Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії та управління
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти другий (магістерський)

Методи та засоби автоматичного масштабування Kubernetes-кластеру в хмарному середовищі AWS
(тема)

Виконав:

здобувач 2 року навчання,

групи СПм-23-3

Денис ПОМЕЛУЙКО
(власне ім’я, прізвище)

Спеціальність

123 «Комп’ютерна інженерія»
(код і повна назва спеціальності)

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

Освітня програма

Системне програмування
(повна назва освітньої програми)

Керівник: ст. викл. Наталія ЄРЬОМІНА
(посада, власне ім’я, прізвище)

Допускається до захисту

Завідувач кафедри ЕОМ _____ Андрій КОВАЛЕНКО
(підпис) (власне ім’я, прізвище)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп'ютерної інженерії та управління

Кафедра електронних обчислювальних машин

Рівень вищої освіти другий (магістерський)

Спеціальність 123 «Комп'ютерна інженерія»
(код і повна назва)

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

Освітня програма Системне програмування
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
"____" _____ 20__ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Помелуйку Денису Андрійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Методи та засоби автоматичного масштабування Kubernetes-кластеру в хмарному середовищі AWS

затверджена наказом по університету від "21" квітня 2025 р. № 296 Ст

2. Термін подання здобувачем роботи до екзаменаційної комісії 16 червня 2025 р.

3. Вхідні дані до роботи

1) Офіційна документація провайдера AWS
2) Технічні документи Kubernetes, Cluster Autoscaler, Karpenter, KEDA, Terraform
3) Методи масштабування Kubernetes-кластеру
4) Обліковий запис в середовищі AWS
5) Програмні засоби: MacOS, Terminal, kubectl, aws cli

4. Перелік питань, що потрібно опрацювати у роботі

1) Аналіз предметної області
2) Огляд засобів масштабування Kubernetes-кластеру
3) Побудова інфраструктури та розробка методів автоматичного масштабування
4) Оцінка реалізованих методів

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій 12 слайдів формату А4.

6. Консультанти розділів роботи (заповнюється за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Строк / терміни виконання етапів роботи	Примітка
1	Аналіз проблеми та предметної області	22.04.25-29.04.25	
2	Огляд засобів масштабування	30.04.25-05.05.25	
3	Розробки методів масштабування	06.05.25-21.05.25	
4	Реалізація та оцінка запропонованих методів	22.05.25-02.06.25	
5	Оформлення матеріалів кваліфікаційної роботи	03.06.25-08.06.25	
6	Подання кваліфікаційної роботи керівникові	09.06.25-10.06.25	
	та її попередній захист		
7	Подання кваліфікаційної роботи на рецензу-	11.06.25-12.06.25	
	вання		

Дата видачі завдання " 21 " квітня 2025 р.

Здобувач _____
(підпис)

Керівник роботи _____ ст. викл. Наталія ЄРЬОМІНА
(підпис) (посада, власне ім'я, прізвище)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 99 с., 17 рис., 2 дод., 15 джерел.

ХМАРНІ ТЕХНОЛОГІЇ, МАСШТАБУВАННЯ КЛАСТЕРУ, МІКРОСЕРВІСИ, KUBERNETES, DEPLOYMENT, AWS, AUTOSCALER, KARPENTER, KEDA, SQS, ALB.

Метою кваліфікаційної роботи є дослідження та впровадження методів автоматичного масштабування Kubernetes-кластеру в хмарному середовищі Amazon Web Services (AWS) із використанням сучасних засобів масштабування.

У ході виконання кваліфікаційної роботи було реалізовано автоматизоване розгортання інфраструктури кластеру за допомогою Terraform та Helm, детально досліджено архітектуру EKS, особливості розгортання та налаштування компонентів масштабування Cluster Autoscaler, Karpenter, HPA та KEDA. Проведено розгортання тестового середовища, розроблено експериментальні застосунки для перевірки різних сценаріїв масштабування, налаштовано моніторинг за допомогою Prometheus і Grafana. Експериментально підтверджено ефективність моделі масштабування для різних типів навантаження, зокрема з використанням зовнішніх. Результати роботи можуть бути використані для впровадження автоматизованих рішень у сфері хмарних технологій, оптимізації витрат та підвищення стабільності роботи розподілених систем.

ABSTRACT

Master’s thesis: 99 pages, 17 figures, 2 appendices, 15 sources.

CLOUD TECHNOLOGIES, CLUSTER SCALING, MICROSERVICES, KUBERNETES, DEPLOYMENT, AWS, AUTOSCALER, KARPENTER, KEDA, SQS, ALB.

The major goal of this thesis is to research and implement methods for automatic scaling of a Kubernetes cluster in the Amazon Web Services (AWS) cloud environment using modern scaling tools.

During the course of the work, automated infrastructure deployment of the cluster was implemented using Terraform and Helm, and the architecture of Amazon EKS was thoroughly examined. Special attention was given to the deployment and configuration of scaling components such as Cluster Autoscaler, Karpenter, HPA, and KEDA. A test environment was deployed, experimental applications were developed to evaluate various scaling scenarios, and monitoring was configured using Prometheus and Grafana. The effectiveness of the scaling model was experimentally confirmed for different types of workloads, including those based on external metrics.

The results of this work can be applied to implement automated solutions in the field of cloud technologies, optimize costs, and improve the stability of distributed systems.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

ALB (Application Load Balancer) — балансувальник навантаження прикладного рівня

AMI (Amazon Machine Image) — образ віртуальної машини AWS

API (Application Programming Interface) — інтерфейс прикладного програмування

ASG (Auto Scaling Group) — група автоматичного масштабування

AWS (Amazon Web Services) — хмарна платформа Amazon

CPU (Central Processing Unit) — центральний процесор

EC2 (Elastic Compute Cloud) — хмарний сервіс віртуальних машин AWS

EKS (Elastic Kubernetes Service) — керований сервіс Kubernetes від AWS

HCL (HashiCorp Configuration Language) – декларативна мова конфігурації

HPA (Horizontal Pod Autoscaler) — горизонтальний масштабувальник подів

HTTP (Hypertext Transfer Protocol) — протокол передачі даних, що використовується в комп'ютерних мережах

IaC (Infrastructure as Code) — інфраструктура як код

KEDA (Kubernetes Event-Driven Autoscaling) — масштабування подів за подіями

SQS (Simple Queue Service) — сервіс черг повідомлень AWS

TCP (Transmission Control Protocol) — протокол управління передачею

VPA (Vertical Pod Autoscaler) — вертикальний масштабувальник подів

VPC (Virtual Private Cloud) — віртуальна приватна хмара

ВСТУП

Сучасні програмні системи стрімко еволюціонують від монолітних архітектур до розподілених мікросервісних рішень, прагнучи досягти більшої гнучкості, надійності та масштабованості. Цей перехід супроводжується масовим впровадженням контейнеризації та платформ для оркестрації контейнерів, таких як Kubernetes, які автоматизують розгортання та масштабування застосунків. У міру перенесення контейнеризованих навантажень до хмарних середовищ, зокрема Amazon Web Services (AWS), взаємодія мікросервісів, Kubernetes і хмарної інфраструктури створює основу для динамічних обчислювальних середовищ [1]. Ключовим елементом є автоматичне масштабування – здатність системи оперативно змінювати обсяг обчислювальних ресурсів відповідно до коливань навантаження.

Аналіз існуючих рішень у сфері масштабування Kubernetes показує, що стандартних можливостей, таких як HPA або Cluster Autoscaler, може бути недостатньо для повного покриття реальних сценаріїв навантаження. У зв’язку з цим актуальним є дослідження інструментів нового покоління, таких як Karpenter та KEDA, які забезпечують гнучкіші та ефективніші механізми масштабування.

В роботі окреслюються стратегії масштабування (вертикальне та горизонтальне; масштабування подів і вузлів), виявляються основні проблеми (затримки масштабування, проблема холодного старту, орієнтоване на події масштабування), а також піднімається питання оптимізації витрат при масштабуванні; пояснюються аспекти розподілу компонентів у кластері Kubernetes, логіка роботи вбудованих та зовнішніх масштабувальників.

Метою даної роботи є дослідження та впровадження методів автоматичного масштабування Kubernetes-кластеру в хмарному середовищі Amazon Web Services (AWS) із використанням сучасних засобів

масштабування.

Об’єктом дослідження є Kubernetes-кластер, розгорнутий у хмарному середовищі AWS. Предметом дослідження є методи та засоби автоматичного масштабування його обчислювальних ресурсів.

Наукова новизна роботи полягає в комплексному підході до автоматизації масштабування подів і вузлів у кластері Kubernetes на основі сучасних інструментів.

Практичне значення роботи полягає в можливості впровадження розробленої моделі масштабування для побудови ефективних, стабільних і адаптивних систем в хмарному середовищі, що дозволяють оптимізувати використання ресурсів та знизити експлуатаційні витрати.

1 ТЕОРЕТИЧНІ ВІДОМОСТІ

1.1 Поняття мікросервісної архітектури

Мікросервісна архітектура є типом проєктування програмних застосунків, у рамках якого система складається з низки невеликих, автономних сервісів. Кожен з них виконується як окремий процес і взаємодіє з іншими за допомогою механізмів обміну даними – здебільшого через HTTP/TCP або систем обміну повідомленнями [2].

Цей підхід суттєво відрізняється від традиційної монолітної архітектури, у якій всі компоненти зосереджені в єдиному, великому застосунку. Натомість мікросервіси орієнтовані на окремі бізнес-функції, забезпечуючи модульність, яка полегшує розробку, тестування та розгортання окремих частин однієї системи.

Перехід до мікросервісної архітектури суттєво впливає на можливості масштабування та керованість системи. Це, у свою чергу, визначає підходи до автоматичного масштабування Kubernetes-кластерів, де важливу роль відіграють ефективність, швидкість розгортання та оптимальне використання ресурсів.

Останніми роками термін «мікросервіс» набув значної популярності серед архітекторів програмного забезпечення. У міру зростання обсягу та складності програмного забезпечення перед кожним архітектором постає непросте завдання – створити ефективну систему, яка б забезпечувала баланс між якістю та швидкістю розробки.

У період переважання монолітних застосунків один програмний застосунок міг охоплювати численні модулі, що належали до різних бізнес-компонентів. Попри простоту початкової реалізації, монолітний підхід мав значні обмеження щодо масштабованості та підтримки. Навіть незначне виправлення помилки могло вимагати повного перескладання та повторного

розгортання всієї системи.

Виходом із ситуації стала поява сервісно-орієнтованої архітектури (SOA), яка вирішила багато проблем: модулі було відокремлено від моноліту та перетворено на сервіси, що могли використовуватись іншими компонентами. У межах SOA обмін повідомленнями між сервісами забезпечується за допомогою єдиної шини повідомлень. Цей підхід ліг в основу мікросервісної архітектури, яка не лише успадкувала, але й значно розширила концепцію SOA (рисунок 1.1).

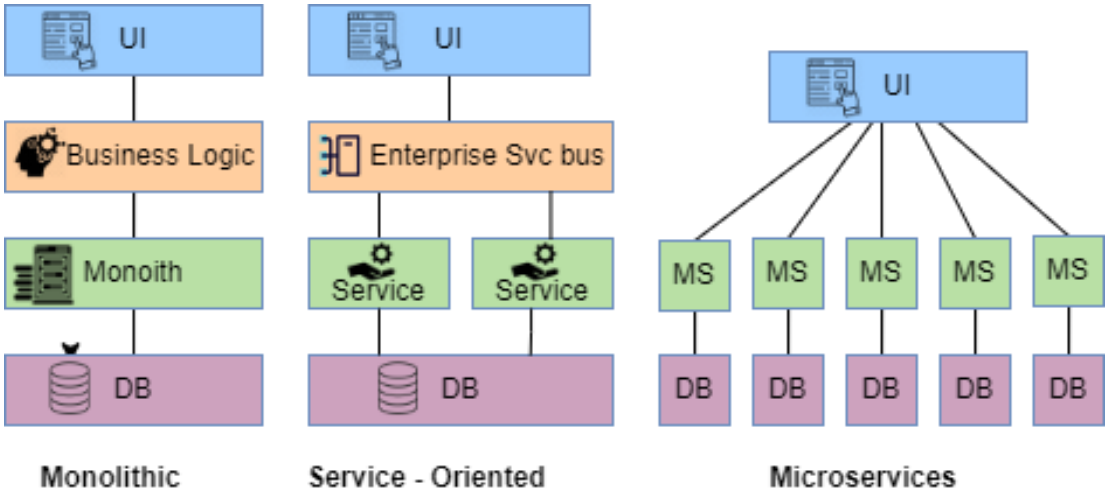

Рисунок 1.1 – Стилі архітектури програмного забезпечення

Основні характеристики мікросервісної архітектури можна звести до чотирьох ключових аспектів [3]. Нижче наведено стислий огляд чотирьох основних характеристик:

- децентралізоване управління. Команди, що розробляють мікросервіси, зазвичай є автономними. Кожна команда має змогу самостійно обирати стек технологій і підхід до управління проєктом – наприклад, Scrum методологію;

- незалежне розгортання. Мікросервіси, як правило, будуються навколо чітко визначених бізнес-доменів, що дозволяє незалежно розгортати та супроводжувати окремі компоненти системи. Команди можуть працювати у власному темпі, а грамотне проектування та документування інтерфейсів

на ранніх етапах дає змогу впроваджувати кілька оновлень паралельно, без конфліктів;

- гнучкість та автоматизація. Автоматизація є невід'ємною частиною підходу до створення мікросервісів. Впровадження таких практик як DevOps, GitOps, або використання інфраструктури як коду (IaC) дозволяє значно прискорити розробку та скоротити час виходу продукту на ринок. На відміну від монолітної архітектури, у мікросервісному середовищі для виправлення помилок чи внесення змін не потрібно перевипускати всю систему;

- точкове масштабування. Мікросервісна архітектура передбачає високу стійкість і гнучкість. Ключові компоненти системи можна масштабувати незалежно, відповідно до поточного навантаження. Автоматичне масштабування й оркестрація сервісів зазвичай реалізуються за допомогою таких платформ, як Kubernetes.

Таким чином, мікросервісний підхід відкриває нові горизонти для побудови ефективних, масштабованих і гнучких програмних систем. Варто зауважити, що попри значні переваги цього підходу, перехід до мікросервісної архітектури викликає низку викликів для організацій, які впроваджують її вперше – як з огляду на технологічні рішення, так і бізнес-дизайну.

1.2 Віртуалізація контейнерів

Віртуалізація контейнерів – це форма віртуалізації на рівні операційної системи (ОС), яка дає змогу створювати кілька ізольованих середовищ (контейнерів) у межах одного ядра ОС. На відміну від повноцінних віртуальних машин, де кожен екземпляр емулює апаратне забезпечення і містить повноцінну операційну систему, контейнери спільно використовують ядро хост-системи, ізолюючи лише необхідне середовище для застосунків.

Серед ключових можливостей, що забезпечують роботу контейнерів, вирізняють:

- простори імен. Вони ізолюють ресурси системи (PID, мережа, файлові системи тощо) так, що процеси всередині контейнера отримують лише виділений для них набір ресурсів. Мережеві простори імен забезпечують кожному контейнеру окремі мережеві інтерфейси, IP-адреси та таблиці маршрутизації;

- групи контролю. Дають змогу обмежити та контролювати споживання ресурсів (процесор, пам’ять, дисковий ввід/вивід тощо) на рівні групи процесів;

- Union-файлові системи. Завдяки їм можливо створювати багато-шарові образи (базовий образ + шар із правами запису для змін). Такі інструменти, як OverlayFS чи AUFS, дозволяють збирати контейнерні образи з окремих шарів, даючи змогу декільком контейнерам спільно використовувати базові шари (наприклад, базовий образ ОС).

Завдяки такому підходу контейнери утворюють простір, подібний до віртуальних машин, але не вимагають повноцінної гостьової ОС у кожному контейнері. Вони вирізняються невеликою вагою та швидкістю запуску, що має вирішальне значення у сценаріях масштабування, коли треба оперативно ініціювати велику кількість нових екземплярів у відповідь на навантаження.

Переваги контейнерів у порівнянні з віртуальними машинами наступні:

- швидкість запуску. Віртуальній машині часто потрібні хвилини для повного старту, оскільки ініціалізується вся гостьова операційна система. Контейнер здатен розпочати роботу за секунди або навіть швидше, адже він створює новий процес у вже працюючому ядрі;

- безпека та ізоляція. Віртуальні машини мають сильніший рівень ізоляції завдяки апаратній віртуалізації, тоді як контейнери покладаються на ізоляцію на рівні ядра (помилка в ядрі ОС може призвести до компрометації). Утім, рівень безпеки контейнерів є прийнятним для більшості сценаріїв. За потреби вищої ізоляції існують такі рішення як Kata Containers, що поєднують апаратну віртуалізацію з інтерфейсом контейнерів;

- ефективність використання ресурсів. Завдяки спільному ядру

контейнери значно менше навантажують систему, ніж віртуальні машини. Це робить контейнеризацію ідеальною для розміщення великої кількості мікросервісів на одному хості без суттєвих накладних витрат, властивих віртуальним машинами.

На рисунку 1.2 зображено порівняння віртуальної машини та контейнеру: зліва кожна ВМ має власну гостьову операційну систему, а справа – кілька контейнерів поділяють одне ядро, ізолюючись лише на рівні застосунків. Така архітектура робить контейнери значно легшими та швидшими у запуску порівняно з повноцінними ВМ.

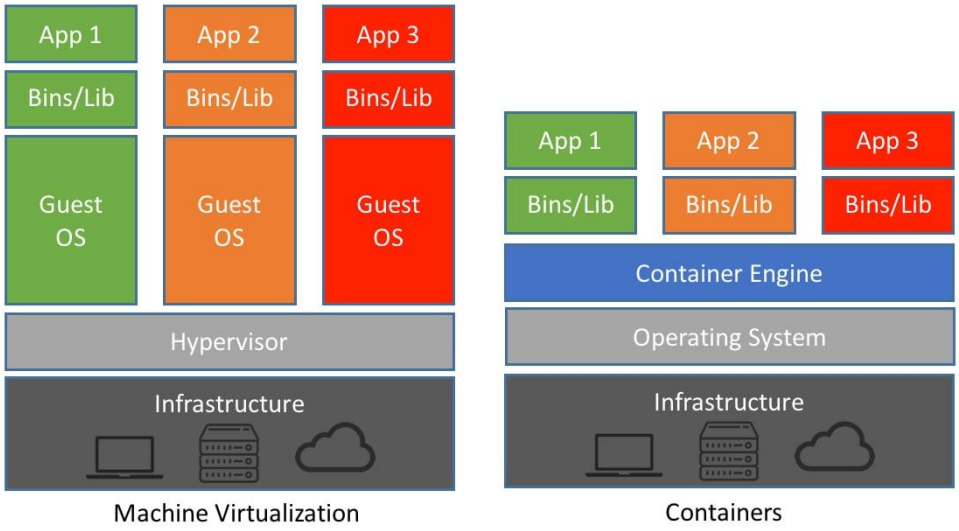

Рисунок 1.2 – Порівняння віртуальних машин та контейнерів

Здатність контейнерів швидко запускатися й мати низькі потреби в ресурсах дозволяє оркестраторам (як от Kubernetes) оперативно запускати нові контейнери.

1.3 Kubernetes

Kubernetes – це платформа для оркестрації контейнерів. Вона автоматизує процеси розгортання, масштабування та управління контейнеризованими застосунками, забезпечуючи стабільну та

передбачувану роботу.

Однією з ключових особливостей Kubernetes є декларативна модель управління: бажаний стан системи описаний як об’єкти. Вбудовані контролери Kubernetes автоматично слідкують за тим, щоб фактичний стан відповідав зазначеному.

У контексті масштабування саме архітектура Kubernetes та його об’єктна модель мають особливе значення. Масштабування контейнерів (подів) здійснюється за допомогою зміни відповідних ресурсів, таких як Deployment або HorizontalPodAutoscaler (HPA). Масштабування ж обчислювальних вузлів виконується через компоненти, що працюють на рівні інфраструктури.

1.3.1 Архітектура Kubernetes кластеру

Kubernetes кластер складається з двох головних частин: площини керування (master nodes або Control Plane) та обчислювальних вузлів (worker nodes), на яких безпосередньо запускаються контейнери. Це розділення відіграє ключову роль у механізмах масштабування, оскільки зазвичай потрібно додавати більше робочих вузлів, але в окремих випадках – масштабувати й саму площину керування. У керованих сервісах, на зразок AWS Elastic Kubernetes Service (EKS), хмарне середовище забезпечує високу доступність площини керування та керує її масштабуванням.

Площина керування відповідає за загальну логіку та обробку API-запитів та містить такі компоненти (рисунок 1.3):

- kube-apiserver. Основний інтерфейс (REST API) керування кластером, що приймає запити від інших компонентів та користувачів (через kubectl чи інші клієнти). Він виконує аутентифікацію, валідацію та оновлює стан кластера у сховищі etcd;
- etcd. Розподілене key-value сховище, де зберігається актуальний стан усього кластера. Воно зберігає інформація про поди, розгортання та

конфігурації інших об’єктів кластеру. Надійність і коректна робота etcd є критичною для стабільності кластера;

- kube-scheduler. Відстежує новостворені поди, яким ще не призначено обчислювальний вузол, і обирає його на основі наявних ресурсів та обмежень. Продуктивність планувальника впливає на швидкість масштабування, адже коли потрібно розгорнути багато нових екземплярів контейнерів, планувальник має оперативно прийняти рішення, де саме їх запустити;

- kube-controller-manager. Вміщує набір контролерів, що відстежують різні ресурси. Наприклад, Deployment controller перевіряє, чи відповідає кількість подів потрібній; або ж Node controller – чи доступні та справні обчислювальні вузли. Кожен контролер постійно співставляє фактичний стан із заданим та виконує необхідні дії для синхронізації;

- cloud-controller-manager. У хмарних середовищах цей компонент взаємодіє з API надавача послуг. Компонент також обробляє події реєстрації вузла, включаючи інтеграцію з хмарним середовищем.

Кожен робочий вузол містить наступні компонени:

- kubelet. Агент, що відповідає за запуск контейнерів на вузлі, дотримуючись інструкцій від площини керування. Компонент реєструє вузол у кластері, надсилає звіти про його стан і забезпечує безперервну роботу контейнерів, здійснюючи перезапуск у разі збоїв;

- kube-proxy. Налаштовує мережеві правила (iptables або IPVS) для маршрутів сервісів для того, щоб мережевий трафік потрапляв до відповідних контейнерів. За допомогою цього компоненту забезпечується віртуальний IP сервісу на кожному вузлі [4].

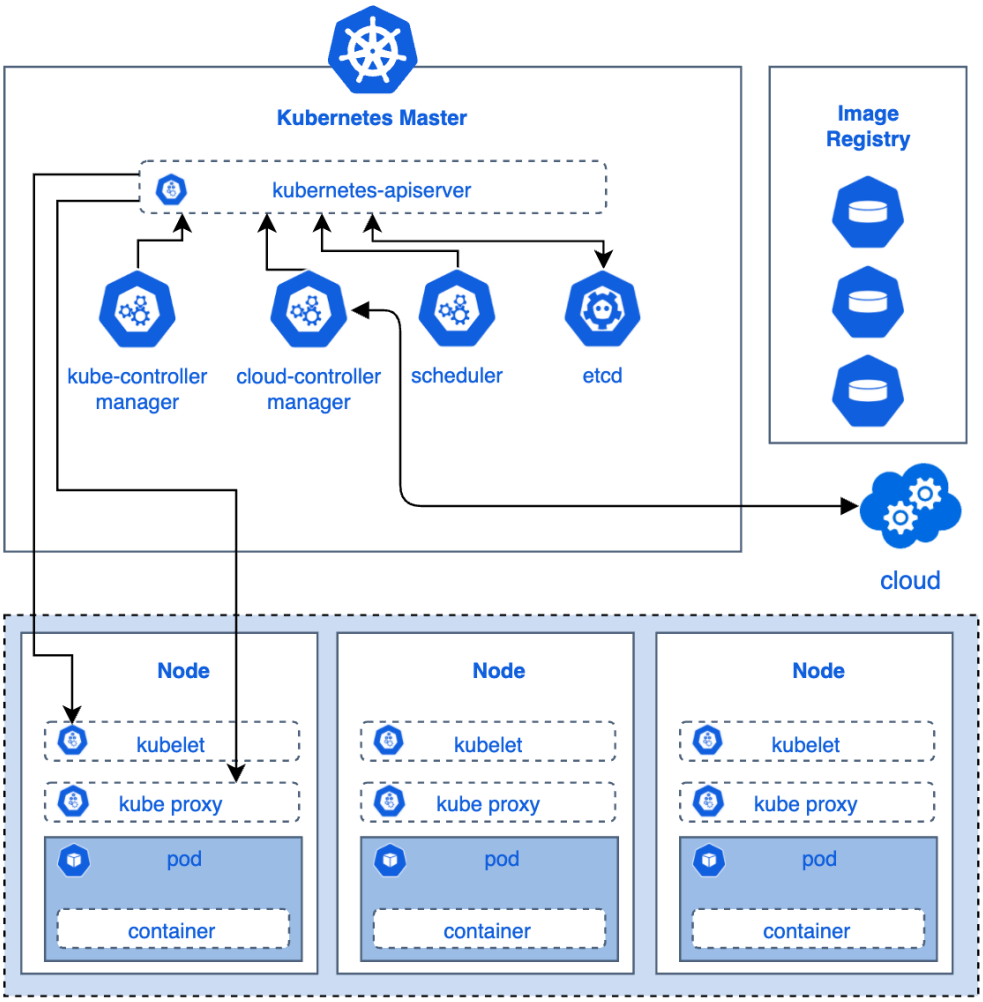

Рисунок 1.3 – Компоненти архітектури кластуру

1.3.2 Основні об’єкти Kubernetes кластеру

Усі сутності в Kubernetes кластері подаються як об’єкти. Для розуміння механізмів масштабування найважливішими є такі:

- Pod. Базова одиниця розгортання, яка може містити один або кілька пов’язаних контейнерів (наприклад, основний контейнер та допоміжний). Зазвичай, у мікросервісних середовищах у поді є тільки один головний контейнер;

- Deployment. Більш високорівнева абстракція, що визначає бажану кількість ідентичних копій подів (реплік) разом із шаблоном пода (Pod Template). Контролер Deployment стежить, щоб реальна кількість подів

відповідала заданій;

- ReplicaSet. Ресурс, який безпосередньо підтримує вказану кількість подів;

- DaemonSet. Забезпечує запуск певного пода на кожному вузлі (наприклад, для збору журналів повідомлень). Він не відповідає безпосередньо за масштабування під навантаженням, проте якщо до кластера додають нові вузли, DaemonSet розміщує свій под і на них;

- Service. Логічне групування подів і політика доступу до них, зазвичай через віртуальну IP-адресу (ClusterIP) та DNS-ім'я. Сервіс виконує балансування навантаження між подами. Якщо змінюється кількість подів, Service автоматично оновлює список доступних кінцевих точок.

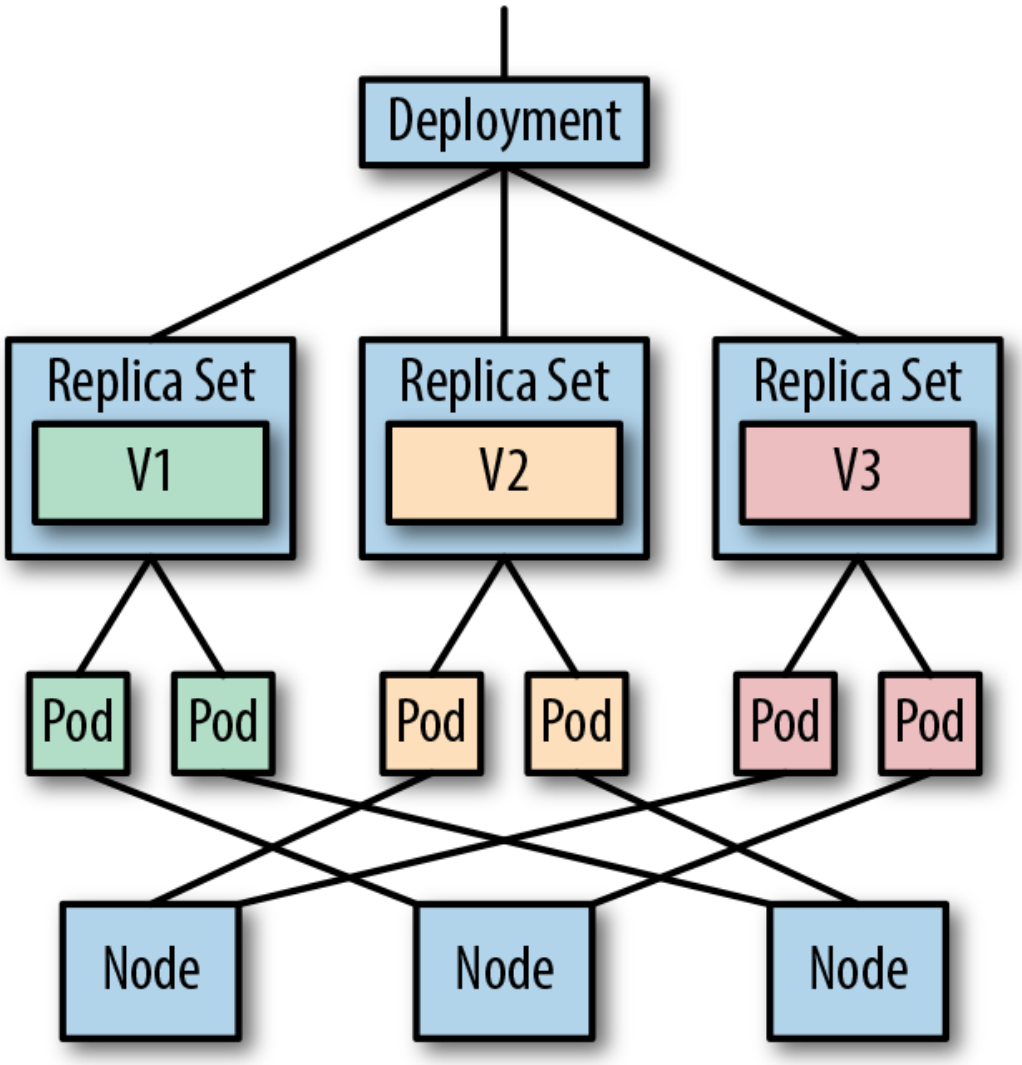

Рисунок 1.4 – Відношення сутностей в кластері

1.4 Хмарне середовище Amazon Web Services

Amazon Web Services (AWS) є одним із провідних постачальників хмарних обчислень, який надає широкий спектр сервісів для побудови, розгортання та масштабування додатків у глобальному масштабі. Незважаючи на платформонезалежний характер Kubernetes, розгортання його на AWS передбачає інтеграцію з конкретними сервісами AWS для організації мережевої взаємодії та забезпечення обчислювальних ресурсів.

1.4.1 Сервіс Amazon Elastic Kubernetes Service

Elastic Kubernetes Service (EKS) є сервісом AWS, який надає платформу для розгортання та експлуатації Kubernetes кластеру.

Площина керування (Control Plane) – складається з Kubernetes API-сервера та інших ключових компонентів Kubernetes, що працюють на керованих AWS потужностях у різних зонах доступності (availability zones) одного регіону. AWS повністю відповідає за оновлення, резервне копіювання та відновлення цих компонентів, гарантуючи їхню постійну роботу відповідно до Service Level Agreement (SLA) [5].

Площина даних (Data Plane) – складається з EC2 екземплярів, які називаються обчислювальними вузлами (worker nodes). Вони можуть бути організовані у спеціальні групи вузлів (Node Groups) або точково керуватися за допомогою контролерів, як от Karpenter. На робочих вузлах розгортаються поди з контейнерами, що забезпечують обчислювальні потужності для застосунків, які запускаються у Kubernetes кластері. Існують два типи груп вузлів.

Керовані групи вузлів (Managed Node Groups) – це групи вузлів, що керуються безпосередньо через API EKS. Вони забезпечують автоматичні оновлення, спрощену інтеграцію з іншими сервісами AWS та зручність при масштабуванні. Managed Node Groups інтегруються із сервісами AWS,

такими як CloudWatch та Auto Scaling Groups (ASG), забезпечуючи автоматичний контроль над змінами.

Самостійно керовані групи вузлів (Self-managed Node Groups) – традиційні групи вузлів, які адміністратор кластеру розгортає, керує та оновлює самостійно, використовуючи стандартні EC2 екземпляри. Цей варіант надає більшу гнучкість, але водночас вимагає більших витрат часу на управління та підтримку вузлів [6].

1.4.2 Сервіс Amazon Elastic Compute Cloud

Elastic Compute Cloud (EC2) надає віртуальні сервери, які називаються інстансами, що формують обчислювальну основу для роботи кластерів Kubernetes. Сервіс також включає балансувальники навантаження, які рівномірно розподіляють мережевий трафік між екземплярами. Розрізняють три типи балансувальників:

- Application Load Balancer (ALB) – маршрутизація HTTP/HTTPS-запитів з інтеграцією в Kubernetes через Ingress-контролери;
- Network Load Balancer (NLB) –балансування на рівні TCP;
- Classic Load Balancer (CLB) – традиційний балансувальник, який поступово замінюється ALB та NLB.

Групи автоматичного масштабування (Auto Scaling Groups, ASG) забезпечують динамічне додавання або видалення EC2 екземплярів базуючись на змінах у навантаженні.

1.4.3 Сервіс Amazon Virtual Private Cloud

Virtual Private Cloud (VPC) є сервісом AWS, що дозволяє створити приватну та ізольовану мережу у хмарному середовищі. EKS кластери розгортаються на основі VPC, що забезпечує:

- безпечну й ізольовану комунікацію між компонентами;

- точне налаштування мережевого доступу за допомогою підмереж, таблиць маршрутів та мережевих шлюзів;

- інтеграцію з іншими сервісами AWS (наприклад, EC2, ELB, RDS тощо).

Крім того, VPC надає можливість створення приватних і публічних підмереж, що дозволяє гнучко організовувати мережевий доступ до ресурсів: публічні підмережі мають безпосередній доступ до Інтернету через Internet Gateway (IGW), тоді як приватні підмережі використовують NAT Gateway для вихідного доступу до Інтернету, забезпечуючи додатковий захист ресурсів. Додатково, VPC підтримує використання VPN-з'єднань або AWS Direct Connect для безпечного підключення до локальних корпоративних мереж. Це створює безпечну й надійну архітектуру для розгортання Kubernetes у хмарному середовищі AWS [7].

1.4.4 Сервіс Amazon Simple Queue Service

Simple Queue Service (SQS) – це повністю керований AWS сервіс черг повідомлень, який забезпечує асинхронну взаємодію між компонентами системи. У контексті Kubernetes-кластерів SQS може використовуватися як джерело подій для подієво-орієнтованого масштабування за допомогою таких інструментів як KEDA.

Завдяки високій масштабованості та надійності, SQS ідеально підходить для обробки пікових навантажень та допомагає зменшити затримки обробки повідомлень, оптимізуючи таким чином продуктивність кластеру.

1.4.5 Сервіс Amazon CloudWatch

CloudWatch – це сервіс моніторингу, що дозволяє збирати та аналізувати метрики ресурсів AWS. CloudWatch підтримує моніторинг

Kubernetes-кластерів, забезпечуючи:

- збір та зберігання метрик;
- налаштування сигналів сповіщення;
- автоматичне реагування на зміни у параметрах системі;
- інтеграцію з інструментами аналітики та візуалізації.

Використання CloudWatch дозволяє оперативно реагувати на зміни у стані кластеру, забезпечуючи ефективне управління ресурсами.

1.4.6 Amazon Spot Instances

Spot Instances – це спеціальний тип віртуальних серверів, які дозволяють використовувати надлишкові обчислювальні ресурси AWS за значно нижчою ціною порівняно зі звичайними On-Demand-інстансами. Вони є одним із методів економії коштів на хмарну інфраструктуру та ефективним інструментом для зниження витрат на запуск ресурсномістких або гнучких щодо часу задач.

Особливість спотових інстансів полягає в тому, що їх вартість постійно змінюється залежно від попиту і пропозиції обчислювальних ресурсів у конкретному регіоні та зоні доступності AWS. Користувач встановлює максимальну ціну, яку він готовий заплатити за спотовий інстанс, і якщо поточна ціна нижча або дорівнює встановленій межі, інстанс запускається. Якщо ж ціна піднімається вище встановленого максимуму або виникає дефіцит ресурсів, AWS може примусово перервати роботу такого інстансу.

З огляду на можливість переривання роботи, спотові екземпляри найкраще підходять для задач, які:

- можуть легко перезапускатися та мають стійкість до переривань (наприклад, пакетна обробка даних, машинне навчання, аналітика, CI/CD);
- мають гнучкий графік запуску й не є критичними щодо часу виконання.

1.5 Основні підходи до масштабування Kubernetes кластеру

1.5.1 Горизонтальне та вертикальне масштабування

Горизонтальне масштабування – це додавання або вилучення екземплярів, зокрема подів або вузлів. Kubernetes ефективно виконує горизонтальне масштабування, особливо для сервісів, де кожна репліка виконує ту саму функцію, і додавання нових екземплярів лінійно підвищує продуктивність.

Вертикальне масштабування означає зміну параметрів ресурсоємності контейнера – наприклад, підвищення ліміту оперативної пам’яті або CPU. У Kubernetes це реалізується через зміну resource requests/limits у специфікації пода. Компонент Vertical Pod Autoscaler (VPA) може рекомендувати або автоматично змінювати ці значення. Проте така зміна зазвичай потребує перезапуску пода, що може вплинути на доступність. Тому вертикальне масштабування частіше використовується для довгострокової оптимізації ресурсів, а не для миттєвого реагування на навантаження.

Обидва підходи виконують різні функції. Горизонтальне масштабування дозволяє швидко реагувати на пікові навантаження, тоді як вертикальне дає змогу економніше використовувати ресурси протягом тривалого часу. Їхня взаємодія важлива: наприклад, якщо поди неправильно розмірені, HPA може неефективно реагувати на метрики.

1.5.2 Масштабування подів і вузлів

У Kubernetes масштабуються як поди, так і вузли, проте механізми та контролери для цих процесів є різними [8]. Масштабування подів реалізується через об’єкти типу Deployment або StatefulSet шляхом зміни кількості реплік. Якщо масштабування відбувається автоматично, то його виконує HPA або інший контролер, який аналізує поточні метрики. Важливо

пам’ятати, що масштабування подів передбачає наявність вільних ресурсів на вузлах. Якщо таких ресурсів немає, поди залишаються у стані «Pending».

Масштабування вузлів відбувається за допомогою Cluster Autoscaler (CA) або аналогічних інструментів, таких як Karpenter. Масштабувальник вузлів відстежує поди, які не можуть бути розміщені через недостатність ресурсів в кластері та ініціює створення нових EC2 інстансів у відповідній групі вузлів або автономно. Цей процес триває довше, ніж запуск подів, оскільки створення нової віртуальної машини потребує більше часу. При видаленні вузлів, конктролер аналізує використання ресурсів й видаляє малозавантажені екземпляри, попередньо видаляючи поди, запущені на них.

Ці два механізми працюють незалежно, але мають бути узгоджені. Наприклад, HPA може створити нові поди швидше, ніж CA встигне забезпечити нові вузли, в результаті чого з’являється черга очікування. У зворотному випадку вузли можуть бути створені без достатньої кількості подів, що призведе до нераціонального використання ресурсів.

1.5.3 Проблема холодного старту

Проблема холодного старту (cold start) є однією з архітектурних і експлуатаційних перешкод при реалізації автоматичного масштабування. Вона виникає у ситуаціях, коли система повинна швидко реагувати на зростання навантаження, але через час, необхідний для ініціалізації ресурсів, не може забезпечити миттєве масштабування [9].

У контексті Kubernetes проблема найчастіше проявляється під час створення нових екземплярів вузлів або запуску нових подів. Затримка включає в себе час на:

- розгортання нового вузла;
- ініціалізацію операційної системи та kubelet-агента;
- завантаження необхідних контейнерних образів з реєстрів;
- запуск подів і готовність сервісів до обробки запитів.

У середньому, повний цикл створення нового вузла може тривати від кількох секунд до кількох хвилин. Особливого значення ця проблема набуває в середовищах, де масштабування ініціюється подіями зі зовнішніх джерел (наприклад, черги повідомлень або брокери подій). У таких випадках навіть незначна затримка в запуску обчислювальних ресурсів може мати суттєвий вплив на загальний час обробки подій або впливати на порядок виконання транзакцій.

Для вирішення проблеми холодного старту застосовуються різноманітні підходи:

- попереднє прогрівання (warm pools): збереження частини вузлів у стані очікування або зниженого споживання ресурсів;
- оптимізація образів контейнерів: зменшення їх розміру і кількості залежностей для пришвидшення завантаження;
- попереднє розміщення подів щоб підготувати середовище до навантаження заздалегідь.

1.5.4 Масштабування орієнтоване на події

Масштабування орієнтоване на події (event-driven scaling) є одним з підходів до динамічного керування ресурсами в Kubernetes, при якому ініціація масштабування відбувається у відповідь на зовнішні або внутрішні події. На відміну від традиційного масштабування, що базується на метриках використання ресурсів (наприклад, CPU або пам’яті), цей тип масштабування активується внаслідок надходження повідомлень, змін стану черг, або інших бізнес-орієнтованих тригерів [10].

Цей підхід є особливо ефективним для мікросервісних і безсерверних архітектур, де робоче навантаження може коливатися раптово і непередбачувано. Він забезпечує більш адаптивну реакцію системи на події, дозволяючи автоматично масштабувати певні компоненти лише тоді, коли виникає фактична потреба в обробці даних.

В Kubernetes реалізація подієвого масштабування можлива завдяки таким інструментам, як KEDA[11]. KEDA виступає проміжною ланкою між джерелами подій і механізмами масштабування Kubernetes, дозволяючи розширювати кількість подів на основі довжини черги повідомлень у AWS SQS, кількості повідомлень у Kafka або кількості подій у Prometheus. Подієво-орієнтоване масштабування має низку переваг:

- економія ресурсів: додатки масштабуються лише тоді, коли це потрібно, що знижує витрати;
- гнучкість у роботі з асинхронними системами: добре підходить для обробки повідомлень, подій, завдань з черги.

У контексті AWS це масштабування може бути тісно інтегроване з сервісами на зразок Amazon SQS, Amazon Kinesis, DynamoDB Streams або CloudWatch Events, що дозволяє створювати глибоко інтегровані подієві архітектури. У такому підході Kubernetes-кластер виступає як обчислювальний рівень, який активується лише при надходженні навантаження, що оптимізує використання ресурсів у масштабованому хмарному середовищі.

1.5.5 Оптимізація витрат

Автоматичне масштабування має не лише забезпечувати відповідність обчислювальних ресурсів поточному навантаженню, а й сприяти економічній ефективності за рахунок зменшення зайвих витрат. Оптимізація витрат у контексті масштабування Kubernetes-кластерів передбачає використання стратегій, які дозволяють мінімізувати витрати на обчислювальні ресурси, зберігаючи при цьому необхідний рівень продуктивності.

Основні підходи до оптимізації витрат включають:

- використання Spot-екземплярів. AWS надає можливість запуску EC2-екземплярів за значно зниженою вартістю (до 90% дешевше за On-Demand), проте з можливістю їхнього переривання. Інтеграція Spot

екземплярів у кластері дозволяє масштабувати робоче навантаження економічно, за умови належного керування ризиками переривання [12];

- правильне визначення ресурсних запитів. Надмірно високі запити на ресурси можуть призводити до неоптимального розміщення подів і, відповідно, до необхідності запуску додаткових вузлів. Застосування Vertical Pod Autoscaler (VPA) допомагає визначити оптимальні значення запитів та лімітів, що дозволяє більш ефективно використовувати наявні ресурси;

- масштабування вниз. Автоматичне зменшення кількості вузлів або подів, коли навантаження знижується, дозволяє уникнути витрат на невикористовувані ресурси. Проте масштабування вниз повинно відбуватися з урахуванням стабільності системи та запобігання проблемі надмірних коливань;

- оптимальний розмір вузлів. Вибір відповідного типу інстансів для робочих навантажень дозволяє зменшити фрагментацію ресурсів. Використання різних типів інстансів в Auto Scaling Groups або використання контролерів, таких як Karpenter, дає можливість запуску вузлів, що найбільш точно відповідають вимогам подів.

2 ОГЛЯД ЗАСОБІВ МАСШТАБУВАННЯ KUBERNETES-КЛАСТЕРУ

У Kubernetes реалізовано як вбудовані, так і зовнішні механізми для масштабування так як:

- Horizontal Pod Autoscaler (HPA) – здійснює горизонтальне масштабування подів на основі метрик;
- Vertical Pod Autoscaler (VPA) – коригує запити до ресурсів подів, забезпечуючи вертикальне масштабування;
- Cluster Autoscaler (CA) – додає та вилучає обчислювальні вузли кластера;
- KEDA (Kubernetes Event-Driven Autoscaler) – дозволяє масштабувати поди на основі зовнішніх подій;
- Karpenter – сучасний інструмент масштабування від AWS, який на відміну від CA не потребує попередньо визначених груп вузлів і забезпечує швидше та гнучкіше масштабування в середовищі AWS.

Кожен із цих інструментів виконує свою роль у контексті масштабування кластерів у хмарній інфраструктурі. У наступних підрозділах буде детально описано механізм роботи кожного з них, а також їхнє застосування у середовищі AWS.

2.1 Horizontal Pod Autoscaler (HPA)

HPA є вбудованим контролером Kubernetes (частина компонента kube-controller-manager), що автоматично керує кількістю реплік подів для цільового ресурсу (Deployment або ReplicaSet) на основі моніторингу метрик. До них належать зокрема CPU та пам'ять.

Горизонтальний масштабувальник функціонує за принципом циклу, який повторюється з певною періодичністю та на кожній ітерації цього циклу виконуються такі дії:

- контроллер отримує поточні значення метрик для цільових подів через Kubernetes Metrics Server API;

- контроллер обчислює кількість реплік за співвідношенням поточних та цільових значень метрик;

- застосовує обмеження, зазначені в конфігурації – не перевищуючи максимальну кількість реплік і не зменшуючи нижче мінімальної;

- якщо отримана кількість реплік відрізняється від поточної, оновлює відповідне значення в цільовому ресурсі через Kubernetes API;

- у випадку, якщо використовується декілька метрик, контролер розраховує кількість реплік для кожної метрики окремо, а потім обирає максимальне значення, яке задовольняє всі метрики.

Попри простоту використання, HPA має певні обмеження. Перш за все, його реактивний характер призводить до того, що нестабільні навантаження можуть спричинити проблему «холодного старту». Крім того, HPA базується на метриках ресурсів (CPU, пам'ять), які не завжди є прямими індикаторами реального навантаження на сервіс. Ще одним недоліком є відсутність координованого масштабування: HPA працює окремо для кожного сервісу та не враховує залежності між ними. У середовищі AWS EKS HPA функціонує аналогічно до звичайних Kubernetes-кластерів. Джерелом метрик зазвичай виступає Metrics Server. Для інтеграції із CloudWatch існують адаптери, що дозволяють використовувати специфічні для середовища метрики як тригери масштабування.

2.2 Vertical Pod Autoscaler (VPA)

VPA – це додатковий інструмент для Kubernetes призначений для автоматичного коригування запитів на ресурси (CPU, пам'ять) для окремих подів. На відміну від горизонтального автоскейлера (HPA), що змінює кількість подів, VPA працює над тим, щоб кожен под мав оптимальну кількість ресурсів достатню для стабільної роботи, але без надмірних витрат

ресурсів [13].

VPA складається з трьох головних компонентів:

- Recommender – аналізує історичні та поточні показники використання ресурсів подами та генерує рекомендації щодо оптимальних значень ресурсів (цільове значення, нижня і верхня межі);

- Updater – за необхідності (в активному режимі) може видалити поди, чиї поточні ресурси суттєво відрізняються від рекомендованих, дозволяючи їм перезапуститись з оновленими параметрами;

- Admission Controller – автоматично модифікує запити на ресурси в специфікації подів під час їх планування або перезапуску, використовуючи останні рекомендації.

Головним сценарієм використання VPA є ситуації, коли складно вручну підібрати ресурси для подів, або коли поведінка робочих навантажень змінюється з часом. За допомогою VPA досягається баланс, коли поди отримують рівно стільки ресурсів, скільки їм потрібно, що дозволяє уникати ситуацій з недостатнім виділенням ресурсів та надлишковим резервуванням, що спричиняє зайві витрати.

Проте використання VPA зустрічається рідше, ніж HPA, з кількох причин:

- VPA несумісний із HPA для одного й того самого набору подів, адже HPA очікує однакові поди з фіксованими параметрами ресурсів, а VPA змінює ці параметри з часом;

- видалення подів для зміни їх параметрів може спричинити порушення в роботі, що робить VPA кращим для стабільних і довготривалих задач, аніж для тих, які мають різке змінення навантаження.

VPA є менш поширеним, проте має свою важливу перевагу. Модуль Recommender вивчає історичні дані та рекомендує підходящі значення ресурсів, що робить VPA цінним інструментом для знаходження оптимального балансу ресурсів.

Загалом, у контексті стратегії масштабування Kubernetes VPA відіграє

допоміжну роль, дозволяючи ефективніше використовувати наявні ресурси.

2.3 Cluster Autoscaler

Cluster Autoscaler – це компонент, що автоматизує масштабування робочих вузлів кластера Kubernetes. Його архітектура базується на інтеграції з API Kubernetes, що дозволяє здійснювати моніторинг поточного завантаження кластеру. Завдяки цьому здійснюється аналіз розподілу ресурсів, а також враховуються як поточні запити на розгортання нових подів, так і можливості для оптимізації існуючих ресурсів. При виявленні недостачі ресурсів Cluster Autoscaler ініціює процес масштабування, звертаючись до відповідних сервісів хмарного провайдера для додавання нових вузлів. Аналогічно, при виявленні надлишкових ресурсів, коли вузли не використовуються ефективно, відбувається їх послідовне усунення із кластеру [14].

Одним із недоліків CA є обмеження на масштабування лише в межах одного типу вузлів. Хоча технічно можливо використовувати кілька типів EC2 екземплярів, вони мають бути однаковими за обсягом ресурсів (CPU і пам’яті). Ще одне обмеження полягає в тому, що контролер неефективно виконує оптимізацію існуючих ресурсів. Окрім цього, Cluster Autoscaler потребує попереднього створення груп вузлів у хмарному провайдері — за межами самого кластера — та ручного внесення їх до конфігурації. Це підвищує адміністративне навантаження, особливо при роботі з навантаженнями, які вимагають спеціалізованих вузлів — наприклад, EC2 екземплярів із великою кількістю ресурсів, з графічними прискорювачами або з відмінною архітектурою процесора. Усі ці типи вузлів потрібно створювати та підтримувати вручну, що ускладнює гнучке масштабування кластеру.

2.4 Kubernetes Event-Driven Autoscaler

KEDA – це контролер, що розширює можливості стандартного HPA, надаючи можливість масштабувати поди на основі зовнішніх подій. Принцип роботи KEDA полягає у створенні спеціального ресурсу ScaledObject або ScaledJob, який пов’язує цільовий Deployment із зовнішнім джерелом подій. KEDA містить набір вбудованих компонентів, що інтегруються із зовнішніми системами, такими як AWS SQS, Kafka, Prometheus та інші. Контролер KEDA перевіряє зовнішнє джерело, обчислює потрібну метрику і оновлює внутрішній об'єкт HPA, який вже безпосередньо відповідає за зміну кількості подів.

Однією з особливостей KEDA є можливість масштабування до нуля. Якщо події відсутні, KEDA може автоматично зменшити кількість подів до нуля, а при надходженні нових подій – швидко підняти їхню кількість. Таким чином, реалізуються безсерверні моделі, коли поди взагалі не існують в моменти простою, чого стандартний HPA забезпечити не може через потребу постійних метрик з активних подів.

KEDA складається з наступних компонентів:

- Scaler - визначає спосіб з’єднання з джерелом подій та умови, за яких ініціюється масштабування (наприклад, кількість повідомлень у черзі на один под);
- Controller - головний цикл, що спостерігає за зовнішніми подіями та приймає рішення про масштабування;
- Metrics Adapter - компонент, що надає спеціальні метрики.

KEDA підтримує багато зовнішніх джерел подій, таких як AWS SQS, AWS Kinesis, RabbitMQ, Kafka, Azure Queue, Azure EventHub, Google PubSub та Prometheus. Наприклад, використовуючи Prometheus, можливо масштабувати поди за будь-якими метриками, які можуть бути виражені через запити PromQL.

У контексті AWS використання KEDA дозволяє значно покращити масштабування асинхронних сервісів. Наприклад, якщо черга повідомлень

AWS SQS різко зростає, KEDA може швидко запустити необхідну кількість подів, не чекаючи, поки зростуть метрики CPU чи пам’яті.

Поширеними випадками застосування KEDA є безсерверні архітектури або event-driven застосунки, зокрема фонові обробники, задачі batch-обробки, а також деякі веб-застосунки, якщо навантаження можна інтерпретувати через спеціальні метрики. Важливо враховувати, що періодичне опитування зовнішніх систем може додатково навантажувати API зовнішніх сервісів.

2.5 Karpenter

Karpenter – це сучасний контролер для автоматичного масштабування вузлів у Kubernetes, який надає гнучкі та широкі можливості керування ресурсами у порівнянні із вбудованим Cluster Autoscaler. Його ключова перевага полягає у тісній інтеграції з API постачальників хмарних сервісів, що дозволяє контролеру аналізувати планування подів у реальному часі [15].

На відміну від традиційного підходу, де вузли додаються лише після виявлення нерозміщених подів, Karpenter постійно відстежує характеристики навантаження (запити на CPU, пам’ять, GPU, мережеві ресурси) й оперативно знаходить оптимальну конфігурацію екземплярів. Це дає змогу мінімізувати затримку при спробі розміщення подів, знижуючи ризик виникнення проблеми холодного старту вузлів.

Контролер дозволяє комбінувати On-Demand та Spot-екземпляри для балансу між доступністю та економією коштів. У процесі вибору екземпляра також можуть враховуватися топологічні обмеження (наприклад, необхідність розмістити вузли в певній зоні доступності).

Karpenter постійно аналізує поточний стан завантаження вузлів і може ініціювати впорядкування подів, завдяки чому відбувається вивільнення надлишкових або слабко завантажених вузлів. Це дає змогу зменшити кількість працюючих інстансів та, відповідно, витрати на інфраструктуру

Ще однією перевагою є можливість гнучкого налаштування політик

масштабування, що дозволяє враховувати специфіку різних застосунків. Karpenter здатний адаптуватися до різних сценаріїв використання, забезпечуючи оптимальне розподілення ресурсів між критичними та менш пріоритетними задачами. Це сприяє підвищенню загальної продуктивності системи та зменшенню ризику виникнення «вузьких місць» у роботі інфраструктури.

Одним із основних аспектів є необхідність ретельного налаштування параметрів масштабування. Неправильне визначення порогових значень може призвести до перевищення або недостатнього використання ресурсів, що негативно впливає на ефективність роботи системи.

Також важливо враховувати сумісність Karpenter з іншими компонентами хмарної інфраструктури. Оскільки система тісно інтегрована з AWS, необхідно враховувати особливості роботи різних типів EC2 екземплярів та можливі обмеження в певних регіонах або з певними конфігураціями. Додатково, у випадку високої динамічності навантаження можуть виникати ситуації, коли швидкість масштабування не встигає відповідати темпам зростання запитів, що вимагає впровадження додаткових механізмів моніторингу та оптимізації самого контролера.

3 ПОБУДОВА ІНФРАСТРУКТУРИ ТА РОЗРОБКА МЕТОДІВ АВТОМАТИЧНОГО МАСШТАБУВАННЯ

3.1 Розгортання інфрастуктури в хмарному середовищі AWS

Підхід до керування інфраструктурою, відомий як Infrastructure-as-Code (IaC), передбачає автоматизоване створення та конфігурацію комп'ютерних ресурсів через декларативні конфігураційні файли. Такий підхід замінює ручні налаштування інфраструктури на код, який може бути версійований, тестований і повторно використовуваний. Основними перевагами IaC є автоматизація процесів, зниження ймовірності людських помилок, покращення консистентності середовищ і спрощення масштабування ресурсів.

Одним із популярних інструментів для реалізації цього підходу є Terraform від компанії HashiCorp. Terraform дозволяє описувати необхідну інфраструктуру декларативним чином у вигляді конфігураційних файлів, написаних мовою HashiCorp Configuration Language (HCL). Основна перевага Terraform полягає у підтримці широкого спектру хмарних провайдерів, включаючи Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform та інші.

Конфігураційні файли Terraform складаються із трьох ключових компонентів: ресурсів (resources), провайдерів (providers) та модулів (modules). Ресурси є базовими компонентами, які описують конкретні інфраструктурні об'єкти, такі як віртуальні машини, мережі або бази даних. Провайдери відповідають за комунікацію Terraform з хмарними сервісами. Наприклад, провайдер AWS надає можливість керувати сервісами EC2, S3, VPC тощо. Модулі у Terraform використовуються для повторного використання типових конфігурацій. Це спрощує код, дозволяє уникнути дублювання і підвищує швидкість розгортання інфраструктури.

Типовий робочий процес Terraform включає три етапи: ініціалізацію (terraform init), планування (terraform plan) та застосування конфігурації (terraform apply). Під час ініціалізації Terraform завантажує необхідні провайдери та модулі. На етапі планування користувач отримує інформацію про ресурси, які будуть створені або змінені. Після підтвердження змін конфігурація застосовується, створюючи реальні ресурси у відповідному хмарному середовищі.

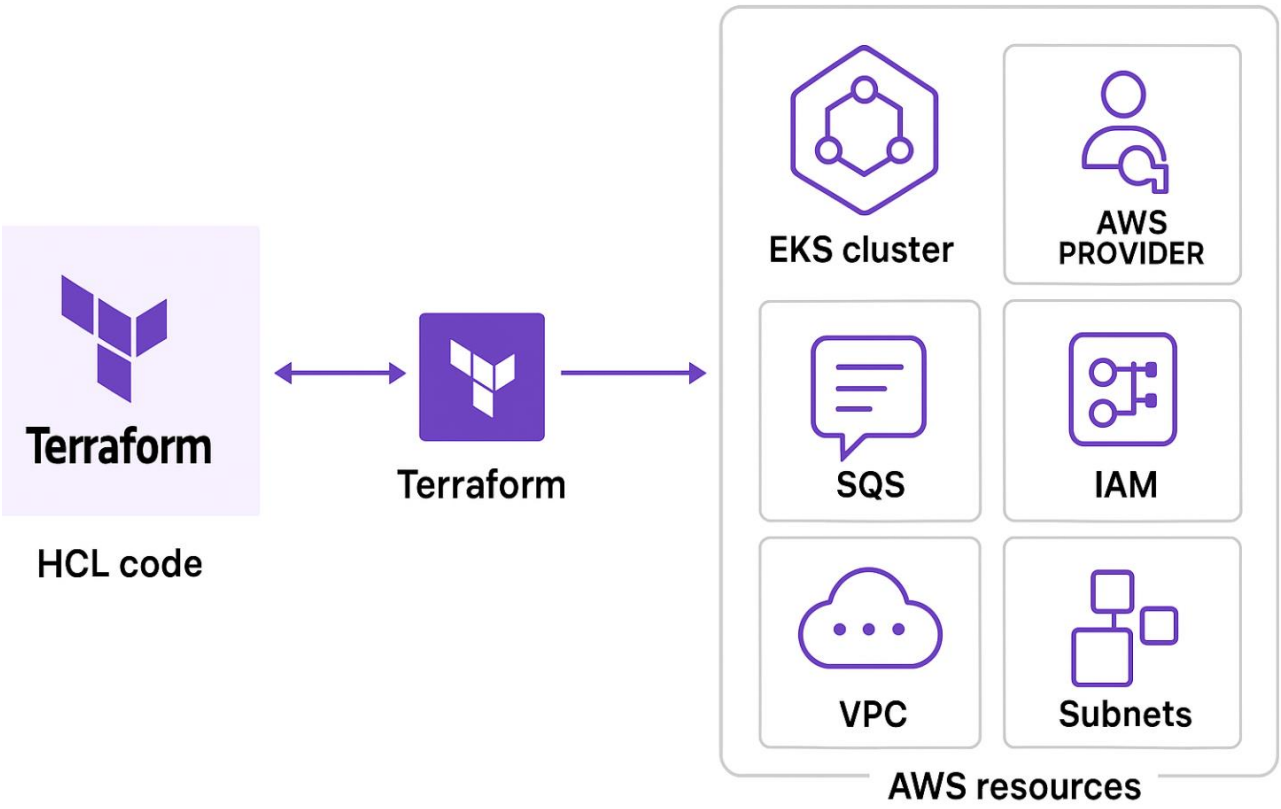

Рисунок 3.1 – Діаграма взаємодії компонентів Terraform

Для встановлення компонентів безпосередньо всередині Kubernetes-кластеру зазвичай використовують менеджер пакетів Helm. У Terraform для цієї задачі використовується ресурс helm_release, який дозволяє автоматично розгортати Helm-чарти з визначеними параметрами конфігурації. Таким чином, компоненти для автоматичного масштабування кластеру та подів, такі як Cluster-Autoscaler, KEDA та Karpenter, можуть бути зручно встановлені та

налаштовані безпосередньо через код Terraform.

У межах цієї роботи використані публічні модулі Terraform для створення мережевої інфраструктури (terraform-aws-vpc) та Kubernetes-кластеру (terraform-aws-eks).

```
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "~> 5.19.0"

  name = local.basename
  cidr = var.vpc_primary_cidr

  azs             = local.zone_names
  public_subnets  = local.vpc_public_subnet_cidrs
  private_subnets = local.vpc_private_subnet_cidrs

  enable_nat_gateway = true
  single_nat_gateway = true

  public_subnet_tags = {
    "kubernetes.io/role/elb" = "1"
  }

  private_subnet_tags = {
    "kubernetes.io/role/internal-elb" = "1"
    "karpenter.sh/discovery"          = local.eks_cluster_name
  }
}
```

Рисунок 3.2 – HCL код для створення ресурсів VPC

Таким чином, використання Terraform разом з Helm-чартами для розгортання інфраструктури та компонентів Kubernetes у хмарному середовищі AWS забезпечує швидкість, надійність і можливість повторного використання конфігурацій, що сприяє зменшенню ризиків, пов’язаних із ручними налаштуваннями.

3.2 Архітектура EKS кластеру та його компоненти

Розгортання Kubernetes-кластеру у хмарному середовищі AWS передбачає використання архітектури, яка забезпечує надійність, безпеку та ефективність роботи контейнерних додатків. Типова архітектура складається з декількох взаємопов'язаних компонентів, які включають мережеву інфраструктуру (VPC), контрольну площину Kubernetes, керовані групи вузлів, а також додаткові сервіси.

Віртуальна приватна мережа (VPC) є фундаментом мережевої архітектури. У межах VPC створюються приватні та публічні підмережі:

- приватні підмережі використовуються для розміщення компонентів Kubernetes-кластеру: контрольної площини EKS (Control Plane) та вузлів кластеру (Node Groups). Це забезпечує додаткову безпеку, оскільки вузли та контрольна площина не мають прямого доступу з Інтернету;
- публічні підмережі використовуються для компонентів, що мають взаємодіяти з Інтернетом, таких як балансувальники навантаження Application Load Balancer (ALB) та Network Load Balancer (NLB), а також публічні інтерфейси для взаємодії з API кластеру.

Для забезпечення доступу до інтернет-ресурсів із приватних підмереж використовується NAT Gateway, що маршрутизує трафік із приватних підмереж у зовнішній інтернет. Натомість, доступ з Інтернету до публічних підмереж забезпечує Internet Gateway.

Центральним елементом архітектури Kubernetes є контрольна площина EKS (Control Plane), яка повністю керується AWS. Control Plane включає такі ключові компоненти Kubernetes, як API Server, Scheduler, Etcd та Controller Manager. AWS також надає додатки (EKS Addons), які автоматично встановлюються та підтримуються сервісом EKS. У середовищі Kubernetes додатками (add-ons) називають допоміжне програмне забезпечення, яке забезпечує функціональні можливості, необхідні для роботи кластерів. До таких компонентів належать драйвери, що дозволяють Kubernetes-кластеру

взаємодіяти з базовими обчислювальними, мережевими й дисковими ресурсами хмарного провайдера AWS. До них належать зокрема:

- CoreDNS – DNS-сервіс Kubernetes;
- kube-proxy – забезпечує мережеву взаємодію між подами в кластері;
- VPC CNI – AWS-плагін для мережевої інтеграції подів з AWS VPC;
- eks-pod-identity – інтеграція IAM-ідентифікації для подів;
- AWS EBS CSI driver – драйвер для роботи з Amazon Elastic Block Store.

Для розміщення робочих навантажень використовуються дві керовані групи вузлів EKS: system та worker. Вони дозволяють автоматично керувати життєвим циклом вузлів, включаючи створення, оновлення та видалення. Кожна керована група вузлів складається з набору EC2-інстансів, об’єднаних у Auto Scaling Group (ASG).

EC2-інстанси у керованих групах вузлів базуються на офіційних образах операційної системи Amazon Linux, яка спеціально оптимізована для запуску Kubernetes. На цих вузлах встановлюється та працює агент kubelet, який відповідає за взаємодію подів з Control Plane, а також інші необхідні компоненти, такі як Container Runtime.

Вузли для системних компонентів (група вузлів "system") і вузли для робочих навантажень ("worker") використовуються для різних типів навантаження. Використання кількох ізольованих керованих груп вузлів забезпечує низку переваг:

- достеменність експерименту – розподіл навантажень і системних компонентів на різні вузли дозволяє проводити чіткіші вимірювання ефективності масштабування робочого навантаження;
- безпека та стабільність – окрема група вузлів для критично важливих компонентів сприяє стабільності кластеру та зменшує ризик виходу з ладу системних компонентів спричинений робочими навантаженням.

Для зручного розподілу навантаження серед вузлів використовуються Labels та Taints, які дозволяють планувальнику Kubernetes визначати

відповідність подів конкретним вузлам. Taints використовуються для керування тим, які саме поди можуть бути розміщені на певних вузлах. Це дозволяє, наприклад, уникнути випадкового розміщення робочого навантаження на системних вузлах.

Таким чином, запропонована архітектура EKS-кластеру в AWS забезпечує чітке розмежування та ізоляцію компонентів, високу доступність і масштабованість, спрощує адміністрування, а також створює ефективні умови для автоматичного масштабування кластеру та навантажень.

На рисунку 3.3 зображена верхнерівнева діаграма (HLD) архітектури інфраструктури.

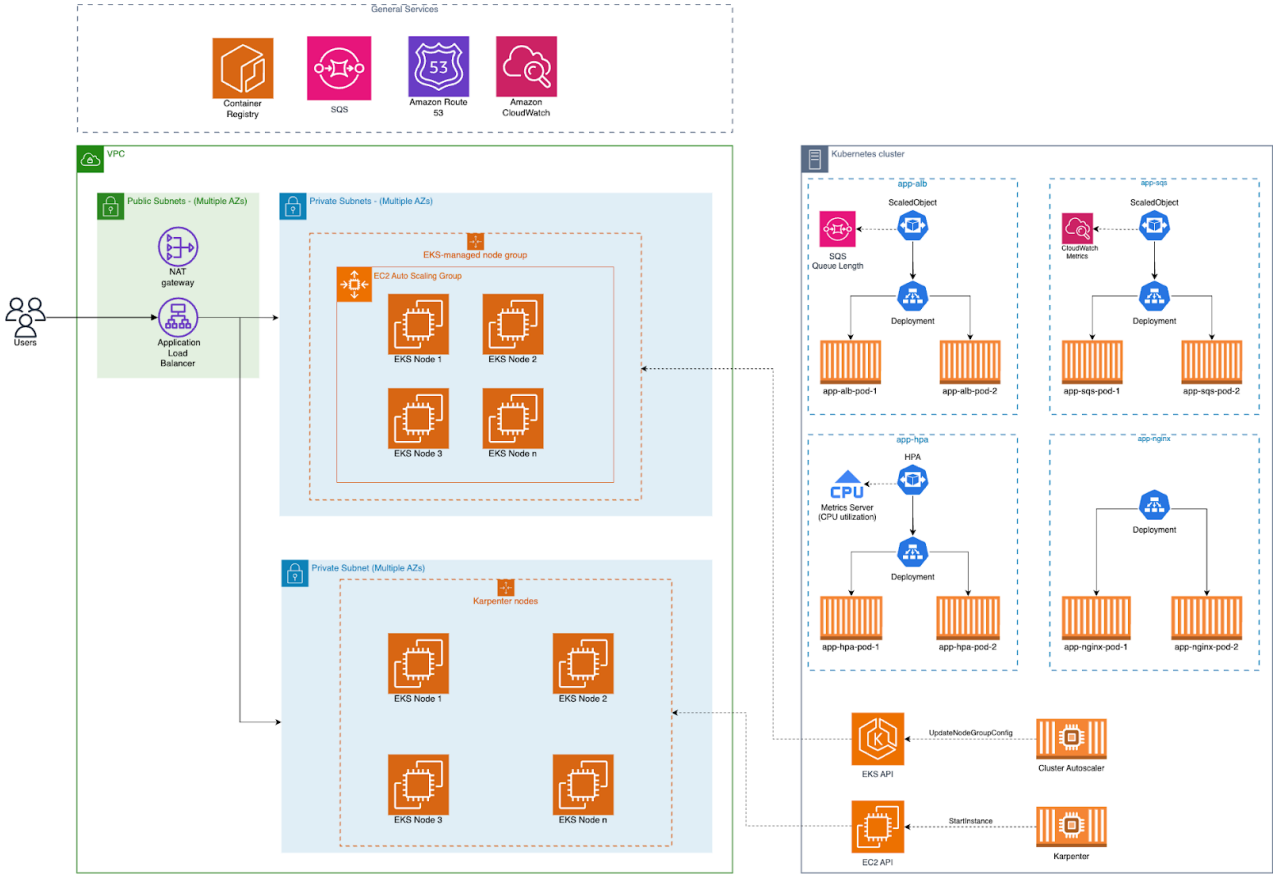

Рисунок 3.3 – Верхнерівнева діаграма архітектури інфраструктури

3.3 Конфігурація Cluster Autoscaler

Cluster Autoscaler — це компонент Kubernetes, призначений для автоматичного масштабування груп відповідно до поточного навантаження в

кластері. У хмарному середовищі AWS його інтеграція з EKS дозволяє динамічно додавати або видаляти EC2-екземпляри залежно від кількості нерозміщених подів або неефективного використання ресурсів.

Для встановлення Cluster Autoscaler використовується менеджер пакетів Helm. Для розгортання Helm-чарту використовується наступна конфігурація Terraform (лістинг 3.1).

Лістинг 3.1 – Конфігурація встановлення Cluster-autoscaler

```
resource "helm_release" "cluster_autoscaler" {
  name       = "cluster-autoscaler"
  repository = "https://kubernetes.github.io/autoscaler"
  chart      = "cluster-autoscaler"
  namespace  = "kube-system"
  version    = "9.46.6"

  values = [
    yamlencode({
      autoDiscovery = {
        clusterName = module.eks.cluster_name
      }
      awsRegion = var.aws_region
      rbac = {
        create = true
        serviceAccount = {
          create = true
          name   = local.eks_cluster_autoscaler_sa
        }
      }
      nodeSelector = local.eks_system_node_selector
      tolerations  = local.eks_system_node_group_tolerations
      extraArgs = {
        logtostderr              = true
        stderrthreshold          = "info"
        v                        = 4
        scale-down-unneeded-time = "2m"
      }
      serviceMonitor = {
        enabled = true
      }
    })
  ]

  depends_on = [module.eks]
}
```

Параметр конфігурації autoDiscovery.clusterName дозволяє компоненту автоматично визначати і взаємодіяти з відповідними групами вузлів, створеними для EKS-кластеру, без необхідності ручного додавання кожної ASG.

Механізм роботи Cluster Autoscaler складається з двох основних процесів:

- масштабування вверх. Коли в кластері з'являється новий под, який не може бути розміщений на існуючих вузлах через брак ресурсів (недостатньо CPU чи пам'яті), Cluster Autoscaler виявляє цю ситуацію і надсилає запит до AWS API для збільшення кількості вузлів у відповідній ASGВона створює новий EC2-екземпляр, який автоматично додається до кластеру, що дозволяє планувальнику розмістити под на цьому новому вузлі;
- масштабування вниз. Cluster Autoscaler регулярно перевіряє вузли на предмет їх завантаженості. Якщо вузли виявляються недовантаженими або протягом певного періоду часу на них немає подів, то контролер позначає такі вузли як непотрібні та поступово видаляє їх з кластеру, зменшуючи кількість вузлів у відповідній ASG.

Параметри конфігурації дозволяють включити моніторинг метрик за допомогою Prometheus, що спрощує подальшу діагностику і аналіз поведінки Cluster Autoscaler.

3.4 Конфігурація Karpenter

Karpenter є динамічним масштабувальником вузлів для Kubernetes, який дозволяє швидко й ефективно реагувати на зміни навантаження у кластері, створюючи необхідні екземпляри EC2 з оптимальними характеристиками. Основною відмінністю Karpenter від EKS-managed node groups є гнучкість та швидкість реагування на зміни. Якщо EKS-managed node groups побудовані на ASG (Auto Scaling Groups) з фіксованими типами екземплярів, то Karpenter використовує AWS EC2 API для запуску інстансів

напряму. Він здатний динамічно створювати екземпляри різних типів і конфігурацій, враховуючи вимоги подів щодо ресурсів і вузлових міток.

Для встановлення Karpenter в EKS-кластері використовується Helm chart, що забезпечує легку інсталяцію і подальшу конфігурацію компонентів (лістинг 3.2).

Лістинг 3.2 – Конфігурація встановлення Karpenter

```
resource "helm_release" "karpenter_crd" {
  name                = "karpenter-crd"
  namespace           = module.karpenter.namespace
  create_namespace    = true
  repository          = "oci://public.ecr.aws/karpenter"
  chart               = "karpenter-crd"
  version             = "1.5.0"
  repository_username =
data.aws_ecrpublic_authorization_token.token.user_name
  repository_password =
data.aws_ecrpublic_authorization_token.token.password
  depends_on          = [module.eks]
}

resource "helm_release" "karpenter" {
  name                = "karpenter"
  namespace           = module.karpenter.namespace
  create_namespace    = true
  repository          = "oci://public.ecr.aws/karpenter"
  chart               = "karpenter"
  version             = helm_release.karpenter_crd.version
  repository_username =
data.aws_ecrpublic_authorization_token.token.user_name
  repository_password =
data.aws_ecrpublic_authorization_token.token.password

  values = [
    yamlencode({
      serviceAccount = {
        name = module.karpenter.service_account
      }
      settings = {
        clusterName       = module.eks.cluster_name
        clusterEndpoint   = module.eks.cluster_endpoint
        interruptionQueue = module.karpenter.queue_name
        featureGates = {
          spotToSpotConsolidation = true
        }
      }
      tolerations = local.eks_system_node_group_tolerations
      serviceMonitor = {
```

```
        enabled = true
      }
    })
  ]

  depends_on = [helm_release.karpenter_crd]
}
```

Розгортання контролера складається з двох етапів: створення CRD (Custom Resource Definitions) та власне інсталяції Karpenter з необхідними налаштуваннями, такими як інтеграція з кластером (назва кластеру та API endpoint), IAM-ролями, чергою переривань для роботи зі spot-інстансами і моніторингом через Prometheus.

Конфігурація Karpenter визначається через два основні ресурси:

- EC2NodeClass – описує конфігурацію EC2-інстансів, які будуть створюватися;

- NodePool – ресурс, який визначає групу вузлів з певними вимогами щодо типу інстансів та конфігурації.

Було створено два типи вузлів керованих контролером Karpenter:

- worker — призначений для стандартного навантаження, з фіксованим типом екземплярів (m5.large), що працюють в режимі on-demand;

- worker-spot — оптимізований для економії коштів, використовує інстанси spot, з можливістю вибору серед різних типів (категорії m, c, r, t), а також обмеженням по поколінню (від 5-го і вище).

Обидва NodePool-и конфігуруються з політиками консолідації ресурсів, встановленими мітками для можливості керування розподілом подів на певні пули.

Лістинг 3.3 – Конфігурація EC2NodeClass

```
apiVersion: karpenter.k8s.aws/v1
kind: EC2NodeClass
metadata:
  name: worker
spec:
  amiFamily: AL2023
  amiSelectorTerms:
```

```
  - alias: al2023@latest
  blockDeviceMappings:
  - deviceName: /dev/xvda
    ebs:
      deleteOnTermination: true
      encrypted: true
      volumeSize: 30Gi
      volumeType: gp3
  role: eks-dev-karpenter-node-role-20250602131144488100000000b
  securityGroupSelectorTerms:
  - tags:
      karpenter.sh/discovery: eks-dev
  subnetSelectorTerms:
  - tags:
      karpenter.sh/discovery: eks-dev
```

Лістинг 3.4 – Конфігурація NodePool

```
apiVersion: karpenter.sh/v1
kind: NodePool
metadata:
  name: worker
spec:
  disruption:
    consolidateAfter: 30s
    consolidationPolicy: WhenEmpty
  limits:
    cpu: "40"
    memory: 80Gi
  template:
    metadata:
      labels:
        pool: worker
    spec:
      expireAfter: 720h
      nodeClassRef:
        group: karpenter.k8s.aws
        kind: EC2NodeClass
        name: worker
      requirements:
      - key: kubernetes.io/arch
        operator: In
        values:
        - amd64
      - key: karpenter.sh/capacity-type
        operator: In
        values:
        - spot
      - key: node.kubernetes.io/instance-type
        operator: In
        values:
        - m5.large
      taints:
```

```
    - effect: NoSchedule
      key: dedicated
      value: worker
```

3.5 Конфігурація KEDA

KEDA (Kubernetes Event-Driven Autoscaling) — це один із контролерів для масштабування застосунків в Kubernetes, який надає можливість масштабування не лише за стандартними ресурсами (CPU та пам’ять), а також за метриками зовнішніх провайдерів. KEDA доповнює стандартний HPA, розширюючи його можливості використанням зовнішніх подій та метрик провайдерів, таких як AWS CloudWatch. Метрики надаються через спеціальний адаптер (Custom Metrics API). KEDA також має прямі інтеграції з багатьма зовнішніми сервісами та підтримує масштабування до нуля, що дозволяє значно ефективніше використовувати ресурси в періоди низького навантаження.

Встановлення KEDA у кластер також здійснюється за допомогою Helm (лістинг 3.5).

Лістинг 3.5 – Розгортання компонентів KEDA

```
resource "helm_release" "keda" {
  name       = "keda"
  repository = "https://kedacore.github.io/charts"
  chart      = "keda"
  namespace  = "kube-system"
  version    = "2.17.1"

  values = [
    yamlencode({
      clusterName = module.eks.cluster_name

      nodeSelector = local.eks_system_node_selector
      tolerations  = local.eks_system_node_group_tolerations

      prometheus = {
        operator = {
          enabled = true
          serviceMonitor = {
            enabled = true
          }
```

```
          podMonitor = {
            enabled = true
          }
        }
      }

      serviceAccount = {
        operator = {
          name = local.eks_keda_operator_sa
        }
      }
    })
  ]

  depends_on = [module.eks]
}
```

У конфігурації ззаначається інтеграція з Prometheus для моніторингу роботи KEDA, а також налаштовується розміщення подів оператора KEDA на вузлах системної.

KEDA дозволяє створювати об'єкти типу ScaledObject, які описують правила масштабування застосунків. Нижче наведено дві конфігурації, використані в роботі:

- масштабування за кількістю запитів в AWS Application Load Balancer (лістинг 3.6). Ця конфігурація автоматично масштабує кількість подів відповідно до кількості запитів, що надходять через Application Load Balancer;

- масштабування за довжиною черги AWS SQS (лістинг 3.7). У цій конфігурації кількість подів змінюється відповідно до довжини черги SQS.

Лістинг 3.6 – Компонент KEDA для масштабування за кількістю запитів ALB

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: app-alb
spec:
  scaleTargetRef:
    name: web-app-deployment
  minReplicaCount: 0
  maxReplicaCount: 10
  cooldownPeriod: 60
```

```
  pollingInterval: 30
  triggers:
  - type: aws-cloudwatch
    metadata:
      namespace: AWS/ApplicationELB
      dimensionName: LoadBalancer
      dimensionValue: "app/my-alb/123456789abcdef0"
      metricName: RequestCountPerTarget
      metricStat: Sum
      targetMetricValue: "50"
      awsRegion: "eu-west-1"
```

Лістинг 3.7 – Компонент KEDA для масштабування за довжиною черги

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: app-sqs
spec:
  scaleTargetRef:
    name: app-sqs
  minReplicaCount: 0
  maxReplicaCount: 15
  pollingInterval: 10
  cooldownPeriod: 60
  triggers:
  - type: aws-sqs-queue
    metadata:
      queueURL: "https://sqs.eu-west-
1.amazonaws.com/123456789012/eks-dev-sqs-queue"
      queueLength: "5"
      awsRegion: "eu-west-1"
```

3.6 Моніторинг Prometheus та Grafana

Ефективне керування Kubernetes-кластером вимагає ретельного моніторингу стану компонентів, аналізу їхньої продуктивності та своєчасного виявлення проблем. У сучасних інфраструктурних рішеннях важливим компонентом є стек моніторингу на основі Prometheus та Grafana. Для інтеграції використовується набір інструментів kube-prometheus-stack, який поєднує в собі компоненти Prometheus, Prometheus Operator, Node Exporter, Grafana та набір ServiceMonitor-конфігурацій. Встановлюються перелічені компоненти також за допомогою одного Helm пакету.

Використання kube-prometheus-stack забезпечує комплексний

моніторинг Kubernetes-кластеру та його компонентів. До його складу входять:

- Node Exporter — забезпечує збір метрик з вузлів, включаючи показники CPU, пам'яті, дискової підсистеми тощо;
- kube-state-metrics — надає детальні метрики щодо стану Kubernetes-об’єктів (поди, вузли, деплойменти);
- Karpenter — збір метрик щодо створення та видалення нод, часу реакції на події, ефективності консолідації;
- Cluster Autoscaler — аналіз частоти спроб масштабування, причин неможливості масштабування, обсягу нерозміщених подів.

Основною перевагою такого моніторингу є централізація інформації про стан усіх компонентів у єдиній системі. Це дає змогу адміністраторам Kubernetes-кластеру легко аналізувати ефективність роботи масштабувальників (Cluster Autoscaler, Karpenter, KEDA), швидко виявляти нештатні ситуації та оперативно реагувати на потенційні проблеми. Grafana надає зручний інтерфейс для візуалізації метрик, які збирає Prometheus. Візуалізація через Grafana також значно спрощує сприйняття інформації та дозволяє зручніше взаємодіяти з моніторинговими даними (рисунок 3.4).

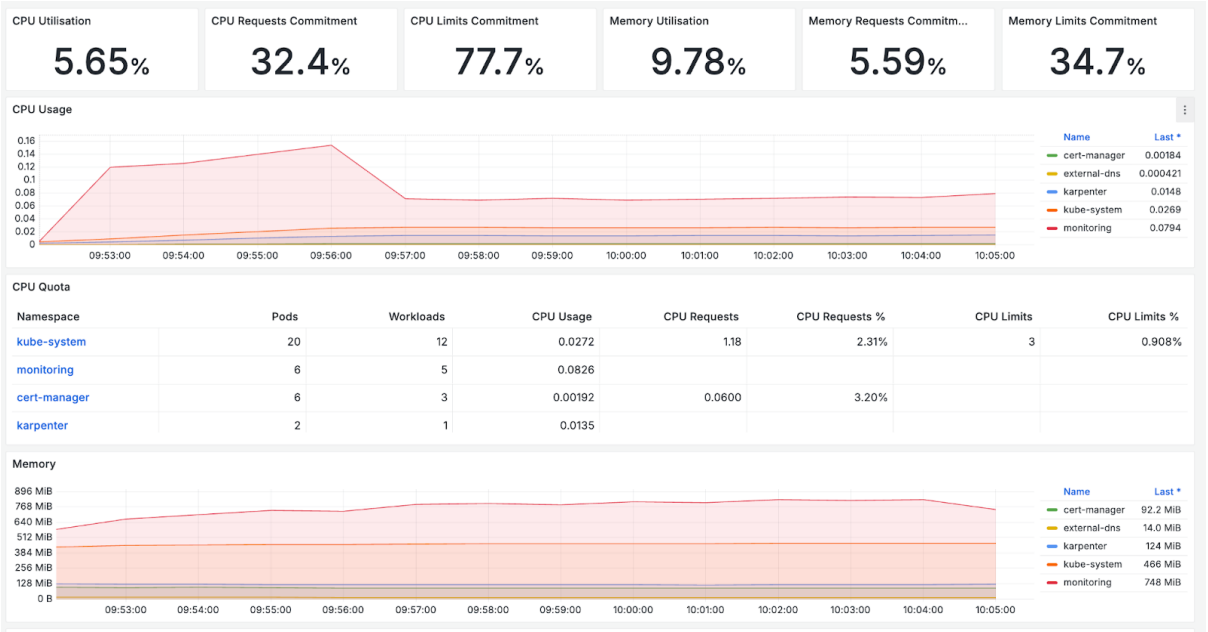

Рисунок 3.4 – Панель моніторингу показників Kubernetes-кластеру

3.7 Розробка застосунків для проведення експериментів

Для експериментальної перевірки ефективності роботи різних масштабувальних механізмів у Kubernetes-кластері було розроблено кілька спеціалізованих тестових застосунків. Кожен із цих застосунків орієнтований на перевірку конкретних сценаріїв масштабування, пов'язаних із використанням Cluster Autoscaler, Karpenter, HPA та KEDA.

Застосунок app-nginx (лістинг 3.8) використовується для тестування роботи Cluster Autoscaler та Karpenter. В його основі лежить звичайний контейнерний образ nginx, який сам по собі не створює значного навантаження, але містить визначені ресурси CPU та пам’яті (requests та limits). Основна мета цього застосунку — перевірка реакції масштабувальників на появу нових подів і створення нових вузлів, що дозволяє оцінити швидкість реакції та ефективність роботи механізмів масштабування.

Лістинг 3.8 – Деплоймент ресурс app-nginx

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app-nginx
spec:
  replicas: 0
  selector:
    matchLabels:
      app: app-nginx
  template:
    metadata:
      labels:
        app: app-nginx
    spec:
      nodeSelector:
        pool: worker-spot
      tolerations:
        - key: dedicated
          operator: Equal
          value: worker-spot
          effect: NoSchedule
      containers:
        - name: nginx
```

```
          image: nginx:latest
          imagePullPolicy: Always
          resources:
            requests:
              cpu: 500m
              memory: 256Mi
            limits:
              cpu: 500m
              memory: 256Mi
```

Інший застосунок, app-hpa, призначений для перевірки масштабування за завантаженням CPU за допомогою стандартного Horizontal Pod Autoscaler (HPA). Для створення контрольного синтетичного наванитаження на процесор, використовується образ stress-ng. Частиною цього застосунку є ресурс HPA (лістинг 3.9), що забезпечує масштабування кількості реплік застосунку відповідно до цільового значення завантаження CPU, визначеного у 50%. Це дозволяє протестувати роботу HPA і переконатися в коректності автоматичного збільшення або зменшення реплік залежно від навантаження на CPU.

Лістинг 3.9 – HPA ресурс app-hpa

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: app-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: app-hpa
  minReplicas: 1
  maxReplicas: 9
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 50
```

Для перевірки масштабування за кількістю повідомлень у черзі Amazon SQS, що реалізовано через KEDA, виористовується застосунок app-sqs. Він

містить Python застосунок, який отримує повдомлення із черги SQS, та симуляє процес їх обробки. Для автоматичного масштабування створюється ресурс ScaledObject (лістинг 3.10), що налаштований на масштабування відповідно до довжини черги SQS. Завдяки інтеграції з AWS Pod Identity застосунок отримує доступ до AWS-ресурсів без передачі явних ключів доступу. Він дозволяє протестувати механізм масштабування KEDA залежно від кількості об’єктів у черзі повідомлень.

Лістинг 3.10 – ScaledObject ресурс app-sqs

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: app-sqs
spec:
  scaleTargetRef:
    name: app-sqs
  pollingInterval: 10
  cooldownPeriod: 30
  minReplicaCount: 0
  maxReplicaCount: 20
  triggers:
    - type: aws-sqs-queue
      authenticationRef:
        name: app-sqs
      metadata:
        queueURLFromEnv: SQS_QUEUE_URL
        queueLength: "5"
        awsRegion: eu-west-1
        identityOwner: pod
```

Для перевірки масштабування за кількістю HTTP-запитів, які надходять через Application Load Balancer (ALB), було створено застосунок app-alb. Використовується стандартний образ echoserver, який повертає HTTP-відповіді. Для масштабування застосовується KEDA з інтеграцією CloudWatch, яка аналізує метрику кількості запитів ALB (лістинг 3.11). Доступ до застосунку через Інтернет реалізовується за допомогою ресурсу Ingress з відповідними анотаціями, які вказують контролеру AWS Load Balancer Controller створити публічний ALB з маршрутизацією трафіку до застосунку.

Лістинг 3.11 – Ingress та ScaledObject ресурси app-alb

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: app-alb
  annotations:
    alb.ingress.kubernetes.io/load-balancer-name: eks-dev-app
    alb.ingress.kubernetes.io/group.name: app
    alb.ingress.kubernetes.io/scheme: internet-facing
    alb.ingress.kubernetes.io/target-type: ip
    alb.ingress.kubernetes.io/ssl-redirect: "443"
    alb.ingress.kubernetes.io/listen-ports: '[{"HTTP": 80},
{"HTTPS": 443}]'
spec:
  ingressClassName: alb
  rules:
    - http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: app-alb
                port:
                  number: 80
---
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: app-alb
spec:
  scaleTargetRef:
    name: app-alb
  pollingInterval: 10
  cooldownPeriod: 30
  minReplicaCount: 0
  maxReplicaCount: 20
  triggers:
    - type: aws-cloudwatch
      authenticationRef:
        name: app-alb
      metadata:
        namespace: AWS/ApplicationELB
        expression: >
          SELECT SUM(RequestCount)
          FROM SCHEMA("AWS/ApplicationELB", LoadBalancer)
          WHERE LoadBalancer = 'app/eks-dev-app/b2e66c64982c55a'
        targetMetricValue: "60"
        minMetricValue: "1"
        metricCollectionTime: "300"
        metricStat: "Sum"
```

```
metricStatPeriod: "60"
metricUnit: "Count"
awsRegion: "eu-west-1"
```

4 ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ РЕАЛІЗОВАНИХ МЕТОДІВ МАСШТАБУВАННЯ

Для перевірки ефективності реалізованих методів масштабування було розроблено й реалізовано експериментальні сценарії, які охоплюють декілька аспекти функціонування розподілених систем. Кожен із них спрямований на оцінку окремих компонентів масштабування, зокрема її продуктивності, та здатності до масштабування, надійності та стійкості до відмов.

4.1 Сценарій 1. Порівняння ефективності Cluster Autoscaler та Karpenter

Метою цього експерименту було дослідити швидкодію двох механізмів масштабування нод у кластері Kubernetes, Cluster Autoscaler та Karpenter, у сценарії різкого збільшення кількості подів. У якості тестового застосунку використовувався app-nginx, який має визначені ресурси CPU та пам’яті. Початково кількість реплік Deployment-об’єкта було змінено з 0 до 1, що призвело до появи пода в стані Pending через відсутність доступних нод. Це ініціювало реакцію Cluster Autoscaler, який розпочав процес масштабування, створивши новий EC2-інстанс через Auto Scaling Group. Час до переходу пода у стан Running складав в середньому 72 секунди — цей інтервал включає ініціалізацію інстанса, реєстрацію ноди в кластері та запуск пода (рисунок 4.1).

Наступним кроком була зміна кількості реплік з 1 до 2. Наступний под запустився майже без затримки (2 секунди), оскільки вже існувала нода з вільними ресурсами.

У наступному кроці кількість реплік було збільшено до 10, що знову спричинило створення нових нод через Cluster Autoscaler. Час запуску нових подів знову склав близько 70 секунд, що свідчить про стабільний результат швидкодії контролера.

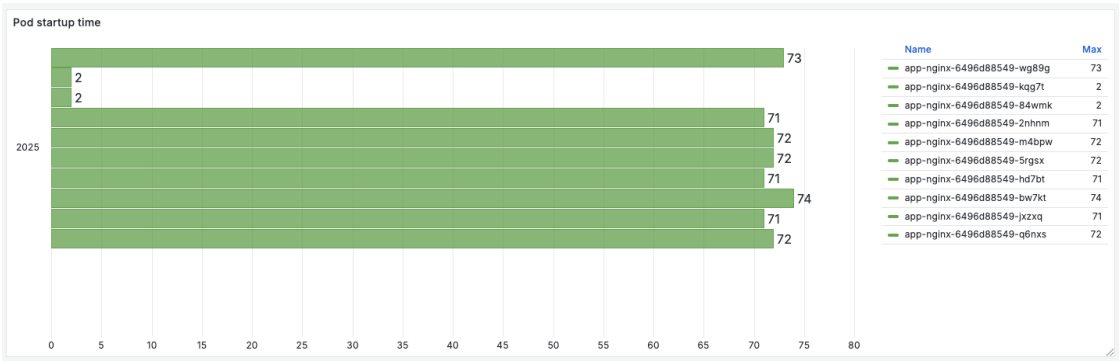

Рисунок 4.1– Час запуску под з Cluster autoscaler

Аналогічний сценарій було відтворений із використанням Karpenter. Після встановлення кількості реплік app-nginx до 1, було зафіксовано час запуску пода — 54 секунди. При збільшенні кількості реплік до 2, наступний под запустився негайно, подібно до попереднього експерименту. Масштабування до 10 реплік також відбулося за середній час – 53 секунди (рисунок 4.2).

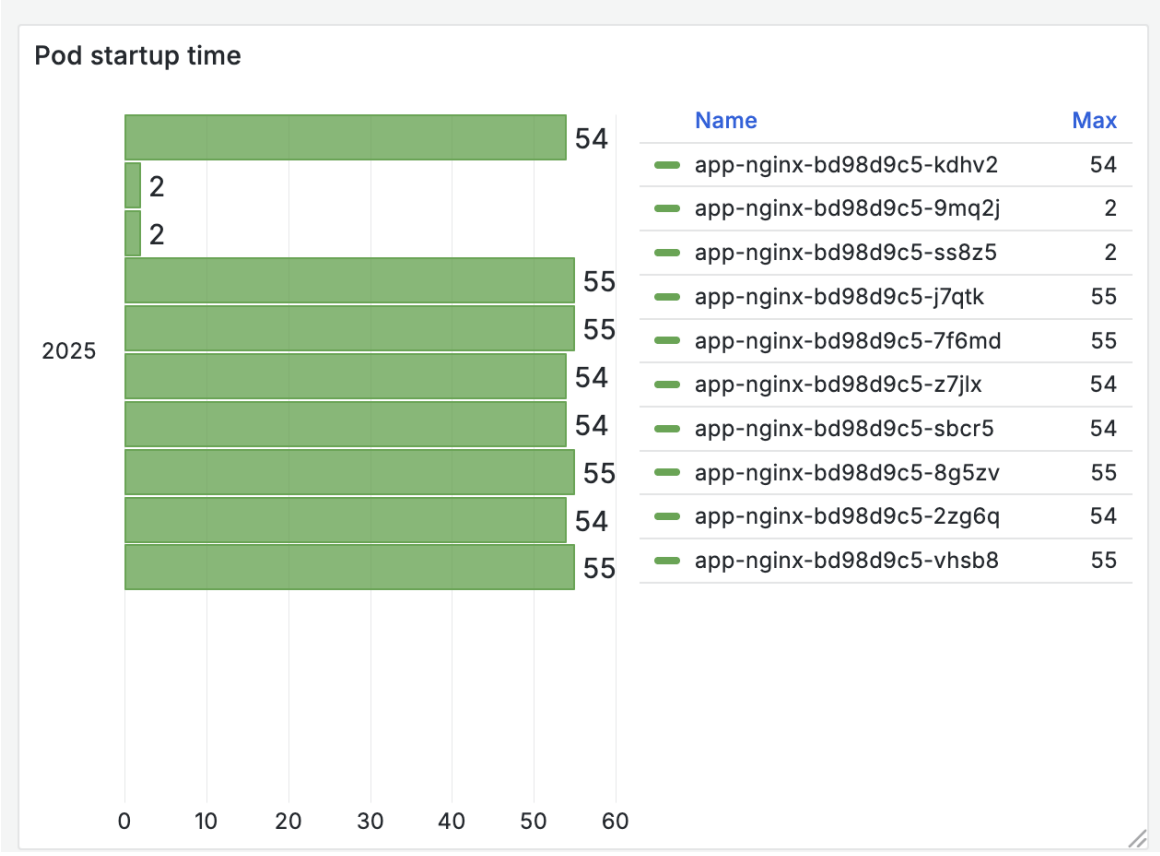

Рисунок 4.2– Час запуску под з Karpenter

Результати експерименту підтверджують, що Karpenter реагує на зміну навантаження швидше на 30%, ніж Cluster Autoscaler, і краще підходить для сценаріїв динамічного масштабування. Завдяки прямій інтеграції з EC2 та відсутності проміжних компонентів, Karenter демонструє менший час реакції і забезпечує гнучкіше масштабування кластера.

4.2 Сценарій 2. Масштабування за CPU навантаженням

У межах цього сценарію було протестовано механізм горизонтального масштабування подів за рівнем навантаження на процесор. Для цього застосовувався тестовий застосунок app-hpa, який побудовано на основі образу stress-ng, що генерує контрольоване синтетичне навантаження на CPU.

Масштабування здійснювалося за допомогою ресурсу Horizontal Pod Autoscaler (HPA), налаштованого на цільове середнє завантаження CPU у межах 50%. У початковому стані був розгорнутий один под, який створював навантаження за допомогою параметрів --cpu 1 та --cpu-load 60, що спричиняло перевищення цільового значення. В результаті HPA почав поступово масштабувати застосунок, збільшуючи кількість реплік для компенсації навантаження. У процесі експерименту спостерігалося, що поди масштабуються відповідно до зростання навантаження, а після завершення синтетичного тестування — поступово зменшуються до початкового значення (рисунок 4.3 та рисунок 4.4). Це підтверджує коректну роботу HPA.

Варто зазначити, що KEDA при масштабуванні за CPU працює як надбудова над HPA, тому не має сенсу проводити їх порівняння у цьому контексті. У випадку метрик CPU саме HPA здійснює масштабування подів, а KEDA — лише передає відповідні дані, якщо джерело метрик — зовнішній провайдер. Отже, логіка масштабування у цьому випадку є ідентичною.

Експеримент засвідчив надійність механізму HPA при роботі із CPU-навантаженням, що є базовим і широко застосовуваним методом

горизонтального масштабування у Kubernetes.

Рисунок 4.3– Графіки масштабування контролеру HPA

```
❯ kubectl get hpa --watch
NAME      REFERENCE              TARGETS              MINPODS   MAXPODS   REPLICAS   AGE
app-hpa   Deployment/app-hpa     cpu: <unknown>/50%   1         9         0          8s
app-hpa   Deployment/app-hpa     cpu: <unknown>/50%   1         9         1          15s
app-hpa   Deployment/app-hpa     cpu: 67%/50%         1         9         1          30s
app-hpa   Deployment/app-hpa     cpu: 67%/50%         1         9         2          45s
app-hpa   Deployment/app-hpa     cpu: 68%/50%         1         9         2          60s
app-hpa   Deployment/app-hpa     cpu: 66%/50%         1         9         3          75s
app-hpa   Deployment/app-hpa     cpu: 66%/50%         1         9         3          90s
app-hpa   Deployment/app-hpa     cpu: 65%/50%         1         9         3          105s
app-hpa   Deployment/app-hpa     cpu: 65%/50%         1         9         4          2m1s
app-hpa   Deployment/app-hpa     cpu: 65%/50%         1         9         4          2m16s
app-hpa   Deployment/app-hpa     cpu: 63%/50%         1         9         6          2m31s
app-hpa   Deployment/app-hpa     cpu: 63%/50%         1         9         6          2m46s
app-hpa   Deployment/app-hpa     cpu: 63%/50%         1         9         8          3m1s
app-hpa   Deployment/app-hpa     cpu: 64%/50%         1         9         8          3m16s
app-hpa   Deployment/app-hpa     cpu: 64%/50%         1         9         8          3m46s
app-hpa   Deployment/app-hpa     cpu: 64%/50%         1         9         8          4m1s
app-hpa   Deployment/app-hpa     cpu: 64%/50%         1         9         8          4m16s
app-hpa   Deployment/app-hpa     cpu: 64%/50%         1         9         9          4m31s
app-hpa   Deployment/app-hpa     cpu: 64%/50%         1         9         9          5m16s
```

Рисунок 4.4– Процес масштабування з терміналу оператора

4.3 Сценарій 3. Масштабування по довжині SQS черги

Цей експеримент мав на меті перевірити можливості KEDA щодо масштабування подів Kubernetes-додатка за зовнішніми метриками, зокрема

за довжиною черги Amazon SQS. Для тестування використовувався застосунок app-sqs, який взаємодіє з чергою повідомлень у сервісі SQS та має визначені ресурси CPU та пам’яті.

У середовищі AWS було попередньо створено чергу типу FIFO, до якої надсилаються повідомлення із зовнішнього продюсера. Масштабування подів реалізовано через ресурс ScaledObject, у конфігурації якого вказано тип тригера aws-sqs-queue, порогове значення довжини черги (5 повідомлень) та мінімальну і максимальну кількість реплік (від 0 до 20).

Завдяки інтеграції з AWS Pod Identity, застосунок отримує доступ до черги на рівні інфраструктури без потреби керування обліковими даними, що спрощує безпеку та адміністрування.

У ході експерименту було штучно згенеровано навантаження шляхом надсилання великої кількості повідомлень до черги (рисунок 4.5 та рисунок 4.6). Після перевищення порогового значення довжини черги було зафіксовано автоматичне масштабування подів — кількість реплік поступово збільшувалась.

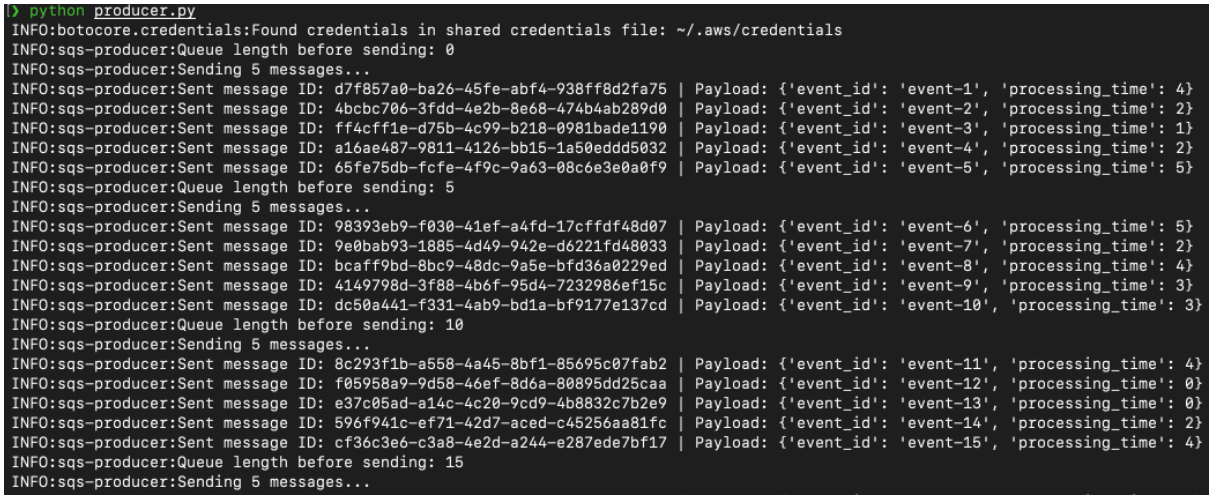

Рисунок 4.5– Запуск скрипту для генерування синтетичного навантаження

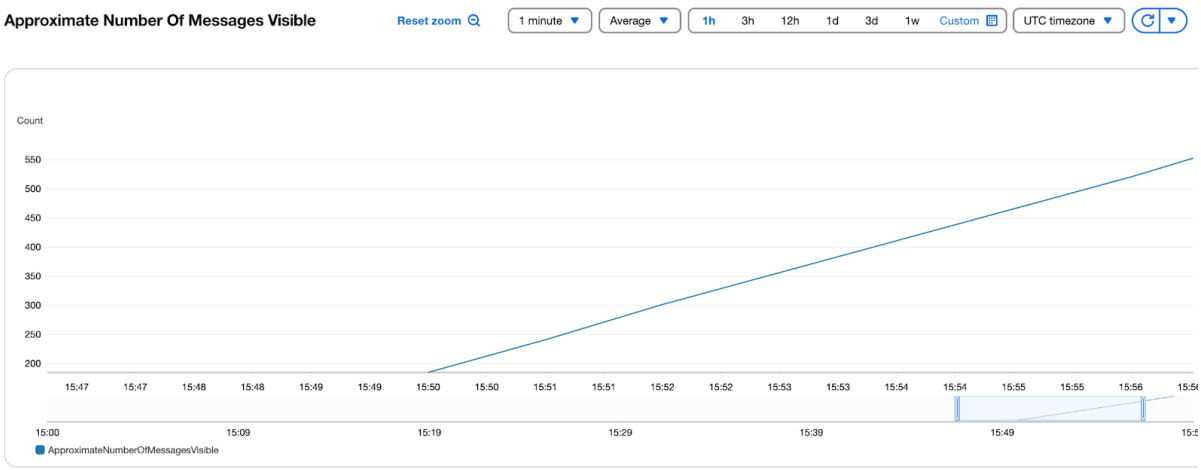

Рисунок 4.6– Графік довжини черги SQS

У рамках цього експерименту також було продемонстровано здатність KEDA масштабувати поди з нуля (cold start). Початково, при відсутності навантаження, кількість реплік застосунку становила 0, і жодні ресурси не споживалися. Після надходження повідомлень до черги, KEDA контролер виявив зміну зовнішньої метрики та ініціював створення нових подів (рисунок 4.7). Такий механізм є важливим для подієво-орієнтованих застосунків, оскільки забезпечує оптимальне використання ресурсів у періоди відсутності навантаження та швидкий запуск у відповідь на події.

Таким чисном, у сценарії було продемонстровано ефективну роботу KEDA при масштабуванні за зовнішньою подієвою метрикою — довжиною черги Amazon SQS. Цей підхід дозволяє гнучко реагувати на зміну навантаження в системах з асинхронною обробкою подій, а також зменшувати витрати на інфраструктуру завдяки можливості масштабування з нуля.

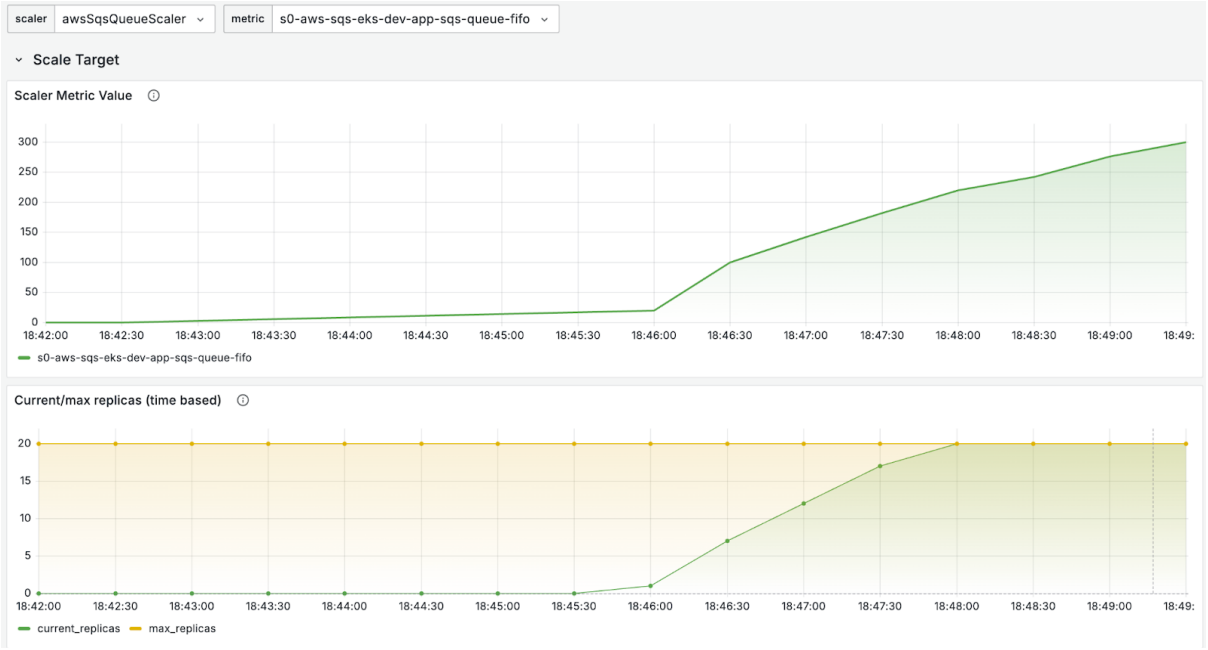

Рисунок 4.7– Графіки масштабування контролеру KEDA

4.4 Сценарій 4. Масштабування по кількості запитів ALB

У цьому експерименті було протестовано можливість масштабування подів на основі зовнішньої метрики — кількості HTTP-запитів, що надходять через Application Load Balancer (ALB). Для цього використовувався застосунок app-alb, що розгортається разом із відповідним ресурсом Ingress та інтеграцією з ALB. В якості тестового застосунку використовувався app-alb, який побудований на образі echoserver і виступає простим HTTP-сервером для приймання вхідних запитів. Застосунок було опубліковано через об'єкт Ingress, який має анотації для інтеграції з ALB-контролером у Kubernetes. ALB маршрутизує трафік до подів, а кількість запитів агрегується за допомогою метрики RequestCount, доступної у CloudWatch.

Для масштабування застосовується ресурс ScaledObject, у якому вказано тип тригера aws-cloudwatch, цільове значення метрики (60 запитів), часовий період агрегації. У ході експерименту було змодельовано зростання кількості запитів до ALB за допомогою скрипту навантаження (рисунок 4.8). Після перевищення заданого значення метрики, KEDA контролер автоматично ініціював масштабування — кількість подів зросла, що

забезпечило можливість ефективної обробки запитів (рисунок 4.9). Після зниження навантаження на ALB, контролер поступово зменшив кількість реплік до мінімального значення. Таким чином, було продемонстровано гнучкість і ефективність масштабування подів у залежності від поточного навантаження зовнішнього балансувальника.

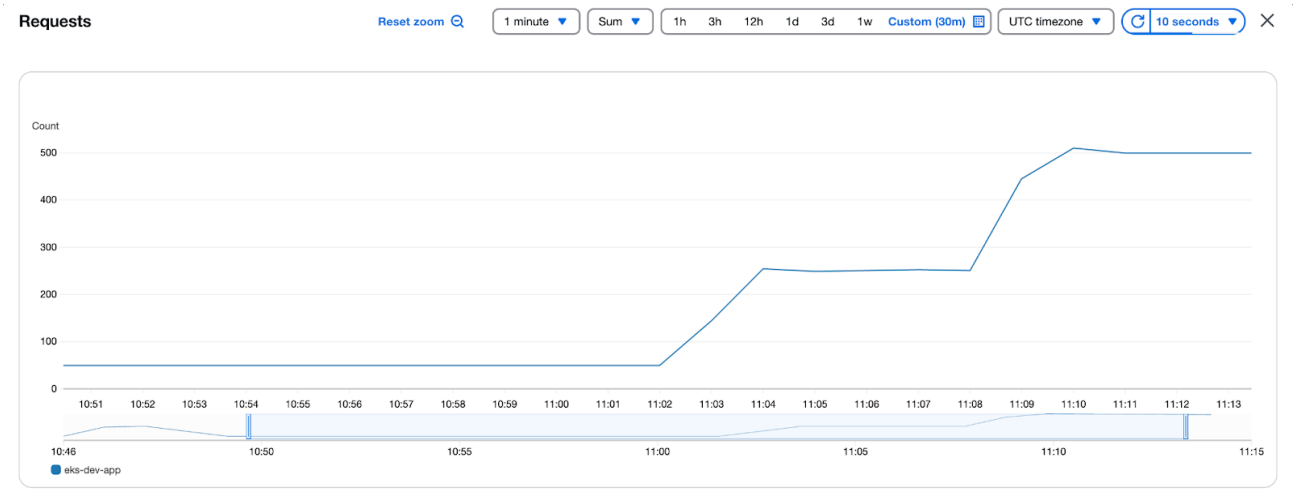

Рисунок 4.8– Графік кількості запитів до ALB

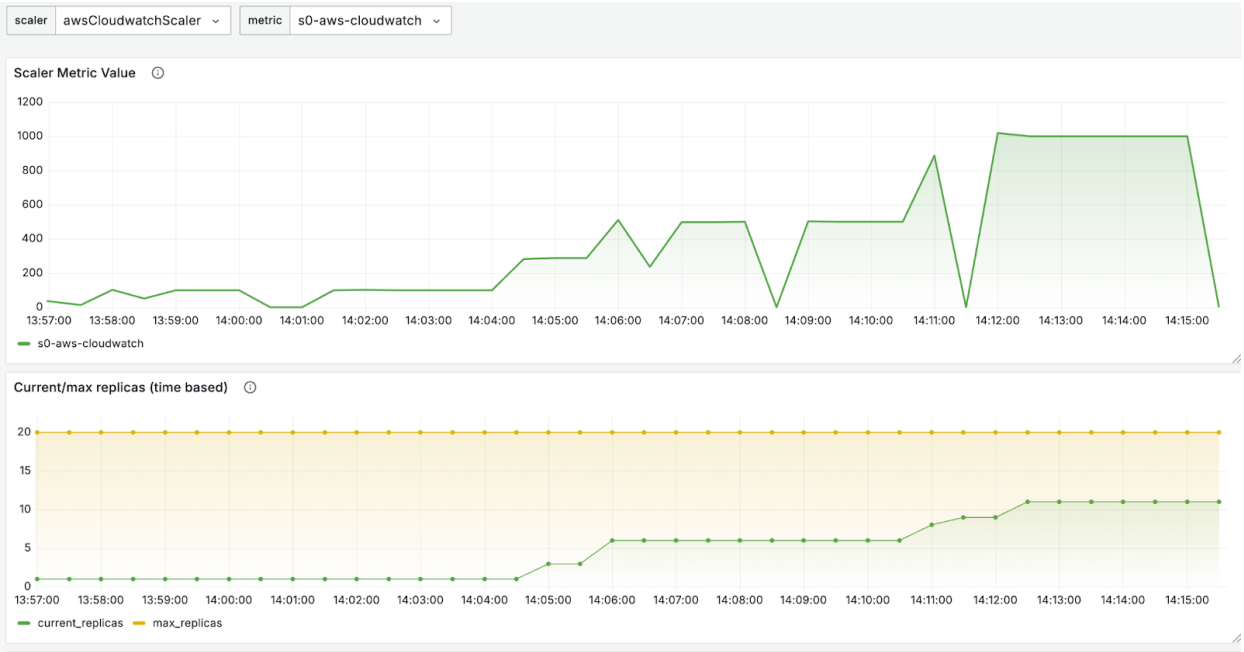

Рисунок 4.9– Графіки масштабування контролеру KEDA

4.5 Оцінка результатів тестування та аналіз реалізованих методів

У результаті проведених експериментів було підтверджено ефективність реалізованих методів автоматичного масштабування Kubernetes-кластеру в хмарному середовищі AWS. Кожен із протестованих сценаріїв дозволив оцінити роботу окремих компонентів системи масштабування — як на рівні вузлів, так і на рівні подів, а також у взаємодії з зовнішніми джерелами навантаження.

Порівняльний аналіз Cluster Autoscaler та Karpenter показав, що останній забезпечує значно швидшу реакцію на навантаження (в середньому на 30%), завдяки відсутності залежності від Auto Scaling Group і прямій інтеграції з EC2 API. Це робить Karpenter більш придатним для сценаріїв із динамічним та непередбачуваним навантаженням. Крім того, завдяки гнучкому механізму підбору інстансів за допомогою NodePool та EC2NodeClass, Karpenter дає змогу точно контролювати конфігурацію віртуальних машин і оптимізувати витрати, зокрема використовуючи Spot-інстанси.

Тестування HPA на прикладі застосунку з CPU-навантаженням підтвердило його здатність забезпечувати стабільну підтримку продуктивності при зростанні навантаження на обчислювальні ресурси. Хоча цей механізм є базовим, його надійність і простота налаштування залишаються ключовими перевагами.

Особливо важливими є результати експериментів із KEDA, яка продемонструвала гнучке масштабування за зовнішніми подієвими метриками, такими як довжина черги Amazon SQS та кількість запитів до ALB. У випадку з SQS було підтверджено здатність масштабування з нуля (cold start), що дозволяє зменшити витрати на інфраструктуру в моменти неактивності системи. Аналогічно, масштабування за CloudWatch-метриками ALB забезпечило динамічне реагування на реальні HTTP-запити від користувачів.

ВИСНОВКИ

У ході дослідження було розглянуто сучасні методи та інструменти автоматичного масштабування кластерів Kubernetes у хмарному середовищі Amazon Web Services. Проведений теоретичний та архітектурний аналіз дозволив сформувати цілісне уявлення про принципи масштабування з акцентом на ефективність, продуктивність і вартісну доцільність впроваджених рішень.

Одним із ключових висновків є те, що класичні механізми масштабування, як-от Horizontal Pod Autoscaler (HPA) та Cluster Autoscaler, залишаються основою динамічного реагування на зміну навантаження, але мають низку обмежень, пов’язаних із затримками реакції, складністю конфігурації та залежністю від метрик ресурсів. У той же час поява інструментів нового покоління, зокрема KEDA та Karpenter, відкриває нові можливості для масштабування в умовах подієво-орієнтованих архітектур і високодинамічного середовища. KEDA дозволяє масштабувати поди, інтегруючись з чергами повідомлень та іншими зовнішніми джерелами, тоді як Karpenter пропонує новий підхід до масштабування вузлів без прив’язки до груп автоматичного масштабування, динамічно добираючи інстанси EC2 згідно з вимогами подів і поточним контекстом хмарної інфраструктури.

Хмарна платформа AWS, зі своїм широким спектром сервісів створює оптимальне середовище для реалізації масштабованої інфраструктури Kubernetes. Забезпечуючи гнучкість у виборі типів екземплярів, підтримку як керованих, так і самостійно налаштованих вузлів, а також вбудовані механізми моніторингу й безпеки, AWS дозволяє не лише масштабувати кластери відповідно до навантаження, а й робити це ефективно, мінімізуючи витрати на ресурси без шкоди для продуктивності.

Узагальнюючи отримані результати, можна стверджувати, що оптимальне масштабування Kubernetes у хмарі потребує комплексного

підходу, який поєднує класичні й сучасні механізми, використовує подієву модель масштабування та активно залучає хмарні сервіси для глибшої інтеграції та автоматизації. Таким чином, архітектура на базі Amazon EKS із використанням KEDA, Karpenter та інших контрольованих сервісів AWS є однією з найбільш перспективних моделей для побудови адаптивної, продуктивної та економічно ефективної інфраструктури сучасних розподілених додатків.

Результати експериментів засвідчили переваги Karpenter у швидкості реагування та гнучкості масштабування, ефективність HPA для класичних сценаріїв за ресурсними метриками, а також широкі можливості KEDA щодо масштабування за зовнішніми подієвими метриками й підтримки масштабування з нуля. Наявність моніторингу на базі Prometheus і Grafana дозволила детально проаналізувати роботу масштабувальників і забезпечити оперативне виявлення аномалій.

У підсумку, результати демонструють, що запропонована модель, побудована на поєднанні Terraform, Helm, Cluster Autoscaler, Karpenter, HPA та KEDA, забезпечує високий рівень автоматизації, адаптивності, масштабованості та економічної ефективності при розгортанні Kubernetes-кластерів в AWS. Такий підхід дозволяє не лише динамічно реагувати на зміну навантаження, а й точно контролювати витрати та оптимізувати використання ресурсів у хмарному середовищі.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Rossi F., Cardellini V., Lo Presti F. Hierarchical Scaling of Microservices in Kubernetes // Proceedings of the IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS). – IEEE, 2020. – P. 28–37.

2. Lewis J. Microservices: A Definition of this New Architectural Term [Електронний ресурс]. – Режим доступу : https://martinfowler.com/articles/microservices.html – Дата звернення: 15.03.2025 р. – Загол. з екрану.

3. Newman, S. Building Microservices: designing fine-grained systems / Sam Newman. – Beijing ; Sebastopol : O'Reilly Media, 2015. – 280 с.

4. Cluster Architecture [Електронний ресурс]. – Режим доступу : https://kubernetes.io/docs/concepts/architecture/ – Дата звернення: 15.03.2025 р. – Загол. з екрану.

5. Amazon EKS Service Level Agreement [Електронний ресурс]. – Режим доступу : https://aws.amazon.com/eks/sla/ – Дата звернення: 15.03.2025 р. – Загол. з екрану.

6. Amazon EKS User guide [Електронний ресурс]. – Режим доступу : https://docs.aws.amazon.com/pdfs/eks/latest/userguide/eks-ug.pdf – Дата звернення: 15.03.2025 р. – Загол. з екрану.

7. Overview of Amazon Web Services – AWS Whitepaper [Електронний ресурс]. – Режим доступу : https://docs.aws.amazon.com/pdfs/whitepapers/latest/aws-overview/aws-overview.pdf – Дата звернення: 15.03.2025 р. – Загол. з екрану.

8. Reactive systems on AWS [Електронний ресурс]. – Режим доступу : https://docs.aws.amazon.com/pdfs/whitepapers/latest/reactive-systems-on-aws/reactive-systems-on-aws.pdf– Дата звернення: 15.03.2025 р. – Загол. з екрану.

9. Can We Solve Serverless Cold Starts? [Електронний ресурс]. – Режим доступу: https://dashbird.io/blog/can-we-solve-serverless-cold-starts/ – Дата звернення: 15.03.2025 р. – Загол. з екрану.

10. Помелуйко Д. А., Єрьоміна Н. С., Петров С. В., Іщенко Н. В., Волощук О. Б. Методи масштабування kubernetes кластеру в хмарному середовищі. Системи управління, навігації та зв'язку. 2025. No2, с. 176 – 180.

11. KEDA: Kubernetes-based Event Driven Autoscaling [Електронний ресурс]. – Режим доступу : https://keda.sh/ – Дата звернення: 15.03.2025 р. – Загол. з екрану.

12. AWS EC2 Spot Instances Documentation [Електронний ресурс]. – Режим доступу : https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-spot-instances.html – Дата звернення: 15.03.2025 р. – Загол. з екрану.

13. Kubernetes Vertical Pod Autoscaler [Електронний ресурс]. – Режим доступу : https://github.com/kubernetes/autoscaler/tree/master/vertical-podautoscaler – Дата звернення: 15.03.2025 р. – Загол. з екрану.

14. Kubernetes Cluster Autoscaler [Електронний ресурс]. – Режим доступу : https://www.perfectscale.io/blog/kubernetes-cluster-autoscaler – Дата звернення: 15.03.2025 р. – Загол. з екрану.

15. Karpenter: An Open-Source Node Provisioner [Електронний ресурс]. – Режим доступу : https://karpenter.sh/ – Дата звернення: 15.03.2025 р. – Загол. з екрану.