Одеський національний університет імені І. І. Мечникова
Факультет математики, фізики та інформаційних технологій
Кафедра оптимального керування та економічної кібернетики

Кваліфікаційна робота

на здобуття ступеня вищої освіти «бакалавр»

«Оптимізація використання ресурсів у Kubernetes кластерах»

«Optimizing resource usage in Kubernetes clusters»

Виконав: здобувач денної форми навчання
спеціальності 113 Прикладна математика
Освітня програма «Прикладна математика»
Рибальченко Олексій Геннадійович

Керівник: ст. викл. Платонов В.В. _____
Рецензент: Канд. техн. наук, доц. Мороз В.В.

Рекомендовано до захисту:

Протокол засідання кафедри

№ ___ від _____ 2025 р.

Завідувач кафедри

_____ _____

Захищено на засіданні ЕК № _____

Протокол № ___ від _____ 2025 р.

Оцінка _____ / _____ / _____

Голова ЕК

_____ _____

Одеса — 2025 р.

ЗМІСТ

ВСТУП

Сучасні інформаційні системи дедалі частіше розгортаються у хмарному середовищі з використанням мікросервісної архітектури та контейнеризації. Одним із найпоширеніших інструментів управління контейнеризованими додатками є Kubernetes — система оркестрації, що забезпечує автоматичне розгортання, масштабування та управління життєвим циклом контейнерів.

Попри гнучкість і масштабованість, Kubernetes-кластери часто стикаються з проблемою неефективного використання ресурсів, зокрема обчислювальних потужностей (CPU), оперативної пам'яті (RAM) та тимчасового сховища (ephemeral storage). Типові причини включають надмірне резервування ресурсів, неправильне конфігурування `requests`/`limits`, невдалу політику розміщення pod'ів та відсутність аналізу реального навантаження.

В умовах хмарної інфраструктури така неефективність безпосередньо призводить до перевитрат бюджету. Наприклад, кожен незавантажений EC2-інстанс у AWS, що працює без подів або з подами з надмірними запитами, генерує зайві фінансові витрати. Крім того, перевантажені pod'и можуть втрачати стабільність через `OOMKilled` або `throttling`, що критично для бізнес-додатків з високими вимогами до доступності.

Виходячи з цього, оптимізація використання ресурсів у Kubernetes-кластерах є актуальною та важливою задачею як з технічної, так і з економічної точки зору. Вона дозволяє не лише знизити витрати на інфраструктуру, але й забезпечити стабільну, передбачувану роботу додатків, зменшити час масштабування та спростити адміністрування кластеру.

Мета цієї дипломної роботи — дослідити існуючі підходи до аналізу використання ресурсів у Kubernetes, розробити власний підхід до валідації конфігурації та розміщення pod'ів, а також реалізувати програмне рішення, яке дозволить:

- автоматично виявляти pod'и з неефективним використанням ресурсів;
- формувати рекомендації щодо змін конфігурацій `requests`/`limits`;
- пропонувати оптимальні типи хмарних інстансів для розгортання;
- оцінювати потенційну економію коштів після оптимізації.

Робота складається з трьох основних розділів:

- У першому розділі розглянуто теоретичні основи Kubernetes та моделі управління ресурсами.
- У другому розділі проведено формалізацію задачі оптимізації, запропоновано аналітичний підхід до аналізу використання ресурсів.
- У третьому розділі реалізовано програмне рішення та проаналізовано результати його застосування на основі логів реального кластеру.

Актуальність та практична цінність дослідження полягають у поєднанні математичного аналізу ефективності конфігурації кластеру з реалізацією інструмента для її валідації в умовах хмарного середовища.

РОЗДІЛ 1

ТЕОРЕТИЧНІ ОСНОВИ KUBERNETES ТА УПРАВЛІННЯ РЕСУРСАМИ

У цьому розділі розглядаються базові поняття та механізми Kubernetes, що стосуються архітектури системи та керування обчислювальними ресурсами. Наведено огляд компонентів Kubernetes-кластера та їхньої взаємодії, визначено ключові терміни (Pod, вузол, контейнер, простір імен), описано типи ресурсів, з якими оперує Kubernetes (процесорний час, пам'ять, ефемерне сховище), а також роз'яснено принципи налаштування ресурсних запитів і обмежень. Окрему увагу приділено політикам планування подів на вузлах – таким як афінність/антиафінність та механізми Taints і Tolerations – та алгоритмам розміщення і масштабування навантаження в кластері.

1.1 Архітектура Kubernetes

Kubernetes — це система оркестрації контейнерів з відкритим кодом, призначена для автоматизації розгортання, масштабування та управління контейнеризованими додатками. Його архітектура побудована за принципом клієнт-серверної моделі з чітким розділенням обов'язків між компонентами керування та виконання.

1.1.1 Основні компоненти Kubernetes-кластера

Архітектура Kubernetes включає два основні рівні:

- **Control Plane (керуючий рівень)** — відповідає за прийняття рішень щодо розміщення подів, контролю їхнього стану, масштабування та реагування на події.
- **Worker Nodes (вузли виконання)** — виконують запущені контейнери та забезпечують обчислювальні ресурси.

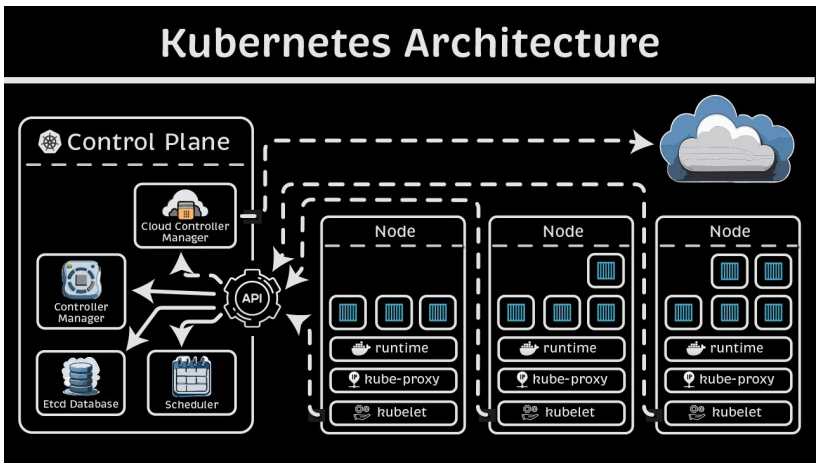

Рис. 1.1. Архітектура Kubernetes

Компоненти Control Plane:

1) **kube-apiserver** — центральний компонент, що приймає всі запити до кластера (через REST API). Є точкою входу для користувачів та інших компонентів.
2) **etcd** — розподілене сховище ключ-значення, яке зберігає всю конфігурацію кластеру та поточний стан ресурсів.
3) **kube-scheduler** — відповідає за розміщення pod'ів на відповідні вузли на основі доступних ресурсів, affinity/taints, політик QoS.
4) **kube-controller-manager** — запускає контролери, які відповідають за підтримку стану кластеру (ReplicaSet, Deployment, Namespace, Node, Job тощо).
5) **cloud-controller-manager** — взаємодіє з API хмарного провайдера (наприклад, AWS, GCP, Azure), створює балансувальники навантаження, volume тощо.

Компоненти Worker Node:

1) **kubelet** — агент, який працює на кожному вузлі, слідкує за станом pod'ів та контейнерів, взаємодіє з API-сервером.
2) **kube-proxy** — налаштовує мережеві правила, забезпечує мережеву маршрутизацію та балансування навантаження між сервісами.
3) **Container Runtime** — відповідає за запуск контейнерів. Наприклад: `containerd`, `CRI-O`, `Docker` (deprecated).

Комунікація між компонентами

Усі запити та взаємодії здійснюються через `kube-apiserver`, який є єдиною точкою входу. Зовнішні клієнти (наприклад, `kubectl`) або внутрішні контролери використовують HTTPS-запити до API.

1.1.2 Розгортання в хмарному середовищі

У хмарі (наприклад, AWS або GCP) Kubernetes часто розгортається як:

- `Managed Kubernetes` — Amazon EKS, Google GKE, Azure AKS;
- `Self-managed` — власне розгортання за допомогою `kubeadm`, `kops`, `terraform`, `eksctl`.

1.1.3 Підсумок

Архітектура Kubernetes є модульною та гнучкою, що дозволяє масштабувати систему, автоматизувати керування, забезпечувати високий рівень відмовостійкості та ефективного розподілу навантаження. Розуміння архітектури є критичним для подальшого аналізу роботи кластеру, ресурсних механізмів та принципів оптимізації, що розглядаються у наступних розділах.

1.2 Поняття Pod, Node, Namespace, Container

Для повноцінного розуміння логіки функціонування Kubernetes-кластера та реалізації механізмів управління ресурсами необхідно чітко розуміти базові об'єкти, які беруть участь у роботі системи. До таких фундаментальних сутностей належать `Pod`, `Node`, `Container` і `Namespace`.

1.2.1 Pod

Pod — це найменша одиниця розгортання в Kubernetes. Він є логічною оболонкою для одного або кількох контейнерів, які:

- працюють на одному вузлі;
- мають спільний мережевий стек (IP-адреса, порти);
- можуть мати спільні томи (volumes) для зберігання даних.

Pod створюється як атомарна одиниця — всі контейнери в ньому запускаються та зупиняються разом. У більшості випадків pod містить лише один контейнер, але у складніших сценаріях можливе використання sidecar-контейнерів (наприклад, `istio-proxy`).

Формально:

Нехай $P = \{p_1, p_2, \ldots, p_n\}$ — множина pod'ів, тоді кожен pod p_i є кортежем:

$$p_i = (\text{containers}, \text{volumes}, \text{network}, \text{metadata})$$

1.2.2 Container

Контейнери є основною одиницею виконання програм у Kubernetes. Kubernetes підтримує різні runtime-оточення, такі як `containerd`, `CRI-O`, `Docker` (не підтримується напряму з версії 1.24).

Кожен контейнер у pod'і:

- має власну файлову систему (image);
- ізольований від інших контейнерів (через cgroups і namespaces);
- виконується в межах того самого pod'а з доступом до спільного мережевого простору.

1.2.3 Node

Node — це фізичний або віртуальний сервер, який є частиною кластера та надає ресурси (CPU, RAM, дисковий простір) для розміщення pod'ів.

Кожен node запускає наступні обов'язкові компоненти:

- `kubelet` — відповідає за взаємодію з API-сервером та запуск контейнерів;
- `kube-proxy` — керує мережевими правилами;
- `Container Runtime` — запускає безпосередньо контейнери.

Node має статус (Ready / NotReady), а також може мати `taints`, які обмежують розміщення pod'ів.

Математична модель: нехай $N = \{n_1, \ldots, n_m\}$ — множина вузлів, тоді кожен node n_j характеризується:

$$n_j = (CPU_j, MEM_j, \text{taints}_j, \text{labels}_j)$$

1.2.4 Namespace

Namespace — логічний роздільник ресурсів у кластері. Дозволяє:

- ізолювати ресурси між командами, проектами або середовищами (staging, production);
- призначати окремі квоти на ресурси (ResourceQuota);
- обмежити доступ через RBAC (Role-Based Access Control).

Приклад: системні pod'и Kubernetes знаходяться в namespace `kube-system`, додатки користувача — в `default` або кастомних (`app-dev`, `xenia-stg` тощо).

1.2.5 Зв'язки між об'єктами

- Pod запускається в конкретному `Namespace`;
- Pod розміщується на одному `Node`;
- Pod складається з одного або кількох `Container`;
- Node містить багато pod'ів.

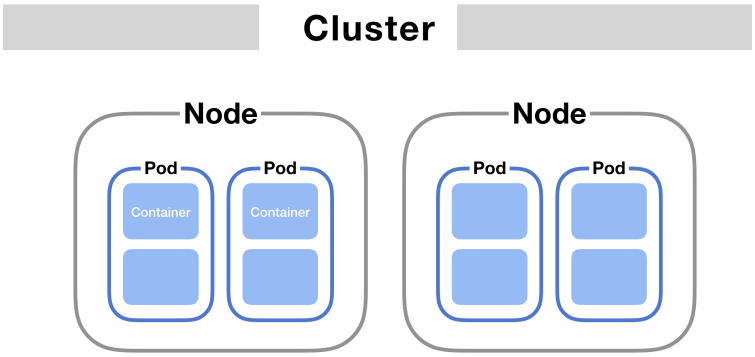

Рис. 1.2. Node-Pod

1.2.6 Висновок

Розуміння фундаментальних об'єктів Kubernetes є необхідною умовою для аналізу ефективності використання ресурсів. Усі подальші метрики, політики розміщення, масштабування та обмеження базуються на рівні pod'ів, node'ів та namespace.

1.3 Ресурси у Kubernetes: CPU, RAM, ephemeral storage

Kubernetes дозволяє визначати ресурси, необхідні для кожного контейнера та pod'а, за допомогою параметрів `requests` та `limits`. Основні ресурси, якими оперує Kubernetes, включають процесорний час (CPU), оперативну пам'ять (RAM) та тимчасове сховище (ephemeral storage).

1.3.1 CPU

Одиниці вимірювання: CPU в Kubernetes вимірюється в міліядрах (`millicores`). 1 CPU = 1000 міліядр.

Приклади:

- `500m` означає половину одного ядра;
- `1` означає одне повне ядро.

Призначення:

- `requests.cpu` — мінімальна кількість CPU, гарантована контейнеру;
- `limits.cpu` — максимальна кількість CPU, яку контейнер може використати.

Алгоритм планування: `kube-scheduler` обирає node, що має вільне місце під сумарні `requests`. Ліміт враховується вже під час виконання (через CFS у cgroups).

1.3.2 Пам'ять (RAM)

Одиниці вимірювання: `Mi`, `Gi`, `M`, `G` (Kubernetes розпізнає як двійкові, так і десяткові одиниці).

Призначення:

- `requests.memory` — обсяг пам'яті, який контейнеру гарантовано;
- `limits.memory` — максимальна межа пам'яті (якщо перевищено — контейнер буде завершено з `OOMKilled`).

Алгоритм планування: node має містити достатній обсяг пам'яті під суму `requests` усіх подів, які плануються.

1.3.3 Ephemeral Storage (тимчасове сховище)

Це дисковий простір, який використовується контейнерами для:

- `/tmp`, `/var/log`, кешу;
- запису логів (якщо немає зовнішніх томів);
- шарів образу контейнера.

Ресурсні параметри:

- `requests.ephemeral-storage`;
- `limits.ephemeral-storage`.

Особливість: хоча scheduler враховує `requests`, ліміти застосовуються лише в процесі роботи через `eviction thresholds` на node-рівні.

1.3.4 Математичне уявлення ресурсу pod'а

Кожен pod p_i можна представити як вектор ресурсного профілю:

$$\vec{r_i} = \left[cpu_i^{\text{req}},\ \ mem_i^{\text{req}},\ \ storage_i^{\text{req}} \right]$$

Загальне навантаження на node n_j:

$$\vec{R_j} = \sum_{p_i \in P_j} \vec{r_i}$$

де P_j — множина pod'ів, розміщених на n_j.

1.3.5 Принцип споживання vs. резервування

- **Резервування:** scheduler планує pod на node, виходячи з `requests`;
- **Споживання:** фактичне використання може бути нижчим або вищим за запит.

Наслідки:

- якщо споживання $>$ ліміт $\rightarrow$ обмеження/eviction;
- якщо споживання $<$ запит $\rightarrow$ неефективне використання.

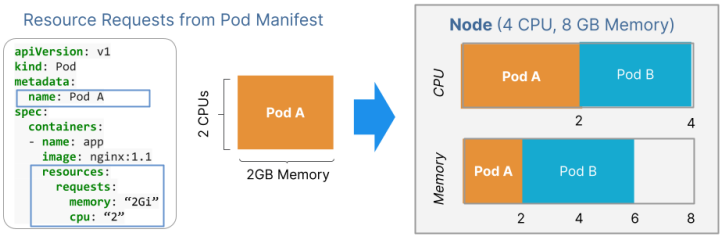

Рис. 1.3

1.3.6 Висновок

Ресурсна модель Kubernetes базується на чіткому поділі між запитами та лімітами. Вміння інтерпретувати ці параметри, аналізувати реальне

споживання та правильно конфігурувати pod'и є основою ефективного використання кластеру та запорукою стабільності додатків.

1.4 Принципи request/limit та класи якості обслуговування (QoS)

Одним з основоположних механізмів управління ресурсами в Kubernetes є параметри `requests` і `limits`. Ці значення використовуються для планування подів, контролю їх споживання ресурсів та класифікації їхньої якості обслуговування (QoS).

1.4.1 Визначення

- `requests` — це кількість ресурсів (CPU, пам'ять), яку pod або контейнер гарантовано отримає від системи.
- `limits` — це максимальна кількість ресурсів, яку pod може спожити. Якщо перевищено, може відбутися throttling (для CPU) або eviction (для пам'яті).

1.4.2 Зв'язок із планувальником

- Планувальник (`kube-scheduler`) використовує лише значення `requests` для визначення можливості розміщення pod'а на певному node.
- Значення `limits` не впливають на розміщення, але обмежують використання під час виконання.

1.4.3 Випадки конфігурацій

Нехай:

$$cpu_i^{\text{req}} \leq cpu_i^{\text{limit}}, \quad mem_i^{\text{req}} \leq mem_i^{\text{limit}}.$$

Тоді можливі комбінації:

1) **Встановлені і `requests`, і `limits` (рівні між собою)** — гарантовані ресурси, захист від throttling або eviction.
2) **Встановлені `requests`, але не `limits`** — контейнер може використовувати більше, ніж зарезервовано (небажано).
3) **Встановлені лише `limits`** — небезпечна конфігурація, бо scheduler не резервує ресурси.
4) **Не встановлено ні `requests`, ні `limits`** — контейнер не має гарантій, працює в режимі BestEffort.

1.4.4 QoS класи в Kubernetes

На основі значень `requests`/`limits` pod'и автоматично класифікуються в один з трьох QoS класів:

1) **Guaranteed**
   - Значення `requests` дорівнює `limits` для всіх контейнерів;
   - Pod має найвищий пріоритет у споживанні ресурсів;
   - Найменша ймовірність eviction при нестачі ресурсів.
2) **Burstable**
   - У pod'а встановлено `requests` і `limits`, але вони не рівні;
   - Контейнер має базову гарантію, але може тимчасово використовувати більше;
   - Середній пріоритет.
3) **BestEffort**
   - Не вказано жодного з `requests` чи `limits`;
   - Контейнер отримує ресурси за залишковим принципом;
   - Найвища ймовірність eviction.

1.4.5 Приклад класифікації pod'ів

Pod	`requests`	`limits`	**QoS клас**
pod-a	CPU = 500m, MEM = 256Mi	CPU = 500m, MEM = 256Mi	Guaranteed
pod-b	CPU = 100m, MEM = 128Mi	CPU = 500m, MEM = 512Mi	Burstable
pod-c	—	—	BestEffort

Табл. 1.1. Приклади класифікації pod'ів за QoS

1.4.6 Математичне формулювання класифікації

Нехай p — pod, c_k — його k-ий контейнер. Тоді:

- p має клас Guaranteed, якщо:

$$\forall c_k : \quad cpu_k^{\text{req}} = cpu_k^{\text{limit}} \wedge mem_k^{\text{req}} = mem_k^{\text{limit}}$$

- p має клас BestEffort, якщо:

$$\forall c_k : \quad cpu_k^{\text{req}} = cpu_k^{\text{limit}} = 0 \wedge mem_k^{\text{req}} = mem_k^{\text{limit}} = 0$$

- В усіх інших випадках: Burstable.

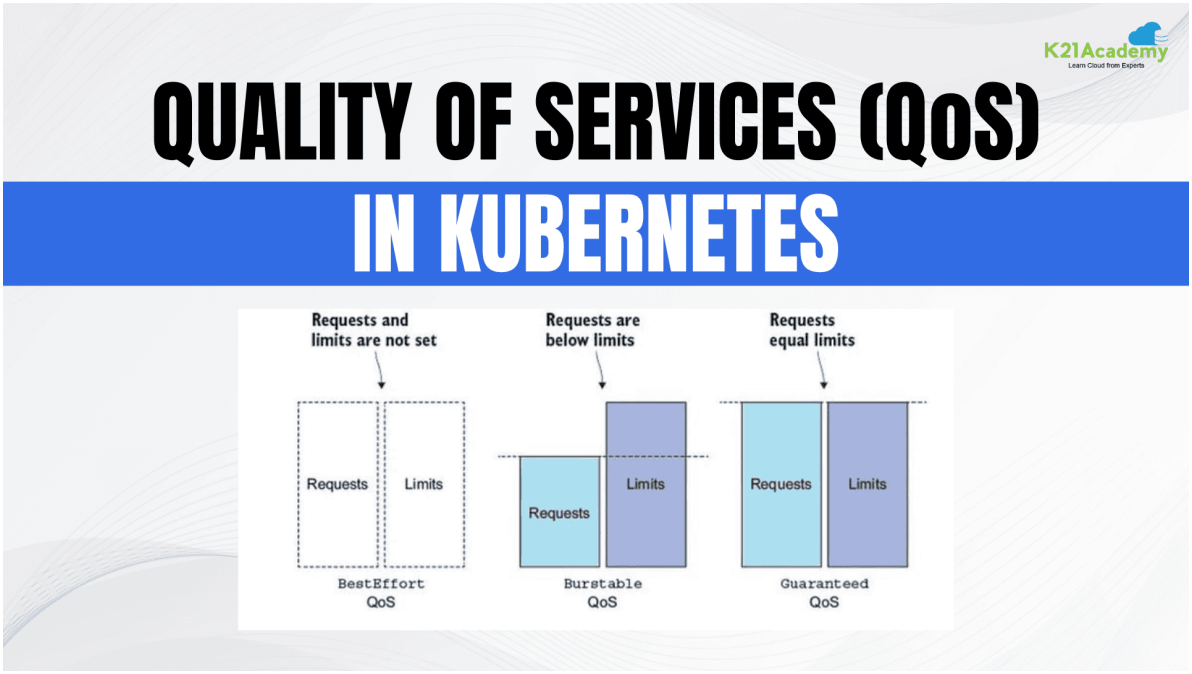

Рис. 1.4. Quality of Services (QoS)

1.4.7 Висновок

Механізм QoS-класифікації — це критично важлива особливість Kubernetes, яка дозволяє забезпечити прогнозовану поведінку додатків під час обмеження ресурсів. Грамотне використання `requests` і `limits` забезпечує як стабільність, так і ефективність використання інфраструктури.

1.5 Аффініті/антіаффініті, Taints і Tolerations

Окрім ресурсних обмежень, Kubernetes надає механізми керування розміщенням pod'ів у кластері за логікою «де pod може або не може бути запущений». До таких механізмів належать:

- **Node Affinity / Anti-Affinity**;
- **Pod Affinity / Anti-Affinity**;
- **Taints і Tolerations**.

Ці механізми відіграють ключову роль у забезпеченні fault-tolerance, зонального балансування, ізоляції середовищ і обмеженні конфігурацій.

1.5.1 Node Affinity / Anti-Affinity

Node Affinity — це обмеження на розміщення pod'ів на певних вузлах, базоване на лейблах (`labels`) нод. Задається в секції

`spec.affinity.nodeAffinity`.

- **RequiredDuringSchedulingIgnoredDuringExecution** — жорстка вимога: pod не буде запущений, якщо не знайдено відповідний вузол.
- **PreferredDuringSchedulingIgnoredDuringExecution** — бажана умова: scheduler намагатиметься дотриматися, але не обов'язково.

Приклад:

```
requiredDuringSchedulingIgnoredDuringExecution:
  nodeSelectorTerms:
  - matchExpressions:
    - key: zone
      operator: In
      values:
      - us-east-1a
```

Anti-affinity — це правило, яке забороняє розміщення pod'ів на певних вузлах.

1.5.2 Pod Affinity / Anti-Affinity

Ці правила описують, з якими іншими pod'ами має (або не має) бути розміщений поточний pod. Задаються в секції `spec.affinity.podAffinity`.

Варіанти використання:

- Розміщення подів одного сервісу разом (наприклад, для кешування);
- Рознесення подів одного типу на різні вузли (fault-tolerance).

Приклад: рознесення подів з однаковим label `app=db`.

```
podAntiAffinity:
  requiredDuringSchedulingIgnoredDuringExecution:
```

```
  - labelSelector:
      matchExpressions:
      - key: app
        operator: In
        values:
        - db
    topologyKey: kubernetes.io/hostname
```

topologyKey визначає рівень розмежування — наприклад, вузол або зона доступності.

1.5.3 Taints і Tolerations

Taints застосовуються до node'ів, щоб заборонити запуск pod'ів, які не мають відповідних `tolerations`. Це — інверсія звичайної логіки affinity: заборона, яку можна «пробачити».

Синтаксис Taint:

```
key=value:effect
```

де effect $\in$ {`NoSchedule`, `PreferNoSchedule`, `NoExecute`}.

Наприклад:

```
kubectl taint nodes node1 env=prod:NoSchedule
```

Цей taint заборонить запуск pod'ів, які не мають `toleration`:

```
tolerations:
- key: "env"
  operator: "Equal"
  value: "prod"
  effect: "NoSchedule"
```

1.5.4 Математичне формулювання

Нехай L_j — множина label'ів вузла n_j, а A_p — умови affinity pod'а p.

Тоді:

$$p \to n_j \iff A_p \subseteq L_j$$

Для podAntiAffinity:

$$p \not\to n_j \iff \exists p' \in P_j : \text{labels}(p') \in \text{забороненому списку}$$

Для taints/tolerations:

$$n_j \text{ має taint } t \Rightarrow p \text{ повинен мати toleration } t$$

1.5.5 Висновок

Механізми affinity, anti-affinity та taints/tolerations дозволяють більш точно контролювати топологію кластеру, підвищити fault-tolerance, забезпечити ізоляцію середовищ і ефективне використання ресурсів. Вони є критично важливими при побудові складних, багатозональних або безпечних інфраструктур.

1.6 Практика розміщення подів: scheduler, bin-packing, autoscaling

У Kubernetes процес розміщення pod'ів на вузлах кластера здійснюється спеціальним компонентом — **kube-scheduler**. Він приймає рішення на основі множини правил, політик і метрик. У цьому підпункті розглядаються принципи його роботи, проблема bin-packing та механізми автоскейлінгу.

1.6.1 Планування pod'ів: як працює kube-scheduler

Процес планування складається з двох основних фаз:

1) **Filtering (відбір кандидатів):** виключаються вузли, які не задовольняють вимог pod'а:
   - недостатньо ресурсів для `requests`;
   - невідповідність `nodeSelector`, `nodeAffinity`, `taints`.
2) **Scoring (оцінювання):** кожен вузол отримує оцінку на основі:
   - поточного навантаження (LeastRequestedPriority);
   - щільності (PodTopologySpread);
   - географії (zone-aware scheduling).

   Pod розміщується на вузлі з найвищою сумарною оцінкою.

1.6.2 Задача bin-packing: теоретичне підґрунтя

Розміщення pod'ів на вузлах можна подати як варіацію задачі пакування (bin-packing): кожен pod — об'єкт з об'ємом CPU, пам'яті, storage, а кожен node — контейнер (вмістилище) з фіксованими межами.

Нехай:

$$\vec{r}_i = \begin{bmatrix} cpu_i^{\text{req}} \\ mem_i^{\text{req}} \end{bmatrix}, \qquad \vec{C}_j = \begin{bmatrix} CPU_j \\ MEM_j \end{bmatrix}$$

Знайти розміщення $f : P \to N$, що:

$$\sum_{p_i \in P_j} \vec{r}_i \leq \vec{C}_j \quad \text{для всіх } j.$$

Оптимізаційна мета — мінімізувати кількість задіяних node'ів або сумарну вартість інстансів.

Проблема: задача є NP-складною, тому kube-scheduler застосовує евристики (наприклад, «least requested resources», «balanced resource allocation»).

1.6.3 Стратегії розміщення pod'ів

- **bin-packing (щільне пакування):** поди розміщуються на максимально завантажених вузлах — підвищення ефективності, зменшення кількості вузлів.
- **spreading (розподілення):** розміщення подів рівномірно між вузлами, зонами — підвищення fault tolerance.
- **affinity-aware scheduling:** врахування розміщення інших подів або політик affinity/anti-affinity.

1.6.4 Autoscaling

Kubernetes підтримує кілька типів автоскейлінгу:

Horizontal Pod Autoscaler (HPA)

Автоматично масштабує кількість pod'ів на основі метрик (наприклад, CPU, custom metrics).

$$\text{targetReplicas} = \text{currentReplicas} \times \frac{currentMetric}{targetMetric}$$

Наприклад:

$$\text{CPU target} = 50\%, \quad \text{usage} = 75\% \Rightarrow \text{масштабувати вгору.}$$

Vertical Pod Autoscaler (VPA)

Автоматично змінює `requests`/`limits` pod'а на основі історичного споживання. Вимагає рестарту pod'а.

Cluster Autoscaler

Змінює кількість вузлів у кластері на основі того, чи можуть заплануватись поди. Працює з autoscaling groups у хмарних платформах (AWS ASG, GCP MIG).

1.6.5 Висновок

Практика розміщення pod'ів у Kubernetes базується на складних евристичних алгоритмах, які намагаються враховувати як ресурсні обмеження, так і топологічні політики. Ефективне планування дозволяє досягти балансу між продуктивністю, fault tolerance і вартістю.

1.7 Приклади розміщення подів: autoscaling, типовий хмарний кластер

Для ефективної експлуатації Kubernetes-кластерів у хмарному середовищі необхідно поєднувати знання ресурсної моделі, політик розміщення та механізмів масштабування. У цьому підпункті розглянуто типові сценарії розміщення pod'ів, приклади конфігурацій autoscaler'ів та особливості хмарних платформ, зокрема AWS.

1.7.1 Приклад розміщення подів у кластері з affinity та autoscaling

Розглянемо кластер, у якому поди мають наступні особливості:

- Критичні сервіси (наприклад, БД) розміщуються на вузлах з `nodeSelector: type=dedicated`;
- Поди front-end мають `podAntiAffinity`, щоб не бути на одному вузлі;
- Використовується `Horizontal Pod Autoscaler` для `xenia-frontend`;
- Cluster Autoscaler керує autoscaling-групою вузлів типу `m6i.large`.

Конфігурація HPA:

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: xenia-frontend-hpa
spec:
```

```
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: xenia-frontend
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
```

Конфігурація podAntiAffinity:

```
affinity:
  podAntiAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
    - labelSelector:
        matchExpressions:
        - key: app
          operator: In
          values:
          - xenia-frontend
      topologyKey: "kubernetes.io/hostname"
```

1.7.2 Типова конфігурація вузлів у хмарі (на прикладі AWS)

У кластері, розгорнутому в AWS через Amazon EKS, типова конфігурація node'ів має такі особливості:

- Поділ вузлів за типом: SPOT, On-Demand, Dedicated;
- Вказання лейблів типу:

```
eks.amazonaws.com/nodegroup = xenia-spot
kubernetes.io/lifecycle = spot
```

- Застосування taints:

```
dedicated=xenia-db:NoSchedule
```

- Використання Node Affinity або Tolerations у pod'ах для розміщення.

1.7.3 Потенційні проблеми при розміщенні подів

- **Затримки масштабування:** HPA може спрацювати раніше, ніж буде доступна нова нода (cold start);
- **Taints без tolerations:** pod не може розміститись;
- **Anti-affinity:** обмежує scheduler, викликає Pending pod'и;
- **Недовикористання SPOT:** невикористання економних вузлів через політики.

1.7.4 Висновок

У реальному хмарному середовищі розміщення pod'ів є складною динамічною задачею, яка залежить від великої кількості факторів — від навантаження на додаток до топології датацентру. Поєднання механізмів HPA, affinity, taints і autoscaler'ів забезпечує гнучкість, але потребує грамотної конфігурації та моніторингу.

РОЗДІЛ 2

ПІДХІД ДО ОПТИМІЗАЦІЇ КОНФІГУРАЦІЇ КЛАСТЕРІВ KUBERNETES

2.1 Постановка задачі оптимізації

Управління ресурсами у Kubernetes-кластері є складовою забезпечення стабільної роботи системи, високої продуктивності додатків і раціонального використання обчислювальних потужностей. Kubernetes надає гнучкі інструменти для конфігурування ресурсів за допомогою параметрів `requests` та `limits` для кожного контейнера, однак неправильне чи надмірне їхнє використання часто призводить до значних втрат — як технічних, так і фінансових.

Метою цього дослідження є формалізація проблеми неефективного використання ресурсів у Kubernetes-кластері та побудова математичної моделі, що дозволяє здійснювати оптимальне розміщення контейнерів та налаштування ресурсів.

2.1.1 Формалізація задачі розміщення

Нехай задано множину подів:

$$P = \{p_1, p_2, \ldots, p_n\},$$

де кожен pod p_i характеризується запитами ресурсів cpu_i^{req} та mem_i^{req}, а також фактичним споживанням cpu_i^{use} та mem_i^{use}.

Є також множина вузлів (нодів):

$$N = \{n_1, n_2, \ldots, n_m\},$$

де кожен n_j має обмеження на ресурси: CPU_j (доступні CPU) та MEM_j (доступна пам'ять).

Потрібно знайти таке відображення:

$$f : P \to N,$$

тобто розміщення pod'ів на node'ах, що:

- задовольняє обмеження на ресурси:

$$\sum_{p_i \in P_j} cpu_i^{\text{req}} \leq CPU_j, \qquad \sum_{p_i \in P_j} mem_i^{\text{req}} \leq MEM_j,$$

  де $P_j = \{p_i \in P \mid f(p_i) = n_j\}$ — множина pod'ів, розміщених на ноді n_j;
- враховує обмеження Kubernetes: `affinity`, `taints`, `tolerations`, QoS класи (BestEffort, Burstable, Guaranteed);
- мінімізує цільову функцію — наприклад, кількість задіяних нод або сумарну вартість оренди ресурсів у хмарі:

$$\min_f \sum_{j=1}^{m} C_j \cdot \delta_j,$$

  де C_j — вартість використання j-го вузла (наприклад, EC2 інстансу), $\delta_j = 1$, якщо нода використовується ($P_j \neq \emptyset$), інакше 0.

2.1.2 Мотивація постановки задачі

У хмарному середовищі навіть невелике перевикористання ресурсів призводить до збільшення кількості задіяних вузлів, що тягне за собою значні витрати. З іншого боку, занадто агресивне зменшення ресурсів може спричинити деградацію сервісів (через throttling або OOM-kill).

Таким чином, задача оптимізації полягає у балансуванні між економічною ефективністю та стабільністю роботи кластеру.

2.1.3 Метрики ефективності, що використовуються надалі

- **Ефективність pod'а:**

$$E_i^{\text{cpu}} = \frac{cpu_i^{\text{use}}}{cpu_i^{\text{req}}}, \quad E_i^{\text{mem}} = \frac{mem_i^{\text{use}}}{mem_i^{\text{req}}};$$

- **Щільність використання ресурсів node'а:**

$$L_j^{\text{cpu}} = \frac{\sum_{p \in P_j} cpu_p^{\text{req}}}{CPU_j}, \quad L_j^{\text{mem}} = \frac{\sum_{p \in P_j} mem_p^{\text{req}}}{MEM_j}.$$

Ці метрики будуть використані в подальших підрозділах для визначення потенційних точок оптимізації та формування автоматичних рекомендацій щодо зміни конфігурації кластеру.

2.2 Основні критерії ефективності використання ресурсів

Для здійснення якісної оцінки конфігурації Kubernetes-кластера необхідно визначити ключові метрики, які дозволяють кількісно описати ефективність використання ресурсів. Основна мета полягає у виявленні надмірного або недостатнього резервування CPU та пам'яті на рівні pod'ів і node'ів, а також у подальшій оптимізації цих параметрів.

2.2.1 Ефективність використання ресурсів pod'ом

Нехай pod p_i має запит на ресурси cpu_i^{req}, mem_i^{req} та фактичне споживання cpu_i^{use}, mem_i^{use}. Тоді коефіцієнти ефективності поду за процесорним часом і пам'яттю визначаються як:

$$E_i^{\text{cpu}} = \frac{cpu_i^{\text{use}}}{cpu_i^{\text{req}}}, \qquad E_i^{\text{mem}} = \frac{mem_i^{\text{use}}}{mem_i^{\text{req}}}.$$

Ці значення дозволяють класифікувати pod'и за типами:

- $E < 0.3$: под значно недовикористовує ресурси (over-provisioning);
- $0.3 \leq E \leq 1.0$: под працює в межах запитів (ефективне використання);
- $E > 1.0$: под перевищує запити, можливе throttling або OOM.

2.2.2 Навантаження на вузли (node utilization)

Для кожного вузла n_j визначаються агреговані коефіцієнти навантаження:

$$L_j^{\text{cpu}} = \frac{\sum\limits_{p \in P_j} cpu_p^{\text{req}}}{CPU_j}, \qquad L_j^{\text{mem}} = \frac{\sum\limits_{p \in P_j} mem_p^{\text{req}}}{MEM_j},$$

де $P_j = \{p \in P \mid f(p) = n_j\}$ — множина pod'ів, розміщених на вузлі n_j.

Значення L_j^{cpu} та L_j^{mem} близькі до 1 свідчать про ефективне використання вузла. Якщо ж значення нижче за 0.5, то node є недовантаженим, що може бути сигналом до переміщення pod'ів або зменшення кількості вузлів.

2.2.3 Коефіцієнти "idle" ресурсів

Для оцінки потенціалу оптимізації можна також визначити "idle" (незадіяні) ресурси як:

$$I_j^{\text{cpu}} = 1 - L_j^{\text{cpu}}, \qquad I_j^{\text{mem}} = 1 - L_j^{\text{mem}}.$$

Ці значення дозволяють оцінити, яка частина ресурсів залишилася невикористаною. Особливо корисно для аналізу хмарних витрат — у випадку AWS, наприклад, кожен "idle" екземпляр EC2 створює втрати бюджету.

2.2.4 Узагальнений індекс ефективності кластера

Можна також ввести глобальні показники ефективності всього кластера за CPU та пам'яттю:

$$E_{\text{cpu}}^{\text{cluster}} = \frac{\sum_{i=1}^{n} cpu_i^{\text{use}}}{\sum_{i=1}^{n} cpu_i^{\text{req}}}, \qquad E_{\text{mem}}^{\text{cluster}} = \frac{\sum_{i=1}^{n} mem_i^{\text{use}}}{\sum_{i=1}^{n} mem_i^{\text{req}}}.$$

Ці значення дозволяють оцінити загальну ефективність конфігурації ресурсу на рівні всього кластера. Значення, що близьке до 1, є оптимальним.

2.2.5 Класифікація pod'ів за типом використання

З урахуванням обчислених коефіцієнтів ефективності pod'и можна класифікувати на наступні категорії:

1) **Over-provisioned:** $E^{\text{cpu}} < 0.3$ або $E^{\text{mem}} < 0.3$;
2) **Balanced:** $0.3 \leq E \leq 1.0$;
3) **Under-provisioned:** $E > 1.0$.

Ця класифікація є основою для автоматичного формування рекомендацій щодо зміни конфігурації ресурсів pod'ів у наступних підрозділах.

2.3 Аналіз типових сценаріїв неефективного використання ресурсів

У процесі експлуатації Kubernetes-кластера досить часто виникають ситуації, які призводять до нераціонального використання обчислювальних ресурсів. У цьому підрозділі розглянуто три основні сценарії неефективності: перевикористання ресурсів (overcommit), недовикористання (underutilization) та часткове або повне простоювання ресурсів вузлів (idle nodes). Для кожного сценарію подано приклад, його причини, потенційні наслідки та способи виявлення.

2.3.1 Перевикористання ресурсів (overcommit)

Опис: pod споживає більше ресурсів, ніж зазначено в полі `requests`, тобто:

$$cpu_i^{\text{use}} > cpu_i^{\text{req}} \quad \text{або} \quad mem_i^{\text{use}} > mem_i^{\text{req}}.$$

Причини:

- занадто оптимістичне заниження `requests`;
- використання `BestEffort` або `Burstable` QoS-класів без моніторингу реального споживання;
- відсутність горизонтального автоскейлінгу (HPA).

Наслідки:

- throttling (штучне обмеження CPU scheduler-ом);
- евікція (eviction) pod'ів при браку пам'яті (OOMKilled);
- деградація продуктивності додатку.

2.3.2 Недовикористання ресурсів (underutilization)

Опис: pod запитує більше ресурсів, ніж йому реально потрібно:

$$cpu_i^{\text{use}} \ll cpu_i^{\text{req}} \quad \text{та/або} \quad mem_i^{\text{use}} \ll mem_i^{\text{req}}.$$

Причини:

- завищені запити через «страх нестачі» ресурсів;
- шаблонні значення конфігурацій, не адаптовані під реальні потреби;
- відсутність адаптивного профілювання під навантаження.

Наслідки:

- ресурси резервуються scheduler-ом, але не використовуються;
- node виглядає як перевантажений (бо багато pod'ів із великими `requests`), але фактично має резерв;
- марнування грошей у хмарному середовищі.

2.3.3 Idle вузли (простої ресурсів вузлів)

Опис: окремі node'и мають занадто низьке сумарне навантаження, наприклад:

$$L_j^{\text{cpu}} < 0.3 \quad \text{та} \quad L_j^{\text{mem}} < 0.3.$$

Причини:

- обмеження за `affinity`, `taints/tolerations`, які не дозволяють scheduler-у розміщувати pod'и на цих вузлах;
- поди з високим запитом пам'яті або CPU «фрагментують» ресурси;
- неправильна стратегія масштабування кластера.

Наслідки:

- вузли споживають ресурси хмарного провайдера, але не приносять користі;
- можливе зниження fault tolerance через нерівномірне розміщення pod'ів;
- неефективна робота autoscaler'ів.

2.3.4 Узагальнення

Для формування автоматичних стратегій оптимізації корисно класифікувати всі поди за сценаріями:

- **Overcommitted:** $E > 1.0$;
- **Underutilized:** $E < 0.3$;
- **Balanced:** $0.3 \leq E \leq 1.0$.

Ідентифікація типового сценарію для кожного пода дозволяє автоматизувати рекомендації щодо зміни `requests`, переміщення pod'ів або зміни типів інстансів у хмарі.

2.4 Можливості зменшення витрат через оптимізацію

Окрім забезпечення стабільності та продуктивності, важливою метою оптимізації Kubernetes-кластерів є зменшення експлуатаційних витрат. Це особливо актуально у випадку використання хмарної інфраструктури, де кожен обчислювальний ресурс має пряму грошову вартість. У цьому підрозділі розглянуто декілька практичних напрямів оптимізації, які дозволяють скоротити витрати без шкоди для функціональності додатків.

2.4.1 Зменшення `requests` у недовантажених pod'ів

Поди, які мають значення ефективності $E^{\text{cpu}} < 0.3$ або $E^{\text{mem}} < 0.3$, можуть бути переоцінені з точки зору запитів на ресурси. Зменшення їхніх `requests` дозволяє:

- звільнити ресурси на поточних нодах;
- розмістити більше подів на тих самих інстансах;
- зменшити кількість нод або перейти на менш потужні (і дешевші) типи інстансів.

2.4.2 Збільшення `requests` у перевантажених pod'ів

Pod-и з $E > 1.0$ можуть страждати від throttling або бути завершеними через перевищення пам'яті. Це не лише знижує стабільність, але і призводить до прихованих втрат (повторні запуску, відкат транзакцій тощо). У таких випадках доцільно підвищити `requests`, щоб забезпечити QoS-клас `Guaranteed` або стабільну роботу у межах `Burstable`.

2.4.3 Перехід на дешевші EC2-інстанси (у випадку AWS)

У випадку, коли оптимізація pod'ів зменшує загальну потребу в ресурсах, виникає можливість:

- або зменшити кількість нод;
- або перейти на дешевші інстанси з меншою ємністю.

Приклад:

Припустимо, кластер використовує тип інстансу `m6i.large` (2 vCPU, 8 GB RAM) в AWS з ціною:

$$C_{\text{old}} = 0.096 \text{ USD/год},$$

але після оптимізації достатньо `t3a.medium` (2 vCPU, 4 GB RAM):

$$C_{\text{new}} = 0.0416 \text{ USD/год}.$$

Тоді щомісячна економія на одному інстансі:

$$S = (C_{\text{old}} - C_{\text{new}}) \times 24 \times 30 = (0.096 - 0.0416) \times 720 \approx 39.26 \text{ USD}.$$

Якщо таких інстансів у кластері 10:

$$S_{\text{total}} \approx 392.6 \text{ USD на місяць}.$$

2.4.4 Використання SPOT-інстансів (де можливо)

SPOT-інстанси в AWS, GCP чи Azure можуть зменшити вартість інфраструктури на 60–80% у порівнянні з On-Demand, однак вони не підходять для критичних сервісів. Автоматичне планування розміщення non-critical pod'ів на SPOT-нодах дозволяє ще більше знизити вартість кластера.

Умови застосування:

- сервіс не є stateful або реплікований (наприклад, індексація, аналітика, cron-jobs);
- под має Toleration до SPOT-нод;
- використовується Node Pool з autoscaling.

2.4.5 Прогнозування економії на рівні кластера

Загальна економія на кластері E_{total} може бути оцінена як:

$$E_{\text{total}} = \sum_{j=1}^{m} (C_j^{\text{old}} - C_j^{\text{new}}) \cdot \delta_j \cdot 720,$$

де C_j^{old} та C_j^{new} — вартість старого і нового типу інстансу, $\delta_j = 1$, якщо вузол був оптимізований.

РОЗДІЛ 3

ПРАКТИЧНА РЕАЛІЗАЦІЯ ТА РЕЗУЛЬТАТИ

3.1 Постановка задачі та вхідні дані

У цьому розділі розглядається практична реалізація аналізу та оптимізації використання ресурсів у кластері Kubernetes на основі даних, отриманих із логів, згенерованих командами `kubectl`. Основна мета роботи полягає у виявленні проблем із розподілом ресурсів у кластері, аналізі перевантажень нод, оптимізації розміщення подів за допомогою алгоритму перепаковки та наданні рекомендацій щодо масштабування кластера шляхом підбору відповідних інстансів Amazon Web Services (AWS) EC2.

3.1.1 Опис задачі

Задача полягала в обробці логів Kubernetes для оцінки стану кластера та розробки рекомендацій щодо його оптимізації. Основні етапи включали:

- Збір та класифікація логів із файлів, отриманих командами `kubectl get pods` та `kubectl describe node`.
- Аналіз статусів подів у namespace `xenia-stg` та їх розподілу по нодах.
- Оцінка доступних ресурсів (CPU, пам'ять) на нодах та виявлення перевантажень (overcommit).
- Перепаковка подів за допомогою алгоритму `best-fit-decreasing` для оптимального використання ресурсів.
- Обчислення дефіциту ресурсів для подів, які не вдалося розмістити.
- Підбір інстансів AWS EC2 для покриття дефіциту ресурсів із урахуванням їх характеристик та вартості.

3.1.2 Вхідні дані

Для аналізу використано п'ять файлів логів, із яких три основні — `01_namespace_sample.txt`, `02_describe_node.txt` та

`03_describe_node_another.txt` — містили ключову інформацію про поди та ноди. Додаткові файли

`04_describe_pod.txt` та `05_describe_pod_another.txt` використовувалися для уточнення деталей про окремі поди. Усі дані оброблялися за допомогою Python.

- **Джерело логів**:
  - Файл `01_namespace_sample.txt` — результат команди `kubectl get pods -n xenia-stg -o wide`, що містив список із 10 подів у namespace `xenia-stg`, їх статуси, IP-адреси та розміщення на нодах.
  - Файли `02_describe_node.txt` та `03_describe_node_another.txt` — результати команди `kubectl describe node` для нод `ip-100-100-31-113.eu-west-1.compute.internal` та `ip-100-100-41-106.eu-west-1.compute.internal`, які містили інформацію про доступні ресурси, поди та їх запити/ліміті на ресурси.
- **Структура логів**:
  - Логи подів містили поля: `NAME`, `READY`, `STATUS`, `RESTARTS`, `AGE`, `IP`, `NODE`.
  - Логи нод містили секції `Capacity`, `Allocatable`, `Non-terminated Pods` та `Allocated resources`, де вказувалися запити (requests) та ліміти (limits) на CPU і пам'ять для кожного поду.

Отримані результати слугували основою для подальших етапів аналізу, описаних у наступних секціях.

3.2 Класифікація логів та обробка namespace

Першим кроком було впровадження механізму класифікації лог-файлів, отриманих за допомогою утиліти `kubectl`. Оскільки файли містили змішану інформацію (`get pods`, `describe node`, `describe pod`), для подальшого аналізу необхідно було розподілити їх за категоріями.

3.2.1 Автоматична класифікація логів

Для цього було реалізовано функцію `classify_kubectl_file()`, яка зчитує перші 10 рядків кожного вхідного файлу та ідентифікує його тип на основі ключових підрядків. Відповідно до результату класифікації файли зберігаються в словнику.

Функція `load_all_logs()` проходить по каталогу з логами, застосовує класифікацію до кожного файлу та виводить результат. У рамках поточного дослідження класифікація завершилася з наступним результатом:

```
Класифікація завершена:
Namespace-файл: all_logs\01_namespace_sample.log
Node-файлів: 2
Pod-файлів: 2
Unknown-файлів: []
```

Рис. 3.1. Вивід результатів функції load_all_logs()

3.2.2 Парсинг логів `kubectl get pods`

На основі класифікації було проведено детальний аналіз файлу `01_namespace_sample.log`, що містить результат виконання команди:

```
kubectl get pods -n xenia-stg -o wide
```

Для цього використано функцію `analyze_namespace()`, яка виконує:

- пошук заголовку таблиці (рядок, що починається з `NAME`);
- розбиття кожного подальшого рядка на колонки за регулярним виразом;
- формування датафрейму `pandas.DataFrame` з інформацією про всі pod'и;
- агрегацію за статусом pod'ів та вузлами.

Результатом обробки став набір із 10 pod'ів, розміщених у namespace `xenia-stg`. Виведення результатів подано нижче:

```
Аналіз namespace з файлу: all_logs\01_namespace_sample.log
Завантажено 10 pod-ів

Статус подів:
STATUS
Completed    5
Running      5
Name: count, dtype: int64

Поди по нодах:
NODE
ip-100-100-24-129.eu-west-1.compute.internal    4
ip-100-100-46-4.eu-west-1.compute.internal      1
ip-100-100-31-113.eu-west-1.compute.internal    1
ip-100-100-35-194.eu-west-1.compute.internal    1
ip-100-100-41-106.eu-west-1.compute.internal    1
ip-100-100-58-211.eu-west-1.compute.internal    1
ip-100-100-17-161.eu-west-1.compute.internal    1
Name: count, dtype: int64
```

Рис. 3.2. Вивід результатів функції analyze_namespace

3.2.3 Попередня обробка даних

На цьому етапі також виконувалося:

- автоматичне заповнення порожніх значень у полі `NODE`;
- нормалізація даних у таблиці (обрізання пробілів, вирівнювання кількості колонок).

3.3 Аналіз ресурсів нод та перевантажень

Для оцінки ефективності використання ресурсів у кластері було реалізовано функцію `analyze_node()`, яка обробляє вивід команди `kubectl describe node`. Аналіз здійснювався для кожного з двох лог-файлів:

`02_describe_node.txt` та `03_describe_node_another.txt`.

Основні етапи обробки включають:

1) Зчитування секцій `Capacity` та `Allocatable`, які задають загальні та доступні ресурси CPU, пам'яті та кількості подів.
2) Витяг секції `Non-terminated Pods`, де перелічено всі активні поди з їхніми запитами (`requests`) та лімітами (`limits`) на CPU та пам'ять.
3) Розрахунок сумарного споживання ресурсів та перевірка на **overcommit**, тобто перевищення limits над allocatable.
4) Виявлення подів, які використовують понад 50% ресурсів ноди, а

також тих, що перевищують 100%.

5) Автоматичне розпізнавання імен подів, пов'язаних із базами даних (`mongo`, `redis`, `mysql` тощо).
6) Формування списків подів, у яких взагалі не задані `requests` або `limits`.

Результати аналізу для першої ноди з `02_describe_node.txt`:

```
Аналіз файлу: 02_describe_node.log
Node Name: ip-100-100-31-113.eu-west-1.compute.internal
Capacity: CPU 4000.0m, Memory 7834.453125 Mi, Pods 58
Allocatable: CPU 3920.0m, Memory 6841.453125 Mi, Pods 58
Non-terminated Pods: 12
CPU Requests: 2185.0m | Limits: 6100.0m
Mem Requests: 2424.0 Mi | Limits: 4864.0 Mi
⚠️ CPU limits > 100% — можливий throttling.
✅ Memory usage в межах норми.

Поди, які використовують >50% ресурсів:
                  Name  CPU %  Mem %
xenia-56b6dcb558-6hdql  51.28  31.81

Жоден pod не перевищує 100% ресурсів.

Бази даних не виявлено.

Non-terminated pods:
 - calico-node-brn8p
 - csi-node-driver-878fn
 - falcon-sensor-k5xzq
 - aws-node-cj5dj
 - aws-node-termination-handler-g98fx
 - ebs-csi-node-hsd8l
 - kube-proxy-ftgcs
 - fluent-bit-kszcv
 - monitoring-prometheus-node-exporter-g9jch
 - compliance-benchmarker-x8s8c
 - fluentd-node-jpch7
 - xenia-56b6dcb558-6hdql
```

Рис. 3.3. Результати для першої ноди

```
⚠ Поди без requests:
 - calico-node-brn8p
 - csi-node-driver-878fn
 - falcon-sensor-k5xzq
 - aws-node-termination-handler-g98fx
 - monitoring-prometheus-node-exporter-g9jch
 - compliance-benchmarker-x8s8c
 - fluentd-node-jpch7

⚠ Поди без limits:
 - calico-node-brn8p
 - csi-node-driver-878fn
 - falcon-sensor-k5xzq
 - aws-node-cj5dj
 - aws-node-termination-handler-g98fx
 - kube-proxy-ftgcs
 - monitoring-prometheus-node-exporter-g9jch
 - compliance-benchmarker-x8s8c
 - fluentd-node-jpch7
```

Рис. 3.4. Результати для першої ноди

Результати для другої ноди з `03_describe_node_another.txt`:

```
Аналіз файлу: 03_describe_node_another.log
Node Name: ip-100-100-41-106.eu-west-1.compute.internal
Capacity: CPU 2000.0m, Memory 7794.93359375 Mi, Pods 29
Allocatable: CPU 1930.0m, Memory 7120.93359375 Mi, Pods 29
Non-terminated Pods: 12
CPU Requests: 1185.0m | Limits: 6100.0m
Mem Requests: 2936.0 Mi | Limits: 8960.0 Mi
❌ CPU limits > 300% — критичне перевищення! Рекомендується реорганізувати вузли.
⚠ Memory limits перевищує allocatable.

Поди, які використовують >50% ресурсів:
              Name  CPU %  Mem %
xenia-stg-mongodb-0  52.33  37.75

Жоден pod не перевищує 100% ресурсів.

Виявлені бази даних:
 - xenia-stg-mongodb-0

Non-terminated pods:
 - calico-node-prf59
 - csi-node-driver-bnw7b
 - falcon-sensor-qq8rj
 - aws-node-fv645
 - aws-node-termination-handler-gr6wc
 - ebs-csi-node-lrnkk
 - kube-proxy-8zpf9
 - fluent-bit-889fl
 - monitoring-prometheus-node-exporter-jpvm6
 - compliance-benchmarker-wn9pz
 - fluentd-node-xp94t
 - xenia-stg-mongodb-0
```

Рис. 3.5. Результати для другої ноди

Рис. 3.6. Результати для другої ноди

Математична формалізація: Нехай R_i^{CPU}, L_i^{CPU} — запит та ліміт CPU i-го pod'а, тоді:

$$\text{TotalRequests}^{\text{CPU}} = \sum_{i=1}^{n} R_i^{\text{CPU}}, \quad \text{TotalLimits}^{\text{CPU}} = \sum_{i=1}^{n} L_i^{\text{CPU}}$$

Відношення до `Allocatable` CPU позначимо:

$$\text{Overcommit}^{\text{CPU}} = \frac{\text{TotalLimits}^{\text{CPU}}}{\text{Allocatable}^{\text{CPU}}} \times 100\%$$

Аналогічно для пам'яті.

Якщо $\text{Overcommit}^{\text{CPU}} > 300\%$, то вважається критичним перевантаженням.

3.4 Перепакування подів та оцінка дефіциту ресурсів

Для оцінки ефективності поточного розміщення подів у кластері та пошуку можливостей для оптимізації ресурсів було реалізовано алгоритм **Best-Fit Decreasing** (BFD), що враховує обмеження по CPU та пам'яті.

3.4.1 Нормалізація подів

Перед початком перепакування кожен pod нормалізується: з усіх наданих значень `requests` та `limits` по CPU та пам'яті вибирається максимум, або мінімально допустиме значення (50 mCPU, 64 Mi). Додатково pod-и типу `DaemonSet` виявляються за допомогою регулярного виразу.

3.4.2 Алгоритм перепакування

Після нормалізації виконується наступна процедура:

1) Ініціалізація доступних ресурсів на кожній ноді.
2) Вирахування зарезервованих ресурсів під `DaemonSet`.
3) Впорядкування workload-подів у порядку зменшення споживання ресурсів.
4) По черзі кожен pod розміщується на найбільш підходящу ноду (мінімізується залишок).
5) Якщо под не може бути розміщений — він виноситься у список `unplaced`.

3.4.3 Результати на кластері

На основі аналізу реальних логів (дві ноди, 24 поди), результати були наступними:

- Поди типу DaemonSet були залишені на своїх нодах.
- Жоден workload-pod не зміг бути розміщений через нестачу ресурсів.

- Зокрема, pod-и `xenia-56b6dcb558-6hdql` та `xenia-stg-mongodb-0` перевищують доступні ресурси на будь-якій ноді.
- Обчислений загальний дефіцит: **7450 mCPU** та **1013.6 Mi RAM**.

Node	**Alloc CPU m**	**Alloc RAM Mi**	**DS CPU**	**DS RAM**	**WL CPU**	**WL RAM**	**Free CPU**	**Free RAM**
ip-100-100-31-113	3920	6841	2650	2368	0	0	1270	4473
ip-100-100-41-106	1930	7120	2650	2368	0	0	-720	4752

Табл. 3.1. Розподіл ресурсів після перепакування

3.4.4 Поди, що не умістились

Два поди було класифіковано як `unplaced`, оскільки жодна з нод не мала достатньої кількості CPU і RAM:

- `xenia-stg-mongodb-0` — потребує 4000 mCPU та 7168 Mi.
- `xenia-56b6dcb558-6hdql` — потребує 4000 mCPU та 3072 Mi.

3.4.5 DEMO-сценарій

Для демонстрації було змодельовано два вузли (node-A та node-B) по 2000 mCPU та 4 GiB, та workload з 4 подів (1 великий, 3 малі) та 2 DaemonSet. Усі поди успішно вмістилися після перепакування. Дефіцит ресурсів дорівнював нулю, що підтверджує коректність реалізованого алгоритму.

DEMO-SCENARIO

	Node	Alloc CPU m	Alloc RAM Mi	DaemonSet CPU m	DaemonSet RAM Mi	Workload CPU m	Workload RAM Mi	FREE CPU m	FREE RAM Mi
0	node-A	2 000	4 096	0	0	1 900	1 452	100	2 644
1	node-B	2 000	4 096	0	0	600	600	1 400	3 496

Рух pod-ов (stay / move / unplaced):

	Pod	Old node	New node	Status	CPU m	RAM Mi
1	ds-logger-B	node-B	node-A	move	50	64
0	ds-logger-A	node-A	node-A	stay	50	64
2	svc-big	node-A	node-A	stay	1500	1024
3	svc-sml1	node-A	node-A	stay	300	300
4	svc-sml2	node-B	node-B	stay	300	300
5	svc-sml3	node-B	node-B	stay	300	300

✅ Усі workload-поди уміщаються.

Δ Дефіцит: CPU 0m | RAM 0Mi

Рис. 3.7. Результати для demo сценарію

3.5 Розрахунок дефіциту ресурсів та підбір типу інстансу AWS

Після завершення перепакування подів важливим етапом є оцінка того, скільки ресурсів не вдалося ефективно розмістити в межах поточної конфігурації кластера. Для цього обчислюється підсумковий дефіцит CPU і оперативної пам'яті.

3.5.1 Оцінка дефіциту

На основі списку подів, які не були розміщені (`unplaced`), та залишків ресурсів кожної ноди (`nodes_cap`) реалізується функція `summarize_deficit`

В результаті аналізу з прикладу кластеру дефіцит становив:

- CPU: **7.45 vCPU** (тобто 7450 mCPU)
- RAM: **0.99 GiB** (тобто ≈ 1013.6 MiB)

Ці значення визначають, наскільки більше ресурсів потрібно для того, щоб ефективно розмістити всі поди, включаючи важкі сервіси на кшталт `xenia` чи `mongodb`.

3.5.2 Автоматичний підбір типу інстансу AWS

Для автоматизації рішення щодо масштабування був реалізований модуль, який:

- Завантажує каталог EC2-інстансів з відкритого API `https://instances.vantage.sh`.
- Фільтрує лише ті типи інстансів, які задовольняють мінімальні вимоги по CPU і пам'яті.
- Обирає два варіанти:
  - мінімальний за розміром (vCPU та RAM);
  - мінімальний за ціною (в $/год).

На основі прикладу з регіону `eu-west-1`, система надала такі рекомендації:

Тип інстансу	**vCPU**	**RAM (GiB)**	**Ціна, $ / год**
`c1.xlarge`	8	7.0	0.592
`a1.2xlarge`	8	16.0	0.2304

Табл. 3.2. Рекомендовані EC2-інстанси для масштабування

Перший варіант оптимізує за розміром (для мінімального перевищення ресурсу), другий — за вартістю (мінімальна щогодинна ціна при достатньому розмірі).

```
========  INSTANCE RECOMMENDATION  ========
Дефіцит потрібно покрити ≥ 8.0 vCPU / ≥ 1.0 GiB (регіон eu-west-1)
Рекомендації Worker Node:
   ▸ За розміром : c1.xlarge     8 vCPU / 7.0 GiB   ≈ $0.592/год
   ▸ За ціною   : a1.2xlarge    8 vCPU / 16.0 GiB   ≈ $0.2304/год
```

Рис. 3.8. Рекомендації по інстансам

3.5.3 Переваги такого підходу

- Автоматична адаптація до конфігурацій зростаючого кластеру.

- Можливість динамічного масштабування у хмарі.
- Мінімізація витрат через розгляд альтернатив з найнижчою вартістю.

ВИСНОВКИ

Проблема перевикористання ресурсів у хмарних середовищах є надзвичайно актуальною: за даними досліджень компаній Densify та Flexera, понад 30% хмарних витрат можна уникнути завдяки правильному налаштуванню обмежень і перепланування навантаження. Тому реалізація інструментів аналізу Kubernetes-конфігурацій, як описано в цій роботі, має практичну цінність для зменшення нераціональних витрат і підвищення ефективності.

У цій роботі було проведено повний цикл аналізу Kubernetes-кластера з метою валідації його конфігурації, виявлення перевантажень та формування рекомендацій щодо оптимізації використання ресурсів.

На основі логів, отриманих за допомогою команд `kubectl`, була реалізована система автоматичного класифікування вхідних даних за типом (namespace, pod, node). Це дозволило формалізувати вхідну інформацію та побудувати уніфікований пайплайн обробки.

Було виконано:

- Аналіз подів у namespace `xenia-stg`, зокрема їх статусів та розміщення по нодах. Усі 10 подів знаходилися в статусах `Running` або `Completed`, що дозволило перейти до подальшого ресурсного аналізу.
- Детальний аналіз конфігурацій двох нод на предмет allocatable ресурсів (CPU/пам'ять) та розміщених на них подів. Визначено поди без вказаних лімітів, поди що перевищують доступні ресурси, та співвіднесено поди з DaemonSet-ами.
- Побудовано алгоритм перепаковки подів на основі жадібного підходу `Best-Fit Decreasing`, з урахуванням DaemonSet-навантаження. Це дозволило змоделювати альтернативне розміщення workload-подів і виявити ті, які не можуть бути розміщені у поточному кластері.
- Визначено загальний дефіцит ресурсів: **7.45 vCPU** та **1.0 GiB** оперативної пам'яті. Для покриття цього дефіциту було підключено зовнішній каталог AWS EC2 інстансів, з якого сформовано дві рекомендації:
  - Найменший тип інстансу за розміром: `c1.xlarge` (8 vCPU / 7.0 GiB) за `$0.592/год`.

– Найдешевший інстанс, що покриває дефіцит: `a1.2xlarge` (8 vCPU / 16 GiB) за `$0.2304/год`.

Таким чином, у ході дослідження було досягнуто поставлену мету: створено універсальний інструмент для аналізу конфігурацій кластера Kubernetes, побудовано механізм перепаковки подів і реалізовано логіку підбору нових нод із урахуванням продуктивності та вартості.

СПИСОК ЛІТЕРАТУРИ

1. Бейтс Б., Хорн Д. *Kubernetes в действии.* — М.: ДМК Пресс, 2020. ".
2. The Kubernetes Authors. *Kubernetes Documentation.* — URL: `https://kubernetes.io/docs/`.
3. AWS EC2 Instance Types. — URL: `https://instances.vantage.sh/`.
4. Gurobi Optimization. *Bin Packing Problem.* — URL: `https://www.gurobi.com/resource/bin-packing-problem/`.
5. Hightower K., Burns B., Beda J. *Kubernetes: Up and Running.* — 3rd ed. — O'Reilly Media, 2021. — 350 p.
6. Wei Y., Zhu Z., Liu W. *Resource Scheduling Algorithms in Kubernetes: A Survey.* — ACM Computing Surveys, 2022. ".
7. Мак-Кінні У. *Python для аналізу даних.* — 2-ге вид. — К.: Наш Формат, 2021. ".
8. Wang Y., Zhang X., Zhang L. *An Adaptive Resource Management Framework in Kubernetes.* — IEEE Access, 2020. — Vol. 8. ".
9. Kubernetes for Beginners [Електронний ресурс] // K21 Academy. — Режим доступу: `https://k21academy.com/docker-kubernetes/kubernetes-for-beginners/`.

ДОДАТОК А

Вміст лог-файлу `01_namespace_sample.txt`

```
kubectl get pods -n xenia-stg -o wide
NAME                            READY   STATUS      RESTARTS   AGE     IP              NODE                                            NOMINATED NODE    READINESS
config-manager-28961525-9ncbb   0/1     Completed   0          3d9h    100.100.18.111  ip-100-100-24-129.eu-west-1.compute.internal    <none>            <none>
config-manager-28965845-dkmjz   0/1     Completed   0          9h      100.100.31.170  ip-100-100-24-129.eu-west-1.compute.internal    <none>            <none>
config-manager-tecrp-g28p9      0/1     Completed   0          9m12s   100.100.47.38   ip-100-100-46-4.eu-west-1.compute.internal      <none>            <none>
mongodb-backup-28964280-9lfqk   0/1     Completed   0          35h     100.100.31.153  ip-100-100-24-129.eu-west-1.compute.internal    <none>            <none>
mongodb-backup-28965720-p75fx   0/1     Completed   0          11h     100.100.26.233  ip-100-100-24-129.eu-west-1.compute.internal    <none>            <none>
xenia-56b6dcb558-6hdql          2/2     Running     0          3d21h   100.100.22.196  ip-100-100-31-113.eu-west-1.compute.internal    <none>            <none>
xenia-56b6dcb558-sr4jr          2/2     Running     0          3d22h   100.100.38.210  ip-100-100-35-194.eu-west-1.compute.internal    <none>            <none>
xenia-stg-mongodb-0             3/3     Running     0          73d     100.100.32.141  ip-100-100-41-106.eu-west-1.compute.internal    <none>            <none>
xenia-stg-mongodb-1             3/3     Running     0          73d     100.100.62.149  ip-100-100-58-211.eu-west-1.compute.internal    <none>            <none>
xenia-stg-mongodb-2             3/3     Running     0          73d     100.100.22.110  ip-100-100-17-161.eu-west-1.compute.internal    <none>            <none>
```

ДОДАТОК Б

Вміст лог-файлу `02_describe_node.txt`

```
kubectl describe node xenia-node
Name:               ip-100-100-31-113.eu-west-1.compute.internal
Roles:              <none>
Labels:             app=xenia_large
                    aws-node-termination-handler/managed=true
                    beta.kubernetes.io/arch=amd64
                    beta.kubernetes.io/instance-type=c5a.xlarge
                    beta.kubernetes.io/os=linux
                    eks.amazonaws.com/capacityType=SPOT
                    eks.amazonaws.com/nodegroup=awspgdev-xenia_spot_large20241122094524184100000002
                    eks.amazonaws.com/nodegroup-image=ami-02e2de73058d55743
                    eks.amazonaws.com/sourceLaunchTemplateId=lt-09c468e4d8e0c1300
                    eks.amazonaws.com/sourceLaunchTemplateVersion=1
                    failure-domain.beta.kubernetes.io/region=eu-west-1
                    failure-domain.beta.kubernetes.io/zone=eu-west-1a
                    k8s.io/cloud-provider-aws=43143ff6646c453b739da35142adf825
                    kubernetes.io/arch=amd64
                    kubernetes.io/hostname=ip-100-100-31-113.eu-west-1.compute.internal
                    kubernetes.io/os=linux
                    node.kubernetes.io/instance-type=c5a.xlarge
                    topology.ebs.csi.aws.com/zone=eu-west-1a
                    topology.kubernetes.io/region=eu-west-1
                    topology.kubernetes.io/zone=eu-west-1a
Annotations:        alpha.kubernetes.io/provided-node-ip: 100.100.31.113
                    csi.volume.kubernetes.io/nodeid: {"csi.tigera.io":"ip-100-100-31-113.eu-west-1.compute.internal","ebs.csi.aws.com":"i-01a0e440b65a2910a"}
                    node.alpha.kubernetes.io/ttl: 0
                    volumes.kubernetes.io/controller-managed-attach-detach: true
CreationTimestamp:  Thu, 23 Jan 2025 14:34:12 +0000
Taints:             dedicated=xenia_large:NoSchedule
Unschedulable:      false
Lease:
  HolderIdentity:  ip-100-100-31-113.eu-west-1.compute.internal
  AcquireTime:     <unset>
  RenewTime:       Mon, 27 Jan 2025 13:18:30 +0000
Conditions:
  Type             Status  LastHeartbeatTime                 LastTransitionTime                Reason                       Message
  ----             ------  -----------------                 ------------------                ------                       -------
  MemoryPressure   False   Mon, 27 Jan 2025 13:17:11 +0000   Thu, 23 Jan 2025 14:34:11 +0000   KubeletHasSufficientMemory   kubelet has sufficient memory

  available
  DiskPressure     False   Mon, 27 Jan 2025 13:17:11 +0000   Thu, 23 Jan 2025 14:34:11 +0000   KubeletHasNoDiskPressure     kubelet has no disk pressure
  PIDPressure      False   Mon, 27 Jan 2025 13:17:11 +0000   Thu, 23 Jan 2025 14:34:11 +0000   KubeletHasSufficientPID      kubelet has sufficient PID available
  Ready            True    Mon, 27 Jan 2025 13:17:11 +0000   Thu, 23 Jan 2025 14:34:25 +0000   KubeletReady                 kubelet is posting ready status
Addresses:
  InternalIP:   100.100.31.113
  InternalDNS:  ip-100-100-31-113.eu-west-1.compute.internal
  Hostname:     ip-100-100-31-113.eu-west-1.compute.internal
Capacity:
  cpu:                4
  ephemeral-storage:  31444972Ki
  hugepages-1Gi:      0
  hugepages-2Mi:      0
  memory:             8022480Ki
  pods:               58
Allocatable:
  cpu:                3920m
  ephemeral-storage:  27905944324
  hugepages-1Gi:      0
  hugepages-2Mi:      0
  memory:             7005648Ki
  pods:               58
System Info:
  Machine ID:                 ec2036f8000e05e9cf1b9b82f434e7af
  System UUID:                ec2036f8-000e-05e9-cf1b-9b82f434e7af
  Boot ID:                    3ac77b60-5998-4afe-a9df-6a19181bb84d
  Kernel Version:             5.10.210-201.852.amzn2.x86_64
  OS Image:                   Amazon Linux 2
  Operating System:           linux
  Architecture:               amd64
  Container Runtime Version:  containerd://1.7.11
  Kubelet Version:            v1.29.0-eks-5e0fdde
```

```
  Kube-Proxy Version:         v1.29.0-eks-5e0fdde
ProviderID:                   aws:///eu-west-1a/i-01a0e440b65a2910a
Non-terminated Pods:          (12 in total)
  Namespace                   Name                                       CPU Requests  CPU Limits   Memory Requests  Memory Limits  Age
  ---------                   ----                                       ------------  ----------   ---------------  -------------  ---
  calico-system               calico-node-brn8p                          0 (0%)        0 (0%)       0 (0%)           0 (0%)         3d22h
  calico-system               csi-node-driver-878fn                      0 (0%)        0 (0%)       0 (0%)           0 (0%)         3d22h
  falcon-system               falcon-sensor-k5xzq                        0 (0%)        0 (0%)       0 (0%)           0 (0%)         3d22h
  kube-system                 aws-node-cj5dj                             25m (0%)      0 (0%)       0 (0%)           0 (0%)         3d22h
  kube-system                 aws-node-termination-handler-g98fx         0 (0%)        0 (0%)       0 (0%)           0 (0%)         3d22h
  kube-system                 ebs-csi-node-hsd8l                         30m (0%)      0 (0%)       120Mi (1%)       768Mi (11%)    3d22h
  kube-system                 kube-proxy-ftgcs                           100m (2%)     0 (0%)       0 (0%)           0 (0%)         3d22h
  logging                     fluent-bit-kszcv                           20m (0%)      2100m (53%)  128Mi (1%)       1Gi (14%)      3d22h
  monitoring                  monitoring-prometheus-node-exporter-g9jch  0 (0%)        0 (0%)       0 (0%)           0 (0%)         3d22h
  tigera-compliance           compliance-benchmarker-x8s8c               0 (0%)        0 (0%)       0 (0%)           0 (0%)         3d22h
  tigera-fluentd              fluentd-node-jpch7                         0 (0%)        0 (0%)       0 (0%)           0 (0%)         3d22h
  xenia-stg                   xenia-56b6dcb558-6hdql                     2010m (51%)   4 (102%)     2176Mi (31%)     3Gi (44%)      3d21h
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  Resource           Requests      Limits
  --------           --------      ------
  cpu                2185m (55%)   6100m (155%)
  memory             2424Mi (35%)  4864Mi (71%)
  ephemeral-storage  0 (0%)        0 (0%)
  hugepages-1Gi      0 (0%)        0 (0%)
  hugepages-2Mi      0 (0%)        0 (0%)
Events:              <none>
```

ДОДАТОК В

Вміст лог-файлу `03_describe_node_another.txt`

```
kubectl describe node mongo-node
Name:               ip-100-100-41-106.eu-west-1.compute.internal
Roles:              <none>
Labels:             app=xenia_stg_db
                    aws-node-termination-handler/managed=true
                    beta.kubernetes.io/arch=amd64
                    beta.kubernetes.io/instance-type=m6i.large
                    beta.kubernetes.io/os=linux
                    eks.amazonaws.com/capacityType=ON_DEMAND
                    eks.amazonaws.com/nodegroup=awspgdev-xenia_stg_db_ondemand20240816174818601100000002
                    eks.amazonaws.com/nodegroup-image=ami-02e2de73058d55743
                    eks.amazonaws.com/sourceLaunchTemplateId=lt-0540e0f6323de5d45
                    eks.amazonaws.com/sourceLaunchTemplateVersion=1
                    failure-domain.beta.kubernetes.io/region=eu-west-1
                    failure-domain.beta.kubernetes.io/zone=eu-west-1b
                    k8s.io/cloud-provider-aws=43143ff6646c453b739da35142adf825
                    kubernetes.io/arch=amd64
                    kubernetes.io/hostname=ip-100-100-41-106.eu-west-1.compute.internal
                    kubernetes.io/os=linux
                    node.kubernetes.io/instance-type=m6i.large
                    topology.ebs.csi.aws.com/zone=eu-west-1b
                    topology.kubernetes.io/region=eu-west-1
                    topology.kubernetes.io/zone=eu-west-1b
Annotations:        alpha.kubernetes.io/provided-node-ip: 100.100.41.106
                    csi.volume.kubernetes.io/nodeid: {"csi.tigera.io":"ip-100-100-41-106.eu-west-1.compute.internal","ebs.csi.aws.com":"i-05424cd68c89bfbac"}
                    node.alpha.kubernetes.io/ttl: 0
                    volumes.kubernetes.io/controller-managed-attach-detach: true
CreationTimestamp:  Fri, 16 Aug 2024 17:49:30 +0000
Taints:             dedicated=xenia_stg_db:NoSchedule
Unschedulable:      false
Lease:
  HolderIdentity:  ip-100-100-41-106.eu-west-1.compute.internal
  AcquireTime:     <unset>
  RenewTime:       Mon, 27 Jan 2025 13:19:12 +0000
Conditions:
  Type             Status  LastHeartbeatTime                 LastTransitionTime                Reason                       Message
  ----             ------  -----------------                 ------------------                ------                       -------
  MemoryPressure   False   Mon, 27 Jan 2025 13:16:47 +0000   Sat, 17 Aug 2024 17:58:24 +0000   KubeletHasSufficientMemory   kubelet has sufficient memory

  available
  DiskPressure     False   Mon, 27 Jan 2025 13:16:47 +0000   Sat, 17 Aug 2024 17:58:24 +0000   KubeletHasNoDiskPressure     kubelet has no disk pressure
  PIDPressure      False   Mon, 27 Jan 2025 13:16:47 +0000   Sat, 17 Aug 2024 17:58:24 +0000   KubeletHasSufficientPID      kubelet has sufficient PID available
  Ready            True    Mon, 27 Jan 2025 13:16:47 +0000   Sat, 17 Aug 2024 17:58:24 +0000   KubeletReady                 kubelet is posting ready status
Addresses:
  InternalIP:   100.100.41.106
  InternalDNS:  ip-100-100-41-106.eu-west-1.compute.internal
  Hostname:     ip-100-100-41-106.eu-west-1.compute.internal
Capacity:
  cpu:                2
  ephemeral-storage:  31444972Ki
  hugepages-1Gi:      0
  hugepages-2Mi:      0
  memory:             7982012Ki
  pods:               29
Allocatable:
  cpu:                1930m
  ephemeral-storage:  27905944324
  hugepages-1Gi:      0
  hugepages-2Mi:      0
  memory:             7291836Ki
  pods:               29
System Info:
  Machine ID:                 ec22f6aa661576c27bfa2d5b501f6d1e
  System UUID:                ec22f6aa-6615-76c2-7bfa-2d5b501f6d1e
  Boot ID:                    6a2e19bc-971e-4018-8f76-af9c93e12bae
  Kernel Version:             5.10.210-201.852.amzn2.x86_64
  OS Image:                   Amazon Linux 2
  Operating System:           linux
  Architecture:               amd64
  Container Runtime Version:  containerd://1.7.11
  Kubelet Version:            v1.29.0-eks-5e0fdde
```

```
  Kube-Proxy Version:         v1.29.0-eks-5e0fdde
ProviderID:                   aws:///eu-west-1b/i-05424cd68c89bfbac
Non-terminated Pods:          (12 in total)
  Namespace                   Name                                       CPU Requests  CPU Limits    Memory Requests  Memory Limits  Age
  ---------                   ----                                       ------------  ----------    ---------------  -------------  ---
  calico-system               calico-node-prf59                          0 (0%)        0 (0%)        0 (0%)           0 (0%)         163d
  calico-system               csi-node-driver-bnw7b                      0 (0%)        0 (0%)        0 (0%)           0 (0%)         163d
  falcon-system               falcon-sensor-qq8rj                        0 (0%)        0 (0%)        0 (0%)           0 (0%)         11d
  kube-system                 aws-node-fv645                             25m (1%)      0 (0%)        0 (0%)           0 (0%)         163d
  kube-system                 aws-node-termination-handler-gr6wc         0 (0%)        0 (0%)        0 (0%)           0 (0%)         163d
  kube-system                 ebs-csi-node-lrnkk                         30m (1%)      0 (0%)        120Mi (1%)       768Mi (10%)    163d
  kube-system                 kube-proxy-8zpf9                           100m (5%)     0 (0%)        0 (0%)           0 (0%)         163d
  logging                     fluent-bit-889fl                           20m (1%)      2100m (108%)  128Mi (1%)       1Gi (14%)      48d
  monitoring                  monitoring-prometheus-node-exporter-jpvm6  0 (0%)        0 (0%)        0 (0%)           0 (0%)         163d
  tigera-compliance           compliance-benchmarker-wn9pz               0 (0%)        0 (0%)        0 (0%)           0 (0%)         163d
  tigera-fluentd              fluentd-node-xp94t                         0 (0%)        0 (0%)        0 (0%)           0 (0%)         163d
  xenia-stg                   xenia-stg-mongodb-0                        1010m (52%)   4 (207%)      2688Mi (37%)     7Gi (100%)     73d
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  Resource           Requests      Limits
  --------           --------      ------
  cpu                1185m (61%)   6100m (316%)
  memory             2936Mi (41%)  8960Mi (125%)
  ephemeral-storage  0 (0%)        0 (0%)
  hugepages-1Gi      0 (0%)        0 (0%)
  hugepages-2Mi      0 (0%)        0 (0%)
Events:              <none>
```

ДОДАТОК Г

Повний вихідний код, реалізований у рамках дипломної роботи, доступний у відкритому репозиторії за посиланням:

`https://github.com/alex-ryb/dipl/blob/main/kub3.0.ipynb`