pod per node)
  selector:
    matchLabels:
      app.kubernetes.io/instance: fluentd
  template:
    metadata:
      labels:
        app.kubernetes.io/instance: fluentd
    spec:
      securityContext:
        runAsUser: 0
        fsGroup: 0
      serviceAccount: fluentd-system
      serviceAccountName: fluentd-system
      tolerations:
        - key: "node-role.kubernetes.io/master"
          operator: "Exists"
          effect: "NoSchedule"
        - key: "nvidia.com/gpu"
          operator: "Exists"
```

```
    effect: "NoSchedule"
  affinity:
   podAntiAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
    - labelSelector:
       matchExpressions:
       - key: app.kubernetes.io/instance
         operator: In
         values:
         - fluentd
      topologyKey: "kubernetes.io/hostname"
  imagePullSecrets:
  - name: gitlab-docker-auth
  containers:
  - name: fluentd
    image:      gitlab.com:5000/okd/apps/infrastructure/fluentd-system/fluentd-syslog-
debian:1.17.1-1.0
    imagePullPolicy: Always
    securityContext:
     privileged: true
    env:
     - name: K8S_NODE_NAME
       valueFrom:
        fieldRef:
         fieldPath: spec.nodeName
     - name:  SYSLOG_HOST
       value: "backup.local"
     - name:  SYSLOG_PORT
       value: "10515"
     - name:  SYSLOG_PROTOCOL
       value: "tcp"
```

```
  resources:
    limits:
      memory: 2000Mi
      cpu: 2000m
    requests:
      cpu: 1000m
      memory: 2000Mi
  volumeMounts:
  - name: logpods
    mountPath: /var/log/pods
    readOnly: true
  - name: varlog
    mountPath: /var/log
  - mountPath: /fluentd/etc/fluent.conf
    name: config-volume
    subPath: fluent.conf
  - mountPath: /fluentd/etc/conf.d
    name: config-volume-apps
  - mountPath: /fluentd/indexes
    name: fluentd-data
terminationGracePeriodSeconds: 30
volumes:
- name: logpods
  hostPath:
    path: /var/log/pods
- name: varlog
  hostPath:
    path: /var/log
- name: config-volume
  configMap:
    name: tail-container-parse
```

```
        items:
          - key: fluent.conf
            path: fluent.conf
    - name: config-volume-apps
      configMap:
        name: tail-container-parse-apps
volumeClaimTemplates:
- metadata:
    name: fluentd-data
  spec:
    accessModes: ["ReadWriteOnce"]
    resources:
      requests:
        storage: 2Gi
    storageClassName: local-ssd
```

ДОДАТОК В

kubernetes.tf

```
resource "kubernetes_namespace" "kube_ns_base_project" {
  metadata {
    name = var.gitlab_project_path
    labels = {
      managed-by = "Terraform"
    }
  }
  lifecycle {
    ignore_changes = [
      metadata[0].annotations
    ]
  }
}

resource "kubernetes_manifest" "kube_sa_for_flux" {
  manifest = {
    "apiVersion" = "v1"
    "kind"       = "ServiceAccount"
    metadata = {
      "namespace" = kubernetes_namespace.kube_ns_base_project.metadata.0.name
      "name"      = var.gitlab_project_path
      labels = {
        managed-by = "Terraform"
      }
    }
    "automountServiceAccountToken" = true
  }
```

```
  lifecycle {
    ignore_changes = [
      manifest.metadata.annotations,
      manifest.metadata.clusterName
    ]
  }

}

resource "kubernetes_secret" "ghcr_auth" {
  metadata {
    name      = "ghcr-auth"
    namespace = kubernetes_namespace.kube_ns_base_project.metadata[0].name
    labels = {
      managed-by = "Terraform"
    }
  }

  type = "kubernetes.io/dockerconfigjson"

  data = {
    ".dockerconfigjson" = jsonencode({
      auths = {
        "${var.registry_server}" = {
          "username" = gitlab_deploy_token.fluxcd.username
          "password" = gitlab_deploy_token.fluxcd.token
          "email"    = var.registry_email
          "auth"                                                    =
base64encode("${gitlab_deploy_token.fluxcd.username}:${gitlab_deploy_token.fluxcd.token
}")
        }
```

```
      }
    })
  }
}

resource "kubernetes_secret" "gitlab_docker_auth" {
  metadata {
    name      = "gitlab-docker-auth"
    namespace = kubernetes_namespace.kube_ns_base_project.metadata[0].name
    labels = {
      managed-by = "Terraform"
    }
  }

  type = "kubernetes.io/dockerconfigjson"

  data = {
    ".dockerconfigjson" = jsonencode({
      auths = {
        "${var.registry_server}" = {
          "username" = gitlab_deploy_token.fluxcd.username
          "password" = gitlab_deploy_token.fluxcd.token
          "email"    = var.registry_email
          "auth"                                                                    =
base64encode("${gitlab_deploy_token.fluxcd.username}:${gitlab_deploy_token.fluxcd.token
}")
        }
      }
    })
  }
}
```

```
resource "kubernetes_secret" "flux_deploy" {
  metadata {
    name      = "flux-deploy-${var.gitlab_project_path}"
    namespace = kubernetes_namespace.kube_ns_base_project.metadata[0].name
    labels = {
      managed-by = "Terraform"
    }
  }

  data = {
    username = gitlab_deploy_token.fluxcd.username
    password = gitlab_deploy_token.fluxcd.token
  }

  type = "Opaque"
}

resource "kubernetes_cluster_role_binding" "bind_patch" {
  metadata {
    name = "${var.gitlab_project_path}-patch"
    labels = {
      managed-by = "Terraform"
    }
  }
  role_ref {
    api_group = "rbac.authorization.k8s.io"
    kind      = "ClusterRole"
    name                                                                    =
data.terraform_remote_state.common_00_dev.outputs.kube_cluter_role_flux_patch
  }
```

```
  subject {
    kind      = "ServiceAccount"
    name      = kubernetes_manifest.kube_sa_for_flux.manifest.metadata.name
    namespace = kubernetes_namespace.kube_ns_base_project.metadata[0].name
  }
}

resource "kubernetes_cluster_role_binding" "bind_admin" {
  metadata {
    name = "${var.gitlab_project_path}-admin"
    labels = {
      managed-by = "Terraform"
    }
  }
  role_ref {
    api_group = "rbac.authorization.k8s.io"
    kind      = "ClusterRole"
    name      = "admin"
  }
  subject {
    kind      = "ServiceAccount"
    name      = kubernetes_manifest.kube_sa_for_flux.manifest.metadata.name
    namespace = kubernetes_namespace.kube_ns_base_project.metadata[0].name
  }
}
```

ДОДАТОК Д

.gitlabci.yml

```
variables:
  SERVICE_USER_SEARCH_LIGHT: "service_user_search_light"
  SERVICE_USER_SEARCH_LIGHT_DOCKERFILE_PATH:
"./Dockerfiles/service_user_search_light.dockerfile"

service_user_search_light_build:
  stage: build
  image: gitlab.com:5000/okd/conf/gitrunner-s2i-podman-push:1.1
  allow_failure: false
  rules:
    - if: '($CI_PIPELINE_SOURCE == "trigger" || $CI_PIPELINE_SOURCE == "web")
&& ($BITBUCKET_PROJECT_ENV=="rel" || $BITBUCKET_PROJECT_ENV=="master")
&& ($REVERT_DEPLOY == "no") && $BUILD_USER_SEARCH_LIGHT == "yes" &&
$CI_COMMIT_REF_NAME == "main"'
  id_tokens:
    VAULT_ID_TOKEN:
      aud: https://gitlab.com
  secrets:
    BITBUCKET_SSH_KEY:
      vault: okd/to/${VAULT_AUTH_ROLE}/ssh-bitbucket/ssh-key.priv@admins
      file: false
      token: $VAULT_ID_TOKEN
    CRM_PIPSERVER_USER:
      vault: okd/to/common/crm/pipserver/username@admins
      file: false
      token: $VAULT_ID_TOKEN
    CRM_PIPSERVER_PASSWORD:
```

```
        vault: okd/to/common/crm/pipserver/password@admins
        file: false
        token: $VAULT_ID_TOKEN
      CRM_PIPSERVER_HOST:
        vault: okd/to/common/crm/pipserver/host@admins
        file: false
        token: $VAULT_ID_TOKEN
      CRM_PIPSERVER_NETRC:
        vault: okd/to/common/crm/pipserver/netrc@admins
        file: false
        token: $VAULT_ID_TOKEN
    before_script:
      - echo "${BITBUCKET_SSH_KEY}" > ~/.ssh/id_rsa
      - echo "${SSH_CONFIG}" > ~/.ssh/config
      - chmod 600 ~/.ssh/id_rsa ~/.ssh/config
      - podman login ${PRIVATE_REGISTRY_URL} -u ${CI_REGISTRY_USER} -p
${CI_REGISTRY_PASSWORD}
      - podman pull
${S2I_SOURCE_IMAGE_PYTHON}:${S2I_SOURCE_TAG_PYTHON} --authfile
${S2I_SOURCE_IMAGE_AUTH_FILE}
    script:
      - s2i build --ref=${BITBUCKET_PROJECT_ENV}
${BITBUCKET_PROJECT_URL} --context-
dir=${BITBUCKET_CONTEXT_DIR}/${SERVICE_USER_SEARCH_LIGHT}
${S2I_SOURCE_IMAGE_PYTHON} --as-dockerfile Dockerfile
${SERVICE_USER_SEARCH_LIGHT}
      - echo $CRM_PIPSERVER_NETRC > netrc
      - cp -r ${SERVICE_USER_SEARCH_LIGHT_DOCKERFILE_PATH} Dockerfile
      - podman build --platform linux/amd64 --build-arg
PIP_INDEX_URL=$PIP_INDEX_URL --build-arg
MICROPIPENV_DEFAULT_INDEX_URLS=$PIP_INDEX_URL -t
```

```
${SERVICE_USER_SEARCH_LIGHT}_${BITBUCKET_PROJECT_ENV}:pipeline-
${CI_JOB_ID} -f Dockerfile
        - podman run -it --rm
${SERVICE_USER_SEARCH_LIGHT}_${BITBUCKET_PROJECT_ENV}:pipeline-
${CI_JOB_ID} /bin/bash -c "python3.9 -m unittest /opt/app-
root/src/${SERVICE_USER_SEARCH_LIGHT}/tests/test_runner.py"; echo $?
        - podman tag
${SERVICE_USER_SEARCH_LIGHT}_${BITBUCKET_PROJECT_ENV}:pipeline-
${CI_JOB_ID}
${PRIVATE_REGISTRY_URL}/${TARGET_REGISTRY_PATH}/${SERVICE_USER_S
EARCH_LIGHT}_${BITBUCKET_PROJECT_ENV}:pipeline-${CI_JOB_ID}
        - podman push --authfile /run/containers/0/auth.json
${PRIVATE_REGISTRY_URL}/${TARGET_REGISTRY_PATH}/${SERVICE_USER_S
EARCH_LIGHT}_${BITBUCKET_PROJECT_ENV}:pipeline-${CI_JOB_ID}
        - echo
"SERVICE_USER_SEARCH_LIGHT_REPOSITORY=${PRIVATE_REGISTRY_URL}/${
TARGET_REGISTRY_PATH}/${SERVICE_USER_SEARCH_LIGHT}_${BITBUCKET_
PROJECT_ENV}" >> build.env
        - echo "BASE_IMAGE_TAG=pipeline-${CI_JOB_ID}" >> build.env
      artifacts:
        reports:
          dotenv: build.env

    service_user_search_light_merge_request:
      stage: merge_request
      image: gitlab.com:5000/okd/conf/gitpusher:1.2
      allow_failure: false
      when: on_success
      rules:
        - if: '($CI_PIPELINE_SOURCE == "trigger" || $CI_PIPELINE_SOURCE == "web")
&& ($BITBUCKET_PROJECT_ENV=="rel" || $BITBUCKET_PROJECT_ENV=="master")
```

```
&& ($REVERT_DEPLOY == "no") && $BUILD_USER_SEARCH_LIGHT == "yes" &&
$CI_COMMIT_REF_NAME == "main"'
    id_tokens:
      VAULT_ID_TOKEN:
        aud: https://gitlab.com
    secrets:
      BITBUCKET_SSH_KEY:
        vault: okd/to/${VAULT_AUTH_ROLE}/ssh-bitbucket/ssh-key.priv@admins
        file: false
        token: $VAULT_ID_TOKEN
    before_script:
      - mkdir ~/.ssh && touch ~/.ssh/id_rsa && touch ~/.ssh/config
      - echo "$BITBUCKET_SSH_KEY" > ~/.ssh/id_rsa
      - echo "${SSH_CONFIG}" > ~/.ssh/config
      - chmod 600 ~/.ssh ~/.ssh/id_rsa ~/.ssh/config
      - git config --global user.email "ci@gitlab.com"
      - git config --global user.name "GitLab CI"
    script:
      - git clone ${GITLAB_SSH_URL}:${TARGET_REGISTRY_PATH}.git
      - cd ${CI_PROJECT_NAME}
      - git switch ${BITBUCKET_PROJECT_ENV}
      - if git branch -a | grep -q "merge_to_${BITBUCKET_PROJECT_ENV}"; then git
switch merge_to_${BITBUCKET_PROJECT_ENV}; git pull origin
merge_to_${BITBUCKET_PROJECT_ENV}; else git switch -c
merge_to_${BITBUCKET_PROJECT_ENV}; fi
      - BASE_IMAGE_TAG_UPDATED=$BASE_IMAGE_TAG yq '.image.tag =
strenv(BASE_IMAGE_TAG_UPDATED)' -i ./${HELM_VALUES_FILE_PATH}
      - git add .
      - git commit -m "CI Deploy with new TAG"
      - git push -o merge_request.create -o
merge_request.target=${BITBUCKET_PROJECT_ENV} -o
```

```
merge_request.remove_source_branch -o merge_request.assign="root" origin
merge_to_${BITBUCKET_PROJECT_ENV}
      needs:
        - job: service_user_search_light_build
          artifacts: true
```