Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет *інфокомунікацій*
(повна назва)

Кафедра *інформаційно-мережної інженерії*
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

на тему *Аналіз та оптимізація витрат на хмарні ресурси в умовах непередбачуванного навантаження з фокусом на автоматизацію виробничих процесів управління інфраструктурою*

Виконав:
студент *2* курсу, групи *ІМІм-21-2*

Кобзєв В.Д.
(прізвище та ініціали)

Спеціальність *172 Телекомунікації і радіотехніка*
(код і повна назва спеціальності)

Тип програми *Освітньо-наукова*
(освітньо-професійна або освітньо-наукова)

Освітня програма *Інформаційно - мережна інженерія*
(повна назва освітньої програми)

Керівник *доцент каф. ІМІ, Костромицький А.І.*
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ *Безрук В.М.*
(підпис) (прізвище, ініціали)

Харків - 2023 рок

Не містить відомостей, заборонених до відкритого публікування

Студент _____

Керівник_____

Харківський національний університет радіоелектроніки
(повне найменування вищого навчального закладу)
Факультет *інфокомунікацій*
Кафедра *інформаційно-мережної інженерії*
Рівень вищої освіти *другий (магістерський)*
Напрям підготовки *172 «Телекомунікації і радіотехніка»*
(шифр і назва)

ЗАТВЕРДЖУЮ

Зав. кафедри _____
(Підпис)
"____" _____ 2023 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Кобзєв Вадима Дмитровича
(прізвище, ім'я, по батькові)

1. Тема роботи *Аналіз та оптимізація витрат на хмарні ресурси в умовах непередбачуванного навантаження з фокусом на автоматизацію виробничих процесів управління інфраструктурою*
затверджені наказом ВНЗ від "17" березня 2023 року №275Ст
2. Строк подання студентом роботи *19.05.2023*
3. Вихідні дані до роботи *Аналіз, оптимізація та автоматизація, управління витратами на хмарні ресурси в умовах непередбачуваного навантаження, з урахуванням конкретних потреб виробничого середовища.*

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)
Вступ
1. *Мета та теоретичні аспекти дослідження*
2. *Технології автоматизації та управління інфраструктурою*
3. *Практична частина: розробка та реалізація системи*
4. *Тестування та аналіз системи*
5. *Рекомендації подальшого розвитку*
6. *Висновки*

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

Слайди у форматі PowerPoint (назва та мета роботи, актуальність, технології автоматизації, практична частина, тестування ссистеми, рекомендації подальшого розвитку б висновки)

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада Консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Основна частина	*доц. Костромицький А.І.*	20.03.2023	

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів атестаційної роботи	Строк виконання етапів роботи	Примітка
1	*Ознайомлення із завданням. Уточнення ТЗ.*	20.03-21.03.2023	
2	*Підбір літератури за темою роботи.*	22.03-25.03.2023	
3	*Виконання розділу 1*	25.03-05.04.2023	
4	*Виконання розділу 2*	06.04-20.04.2023	
5	*Виконання розділу 3 та 4*	21.04-10.05.2023	
6	*Оформлення презентаційного матеріалу,*	11.05-19.05.2023	
	підготовка до захисту у ЕК		

Дата видачі завдання 20.03.2023 року.

Студент _____ *Кобзєв В.Д.*
(підпис) (прізвище та ініціали)

Керівник роботи _____ *Костромицький А.І*
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Пояснювальна записка: 67 с., 41 рис., 11 джерел, 22 слайдів презентації., наукова публікація 2 с.

ОПТИМІЗАЦІЯ, ХМАРНІ РЕСУРСИ, НЕПЕРЕДБАЧУВАНЕ НАВАНТАЖЕННЯ, EKS, KARPENTER, KEDA, АВТОМАТИЧНЕ МАСШТАБУВАННЯ, TERRAFORM, АВТОМАТИЗАЦІЯ УПРАВЛІННЯ ІНФРАСТРУКТУРОЮ, КОНСИСТЕНТНІСТЬ КОНФІГУРАЦІЙ, СТРАТЕГІЇ УПРАВЛІННЯ РЕСУРСАМИ.

Це дослідження присвячене оптимізації та управлінню хмарними ресурсами в умовах непередбачуваного навантаження. В ньому розглядаються технології як EKS, Karpenter та KEDA для автоматичного масштабування та адаптації хмарної інфраструктури до змін навантаження, а також використання Terraform для автоматизації управління інфраструктурою. Результати показують важливість таких підходів для ефективного використання хмарних ресурсів і зниження витрат.

ABSTRACT

Explanatory Note: 67 pages, 41 pictures, 11 references, 22 presentation slides, 2 pages scientific publications.

OPTIMIZATION, CLOUD RESOURCES, UNPREDICTABLE WORKLOAD, EKS, KARPENTER, KEDA, AUTOMATIC SCALING, TERRAFORM, INFRASTRUCTURE MANAGEMENT AUTOMATION, CONFIGURATION CONSISTENCY, RESOURCE MANAGEMENT STRATEGIES.

This research is dedicated to the optimization and management of cloud resources in the face of unpredictable workload. It examines technologies such as EKS, Karpenter, and KEDA for automatic scaling and adaptation of cloud infrastructure to changing workloads, as well as the use of Terraform for infrastructure management automation. The results demonstrate the importance of these approaches for efficient utilization of cloud resources and cost reduction.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

CI/CD (Continuous Integration/Continuous Delivery) - Постійна інтеграція/постійна доставка.

AWS (Amazon Web Services) - хмарні сервіси компанії Amazon.

KEDA (Kubernetes-based Event-driven Autoscaling) - Автомасштабування на основі подій для Kubernetes.

EKS (Amazon Elastic Kubernetes Service) - Керований сервіс для розгортання та управління кластерами Kubernetes.

IaaS (Infrastructure as a Service) - Інфраструктура як сервіс.

PaaS (Platform as a Service) - Платформа як сервіс.

SaaS (Software as a Service) - Програмне забезпечення як сервіс.

IT (Information Technology) - Інформаційні технології.

EC2 (Elastic Compute Cloud) - Сервіс віртуальних серверів.

API (Application Programming Interface) - Інтерфейс програмування додатків.

CPU (Central Processing Unit) - Центральний процесор.

VPC (Virtual Private Cloud) - Віртуальна приватна хмара.

ОС (Operating System) - Операційна система.

HPA (Horizontal Pod Autoscaler) - Горизонтальний автомасштабувальник підпроцесів.

ALB (Application Load Balancer) - Балансувальник навантаження додатків.

NLB (Network Load Balancer) - Балансувальник навантаження мережі.

CLB (Classic Load Balancer) - Класичний балансувальник навантаження.

IaC – (Infrastructure as Code) - Інфраструктура як код.

ASG (Auto Scaling Group) - Група автоматичного масштабування.

JCasC (Jenkins Configuration as Code) - Конфігурація Jenkins як коду.

ВСТУП

Тема даного дослідження присвячена проблемам оптимізації використання та управління хмарними ресурсами в умовах непередбачуваного навантаження. Актуальність вибраного напрямку полягає в розширенні присутності сучасного бізнесу в хмарних технологіях, що вимагає від компаній більшої гнучкості та ефективності управління IT-інфраструктурою. Неправильне або недостатньо гнучке управління хмарними ресурсами може призвести до їх неефективного використання та збільшення витрат, особливо при непередбачуваному навантаженні.

У сучасному динамічному світі навантаження на хмарні ресурси може суттєво варіюватися, що вимагає від компаній здатності швидко адаптуватися до змін. Зокрема, потрібно ефективно прогнозувати навантаження, використовувати механізми автоматичного масштабування, вибирати оптимальні типи ресурсів, управляти життєвим циклом ресурсів та впроваджувати стратегії зниження вартості використання хмарних ресурсів. Автоматизація процесів управління інфраструктурою є також важливим аспектом для забезпечення ефективності та зниження витрат, що зумовлено зменшенням помилок, пов'язаних з людським фактором, та підвищенням продуктивності систем.

Незважаючи на те, що в багатьох сферах бізнесу дуже складно провести цей аналіз не маючи об'єкта дослідження, а кожна нова інтеграція може знизити актуальність попередньої стратегії своєю непередбачуваністю та динамікою змін, але існують області, де можна застосувати певну стандартизацію та прогнозування. Однією з таких областей є системи неперервної інтеграції та доставки (CI/CD), які використовуються в більшості сучасних IT-інфраструктур.

CI/CD системи, такі як Jenkins, є важливими елементами більшості сучасних IT-інфраструктур, вони також вимагають значних ресурсів для своєї роботи та потребують постійного управління та оптимізації.

Оптимізація використання систем як Jenkins та хмарних ресурсів, на яких вони працюють, може стати важливою частиною загальної стратегії ефективного використання хмарних ресурсів. Тому це дослідження має на меті вивчити можливості оптимізації використання хмарних ресурсів на прикладі Jenkins, а також розглянути можливість перенесення отриманих результатів на інші веб-додатки.

1. МЕТА ТА ТЕОРЕТИЧНІ АСПЕКТИ ДОСЛІДЖЕННЯ

1.1. Мета та структура завдання дослідження

Дослідження актуальних підходів до оптимізації витрат на хмарні ресурси стає все більш важливим для успішної діяльності компаній в умовах постійної зміни та конкуренції на ринку. Правильний вибір хмарного провайдера для нашого дослідження, такого як AWS відіграє важливу роль у досягненні максимальної користі від використання хмарних технологій. AWS відомий своєю відмінною репутацією, широким спектром доступних послуг, гнучкістю, масштабованістю та надійністю.

На меті ми прийняли рішення розглянути та розробити стратегію для оптимізації на прикладі системи неперервної інтеграції та доставки Jenkins. В контексті цього дослідження, особливу увагу буде приділено сервісам AWS но і також використанню інших технологій, включаючи Terraform, Kubernetes, Karpenter, KEDA, які можуть допомогти в оптимізації використання ресурсів.

Структура завдання дослідження:

1. Оглянути хмарні технології та їх вплив на бізнес.
2. Дослідити моделі ціноутворення обраних хмарних послуг AWS.
3. Дослідити особливості використання хмарних ресурсів системою Jenkins враховуючи Kubernetes, включаючи в умовах непередбачуваного навантаження.
4. Розгортання оптимального середовища для Jenkins на AWS використовуючи EKS.
5. Розробити стратегію для оптимізації використання хмарних ресурсів системою Jenkins, в тому числі з використанням AWS, KEDA та Karpenter.
6. Провести експерименти з розробленою стратегіями моделювальному навантаженні для оцінки їхньої ефективності.

7. Розглянути можливість перенесення розроблених стратегій на інші веб-додатки.
8. Зробити висновки на основі отриманих результатів та обговорити можливі напрямки подальшого дослідження.

1.2. Опис хмарних технологій та їх вплив на бізнес

З розвитком інформаційних технологій та глобалізацією світової економіки бізнесу потрібні ефективні й гнучкі рішення для управління своїми IT-ресурсами. Хмарні технології надають компаніям можливість забезпечити такі рішення, оскільки вони пропонують широкий спектр послуг, що дозволяють компаніям легко масштабувати, управляти та оптимізувати свої IT-інфраструктури.

Хмарні технології пропонують три основні моделі розгортання:

1. Публічні хмари — публічні хмари пропонують послуги через глобальну мережу інтернет, дозволяючи компаніям сплачувати тільки за використані ресурси. Це дозволяє компаніям значно знизити витрати на IT-інфраструктуру та забезпечити більшу гнучкість в управлінні ресурсами.[1]

2. Приватні хмари — приватні хмари розгортаються в межах корпоративної мережі, забезпечуючи компаніям більший контроль над своїми даними та ресурсами. Це дозволяє компаніям забезпечити відповідність своїх систем безпеки та забезпечити відповідний рівень приватності. Приватні хмари можуть бути внутрішніми (розгорнуті на власному обладнанні компанії) або зовнішніми (розміщеними у провайдера хмарних послуг).[1]

3. Гібридні хмари — гібридні хмари комбінують публічні та приватні хмари, дозволяючи компаніям використовувати переваги обох моделей. Гібридні хмари дозволяють компаніям забезпечити оптимальне розподілення ресурсів між публічними та приватними хмарами відповідно до своїх потреб та стратегії. Гібридні хмари можуть допомогти компаніям розподіляти

навантаження, забезпечувати високу доступність та розширювати можливості для інновацій.[1]

Хмарні технології пропонують також три основні сервісні моделі:

1. Infrastructure as a Service (IaaS) — IaaS надає користувачам доступ до віртуальної інфраструктури, такої як віртуальні сервери, мережі та сховища даних. Компанії можуть використовувати IaaS для розгортання та управління своїми віртуальними машинами, використовуючи хмарні ресурси відповідно до своїх потреб. IaaS дозволяє компаніям легко масштабувати свої ресурси та оптимізувати витрати на інфраструктуру, окремі деталі на рис 1.1.[1]

2. Platform as a Service (PaaS) — PaaS надає користувачам доступ до платформи, яка включає операційні системи, бази даних, розробницькі інструменти та інші компоненти, необхідні для розробки, розгортання та управління додатками. PaaS дозволяє компаніям зосередитися на розробці та оптимізації своїх додатків, замість забезпечення та управління інфраструктурою. PaaS може бути використаний для розробки, тестування та розгортання додатків, що забезпечують швидке впровадження нових рішень на ринку, окремі деталі на рис 1.[1]

3. Software as a Service (SaaS) — SaaS надає користувачам доступ до готових додатків, які розгортаються на хмарній інфраструктурі. SaaS дозволяє компаніям використовувати додатки без необхідності встановлення, налаштування та управління інфраструктурою. SaaS пропонує широкий спектр додатків, від простих офісних програм до складних бізнес-систем, окремі деталі на рис 1.1.[1]

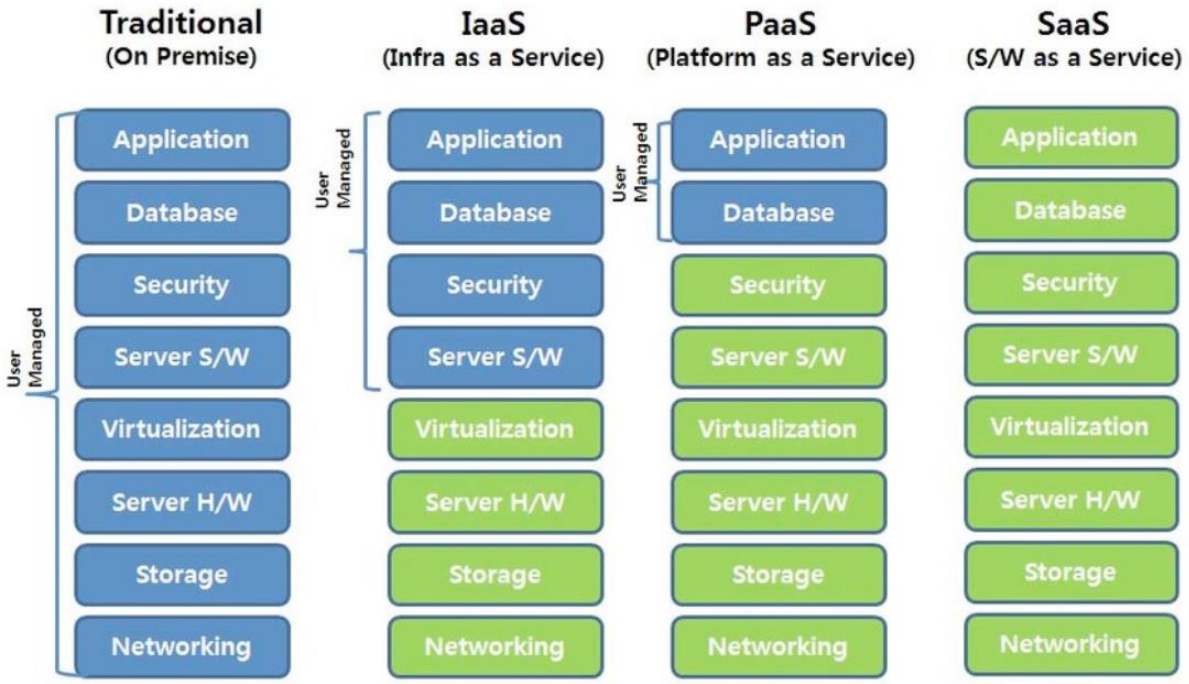

Рисунок 1.1 – Структура сервісної моделі хмарних послуг

Хмарні технології мають значний вплив на бізнес, забезпечуючи компаніям ряд переваг:

1. Зниження витрат - завдяки хмарним технологіям компанії можуть знизити витрати на IT-інфраструктуру, оплату за використання ресурсів та забезпечення підтримки та обслуговування.

2. Гнучкість та масштабованість — хмарні ресурси дозволяють компаніям легко масштабувати свою інфраструктуру відповідно до потреб, реагуючи на зміни навантаження та бізнес-потреб.

3. Доступність та надій — хмарні рішення забезпечують високу доступність та надійність, що дозволяє компаніям підтримувати безперебійну роботу своїх додатків та сервісів, незалежно від місцезнаходження або часу.

4. Безпека та приватність — багато провайдерів хмарних послуг розробляють заходи безпеки на рівні даних, застосунків та інфраструктури, що допомагає компаніям забезпечити безпеку та приватність своїх даних.

5. Спрощення IT-управління — хмарні технології спрощують управління IT-інфраструктурою, дозволяючи компаніям зосередитися на

своєму основному бізнесі, замість підтримки та обслуговування традиційних IT-ресурсів.

6. Швидкість інновацій — завдяки хмарним технологіям компанії можуть швидко розробляти, тестувати та впроваджувати нові рішення, сприяючи інноваційному розвитку та конкурентоспроможності на ринку.

Переходячи від загального розуміння хмарних технологій до їх застосування в AWS, можна зазначити, що цей провайдер надає комплексний набір інструментів і сервісів, що допомагають компаніям ефективно адаптуватися до викликів сучасного бізнесу. Комбінація основних хмарних моделей та сервісних моделей хмарних послуг AWS дозволяє організаціям вибрати найбільш відповідний підхід до розробки, впровадження та управління своїми IT-рішеннями, забезпечуючи максимальну гнучкість та ефективність.

Зокрема, AWS пропонує широкий спектр сервісів, які сприяють швидкому розвитку інновацій, таких як AWS Lambda для створення безсерверних додатків, Amazon SageMaker для розробки та навчання моделей машинного навчання, а також AWS IoT для розробки рішень інтернету речей. Використання цих та інших AWS сервісів дозволяє компаніям прискорити розробку та впровадження нових продуктів та послуг, підвищуючи свою конкурентоспроможність на ринку та забезпечуючи зростання бізнесу.

1.3. Аналіз моделей ціноутворення хмарних послуг

Аналіз моделей ціноутворення хмарних послуг є дійсно важливим кроком для компаній, які планують використовувати хмарні ресурси. Розглянемо основні моделі ціноутворення, які використовуються провайдерами хмарних послуг, зокрема AWS, і дослідимо їх застосування на практиці.

1. Pay-as-you-go (Платить за використання) — Модель pay-as-you-go передбачає оплату за фактично використані ресурси без необхідності

попередньої покупки або резервування. Користувачі сплачують вартість ресурсів на основі їх фактичного використання, яке може бути обраховане за годинами, трафіком або іншими параметрами. Ця модель надає гнучкість та швидкий доступ до ресурсів, що є корисним для компаній з непередбачуваним навантаженням.[2]

2. Зарезервовані ресурси (Reserved Instances) — Модель зарезервованих ресурсів передбачає укладення договору на використання ресурсів протягом певного терміну з фіксованою оплатою. Зарезервовані ресурси дозволяють компаніям знизити витрати на хмарні ресурси завдяки знижкам, які надаються за довготривалі зобов'язання. Ця модель особливо підходить для компаній з стабільним та передбачуваним навантаженням, оскільки вона дозволяє значно знизити витрати.[2]
3. Відповідно до потреби (On-demand) — Модель відповідно до потреби надає користувачам миттєвий доступ до хмарних ресурсів без необхідності попереднього планування або резервування. Хоча ця модель може мати вищу вартість порівняно з іншими моделями ціноутворення, вона надає гнучкість та швидкість дій при необхідності миттєвого доступу до ресурсів. Це особливо корисно для компаній з непередбачуваним навантаженням або тимчасовими потребами, оскільки вони можуть використовувати ресурси тільки тоді, коли це потрібно, і сплачувати лише за використані години або обсяги.[2]
4. Знижки за відданість (Savings Plans) — Модель знижок за відданість дозволяє користувачам зекономити на витратах за рахунок попередньої оплати за використання ресурсів на певний період часу. AWS пропонує різновиди знижок за відданість, такі як Compute Savings Plans та EC2 Instance Savings Plans. Ці знижки дозволяють отримувати значні знижки від стандартної вартості, що є корисним для компаній зі стабільним та довготривалим навантаженням.[2]
5. Безкоштовний рівень (Free Tier) — Модель безкоштовного рівня дозволяє користувачам використовувати певний обсяг ресурсів

безкоштовно протягом обмеженого періоду, зазвичай це 12 місяців. AWS пропонує безкоштовний рівень для деяких своїх послуг, що дозволяє користувачам ознайомитись з хмарними технологіями та розробляти свої додатки без значних фінансових витрат.[2]

6. Маркетплейс (AWS Marketplace) — AWS Marketplace є онлайн-магазином, де користувачі можуть вибирати з багатого вибору сторонніх програмних продуктів та сервісів, розроблених спеціально для AWS. Користувачі можуть вибирати сервіси, що пропонуються за вартість або безкоштовно, і використовувати їх у своїх проектах, сплачуючи за них відповідно до обраної моделі.[2]

У рамках хмарного середовища AWS також існує інструмент AWS Pricing Calculator. Цей сервис надає можливість оцінити вартість використання різних послуг та ресурсів, що пропонуються AWS, та зрозуміти, як впливають на бюджет компанії. Завдяки AWS Pricing Calculator компанії можуть розрахувати приблизну вартість використання послуг та ресурсів AWS заздалегідь, що допомагає планувати бюджет та зробити обґрунтовані рішення щодо використання хмарних послуг. Крім того, сервіс дозволяє порівняти різні варіанти ціноутворення та знайти оптимальний варіант для конкретних потреб компанії, приклад розрахунку на рис 1.2.

Отже враховуючи конкретні потреби та характеристики вашої компанії, ви можете вибрати найбільш підходящу модель ціноутворення для ефективного використання хмарних послуг.

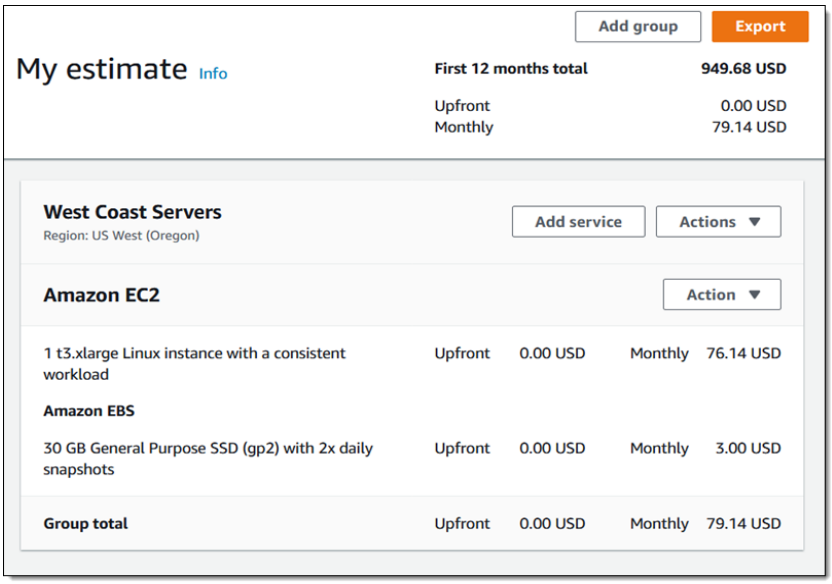

Рисунок 1.2 – Приклад виконаного розрахунку використовуючи AWS Pricing Calculator

1.4. Обгрунтування вибору Jenkins як сервера CI/CD і його актуальність на ринку.

CI (Continuous Integration) і CD (Continuous Deployment/Delivery) є методологіями розробки програмного забезпечення, які спрямовані на автоматизацію процесу збірки, тестування та розгортання програмного коду. CI включає регулярне інтегрування коду з різних розробників в спільний репозиторій. Після кожного коміту в репозиторій, CI-система автоматично збирає та тестує код, переконуюсь, що він інтегрується правильно та не порушує функціональність додатку. CD розширює CI, автоматизуючи процес розгортання програмного коду на середовище продакшн. Це означає, що після успішного проходження тестів код автоматично розгортається на продакшн серверах, готовий до використання користувачами.[3]

CI/CD сервер виконує ці процеси автоматично і забезпечує зручність управління та контролю над ними та. Jenkins є одним з найпопулярніших CI/CD-серверів. Він написаний на мові програмування Java та має широкий набір плагінів та розширень, які дозволяють налаштовувати його під потреби проекту.

Основна причина вибору Jenkins як об'єкта дослідження полягає у його популярності та широкому використанні в галузі розробки програмного забезпечення. Його можливості та особливості роботи можуть бути представниками типових проблем, з якими стикаються багато проектів, що використовують схожі технології. Jenkins може працювати з великими обсягами даних та запитів, що робить його схожим на багато інших систем, які також мають високу навантаженість та потребують оптимізації для забезпечення швидкої та ефективної роботи хмарних ресурсів на яких він працює.

Дослідження та оптимізація Jenkins можуть надати цінні практичні результати та рекомендації для вирішення проблем ефективного використання CI/CD-систем. Крім того, актуальність Jenkins на ринку підтверджується його широким сприйняттям та використанням у різних проектах та організаціях. Він має велику спільноту користувачів та розробників, що забезпечує постійну підтримку, розвиток та покращення функціональності системи. Оптимізована робота Jenkins може призвести до збільшення швидкості, надійності та якості релізів, а також зниження зусиль та витрат, пов'язаних з розробкою та управлінням IT-інфраструктурою та заощадити ці витрати з подальшим їх використанням.

Також ми виділимо основні аргументи, які обґрунтовують вибір Jenkins як об'єкта дослідження для нашої роботи:

1. Поширеність проблем: Jenkins, як популярний CI/CD сервер, використовується великою кількістю проектів та організацій. Таким чином, його проблеми та виклики можуть бути репрезентативними для багатьох інших систем, що базуються на схожих технологіях.
2. Технологічні особливості: Jenkins реалізований на мові програмування Java та використовує велику кількість додаткових модулів та плагінів. Це робить його подібним до багатьох інших додатків, які також базуються на Java та мають аналогічні технічні особливості.

3. Масштабованість: Jenkins може працювати з великими обсягами даних та запитів, що робить його схожим на багато інших систем, які також мають високу навантаженість та потребують оптимізації для забезпечення швидкої та ефективної роботи.

2. ТЕХНОЛОГІЇ АВТОМАТИЗАЦІЇ ТА УПРАВЛІННЯ ІНФРАСТРУКТУРОЮ

2.1 Загальний опис технології Kubernetes та його розширень, та обґрунтування актуальності його використання

Kubernetes - це відкрита та розширювана платформа з управління контейнеризованими робочими навантаженнями та сервісами. Вона надає зручність у декларативному налаштуванні та автоматизації. Kubernetes має широко розповсюджену та швидко зростаючу екосистему з доступними сервісами, підтримкою та інструментами.[4]

Назва "Kubernetes" походить від грецького слова, що означає "рулевий" або "штурман". Google відкрив вихідний код Kubernetes у 2014 році. Ця платформа ґрунтується на десятирічному досвіді Google у роботі з масштабними робочими навантаженнями, поєднуючи найкращі практики та ідеї з відкритою спільнотою.[4]

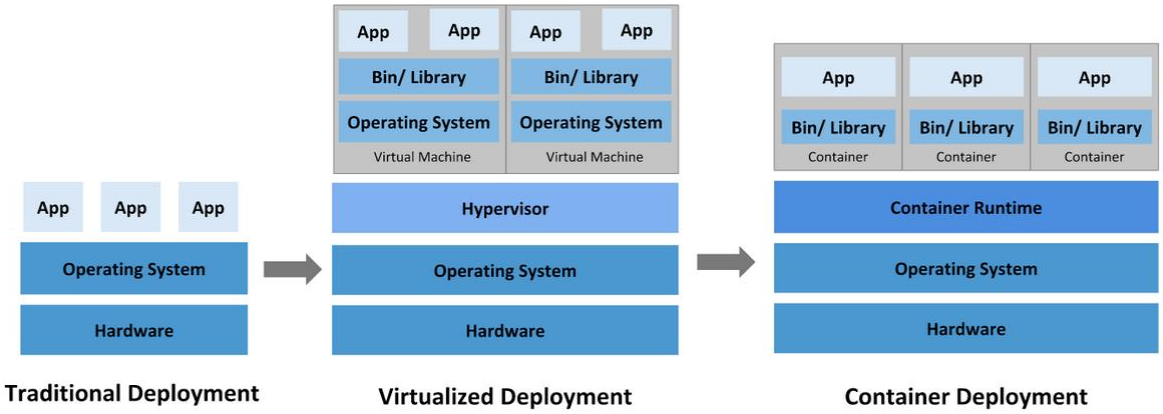

Рисунок 2.1 – Порівняння розгортання додатка на віртуальній машині з контейнеризацією.

Ера розгортання контейнерів народилася як відповідь на потребу у більш ефективному та гнучкому використанні ресурсів. Контейнери дозволяють упаковувати та виконувати програмні застосунки з усіма їх

залежностями у легкому та стандартизованому середовищі. Ключовою технологією у сфері контейнеризації є Docker.[4]

У контейнеризованому середовищі кожен застосунок запускається у власному контейнері, який містить ізольовану копію операційної системи та всі необхідні компоненти для його роботи. Контейнери ділять ресурси фізичного сервера та працюють побічно один з одним, забезпечуючи безпеку та ізоляцію між застосунками. Крім того, контейнери легко переміщуються між різними середовищами, включаючи хмарні платформи та різні операційні системи, завдяки своїй портативності та стандартизованому формату.

Одна з головних переваг контейнерів полягає у швидкому та легкому розгортанні. Контейнери можуть бути запущені всього за кілька секунд, порівняно зі значно більшим часом, необхідним для віртуальних машин. Вони також забезпечують гнучкість масштабування, дозволяючи легко збільшувати або зменшувати кількість контейнерів в залежності від потреб застосунків. Крім того, контейнери дозволяють ефективніше використовувати ресурси сервера, оскільки вони не потребують окремої операційної системи для кожного застосунку, як це відбувається у віртуалізації.[4]

Застосування контейнерів стає все більш популярним у сфері розробки програмного забезпечення та IT-інфраструктури. Ось кілька причин, чому контейнери є вигідним вибором:

1. Портативність — Контейнери можуть бути легко перенесені між різними середовищами, такими як розробка, тестування та виробництво, без змін у програмному забезпеченні. Це дозволяє швидко розгортати та масштабувати застосунки, забезпечуючи єдинообразність середовищ у всьому життєвому циклі розробки.[4]
2. Швидкість розгортання та масштабування — Запуск контейнерів відбувається майже миттєво, що дозволяє швидко реагувати на змінні потреби у ресурсах та масштабувати застосунки без перебудови всієї інфраструктури. Ви можете легко змінювати кількість контейнерів або

навіть автоматично масштабувати їх, використовуючи оркестратори контейнерів, такі як Kubernetes.

3. Ефективне використання ресурсів — Контейнери дозволяють більш ефективно використовувати ресурси сервера, оскільки кілька контейнерів можуть спільно використовувати одну операційну систему та ядро. Вони мають менше накладних витрат, порівняно з віртуальними машинами, оскільки не потребують повного емуляційного середовища.
4. Гнучкість та ізоляція — Кожен контейнер є ізольованим середовищем, що дозволяє запускати та управляти застосунками з різними залежностями та версіями, не впливаючи на решту системи.
5. Легке управління — Контейнери дозволяють автоматизувати процеси розгортання, масштабування та керування застосунками. Інструменти для управління контейнерами, такі як Docker або Kubernetes, надають широкий набір функцій, включаючи автоматичне моніторинг, балансування навантаження та відновлення в разі збоїв.
6. Економія ресурсів — Завдяки масштабованості та ефективному використанню ресурсів, контейнери дозволяють економити фізичні сервери та зменшувати витрати на обладнання. Організації можуть скоротити свої інфраструктурні витрати, запускаючи більше застосунків на меншій кількості серверів.
7. Мікросервісна архітектура — Контейнери добре підходять для реалізації мікросервісної архітектури, де застосунок розбивається на невеликі та незалежні компоненти, що працюють у власних контейнерах. Це полегшує розробку, тестування та масштабування складних систем, а також дозволяє швидше впровадження оновлень та зменшує вплив на весь застосунок у разі збоїв або проблем.

Загалом, контейнеризація надає багато переваг у порівнянні з традиційними методами розгортання. Вона стає стандартом у розробці та управлінні програмними застосунками, дозволяючи організаціям ефективно

використовувати ресурси, забезпечувати безпеку та масштабованість, а також прискорювати процес розробки та впровадження нових застосунків.

Kubernetes складається з декількох ключових компонентів див рис 2.2, які спільно працюють для керування та оркестрації контейнеризованих застосунків.

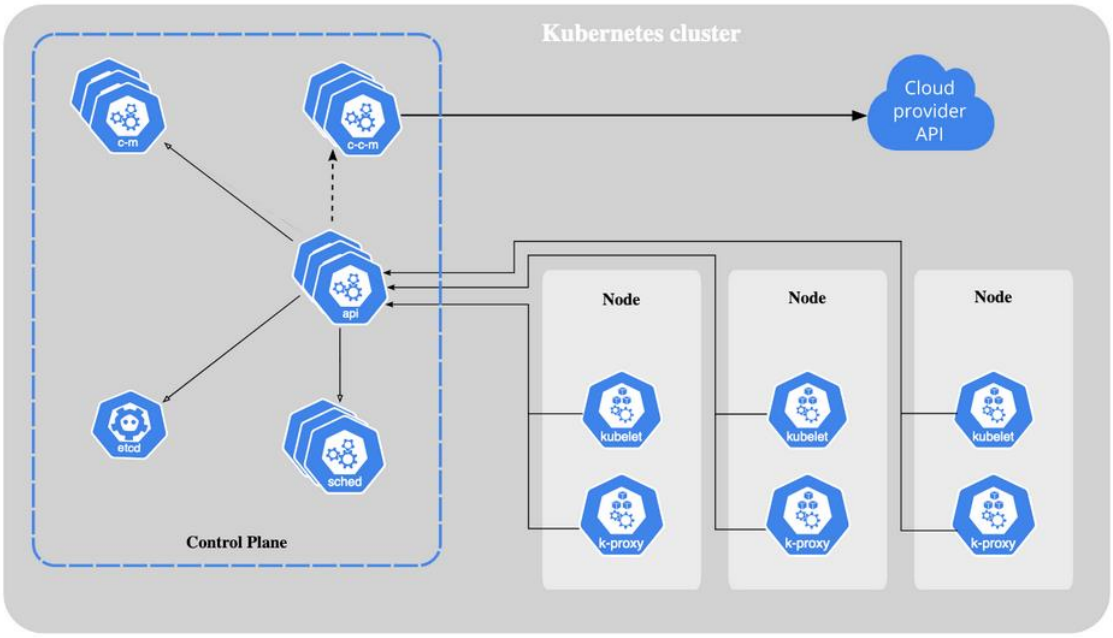

Рисунок 2.2 – Спрощена архітектура типового кластеру Kubernetes[5]

Основні компоненти Kubernetes включають:

1. Master-компоненти:
    a. kube-apiserver: Публікує API Kubernetes та керує всією взаємодією з кластером.
    b. kube-controller-manager: Відповідає за контрольні засоби, такі як контролери реплікації, контролери вузлів та інші.
    c. kube-scheduler: Відповідає за призначення (схематизацію) подій (завдань) на вузли кластеру.[5]

2. Node-компоненти:

    a. kubelet: Керує і контролює кожний вузол кластеру і взаємодіє з майстер-компонентами.

    b. kube-proxy: Забезпечує мережеве проксі для з'єднання засобів між вузлами та подіями (завданнями).[5]

3. etcd: Розподілена ключ-значення база даних, в якій зберігається стан кластеру Kubernetes. Вона забезпечує надійне зберігання даних та синхронізацію між майстер-компонентами.

Важливо також враховувати, що конкретні системні вимоги та характеристики Kubernetes можуть залежати від версії Kubernetes, використовуваного хмарного провайдера та налаштувань кластеру. Рекомендується детально вивчити документацію та рекомендації Kubernetes для встановлення та конфігурації кластеру згідно з вашими потребами.

Приклад системних вимог для встановлення Kubernetes, який можна знайти в офіційній документації Kubernetes:

1. CPU та пам'ять — Рекомендовано мати не менше 2 ядр CPU та 2 ГБ оперативної пам'яті на кожний вузол кластеру. Для кластерів більшого розміру рекомендується більша кількість ресурсів, наприклад, 4 ядра CPU та 8 ГБ оперативної пам'яті на вузол.[4]
2. Дисковий простір — Рекомендовано мати не менше 20 ГБ вільного дискового простору на кожному вузлі кластеру для збереження контейнерів, конфігураційних файлів та журналів.[4]
3. Мережеві вимоги — Кожен вузол кластеру повинен мати доступ до мережі для комунікації з іншими вузлами та зовнішніми ресурсами. Для забезпечення комунікації між компонентами кластеру потрібна наявність мережевих з'єднань між вузлами та підтримка внутрішньокластерних мереж (наприклад, Virtual Private Cloud - VPC).[4]

4. Підтримка ОС — Kubernetes підтримує різні операційні системи, зокрема Linux (зокрема, Ubuntu, CentOS, Red Hat, Fedora), Windows і macOS. Рекомендується встановлювати на надійні та підтримувані версії ОС.[4]

2.1.1. Загальний опис розширення Karpenter для Kubernetes

Karpenter - це вільно поширюваний проект відкритого коду, який був створений командою Amazon Web Services (AWS). Він представляє собою високоефективний менеджер масштабування ресурсів для Kubernetes, що забезпечує більш гнучке та ефективне використання ресурсів, ніж традиційні менеджери масштабування в Kubernetes.[7]

Ключовою особливістю Karpenter є його здатність автоматично налаштовувати та оптимізувати розміщення подів на основі актуальних ресурсів і потреб. Він також може автоматично пристосовувати кількість робочих вузлів у кластері Kubernetes, забезпечуючи найбільш ефективне використання ресурсів.[7]

Відмінність Karpenter від інших рішень для автоматичного масштабування полягає в його способі оптимізації використання ресурсів. Він не просто масштабує вузли на основі загального навантаження, але також враховує багато різних факторів, включаючи типи завдань, які виконуються, їхні потреби в ресурсах, час дії, політики масштабування та багато інших.

Завдяки цьому Karpenter забезпечує дуже високу ефективність ресурсів, знижує витрати на інфраструктуру та забезпечує високу продуктивність застосунків, які працюють в Kubernetes.

Karpenter встановлюються у рабочий кластер як окремий контроллер з додатковими компонентами:

1. Provisioner — це основний компонент Karpenter. Він відповідає за створення та управління вузлами в кластері Kubernetes. Provisioner

моніторить навантаження на кластер і вирішує, коли потрібно додати або видалити вузли.[7]

2. Metrics Server — Karpenter використовує Metrics Server для збору даних про використання ресурсів в кластері. Ці дані використовуються для визначення поточного навантаження на кластер і визначення, коли потрібно масштабувати вузли.[2]
3. Controller — Controller в Karpenter відповідає за виконання дій масштабування на основі даних, отриманих від Provisioner та Metrics Server. Він виконує рішення про масштабування вузлів, створює або видаляє вузли за потреби.[7]
4. Cloud Provider Integration — Karpenter має інтеграцію з різними хмарними провайдерами, включаючи AWS, що дозволяє йому автоматично використовувати ресурси хмари для масштабування кластерів.[7]

В цілому, робота Karpenter полягає в постійному моніторингу стану кластеру Kubernetes, аналізі навантаження та потреби в ресурсах, та відповідному масштабуванні робочих вузлів для забезпечення оптимального використання ресурсів та високої продуктивності застосунків.

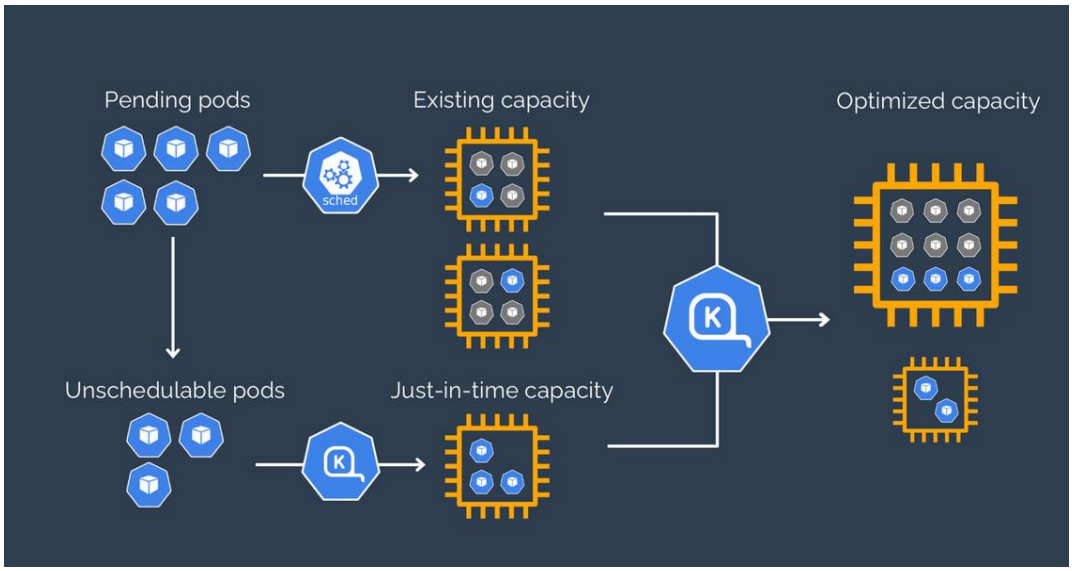

Рисунок 2.3 – Життєвий цикл роботи Karpenter[7]

2.1.2. Загальний опис розширення KEDA або Event-Driven Autoscaling для Kubernetes

KEDA, або Kubernetes Event-Driven Autoscaling, також є додатком для Kubernetes, яке забезпечує швидке та гнучке автоматичне масштабування за подіями.

Основні компоненти KEDA:

1. Scalers — У KEDA є різні "scalers", кожен з яких відповідає за слухання конкретного типу події. Ці події можуть надходити від різних джерел, включаючи бази даних, черги повідомлень, потоки даних, сервіси хмари та багато інших.[8]
2. ScaledObject — Користувач KEDA визначає ScaledObject у своєму кластері Kubernetes. ScaledObject описує, який застосунок масштабувати, на якому scaler'і слідкувати за подіями, а також параметри масштабування, такі як мінімальна та максимальна кількість реплік.[8]
3. Автоматичне масштабування — Коли scaler виявляє подію, KEDA автоматично масштабує репліки застосунку вгору або вниз, залежно від навантаження. Це дозволяє застосункам відповідати на піки навантаження та ефективно використовувати ресурси, коли навантаження низьке.[8]
4. Сумісність з Kubernetes HPA — KEDA розширює функціональність автоматичного масштабування Kubernetes, але також сумісна з вбудованою системою автоматичного масштабування Kubernetes, відомою як HPA (Horizontal Pod Autoscaler). Це означає, що ви можете використовувати KEDA і HPA разом в одному кластері.[8]

В цілому, KEDA - це потужний інструмент для автоматичного масштабування застосунків в Kubernetes, який реагує на реальні події, що відбуваються в вашій системі, що дозволяє вашим застосункам масштабуватися відповідно до ваших реальних потреб.[8]

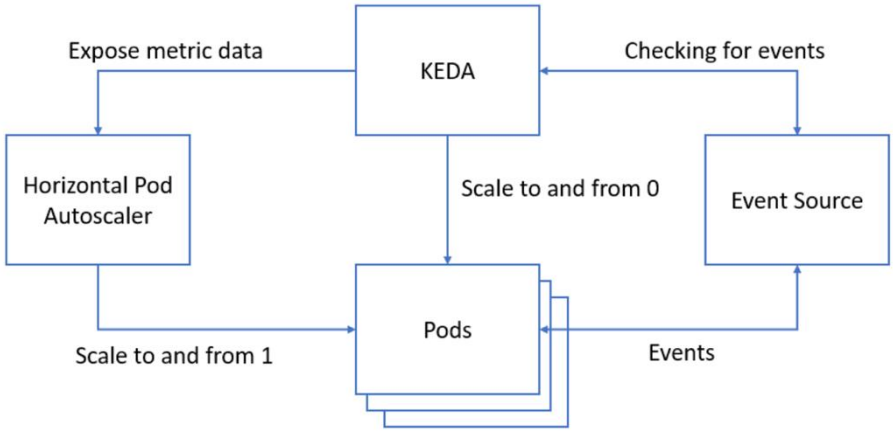

Рисунок 2.4 – Життєвий цикл роботи KEDA

Вибір технологій Kubernetes, Karpenter та KEDA для даного дослідження є стратегічним рішенням, зумовленим їх унікальними можливостями у вирішенні задач оптимізації використання хмарних ресурсів.

Kubernetes дозволяє оркеструвати та управляти розподіленими застосунками в масштабі, що є критичним для ефективного використання хмарних ресурсів. Крім того, Kubernetes є стандартом де-факто у світі контейнерів, що робить його вибір логічним для більшості сучасних хмарних застосунків.

Karpenter та KEDA доповнюють можливості Kubernetes, пропонуючи більш динамічне та подієве масштабування. Ці технології допомагають забезпечити, що ресурси хмари використовуються оптимально, забезпечуючи високу продуктивність при мінімальних витратах.

Вибір технологій Kubernetes, Karpenter, KEDA для розгортання Jenkins є стратегічним і обґрунтованим їх унікальними можливостями у вирішенні задач оптимізації використання хмарних ресурсів.

2.2 Загальний опис основних сервісів AWS та їх можливостей для оптимізації витрат

Amazon Web Services (AWS) надає найширший спектр хмарних сервісів, і вони дозволяють вирішувати величезний діапазон завдань, включаючи питання оптимізації використання ресурсів. Надалі у практичній частині особливу увагу ми будемо приділяти сервісам Amazon EKS (Elastic Kubernetes Service) та Amazon CloudWatch тому зараз розглянемо основні їх визначення.

Amazon EKS — це повністю керований сервіс Kubernetes, який дозволяє легко запускати, масштабувати та управляти контейнеризованими застосунками в хмарі. Використання EKS дає нам можливість зосередитися на розробці застосунків без турбот про налаштування, розгортання та управління Kubernetes - все це бере на себе Amazon. Karpenter, як інструмент автоматичного масштабування, ідеально підходить для інтеграції з EKS, оскільки він може динамічно масштабувати кількість вузлів Kubernetes на основі поточного навантаження, що допомагає оптимізувати витрати.[10]

EKS дозволяє запускати контейнерні застосунки за допомогою HTTP та HTTPS маршрутизації за допомогою Application Load Balancer (ALB), Network Load Balancer (NLB) або Classic Load Balancer (CLB).

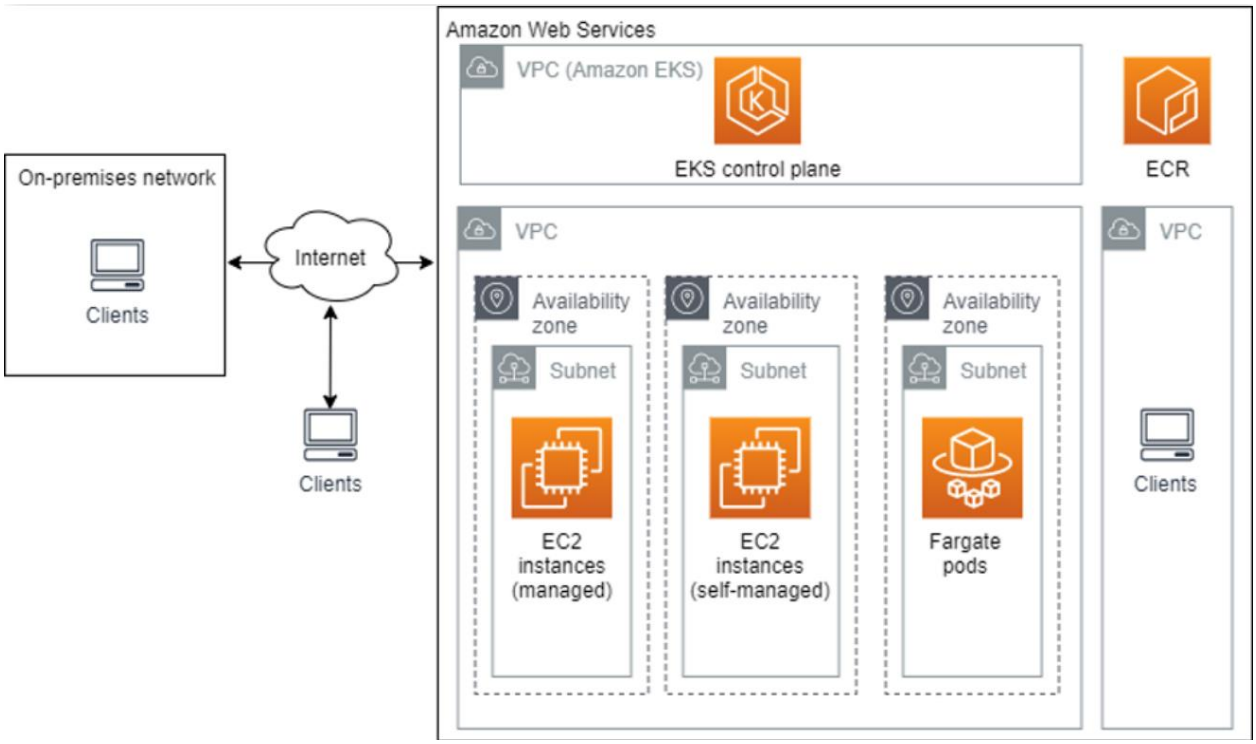

Рисунок 2.5 — Спрошена архітектура типового кластеру EKS

Сервіс складається з наступних основних компонентів:

1. Control Plane — EKS автоматично управляє доступністю та масштабуванням control plane Kubernetes на кластері. AWS забезпечує високий рівень доступності та надійності для control plane, розподіляючи його по різним Availability Zones.[10]
2. Worker Nodes — EKS підтримує EC2 інстанси та AWS Fargate для запуску контейнерів. Ви можете вибрати оптимальний тип інстансу в залежності від потреб вашого застосунку.[10]
3. Data Plane — На вузлах worker nodes виконуються ваші застосунки. Ви можете використовувати EKS для запуску, масштабування та управління контейнерами.[10]
4. Managed Node Group — Managed Node Groups автоматизують процеси розгортання та обновлення вузлів. Ви можете визначити кількість вузлів, типи інстансів.[10]

В цілому, EKS - це потужний та гнучкий сервіс, який вбирає в себе всі переваги Kubernetes, додаючи до них масштабованість, надійність та інтеграцію з іншими сервісами AWS.

Amazon CloudWatch — це служба моніторингу ресурсів і застосунків AWS, яка дозволяє збирати та відстежувати метрики, збирати та переглядати протоколи, встановлювати сповіщення та автоматично реагувати на зміни в ваших AWS-ресурсах. CloudWatch дозволяє вам отримувати повний обсяг даних про використання ресурсів, що надає можливість виявити та вирішити проблеми з продуктивністю та відповісти на зміни у використанні ресурсів для оптимізації витрат.[11]

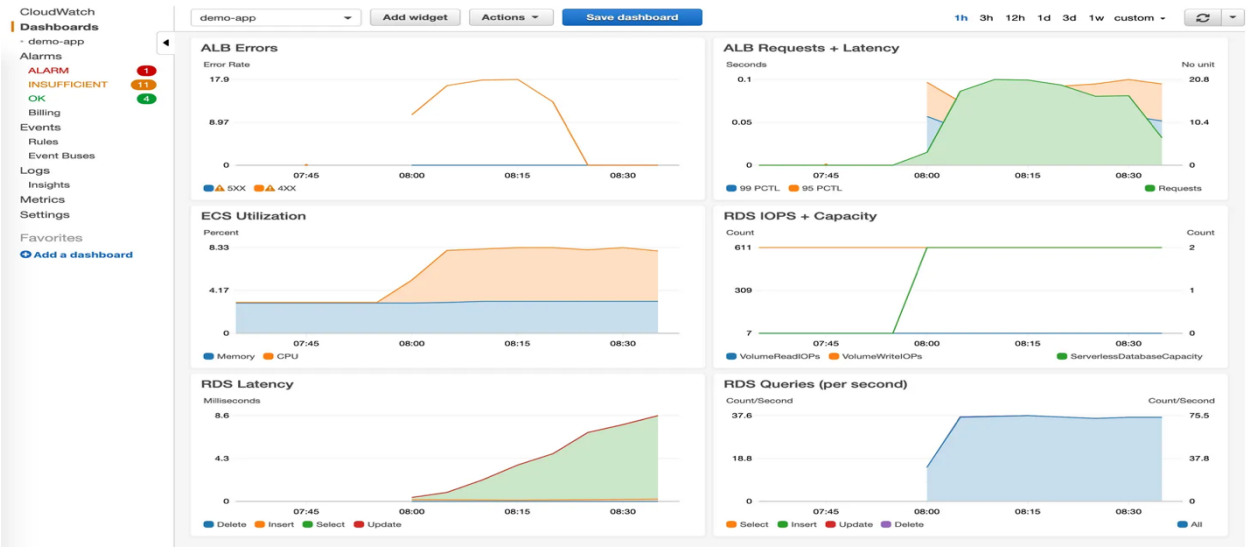

Рисунок 2.6 — Приклад налаштованні CloudWatch Dashboard

Інтеграція цих служб AWS з Kubernetes, Karpenter, KEDA та Jenkins створює потужну екосистему, що здатна автоматично реагувати на зміни навантаження, оптимізувати використання ресурсів та зменшувати витрати. AWS EKS обеспечує високодоступність та безпеку для ваших застосунків, автоматично управляючи розподіленням трафіку та масштабуванням ваших сервісів. В той же час, Karpenter динамічно розподіляє контейнери та вузли з урахуванням наявних ресурсів, що дозволяє оптимізувати використання ресурсів і зменшити витрати. З іншого боку, KEDA дозволяє Kubernetes автоматично масштабувати застосунки на основі подій, таких як черги повідомлень, вхідні потоки, розподілені потоки тощо. Це забезпечує гнучкість та ефективність, зменшуючи необхідність передбачення навантаження та вручну настройку ресурсів.

Нарешті, Amazon CloudWatch дозволяє вам відстежувати використання ресурсів, встановлювати сповіщення і автоматично реагувати на зміни в AWS-ресурсах. Таким чином, ви можете проактивно відстежувати стан вашої інфраструктури, зменшувати витрати та підвищувати ефективність застосунків.

Отже, використання AWS EKS, Karpenter, KEDA, Jenkins та Amazon CloudWatch в сукупності надає нам потужний інструментарій для

автоматичного масштабування, оптимізації ресурсів та моніторингу застосунків. Використання цих технологій дозволяє нам реагувати на зміни навантаження в режимі реального часу, зменшувати витрати та підвищувати продуктивність системи неперервної інтеграції та доставки (CI/CD), яка базується на Jenkins.

2.4 Загальний опис Terraform як інструменту управління інфраструктурою

Terraform є інструментом відкритого коду для управління та оркестрації інфраструктурою як кодом (IaC). Він випущений компанією HashiCorp і дозволяє користувачам визначати та надавати інфраструктуру в різних сервісах хмарних провайдерів з використанням декларативного мови програмування.[6]

Основні характеристики Terraform включають:

1. Підтримка багатьох провайдерів — Terraform підтримує різні хмарні платформи, включаючи AWS, Google Cloud, Azure, і багато інших, також підтримуються такі платформи як GitHub, GitLab і інші.
2. Інфраструктура як код (IaC) — Terraform дозволяє користувачам визначати та надавати інфраструктуру з використанням декларативного коду, що забезпечує консистентність і повторюваність.
3. Планування та перегляд змін — Terraform вміє показувати зміни, які будуть застосовані до інфраструктури, перед тим, як вони будуть виконані, що дозволяє користувачам переглядати та затверджувати зміни.
4. Модульна архітектура — Terraform підтримує модулі, які дозволяють користувачам повторно використовувати та комбінувати конфігурації, що спрощує управління та розгортання складної інфраструктури.

5. Управління станом — Terraform веде облік поточного стану інфраструктури і здійснює зміни на основі відмінностей між поточним станом та бажаним.

За допомогою Terraform, ми можемо автоматизувати розгортання та управління інфраструктурою на AWS, включаючи EKS. Terraform пропонує модулі для створення EKS кластерів, налаштування

Terraform має визначений життєвий цикл для керування ресурсами, що включає такі етапи:

1. Ініціалізація — На цьому етапі Terraform ініціалізує робочий каталог, в якому зберігаються конфігурації Terraform. Він завантажує всі необхідні провайдери та підготовку їх до використання.
2. Планування — Terraform створює план, в якому відображає, які ресурси будуть створені, змінені або видалені. Це дає можливість користувачеві перевірити, чи відповідають заплановані дії його очікуванням, перед тим як вносити будь-які зміни в реальну інфраструктуру.
3. Застосування — Після затвердження плану Terraform застосовує зміни, створюючи, модифікуючи або видаляючи ресурси відповідно до плану.

Приклад базового коду Terraform для створення EC2 інстансу в AWS:

```
1  provider "aws" {
2    region = "us-west-2"
3  }
4
5  resource "aws_instance" "example" {
6    ami           = "ami-0c94855ba95c574c8"
7    instance_type = "t2.micro"
8
9    tags = {
10   Name = "example-instance"
11   }
12 }
```

Рисунок 2.7 - Приклад коду для розгортання EC2

В цьому прикладі ми використовуємо провайдера AWS для роботи з AWS. Ресурс aws_instance вказує на те, що ми хочемо створити інстанс EC2.

Властивості ami та instance_type вказують на AMI, яку ми хочемо використовувати для нашого інстансу, та тип інстансу, який ми хочемо створити. tags використовуються для додавання метаданих до нашого інстансу.

Terraform відіграє важливу роль в нашому дослідженні, оскільки він забезпечує автоматизоване та систематичне управління інфраструктурою в AWS. Його використання дозволяє нам створювати, модифікувати та видаляти ресурси відповідно до наших потреб, що робить процес керування інфраструктурою більш простим, ефективним та надійним.

Використання Terraform в контексті нашого дослідження дозволяє нам використовувати та оптимізувати хмарні ресурси більш ефективно. Завдяки його можливостям ми можемо автоматизувати процес розгортання Jenkins на EKS, а також інтеграцію з Karpenter та KEDA. Це значно спрощує управління навантаженням та ресурсами, а також оптимізує використання хмарних ресурсів, що в кінцевому результаті може призвести до зменшення витрат.

3. ПРАКТИЧНА ЧАСТИНА: РОЗРОБКА ТА РЕАЛІЗАЦІЯ СИСТЕМИ

3.1 Використання Terraform для налаштування EKS та створення Kubernetes кластеру

У цій роботі будуть наведені приклади коду рис 3.1-3.14,3.15-3.21 для розгортання кластера Amazon Elastic Kubernetes Service (EKS) з використанням Terraform та Helm. Кластер буде розгорнутий у Virtual Private Cloud (VPC) з публічними та приватними підмережами, а також включатиме IAM ролі, Auto Scaling Group (ASG), Load Balancer та Ingress. У кластері буде встановлено два додаткові компоненти: Karpenret та KEDA (Kubernetes Event-driven Autoscaling).

Створення VPC, підсетей і налаштування безпеки Створюємо файл vpc.tf для визначення VPC, підсетей і налаштувань безпеки:

```
resource "aws_vpc" "this" {
  cidr_block           = var.vpc_cidr
  enable_dns_hostnames = true

  tags = {
    Name = var.cluster_name
  }
}

resource "aws_subnet" "private" {
  count = length(var.private_subnet_cidrs)

  cidr_block = var.private_subnet_cidrs[count.index]
  vpc_id      = aws_vpc.this.id

  tags = {
    Name = "${var.cluster_name}-private-${count.index + 1}"
  }
}
```

Рисунок 3.1 - Перша частина коду реалізації проекту на terraform

```
resource "aws_subnet" "public" {
  count = length(var.public_subnet_cidrs)

  cidr_block = var.public_subnet_cidrs[count.index]
  vpc_id      = aws_vpc.this.id

  tags = {
    Name = "${var.cluster_name}-public-${count.index + 1}"
  }
}

resource "aws_security_group" "worker_group_mgmt_one" {
  name_prefix = "worker_group_mgmt_one"
  vpc_id      = aws_vpc.this.id
}

resource "aws_security_group_rule" "worker_group_mgmt_one" {
  security_group_id = aws_security_group.worker_group_mgmt_one.id

  type        = "ingress"
  from_port   = 0
  to_port     = 65535
  protocol    = "tcp"
  cidr_blocks = ["10.0.0.0/8"]
}
```

Рисунок 3.2 - Друга частина коду реалізації проекту на terraform

```
locals {
  cluster_name = var.cluster_name
}

resource "aws_eks_cluster" "this" {
  name     = local.cluster_name
  role_arn = aws_iam_role.eks_cluster.arn

  vpc_config {
    subnet_ids = aws_subnet.private.*.id
  }

  depends_on = [
    aws_security_group_rule.worker_group_mgmt_one,
  ]
}
```

Рисунок 3.3 - Третя частина коду реалізації проекту на terraform

```
resource "aws_iam_role" "eks_cluster" {
  name = "eks-cluster"

  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Action = "sts:AssumeRole"
        Effect = "Allow"
        Principal = {
          Service = "eks.amazonaws.com"
        }
      }
    ]
  })
}

resource "aws_iam_role_policy_attachment" "eks_cluster" {
  policy_arn = "arn:aws:iam::aws:policy/AmazonEKSClusterPolicy"
  role       = aws_iam_role.eks_cluster.name
}
```

Рисунок 3.4 - Четверта частина коду реалізації проекту на terraform

Створюємо файл eks_node_group.tf для визначення групи вузлів EKS на рис 3.5-3.6:

```
resource "aws_eks_node_group" "this" {
  cluster_name    = aws_eks_cluster.this.name
  node_group_name = "${aws_eks_cluster.this.name}-node-group"
  node_role_arn   = aws_iam_role.eks_worker_nodes.arn
  subnet_ids      = aws_subnet.private.*.id

  scaling_config {
    desired_size = var.node_desired_size
    max_size     = var.node_max_size
    min_size     = var.node_min_size
  }

  instance_types = ["t3.medium"]
  disk_size      = 20

  remote_access {
    ec2_ssh_key               = aws_key_pair.deployer.key_name
    source_security_group_ids = [aws_security_group.worker_group_mgmt_one.id
  }
```

Рисунок 3.5 - П'ята частина коду реалізації проекту на terraform

```
  depends_on = [
    aws_iam_role_policy_attachment.eks_worker_nodes,
  ]
}

resource "aws_iam_role" "eks_worker_nodes" {
  name = "eks-worker-nodes"

  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Action = "sts:AssumeRole"
        Effect = "Allow"
        Principal = {
          Service = "ec2.amazonaws.com"
        }
      }
    ]
  })
}
```

Рисунок 3.6 - Шоста частина коду реалізації проекту на terraform

```
resource "aws_iam_role_policy_attachment" "eks_worker_nodes" {
  policy_arn = "arn:aws:iam::aws:policy/AmazonEKSWorkerNodePolicy"
  role       = aws_iam_role.eks_worker_nodes.name
}
```

Рисунок 3.7 - Сьома частина коду реалізації проекту на terraform

Нам також знадобиться helm провайдер і скрипти для розвертання helm чартів Karpenter та KEDA. Створюємо файл helm.tf:

```
provider "helm" {
  kubernetes {
    config_path = "~/.kube/config"
  }
}

resource "helm_release" "karpenret" {
  name       = "karpenret"
  repository = "https://charts.karpenret.io"
  chart      = "karpenret"
}

resource "helm_release" "keda" {
  name       = "keda"
  repository = "https://keda.github.io/charts"
  chart      = "keda"
}
```

Рисунок 3.8 - Восьма частина коду реалізації проекту на terraform

Наш терраформ модуль який ми створимо використовуючи код з рисунків 1-5 являє собою розгортання високодоступного кластера EKS в AWS за допомогою Terraform. В результаті будуть створені наступні ресурси:

1.VPC (Virtual Private Cloud) з публічними та приватними підсетями.

2. IAM роли для управління кластером EKS і узлами робочого вузла.

3. EKS кластер.

4. Група узлов EKS з конфігураціями масштабування.

5. Установка програм Karpenret і KEDA з використанням Helm.

Для розгортання високодоступного кластера додаємо файл variables.tf, що містить усі необхідні змінні:

```
variable "cluster_name" {
  description = "The name of the EKS cluster"
  type        = string
}

variable "vpc_cidr" {
  description = "CIDR block for the VPC"
  type        = string
}

variable "private_subnet_cidrs" {
  description = "List of CIDR blocks for private subnets"
  type        = list(string)
}

variable "public_subnet_cidrs" {
  description = "List of CIDR blocks for public subnets"
  type        = list(string)
}

variable "node_desired_size" {
  description = "Desired number of worker nodes"
  type        = number
}
```

Рисунок 3.9 - Дев’ята частина коду реалізації проекту на terraform

```
variable "node_min_size" {
  description = "Minimum number of worker nodes"
  type        = number
}

variable "node_max_size" {
  description = "Maximum number of worker nodes"
  type        = number
}
```

Рисунок 3.10 - Десята частина коду реалізації проекту на terraform

Створимо файл terraform.tfvars із зразковими значеннями для наших змінних:

```
cluster_name          = "my-eks-cluster"
vpc_cidr              = "10.0.0.0/16"
private_subnet_cidrs  = ["10.0.1.0/24", "10.0.2.0/24", "10.0.3.0/24"]
public_subnet_cidrs   = ["10.0.101.0/24", "10.0.102.0/24", "10.0.103.0/24"]
node_desired_size     = 3
node_min_size         = 1
node_max_size         = 5
```

Рисунок 3.11 - Одинадцята частина коду реалізації проекту на terraform

У цьому прикладі ми створюємо VPC з CIDR-блоком 10.0.0.0/16, три приватні підсети з CIDR-блоками 10.0.1.0/24, 10.0.2.0/24 і 10.0.3.0/24, а також три публічні підсети з CIDR-блоками 10.0.101.0/24, 10.0.102.0/24 і 10.0.103.0/24. Робочі вузли мають бажаний розмір 3, мінімальний розмір 1 і максимальний розмір 5.

Після налаштування всіх файлів виконайте наступні команди для розвертання кластера:

```
terraform init
terraform apply
```

Рисунок 3.12 - Розгортання проекту на terraform

Зверніть увагу, що цей код є базовим прикладом який використовується для пояснення і може потребувати додаткових налаштувань для відповідності вашим вимогам окремого AWS акаунту.

Після розгортання та налаштування ми отримаємо наступну спрошену для розуміння схему інфраструктури проекту рис 3.13:

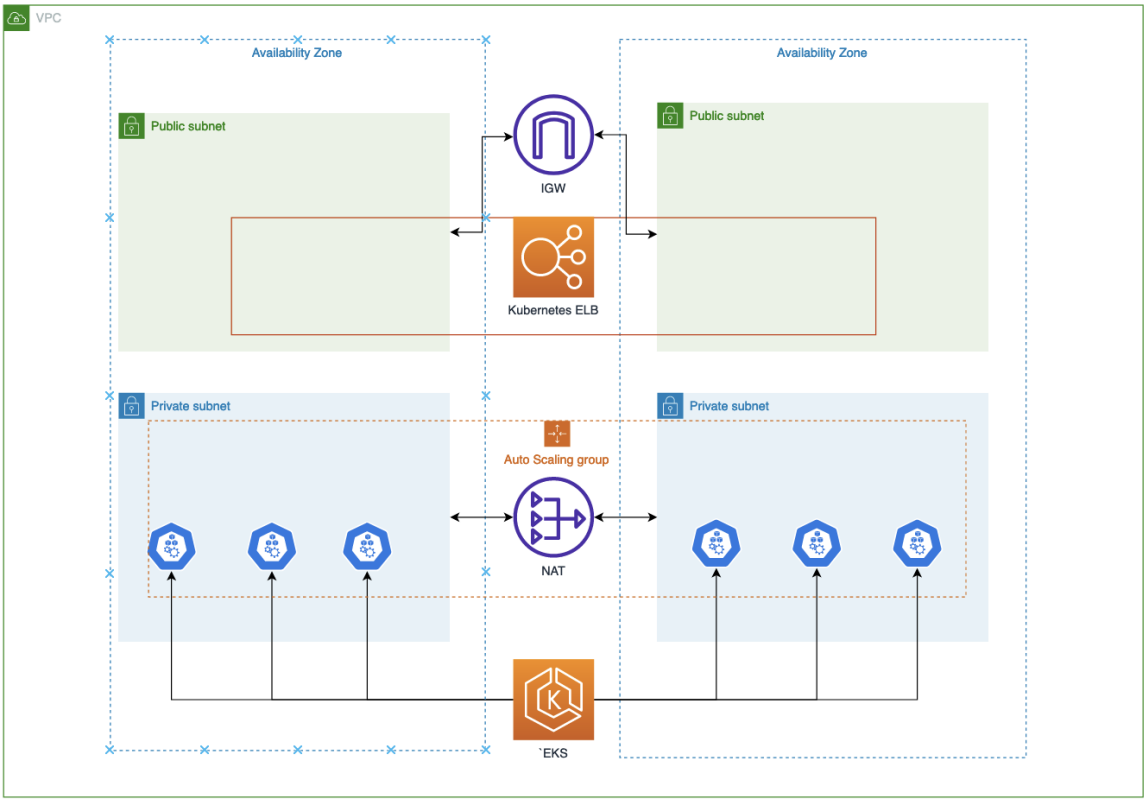

Рисунок 3.13 - Інфраструктура розгортання EKS

3.2 Розгортання Jenkins на EKS

Для встановлення Jenkins у кластері EKS та налаштування його для запуску завдань у Docker у подах EKS ми спочатку повинні встановити Jenkins за допомогою Helm. Потім потрібно налаштувати плагін Kubernetes для Jenkins, який дозволяє Jenkins запускати завдання в подах Kubernetes.

Спочатку додаємо Jenkins у кластер за допомогою Helm. Створимо файл helm_jenkins.tf:

```
resource "helm_release" "jenkins" {
  name       = "jenkins"
  repository = "https://charts.jenkins.io"
  chart      = "jenkins"
  namespace  = "default"

  set {
    name  = "master.serviceType"
    value = "LoadBalancer"
  }
}
```

Рисунок 3.14 - Дванадцята частина коду реалізації проекту на terraform

У цьому прикладі ми використовуємо LoadBalancer як serviceType, що призведе до створення ELB і надають нам загальнодоступну URL для доступу до Jenkins. Тепер додаймо файл output.tf для виведення URL Jenkins:

```
output "jenkins_url" {
  description = "The endpoint for accessing Jenkins"
  value       = "http://${kubernetes_service.jenkins.load_balancer_ingress.0
}
```

Рисунок 3.15 - Тринадцята частина коду реалізації проекту на terraform

Після успішного застосування конфігурації Terraform ви повинні побачити URL-адресу Jenkins у виведенні команди terraform apply.

Також налаштуємо плагін, ви можете використовувати плагін Jenkins Configuration as Code (JCasC) для автоматичного налаштування Jenkins під час розвертання, включаючи налаштування плагіна Kubernetes і створення завдання для тестування.

Спочатку вам потрібно створити конфігураційний файл JCasC. Це YAML-файл, який визначає всі налаштування Jenkins. Нижче наведено приклад конфігурації, яка налаштовує плагін Kubernetes і створює просту задачу freestyle для тестування:

```
jenkins:
  clouds:
    - kubernetes:
        name: "kubernetes"
        serverUrl: "https://kubernetes.default"
        jenkinsUrl: "http://jenkins.jenkins.svc.cluster.local:8080"
        jenkinsTunnel: "jenkins-agent.jenkins.svc.cluster.local:50000"
        connectTimeout: 5
        readTimeout: 15
        containerCapStr: "10"
        maxRequestsPerHostStr: "32"
        retentionTimeout: 5
  jobs:
    - script: >
        job('example') {
          steps {
            shell('echo Hello, Kubernetes!')
          }
        }
```

Рисунок 3.16 - Чотирнадцята частина коду реалізації проекту на terraform

Наступним кроком буде додавання цього конфігураційного файлу на вашій Helm-схемі Jenkins. Це можна зробити, додавши наступні рядки у ваш файл helm_jenkins.tf:

```
resource "helm_release" "jenkins" {
  ...
  set {
    name  = "controller.JCasC.configScripts.test"
    value = file("${path.module}/jenkins.yaml")
  }
}
```

Рисунок 3.17 - П’ятнадцята частина коду реалізації проекту на terraform

Це говорить Helm про включення вмісту файлу jenkins.yaml у конфігурації Jenkins у вигляді скрипта JCasC.

Також зверніть увагу, що наведена вище конфігурація є прикладом який використовується у нашій роботі в окремих ситуаціях плагін може потребувати додаткових налаштувань відповідно до ваших вимог.

3.3 Налаштування CloudWatch для моніторингу.

Amazon CloudWatch надає потужні інструменти для моніторингу та спостереження за вашими ресурсами AWS, включаючи Amazon EKS. CloudWatch збирає метрики операційного рівня, які можуть бути використані для відстеження продуктивності, спостереження за ресурсами для їх роботи та реагування на аномалії.

Amazon EKS генерує безліч корисних показників, які надсилаються в CloudWatch і які ми можемо використовувати для налаштувань оповіщень і моніторингу продуктивності кластера. Перш за все, для роботи з метриками EKS у CloudWatch, нам необхідно встановити агент CloudWatch у нашому кластері. Це можна зробити за допомогою Helm:

```
resource "helm_release" "cloudwatch_agent" {
  name       = "cloudwatch-agent"
  repository = "https://aws.github.io/eks-charts"
  chart      = "aws-cloudwatch-metrics"
  namespace  = "amazon-cloudwatch"
}
```

Рисунок 3.17 - Щіснадцята частина коду реалізації проекту на terraform

Ця діаграма Helm встановлює агент CloudWatch у кластері EKS, який автоматично збиратиме та надсилати метрики в CloudWatch. Тепер ми можемо створити оповіщення в CloudWatch, яке буде активовано при досягненні певного порога. Наприклад, ми можемо налаштувати оповіщення, яке буде активовано, коли рівень використання процесора в кластері досягне 80%.

```
resource "aws_cloudwatch_metric_alarm" "high_cpu_utilization" {
  alarm_name          = "high_cpu_utilization"
  comparison_operator = "GreaterThanOrEqualToThreshold"
  evaluation_periods  = "2"
  metric_name         = "cpu_utilization"
  namespace           = "AWS/EKS"
  period              = "120"
  statistic           = "Average"
  threshold           = "80"
  alarm_description   = "This metric checks cpu utilization"
  alarm_actions       = [aws_sns_topic.cpu_alerts.arn]
  dimensions = {
    ClusterName = var.cluster_name
  }
}

resource "aws_sns_topic" "cpu_alerts" {
  name = "cpu-alerts"
}
```

Рисунок 3.18 - Сімнадцята частина коду реалізації проекту на terraform

Цей код створює оповіщення CloudWatch, яке активується, при середньому використанні ЦП протягом двох періодів на 2 хвилини понад 80%. Коли це відбувається, CloudWatch надсилає сповіщення про SNS тему cpu_alerts.

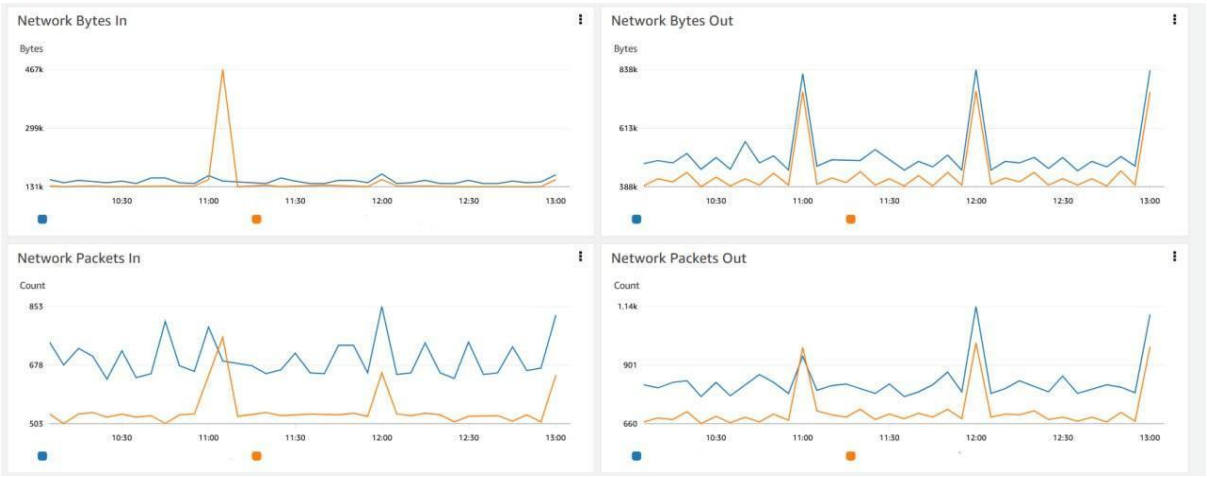

Рисунок 3.19 - Дашборд CloudWatch з моніторингом мережної загрузки на EKS

3.4 Налаштування Karpenter і KEDA для автоматичного масштабування.

Для забезпечення управління ресурсами та автоматичного масштабування в нашій інфраструктурі Kubernetes ми будемо використовувати Karpenter і KEDA.

Karpenter надає можливість автоматичного горизонтального масштабування вузлів Kubernetes на основі вимог робочих навантажень, запускаючи нові екземпляри EC2 за потреби. Крім того, Karpenter може використовувати spot-екземпляри для зниження витрат, підбираючи оптимальний тип экземпляра залежно від вимог робочого навантаження та доступності spot-екземплярів.

KEDA (Kubernetes Event-Driven Autoscaling) надає можливість вертикального автоматичного масштабування на основі подій, змінюючи кількість підів залежно від навантаження або інших подій, таких як кількість елементів у черзі повідомлень.

Для того, щоб Karpenter і KEDA могли працювати, нам потрібно забезпечити їм відповідні ролі IAM. Для цього ми можемо використовувати модуль aws_iam_role в Terraform.

Щодо Jenkins, ми можемо налаштувати його таким чином, щоб він запускав свої завдання в подах Kubernetes. Коли Jenkins буде потребувати більше ресурсів для виконання завдань, він буде запускати нові поди, які будуть вимагати більше ресурсів від вузлів Kubernetes. Це, в свою чергу, призведе до того, що Karpenter буде створювати нові екземпляри EC2 для задоволення цієї потреби.

Для оптимізації витрат ми можемо налаштувати Karpenter на використання spot-екземплярів EC2, а KEDA - на автоматичне масштабування подів униз, коли навантаження зменшується. Наприклад, KEDA може автоматично зменшувати кількість подів Jenkins у вихідні дні, коли навантаження зазвичай менше.

Спочатку вам потрібно створити або доповнити роль IAM для Karpenter, яка дозволяє йому створювати та керувати інстансами EC2:

```
resource "aws_iam_role" "karpenter" {
  name               = "karpenter"
  assume_role_policy = <<EOF
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": "sts:AssumeRole",
      "Principal": {
        "Service": "ec2.amazonaws.com"
      },
      "Effect": "Allow",
      "Sid": ""
    }
  ]
}
EOF
}

resource "aws_iam_role_policy_attachment" "karpenter-AmazonEC2FullAccess" {
  role       = aws_iam_role.karpenter.name
  policy_arn = "arn:aws:iam::aws:policy/AmazonEC2FullAccess"
}
```

Рисунок 3.20 - Вісімнадцята частина коду реалізації проекту на terraform

```
resource "helm_release" "karpenter" {
  name       = "karpenter"
  repository = "https://awslabs.github.io/karpenter/charts"
  chart      = "karpenter"
  namespace  = "kube-system"

  set {
    name  = "serviceAccount.create"
    value = "false"
  }

  set {
    name  = "serviceAccount.name"
    value = "karpenter"
  }

  set {
    name  = "controller.clusterName"
    value = "my-eks-cluster"
  }

  set {
    name  = "controller.clusterEndpoint"
    value = "https://eks.amazonaws.com"
  }
}
```

Рисунок 3.21 - Дев’ятнадцята частина коду реалізації проекту на terraform

Для оптимізації витрат, ми будемо використовувати spot-примірники EC2 для Karpenter та налаштуємо KEDA на автоматичне масштабування подів вниз при зменшенні навантаження. Як приклад, KEDA може автоматично

зменшувати кількість подів Jenkins у вихідні дні, коли навантаження зазвичай менше. Спочатку додаємо налаштування для використання spot-екземплярів EC2 у Karpenter. У файлі karpenter-values.yaml оновимо такі налаштування:

```
provisioning:
  provider: AWS
  launchTemplate:
    instanceProfile: KarpenterNodeInstanceProfile
    instanceTypes: ["t3.small", "t3.medium", "t3.large"]
    capacityType: SPOT
```

Рисунок 3.22 - Налаштування Karpenter

Тепер Karpenter буде використовувати спот-екземпляри EC2, що може знизити витрати на інфраструктуру. Далі, створіть KEDA для автоматичного масштабування подів вниз при зменшенні навантаження. Створюємо файл keda-scaler.yaml з конфігураціями масштабування на основі метрик з CloudWatch.

```yaml
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: jenkins-agent-scaler
  namespace: jenkins
spec:
  scaleTargetRef:
    name: jenkins-kubernetes-agent
  minReplicaCount: 1
  maxReplicaCount: 10
  triggers:
  - type: aws-cloudwatch
    metadata:
      namespace: AWS/EC2
      metricName: CPUUtilization
      dimensions: InstanceId=i-1234567890abcdef0
      statistic: Average
      targetValue: "50"
      minMetricValue: "0"
      maxMetricValue: "100"
      awsRegion: "us-west-2"
      identityOwner: pod
      awsAccessKeyID: AWS_ACCESS_KEY_ID
      awsSecretAccessKey: AWS_SECRET_ACCESS_KEY
      roleArn: arn:aws:iam::123456789012:role/keda-cloudwatch
```

Рисунок 3.23 - Налаштування KEDA

Тепер, коли наші конфігураційні файли готові, оновлюємо код Terraform для інтеграції Karpenter і KEDA з нашою інфраструктурою. У файлі main.tf додайте наступні рядки для встановлення Karpenter і KEDA з нашими оновленими значеннями:

```
module "karpenter" {
  source = "github.com/your-repo/karpenter-helm-chart"

  values_file = "karpenter-values.yaml"
}

module "keda" {
  source = "github.com/your-repo/keda-helm-chart"

  values_file = "keda-values.yaml"
}

resource "kubernetes_manifest" "keda_scaler" {
  provider = kubernetes

  manifest = yamldecode(file("keda-scaler.yaml"))
}
```

Рисунок 3.24 - Двадцята частина коду реалізації проекту на terraform

Тепер наша інфраструктура автоматично масштабується вгору та вниз залежно від навантаження, і використовує більш дешеві спот-екземпляри EC2, що допомагає оптимізувати витрати на хмарні ресурси.

Припустимо, у нас є система з 10 вузлами EKS, кожен з яких використовує m5.large instance. За стандартними цінами, це коштуватиме приблизно $0.192 на годину за instance, або $1.92 на годину за всю систему. Проте, Karpenter використовує Spot instances для половини цих вузлів, ми можемо ми економимо до 90% вартості цих instances.

4. ТЕСТУВАННЯ ТА АНАЛІЗ СИСТЕМИ

4.1 Створення тестового непередбачуваного навантаження з використанням Apache JMeter.

Apache JMeter - це потужний інструмент для тестування навантаження, який ми використовуємо для створення високого рівня навантаження на нашу систему Jenkins. Мета цього тесту - перевірити, наскільки добре наша система впорається з непередбачуваним навантаженням, і чи може вона продовжувати працювати ефективно, використовуючи мінімум ресурсів.

Після встановлення додатка ми можемо створити новий тестовий план у JMeter. У нашому тестовому плані ми використовуємо HTTP Request Sampler для надсилання HTTP-запитів до нашого сервера Jenkins. Ми можемо налаштувати цей Sampler, щоб надсилати запити до API Jenkins для створення нових завдань. Після запуску нашого тесту ми можемо проаналізувати результати, щоб дізнатися, як наша система впоралася із навантаженням. Ми повинні шукати будь-які ознаки перевантаження, такі як високі годинники відповіді, помилки HTTP або відмови в обслуговуванні. На основі цих результатів ми можемо вирішити, чи потрібно оптимізувати нашу систему, наприклад, налаштувавши параметри масштабування в Karpenter або змінивши конфігу.

Маємо базовий приклад скрипту JMX для JMeter, який надсилає HTTP-запит до Jenkins API для запуску роботи з ім’ям «example»:

```
<?xml version="1.0" encoding="UTF-8"?>
<jmeterTestPlan version="1.2" properties="5.0" jmeter="5.4.1">
  <hashTree>
    <TestPlan guiclass="TestPlanGui" testclass="TestPlan" testname="Test Pla
      <stringProp name="TestPlan.comments"></stringProp>
      <boolProp name="TestPlan.functional_mode">false</boolProp>
      <boolProp name="TestPlan.tearDown_on_shutdown">true</boolProp>
      <boolProp name="TestPlan.serialize_threadgroups">false</boolProp>
      <elementProp name="TestPlan.user_defined_variables" elementType="Argum
        <collectionProp name="Arguments.arguments"/>
      </elementProp>
      <stringProp name="TestPlan.user_define_classpath"></stringProp>
    </TestPlan>
    <hashTree>
      <ThreadGroup guiclass="ThreadGroupGui" testclass="ThreadGroup" testnam
        <stringProp name="ThreadGroup.on_sample_error">continue</stringProp>
        <elementProp name="ThreadGroup.main_controller" elementType="LoopCon
          <boolProp name="LoopController.continue_forever">false</boolProp>
          <stringProp name="LoopController.loops">10</stringProp>
        </elementProp>
        <stringProp name="ThreadGroup.num_threads">5</stringProp>
        <stringProp name="ThreadGroup.ramp_time">1</stringProp>
        <boolProp name="ThreadGroup.scheduler">false</boolProp>
        <stringProp name="ThreadGroup.duration"></stringProp>
        <stringProp name="ThreadGroup.delay"></stringProp>
      </ThreadGroup>
      <hashTree>
```

Рисунок 4.1 - Перша частина коду плану JMeter

```
        <HTTPSamplerProxy guiclass="HttpTestSampleGui" testclass="HTTPSample
          <elementProp name="HTTPsampler.Arguments" elementType="Arguments"
            <collectionProp name="Arguments.arguments">
              <elementProp name="" elementType="HTTPArgument">
                <boolProp name="HTTPArgument.always_encode">false</boolProp>
                <stringProp name="Argument.value"></stringProp>
                <stringProp name="Argument.metadata">=</stringProp>
                <boolProp name="HTTPArgument.use_equals">true</boolProp>
                <stringProp name="Argument.name"></stringProp>
              </elementProp>
            </collectionProp>
          </elementProp>
          <stringProp name="HTTPSampler.domain">your-jenkins-domain</stringP
          <stringProp name="HTTPSampler.port">8080</stringProp>
```

Рисунок 4.2 - Друга частина коду плану JMeter

Цей скрипт створює тестовий план JMeter, який виконує HTTP POST запит до Jenkins API для запуску роботи з ім'ям "example". Це виконується в циклі 100 разів з 5 потоками, тобто загалом 500 запитів до Jenkins API.

Файл .jtl, згенерований JMeter після тестування, містить деталі про кожен індивідуальний запит, включаючи таку інформацію, як час відповіді, статус відповіді, чи був запит успішним, тощо.

Ось приклад фрагменту, файлу .jtl після тестування нашого Jenkins сервера:

```
timeStamp,elapsed,label,responseCode,responseMessage,threadName,dataType,suc
1622635096000,567,HTTP Request,200,OK,Thread Group 1-1,text,true,1024,315,1,
1622635097000,654,HTTP Request,200,OK,Thread Group 1-2,text,true,1024,315,2,
1622635098000,623,HTTP Request,200,OK,Thread Group 1-3,text,true,1024,315,3,
...
```

Рисунок 4.3 - Події тестування плану JMeter

Цей приклад включає такі поля:

1. timeStamp: Час, коли запит було виконано.
2. elapsed: Час відповіді (в мілісекундах).
3. label: Назва запиту.
4. responseCode: Код відповіді HTTP.
5. responseMessage: Повідомлення відповіді HTTP.
6. threadName: Назва потоку, що виконує запит.
7. dataType: Тип даних відповіді.
8. success: Чи був запит успішним.
9. bytes: Кількість байтів, отриманих відповіддю.
10. sentBytes: Кількість байтів, відправлених запитом.
11. grpThreads: Кількість активних потоків у групі потоків.
12. allThreads: Загальна кількість активних потоків.
13. URL: URL запиту.
14. Latency: Затримка (в мілісекундах).
15. IdleTime: Час простою (в мілісекундах).
16. Connect: Час підключення (в мілісекундах).

Система працює стабільно та ми можемо побачити за допомогою kubectl як відпрацювала наша система:

```
NAME              READY   STATUS    RESTARTS   AGE
jenkins-agent-1   1/1     Running   0          5m
jenkins-agent-2   1/1     Running   0          5m
jenkins-agent-3   1/1     Running   0          5m
```

Рисунок 4.4 - Огляд под за допомогою kubectl

У цьому прикладі видно, що наші агенти Jenkins успішно розвернуті та готові до виконання завдань. Всі піди знаходяться в стані "Running", і не вимагають перезапусків. Зростання подов становить 5 хвилин з висновку .jtl ми можемо побачити що всі запроси були відпрацьовані.

4.2 Додаткові налаштування для оптимізації витрат.

Ми налаштовуємо Karpenter для автоматичного масштабування вузлів EKS, залежно від навантаження на кластер. Це дозволяє нам використовувати лише необхідну кількість ресурсів, уникаючи зайвих витрат. Ми використовуємо KEDA для автоматичного масштабування наших програм на основі подій. Наприклад, якщо у нас є програма для обробки повідомлень із черги, KEDA може автоматично масштабувати кількість екземплярів програми залежно від кількості повідомлень у черзі. Це дозволяє нам обробляти події ефективно, використовуючи лише необхідну кількість ресурсів.

Якщо ми додамо ще AWS Budgets з лімітом $500 на місяць. Коли витрати сягають 100% від цього бюджету, ми можемо отримувати повідомлення. Наприклад, якщо наші програми починають використовувати більше ресурсів через збільшений трафік або інші фактори, ми отримаємо

попередження і зможемо вжити заходів для оптимізації використання ресурсів та змінити налаштування Karpenter щоб знизити використання ресурсів та заощадити гроші. Таким чином, використання Karpenter AWS Budgets дозволяє нам оцінити результати та контролювати витрати в реальному часі, що допомагає заощадити гроші та оптимізувати використання ресурсів у AWS.

Тепер, припустимо, що ми хочемо встановити бюджет у $500 на місяць для нашого EKS кластера. Якщо наші витрати перевищують цей ліміт, ми отримаємо сповіщення на електронну пошту.

Для створення AWS Budgets, ми можемо використати Terraform:

```
provider "aws" {
  region = "us-west-2"
}

resource "aws_budgets_budget" "cost_budget" {
  name              = "example"
  budget_type       = "COST"
  limit_amount      = "500"
  limit_unit        = "USD"
  time_unit         = "MONTHLY"
  cost_types {
    include_credit           = true
    include_other_subscription = true
    include_recurring         = true
    include_refund            = true
    include_subscription      = true
    include_support           = true
    include_upfront           = true
    use_amortized             = false
  }

  time_period_start = "2023-07-01_00:00"
```

Рисунок 4.5 - Двадцять перша частина коду реалізації проекту на terraform

```
  notification {
    comparison_operator = "GREATER_THAN"
    threshold           = 100
    threshold_type      = "PERCENTAGE"
    notification_type   = "ACTUAL"
    subscriber_email_addresses = ["your-email@example.com"]
  }
}
```

Рисунок 4.6 - Двадцять друга частина коду реалізації проекту на terraform.

Цей Terraform скрипт створює бюджет на витрати AWS з місячним лімітом в $500. Якщо витрати перевищують 100% від цього бюджету, буде надіслано сповіщення на вказану електронну пошту.

Для контролю за витратами, ми можемо створити систему, яка заснована на AWS Lambda та AWS Simple Notification Service (SNS).

Коли AWS Budgets виявляє, що ми пєревищуємо бюджет, він може відправити повідомлення до SNS. Після цього, SNS може викликати Lambda функцію, яка змінює налаштування Karpenter та EKS для зниження використання ресурсів.

Нижче наведено приклад AWS Lambda функції, написаної на Python, яка виконує це:

```
import boto3
import yaml
import json

def lambda_handler(event, context):
    # Create AWS clients
    eks = boto3.client('eks')
    s3 = boto3.resource('s3')

    # Update EKS Nodegroup Configuration
    eks.update_nodegroup_config(
        clusterName='your_cluster_name',
        nodegroupName='your_nodegroup_name',
        scalingConfig={
            'minSize': 1,
            'maxSize': 1,
            'desiredSize': 1
        }
    )

    # Get current Karpenter configuration
    bucket_name = 'your_bucket_name'
    config_key = 'karpenter-config.yaml'
    karpenter_config_object = s3.Object(bucket_name, config_key)
    karpenter_config = yaml.safe_load(karpenter_config_object.get()['Body'].
    karpenter_config['spec']['provisioners'][0]['ttlSecondsAfterEmpty'] = 0
    karpenter_config['spec']['provisioners'][0]['ttlSecondsUntilExpired'] =
```

Рисунок 4.7 - Lambda функція для управління налаштуваннями карпентер.

5. РЕКОМЕНДАЦІЇ ПОДАЛЬШОГО РОЗВИТКУ

5.1 Пропозиції щодо оптимізації системи

Використання хмарних технологій для управління IT-інфраструктурою має безліч переваг, зокрема гнучкість, масштабованість та ефективність. Однак для повної реалізації цих переваг потрібно забезпечити ефективне управління та оптимізацію використання хмарних ресурсів. Це особливо актуально для систем з високим та/або непередбачуваним навантаженням, таких як системи неперервної інтеграції та доставки (CI/CD), які використовуються в більшості сучасних IT-інфраструктур.

У рамках даного дослідження було впроваджено рішення на базі Amazon EKS, Karpenter, Jenkins та AWS Lambda, що дозволяє автоматизувати процеси масштабування та управління ресурсами згідно з поточним навантаженням та бюджетом. Проте, як і будь-яка технологічна система, вона має потенціал для подальшої оптимізації.

У цьому розділі ми розглянемо декілька додаткових стратегій та підходів, які можуть бути використані для підвищення ефективності та оптимізації використання ресурсів у нашій системі:

1. Оптимізація роботи Jenkins — У вас може виникнути потреба в оптимізації роботи Jenkins, щоб зменшити витрати на ресурси.
2. Контроль над затратами Spot Instances — Переконайтеся, що ви контролюєте свої затрати на Spot Instances. Це може включати використання таких інструментів, як AWS Price List Service, для отримання останніх цін на Spot Instances, та використання стратегій, таких як Spot Fleet, для автоматичного масштабування ваших Spot Instances залежно від ваших потреб.
3. Оптимізація використання Lambda — Переконайтеся, що ви оптимально використовуєте AWS Lambda. Наприклад, ви можете використовувати AWS Lambda Power Tuning для оптимізації розміру пам'яті для ваших

функцій Lambda, або використовувати AWS Step Functions для координації ваших функцій Lambda.

4. Покращення автоматизації та моніторингу — Використання інструментів автоматизації, таких як Terraform, може допомогти зменшити помилки та підвищити продуктивність. Ви також можете використовувати інструменти моніторингу, такі як Amazon CloudWatch, для відстеження стану вашої інфраструктури та виявлення проблем на ранніх стадіях.
5. Оптимізація використання сховища — Використання ефективних стратегій зберігання даних, таких як використання Amazon S3 для зберігання неструктурованих даних або Amazon EBS для зберігання даних, які потребують високої продуктивності, може також допомогти оптимізувати використання ресурсів.

У кожному випадку, важливо постійно моніторити та аналізувати використання ресурсів вашої системи, щоб виявляти можливі проблеми та шукати шляхи до їх оптимізації.

5.2 Відображення даної работи на альтернативні проекти

При плануванні та розробці даного проекту було використано Jenkins як пример веб-додатку для демонстрації можливостей впровадженої інфраструктури. Однак концепція та технології, які ми тут застосували, не обмежуються лише одним застосуванням. Стратегії та інструменти, які ми розглядали, можуть бути застосовані до широкого спектра веб-додатків і бізнес-вимог.

1. Веб-додатки з великою кількістю користувачів: Для веб-додатків, що обслуговують велику кількість користувачів, таких як соціальні медіа або електронна комерція, навантаження може суттєво змінюватися протягом дня або в залежності від сезонності. Це може включати піки навантаження під час певних подій, вихідних або сезонних розпродажів.

Використання Karpenter для автоматичного масштабування нод у відповідь на ці піки може допомогти забезпечити високу доступність та продуктивність, зберігаючи при цьому оптимальні витрати.

2. Веб-додатки з ресурсоємними операціями: Для веб-додатків, які виконують ресурсоємні операції, такі як обробка великих наборів даних, машинне навчання або відео транскодування, може бути корисною стратегія масштабування на основі навантаження, що реалізована за допомогою Karpenter. Це дозволить системі автоматично додавати ресурси, коли вони потрібні, та звільняти їх, коли вони вже не потрібні.
3. Стартапи та малі бізнеси: Для стартапів та малих бізнесів, які шукають гнучкість та ефективність витрат, дана система може бути особливо корисною. Використання AWS Lambda для автоматичного управління бюджетом може допомогти контролювати витрати та запобігти несподіваним витратам.
4. Бізнеси з високими вимогами до доступності: Для бізнесів, яким потрібна висока доступність, таких як фінансові установи або служби надзвичайних ситуацій, використання EKS з автоматичним масштабуванням та управлінням життєвим циклом ресурсів може допомогти забезпечити неперервну роботу додатків.

Таким чином, досліджені в даній роботі підходи та технології можуть бути успішно застосовані в різних контекстах і сценаріях, незалежно від конкретного додатка, який використовується як приклад.

ВИСНОВКИ

У сучасному світі хмарні технології стали невід'ємною частиною IT-інфраструктури більшості компаній. Проте з урахуванням постійно зростаючого обсягу даних та динаміки зміни навантаження на системи, важливо використовувати ці технології ефективно, щоб оптимізувати витрати і забезпечити високу продуктивність.

В даному дослідженні ми розглядали використання EKS, Karpenter та KEDA для створення гнучкої, високо доступної та масштабованої хмарної інфраструктури, яка може автоматично адаптуватися до змін навантаження. Ці технології дозволяють компаніям ефективно використовувати свої хмарні ресурси, автоматично масштабуючи свою інфраструктуру відповідно до потреб, знижуючи тим самим витрати на невикористані ресурси.

Автоматизація процесів управління інфраструктурою за допомогою Terraform також є ключовим фактором економії ресурсів. Вона зменшує помилки, пов'язані з людським фактором, покращує продуктивність системи та забезпечує консистентність конфігурацій.

Подальше дослідження в цій області може запропонувати розробку більш досконалих стратегій управління ресурсами, використовуючи високопродуктивні та економічно ефективні рішення для хмарних інфраструктури. Також можна розглянути інтеграцію додаткових сервісів для автоматизації та оптимізації, які можуть подальше покращити ефективність системи.

Загалом, дане дослідження підтверджує значимість автоматизації та оптимізації в управлінні хмарними ресурсами та демонструє практичну вартість застосування цих підходів в реальних бізнес-сценаріях. Це важливий вклад в наукове поле хмарних технологій та управління IT-інфраструктурою, який може стати основою для подальших досліджень і розробок.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Types-of-cloud-computing [Електронний ресурс] / Amazon Web Services, Inc. – 2023. – Режим доступу до ресурсу: https://aws.amazon.com/ru/types-of-cloud-computing/.

2. AWS Products Pricing Types [Електронний ресурс] // Amazon Web Services, Inc. – 2023. – Режим доступу до ресурсу: https://aws.amazon.com/ru/pricing

3. What is CI/CD? [Електронний ресурс] // Red Hat, Inc.. – 2022 - Режим доступу до ресурсу: https://www.redhat.com/en/topics/devops/what-is-ci-cd.

4. Kubernetes Overview [Електронний ресурс] // The Kubernetes Authors, Inc. – 2023. – Режим доступу до ресурсу: https://kubernetes.io/docs/concepts/overview/

5. Kubernetes Components [Електронний ресурс] // The Kubernetes Authors, Inc. – 2023. – Режим доступу до ресурсу: https://kubernetes.io/docs/concepts/overview/

7. Karpenter How It Works [Електронний ресурс] // Amazon Web Services, Inc. – 2023. – Режим доступу до ресурсу: https://karpenter.sh/

8. Kubernetes Event-driven Autoscaling [Електронний ресурс] // KEDA Authors . – 2023. – Режим доступу до ресурсу: https://keda.sh/

9. What is Terraform [Електронний ресурс] // HashiCorp. – 2023. – Режим доступу до ресурсу: https://developer.hashicorp.com/terraform/intro.

10. What is EKS [Електронний ресурс] // Amazon Web Services, Inc. – 2023. – Режим доступу до ресурсу: https://docs.aws.amazon.com/eks/latest/userguide/what-is-eks.html

11. What is Amazon CloudWatch [Електронний ресурс] // Amazon Web Services, Inc. – 2023. – Режим доступу до ресурсу: https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/WhatIsCloudWatch.html