

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ
СІКОРСЬКОГО»**

**Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій.**

"На правах рукопису"
УДК 004.453

ДО ЗАХИСТУ ДОПУЩЕНО
Завідувач кафедри

_____ Олександр РОЛІК

“ _____ ” _____ 2023 р.

МАГІСТЕРСЬКА ДИСЕРТАЦІЯ

на здобуття ступеня магістра

за освітньо-науковою програмою

«Інформаційне забезпечення робототехнічних систем»

зі спеціальності 126 *«Інформаційні системи та технології»*

на тему:

***«Система ініціалізації та управління контейнерами і зовнішніми сервісами
у Kubernetes кластері»***

Виконав:

студент VI курсу, групи ІК-11мн

Калашник Роман Сергійович _____

Керівник:

доцент, к.т.н., доцент,

Крилов Євген Володимирович _____

Рецензент:

завідувач кафедри СП, д.т.н., проф.,

Мухін Вадим Євгенійович _____

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

Київ – 2023 року

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»**

Факультет інформатики та обчислювальної техніки

Кафедра інформаційних систем та технологій

Рівень вищої освіти – другий (магістерський)

Спеціальність – 126 «Інформаційні системи та технології»

Освітньо-наукова програма «Інформаційне забезпечення робототехнічних систем»

ЗАТВЕРДЖУЮ
Завідувач кафедри

_____ Олександр РОЛІК

«__» _____ 2023 р.

**ЗАВДАННЯ
на магістерську дисертацію студенту
Калашник Роман Сергійович**

1. Тема дисертації «Система ініціалізації та управління контейнерами і зовнішніми сервісами у Kubernetes кластері»

науковий керівник дисертації Крилов Євген Володимирович, к.т.н., доцент,
затверджені наказом по університету від «20» березня 2023 р. № 1275-с

2. Строк подання студентом дисертації “ 12 ” травня 20 23 р.

3. Об'єкт дослідження: дослідження є процес створення та управління Kubernetes кластерів, сервісів, контейнерів

4. Предмет дослідження: система ініціалізації та управління контейнерами і зовнішніми сервісами у Kubernetes кластері

5. Перелік завдань, які потрібно розробити: є аналіз інструментів, аналіз комбінацій інструментів, аналіз методів, аналіз взаємодій, проведення експериментів, розробка нових методів, розробка нових схем, розробка нових комбінацій, вивчення проблеми, проведення дослідів, розробка системи, розробка алгоритму користування, проведення експериментів для перевірки ефективності розроблених методів.

6. Перелік графічного (ілюстративного) матеріалу: загальна схема процесів, схема процесів ініціалізації та управління зовнішніх сервісів та кластеру, схема процесів ініціалізації контейнерів, схема процесів управління контейнерами, алгоритм використання системи, діаграма branching strategy, архітектура системи.

7. Орієнтовний перелік публікацій

Тези на IV Міжнародна науково практична конференція молодих вчених та студентів «ІНЖЕНЕРІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ І ПЕРЕДОВІ ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ», назва статті “Система ініціалізації та управління контейнерами і зовнішніми сервісами у Kubernetes кластері”

8. Консультанти розділів дисертації

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

9. Дата видачі завдання “ 31 ” січня 20 23 р.

Календарний план

	№ з/п	Назва етапів виконання магістерської дисертації	Строк виконання етапів магістерської дисертації	Примітка
	1	Огляд існуючих рішень	14.02	
	2	Аналіз Kubernetes, сервісів та контейнерів	27.02	
	3	Вибір сервісів та інструментів для створення системи	01.03	
	4	Розробка системи	01.03	
	5	Розгортання кластеру та сервісів і контейнерів	02.03	
	7	Тестування розробленої системи	01.04	
	8	Оформлення документації	19.04	
	9	Подання роботи на попередній захист	20.04	
	10	Подання роботи на основний захист	17.05	

Студент

Роман КАЛАШНИК

Науковий керівник

Євген КРИЛОВ

РЕФЕРАТ

Магістерська дисертація: 123 с., 25 рис., 2 табл., 35 джерел, 1 додаток.

Актуальність оскільки підприємства широко використовують контейнерні технології для розгортання додатків, а з ростом їх складності та кількості, управління ними стає важким завданням, то розробка системи ініціалізації та управління контейнерами та зовнішніми сервісами у Kubernetes кластері є актуальною. Розробка ефективної системи ініціалізації та управління контейнерами та зовнішніми сервісами у Kubernetes кластері може поліпшити продуктивність, зменшити витрати та покращити доступність технологій.

Об’єктом дослідження є процес створення та управління Kubernetes кластерів, сервісів, контейнерів.

Метою магістерської дисертації є зменшення часу на створення та розгортання контейнерів (на прикладі Kubernetes cluster).

Завданням є аналіз інструментів, аналіз комбінацій інструментів, аналіз методів, аналіз взаємодій, проведення експериментів, розробка нових методів, розробка нових схем, розробка нових комбінацій, вивчення проблеми, проведення дослідів, розробка системи, розробка алгоритму користування, проведення експериментів для перевірки ефективності розроблених методів.

Предметом дослідження є система ініціалізації та управління контейнерами і зовнішніми сервісами у Kubernetes кластері.

Розроблено підсистему для ініціалізації контейнерів, зовнішніх сервісів та Kubernetes кластерів мовами YAML та HCL з використанням сервісів Gitlab та Terraform і також розроблено підсистему управління контейнерами, зовнішніми сервісами та Kubernetes кластерами мовами YAML та HCL з використанням сервісів Gitlab, Terraform.

Розроблена система може бути далі розроблена та застосована у сфері розробки програмного забезпечення та операцій, особливо при використанні мікросервісної архітектури.

Наукова новизна створено нову конфігурацію системи яка дозволяю значно пришвидшити та покращити процес ініціалізації та управління контейнерами, зовнішніми сервісами та Kubernetes кластером.

Публікації. Тези на IV Міжнародна науково практична конференція молодих вчених та студентів «ІНЖЕНЕРІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ І ПЕРЕДОВІ ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ», назва статті “Система ініціалізації та управління контейнерами і зовнішніми сервісами у Kubernetes кластері”.

KUBERNETES, GITLAB, TERRAFORM, МІКРОСЕРВІСНА АРХІТЕКТУРА, ХМАРНІ ОБЧИСЛЕННЯ, УПРАВЛІННЯ КОНТЕЙНЕРАМИ, УПРАВЛІННЯ СЕРВІСАМИ, УПРАВЛІННЯ КЛАСТЕРОМ, ІНІЦІАЛІЗАЦІЯ КОНТЕЙНЕРІВ, ІНІЦІАЛІЗАЦІЯ СЕРВІСІВ, ІНІЦІАЛІЗАЦІЯ КЛАСТЕРУ.

ABSTRACT

Master's thesis: 123 p., 25 figures, 2 tables, 35 sources, 1 appendix.

Relevance Since enterprises widely use container technologies to deploy applications, and with the growth of their complexity and number, managing them becomes a difficult task, that's why the development of the system for initializing and managing containers and external services in a Kubernetes cluster is relevant. Developing an effective system for initializing and managing containers and external services in a Kubernetes cluster can improve performance, reduce costs, and improve technology availability.

The object of research is the process of creating and managing Kubernetes clusters, services, containers.

The purpose of the master's thesis is to reduce the time for creating and deploying containers (using Kubernetes cluster as an example).

The task is to analyze tools, analyze combinations of tools, analyze methods, analyze interactions, conduct experiments, develop new methods, develop new schemes, develop new combinations, study the problem, conduct experiments, develop a system, develop a user algorithm, conduct experiments to test the effectiveness of the developed methods.

The subject of the research is a system for initializing and managing containers and external services in a Kubernetes cluster.

A subsystem for initializing containers, external services and Kubernetes clusters in YAML and HCL using Gitlab and Terraform services has been developed, and a subsystem for managing containers, external services and Kubernetes clusters in YAML and HCL using Gitlab and Terraform services has also been developed.

The developed system can be further developed and applied in the field of software development and operations, especially when using microservice architecture.

The scientific novelty is a new system configuration that allows to significantly speed up and improve the process of initialization and management of containers, external services and Kubernetes cluster.

Publications. Abstracts at the IV International Scientific and Practical Conference of Young Scientists and Students "SOFTWARE ENGINEERING AND ADVANCED INFORMATION TECHNOLOGIES", the title of the article "System for initializing and managing containers and external services in the Kubernetes cluster".

KUBERNETES, GITLAB, TERRAFPM, MICROSERVICE ARCHITECTURE, CLOUD COMPUTING, CONTAINER MANAGEMENT, SERVICE MANAGEMENT, CLUSTER MANAGEMENT, CONTAINER INITIALIZATION, SERVICE INITIALIZATION, CLUSTER INITIALIZATION.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ	10
ВСТУП.....	12
1 ОГЛЯД ІСНУЮЧИХ РІШЕНЬ	15
1.1 Ініціалізація Kubernetes.....	15
1.1.1 Конфігураційні файли Kubernetes.....	16
1.1.2 Контроллер змін та конфігурацій.....	19
1.2 Ініціалізація контейнерів	21
1.2.1 Мануальна ініціалізація	23
1.2.2 Автоматична ініціалізація.....	25
1.3 Управління контейнерами.....	26
1.4 Ініціалізація сервісів.....	27
1.5 Висновки до розділу.....	33
2 АНАЛІЗ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ ТА KUBERNETES SERVICU ..	34
2.1 Мікросервісна архітектура	35
2.1.1 Аналіз підходу	37
2.1.1 Взаємодія між мікросервісами.....	40
2.1.3 Розгортання контейнерів у хмарах.....	41
2.1.4 Пакування контейнеру за допомогою технології Docker.....	45
2.1.5 Розгортання сервісів у хмарах.....	50
2.2. Оркестрація контейнерів.....	56
2.2.1 Огляд використання Kubernetes	61
2.2.2 Огляд складових Kubernetes	62
2.2.3 Аналіз типів оточень запуску контейнерів.....	68
2.2.4 Поди у Kubernetes.....	69
2.2.5 Deployments у Kubernetes	81
2.2.6 Replica Sets у Kubernetes.....	85
2.2.7 ReplicationController у Kubernetes	87
2.3 Висновки до розділу.....	88

3 РОЗРОБЛЕННЯ СИСТЕМИ ІНІЦІАЛІЗАЦІЇ ТА УПРАВЛІННЯ КОНТЕЙНЕРІВ І ЗОВНІШНІХ СЕРВІСІВ	90
3.1 Аналіз вимог до оточення	90
3.2 Аналіз вимог до системи	90
3.3.1 Namespaces	92
3.3.2 RBAC	94
3.3.3 VPC	94
3.3.4 Subnet	95
3.4 Сценарій використання системи	96
3.5 Вибір та обґрунтування сервісів та засобів реалізації	98
3.5.1 Вибір мов програмування	98
3.5.2 Вибір середовища програмування	100
3.5.3 Засоби для розробки	101
3.5.4 Вибір підходу ініціалізації та управління кластера у AWS	103
3.5.5 Вибір підходу ініціалізації та управління контейнерів	105
3.5.6 Вибір підходу ініціалізації та управління зовнішніх сервісів	106
3.6 Короткий опис програми ініціалізації контейнерів	108
3.8 Короткий опис системи ініціалізації та управління кластера	110
3.9 Короткий опис системи ініціалізації та управління сервісами	112
3.10 Висновки до розділу	114
4 АНАЛІЗ ЕФЕКТИВНОСТІ РОБОТИ СИСТЕМИ	116
4.1 Ініціалізація та управління кластеру та сервісів	116
4.2 Ініціалізація та управління контейнерів	117
4.3 Висновки до розділу	118
ВИСНОВКИ	120
ПЕРЕЛІК ПОСИЛАНЬ	122
ДОДАТОК А	126
Плакат 1 Загальна Схема процесів	127
Плакат 2 Схема процесів ініціалізації та управління зовнішніх сервісів та кластеру	128
Плакат 3 Схема процесів ініціалізації контейнерів	129

Плакат 4 Схема процесів управління контейнерами.....	130
Плакат 5 Алгоритм використання системи	131
Плакат 6 Діаграма branching strategy	132
Плакат 7 Архітектура системи	133

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ

VM – Віртуальна машина
OS – Операційна система
RBAC- Role-based access control
БД – База даних
AWS – Amazon Web Services
CRUD – Create Read Update
DNS – Domain Name System
EKS – Elastic Kubernetes Service
GA – Generally Available
GCP – Google Cloud Platform
ABAC – Attribute Based Access
FailureSSD – Solid State Drive
SaaS – Software as a Service
PaaS – Platform as a Service
NFS – Network File System
IAM – Identity and Access
API – Application Programming
CRD – Custom Resource Definition
GKE – Google Kubernetes Engine
NAT – Network Address Translation
EC2 – Elastic Compute Cloud
HDD – Hard Disk Drive
IaaS – Infrastructure as a Service
ManagementLTS – Long Term Support
AKS - Azure Kubernetes Service
ACI - Azure Container Instances
ECS - Elastic Container service
DevOps - Development Operations
CLI - command-line interface

CI/CD - continuous integration / continuous delivery

ВСТУП

Стрімкий розвиток технологій і величезна кількість користувачів є однією з основних труднощів при побудові складних сучасних інформаційних систем. В умовах сучасного світу програмний продукт як єдине ціле складно змінити під мінливі вимоги ринку. Тому незалежні функції системи повинні бути призначені різним компонентам відповідно до різних потреб і вимог.

Створення програмного забезпечення з використанням архітектур мікросервісів стає все більш важливим підходом порівняно з монолітними архітектурами. Але контролювати велику кількість послуг одночасно досить складно. Таким чином, у процесі розробки та супроводу програмного забезпечення виникає новий клас проблем, включаючи проблему оновлень і версій протоколів мікросервісів, контролю за їх розгортанням та ініціалізацією, диференціації рівнів доступу до ресурсів інфраструктури.

Віртуалізація в сучасному світі є невід’ємною частиною побудови будь-якої платформи, оскільки дозволяє використовувати фізичні обчислювальні ресурси оптимальним чином, гарантуючи безпеку та простоту використання. Однак віртуалізатор використовує значну кількість обчислювальних ресурсів для імітації повного ядра операційної системи, процесора та інших фізичних компонентів ПК. З іншого боку, контейнеризація покладається на вбудовані механізми операційної системи для ізоляції робочого середовища процесу. Кожен контейнер є окремим процесом з окремими частинами пам’яті та процесорного часу.

Контейнеризація або віртуалізація на рівні ОС — це метод віртуалізації, за якого ядро операційної системи підтримує кілька незалежних екземплярів простору користувача замість одного. З точки зору користувача, ці екземпляри або контейнери точно еквівалентні одному екземпляру операційної системи. Для систем на базі Unix ця техніка схожа на модифіковану реалізацію механізму chroot. Ядро забезпечує повну ізоляцію контейнерів, і програми з різних контейнерів не впливають одна на одну.

Однією з відмінностей між апаратною віртуалізацією, яка повністю емулює апаратне середовище, є те, що контейнери можуть запускати лише операційну

систему, яка має те саме ядро, що й операційна система хоста. Контейнеризація також не витрачає час на моделювання віртуального обладнання, тому створення контейнера займає значно менше часу, ніж створення віртуальної машини.

Docker — це програмне забезпечення для автоматизації розгортання та керування програмами в середовищах, які підтримують контейнеризацію. За своєю суттю Docker дозволяє безпечно запускати майже будь-яку програму в контейнері. Ізоляція безпеки дозволяє запускати кілька контейнерів одночасно на одному сервері для оптимального використання обчислювальних ресурсів.

Terraform — це програмне забезпечення з відкритим кодом, розроблене HashiCorp. Користувачі визначають і налаштовують інфраструктуру центру обробки даних за допомогою декларативної мови конфігурації під назвою HashiCorp Configuration Language (HCL). Terraform співпрацює з постачальниками для управління зовнішніми ресурсами (такими як публічна хмарна інфраструктура, приватна хмарна інфраструктура, мережеве обладнання, програмне забезпечення як послуга та платформа як послуга). Замість використання імперативних команд для створення ресурсів Terraform використовує декларативну конфігурацію для опису бажаного кінцевого стану. Щойно користувач викликає Terraform для цього ресурсу, Terraform виконуватиме операції CRUD від імені користувача для досягнення потрібного стану. Інфраструктура як код може бути написана у формі модулів, що забезпечує повторне використання коду та зручність обслуговування.

GitLab — це веб-сховище Git, яке надає безкоштовні загальнодоступні та приватні сховища, відстеження проблем і вікі. Це повна платформа DevOps, яка дозволяє професіоналам виконувати всі проектні завдання, від планування проекту та керування вихідним кодом до моніторингу та безпеки. Це також дозволяє командам співпрацювати та створювати краще програмне забезпечення. GitLab допомагає командам скоротити життєвий цикл продуктів і підвищити продуктивність, створюючи цінність для клієнтів. Програма не вимагає від користувача керування авторизацією для кожного інструменту. Якщо ви налаштуєте дозволи один раз, кожен у вашій організації матиме доступ до кожного компонента.

Щоб побудувати повну платформу з використанням архітектури мікросервісів, потрібен механізм для керування сотнями, іноді тисячами, контейнерів одночасно. Це завдання вирішують оркестрови, найрозвиненішим і найпопулярнішим з яких є Kubernetes від Google.

Однак керування контейнерами в кластері Kubernetes є досить складним завданням з точки зору часу та ресурсів. Крім того, етап ініціалізації контейнерів, включаючи їх запуск і налаштування, часто є складною частиною керування платформою. Також великою проблемою є ініціалізація та управління сервісами та самим кластером у створюваній системі. Завданням магістерської роботи була розробка системи вирішення цих проблем.

1 ОГЛЯД ІСНУЮЧИХ РІШЕНЬ

По-перше, розглядаються існуючі рішення для ініціалізації робочих навантажень Kubernetes та ініціалізації ресурсів. Аналіз комерційного ринку та open-source рішень показує, що існує досить багато програмних комплексів, які можуть вирішити конкретний перелік існуючих проблем, і навіть деякі з них можуть використовуватися без додаткових компонентів завдяки використанню механізмів, вбудованих у Kubernetes.

1.1 Ініціалізація Kubernetes

Ініціалізація Kubernetes — це процес підготовки кластера Kubernetes до роботи. Включаючи конфігурацію вузла кластера, конфігурацію головного вузла, конфігурацію мережі, інсталяцію програми та інші операції.

Першим кроком у процесі ініціалізації Kubernetes є встановлення та налаштування головного вузла. Головний вузол відповідає за керування всім кластером Kubernetes. Після встановлення вузла керування конфігурація його параметрів (наприклад, встановлення API сервера, додавання сертифікатів, налаштування мережі тощо) забезпечить надійну роботу кластера.

Другим кроком є додавання вузлів до кластера. Це можна зробити за допомогою різних інструментів, таких як kubeadm, kops, eksctl тощо. Після налаштування вузлів і додавання їх до кластера ви можете перевірити, чи вони належним чином підключені до головного вузла, і надати вузлу необхідні ресурси.

Наступним кроком буде налаштування мережі. Мережа Kubernetes дозволяє вузлам і програмам спілкуватися один з одним. Конфігурація мережі включає такі операції, як створення підмереж, додавання мережевих політик і налаштування DNS для забезпечення безперебійної роботи мережі кластера.

Останнім кроком є встановлення таких програм, як моніторинг, журналювання тощо. Це дозволить контролювати та аналізувати продуктивність кластера та додатків.

Таким чином, ініціалізація Kubernetes є важливим процесом, який необхідно забезпечити

1.1.1 Конфігураційні файли Kubernetes

У Kubernetes є два основних типи файлів конфігурації: секрети ("secrets") і файли конфігурації ("configMap").

Секрети — це невеликі об'єкти, які містять конфіденційну інформацію, наприклад облікові дані або ключі доступу. Вони зберігаються в etcd у вигляді звичайного тексту, доступні для сервера Kubernetes API, і можуть бути змонтовані як файли до підсистем, які потребують доступу до них (за допомогою спеціальних секретних томів, перетворених на звичайні томи даних). Той самий секрет можна вставити в кілька контейнерів. Kubernetes сам створює секрети для своїх компонентів, і ви можете створити свої власні. Слід зазначити, що всі значення в секретах кодуються і зберігаються у форматі base64 (рис. 1.1).

Однак секрети Kubernetes не мають додаткового рівня захисту: кожен, хто має доступ до Kubernetes, може отримати доступ до секрету. Ситуація така ж, коли вони змонтовані як томи - будь-хто, хто має доступ до файлової системи контейнера, може прочитати секрети. Секрет Kubernetes — це лише назва.

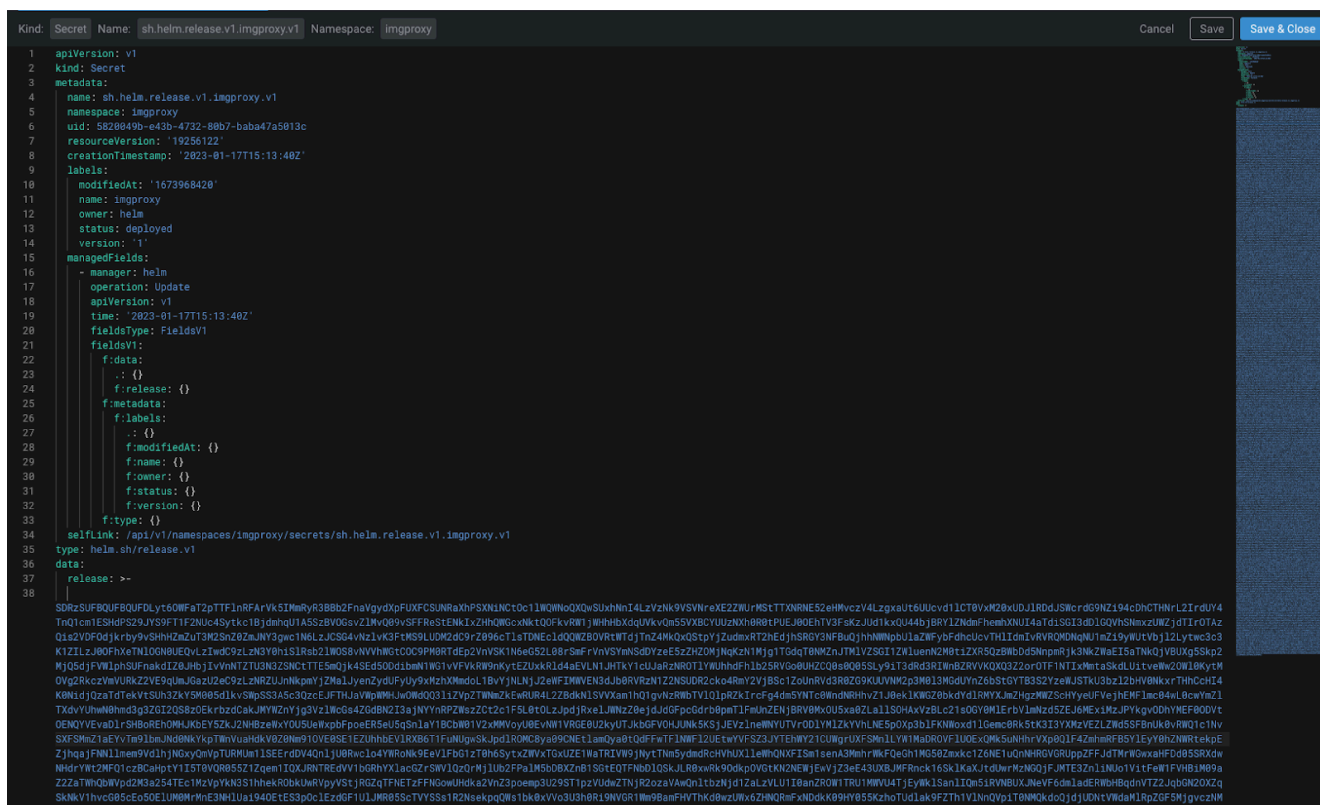
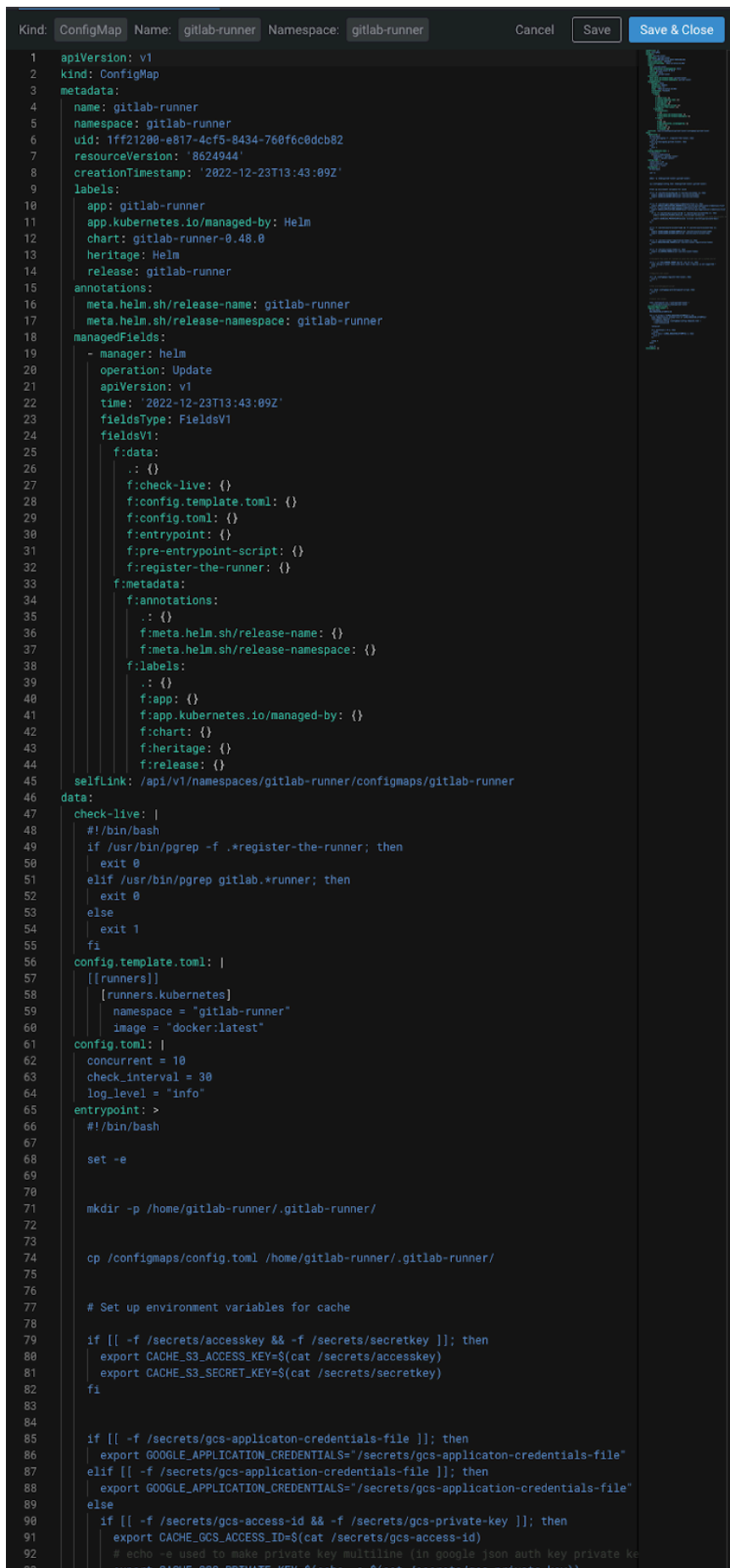


Рисунок 1.1 — Вигляд секрету.

Файли конфігурації схожі на секрети. Таким же чином їх можна створювати та використовувати в контейнерах. Єдиною відмінністю між ними є відсутність кодування base64 (рис. 1.2). Профілі призначені для неконфіденційних даних — даних конфігурації, таких як файли конфігурації та змінні середовища — і є чудовим способом змінити конфігурацію контейнера без створення нового зображення, наприклад, за допомогою загальнодоступного зображення.



```

Kind: ConfigMap Name: gitlab-runner Namespace: gitlab-runner Cancel Save Save & Close

1 apiVersion: v1
2 kind: ConfigMap
3 metadata:
4   name: gitlab-runner
5   namespace: gitlab-runner
6   uid: 1ff21200-e817-4cf5-8434-760f6c0dc82
7   resourceVersion: '8624944'
8   creationTimestamp: '2022-12-23T13:43:09Z'
9   labels:
10    app: gitlab-runner
11    app.kubernetes.io/managed-by: Helm
12    chart: gitlab-runner-0.48.0
13    heritage: Helm
14    release: gitlab-runner
15   annotations:
16    meta.helm.sh/release-name: gitlab-runner
17    meta.helm.sh/release-namespace: gitlab-runner
18   managedFields:
19    - manager: helm
20      operation: Update
21      apiVersion: v1
22      time: '2022-12-23T13:43:09Z'
23      fieldsType: FieldsV1
24      fieldsV1:
25        f:data:
26          .: {}
27          f:check-live: {}
28          f:config.template.toml: {}
29          f:config.toml: {}
30          f:entrypoint: {}
31          f:pre-entrypoint-script: {}
32          f:register-the-runner: {}
33          f:metadata:
34            f:annotations:
35              .: {}
36              f:meta.helm.sh/release-name: {}
37              f:meta.helm.sh/release-namespace: {}
38            f:labels:
39              .: {}
40              f:app: {}
41              f:app.kubernetes.io/managed-by: {}
42              f:chart: {}
43              f:heritage: {}
44              f:release: {}
45      selfLink: /api/v1/namespaces/gitlab-runner/configmaps/gitlab-runner
46   data:
47     check-live: |
48       #!/bin/bash
49       if /usr/bin/pgrep -f .*register-the-runner; then
50         exit 0
51       elif /usr/bin/pgrep gitlab.*runner; then
52         exit 0
53       else
54         exit 1
55       fi
56     config.template.toml: |
57       [[runners]]
58       [runners.kubernetes]
59       namespace = "gitlab-runner"
60       image = "docker:latest"
61     config.toml: |
62       concurrent = 10
63       check_interval = 30
64       log_level = "info"
65     entrypoint: >
66       #!/bin/bash
67
68       set -e
69
70
71       mkdir -p /home/gitlab-runner/.gitlab-runner/
72
73
74       cp /configmaps/config.toml /home/gitlab-runner/.gitlab-runner/
75
76
77       # Set up environment variables for cache
78
79       if [[ -f /secrets/accesskey && -f /secrets/secretkey ]]; then
80         export CACHE_S3_ACCESS_KEY=$(cat /secrets/accesskey)
81         export CACHE_S3_SECRET_KEY=$(cat /secrets/secretkey)
82       fi
83
84
85       if [[ -f /secrets/gcs-application-credentials-file ]]; then
86         export GOOGLE_APPLICATION_CREDENTIALS="/secrets/gcs-application-credentials-file"
87       elif [[ -f /secrets/gcs-application-credentials-file ]]; then
88         export GOOGLE_APPLICATION_CREDENTIALS="/secrets/gcs-application-credentials-file"
89       else
90         if [[ -f /secrets/gcs-access-id && -f /secrets/gcs-private-key ]]; then
91           export CACHE_GCS_ACCESS_ID=$(cat /secrets/gcs-access-id)
92           # echo -e used to make private key multiline (in google json auth key private key
93           export CACHE_GCS_PRIVATE_KEY=$(echo -e $(cat /secrets/gcs-private-key))

```

Рисунок 1.2 — Вигляд файлу конфігурацій.

При такому способі налаштування самі параметри зберігаються в певному кластері (в etcd). Переміщення програм між кластерами викликає чимало проблем, оскільки немає зручного способу оптимально розвантажити дані з etcd і перемістити їх у сторонній кластер. Крім того, у разі збою кластера неможливо просто перезапустити робоче навантаження на резервному кластері.

Цей метод конфігурації можна використовувати в досить обмежених ситуаціях.

До переваг цього рішення можна віднести:

- є стандартним компонентом Kubernetes, тому немає додаткових витрат;
- можливість встановлювати секрети як змінні середовища та як окремі файли, доступні у файловій системі;
- простий у використанні;
- у свою чергу, серед недоліків можна виділити наступні;
- об'єкти типу Secret зберігаються в незахищеному вигляді;
- неможливість передачі секретів між кластерами;
- якщо є проблема з кластером, конфігураційні дані також будуть втрачені;
- відсутність зручного способу внесення змін (тільки через стандартну утиліту командного рядка kubectl);
- відсутність аудитів доступу;
- додаткове навантаження на компоненти системи Kubernetes.

Тому стандартний спосіб надання додатків від Kubernetes не є оптимальним рішенням, враховуючи наведені вище переваги та недоліки цього рішення. Його можливості можуть бути придатними лише для невеликих систем, які не відповідають певним стандартам безпеки та надійності.

1.1.2 Контроллер змін та конфігурацій

Kubernetes — це система оркестровки контейнерів, яка дозволяє розгортати програми в контейнерах і керувати ними. Щоб забезпечити високу доступність програм у кластері Kubernetes, ви можете використовувати контролери змін і

конфігурації. У цій доповіді ми розглянемо контролер змін і конфігурації в Kubernetes і наведемо приклади його використання.

Змінено та налаштовано контролери в Kubernetes

Контролер змін і конфігурації — це компонент Kubernetes, який дозволяє керувати розгортанням і оновленнями програм у кластері Kubernetes. Контролери змін і конфігурації використовуються для забезпечення високої доступності та автоматичного масштабування програм. Зміна та налаштування контролерів включає кілька різних типів контролерів, наприклад:

- контролер реплікації;
- контролер набору копій (контролер ReplicaSet);
- контролер розгортання;
- контролер DaemonSet.

Кожен із цих контролерів має власну функціональність і використовується в різних типах програм.

Контролер реплікації (Replication Controller) використовується для забезпечення доступності певної кількості реплік (примірників) програми в кластері Kubernetes. Контролер реплікації відстежує кількість реплік додатків і автоматично розгортає нові, коли кількість зменшується або не існує. Контролери реплікації дозволяють забезпечити високу доступність програм у кластерах Kubernetes.

Контролер змін і конфігурації в Kubernetes дозволяє автоматично оновлювати програми у вашому кластері Kubernetes зі змінами конфігурації. Це дає змогу змінювати параметри програми без необхідності переробляти та розгортати нову версію програми.

Зміна та налаштування основних компонентів контролера:

- Deployment, це ресурс Kubernetes, який дозволяє вам керувати розгортанням вашої програми в кластері Kubernetes;
- ConfigMap, це ресурс Kubernetes, який дозволяє зберігати конфігурацію програми в кластері Kubernetes;
- секрет, це ресурс Kubernetes, який дозволяє зберігати конфіденційну інформацію, таку як паролі та ключі доступу, у кластері Kubernetes;

- сервіс, це ресурс Kubernetes, який дозволяє вам керувати доступом до програм у кластері Kubernetes.

1.2 Ініціалізація контейнерів

Kubernetes — це надзвичайно гнучке та потужне рішення для створення складних систем. Це особливо необхідно для платформ, побудованих з використанням архітектури мікросервісів. Однак питання оркестрування та ефективного управління робочим навантаженням є беззаперечним викликом. Для всієї цієї гнучкої архітектури потрібно певним чином керувати версіями, і слід контролювати загальний стан системи. На жаль, це завдання неможливо вирішити за допомогою стандартних засобів Kubernetes.

Ось чому був створений проект з відкритим кодом для керування конфігурацією Kubernetes кластерів Helm.

Helm є аналогом менеджерів пакунків, які зазвичай використовуються різними операційними системами, але Helm здатний керувати версіями та контролювати ресурси Kubernetes, а не системні пакунки. Крім того, у своїй основі Helm використовує досить потужну бібліотеку шаблонів для мови програмування Go. Таким чином, однією з підтримуваних функцій є візуалізація шаблонів, яка дозволяє використовувати шаблони.

Helm підтримує кілька важливих випадків використання:

- управління складними абстракціями;
- легке оновлення та контроль версій;
- повторне використання кодової бази;
- безпечне відновлення старішої версії програми (відкат), якщо щось піде не так з новою версією.

Цей диспетчер робочого навантаження дозволяє об'єднати всі ресурси, необхідні для запуску вашої програми, в єдиний блок під назвою «граф». Таким чином, кожна програма, що працює в кластері Kubernetes, має власний графік, який може складатися з набору шаблонів, файлів зі списками параметрів для відтворення шаблонів та інших системних програм.

Графіки можуть описувати навіть найскладніші програми, забезпечуючи повторне розгортання програм і слугуючи єдиним джерелом для розгортання великої кількості різноманітних робочих навантажень. Швидкі оновлення та спеціальні маркери спрощують розгортання нових версій програм. Дуже легко ділитися діаграмами, які можна контролювати версіями та розміщувати на публічних або приватних серверах. Коли вам потрібно відкотити нещодавнє оновлення, Helm надає команду для відкоту цілого набору змін, внесених у вашу інфраструктуру.

Варто відзначити, що кодова база публічного сервера з додатком досить обширна. При необхідності ви можете знайти загальнодоступні діаграми практично для будь-якого публічного програмного забезпечення. Однак слід бути обережним, тому що у випадку публічного реєстру Docker ніхто не контролює якість розміщеного коду та наявність шкідливих програм.

Будь-який компонент системи можна відключити. Тому варто віддати перевагу офіційним діаграмам, оскільки їх використання пов'язане з небезпекою на найнижчій межі

HELM — це популярний менеджер пакунків Kubernetes, який спрощує встановлення контейнерних програм і керування ними. Діаграми HELM — це пакети HELM, які містять усі компоненти, необхідні для розгортання та керування програмами в Kubernetes. У цьому звіті ми розглянемо, скільки існує діаграм HELM, де їх знайти та чому деякі з них можуть не працювати.

Скільки існує діаграм HELM?

Кількість існуючих діаграм HELM велика і постійно зростає. В Інтернеті є багато веб-сайтів і сховищ, де можна знайти різноманітні діаграми HELM. Одним із найвідоміших сховищ діаграм HELM є Helm Hub, де можна знайти понад 5000 готових діаграм HELM для різних програм і сервісів.

Де я можу знайти діаграми HELM?

Окрім Helm Hub, ви можете знайти діаграми HELM у багатьох різних місцях. Наприклад, GitHub має ряд сховищ з відкритим кодом, які містять діаграми HELM для різних програм і служб. Деякі розробники та компанії також розміщують власні діаграми HELM на своїх веб-сайтах або репозиторіях.

Чому деякі діаграми HELM можуть не працювати?

Хоча діаграми HELM дозволяють швидко розгортати програми та служби в кластерах Kubernetes, деякі з них можуть не працювати належним чином. Це може бути викликано різними причинами, такими як неправильні файли конфігурації, неправильно налаштовані залежності, неправильна сама діаграма або несумісність із версією

1.2.1 Мануальна ініціалізація

Контейнери роблять простим і ефективним розгортання програм і керування ними в кластерах Kubernetes. У цій доповіді ми розглянемо, як запускати контейнери на Kubernetes.

Існують різні способи запуску контейнерів у Kubernetes. Ми розглянемо два найпоширеніші підходи — використання файлу конфігурації розгортання та використання командного рядка `kubectl`.

Використовуйте профіль розгортання

Розгортання є одним із основних ресурсів Kubernetes, який дозволяє розгортати програми та керувати ними в кластерах Kubernetes. Щоб використовувати розгортання для запуску контейнерів у Kubernetes, вам потрібно створити файл конфігурації для розгортання, який містить налаштування для запуску контейнерів.

Нижче наведено приклад файлу конфігурації для розгортання (рис. 1.3) :

```

Kind: Deployment Name: be-test Namespace: test-be
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: my-app-deployment
5  spec:
6    replicas: 3
7    selector:
8      matchLabels:
9        app: my-app
10   template:
11     metadata:
12       labels:
13         app: my-app
14     spec:
15       containers:
16       - name: my-app-container
17         image: my-app-image:latest
18         ports:
19         - containerPort: 8080

```

Рисунок 1.3 — Файлу конфігурації для мануального розгортання.

У цьому прикладі ми створюємо розгортання під назвою "my-app-deployment". Розгортання має 3 репліки, що означає, що Kubernetes запускатиме три репліки контейнера. Крім того, ми вказуємо, що контейнер повинен мати мітку «app: my-app». У розділі Шаблони вказуємо параметри для кожної копії контейнера. У цьому прикладі ми вказуємо, що контейнер повинен мати назву «my-app-container» і використовувати зображення «my-app-image:latest». Ми також вказали, що контейнер повинен слухати порт 8080.

Щоб запустити контейнер у Kubernetes за допомогою цього конфігураційного файлу, виконайте таку команду:

kubectl apply -f deployment-config.yaml

де "deployment-config.yaml" - це ім'я файлу конфігурації

1.2.2 Автоматична ініціалізація

Інструменти автоматизації, такі як фреймворки безперервної інтеграції/безперервного розгортання (CI/CD) або інші засоби автоматизації, використовуються для автоматичного запуску контейнерів у Kubernetes без втручання людини.

Найпоширенішим способом автоматизації запуску контейнера в Kubernetes є використання платформи CI/CD або оператора Kubernetes.

Використання платформи CI/CD

Фреймворки CI/CD, такі як Jenkins, GitLab CI/CD, CircleCI та Travis CI, дозволяють автоматизувати процес розгортання програм у Kubernetes. Ці платформи часто використовуються з контейнерами Docker і допомагають автоматизувати створення та розгортання контейнерів у кластерах Kubernetes.

Наприклад, щоб використовувати Jenkins для автоматичного розгортання контейнера в Kubernetes, вам потрібно створити Jenkinsfile, щоб визначити кроки для створення та розгортання контейнера.

Приклад Jenkinsfile для автоматизації розгортання контейнера в Kubernetes зображено на рисунку 1.4.

```

1 pipeline {
2     agent any
3
4     stages {
5         stage('Build') {
6             steps {
7                 sh 'docker build -t my-app-image .'
8             }
9         }
10
11        stage('Deploy') {
12            steps {
13                script {
14                    def app = 'my-app'
15                    def env = 'production'
16                    def image = "my-registry/my-app:${env}-${env.BUILD_NUMBER}"
17
18                    sh "docker tag my-app-image ${image}"
19                    sh "docker push ${image}"
20                    sh "kubectl set image deployment/${app} ${app}=${image}"
21                }
22            }
23        }
24    }
25 }

```

Рисунок 1.4 — Приклад Jenkinsfile для автоматичного розгортання контейнера у Kubernetes.

У цьому прикладі ми використовуємо Jenkins для створення контейнера та розгортання його в Kubernetes. У розділі «build» ми запускаємо команду «docker build», щоб створити контейнер під назвою «my-app-image». У частині розгортання ми використовуємо Kubernetes для розгортання контейнерів у кластері. Ми вказуємо назву програми ("my-app"), назву середовища ("production") і мітку контейнера для використання ("production-BUILD_NUMBER"). Після створення контейнера та розміщення його в реєстрі Docker запускаємо

1.3 Управління контейнерами

Керування контейнерами в Kubernetes

Kubernetes пропонує кілька способів керування контейнерами, що полегшує розгортання, масштабування та оновлення програм. Нижче наведено деякі ключові аспекти керування контейнерами в Kubernetes.

Розгортання контейнера

Kubernetes дозволяє розгорнути контейнери за допомогою маніфестів YAML або JSON, які описують ресурси, необхідні програмі. Ці маніфести можна створити та застосувати за допомогою команди *kubectl apply -f<file-name>* .

Ваги контейнер

Kubernetes масштабується горизонтально через реплікацію модулів. Це можна зробити автоматично за допомогою контролерів ReplicaSet або Deployment. Щоб змінити кількість реплік, ви можете скористатися командою *kubectl scale*.

Подавати

Сервіси в Kubernetes дозволяють створювати стабільні мережеві адреси, до яких модулі мають доступ. Це полегшує використання мікросервісів і спілкування між компонентами програми.

Оновлення та відкат програми

За допомогою ресурсу розгортання Kubernetes дозволяє оновлювати контейнери без зупинки програми. Kubernetes застосовує стратегію «Поступове оновлення» для поступової заміни старих модулів новими, щоб забезпечити безперервність додатків. Якщо щось піде не так, Kubernetes дозволяє відкотити оновлення до попередньої версії.

Самовідновлення

Kubernetes стежить за справністю контейнерів і вузлів. Якщо контейнер виходить з ладу, Kubernetes автоматично перезапускає його. У разі збою вузла Kubernetes перемістить контейнери на інші вузли.

1.4 Ініціалізація сервісів

За допомогою інтерфейсу командного рядка (CLI) Amazon Web Services (AWS) ви можете ініціалізувати та створювати такі служби AWS, як екземпляри EC2, сегменти S3, екземпляри RDS, функції Lambda тощо. У цьому виступі детально описано принципи та команди для ініціалізації та створення сервісів AWS за допомогою CLI.

Перш ніж почати використовувати AWS CLI, вам потрібно встановити його на свій комп'ютер і налаштувати для взаємодії з AWS. Щоб інсталивати AWS CLI, потрібно завантажити виконуваний файл з офіційного сайту AWS. Потім вам потрібно розпакувати файли та виконати команду для встановлення.

```
curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"  
unzip awscliv2.zip  
sudo ./aws/install
```

Після встановлення AWS CLI вам потрібно налаштувати його для взаємодії з AWS. Для цього потрібно запустити команду `aws configure`, яка запитає ключ доступу та секретний ключ. Ви можете знайти ці ключі в Консолі керування AWS у розділі Облікові дані безпеки.

```
aws configure
```

Після введення ключа вам буде запропоновано вказати регіон і формат виводу за умовчанням. Ви можете вибрати регіон зі списку доступних регіонів AWS. Вихідний формат можна встановити будь-яким із доступних параметрів.

Ініціалізацію та створення служб AWS за допомогою CLI можна виконати за допомогою таких команд, як `aws ec2`, `aws s3`, `aws rds`, `aws lambda` тощо. Для кожного сервісу є спеціальні команди, які дозволяють створювати, змінювати та видаляти ресурси.

EC2 інстанси

Amazon Elastic Compute Cloud (EC2) — це сервіс AWS, який надає можливість запускати віртуальні сервери в ресурсах хмарних обчислень. Щоб створити екземпляр EC2 за допомогою CLI, виконайте таку команду:

```
aws ec2 run-instances --image-id ami-0c55b159cbfafelf0 --count 1 --instance-type t2.micro --key-name my-key-pair --security-group-ids sg-0123456789abcdef0 --subnet-id subnet-0123456789abcdef0
```

У цьому прикладі ми використовуємо команду `aws ec2 run-instances` для створення нового екземпляра EC2. Ми вказуємо AMI (образ машини Amazon), кількість примірників, тип примірника, назву ключа доступу, ідентифікатор групи безпеки та ідентифікатор підсистеми мережі.

S3 бакети

Amazon Simple Storage Service (S3) — це служба AWS, яка надає можливість зберігати та відновлювати дані в хмарі. Щоб створити сегмент S3 за допомогою CLI, потрібно виконати таку команду:

```
aws s3api create-bucket --bucket my-bucket --region us-west-2 --create-bucket-configuration LocationConstraint=us-west-2
```

У цьому прикладі ми використовуємо команду `aws s3api create-bucket` для створення нового сегмента S3. Ми вказуємо назву сегмента, регіон за замовчуванням і конфігурацію для створення контейнера.

RDS інстанси

Amazon Relational Database Service (RDS) — це служба AWS, яка надає можливість створювати та керувати базами даних у хмарі. Щоб створити примірник RDS за допомогою CLI, виконайте таку команду:

```
aws rds create-db-instance --db-instance-identifier my-db-instance --db-instance-class db.t2.micro --engine mysql --master-username myuser --master-user-password mypassword --allocated-storage 20 --vpc-security-group-ids sg-0123456789abcdef0 --availability-zone us-west-2a
```

У цьому прикладі ми використовуємо команду `aws rds create-db-instance` для створення нового екземпляра RDS. Ми вказуємо ідентифікатор екземпляра, тип екземпляра, тип бази даних, ім'я користувача та пароль, обсяг пам'яті, ідентифікатор групи безпеки та зону доступності.

Lambda функції

AWS Lambda — це служба AWS, яка надає можливість запускати код без віртуальних серверів. Щоб створити функцію Lambda за допомогою CLI, виконайте таку команду:

```
aws lambda create-function --function-name my-function --runtime nodejs14.x --  
handler index.handler --zip-file fileb://my-function.zip --role  
arn:aws:iam::123456789012:role/my-role
```

У цьому прикладі ми використовуємо команду `aws lambda create-function` для створення нової функції Lambda. Ми вказуємо назву функції, версію Node.js, ім'я обробника, zip-файл, що містить код функції, і роль, яку має виконувати функція.

Elastic Beanstalk

Amazon Elastic Beanstalk — це служба AWS, яка надає можливість розгортати та масштабувати веб-додатки та служби. Щоб створити Elastic Beanstalk за допомогою CLI, виконайте таку команду:

```
aws elasticbeanstalk create-application --application-name my-application --  
description "My Application" --region us-west-2
```

У цьому прикладі ми використовуємо команду `aws elasticbeanstalk create-application` для створення нового Elastic Beanstalk. За замовчуванням ми вказуємо назву програми, опис і регіон.

CloudFormation

AWS CloudFormation — це сервіс AWS, який дозволяє автоматизувати процес створення та керування ресурсами AWS. Щоб створити CloudFormation за допомогою CLI, виконайте таку команду:

```
aws cloudformation create-stack --stack-name my-stack --template-body file://my-  
template.json --parameters ParameterKey=KeyPairName,ParameterValue=my-key-  
pair --capabilities CAPABILITY_IAM
```

У цьому прикладі ми використовуємо команду `aws cloudformation create-stack` для створення нового CloudFormation. Вказуємо назву стеку, файл шаблону та параметри створення стеку.

У цій доповіді ми представили принципи та команди для створення різних сервісів AWS за допомогою CLI. AWS CLI дозволяє ініціалізувати різні служби AWS і керувати ними, виконуючи команди з терміналу. Це дозволяє швидко та ефективно створювати, конфігурувати та керувати ресурсами AWS. Використовуючи CLI, ви можете виконувати такі операції, як створення, оновлення та видалення ресурсів, налаштування доступу до ресурсів, отримання інформації про стан та виконання інших завдань.

AWS CLI дозволяє створювати служби AWS без використання консолі керування AWS. Це дає змогу швидко й ефективно використовувати ресурси AWS і особливо корисно для автоматизації рутинних завдань, таких як створення, налаштування та видалення ресурсів.

Крім того, AWS CLI дозволяє розширити можливості консолі керування AWS і виконувати інші операції з ресурсами AWS, які не підтримуються консоллю керування. Це дозволяє користувачам AWS бути більш продуктивними та ефективними при використанні ресурсів AWS.

Загалом, AWS CLI — це потужний інструмент, який дозволяє створювати ресурси AWS і керувати ними без використання консолі керування AWS. Це дає змогу швидко й ефективно використовувати ресурси AWS і особливо корисно для автоматизації рутинних завдань, таких як створення, налаштування та видалення ресурсів. Для кожного сервісу AWS існують спеціальні команди, які дозволяють створювати, змінювати та видаляти ресурси. CLI дозволяє розширити можливості консолі керування AWS і виконувати додаткові операції з ресурсами AWS, які не підтримуються консоллю керування. Крім того, CLI особливо корисний для команд, які розробляють, тестують і випускають програмне забезпечення в середовищі AWS, оскільки вони можуть автоматизувати процес розгортання та керування ресурсами AWS як частину робочого процесу.

Користувачі AWS CLI можуть використовувати різні команди та параметри для керування різними службами AWS. Ці команди можна виконувати з терміналу, дозволяючи користувачам керувати ресурсами AWS з будь-якого місця. Для кожного сервісу AWS існують спеціальні команди, які дозволяють створювати, змінювати та видаляти ресурси.

AWS CLI дозволяє користувачам налаштовувати доступ до ресурсів AWS, установлюючи ключі доступу та конфігуруючи профілі користувачів. Ключі доступу — це пари ідентифікаторів і ключів, які дозволяють користувачам отримувати доступ до ресурсів AWS. Профіль користувача — це набір параметрів, які використовуються для підключення до AWS і виконання команд.

AWS CLI може бути корисним інструментом для автоматизації процесу створення, налаштування та керування ресурсами AWS. Це особливо корисно для команд, які розробляють і тестують програмне забезпечення в середовищі AWS. Використовуючи AWS CLI, ви можете автоматизувати рутинні завдання, звільняючи розробників витратити більше часу на розробку і тестування програмного забезпечення.

Загалом, AWS CLI дозволяє користувачам створювати ресурси AWS і керувати ними за допомогою команд з терміналу. Це забезпечує швидкий і ефективний спосіб використання ресурсів AWS, що особливо корисно для розробників, системних адміністраторів та інших користувачів AWS. Інструмент дозволяє використовувати команди та параметри для створення, налаштування та керування різними ресурсами AWS, включаючи EC2, S3, RDS, Lambda та Elastic Beanstalk.

AWS CLI може бути корисним інструментом для керування ресурсами AWS, оскільки він дозволяє користувачам автоматизувати процес створення, налаштування та керування ресурсами AWS. Це особливо корисно для команд, які розробляють, тестують і випускають програмне забезпечення в середовищі AWS, оскільки вони можуть автоматизувати процес розгортання ресурсів AWS і керування ними як частину робочого процесу.

Загалом, AWS CLI дозволяє користувачам керувати різними ресурсами AWS за допомогою команд і параметрів терміналу. Це робить роботу з ресурсами AWS швидкою та ефективною, і особливо корисно для розробників, системних адміністраторів та інших користувачів AWS. Використовуючи AWS CLI, ви можете створювати, налаштовувати та керувати різними службами AWS, що дозволяє ефективно використовувати ресурси AWS.

1.5 Висновки до розділу

Існуючі рішення для ініціалізації та керування контейнерами та зовнішніми службами в кластері Kubernetes мають багато переваг і недоліків. Схема ініціалізації є занадто простою і негнучкою, яка може задовольнити потреби лише простих систем і не має високих вимог до надійності даних і масштабованості. Серед багатьох рішень управління лише одна група програмного забезпечення намагається задовольнити потреби більшості користувачів. У цьому випадку програмне забезпечення виявилось комплексним, але складним у використанні та впровадженні.

З огляду на особливості схеми перегляду, розробка системи, яка поєднує в собі переваги існуючих рішень і усуває їх недоліки, є найкращим способом вирішення проблеми ініціалізації та управління контейнерами та зовнішніми службами в кластері Kubernetes

2 АНАЛІЗ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ ТА KUBERNETES SERVICE

Архітектура мікросервісу -це рішення, яке може оптимізувати та сприяти процесу розробки. В даний час Kubernetes - це найпопулярніша система домовленостей. Kubernetes - це контейнер з відкритим кодом для розгортання, масштабування та управління. Тривалий час він використовувався для задоволення внутрішніх потреб Google під назвою Borg.

Налаштування - це процес координації взаємодії декількох контейнерів. Звичайно, вам не доведеться влаштувати роботу -ніхто не забороняє створити контейнер, який розпочне всі необхідні процеси. Однак у цьому випадку гнучкості, масштабованості та безпеки не буде, оскільки процес роботи в контейнері не буде ізольованим і може впливати один на одного.

Домовленість дозволяє використовувати багато контейнерів для створення інформаційної системи. Кожен контейнер несе відповідальність лише за певне завдання, і спілкується через мережевий порт та загальний каталог. При необхідності кожен такий контейнер може бути замінений іншими контейнерами. Наприклад, якщо це необхідно, ви можете швидко перейти на іншу версію бази даних.

Сервіс - ще один абстрактний рівень контейнера. Як і контейнер, послуга матиме імена, основні зображення та опубліковані порти та рулони. На відміну від контейнера, послуга може встановлювати вимоги хоста, який можна запустити на ньому. Зазвичай послуга може бути збільшена безпосередньо під час створення, щоб вказати, скільки контейнерів потрібно почати.

Kubernetes - це складний продукт про конкурентні продукти. Його структура вимагає не лише самої системи, але й інших основних речей, таких як проектування операційних систем, тестування та архітектури додатків.

Ця домовленість вимагає досить складної системи конфігурації робочого навантаження. У можливій конфігурації є файли YAML, JSON та інших. Однак важко контролювати десятки, сотні, а іноді навіть тисячі файлів конфігурації.

Теза магістра спеціально використовується для створення системи для ініціалізації та управління веб -додатком для ініціалізації та управління Kubernetes. Завдання планування інфраструктури планування кластерів доводить її відповідні заходи.

У цьому розділі буде обговорено огляд методу в методі архітектури мікро-сервісу та коментарів архітектури Kubernetes. Система розробки багато в чому залежить від функції цього розташування, оскільки вона використовуватиме його об'єкт для роботи. Ось чому огляд архітектури Kubernetes важливий.

2.1 Мікросервісна архітектура

Мікросервісна архітектура - це підхід до розробки програмного забезпечення, який полягає у створенні складових додатків у вигляді набору незалежних сервісів. Кожен мікросервіс має власну область відповідальності, реалізує конкретну функціональність і спілкується з іншими мікросервісами через стандартні мережеві протоколи, такі як HTTP або gRPC.

Основні характеристики мікросервісної архітектури:

- 1) незалежність компонентів, кожен мікросервіс може розвиватися, тестуватися та розгортатися незалежно від інших, що забезпечує гнучкість і швидкість розробки;
- 2) модульність, мікросервіси розбивають великі та складні додатки на менші та простіші модулі, що спрощують розуміння, розробку та супроводження коду;
- 3) масштабованість, мікросервіси можуть масштабуватися незалежно один від одного, дозволяючи розподіляти ресурси лише на ті компоненти, яким вони потрібні;
- 4) стійкість до збоїв, через незалежність мікросервісів від інших компонентів, збій у одному сервісі менш ймовірно призведе до збою всього додатка;

- 5) технологічна гнучкість, мікросервіси можуть використовувати різні мови програмування, фреймворки та бази даних, що дозволяє використовувати найкращі технології для кожної конкретної задачі.

Мікросервісна архітектура підходить для великих, розподілених та високонавантажених додатків, де необхідна швидка реакція на зміни ринкових умов та бізнес-вимог. Однак, слід зауважити, що мікросервісна архітектура також має свої виклики та складності.

Виклики мікросервісної архітектури:

- 1) складність розподіленої системи, мікросервіси можуть додати складність до системи через потребу в управлінні взаємодією між різними сервісами та моніторингу їх стану;
- 2) мережеві затримки, спілкування між мікросервісами зазвичай залежить від мережі, що може призвести до затримок та збоїв;
- 3) керування даними, розподілені дані між різними мікросервісами можуть створити проблеми з управлінням та забезпеченням послідовності даних.
- 4) організаційні зміни, мікросервісна архітектура може вимагати змін в організаційній структурі та процесах розробки, оскільки кожен мікросервіс може мати власну команду розробників;
- 5) забезпечення безпеки, оскільки мікросервіси спілкуються через мережу, необхідно врахувати можливі ризики для безпеки та забезпечити захист даних та сервісів.

Щоб долати ці виклики, розробники можуть використовувати різні практики та інструменти, такі як контейнеризація, оркестрація контейнерів (наприклад, Kubernetes), моніторинг, протоколювання та застосування шаблонів та принципів, специфічних для мікросервісної архітектури, таких як:

- 1) API Gateway, централізований вхід до вашої мікросервісної архітектури, який маршрутизує запити до відповідних мікросервісів, забезпечуючи безпеку, моніторинг та управління доступом;

- 2) схема розподіленої трасування, допомагає слідкувати за запитами, що проходять через різні мікросервіси, полегшуючи знаходження проблем та аналізує роботу системи;
- 3) централізоване конфігурування, управління конфігурацією мікросервісів з одного місця, що полегшує зміни та оновлення конфігурацій;
- 4) Service Mesh, абстрагування комунікації між мікросервісами, що дозволяє забезпечити безпеку, стійкість та моніторинг без зміни коду самого мікросервісу;
- 5) обробка помилок та збоїв, використання шаблонів, таких як Circuit Breaker, Bulkhead та Timeout, для забезпечення стійкості системи під час збоїв або проблем з окремими мікросервісами.

Успішне застосування мікросервісної архітектури вимагає від розробників та команд високої дисципліни, хорошого розуміння принципів розподіленої системи та уміння працювати з відповідними інструментами. За належного підходу, мікросервісна архітектура може допомогти створити гнучкі, масштабовані та стійкі додатки, що забезпечують кращий користувацький досвід та відповідність сучасним бізнес-вимогам.

2.1.1 Аналіз підходу

Протягом тривалого часу програмне забезпечення було розроблено за допомогою одного методу. Цей метод визначає розробку багатofункціональних продуктів як єдиної програми (єдина бібліотека коду, система управління залежністю тощо). Історично цей метод завжди з'являвся, оскільки з самого початку програмні програми не такі великі, як зараз існуючі. Тому, починаючи з невеликого простого застосування, він може перетворитися на величезний проект. Використовуючи цей метод, всі системи системи включають програму. Ці додатки часто мають складні багато-шлемічні архітектури,

Використовуючи єдину (зазвичай перевантажену) базу даних, це незручно розробити та підтримувати.

З часом недоліки цього методу свідомо використовуються і почали використовувати протистояння загальної будівлі. Структура архітектури призводить до розкладання системи на незалежний компонент, і кожен компонент відповідає за окремі функції. Кожен компонент, як правило, є незалежним і незалежним програмним додатком із власною базою даних (якщо це необхідно), власною системою контролю залежності.

У галузі розробки та обслуговування програмного забезпечення поява мікросервісів є дуже важливим кроком. Використовуючи мікросервіси, платформа розробки розкладається на окремі компоненти. Ці компоненти взаємодіють між собою через інтерфейс-API.

Кожен мікро-сервіс є самостійним, має власний шар зберігання даних і може бути оновлений незалежно, як показано на рисунку 2.1.

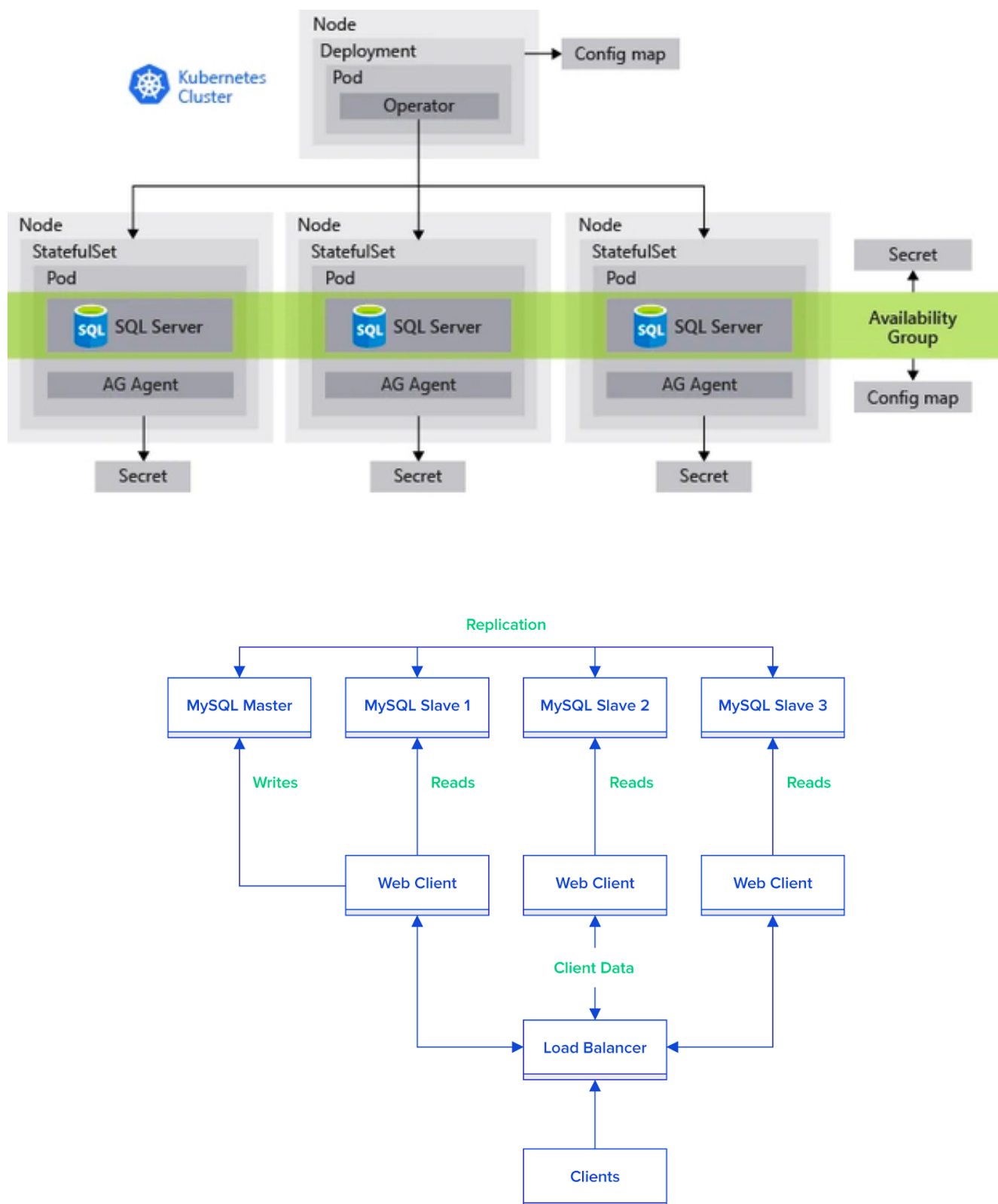


Рисунок 2.1 — Порівняння взаємодії загальних та мікросервісів.

Перехід до цього методу прискорив та спростив розробку програмного забезпечення комплексу, оскільки запуск необхідних змін у продукті.

Розробникам потрібно менше часу для введення програмного коду.

Крім того, такі програми простіше розмістити на декількох серверах. Такі функції іноді дуже корисні для задоволення зміни попиту на ринку. Наприклад, якщо функція, яка дає частину платформи, стала популярною, і багато людей почали її використовувати, то ви можете легко бути незбалансованими на навантаженнях на декількох серверах, що допоможе обробляти запити користувачів.

Елементи є основними факторами багатьох культурних та інженерних змін, які з'являються після популярних методів обслуговування. Метод розвитку еластичного програмного забезпечення передає власні дані на хмару, появу концепцій та впровадження непокіненого розвитку, використання контейнерів -це зовнішній вигляд та популярність архітектури мікро -сервісу розробки програмного забезпечення у світі

Окрім великої кількості вдосконалень у розробці програмного забезпечення, виникло багато проблем. Як показано нижче:

- збільшити складність платформи моніторингу мікросервісу;
- проблема послідовності API;
- скопіюйте проблему даних;
- поява єдиних проблемних точок (SPOF);
- проблема розгортання додатків;
- проблема ініціалізації великих масштабів додатків.

Це останні два питання, розроблені для вирішення системи, розробленої в дисертації магістра.

2.1.1 Взаємодія між мікросервісами

Взаємодія між мікро -сервісом базується на контракті. Контракт - це умова розробників послуг взаємодіє з ним

обслуговування клієнтів. Ці послуги можуть бути розроблені однією людиною або від команди іншого відділу чи інших компаній, як чорна скринька.

Послуга може надати будь -які об'єкти вмісту, але формат зв'язку між договором, щоб оголосити послугу, а не вміст. Внутрішній контент коригується лише на рівні логіки бізнесу. Тому договір вирішує повернутися до зовнішнього користувача та як отримати доступ до API.

Зі складністю цього завдання немає десятків мікро -послуг, але тисячі традиційних "класичних" контрактів більше не відповідають потребам. Тоді необхідно оновити договір, переглянувши структуру мікро -послуг. Складіть робочий план для них та їх розробників

"Кінцевий користувач" мікросервісів -це інформація про обмін -Цей метод називається контактом користувача.

Вважається, що користувачі можуть створювати договори, вказати та надсилати їх розробникам, які виконують свої послуги. Оскільки кількість користувачів у кожній службі дуже обмежена, необхідно отримати кілька специфікацій з кожної послуги, і найзручніше ввести їх, ніж намагатися вгадати формат відповіді, і який формат повинен бути наданий у кожному випадку .

Як правило, спілкування між командами з розробки платформ непросто, оскільки жодна команда не може брати участь у зустрічі, оскільки це може означати, що необхідно докласти додаткових зусиль при розробці продукції. Невідповідність договору спричинить фактичну збій системи та втрату довіри користувачів, або втручання шляху тестерів програмного забезпечення в новій версії нової версії деяких мікрокрекцій. Ось чому послідовність контракту між мікро -послугами є надзвичайно важливим.

2.1.3 Розгортання контейнерів у хмарах

Процес розгортання контейнерів у хмарних сервісах може мати деякі відмінності залежно від обраного хмарного провайдера. Однак, загальні кроки для

розгортання контейнерів у Docker та Kubernetes хмарних сервісах залишаються однаковими. Нижче описано типовий процес розгортання контейнерів в обох сервісах.

Створення Docker-образу

Напишіть Dockerfile, який містить інструкції щодо побудови образу вашого додатка. Збудуйте Docker-образ, використовуючи команду `docker build`.

Завантаження Docker-образу до реєстру контейнерів.

Завантажте образ до реєстру контейнерів, такого як Docker Hub, Google Container Registry або Amazon Elastic Container Registry, використовуючи команду `docker push`.

Розгортання контейнерів у Docker хмарному сервісі

Виберіть хмарний провайдер, що пропонує Docker-сумісний сервіс, такий як Amazon Web Services (AWS) Elastic Beanstalk, Google Cloud Run, Azure Container Instances або Heroku. За допомогою інструментів та веб-консолі вашого хмарного провайдера, створіть новий інстанс або службу, яка використовує ваш Docker-образ. Налаштуйте параметри розгортання, такі як мережеві налаштування, обмеження ресурсів та масштабування. Запустіть створену службу або інстанс, і ваш контейнер буде розгорнуто та запущено в хмарі.

Розгортання контейнерів у Kubernetes хмарному сервісі

Виберіть хмарний провайдер, що пропонує Kubernetes-сумісний сервіс, такий як Google Kubernetes Engine (GKE), Amazon Elastic Kubernetes Service (EKS), Microsoft Azure Kubernetes Service (AKS) або DigitalOcean Kubernetes. Встановіть та налаштуйте кластер Kubernetes, використовуючи інструменти та веб-консоль вашого хмарного провайдера. Встановіть та налаштуйте `kubectl`. `kubectl` - це інструмент командного рядка для управління кластером Kubernetes. Встановіть `kubectl` на вашому комп'ютері та налаштуйте його для роботи з вашим кластером Kubernetes. Створіть маніфест Kubernetes. Маніфест - це YAML-файл, який описує ресурси, необхідні для розгортання вашого додатка в Kubernetes.

Маніфест може містити опис Deployment, Service, ConfigMap та інших об'єктів Kubernetes. Вам слід вказати назву вашого Docker-образа та параметри розгортання в маніфесті. Застосуйте маніфест Kubernetes. Використовуючи kubectl, застосуйте маніфест до вашого кластера Kubernetes, виконавши команду `kubectl apply -f <your_manifest.yaml>`. Kubernetes створить відповідні об'єкти, розгорне контейнери та налаштує мережу. Моніторинг та управління. Використовуйте веб-консоль хмарного провайдера, kubectl та інші інструменти для моніторингу стану вашого розгорнутого додатка, масштабування, оновлення або вирішення проблем.

Незалежно від того, чи використовуєте ви Docker чи Kubernetes хмарний сервіс, процес розгортання контейнерів в хмарі забезпечує гнучкість, масштабованість та стійкість, які відповідають сучасним вимогам розробки програмного забезпечення.

Розгортання додаткових сервісів. В залежності від вашої архітектури та вимог до додатка, можливо, вам доведеться розгорнути додаткові сервіси, такі як бази даних, кеші, черги повідомлень та інші компоненти. Ви можете використовувати хмарні сервіси або розгорнути власні контейнери, які виконують ці функції.

Налаштування безпеки. Переконайтеся, що ваші контейнери та хмарні сервіси налаштовані відповідно до кращих практик безпеки. Це може включати налаштування мережевих політик, управління доступом, оновлення ПЗ, аудит та моніторинг.

Встановлення та інтеграція з інструментами DevOps. Інтегруйте ваш процес розгортання контейнерів з іншими інструментами DevOps, такими як системи контролю версій, системи неперервної інтеграції/неперервного розгортання (CI/CD), системи спостереження та логування.

Оптимізація витрат. Відслідковуйте використання ресурсів та витрати на ваші контейнери та хмарні сервіси. Використовуйте стратегії масштабування,

резервування ресурсів, спот-інстансів та інші опції, щоб оптимізувати витрати на розгортання.

Успішне розгортання контейнерів у хмарних сервісах вимагає координації різних елементів інфраструктури, налаштування параметрів та роботи з хмарними провайдерами. За дотриманням найкращих практик та правильної налаштуванки, розгортання контейнерів у хмарі може поліпшити швидкість розробки, забезпечити стабільність, еластичність та знизити загальні витрати на інфраструктуру. Ось деякі додаткові аспекти, які слід враховувати:

Резервне копіювання та відновлення: Плануйте регулярне резервне копіювання ваших даних та конфігурацій, які використовуються у контейнерах та хмарних сервісах. Розробіть стратегію відновлення в разі відмови або загублення даних.

Управління конфігурацією: Використовуйте інструменти для управління конфігурацією, як-от Kubernetes ConfigMaps або секрети для зберігання та впровадження конфігураційних даних та чутливої інформації безпосередньо у контейнери.

Забезпечення високої доступності: Розгляньте розгортання критичних компонентів додатка на різних зонах доступності або регіонах для забезпечення високої доступності та стійкості до відмов.

Розгортання та управління сертифікатами TLS: Забезпечте безпечне з'єднання та передачу даних між компонентами вашого додатка, використовуючи сертифікати TLS. Використовуйте хмарні сервіси або Kubernetes сертифікат менеджери для розгортання та управління сертифікатами.

Аудит та відповідність: Врахуйте вимоги аудиту та відповідності до регулятивних стандартів для вашого додатка та інфраструктури. Використовуйте інструменти та сервіси для моніторингу, аудиту та забезпечення відповідності до стандартів, таких як GDPR, HIPAA, PCI DSS тощо.

Враховуючи ці аспекти при розгортанні контейнерів у хмарних сервісах, ви можете забезпечити ефективне, безпечне та стабільне середовище для вашого додатка. Ось деякі додаткові поради для покращення вашого досвіду з розгортання контейнерів.

Оптимізація розміру образів Docker

Створюйте легкі та оптимізовані Docker-образи, використовуючи багатоступінчасті збірки та обираючи легкі базові образи, щоб зменшити використання ресурсів та час завантаження.

Використання мікросервісної архітектури

Розгляньте розбиття вашого додатка на менші, незалежні мікросервіси, які можна розгортати та масштабувати незалежно. Це поліпшить стабільність, швидкість розробки та забезпечить більшу гнучкість при масштабуванні.

Використання стратегій оновлення

Використовуйте стратегії оновлення, такі як рулонне оновлення або синій/зелений розгортання, для плавного впровадження змін у вашому додатку без зупинки роботи.

Тестування

Переконайтеся, що ваші контейнери проходять ретельне тестування, включаючи єдині тести, інтеграційні тести та навантажувальні тести, перш ніж вони будуть розгорнуті у продакшн.

Автоматизація

Автоматизуйте процеси розгортання, моніторингу та управління контейнерами, використовуючи скрипти, CI/CD та інші інструменти DevOps, щоб зменшити витрати на рутинну роботу та забезпечити швидке розгортання та відновлення.

Оцінювання та навчання

Регулярно оцінюйте та адаптуйте свій підхід до розгортання контейнерів, слідкуючи за новими технологіями.

2.1.4 Пакування контейнеру за допомогою технології Docker

Віртуалізація - це компонент і досить основна технологія, яка створює сучасну тріску, особливо хмару. Останнім часом такі технології, як

контейнерзація, стали все більш популярними, і вона була інтегрована у список хмарних постачальників. Він широко використовувався останнім часом.

Насправді, близька інтеграція контейнера до ядра ОС може зменшити накладні витрати програмного забезпечення, що необхідно для забезпечення роботи віртуальної машини.

Однак ця щільна інтеграція призвела до збільшення потенційних атак, що привернуло увагу людей до безпеки. Існуючий механізм безпеки в основному фокусується на встановленні взаємозв'язку безпеки між системою хоста та контейнером. Зокрема, контейнерна система була побудована в автоматизованому логічному ланцюзі розгортання, яка має на меті прискорити розгортання коду. Ці мережеві магазини, як правило, складаються з третім -партійних елементів. Ці елементи працюють на різних платформах різних постачальників. Це звертає увагу на цілісність та безпеку Фонду коду. Це може призвести до використання багатьох лазівки окупантами.

Docker - це програмна платформа, яка дозволяє швидко створювати, тестувати та розширювати програми. Docker надає програмне забезпечення в стандартизованих компонентах. Ці програмні засоби називаються контейнерами, що містить весь вміст вашого програмного забезпечення для запуску, включаючи бібліотеку, системні інструменти, код та робоче середовище. Docker може швидко розгорнути та розширювати програми майже в будь -якому середовищі. Порівняно зі звичайною віртуальною машиною, Docker швидше розподіляє обчислювальні ресурси з традиційними віртуальними машинами з різними типами (Xen, KVM та іншими типами віртуалізованих машин) з різними типами (Xen, KVM та іншими типами віртуалізованих машин) на рисунок 2.2.

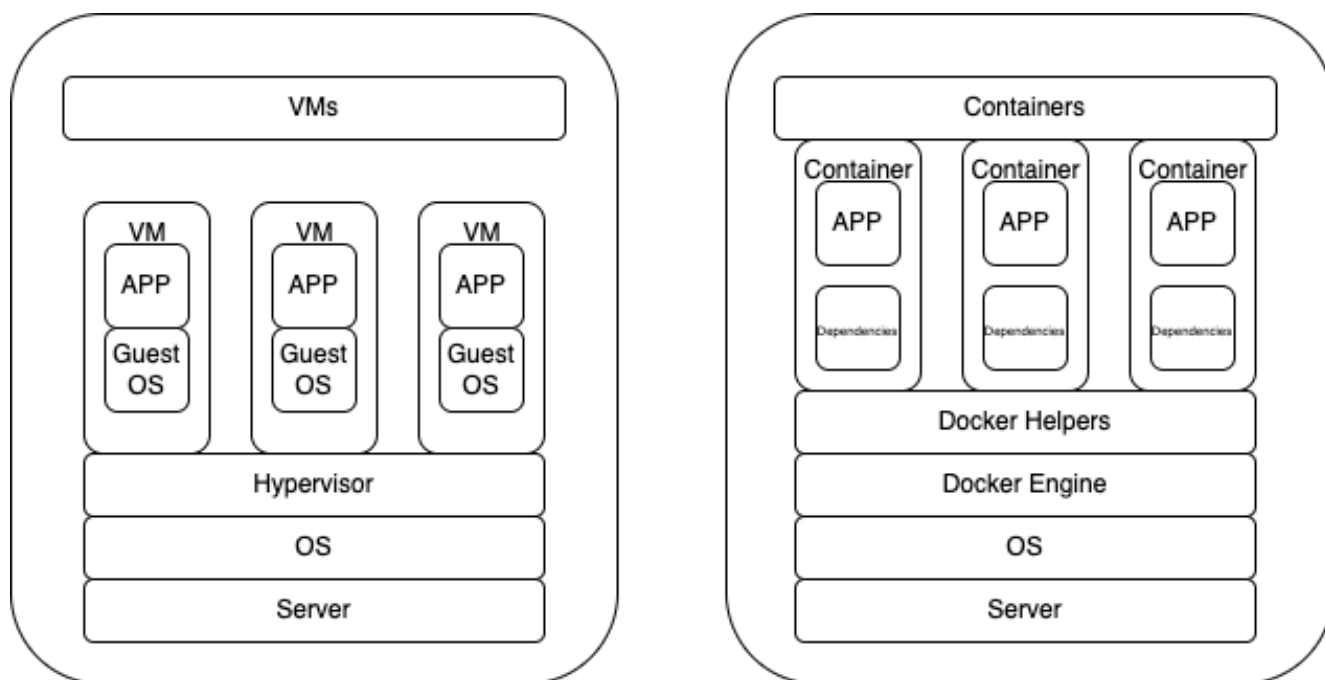


Рисунок 2.2 — Порівняння віртуалізації та контейнеризації.

Однак Docker - це не лише інструмент для створення та запуску контейнерів. Це багато компонентів, які утворюють екосистему навколо програмного продукту.

Програмне забезпечення складається з таких компонентів.

- Docker Demon-Server;
- Docker Demon-Server із запиту на обробку контейнерів;
- Demon-Control;
- Demon-Control об'єкта екосистеми і може спілкуватися з іншими дияволами для управління своїми компонентами;
- Docker-Client;
- Docker-Client взаємодіє з інтерактивним інтерфейсом Docker Demon. клієнти можуть спілкуватися з кількома дияволами, які можна встановити на декількох різних машинах.

Файл Image

Файл Image, який містить залежність програмного забезпечення для запуску програмного забезпечення, як подальше розгортання конфігурації програмного

забезпечення програмного забезпечення та даних, які повинні бути ініціалізовані контейнером.

Контейнер

Контейнер - це програмний пакет, створений певною інструкцією із зображення. Він включає всі необхідні залежності, код програмного забезпечення та інші компоненти, які можуть знадобитися для запуску програмного забезпечення. Життєвий цикл додатків контейнерів показаний на рисунку 2.3.

Том

Том - це симуляція файлової системи, яка виконує читання та запуск -up під час виконання програмного забезпечення. Об'єм буде створений автоматично при запуску нового контейнера, але він може бути створений та встановлений поодиночці на контейнері, щоб переконатися, що дані зберігаються між контейнерами.

Реєстр

Реєстр - це сервер для контейнерів для зберігання. Це необхідно, оскільки зображення, як правило, є результатом успішного тестування програмного коду та багатьох автоматичних перевірок. Такі перевірки проводяться на віддалених господарях, тісно пов'язаних з версіями з версіями, Gitlab, Github та іншими версіями.

Докер успішно став лідером ринку контейнерів та пов'язаних з ними ринків екосистем DevOps. Зокрема, у контейнерному середовищі 92 % слідчих використовують або планують використовувати Docker. Безпека - надзвичайно важливий аспект будь -якого аналізу технологій спектру. Через це команда розробників Docker проводить дослідження, що дозволяє спробувати вивчити фактичну можливість певних атак.

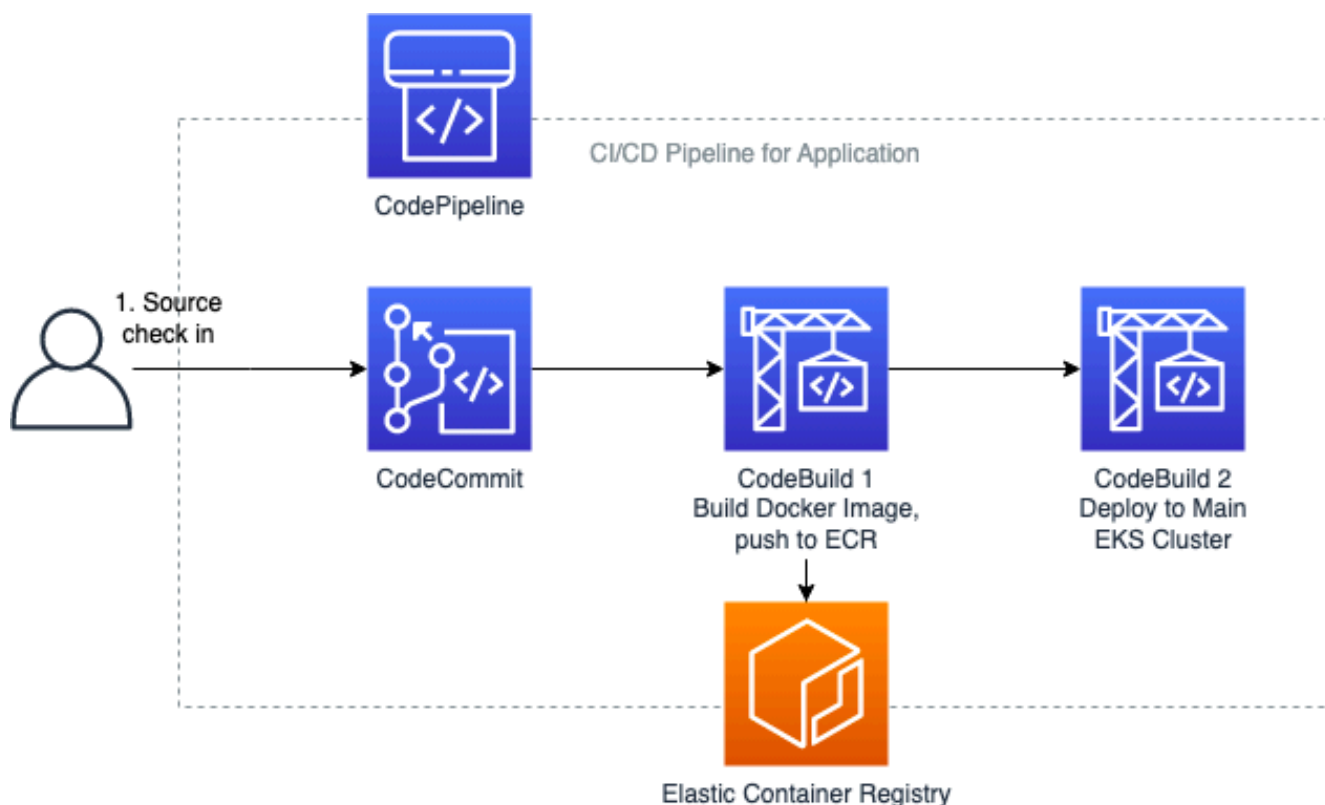


Рисунок 2.3 — Життєвий цикл контейнеру.

Тому переваги програмного рішення можна віднести до наступного.

Мінімальне використання ресурсів

Контейнер насправді не є всіма гостями, а використовує ядро хост -системи та ізолюють рівень процесу. Порівняно з віртуальною машиною, ресурси сервера, необхідні для таких процесів, значно менші.

Швидке розгортання

Швидке розгортання - спомогачі компоненти (бази даних, системи обробки черг тощо) не потрібно встановлювати окремо, але їх можна легко завантажити з публічних безкоштовних реєстрів. Крім того, вам не потрібно завантажувати одне і те ж зображення кілька разів -файлова система Docker розпізнає шар зображення і дозволяє занадто сильно використовувати той самий хеш-сум.

Легка обробка процесу

Колекція може використовувати різні методи обробки даних, щоб приховати процес примусового виконання на задньому плані.

Просте масштабування

Контейнер дуже легко бути горизонтальним і вертикальним. Тому ви можете легко адаптуватися до різкого збільшення навантаження.

Зручний старт

Контейнер може легко запустити на будь-якому господарстві пристані встановлення.

Оптимізація файлової системи

Docker використовує так названу багатошарову файлову систему. Його особливість полягає в тому, що зміни файлу зберігаються в шарі. Цей метод дозволяє легко використовувати один і той же шар (наприклад, зображення операційної системи може підкорити для запуску декількох контейнерів з різними програмами, включаючи різні списки залежностей, різний код програмного забезпечення тощо).

2.1.5 Розгортання сервісів у хмарах

Розгортання сервісів може відбуватися різними способами, залежно від інструментів та підходів, що використовуються. Ось деякі загальні способи розгортання сервісів за допомогою різних CLI, Terraform та вручну.

Використання CLI (командного рядка)

Більшість хмарних провайдерів (AWS, GCP, Azure тощо) надають інтерфейс командного рядка (CLI) для управління ресурсами хмари та розгортання сервісів. Наприклад, для розгортання сервісів у AWS, ви можете використовувати AWS CLI, для GCP - gcloud CLI, для Azure - Azure CLI. За допомогою команд CLI, ви можете створювати, оновлювати, масштабувати та видаляти ресурси, а також налаштувати параметри розгортання.

Використання Terraform

Terraform - це інструмент для кодифікації інфраструктури як коду (IAC). Він дозволяє вам автоматично створювати, змінювати та керувати інфраструктурою різних хмарних провайдерів за допомогою декларативних конфігураційних файлів. Ви створюєте файл конфігурації Terraform (зазвичай з розширенням .tf), в

якому описуєте ресурси та їх властивості. Потім ви виконуєте команди `terraform init`, `terraform plan` та `terraform apply` для ініціалізації, планування та розгортання ваших сервісів відповідно.

Вручну

Розгортання сервісів вручну зазвичай включає в себе використання веб-інтерфейсу (консолі) хмарного провайдера для створення та налаштування ресурсів. Ви можете виконувати ці дії крок за кроком, використовуючи інструкції та налаштування, надані хмарним провайдером. Хоча цей підхід може бути корисним для навчання та дослідження, він є менш ефективним та схильним до помилок у порівнянні з автоматизованими методами, такими як використання CLI або Terraform.

Підсумовуючи, ось процес розгортання сервісів за допомогою різних методів:

1) Використання CLI:

- а) встановіть та налаштуйте CLI хмарного провайдера;
- б) виконайте команди CLI для створення, оновлення, масштабування та видалення ресурсів та сервісів;

2) Використання Terraform:

- в) встановіть та налаштуйте Terraform;
- б) створіть файл(и) конфігурації (.tf) з описом ресурсів та їх властивостей;
- в) виконайте команди `terraform init`, `terraform plan` та `terraform apply` для ініціалізації, планування та розгортання сервісів;

3) Вручну:

- а) Увійдіть до веб-консолі вашого хмарного провайдера;
- б) Виконайте кроки для створення та налаштування ресурсів та сервісів, відповідно до документації та вимог.

Успішне розгортання сервісів вимагає обрання підходу, який найкраще підходить для вашої команди та вашого проекту. Автоматизовані підходи, такі як

використання CLI або Terraform, надають більшу швидкість, стабільність та зручність у порівнянні з ручним розгортанням. За вибору правильного інструменту та підходу, ви можете забезпечити успішне розгортання сервісів та підтримку вашої інфраструктури.

Ще кілька важливих аспектів та принципів, які слід враховувати під час розгортання сервісів у хмарі.

Моніторинг та логування

Встановіть системи моніторингу та логування для відстеження стану, продуктивності та доступності вашого сервісу. Хмарні провайдери зазвичай пропонують власні інструменти моніторингу, такі як Amazon CloudWatch, Google Stackdriver або Azure Monitor.

Безпека

Забезпечте захист вашого сервісу, використовуючи такі механізми, як шифрування, мережеві брандмауери, системи управління доступом та оновлення безпеки. Також переконайтеся, що ваші хмарні ресурси налаштовані відповідно до найкращих практик безпеки.

Резервне копіювання та відновлення

Створюйте регулярні резервні копії ваших даних та конфігурацій, щоб мати можливість відновити сервіс у разі відмови або пошкодження. Врахуйте використання служб хмарного зберігання, таких як Amazon S3, Google Cloud Storage або Azure Blob Storage, для зберігання резервних копій.

Неперервна інтеграція та неперервна доставка (CI/CD)

Застосуйте принципи CI/CD для автоматизації процесу розгортання сервісів та оновлення. Використовуйте інструменти, такі як Jenkins, GitLab CI/CD, GitHub Actions або хмарні рішення, як AWS CodePipeline та Google Cloud Build.

Скрипти для автоматизації

Створіть скрипти для автоматизації рутинних задач, таких як розгортання, оновлення, моніторинг та відновлення сервісів. Скрипти можуть бути написані на різних мовах програмування, таких як Python, Bash або PowerShell.

Використання цих доповнюючих практик та принципів допоможе забезпечити стабільність, продуктивність та безпеку вашого сервісу під час розгортання та підтримки.

Хмарна оптимізація

Враховуйте різні хмарні опції, щоб оптимізувати витрати та пропускну здатність вашого сервісу. Розгляньте використання різних типів віртуальних машин, способів зберігання та мережових конфігурацій для досягнення оптимального співвідношення ціни та продуктивності.

Автоматичне масштабування

Застосуйте автоматичне масштабування для динамічної зміни кількості ресурсів, що відповідають поточному навантаженню на ваш сервіс. Хмарні провайдери зазвичай пропонують рішення для автоматичного масштабування, такі як AWS Auto Scaling, Google Cloud AutoScaling або Azure Virtual Machine Scale Sets.

Відмовостійкість та розподіленість

Розгляньте розгортання вашого сервісу у різних регіонах та зонах доступності для забезпечення відмовостійкості та зменшення часу затримки. Також розгляньте використання хмарних послуг для розподіленого кешування та балансування навантаження, щоб забезпечити ефективне розподілення трафіку між ресурсами.

Контроль версій та співпраця

Використовуйте системи контролю версій, такі як Git, для стеження за змінами у вашому коді та конфігураціях. Використовуйте платформи співпраці, такі як GitHub, GitLab або Bitbucket, для організації проектів, співпраці та розгляду коду.

Застосування цих принципів та практик допоможе вам підвищити якість, стабільність та здатність вашого сервісу до масштабування, а також полегшить процес розгортання та управління сервісами у хмарі. Важливо забезпечити, що ваша команда розуміє та дотримується цих практик, щоб мінімізувати ризики, пов'язані з розгортанням та підтримкою хмарних сервісів.

Тестування

Розробіть стратегію тестування для перевірки функціональності, продуктивності та стабільності вашого сервісу. Використовуйте автоматизовані інструменти для тестування, щоб перевірити роботу вашого сервісу на різних рівнях, включаючи юніт-тестування, інтеграційне тестування та навантажувальне тестування.

Документація

Створіть документацію, яка допоможе вашій команді та іншим зацікавленим сторонам розуміти та працювати з вашим сервісом. Включіть в документацію опис архітектури, конфігурації, процесів розгортання, API, стандартів безпеки та моніторингу.

Оновлення та підтримка

Регулярно оновлюйте ваш сервіс, включаючи виправлення помилок, покращення продуктивності, нові функції та оновлення безпеки. Встановіть процеси для підтримки, вирішення проблем та забезпечення неперервної роботи вашого сервісу.

З метою забезпечення успішного розгортання сервісів та ефективного управління хмарними ресурсами, важливо підходити до цього процесу комплексно, враховуючи різні аспекти, такі як автоматизація, моніторинг, безпека, тестування, документація та підтримка. Ось декілька додаткових рекомендацій для забезпечення успішного розгортання та управління сервісами у хмарі.

Співпраця між командами

Сприяти співпраці між різними командами у вашій організації, такими як розробники, адміністратори систем, архітектори хмари та інженери безпеки. Це полегшить обмін знаннями та досвідом, а також забезпечує швидше вирішення проблем та покращення сервісу.

Навчання та сертифікація

Забезпечте навчання та підвищення кваліфікації вашої команди в області хмарних технологій та практик. Розгляньте отримання сертифікацій від

провайдерів хмарних послуг, які підтверджують знання та досвід у роботі з конкретними хмарними платформами.

Оцінка та оптимізація витрат

Слідкуйте за витратами на ваші хмарні ресурси та періодично проводьте оцінку та оптимізацію, щоб забезпечити ефективне використання ресурсів та контроль над витратами. Використовуйте інструменти аналізу витрат та рекомендації провайдера хмарних послуг для оптимізації ресурсів.

Стратегія розгортання

Оберіть стратегію розгортання, що найкраще відповідає вашим бізнес-вимогам та цілям, такі як "blue-green" розгортання, канарейкове розгортання або розгортання з нуля. Різні стратегії розгортання мають свої переваги та недоліки, тому важливо ретельно обрати найбільш підходящий підхід.

Відповідальність та SLA (Service Level Agreements)

Встановіть чіткі рівні відповідальності між вашою командою та провайдером хмарних послуг. Ознайомтеся з SLA, які пропонуються провайдерами, та переконайтеся, що вони відповідають вашим очікуванням щодо доступності, продуктивності та підтримки. Не вагайтеся переговорити про можливість внесення змін до SLA, якщо вони не відповідають вашим вимогам.

Відновлення після відмови (Disaster Recovery)

Розробіть план відновлення після відмови, який враховує різні сценарії, такі як відмови обладнання, збої програмного забезпечення, атаки на безпеку та природні катастрофи. Ваш план повинен включати процедури відновлення, резервне копіювання та тестування, а також планування забезпечення неперервної роботи.

Регулярне огляд та оновлення

Періодично проводьте огляд вашого сервісу, архітектури, процесів розгортання та управління хмарними ресурсами. Оновлюйте ваші практики, технології та інструменти відповідно до змін у вашому бізнесі, ринку та хмарних технологіях.

Застосування цих рекомендацій допоможе вашій команді підтримувати високий рівень якості та ефективності розгортання та управління сервісами у хмарі. Врахуйте ці принципи та практики під час планування, розробки, розгортання та підтримки вашого хмарного сервісу, щоб забезпечити його стабільність, продуктивність, безпеку та масштабованість.

2.2. Оркестрація контейнерів

Основне завдання розташування контейнера - розмістити контейнер на кластері машини та контролювати стан, збір індикаторів тощо. Це означає, що всі контейнери, які виконують різні навантаження, що надаються домовленістю, можуть бути запущені на фізичних або віртуальних машинах.

Необхідно дотримуватися обмежень середовища розгортання та конфігурації кластера, а контейнери повинні бути ефективно розподілені. Крім того, організатори повинні стежити за всіма робочими контейнерами та замінити неактивні (тих, хто не вдається, і більше не виконують свої функції), не відповідаючи на запит запиту системи чи не у відповідь на систему огляду. Усі ці функції та багато більш сучасних популярних домовленостей можуть бути виконані. Крім того, розташування може використовувати плагін, який користувачі можуть створювати та встановлювати за допомогою стандартного інтерфейсу для розширення власних наборів функцій.

Для того, щоб запустити навантаження, вам потрібно забезпечити робоче середовище. До них належать фактичні фізичні машини з певними функціями розрахунку (з різними фізичними процесорами або ядрами), оперативною пам'яттю та зберіганням даних (HDD або SSD). Крім того, потрібна комбінація постійного складу даних (наприклад, NFS), і всі ці машини підключені до загальної мережі, щоб вони могли знайти один одного та спілкуватися між собою. На цьому етапі попит - це лише певна кількість машин (віртуальних чи фізичних), які мають безкоштовні обчислювальні ресурси і можуть спілкуватися між собою. Упорядкування контейнерів можна розгорнути на реальному кластері (реальному фізичному обладнанні) або групі віртуальної машини. Тому оркестр може

організувати контейнер, керований безпосередньо на фізичних або віртуальних машинах.

Кластери розташування контейнерів також можуть складатися з фізичних комп'ютерів та віртуальних машин, але цього достатньо

Загальний метод. Коли в контейнері розміщується велика кількість мікро-сервісів, користувачі зазвичай не отримують занадто багато додаткових витрат, а віртуальні машини зазвичай такі. Це робить контейнер дуже підходящим для розгортання хмари, що невидимим вибрати всю віртуальну машину для кожної мікросервісу.

Сьогодні всі основні хмарні постачальники, такі як AWS, GCP та Azure, надають послуги хостингу контейнерів. Деякі з них (наприклад, GKE Goom) базуються на Kubernetes. Інші, такі як послуги контейнерів Microsoft Azure, базуються на інших рішеннях. До речі, AWS має власні системи System-EC (через службу розгортання контейнерів EC2), яка використовує власне рішення. Hashicorp Nomad - ще одна дуже популярна і потужна домовленість. Компанія - це велика кількість програмного забезпечення, яке тісно пов'язане з хмарними технологіями. У таблиці 2.4 наведено більш детальне порівняння контейнерного розташування.

Таблиця 2.4 — Порівняння контейнерного розташування.

	Kubernetes	Nomad	Apache Mesos	ECS	ACI	Jenkins
Швидкість розгортання	+++	+	++++	++++	++++	++++
Складність	++	++	+++	++	++	++
Масштабування	+++++	+++++	+++++	+	+	+
Синтаксис конфігурації	YAML	YAML	JSON	JSON	JSON	YAML
Хмарні провайтери	AWS, GCP, Azure	AWS, GCP, Azure	AWS, GCP, Azure	AWS	Azure	Jenkins
Відкритий код	+	-	+	-	-	+
Мінімальний розмір	1	1	1	1	1	1

Важлива відмінність між Kubernetes полягає в тому, що його можна запустити в будь-якому середовищі та купувати як послугу від більшості хмарних постачальників. Kubernetes має інтерфейс для постачальника хмари, який дозволяє кожному постачальнику хмара встановлювати його та легко інтегруватися з екосистемою, про що свідчить статистика з рисунку 2.5. Статистика чітко вказує на широке використання Kubernetes, що також пояснюється такими причинами, як простота використання та можливості інтеграції з існуючою інфраструктурою. Установка Kubernetes також можлива для

очисних машин з різною архітектурою процесора, такими як ARM та X86, які дозволяють повторно використовувати вже реагувати сервери в Clary CDO або розміщувати приватні дані на даних.

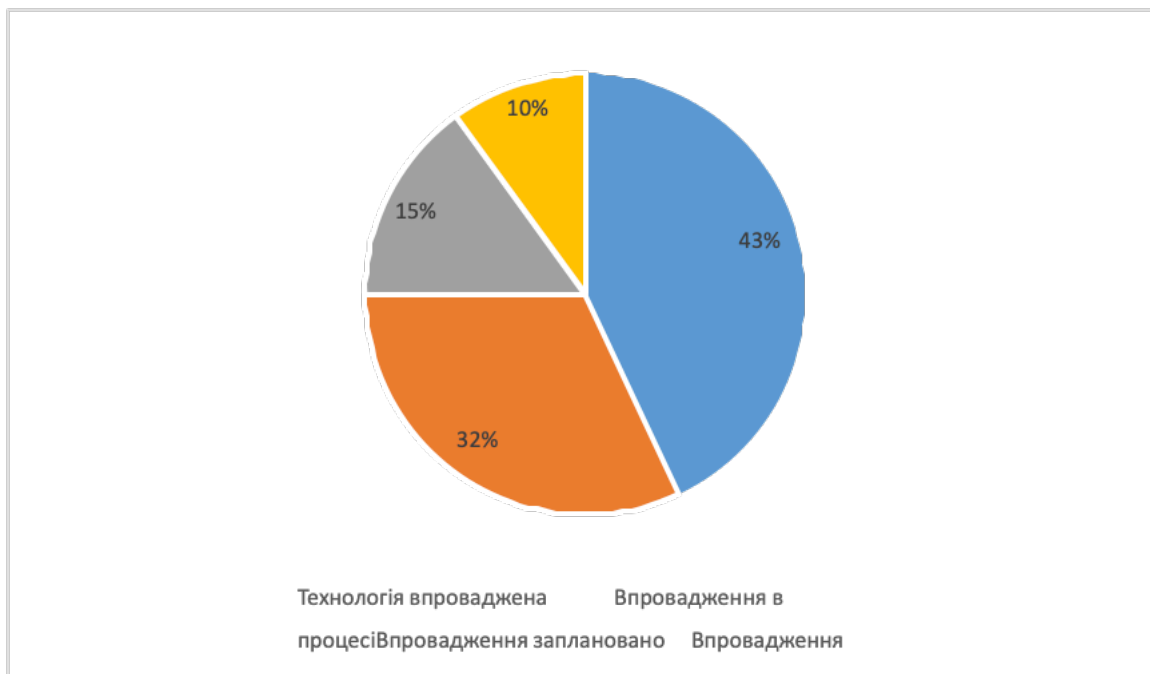


Рисунок 2.5 — Статистика впровадження технології.

Орнаменти Kubernetes створюються в стіні Google, щоб задовольнити їхні програми. Згодом рішення також було застосовно до системи розгортання, і система також використовувалася кінцевими користувачами компанії (сотні чи навіть тисячі продуктів у всьому світі). Після того, як програмне забезпечення може найкраще показати власні продукти в компанії, деякі програмні коди влаштовуються на ім'я Kubernetes для відкритого доступу.

Зараз Kubernetes використовує тисячі компаній у всіх галузях. Нова версія програми випускається щотижня, що вказує на швидкий розвиток технології та надзвичайно широко використовується.

Крім того, Kubernetes - це рішення PAAS від багатьох постачальників хмарних обчислень. включати:

- GCP;
- AWS;

- Digitalocean;
- Azure;
- IBM Cloud.

За даними агентства Dzone, Kubernetes використовується більш ніж 40 % команди (рис. 2.6), яка впроваджена та в даний час запровадила технологію Docker. Фонд обчислювальних обчислень CNCF -cloud також підтвердив це. Це неофіційне регуляторне агентство (не офіційне), яке визначило, які технології вважаються повністю стандартизованими та придатними для використання в хмарному середовищі. Крім того, CNCF є частиною фонду некомерційної організації Linux, яка займається підтримкою, розробкою та стандартизацією компонента Linux Environment.

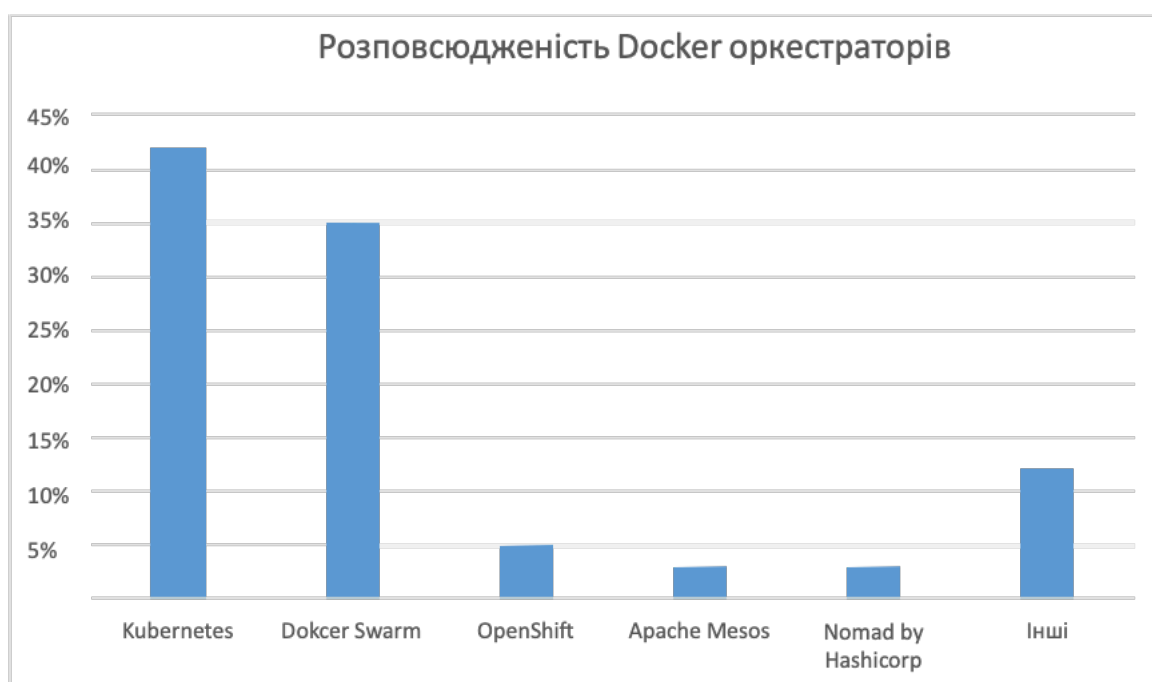


Рисунок 2.6 — Розповсюдження Docker.

Тому, враховуючи технологію Kubernetes, динаміку та її потенціал розвитку, поширеність відкритого програмного коду та можливості розширення, можна сказати, що це беззаперечний лідер у рішеннях -оркестрації.

Завдяки вищезазначеним характеристикам та перевагам автоматизована система, розроблена в дисертації цього магістра, розроблена для взаємодії з оркестром Kubernetes.

2.2.1 Огляд використання Kubernetes

Як правило, коли система складається з менших послуг та серверів, кожен сервер має власну статичну назву та конфігурацію. Розробники та користувачі точно знають, яке програмне забезпечення на кожній машині. Багато компаній обговорювали багато днів, щоб вибрати назву серверного парку.

Коли сервер перестає працювати, може виникнути велика криза. Усі намагалися з'ясувати, де взяти інший сервер, який навіть почав на невдалому сервері, і як змусити їх працювати на новому сервері. Якщо сервер зберігає деякі важливі дані, то кожен хоче локальну резервну копію або навіть відновити.

Очевидно, цей метод не розширився. Коли є десятки або сто серверів, вам слід почати бачити їх як динамічні. Загальні умови системи слід враховувати, а не параметри кожного окремого сервера чи послуги.

Kubernetes повністю відповідає за розподіл контейнерів на певних машинах. Більшу частину часу вам не потрібно взаємодіяти з одним вузлом. Наприклад, на відміну від бази даних є навантаження, яке не потребує поточного стану (з англійської).

Що стосується країни, ситуація відрізняється, але Kubernetes забезпечує рішення під назвою Державне задоволення.

Kubernetes має на меті керувати та спростити організацію, розгортати та підтримувати розподілені системи в широкому спектрі навколишнього середовища та хмарних постачальників. Він надає безліч можливостей і повинно відігравати роль у всіх цих різновидах, розробляє та підтримує досить прості, щоб використовувати звичайних розробників. Це дуже складна мета. Kubernetes досягає цієї мети, дотримуючись чіткої конструкції високого рівня та продуманих будівель, тим самим сприяючи розширенню та розвитку. Багато частин Kubernetes

все ще мають досить суворі обмеження або конкретні обмеження для компаній-розробників, але тенденція полягає в тому, щоб обробити їх як плагін та зберегти основні загальні та абстрактні частини.

2.2.2 Огляд складових Kubernetes

У основних компонентах Kubernetes, таких як кластер тощо, менеджер контролера, планувальник та вузол.

Кластер - це обчислювальні ресурси, зберігання даних та мережеві ресурси, які Kubernetes використовує для запуску системи системи, що утворює систему. Слід також зазначити, що завдяки встановленню комбінації постійного струму комбінацій постійного струму в різних географічних областях може бути об'єднати федерацію федерації різних хмар з декількох постачальників хмарних послуг.

Основним компонентом управління кластерами є панель управління Kubernetes. Він складається з декількох компонентів, таких як сервер API, планувальник завдань та менеджер контролера. Планувальник завдань відповідає за глобальний план

Об'єкт кластерного рівня та об'єкт лікування подій. Як правило, всі основні компоненти встановлюються на одному хості. Враховуючи схему забезпечення високої доступності або дуже великих кластерів, ці компоненти можна розділити через різних хостів або компоненти, перелічені на кількох хостах.

Кластер Kubernetes має кілька основних компонентів для управління кластером, а також вузлів, що працюють на кожному вузлі кластера.

Основні компоненти зазвичай працюють на вузлі, але вони можуть бути розподілені в декількох вузлах з високим доступом або великими кластерами. Інтерактивна схема компонентів відображається на рисунках 2.7, 2.8, 2.9.

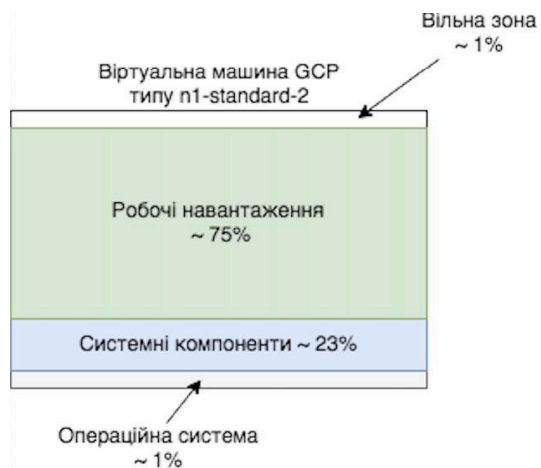


Рисунок 2.7 — Розподіл ресурсів GSP.



Рисунок 2.8 — Розподіл ресурсів AWS.

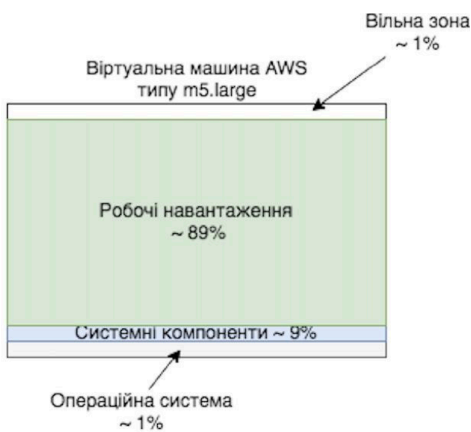


Рисунок 2.9 — Розподіл ресурсів Azure.

Сервер API може легко збільшити масштаб горизонтально, оскільки він не має чіткого поточного стану та зберігає всі дані в кластері ETCD. Сервер API - це реалізація панелі управління Kubernetes.

ETCD - це високо розподілене зберігання даних. Kubernetes використовує його для зберігання всього кластера. У невеликому кластері (десятки вузлів) або такої високої надійності екземпляр ETCD може використовувати вузол з усіма іншими основними компонентами. Але для великих кластерів тощо.

Менеджер контролерів - це сукупність різних менеджерів і перетворює його на компонент програмного забезпечення. Він містить контролер копіювання, контролер контейнера, контролер служби, точка входу тощо. Усі ці менеджери можуть стежити за статусом кластерів через API, а їх завдання - керувати кластерами в необхідному стані.

Планувальник несе відповідальність за планування контейнера на вузлі. Це дуже складне завдання, оскільки воно повинно враховувати багато факторів, таких як:

- наявність обмежень розгортання;
- наявність безкоштовних обчислювальних ресурсів;
- доступність вузла;
- найкращий характер вибору вузлів;
- інший.

Планувальник розгортається як звичайний контейнер. Кожна служба в кластері (за винятком So -So -Called "без голови") отримає відповідні унікальні записи в DNS. Стручки також можуть отримувати імена DNS. Дуже зручно створити систему за допомогою компонентів автоматичного виявлення.

Кластерні вузли повинні взаємодіяти з основними компонентами кластера, наприклад, для отримання навантаження для виконання та оновлення стану, пов'язаного з кластером. Від розподілу ресурсів до одного компонента, ми можемо зробити висновки. Найкращий користувач - розподіл AWS. Рисунок 2.10 Порівняння розподілу ресурсів

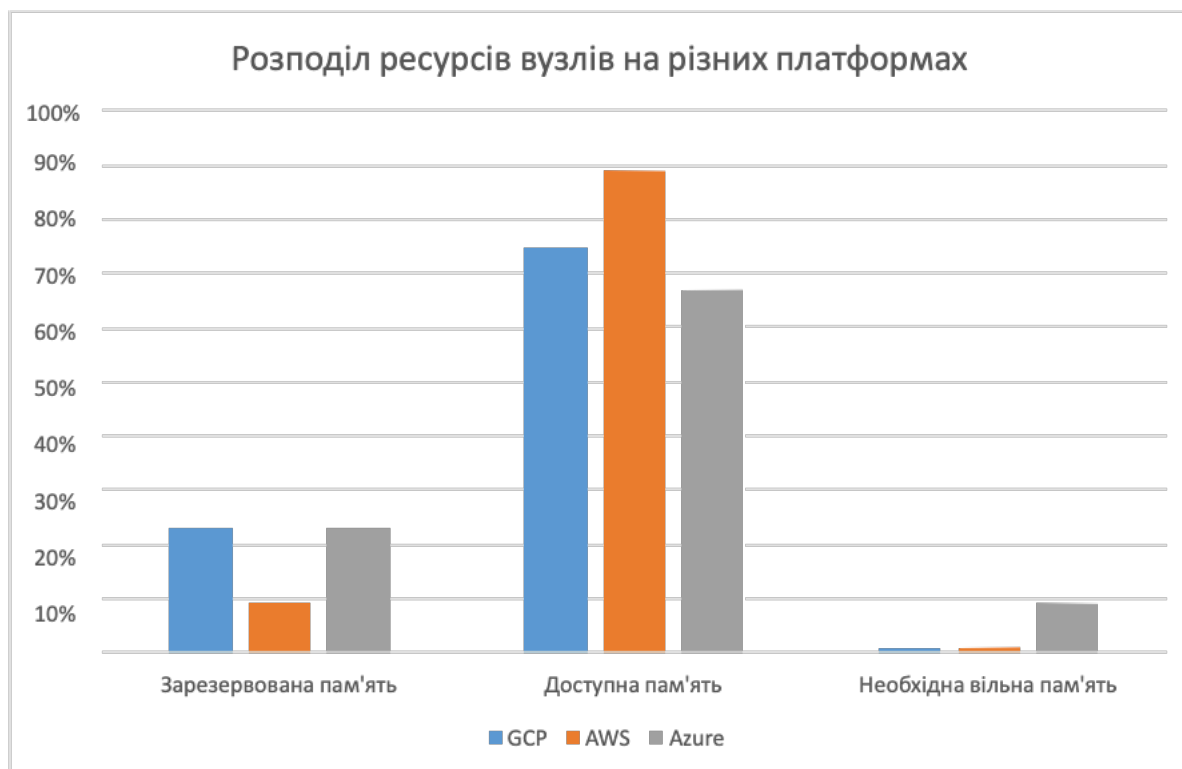


Рисунок 2.10 — Порівняння розподілу ресурсів.

Проксі Kube виконує системні операції з низьким рівнем на кожному вузлі. Він відображає послугу Kubernetes локально і може переслати протокол передачі через TCP та UDP. Він використовує змінні або DNS для пошуку IP -адреси кластера.

Наступним компонентом вузла є Kubelet, який контролює взаємодію з основними компонентами та керує програмою програми. Сюди входить наступне:

- завантажити секрет програми з сервера API;
- встановити зовнішній рулон для зберігання даних;
- запустити заявку (Docker або RKT);
- стан вузла сповіщення та кожної програми;
- початок перевірки на правильність роботи контейнера.

Крім того, Kubernetes, особливо поняття основної архітектури контейнера, особливо:

- допоміжний контейнер (Sidecar);
- контейнер для посла;

- контейнер адаптера;
- мультикольна концепція.

Архітектура допоміжних контейнерів заснована на основному контейнері програми. Основний контейнер програми нічого не знає про допоміжний контейнер, але продовжує виконувати свої прямі функції. Хороший приклад - зібрати агент журналу. Основний контейнер може бути просто записаний у потоці `stdout` або `stderr`. Допоміжний контейнер може перехоплювати відкликання та надсилати дані до центрального зберігання даних у журналі (наприклад, стек EFK/Elk).

Перевага додавання централізованої системи входу до основного контейнера додатків до основного контейнера програми величезна.

Перш за все, вкладення більше не дозволяють величезній логіці централізованих систем та збирати журнали, що може спричинити проблеми. Якщо вам потрібно оновити або змінити логічні стратегії або перейти на нове рішення, вам потрібно лише оновити допоміжний контейнер та розгорнути його. Основні контейнери програми не змінилися, тому неможливо випадково знищити основну систему.

Контейнер Ambassador - це можливість представляти віддалені послуги. Це, здається, місцеві послуги та поєднує конкретні стратегії. Хороший приклад методу полягає в, що існує кластер для повторного кластера, один з яких записи та багато читань. Місцеві контейнери можуть використовуватися як агенти та представляти повторний контейнер на програмі на локальному хост-інтерфейсі. Основний контейнер додатків підключений лише до Redis на LocalHost: 6379 (порт Redis - порт Redis), але в цьому випадку він підключений до посла, який працює в тому ж кластері у запиті фільтрації та надсилання. Вибрана репліка використовується для читання. Так само, як використання допоміжних контейнерів, основна програма може зрозуміти, що відбувається, а шар абстракції буде доданий, якщо це необхідно. Цей метод може використовувати реальний локальний кластерний повтор, щоб допомогти тестам. Крім того, якщо

конфігурація кластера Redis змінюється, вам потрібно лише змінити конфігурацію посол та головний план залишаються незмінними.

Контейнер адаптера зазвичай використовується для виходу основного контейнера програми для стандартизації. Розглянемо ситуацію поступової служби: вона генерує відповіді форматом, який не відповідає попередній версії. Послуги та заявки цих відповідей не були оновлені. Контейнер адаптера може бути розгорнутий у модулі з новим контейнером програми, а вихід форматується для їх форматування, щоб стара версія була придатною перед оновленням усіх сусідніх служб. Контейнер адаптера використовує файлову систему разом з основним контейнером програми, щоб він міг переглянути локальну файлову систему, а коли нова програма записується в якийсь вміст, він негайно регулює вихід.

Концепція множинних широко використовується. Це особливо використовується в системі, спрямованій на підтримку поточного стану даних (наприклад, бази даних кластерів):

- Cassandra;
- Elasticsearch;
- Redis;
- Hadoop;
- Kafka;
- MySQL;
- MongoDB;
- Apache hbase.

Cobenite безпосередньо підтримує концепцію однієї закритої програми за допомогою розгортання. Не підтримуйте безпосередньо використання виборів лідерів, робочих черг та операцій з розповсюдження, але ви можете використовувати стандартні компоненти Kubernetes або так, що спрямовують оператори для розгортання. Наступні розділи розглянуть концепцію операторів, оскільки це дуже важлива частина всієї екосистеми Kubernetes.

2.2.3 Аналіз типів оточень запуску контейнерів

Kubernetes спочатку підтримував докер лише як механізм запуску контейнера. Але це не так. RKT - це ще один механізм виконання підтримуючих, і цікаво використовувати його з контейнером Hyper.sh через гіпернети. Основна політика проектування полягає в тому, що сам Kubernetes повинен бути повністю відокремлений від певного режиму роботи.

Взаємодія між Kubernetes та реалізацією навколишнього середовища проводиться через досить поширений інтерфейс і повинна бути реалізована. Більшість комунікацій використовує концепцію стручка, контейнерів та операцій, які можна виконати над контейнером. Кожен механізм виконання відповідає за реалізацію інтерфейсу Kubernetes, сумісного з оркестром.

Проект Github Kubernetes перераховує інтерфейс контейнера. Kubernetes має відкритий програмний код, тому ви можете переглянути опис інтерфейсів, відповідальних за контейнер, і побачити, що набір функцій незначний без конкретного вмісту.

Це означає, що при реалізації інтерфейсу слід реалізувати метод інтерфейсу. Інтерфейс визначається у файлі. Просто назва інтерфейсу дає багато інформації про їх роботу. Очевидно, що Kubernetes потрібно запускати команди в контейнері, і йому потрібно прикріпити контейнер до своєї програми та намалювати зображення контейнера.

Звичайно, Docker є фактичним стандартом світу контейнерності. Kubernetes спочатку був розроблений для управління контейнером Docker. Можливість виконання в декількох середовищах у кількох середовищах Kubernetes 1.3. До того часу Kubernetes може контролювати лише контейнер Docker.

Докер дуже популярний і росте, але є багато критики. Критики часто згадують таку увагу:

- безпека;
- важко встановити кілька програм упаковки (включаючи мережу);
- розробка, моніторинг та обслуговування журналів;

- контейнер Docker обмежує лише один процес.

Докер вирішує деякі з них. Особливо створив новий оркестр. Docker Swarm - це рішення для Docker і конкурує з Kubernetes. Це легше використовувати, ніж Kubernetes, але не настільки потужні та зрілі. Починаючи з Docker 1.12, Docker Daemon містить режим рою, який спричинив багато недоліків, включаючи розмір та потенційні проблеми коду Docker демона. І навпаки, це перетворює більше людей на CoreOS RKT як альтернативне рішення.

Починаючи з Docker 1.11, випущеного в квітні 2016 року, Docker змінив шлях до контролю контейнера. У середовищі виконання використовується Contanel та Runc для запуску відкритого контейнера (OCI) у контейнері. Ця архітектура зображена на рисунку 2.11.

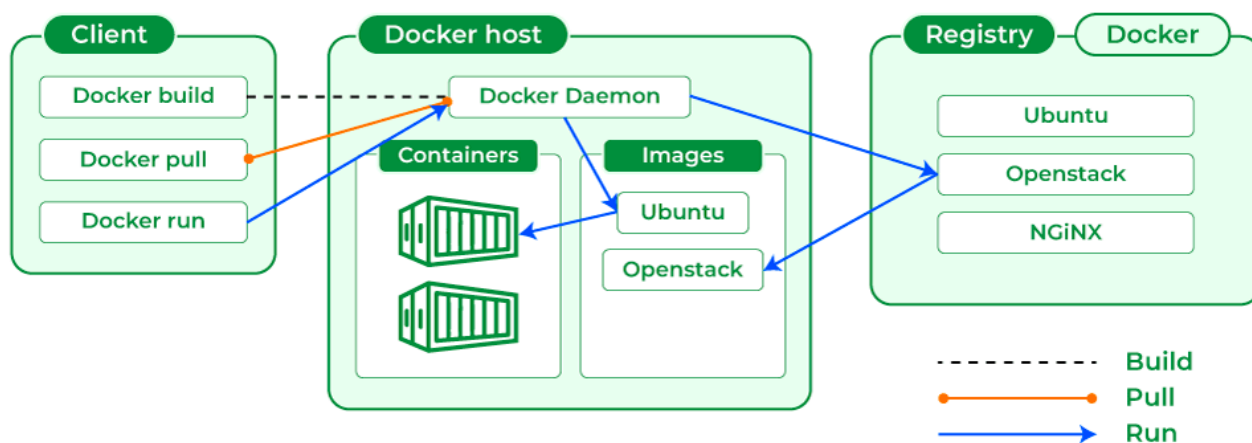


Рисунок 2.11 — Архітектура Docker.

Тому OCI забезпечує кілька середовищ для контейнерів. Ці середовища відповідають потребам більшої кількості кінцевих споживачів. Ці потреби можна розглядати як альтернативу RKT як альтернативу міграції, і користувачі можуть довго використовувати Docker.

2.2.4 Поди у Kubernetes

Стейтфул сет - це найменша одиниця в Кубернетах. Кожен містить один або кілька контейнерів -це можуть бути як допоміжні контейнери, або програми, що

складаються з декількох контейнерів. Усі контейнери у відділі мають однакову IP-адресу та простір портів; вони можуть спілкуватися з інтерфейсом Localhost або обмінюватися інформацією через публічну файлову систему.

Крім того, всі контейнери в модулі можуть отримати доступ до універсального локального зберігання вузлів, де знаходиться нижня частина. У цьому випадку кожен контейнер може отримати доступ до кожного контейнера. SAF - важлива частина Kubernetes. Ви можете запустити деякі програми в контейнері Docker. Існують процедури, подібні до керівника, як основна програма Docker. Ця програма запускає кілька процесів. Однак через наступні причини такий підхід часто скаржиться.

Прозорість

Робота контейнера в осцилоскопі дозволяє всім контейнерам бачити інфраструктуру та дозволяти інфраструктурі надавати послуги цим контейнерам, такими як управління процесами та спостереження за ресурсами. Це полегшує вирішення всього посилання у кінцевих користувачів.

Програмні залежності

Суцільні контейнери можуть незалежно обмежувати різні версії залежності програмного забезпечення, і різні архітектори ОС можуть бути навіть доступними або відсутніми компонентами. Kubernetes може навіть підтримувати контейнери в майбутньому.

Простий у використанні

Користувачам не потрібно запускати власний менеджер процесів, турбуйтеся про сигнали системи обробки та код виходу тощо;

Ефективність

Оскільки компонент інфраструктури бере на себе найбільшу відповідальність за відповідну продуктивність програми, контейнер програмного забезпечення може бути дуже нестабільним (реорганізація, невдача тощо). Приклад конфігурацій коду зображено на рисунках. 2.12 - 2.19 та 2.20.

POD - це найменша одиниця в Кубернетах. Кожен містить один або кілька контейнерів -це можуть бути як допоміжні контейнери, або програми, що

складаються з декількох контейнерів. Усі контейнери у відділі мають однакову IP-адресу та простір портів; вони можуть спілкуватися з інтерфейсом Localhost або обмінюватися інформацією через публічну файлову систему.

```

1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: ingress-nginx-controller-6bf7bc7f94-pn7sn
5    generateName: ingress-nginx-controller-6bf7bc7f94-
6    namespace: ingress-nginx
7    uid: 73bd6192-8261-4765-847e-bd963dab0c96
8    resourceVersion: '26381730'
9    creationTimestamp: '2023-04-06T15:37:46Z'
10   labels:
11     app.kubernetes.io/component: controller
12     app.kubernetes.io/instance: ingress-nginx
13     app.kubernetes.io/name: ingress-nginx
14     pod-template-hash: 6bf7bc7f94
15   ownerReferences:
16     - apiVersion: apps/v1
17       kind: ReplicaSet
18       name: ingress-nginx-controller-6bf7bc7f94
19       uid: dabcdabc-82e3-4213-a3d0-31e5530f4c16
20       controller: true
21       blockOwnerDeletion: true
22   managedFields:
23     - manager: kube-controller-manager
24       operation: Update
25       apiVersion: v1
26       time: '2023-04-06T15:37:46Z'
27       fieldsType: FieldsV1
28       fieldsV1:
29         f:metadata:
30           f:generateName: {}
31           f:labels:
32             .: {}
33             f:app.kubernetes.io/component: {}
34             f:app.kubernetes.io/instance: {}
35             f:app.kubernetes.io/name: {}
36             f:pod-template-hash: {}
37         f:ownerReferences:
38           .: {}
39           k:{"uid":"dabcdabc-82e3-4213-a3d0-31e5530f4c16"}: {}
40         f:spec:
41           f:containers:
42             k:{"name":"controller"}:
43               .: {}
44               f:args: {}
45               f:env:
46                 .: {}
47                 k:{"name":"LD_PRELOAD"}:
48                   .: {}

```

Рисунок 2.12 — Конфігурація поду NGINX.

```

49     f:name: {}
50     f:value: {}
51     k:{"name":"POD_NAME":
52     .: {}
53     f:name: {}
54     f:valueFrom:
55     .: {}
56     f:fieldRef: {}
57     k:{"name":"POD_NAMESPACE":
58     .: {}
59     f:name: {}
60     f:valueFrom:
61     .: {}
62     f:fieldRef: {}
63     f:image: {}
64     f:imagePullPolicy: {}
65     f:lifecycle:
66     .: {}
67     f:preStop:
68     .: {}
69     f:exec:
70     .: {}
71     f:command: {}
72     f:livenessProbe:
73     .: {}
74     f:failureThreshold: {}
75     f:httpGet:
76     .: {}
77     f:path: {}
78     f:port: {}
79     f:scheme: {}
80     f:initialDelaySeconds: {}
81     f:periodSeconds: {}
82     f:successThreshold: {}
83     f:timeoutSeconds: {}
84     f:name: {}
85     f:ports:
86     .: {}
87     k:{"containerPort":80,"protocol":"TCP":
88     .: {}
89     f:containerPort: {}
90     f:name: {}
91     f:protocol: {}
92     k:{"containerPort":443,"protocol":"TCP":
93     .: {}
94     f:containerPort: {}
95     f:name: {}
96     f:protocol: {}

```

Рисунок 2.13 — Конфігурація поду NGINX.

```

97       k:{"containerPort":8443,"protocol":"TCP"}:
98         .: {}
99         f:containerPort: {}
100        f:name: {}
101        f:protocol: {}
102      f:readinessProbe:
103        .: {}
104        f:failureThreshold: {}
105        f:httpGet:
106          .: {}
107          f:path: {}
108          f:port: {}
109          f:scheme: {}
110        f:initialDelaySeconds: {}
111        f:periodSeconds: {}
112        f:successThreshold: {}
113        f:timeoutSeconds: {}
114      f:resources:
115        .: {}
116        f:requests:
117          .: {}
118          f:cpu: {}
119          f:memory: {}
120      f:securityContext:
121        .: {}
122        f:allowPrivilegeEscalation: {}
123        f:capabilities:
124          .: {}
125          f:add: {}
126          f:drop: {}
127          f:runAsUser: {}
128      f:terminationMessagePath: {}
129      f:terminationMessagePolicy: {}
130      f:volumeMounts:
131        .: {}
132        k:{"mountPath":"/usr/local/certificates/"}:
133          .: {}
134          f:mountPath: {}
135          f:name: {}
136          f:readOnly: {}
137      f:dnsPolicy: {}
138      f:enableServiceLinks: {}
139      f:nodeSelector: {}
140      f:restartPolicy: {}
141      f:schedulerName: {}
142      f:securityContext: {}
143      f:serviceAccount: {}
144      f:serviceAccountName: {}

```

Рисунок 2.14 — Конфігурація поду NGINX.

```

145     f:terminationGracePeriodSeconds: {}
146     f:volumes:
147       .: {}
148       k:{"name":"webhook-cert"}:
149         .: {}
150         f:name: {}
151         f:secret:
152           .: {}
153           f:defaultMode: {}
154           f:secretName: {}
155   - manager: kubelet
156     operation: Update
157     apiVersion: v1
158     time: '2023-04-06T15:38:19Z'
159     fieldsType: FieldsV1
160     fieldsV1:
161       f:status:
162         f:conditions:
163           k:{"type":"ContainersReady"}:
164             .: {}
165             f:lastProbeTime: {}
166             f:lastTransitionTime: {}
167             f:status: {}
168             f:type: {}
169           k:{"type":"Initialized"}:
170             .: {}
171             f:lastProbeTime: {}
172             f:lastTransitionTime: {}
173             f:status: {}
174             f:type: {}
175           k:{"type":"Ready"}:
176             .: {}
177             f:lastProbeTime: {}
178             f:lastTransitionTime: {}
179             f:status: {}
180             f:type: {}
181       f:containerStatuses: {}
182       f:hostIP: {}
183       f:phase: {}
184       f:podIP: {}
185       f:podIPs:
186         .: {}
187         k:{"ip":"10.244.4.10"}:
188           .: {}
189           f:ip: {}
190       f:startTime: {}
191     subresource: status
192   selfLink: >-

```

Рисунок 2.15 — Конфігурація поду NGINX.

```

192 --selfLink: >-
193 .../api/v1/namespaces/ingress-nginx/pods/ingress-nginx-controller-6bf7bc7f94-pn7sn
194 status:
195   phase: Running
196   conditions:
197     - type: Initialized
198       status: 'True'
199       lastProbeTime: null
200       lastTransitionTime: '2023-04-06T15:37:46Z'
201     - type: Ready
202       status: 'True'
203       lastProbeTime: null
204       lastTransitionTime: '2023-04-06T15:38:19Z'
205     - type: ContainersReady
206       status: 'True'
207       lastProbeTime: null
208       lastTransitionTime: '2023-04-06T15:38:19Z'
209     - type: PodScheduled
210       status: 'True'
211       lastProbeTime: null
212       lastTransitionTime: '2023-04-06T15:37:46Z'
213   hostIP: 10.224.0.4
214   podIP: 10.244.4.10
215   podIPs:
216     - ip: 10.244.4.10
217   startTime: '2023-04-06T15:37:46Z'
218   containerStatuses:
219     - name: controller
220       state:
221         running:
222           startedAt: '2023-04-06T15:38:06Z'
223         lastState: {}
224       ready: true
225       restartCount: 0
226       image: sha256:4d43c7489bf235a235427c22537bd2ae5009fb1a594bb484e9a0187baa4b35ca
227       imageID: >-
228         registry.k8s.io/ingress-nginx/controller@sha256:d1707ca76d3b044ab8a28277a2466a02100ee9f58a86af1535a3edf9323ea1b5
229       containerID: >-
230         containerd://3d055b88aaec352260fcc1ed84d2ce7158ed644561316dc87b0f6d2fd93a96f4
231       started: true
232   qosClass: Burstable
233 spec:
234   volumes:
235     - name: webhook-cert
236       secret:
237         secretName: ingress-nginx-admission
238         defaultMode: 420

```

Рисунок 2.16 — Конфігурація поду NGINX.

```

239 - name: kube-api-access-2fd7s
240   projected:
241     sources:
242       - serviceAccountToken:
243         expirationSeconds: 3607
244         path: token
245       - configMap:
246         name: kube-root-ca.crt
247         items:
248         - key: ca.crt
249           path: ca.crt
250       - downwardAPI:
251         items:
252         - path: namespace
253           fieldRef:
254             apiVersion: v1
255             fieldPath: metadata.namespace
256       defaultMode: 420
257 containers:
258   - name: controller
259     image: >-
260       registry.k8s.io/ingress-nginx/controller:v1.3.0@sha256:d1707ca76d3b044ab8a28277a2466a02100ee9f58a86af1535a3edf9323ea1b5
261     args:
262       - /nginx-ingress-controller
263       - '--publish-service=$(POD_NAMESPACE)/ingress-nginx-controller'
264       - '--election-id=ingress-controller-leader'
265       - '--controller-class=k8s.io/ingress-nginx'
266       - '--ingress-class=nginx'
267       - '--configmap=$(POD_NAMESPACE)/ingress-nginx-controller'
268       - '--validating-webhook=:8443'
269       - '--validating-webhook-certificate=/usr/local/certificates/cert'
270       - '--validating-webhook-key=/usr/local/certificates/key'
271     ports:
272       - name: http
273         containerPort: 80
274         protocol: TCP
275       - name: https
276         containerPort: 443
277         protocol: TCP
278       - name: webhook
279         containerPort: 8443
280         protocol: TCP
281     env:
282       - name: POD_NAME
283         valueFrom:
284           fieldRef:
285             apiVersion: v1
286             fieldPath: metadata.name

```

Рисунок 2.17 — Конфігурація поду NGINX.

```

286 .....fieldPath: metadata.name
287 - name: POD_NAMESPACE
288   valueFrom:
289     fieldRef:
290       apiVersion: v1
291       fieldPath: metadata.namespace
292 - name: LD_PRELOAD
293   value: /usr/local/lib/libmimalloc.so
294   resources:
295     requests:
296       cpu: 100m
297       memory: 90Mi
298     volumeMounts:
299       - name: webhook-cert
300         readOnly: true
301         mountPath: /usr/local/certificates/
302       - name: kube-api-access-2fd7s
303         readOnly: true
304         mountPath: /var/run/secrets/kubernetes.io/serviceaccount
305     livenessProbe:
306       httpGet:
307         path: /healthz
308         port: 10254
309         scheme: HTTP
310       initialDelaySeconds: 10
311       timeoutSeconds: 1
312       periodSeconds: 10
313       successThreshold: 1
314       failureThreshold: 5
315     readinessProbe:
316       httpGet:
317         path: /healthz
318         port: 10254
319         scheme: HTTP
320       initialDelaySeconds: 10
321       timeoutSeconds: 1
322       periodSeconds: 10
323       successThreshold: 1
324       failureThreshold: 3
325     lifecycle:
326       preStop:
327         exec:
328           command:
329             - /wait-shutdown
330     terminationMessagePath: /dev/termination-log
331     terminationMessagePolicy: File
332     imagePullPolicy: IfNotPresent
333     securityContext:
334       capabilities:

```

Рисунок 2.18 — Конфігурація поду NGINX.


```
apiVersion: v1
kind: Pod
metadata:
  name: serious-pod
  labels:
    app: serious-app
spec:
  containers:
    - name: serious-container
      image: serious-image:latest
      ports:
        - containerPort: 8080
      env:
        - name: SERIOUS_ENV_VARIABLE
          value: "serious-value"
      resources:
        requests:
          cpu: 100m
          memory: 128Mi
        limits:
          cpu: 500m
          memory: 256Mi
      volumeMounts:
        - name: serious-volume
          mountPath: /data
  volumes:
    - name: serious-volume
      emptyDir: {}
  restartPolicy: Always
```

Рисунок 2.20 — Пуста та проста конфігурація поду.

Пояснення різних секцій у цьому файлі:

- `apiVersion`, версія API Kubernetes, яку використовує ця конфігурація (у даному випадку, `v1`);
- `kind`, тип ресурсу, який створюється (у даному випадку, `Pod`);
- `metadata`, метадані, які включають ім'я `Pod` (`serious-pod`) та мітки (`app: serious-app`), що допомагають ідентифікувати та групувати `Pods`;
- `spec`, специфікація `Pod`, яка включає контейнери та їх налаштування;
- `containers`, список контейнерів у `Pod`;
- `name`, ім'я контейнера (`serious-container`);
- `image`, ім'я та тег образу Docker для цього контейнера (`serious-image:latest`);
- `ports`, список портів, які відкриті для цього контейнера (у даному випадку, порт `8080`);
- `env`, список змінних оточення для контейнера;
- `name`, ім'я змінної оточення (`SERIOUS_ENV_VARIABLE`);
- `value`, значення змінної оточення (`serious-value`);
- `resources`, обмеження та запити ресурсів для контейнера;
- `requests`, запити ресурсів, такі як `cpu` та `memory`;
- `limits`, обмеження ресурсів, такі як `cpu` та `memory`;
- `volumeMounts`, список точок монтування для томів у контейнері;
- `name`, ім'я тому, який буде змонтований (`serious-volume`);
- `mountPath`, шлях у контейнер.

Крім того, Kubernetes також використовує свої ресурси (особливо мітки) для запуску. Мітка - це пара ключових значень для наборів об'єктів. Це стосується кількох інших концепцій, таких як робота над групою динамічних об'єктів та іншими концепціями членів групи, таких як копіювання контролерів, копії та послуг. Існують `NXN` -з'єднання між об'єктами та мітками. Кожен об'єкт може мати кілька мітків, і кожен об'єкт може бути застосований до різних об'єктів. На дизайні етикетки є деякі обмеження. Кожна мітка на об'єкті повинна мати унікальний ключ. Клавیشі тегів повинні бути суворо граматикою. Він має дві частини: префікс та ім'я. Префікси необов'язкові. Якщо він існує, він буде

відокремлений від ризику (/), який повинен бути ефективним DNS -подоменом. Префікс повинен містити не більше 253 символів. Назва є обов'язковим і повинна містити не більше 63 символів. Назва повинна починатися і закінчуватися символами номера літери (A-Z, a-z, 0-9) і містить лише кількість літер, точок, переломів та ліній подряпин. Значення повинно відповідати тому ж встановленому ліміту, що і ім'я. Слід також зазначити, що мітка має на меті ідентифікувати об'єкти, а не додати до них будь-які метадані. Примітка повинна бути для цієї мети.

Примітка дозволяє будь-які метадані пов'язані з Kubernetes. Kubernetes просто зберігає нотатки та робить їх метадані доступними. На відміну від етикетки, не існує суворих обмежень на межі дозволених ролей та розміру. Як правило, мета-модель завжди достатньо, щоб створити власні метадані сховища та порівняти об'єкт з його метаданими.

2.2.5 Deployments у Kubernetes

Засудження Kubernetes -це високий ресурс, який дозволяє користувачам керувати ним на його кластері, оновлювати та розширювати набір стручок (POD). Бажання автоматично надавати оновлення та замінити POD, копіювати та відновити управління.

Далі наведено деякі ключові аспекти, розгорнуті Kubernetes.

Умови

Розгортання дозволяє користувачам визначати стан своїх контейнерів, вказати кількість зображень Docker, порту та інших налаштувань контейнера та кількість реплік (зразків) кожного контейнера.

Копія

Я хочу створити та керувати групою реплікації, яка забезпечує реплікацію стручок. Копія гарантує, що якщо ви створите новий стручок і видаляєте непотрібні стручки, якщо це необхідно, ви можете створити новий стручок, щоб забезпечити кількість контейнерів.

Оновлення та запис

Розгортання надає оновлення, такі як Rollingupdate, що дозволяє безпечно оновлювати контейнер, не зупиняючи програму. Якщо оновлення не вдається, розгортання дозволяє переключитись до попередньої версії.

ZOING

Розгортання дозволяє горизонтальні контейнери збільшення шляхом збільшення або зменшення кількості реплік. Це може бути динамічно завершено відповідно до навантаження на систему.

Само-обробка

Розгортання автоматично контролюватиме статус стручок та відновлює їх, коли він не вдається або видаляє. Це може забезпечити стабільність та наявність додатків у кластері.

Конфігурація декларації

Розгортання використовує конфігурацію декларації, що дозволяє користувачам описати статус системи, необхідний для файлів YAML або JSON. Декларуючи ресурси управління файлами, користувачі можуть контролювати зміни в конфігурації та відновити систему в попередній стан, коли це необхідно.

Ізоляція ресурсів

У розгортанні кожен стручок ізолюється від інших стручок, а ресурси відокремлюють ресурси від контейнерів в одному стручку. Це забезпечує безпеку, гнучкість та надійність.

Конфігурація та секрет

Розгортання дозволяє конфігураційним даним та секретам (наприклад, паролі та клавіші API) використовувати для контейнера за допомогою конфігурації та секретів. Це допомагає налаштувати управління та забезпечити безпеку конфіденційних даних.

Автоматична позиція

Враховуючи ресурси, обмеження та спорідненість, поважаючи автоматичне положення стручок вузла кластера з Kubernetes Scareduler.

Здоров'я та підготовка

Бажання використовувати статус та підготовчі тести для перевірки статусу контейнера та їх готовності до запитів на обслуговування.

Усі ці функції дозволяють розгортання в Kubernetes створювати надійні, стабільні та масштабні рішення для розгортання та контролю додатків.

Сумісність з іншими Kubernetes

Ви можете інтегрувати розгортання з іншими ресурсами Kubernetes, такими як вхід, послуга та стійка сума, що дозволяє користувачам створювати складні архітектури та надавати дозволи доступу до своїх програм або забезпечувати суть зберігання даних.

Політика безпеки

Розгортання може застосовуватися через стратегію безпеки кластерних стратегій Kubernetes.

Шаблон конфігурації

Розгортання може використовувати кермо для використання шаблону конфігурації, який спрощує розгортання та управління складними програмами в Kubernetes. Helm дозволяє створювати програмні пакети, що містять усі необхідні ресурси та конфігурації, що містять програми, щоб забезпечити зручне розгортання та оновлення.

Спостереження та записи

Він може інтегрувати розгортання та системи спостереження та логіки (наприклад, Prometheus, Grafana та Elasticsearch), що дозволяє збирати, аналізувати та візуалізацію показники та журнали.

Автоматичне масштабування

Kubernetes використовує горизонтальний AutoScaler (HPA) для підтримки автоматичного масштабування на основі шкали процесора або власних показників. HPA аналізує поточне навантаження на контейнер і збільшує або зменшує кількість реплік за потребою.

Завдяки всім цим можливостям, розгортання в Kubernetes забезпечує міцні, гнучкі та надійні інструменти розгортання в Раді директорів та розширення додатків, що містяться в хмарних медіа. Ці функції надають вам високу доступність, еластичність та безпеку в централізованому застосуванні Kubernetes.

Співпраця з CI/CD

Kubernetes та розгортання можуть інтегруватися з системою безперервної інтеграції та передачі (CI/CD), такими як Jenkins, Gitlab CI та Argo CD. Це дозволяє автоматично централізувати застосування процесу розгортання, тестування та оновлення, щоб швидко доставити та змінити та покращити програму для користувачів.

Різні стратегії розгортання

Крім стратегій прокатки, розгортання також дозволяє застосовувати інші стратегії розгортання, такі як сині/зелені та золоті птахи. Ці стратегії можуть допомогти вам уникнути використання оновлених програм, тим самим забезпечуючи безперебійну роботу та контролювати нову версію нових версій.

Оптимізація ресурсів

Розгортання може контролювати та оптимізувати використання ресурсів (таких як процесор та пам'ять) з контейнера та POD за допомогою таких функцій, як вимірювання та обмеження ресурсів. Це допомагає підтримувати ефективне використання ресурсів у кластері та забезпечити стабільність та продуктивність програми.

Усі ці функції та інтеграції перетворюють розгортання в Kubernetes важливим інструментом для розгортання, управління та масштабування, включаючи програми в сучасному хмарному середовищі. Вони дозволяють розробникам та менеджерам подати заявку на гнучкість, швидкість та стабільність, а також допомагають їм швидко реагувати на нові потреби та можливості, які з'являються в хмарних технологіях.

Відновлення після відмови

Розгортання дозволяє легко відновити систему після відмови або помилки. Завдяки методі та перевірці конфігурації оператора, користувачі можуть легко повернути попередню версію програми або конфігурацію для відновлення системи як умов праці.

Керівництво спрощує

Kubernetes та розгортання спрощують включення управління додатками, тим самим абстрагуючи складність розподіленої інфраструктури та кластерних

ресурсів. Це дає змогу розробникам та адміністраторам зосередитись на програмах розробки та управління, а не Турбується про низький рівень кластера та ресурси.

Як правило, розгортання в Kubernetes відіграє важливу роль у розгортанні додатків, розширення та управління. Вони надають розробникам та менеджерам потужним, гнучким та надійним інструментом для створення складних та високоякісних хмарних рішень для швидкого реагування на попит на ринок та задоволення потреб сучасних організацій.

2.2.6 Replica Sets у Kubernetes

Replica Sets - це один з основних компонентів у Kubernetes, який забезпечує стабільність та масштабованість контейнеризованих додатків. Завдяки Replica Sets, Kubernetes може автоматично створювати, керувати та моніторити кілька копій (реплік) одного і того ж контейнера, який виконує ваш додаток.

Основні характеристики та можливості Replica Sets наведено нижче.

Відновлення після відмов

Replica Sets стежать за кількістю активних реплік контейнера та відновлюють їх у разі відмови або збою. Це забезпечує стабільність та неперервність роботи вашого додатка.

Масштабування

Replica Sets дозволяють з легкістю масштабувати ваш додаток, збільшуючи або зменшуючи кількість реплік контейнера за потребами. Це може бути корисним для впорядкування відповідно до збільшення або зменшення навантаження на додаток.

Рівномірне розподілення навантаження

За допомогою Replica Sets та сервісів Kubernetes, вхідні запити можуть бути рівномірно розподілені між активними репліками контейнера. Це допомагає оптимізувати роботу додатка та забезпечити кращу продуктивність.

Самовідновлення

Якщо контейнер в репліці зазнає збою або відмови, Replica Sets автоматично створюють нову репліку для заміни непрацюючого контейнера. Це забезпечує високу доступність та стабільність вашого додатка.

Відокремлення логіки додатку від інфраструктури

Завдяки Replica Sets розробники можуть зосередитись на логіці додатку та контейнерах, не турбуючись про низькорівневі деталі управління інфраструктурою та масштабуванням. Replica Sets спрощують процес управління ресурсами, автоматизуючи створення, знищення та оновлення реплік.

Контроль стану

Replica Sets дозволяють використовувати проби стану (liveness probes) та проби готовності (readiness probes) для визначення стану реплік. Це допомагає забезпечити, що лише здорові репліки обслуговують користувачів та приймають нові запити.

Обмеження та квоти ресурсів

За допомогою Replica Sets можна встановити обмеження на використання ресурсів, таких як ЦПУ та оперативна пам'ять, для окремих контейнерів. Це дозволяє контролювати та оптимізувати використання ресурсів кластера.

Застосування оновлень

Replica Sets можуть бути використані у поєднанні з Deployments для контрольованого застосування оновлень вашого додатка. Завдяки стратегіям розгортання, таким як RollingUpdate, ваш додаток може бути оновлений з мінімальними перервами у роботі та впливом на користувачів.

Сумісність з іншими компонентами Kubernetes

Replica Sets можуть співпрацювати з іншими компонентами Kubernetes, такими як ConfigMaps, Secrets та Persistent Volumes, що дозволяє створювати більш гнучкі та налаштовувані конфігурації для вашого додатка.

Загалом, Replica Sets в Kubernetes надають набір інструментів та можливостей для забезпечення стабільності, масштабованості та відмінної продуктивності контейнеризованих додатків. Вони дозволяють розробникам та адміністраторам кластера зосередитись на важливих аспектах розробки додатків, забезпечуючи надійне та ефективне управління ресурсами кластера.

2.2.7 ReplicationController у Kubernetes

ReplicationController у Kubernetes — це компонент, який був створений для забезпечення стабільності, відновлення після відмов та масштабованості контейнерізованих додатків. Він відповідає за створення, керування та моніторинг кількох копій (реплік) контейнера, який виконує ваш додаток. Однак ReplicationController з часом став менш популярним, оскільки був замінений на більш потужний та гнучкий компонент - ReplicaSet.

Основні характеристики та можливості ReplicationController наведені нижче.

Відновлення після відмов

ReplicationController стежить за кількістю активних реплік контейнера та відновлює їх у разі відмови або збою. Це забезпечує стабільність та неперервність роботи вашого додатка.

Масштабованість

ReplicationController дозволяє масштабувати ваш додаток, збільшуючи або зменшуючи кількість реплік контейнера за потребами. Це може бути корисним для впорядкування відповідно до збільшення або зменшення навантаження на додаток.

Рівномірне розподілення навантаження

За допомогою ReplicationController та сервісів Kubernetes, вхідні запити можуть бути рівномірно розподілені між активними репліками контейнера. Це допомагає оптимізувати роботу додатка та забезпечити кращу продуктивність.

Самовідновлення

Якщо контейнер в репліці зазнає збою або відмови, ReplicationController автоматично створить нову репліку для заміни непрацюючого контейнера. Це забезпечує високу доступність та стабільність вашого додатка.

Відокремлення логіки додатку від інфраструктури

Завдяки ReplicationController розробники можуть зосередитись на логіці додатку та контейнерах, не турбуючись про низькорівневі деталі управління

інфраструктурою та масштабуванням. ReplicationController спрощує процес управління ресурсами, автоматизуючи створення, знищення та оновлення реплік.

Контроль стану

ReplicationController дозволяє використовувати проби стану (liveness probes) та проби готовності (readiness probes) для визначення стану реплік. Це допомагає забезпечити, що лише здорові репліки обслуговують користувачів та приймають нові запити.

Обмеження та квоти ресурсів

За допомогою ReplicationController можна встановити обмеження на використання ресурсів, таких як ЦПУ та оперативна пам'ять, для окремих контейнерів. Це дозволяє контролювати та оптимізувати використання ресурсів кластера.

Сумісність з іншими компонентами Kubernetes: ReplicationController може співпрацювати з іншими компонентами Kubernetes, такими як ConfigMaps, Secrets та Persistent Volumes, що дозволяє створювати більш гнучкі та налаштовувані конфігурації для вашого додатка.

Зауважте, що на сьогоднішній день ReplicationController вважається застарілим, і він був замінений на ReplicaSet. ReplicaSet має подібну функціональність, але надає більше можливостей та гнучкості, особливо коли використовується разом з Deployments. Тому рекомендується використовувати ReplicaSet та Deployments для розгортання та управління контейнеризованими додатками в Kubernetes.

2.3 Висновки до розділу

У цьому розділі проаналізовано переваги та недоліки архітектури мікросервісу. Цей дизайн вирішує та породжує проблеми

Завдяки поширенню мікросервісів визначається, як розгорнути програмне забезпечення в робочому середовищі та виявити найбільш широко використовуваний метод для використання системи контейнерування Docker.

Короткий огляд екосистеми Docker виявив переваги та проблеми його використання. Особливо враховуючи розташування контейнерів. Як показує аналіз, найпоширенішою домовленістю є продукт AWS (EKS).

Коротко окреслити характеристики архітектури Kubernetes, що вказує на те, що оригінальних слів, які виконуються за замовчуванням, недостатньо для розгортання складних систем. Зокрема, процес ініціалізації програми зазвичай невірний.

Архітектура та характеристики використання всіх компонентів можуть повністю зрозуміти обсяг існуючих проблем і максимально зробити вимоги до системи розробки.

3 РОЗРОБЛЕННЯ СИСТЕМИ ІНІЦІАЛІЗАЦІЇ ТА УПРАВЛІННЯ КОНТЕЙНЕРІВ І ЗОВНІШНІХ СЕРВІСІВ

Цей розділ описує власне розробку програмного продукту – системи ініціалізації та управління контейнерів і зовнішніх сервісів у Kubernetes кластерах. А саме, включає формування вимог до розробки, обґрунтування вибору засобів реалізації та короткий опис роботи.

3.1 Аналіз вимог до оточення

Оточення, в котрому може працювати система, це Gitlab. Робоча область також повинна містити ключі доступу до ресурсів AWS і Terraform.

Крім того, для правильного функціонування системи необхідно встановити змінні середовища з назвами:

- AWS_ACCESS_KEY_ID;
- AWS_DEFAULT_REGION;
- AWS_SECRET_ACCESS_KEY;
- CI_DEPLOY_PASSWORD;
- CI_DEPLOY_USER;
- K8S_CLUSTER_NAME;
- SERVICE_NAME.

Для авторизації необхідно створити IAM з правами адміністратора та root користувача.

3.2 Аналіз вимог до системи

Розробка вимог до програмного продукту є невід’ємною частиною будь-якого життєвого циклу програмного забезпечення. Цей етап дуже важливий, оскільки саме на цьому етапі визначається точна функціональність продукту, хоча

під час розробки в продукт часто можна вносити зміни. Іноді такі зміни настільки драматичні, що можуть майже повністю змінити сам продукт.

Вимоги компонента до ініціалізації повинні включати переваги розглянутих аналогічних рішень та їх недоліки, зазначені під час розробки, а саме:

- можливість додавати параметри конфігурації як змінні середовища;
- можливість додавати параметри конфігурації у вигляді файлів;
- правильне відслідковування статусу;
- можливість додавати або видаляти параметри конфігурації без зміни кодових баз програми та зображень контейнерів;
- відсутність додаткового навантаження на компоненти системи Kubernetes;
- можливість використовувати окрему утиліту, що викликається з командного рядка та як допоміжний контейнер;
- легко впроваджувати існуючі та нові програми.

Виконання вищезазначених вимог зробить розроблений продукт беззаперечним лідером серед представлених на ринку аналогів.

Компонент управління повинен відповідати таким вимогам:

- надає зручний інтерфейс для контролю версій;
- успішно контролювати службу;
- витончено обробляє системні помилки та відображає відповідні повідомлення;
- взаємодіяти з Kubernetes API без будь-яких проксі чи інших компонентів;
- простий і зрозумілий інтерфейс.

Система задовольняє вищевказаним вимогам і розробка її структурної схеми представлена в наступному підрозділі.

3.3 Опис структурної схеми

Структурну схему розробки системи див. у Додатку 1. Вона складається з трьох основних підсистем:

- підсистема ініціалізації контейнера;

- підсистеми ініціалізації та управління сервісами та кластерами;
- підсистема управління контейнерами.

3.3.1 Namespaces

В Kubernetes, namespace - це віртуальна група ресурсів, яка дозволяє виділити та організувати ресурси кластеру. Namespace дозволяє відділити кластер на кілька віртуальних кластерів, що дозволяє розділити ресурси між різними користувачами, командами та проектами.

У кожному namespace в Kubernetes можна створювати та управляти різними ресурсами, такими як розгортання (deployments), служби (services), набори реплік (replication sets) та багато інших. Кожен ресурс можна створити в конкретному namespace, а згодом звернутися до нього через ім'я ресурсу, яке містить ім'я namespace.

Namespace дозволяє забезпечити ізольоване середовище для кожного проекту або команди, що дозволяє запобігти конфліктам імен, а також забезпечити більшу безпеку та контроль доступу до ресурсів. Крім того, namespace дозволяє зменшити навантаження на кластер Kubernetes, оскільки кожен namespace має свій власний обмежений діапазон IP-адрес, який дозволяє кожному namespace використовувати власні мережеві ресурси та відмовлятися від обмежень, пов'язаних зі спільним використанням мережевих ресурсів з іншими namespace. Крім зазначеного вище, namespace може використовуватися для керування використанням ресурсів та обмеження їх доступу в межах кластеру. За допомогою namespace можна налаштувати обмеження доступу до ресурсів та API-інтерфейсів, які надають доступ до них, а також обмеження кількості ресурсів, які можуть бути створені в межах namespace.

Namespace також може бути використаний для розділення середовищ, таких як розробка, тестування та виробництво, що дозволяє кожному середовищу мати

свій власний набір ресурсів та дозволяє керувати ресурсами відповідно до потреб кожного середовища.

Для кращого керування Kubernetes-кластером та його ресурсами, кожен користувач може створювати власні namespaces та налаштовувати їх для виконання своїх завдань. Namespace дозволяє зробити кластер більш гнучким та ефективним, що робить його більш підходящим для використання в складних середовищах, де потрібно розподіляти ресурси між декількома проектами та командами.

GitLab Namespace - це механізм, який дозволяє розділяти проекти в GitLab на окремі простори імен (namespaces). Кожен простір імен має свій власний ідентифікатор та може містити один або більше проектів.

Namespace в GitLab дозволяє розділити різні типи проектів, такі як робочі групи, проекти клієнтів або внутрішні проекти компанії. Кожен простір імен може мати власні права доступу та настройки безпеки.

Крім того, GitLab дозволяє створювати підпростори імен в межах батьківського простору імен. Це дозволяє додатково організовувати проекти та групувати їх за різними критеріями, такими як функціональність, власник або часовий період.

Namespace в GitLab також дозволяє керувати доступом до ресурсів та правами користувачів. Адміністратори можуть налаштовувати права доступу до проектів та контролювати доступ до функцій GitLab для різних користувачів та груп.

Узагальнюючи, Namespace в GitLab є механізмом організації проектів та ресурсів в GitLab. Namespace дозволяє розділити проекти на окремі простори імен, керувати доступом та настройками безпеки для різних проектів та груп, що дозволяє ефективно організовувати роботу в GitLab.

3.3.2 RBAC

RBAC (Role-Based Access Control) - це механізм управління доступом, який дозволяє контролювати доступ до різних ресурсів в системі на основі ролей та дозволів.

У Kubernetes RBAC використовується для керування доступом до API-ресурсів кластеру. RBAC дозволяє визначати ролі, які мають доступ до певних ресурсів та дій в Kubernetes, і присвоювати їх користувачам та групам. Ролі можуть визначатися як для внутрішніх користувачів Kubernetes, так і для зовнішніх інтеграцій, наприклад, управлінням кластером через кінцеві точки API. Кожна роль визначається за допомогою набору дозволів (permissions), які вказують, які дії дозволено виконувати користувачам з цієї ролі. Дозволи в Kubernetes визначаються для кожного API-ресурсу окремо. Наприклад, дозвіл на створення розгортання (Deployment) дозволяє користувачеві створювати нові розгортання в кластері.

У RBAC в Kubernetes також визначаються користувачі та групи, які мають доступ до ролей та дозволів. Користувачі можуть бути визначені за допомогою внутрішніх механізмів аутентифікації Kubernetes або зовнішніх інтеграцій. RBAC дозволяє докладно налаштовувати права доступу для різних користувачів та груп, забезпечує безпеку кластеру та контроль над виконанням дій в Kubernetes.

3.3.3 VPC

VPC (Virtual Private Cloud) - це віртуальна приватна мережа, яка дозволяє створювати та налаштовувати власну ізольовану мережу в Amazon Web Services (AWS). VPC дозволяє створювати власні підмережі, маршрутизацію трафіку та налаштовувати безпеку для мережевих ресурсів в AWS.

Кожен VPC в AWS має свій власний діапазон IP-адрес, що дозволяє виділяти та керувати мережевими ресурсами в рамках даного діапазону. VPC також дозволяє створювати підмережі, які можна використовувати для розміщення різних ресурсів, таких як бази даних, сервери та інші служби.

VPC дозволяє налаштовувати правила безпеки, такі як списки доступу до мережеских портів, що дозволяє контролювати доступ до ресурсів в мережі. Також можна використовувати маршрутизацію, щоб керувати маршрутизацією трафіку між різними підмережами та іншими мережевими ресурсами в AWS.

VPC також дозволяє з'єднувати ваші VPC з іншими мережами, що дозволяє створювати гібридні рішення, які об'єднують різні мережеві середовища. Наприклад, можна створити VPN-з'єднання між вашим VPC та вашою локальною мережею, щоб дозволити забезпечити доступ до вашої AWS-мережі з вашої локальної мережі.

Узагальнюючи, VPC в Amazon Web Services є ключовим сервісом, який дозволяє створювати та керувати власною ізольованою мережею в AWS, що дозволяє забезпечити безпеку та контроль над мережевими ресурсами в AWS.

3.3.4 Subnet

Subnet - це підмережа, яка створюється в межах VPC (Virtual Private Cloud) в Amazon Web Services (AWS) та дозволяє розбити вашу VPC на менші логічні сегменти.

Кожен Subnet має свій власний діапазон IP-адрес, який належить до діапазону IP-адрес VPC. Subnet дозволяє визначити границі мережі та розмістити різні мережеві ресурси (наприклад, сервери, бази даних, засоби зберігання даних тощо) у відповідних мережеских сегментах.

Subnet дозволяє налаштовувати безпеку та контролювати доступ до ресурсів в мережі. Ви можете використовувати списки доступу до мережеских портів та мережеві політики безпеки для контролю доступу до ресурсів в межах підмережі.

Крім того, Subnet дозволяє налаштовувати маршрутизацію трафіку між різними Subnet в межах VPC та іншими мережевими ресурсами в AWS.

Subnet в AWS також дозволяє забезпечувати високу доступність та відказостійкість. Мережеві ресурси можуть бути розподілені між декількома Subnet в межах VPC для забезпечення резервування та відновлення роботи систем у разі відмови.

Узагальнюючи, Subnet в AWS дозволяє розбити вашу VPC на менші логічні сегменти, що дозволяє вам керувати доступом до мережеских ресурсів та забезпечувати безпеку в межах мережі. Subnet дозволяє також налаштовувати маршрутизацію трафіку та забезпечувати високу доступність та відказостійкість в AWS.

3.4 Сценарій використання системи

За допомогою системи ініціалізації та управління контейнерами і зовнішніми сервісами ви можете легко налаштувати та управляти вашим Kubernetes кластером. Ось приклад сценарію використання вашої системи:

Перевірте, що ваш Kubernetes кластер налаштований і працює правильно.

Створіть YAML-файл для деплою вашої додатку в Kubernetes кластері. Використовуйте вашу систему ініціалізації та управління контейнерами, щоб збілдити ваш додаток і згенерувати контейнер.

Застосуйте конфігурацію деплою вашого додатку до вашого Kubernetes кластеру. Використовуйте вашу систему для автоматичного виконання пакування, публікації та масштабування контейнерів.

Перевірте, що ваш додаток успішно запущений в Kubernetes кластері, виконавши команду `kubectl get pods`.

Запустіть ваш додаток, виконавши команду `kubectl apply -f deployment.yaml`. Налаштуйте ваш додаток, використовуючи Kubernetes ConfigMap або Secret, і забезпечте доступ до зовнішніх сервісів, якщо це потрібно.

Моніторте ваш додаток, використовуючи Kubernetes Dashboard або інші інструменти моніторингу.

У разі необхідності змініть конфігурацію додатку або Kubernetes кластеру та виконайте оновлення деплою, використовуючи вашу систему ініціалізації та управління контейнерами.

Це лише загальний сценарій, і ваша система може мати багато додаткових функцій та можливостей для управління контейнерами та Kubernetes кластером.

Додатково до описаного сценарію використання системи ініціалізації та управління контейнерами та Kubernetes кластером, можна зазначити наступні пункти:

Використовуйте вашу систему для автоматичного розгортання та масштабування контейнерів у Kubernetes кластері. Наприклад, ви можете налаштувати автоматичне масштабування горизонтально або вертикально залежно від навантаження.

Налаштуйте моніторинг та логування ваших контейнерів та Kubernetes кластеру. Використовуйте вашу систему для збору та аналізу логів та метрик, щоб відстежувати стан вашого додатку та Kubernetes кластеру.

Застосуйте заходи безпеки для вашого додатку та Kubernetes кластеру. Налаштуйте права доступу та політики безпеки для вашого додатку, використовуйте TLS для захисту мережевого трафіку, налаштуйте контроль доступу до Kubernetes API та інших сервісів.

Проводьте регулярні оновлення вашого додатку та Kubernetes кластеру. Використовуйте вашу систему для автоматичного оновлення вашого додатку та Kubernetes кластеру до останніх версій та патчів безпеки.

Налаштуйте резервне копіювання та відновлення для вашого додатку та Kubernetes кластеру. Використовуйте вашу систему для створення резервних копій додатку та Kubernetes кластеру та налаштування процедури відновлення в разі виникнення проблем.

Ці доповнення можуть збільшити безпеку та надійність вашого додатку та Kubernetes кластеру та забезпечити більш ефективне управління вашими контейнерами та зовнішніми сервісами.

У додатку 1 наведена схема використання системи. На схемі зображені можливі сценарії використання системи в різних ситуаціях. Слід зазначити, що діаграми використання точно описують можливості використання системи, не вдаючись у деталі реалізації.

3.5 Вибір та обґрунтування сервісів та засобів реалізації

Для реалізації поставлених завдань необхідно обирати засоби, які зараз активно підтримуються та мають хороший потенціал для розвитку та використання. Це дуже важливо, оскільки застарілі програмні комплекси (від англійського «legacy») часто створюють масу проблем розробникам і роботі програмного забезпечення. Такі проблеми можуть включати загрози безпеці та неможливість використання застарілого програмного забезпечення в новіших версіях операційної системи

3.5.1 Вибір мов програмування

Вибір мови програмування для реалізації програмного забезпечення є надзвичайно важливим фактором. Сьогодні існує велика кількість мов з різними функціями, використанням і популярністю.

Однак ви повинні почати з вибору мови, якою буде реалізований ваш проект, враховуючи існуючі обмеження

Було обрано такі мови програмування як:

- YAML;
- HCL;
- BASH.

Не можна стверджувати, що YAML, HCL та Bash є найкращими мовами програмування, так як кожна мова має свої особливості та використовується для різних завдань. Однак, ці мови мають свої переваги та використовуються в певних областях програмування, про які можна сказати наступне.

YAML є мовою для структурованого представлення даних та конфігурації. Він дозволяє легко створювати та змінювати конфігураційні файли, такі як файли деплою для Kubernetes, та є одним з основних форматів для збереження налаштувань в інструментах DevOps. YAML також має читабельний синтаксис, що полегшує зрозуміння та редагування файлів конфігурації.

HCL (HashiCorp Configuration Language) є мовою для опису конфігурацій HashiCorp-інструментів, таких як Terraform та Consul. HCL також є структурованою мовою та дозволяє легко описувати інфраструктуру, яка повинна бути створена або змінена, та забезпечує читабельність та легку зрозумілість.

Bash є мовою командного рядка, що дозволяє легко автоматизувати повторювані завдання на Unix-подібних операційних системах. Bash можна використовувати для створення скриптів, які автоматично виконують завдання, такі як резервне копіювання, очищення старих файлів або налаштування середовища розробки. Bash також має вбудовану підтримку управління файлами та каталогами, змінних середовища, аргументів командного рядка, циклів та умовних операторів, що робить його потужним інструментом для автоматизації повторюваних завдань.

Отже, YAML, HCL, та Bash мають свої переваги та використовуються в різних областях програмування, але не можна стверджувати, що вони є найкращими мовами програмування загалом. Особливо важливо зазначити, що кожна мова має свої обмеження та використовується для різних завдань.

Окрім того, існує багато інших мов програмування, які також можуть мати свої переваги в певних сферах програмування. Наприклад, Python має широке застосування у сфері науки даних та машинного навчання, а C++ є популярною мовою програмування для високопродуктивних систем, таких як вбудовані системи та графічні додатки.

Отже, вибір мови програмування залежить від завдання, яке потрібно вирішити, а також від індивідуальних вподобань та досвіду розробника. Важливо розуміти переваги та обмеження різних мов програмування та використовувати їх у відповідності з потребами проекту.

3.5.2 Вибір середовища програмування

Gitlab IDE - це інтегроване середовище розробки, яке забезпечує повний цикл розробки програмного забезпечення в одній платформі, починаючи з управління проектом та контролю версій, і закінчуючи CI/CD та моніторингом.

Ось декілька переваг використання Gitlab IDE.

Зручність

Gitlab IDE дозволяє здійснювати всі етапи розробки програмного забезпечення в одній платформі, що дуже зручно. Ви можете створювати проекти, редагувати код, створювати запити на злиття, налаштовувати CI/CD і моніторити проект, не виходячи з платформи Gitlab.

Інтеграція

Gitlab IDE інтегрується з багатьма іншими інструментами розробки, такими як Jira, Slack, Jenkins, Kubernetes та інші. Це дозволяє спростувати процес розробки та зменшувати час на конфігурацію.

Контроль версій

Gitlab IDE забезпечує повний цикл контролю версій, включаючи можливість створення гілок, злиття гілок та відновлення до попередніх версій. Ви можете легко відстежувати зміни та управляти ними в Gitlab.

CI/CD

Gitlab IDE має вбудований CI/CD, що дозволяє автоматизувати процеси тестування та розгортання вашого програмного забезпечення. Ви можете легко створювати пайплайни CI/CD без виходу з платформи Gitlab.

Моніторинг

Gitlab IDE дозволяє моніторити ваші проекти та метрики за допомогою вбудованої панелі моніторингу. Ви можете відслідковувати роботу вашого програмного забезпечення та виконувати аналіз даних.

Безпека

Gitlab IDE має вбудовані інструменти безпеки, такі як автоматична перевірка коду на вразливості та можливість використання статичного аналізу коду. Це дозволяє підвищити рівень безпеки програмного забезпечення та зменшити можливість виникнення помилок.

Отже, Gitlab IDE є потужним інструментом розробки, який забезпечує повний цикл розробки програмного забезпечення в одній платформі та має вбудовані інструменти безпеки. Використання Gitlab IDE дозволяє підвищити ефективність та якість розробки, зменшити час на конфігурацію та забезпечити безпеку програмного забезпечення.

3.5.3 Засоби для розробки

Для процесу розробки було використано:

- Gitlab;
- Docker;
- Terraform;
- AWS;
- Gitlab;
- Runners;
- Gitlab;
- CI/CD;
- Gitlab Registry;
- Global Secret Values;
- Docker self-contained images.

GitLab Runner - це вільно поширюваний відкритий інструмент, який дозволяє виконувати автоматизовані процеси CI/CD у GitLab. Він підключається

до GitLab та виконує задачі CI/CD, такі як збірка, тестування, створення пакетів та розгортання.

GitLab Runner може бути встановлений на будь-якому сервері, який може бути доступний для GitLab, включаючи локальні сервери, облачні платформи та віртуальні машини. Runner може запускати різні типи задач, включаючи Docker-контейнери, шел-скрипти та бінарні файли.

GitLab Runner можна налаштувати для виконання завдань на різних платформах та середовищах, що дозволяє вам виконувати процеси CI/CD відповідно до ваших потреб. Використання GitLab Runner дозволяє автоматизувати процеси CI/CD та підвищити ефективність розробки, зменшити час на вирішення проблем та підвищити якість програмного забезпечення.

GitLab CI/CD - це система автоматичної збирання, тестування та розгортання програмного забезпечення. Вона дозволяє автоматизувати процеси CI/CD від початкового коду до розгортання продукту на продуктивному середовищі.

GitLab CI/CD дозволяє використовувати різні типи тестування, такі як юніт-тести, функціональне тестування та інтеграційне тестування. Крім того, GitLab CI/CD може бути налаштована для автоматичного розгортання програмного забезпечення на будь-якому середовищі, включаючи тестові сервери, облікові записи та продуктивні сервери.

GitLab CI/CD має вбудовані інструменти для автоматизації процесів CI/CD, такі як пайплайни, GitLab Runner та інші. Вона також підтримує інтеграцію з іншими інструментами, такими як Docker, Kubernetes та Helm.

Використання GitLab CI/CD дозволяє автоматизувати процеси CI/CD та зменшити час на конфігурацію та розгортання програмного забезпечення. Крім того, GitLab CI/CD дозволяє підвищити якість програмного забезпечення та забезпечити безпеку від початкового коду до продуктивного середовища.

GitLab Container Registry - це вбудована у GitLab функція, яка дозволяє зберігати, керувати та розповсюджувати Docker-образи та інші типи контейнерів.

GitLab Registry дозволяє легко зберігати та керувати образами контейнерів в одній платформі разом з іншими елементами вашого проекту.

GitLab Registry може бути налаштована для зберігання образів Docker, Helm, NPM та інших контейнерів. Крім того, вона може бути інтегрована з GitLab CI/CD для автоматичного розгортання контейнерів.

GitLab Registry забезпечує безпеку образів контейнерів за допомогою функції підписування, що дозволяє перевірити, що контейнери не були змінені. Вона також забезпечує можливість керування дозволами доступу до образів та забезпечує можливість відновлення старих версій образів.

GitLab Registry дозволяє легко керувати та розповсюджувати контейнери для вашого проекту та підвищує ефективність розробки. Крім того, вона забезпечує безпеку та можливість керування доступом до контейнерів для забезпечення безпеки вашого проекту.

3.5.4 Вибір підходу ініціалізації та управління кластера у AWS

AWS EKS (Amazon Elastic Kubernetes Service) - це сервіс керування контейнерами на основі Kubernetes, який надається Amazon Web Services (AWS). Для управління та ініціалізації AWS EKS з використанням GitLab та Terraform можна використовувати наступний підхід.

Створення GitLab репозиторію

Спочатку потрібно створити GitLab репозиторій, де буде зберігатися конфігураційний файл Terraform та скрипти розгортання Kubernetes.

Налаштування GitLab CI/CD

Потім потрібно налаштувати GitLab CI/CD для автоматичного розгортання Kubernetes в AWS EKS. Для цього можна використовувати GitLab Runner, який буде виконувати задачі CI/CD.

Налаштування Terraform

Потім потрібно створити конфігураційний файл Terraform, який буде описувати конфігурацію Kubernetes, яку потрібно розгорнути в AWS EKS. В конфігураційному файлі Terraform можна визначити ресурси, такі як кластер Kubernetes, вузли, мережі, зберігання тощо.

Налаштування підключення до AWS

Для того, щоб Terraform міг взаємодіяти з AWS EKS, потрібно налаштувати підключення до AWS. Для цього можна використовувати AWS CLI або налаштувати Terraform для використання AWS access keys.

Збереження конфігурації у GitLab репозиторії

Після налаштування Terraform конфігурації та підключення до AWS, потрібно зберегти конфігураційний файл Terraform у GitLab репозиторії.

Запуск GitLab CI/CD

Після збереження конфігураційного файлу Terraform у GitLab репозиторії можна запустити GitLab CI/CD, який автоматично розгорне Kubernetes в AWS EKS з використанням налаштувань Terraform.

Цей підхід дозволяє автоматизувати процеси ініціалізації та управління AWS EKS, що дозволяє підвищити ефективність та якість розробки програмного забезпечення, зменшити час на налаштування та забезпечити безпеку від початкового коду до продуктивного середовища.

Крім того, використання Terraform дозволяє створити повторно використовувані шаблони, які можна використовувати для розгортання інших середовищ в AWS EKS. Це дозволяє швидко налаштувати та розгорнути нові середовища для розробки та тестування без необхідності повторювати ручний процес налаштування.

Отже, використання GitLab та Terraform для ініціалізації та управління AWS EKS дозволяє підвищити ефективність та якість розробки, зменшити час на конфігурацію та забезпечити безпеку програмного забезпечення.

3.5.5 Вибір підходу ініціалізації та управління контейнерів

GitLab дозволяє ініціалізувати та управляти контейнерами за допомогою GitLab Container Registry та GitLab CI/CD.

Створення GitLab репозиторію: Спочатку потрібно створити GitLab репозиторій, де будуть зберігатися Docker-образи та інші типи контейнерів.

Створення Dockerfile: Потім потрібно створити Dockerfile, який описує, як буде складатися Docker-образ. Dockerfile містить інструкції для встановлення та налаштування програмного забезпечення в контейнері.

Збирання Docker-образів

Після створення Dockerfile можна використовувати Docker для збирання Docker-образів. Docker-образи потрібно зберегти в GitLab Container Registry, де вони будуть доступні для подальшого використання.

Налаштування GitLab CI/CD

Потім потрібно налаштувати GitLab CI/CD для автоматичного розгортання контейнерів. Для цього можна використовувати GitLab Runner, який буде виконувати задачі CI/CD.

Налаштування розгортання контейнерів

Потім потрібно налаштувати розгортання контейнерів в середовище, яке буде використовуватися для тестування та розробки. Для цього можна використовувати Docker Compose або Kubernetes.

Запуск GitLab CI/CD

Після налаштування GitLab CI/CD можна запустити GitLab CI/CD, який автоматично розгорне контейнери в середовищі з використанням налаштувань, які були задані в Dockerfile та конфігураційних файлах GitLab CI/CD.

Цей підхід дозволяє автоматизувати процеси ініціалізації та управління контейнерами з використанням GitLab та забезпечує безпеку від початкового коду до продуктивного середовища. Крім того, використання GitLab CI/CD дозволяє підвищити ефективність

Крім того, використання GitLab CI/CD дозволяє підвищити ефективність розробки, зменшити час випуску нових версій програмного забезпечення та забезпечує більшу автоматизацію процесів розробки.

GitLab Container Registry забезпечує безпеку образів контейнерів за допомогою функції підписування, що дозволяє перевірити, що контейнери не були змінені. Вона також забезпечує можливість керування дозволами доступу до образів та забезпечує можливість відновлення старих версій образів.

Отже, використання GitLab та GitLab Container Registry дозволяє підвищити ефективність та якість розробки, зменшити час на конфігурацію та забезпечити безпеку програмного забезпечення. Крім того, GitLab CI/CD дозволяє автоматизувати процеси розробки та підвищити їх ефективність, що дозволяє швидше випускати нові версії програмного забезпечення та підвищити якість розробки.

3.5.6 Вибір підходу ініціалізації та управління зовнішніх сервісів

Ініціалізація та управління зовнішніми сервісами AWS з використанням GitLab та Terraform може бути здійснена за допомогою наступних кроків: Створення GitLab репозиторію: Спочатку потрібно створити GitLab репозиторій, де буде зберігатися конфігураційний файл Terraform та скрипти розгортання зовнішніх сервісів AWS.

Налаштування GitLab CI/CD

Потім потрібно налаштувати GitLab CI/CD для автоматичного розгортання зовнішніх сервісів AWS. Для цього можна використовувати GitLab Runner, який буде виконувати задачі CI/CD.

Налаштування Terraform

Потім потрібно створити конфігураційний файл Terraform, який буде описувати конфігурацію зовнішніх сервісів AWS, яку потрібно розгорнути. В

конфігураційному файлі Terraform можна визначити ресурси, такі як бази даних, мережі, віртуальні машини, зберігання тощо.

Налаштування підключення до AWS

Для того, щоб Terraform міг взаємодіяти з AWS, потрібно налаштувати підключення до AWS. Для цього можна використовувати AWS CLI або налаштувати Terraform для використання AWS access keys.

Збереження конфігурації у GitLab репозиторії

Після налаштування Terraform конфігурації та підключення до AWS, потрібно зберегти конфігураційний файл Terraform у GitLab репозиторії.

Запуск GitLab CI/CD

Після збереження конфігураційного файлу Terraform у GitLab репозиторії можна запустити GitLab CI/CD, який автоматично розгорне зовнішні сервіси AWS з використанням налаштувань Terraform.

Цей підхід дозволяє автоматизувати процеси ініціалізації та управління зовнішніми сервісами AWS з використанням GitLab та Terraform, що дозволяє підвищити ефективність та якість розробки, зменшити час на налаштування та забезпечити безпеку від початкового коду до продуктивного середовища.

Крім того, використання Terraform дозволяє створити повторно використовувані шаблони, які можна використовувати для розгортання інших середовищ в AWS. Це дозволяє швидко налаштувати та розгорнути нові середовища для розробки та тестування без необхідності повторювати ручний процес налаштування.

Окрім того, GitLab CI/CD дозволяє автоматизувати процеси розгортання зовнішніх сервісів AWS, що дозволяє підвищити ефективність розробки, зменшити час випуску нових версій програмного забезпечення та забезпечує більшу автоматизацію процесів розробки.

Отже, використання GitLab та Terraform для ініціалізації та управління зовнішніми сервісами AWS дозволяє підвищити ефективність та якість розробки, зменшити час на конфігурацію та забезпечити безпеку програмного забезпечення.

Крім того, GitLab CI/CD дозволяє автоматизувати процеси розгортання зовнішніх сервісів AWS, що дозволяє підвищити ефективність розробки та забезпечує більшу автоматизацію процесів розробки.

3.6 Короткий опис програми ініціалізації контейнерів

Давайте розглянемо приклад YAML-файлу для ініціалізації контейнерів на GitLab. Приклад YAML-файлу для ініціалізації контейнерів зображено на рисунок 3.1.

```
1  image: docker:latest
2
3  services:
4  | - docker:dind
5
6  stages:
7  | - build
8  | - test
9  | - deploy
10
11 variables:
12 |   IMAGE_NAME: my-image
13 |   CONTAINER_NAME: my-container
14
15 before_script:
16 | - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD $CI_REGISTRY
17
18 build:
19 |   stage: build
20 |   script:
21 |     - docker build -t $CI_REGISTRY_IMAGE/$IMAGE_NAME:$CI_COMMIT_SHA .
22 |     - docker push $CI_REGISTRY_IMAGE/$IMAGE_NAME:$CI_COMMIT_SHA
23
24 test:
25 |   stage: test
26 |   script:
27 |     - docker run -d --name $CONTAINER_NAME $CI_REGISTRY_IMAGE/$IMAGE_NAME:$CI_COMMIT_SHA
28 |     - sleep 5
29 |     - docker ps | grep $CONTAINER_NAME
30
31 deploy:
32 |   stage: deploy
33 |   script:
34 |     - kubectl apply -f deployment.yaml
35
```

Рисунок 3.1 — YAML-файлу для ініціалізації контейнерів.

Ця програма складається з наступних елементів:

- `image`, вказує, що для виконання процесів будуть використовуватись Docker-контейнери, які базуються на останній версії Docker;
- `services`, вказує, що для виконання Docker-контейнерів буде використовуватись демон Docker (`docker:dind`);
- `stages`, перелічує етапи, які будуть виконуватись в процесі розгортання;
- `variables`, визначає змінні, які використовуються в процесі розгортання, такі як ім'я контейнера та ім'я образу;
- `before_script`, визначає скрипти, які виконуються перед виконанням кожного етапу;
- `build`, визначає етап збирання Docker-образу, його побудову та завантаження до GitLab Container Registry;
- `test`, визначає етап тестування, включаючи запуск контейнера зі збудованим образом та перевірку його роботи;
- `deploy`, визначає етап розгортання контейнера на кластері Kubernetes за допомогою файлу конфігурації `deployment.yaml`.

Таким чином, якщо виконати цю програму в GitLab CI/CD, вона буде автоматично збирати Docker-образ, запускати його для тестування, а потім розгорнути контейнер на кластері Kubernetes.

Крім того, у YAML-файлі використовуються змінні, які дозволяють динамічно змінювати значення в залежності від умов. Наприклад, змінна `$CI_COMMIT_SHA` використовується для генерації унікального ідентифікатора образу Docker на основі останньої версії коміту GitLab.

До того ж, ця програма може бути легко налаштована для використання з іншими сервісами, такими як Amazon Web Services, Microsoft Azure, Google Cloud, Docker Swarm та інші. За допомогою GitLab CI/CD можна автоматизувати процеси розгортання контейнерів на будь-яких хмарних платформах.

Крім того, GitLab CI/CD дозволяє розгорнути багато контейнерів одночасно, що забезпечує ефективне використання ресурсів та швидке розгортання програмного забезпечення. Крім того, GitLab CI/CD дозволяє налаштовувати різні

етапи взаємодії між контейнерами та іншими сервісами, що дозволяє динамічно змінювати конфігурацію та оптимізувати розгортання.

3.8 Короткий опис системи ініціалізації та управління кластера

Розглянемо приклад YAML-файлу для ініціалізації та управління кластером Amazon EKS з використанням Terraform у GitLab. Рисунок 4.2

```

1  image: hashicorp/terraform:light
2
3  stages:
4    - plan
5    - apply
6
7  variables:
8    AWS_DEFAULT_REGION: us-east-1
9    TF_VAR_cluster_name: my-eks-cluster
10
11 before_script:
12   - echo "AWS_ACCESS_KEY_ID=$AWS_ACCESS_KEY_ID AWS_SECRET_ACCESS_KEY=$AWS_SECRET_ACCESS_KEY" >> ~/.env
13   - terraform init
14
15 plan:
16   stage: plan
17   script:
18     - terraform plan -out=tfplan
19
20 apply:
21   stage: apply
22   script:
23     - terraform apply tfplan
24

```

Рисунок 4.2 — Приклад YAML-файлу для ініціалізації та управління кластером Amazon EKS з використанням Terraform у GitLab.

Розглянемо приклад YAML-файлу для ініціалізації та управління кластером Amazon EKS з використанням Terraform у GitLab:

image: hashicorp/terraform:light

stages:

- plan

- apply

variables:

AWS_DEFAULT_REGION: us-east-1

TF_VAR_cluster_name: my-eks-cluster

before_script:

```
- echo "AWS_ACCESS_KEY_ID=$AWS_ACCESS_KEY_ID
AWS_SECRET_ACCESS_KEY=$AWS_SECRET_ACCESS_KEY" >> ~/.env
- terraform init
```

plan:

stage: plan

script:

```
- terraform plan -out=tfplan
```

apply:

stage: apply

script:

```
- terraform apply tfplan
```

Ця програма складається з наступних елементів:

- *image*, вказує, що для виконання процесів будуть використовуватись контейнери з Terraform;
- *stages*, перелічує етапи, які будуть виконуватись в процесі розгортання;
- *variables*, визначає змінні, які використовуються в процесі розгортання, такі як ім'я кластера та регіон AWS;
- *before_script*, визначає скрипти, які виконуються перед виконанням кожного етапу;
- *plan*, визначає етап планування, що дозволяє перевірити правильність конфігурації кластера;
- *apply*, визначає етап застосування, який розгортає кластер EKS в AWS.

Ця програма працює таким чином.

Контейнер з Terraform завантажується на GitLab.

В залежності від налаштувань змінних, програма підключається до AWS, використовуючи відповідні ключі доступу.

Після підключення програма ініціалізує Terraform.

На етапі `plan` програма перевіряє правильність конфігурації кластера та генерує план розгортання.

На етапі `apply` програма застосовує згенерований план та розгортає кластер EKS на AWS.

Якщо програма успішно виконається, то кластер EKS буде розгорнутий у короткий час.

Крім того, ця програма може бути легко налаштована для використання з іншими хмарними платформами, такими як Google Cloud Platform, Microsoft Azure та інші. За допомогою Terraform та GitLab CI/CD можна автоматизувати процеси розгортання та управління інфраструктурою на будь-яких хмарних платформах.

Крім того, програма може бути доповнена додатковими етапами, наприклад:

- етап видалення кластера EKS, що дозволяє легко видалити кластер після завершення роботи;
- етап планування та розгортання додаткових ресурсів, таких як Amazon RDS, Elastic Load Balancer, S3 Bucket та інші.

Це дозволяє забезпечити повний контроль над управлінням інфраструктурою та забезпечити автоматичне розгортання та управління комплексними сервісами на хмарних платформах.

3.9 Короткий опис системи ініціалізації та управління сервісами

Розглянемо приклад YAML-файлу для ініціалізації та управління сервісами AWS з використанням Terraform у GitLab, приклад зображено на рисунку 4.3.

```

1  image: hashicorp/terraform:light
2
3  stages:
4    - plan
5    - apply
6
7  variables:
8    AWS_DEFAULT_REGION: us-east-1
9    TF_VAR_vpc_id: "vpc-0123456789abcdefg"
10   TF_VAR_subnet_id: "subnet-0123456789abcdefg"
11   TF_VAR_bucket_name: "my-s3-bucket"
12   TF_VAR_dynamodb_table_name: "my-dynamodb-table"
13
14  before_script:
15    - echo "AWS_ACCESS_KEY_ID=$AWS_ACCESS_KEY_ID AWS_SECRET_ACCESS_KEY=$AWS_SECRET_ACCESS_KEY" >> ~/.env
16    - terraform init
17
18  plan:
19    stage: plan
20    script:
21      - terraform plan -out=tfplan
22
23  apply:
24    stage: apply
25    script:
26      - terraform apply tfplan
27

```

Рисунок 4.3 — Приклад YAML-файлу для ініціалізації та управління сервісами AWS з використанням Terraform у GitLab.

Ця програма складається з наступних елементів:

- `image`, вказує, що для виконання процесів будуть використовуватись контейнери з Terraform;
- `stages`, перелічує етапи, які будуть виконуватись в процесі розгортання;
- `variables`, визначає змінні, які використовуються в процесі розгортання, такі як ID VPC, ID Subnet, ім'я S3 Bucket, назва таблиці DynamoDB тощо;
- `before_script`, визначає скрипти, які виконуються перед виконанням кожного етапу;
- `plan`, визначає етап планування, що дозволяє перевірити правильність конфігурації сервісів AWS;
- `apply`, визначає етап застосування, який розгортає сервіси AWS.

Розглянемо яким чином працює ця програма.

Контейнер з Terraform завантажується на GitLab.

В залежності від налаштувань змінних, програма підключається до AWS, використовуючи відповідні ключі доступу.

Після підключення програма ініціалізує Terraform.

На етапі plan програма перевіряє правильність конфігурації сервісів AWS та генерує план розгортання.

На етапі apply програма застосовує згенерований план та розгортає сервіси AWS на основі визначених змінних, таких як VPC ID, Subnet ID, ім'я S3 Bucket, назва таблиці DynamoDB тощо.

Ця програма може бути легко налаштована для використання з іншими сервісами AWS, такими як Elastic Load Balancer, Amazon RDS, Amazon SNS, Amazon SQS та інші. За допомогою Terraform та GitLab CI/CD можна автоматизувати процеси розгортання та управління інфраструктурою на AWS.

Крім того, програма може бути доповнена додатковими етапами, наприклад:

- етап видалення сервісів AWS, що дозволяє легко видалити сервіси після завершення роботи;
- етап планування та розгортання додаткових ресурсів, таких як Amazon CloudFront, Amazon CloudWatch, Amazon API Gateway та інші.

Це дозволяє забезпечити повний контроль над управлінням інфраструктурою та забезпечити автоматичне розгортання та управління комплексними сервісами на AWS.

3.10 Висновки до розділу

Таким чином, розроблена система відповідає встановленим критеріям розвитку. Впровадження системи в реальні проекти показує підвищення ефективності. Про це свідчать такі фактори:

- прискорити процес;
- зменшити витрати на обслуговування кластера;
- спростити використання техніки;
- економити час;
- поліпшення безпеки інфраструктури.

Слід зазначити, що відгуки про роботу системи також були позитивними, що свідчить про правильність обраних векторів розвитку системи. Система має потенціал для широкого використання. Крім того, компоненти, вибрані для впровадження системи, були дуже вдалимими, оскільки більшість розробників у групі впровадження змогли оцінити програмний код і, сподіваюся, внести виправлення для майбутньої роботи системи.

4 АНАЛІЗ ЕФЕКТИВНОСТІ РОБОТИ СИСТЕМИ

Ефективність системи оцінювали шляхом порівняння продуктивності команди розробників протягом двох циклів розробки однакової тривалості. Інженери використовують традиційні методи для повсякденної роботи з ініціалізації та керування контейнерами, сервісами та кластером на першому етапі розробки, а потім впроваджують систему розробки, яку команда використовує на наступних етапах розробки.

4.1 Ініціалізація та управління кластеру та сервісів

Після впровадження системи досить ефективним проявило себе, а саме тим що ініціалізація пройшла швидко та без помилок. Через те що система перевіряє налаштування та конфігураційні файли на самому початку роботи результати досягались швидше та без помилок. Також система зменшила час отримання результатів та майже замінила людську роботу чим самим скоротила час.

В той час, як з використанням традиційних способів, були допущені помилки при конфігурації та ініціалізації що призводило до затримок, складнощів з розгортанням, що у свою чергу призводило до відкладання реалізацій та розробки проектів.

Такий підхід забезпечує більш коректне функціонування середовища, що позитивно позначається на стабільності системи. Крім того, існує велика ймовірність того, що система буде реалізована в середовищі, яке забезпечує фактичний запуск проекту, а не лише в тестовому кластері проекту.

Крім того, розроблений спосіб завантаження конфігурації робить систему більш безпечною. Також слід зазначити, що зменшено кількість звернень до операційної групи щодо зміни конфігурації середовища, оскільки користувачі можуть виконувати такі завдання самостійно за допомогою зручного веб-інтерфейсу.

4.2 Ініціалізація та управління контейнерів

Після введення в систему той факт, що сигнали системи почали оброблятися правильно, виявився дуже ефективним. Таким чином, кластер своєчасно попереджається про завершення роботи і встигає завершити обробку поточних викликів, не приймаючи нових запитів від клієнтів.

Під час запуску контейнерів традиційними методами часто виникають помилки під час налаштування зображень контейнерів і отримання контейнерів. Лише останній сигнал завершується, викликаючи активне завершення роботи, що, у свою чергу, призводить до переривання обробки запиту та отримання клієнтом відповідного коду помилки замість очікуваної відповіді.

Такий підхід забезпечує більш коректну роботу тестового середовища, що позитивно впливає на стабільність системи розробки. Крім того, дуже ймовірно, що цю автоматизовану систему буде реалізовано в середовищі, яке забезпечує реальну роботу користувача, а не лише в тестовому кластері проекту.

Крім того, розроблений метод завантаження параметрів конфігурації робить систему більш безпечною, оскільки користувачі без суперправ доступу не можуть переглядати значення параметрів конфігурації, якщо входять у робочий контейнер у кластері Kubernetes.

Також слід зазначити, що зменшено кількість звернень до команди щодо зміни конфігурації версій контейнера, оскільки користувачі можуть виконувати такі завдання самостійно за допомогою зручного веб-інтерфейсу.

Досить суттєвим недоліком мікросервісної архітектури є те, що дуже складно тестувати системи, побудовані з використанням такого підходу. Розроблена автоматизована система дозволяє розгортати нові системні середовища

Виконайте певні тести або надайте новим потенційним клієнтам доступ до свого тестового середовища на 30% швидше, ніж традиційні методи. У результаті команда тестувальників змогла попрацювати над приблизно на 30% більше завдань на другому етапі розробки, ніж на першому етапі розробки.

Крім того, запровадження динамічного робочого середовища призвело до приблизно 30% економії витрат, пов'язаних із обслуговуванням тестового кластера Kubernetes. Динамічне середовище – це середовище, яке можна знищити під час простою команди тестування та відтворити до початку наступного робочого дня. Розроблена система дозволяє легко і швидко налаштовувати нові середовища інтегрованих систем.

4.3 Висновки до розділу

Таким чином, розроблена система відповідає встановленим критеріям розвитку. Впровадження системи в реальний проект показало зростання продуктивності команди. Про це свідчать такі фактори:

- 1) надання більшої автономії розробникам;
- 2) прискорити процес тестування ПЗ;
- 3) зниження витрат на підтримку тестових кластерів;
- 4) покращення безпеки інфраструктури.

Слід зазначити, що відгуки про роботу системи також були позитивними, що свідчить про правильність обраного вектора розвитку автоматизованої системи. Система має потенціал для широкого використання.

Крім того, компоненти, вибрані для впровадження системи, були дуже вдалимими, оскільки більшість розробників у групі тестування впровадження змогли оцінити програмний код і сподіватися на майбутнє використання. Результати тестів роботи системи у порівнянні з роботою людей наведені у таблиці 4.1.

Таблиця 4.1 — Результати дослідів роботи системи у порівнянні з роботою людей.

Завдання/ Проект	№1	№2	№3	№4	№5	№6	№7	№8	Серед не з 100
Управління кластером/ Системою	3 год / 0.7 год	2 год / 0.5 год	5 год / 0.8 год	6 год / 1.1 год	30 год / 2 год	23 год / 1.5 год	27 год / 1 год	37 год / 1.8 год	24 год / 1.1 год
Управління контейнерами /Системою	0.1 год / 0.1 год	1 год / 0.5 год	0.17 год / 0.1 год	0.7 год / 0.1 год	3 год / 1 год	14 год / 2 год	1 год / 0.1 год	4 год / 0.3 год	4 год / 0.5 год
Управління сервісами/ Системою	1 год / 0.7 год	0.8 год / 0.5 год	3 год / 0.8 год	4 год / 1.1 год	- / -	12 год / 1.3 год	14 год / 1 год	20 год / 1.8 год	10 год / 1 год
Ініціалізація кластеру/ Системою	30 год / 0.5 год	20 год / 0.5 год	8 год / 0.5 год	33 год / 0.5 год	25 год / 0.5 год	60 год / 0.5 год	47 год / 0.5 год	50 год / 0.5 год	40 год / 0.5 год
Ініціалізація сервісів/ Системою	3 год / 0.5 год	1 год / 0.5 год	0.5 год / 0.5 год	2 год / 0.5 год	- / -	18 год / 0.5 год	3 год / 0.5 год	5 год / 0.5 год	4 год / 0.5 год
Ініціалізація контейнерів/ Системою	0.5 год / 0.5 год	1 год / 0.5 год	1 год / 0.5 год	0.7 год / 0.1 год	3 год / 0.3 год	5 год / 0.5 год	1 год / 0.1 год	3 год / 0.3 год	2 год / 0.5 год

ВИСНОВКИ

У даній магістерській дисертації розроблена система ініціалізації та управління контейнерами і зовнішніми сервісами у Kubernetes кластері.

Аналіз переваг і недоліків мікросервісного підходу до розробки програмного забезпечення, який став все більш популярним за останні кілька років, показує, що проблема оркестровки додатків, що працюють у контейнерах Docker, є актуальною. Цей факт створив величезний попит на швидкі, прості у використанні та надійні оркестрови, серед яких Kubernetes є беззаперечним лідером. Однак аналіз архітектури Kubernetes показує, що стандартного набору примітивів, який вона використовує, недостатньо для задоволення потреб більшості розробників.

Ця розробка поєднує в собі переваги аналітичних аналогів і виправляє їх недоліки, що робить цю систему дуже ефективною та простою у використанні. Зокрема, підсистема ініціалізації контейнера забезпечує високий рівень безпеки та виводить достатньо інформативних повідомлень під час виконання, що дозволяє швидко та легко виявляти помилки у вашій роботі. Підсистема управління контейнерами забезпечує постійне та безперебійне керування контейнерами з простим інтерфейсом та розгалуженим функціонуванням. Підсистема ініціалізації та управління сервісами та кластером забезпечує високий рівень безпеки, швидкості, стабільності, безперебійності та простоті використання з простим інтерфейсом.

Обрані компоненти для реалізації системи, а саме мова програмування YAML, HCL та система Gitlab є доцільним та вдалим, про що свідчить проведений вибір компонентів.

Проведений дослід показав ефективність системи. Під час проведення дослідів було встановлено, що розроблена система ініціалізації та управління контейнерами і зовнішніми сервісами у Kubernetes кластері демонструє високу продуктивність, стабільність та здатність до масштабування. Ці характеристики важливі для сучасних розподілених систем, що вимагають гнучкості та надійності.

Використання Kubernetes як основи для системи управління контейнерами та сервісами виявилося успішним рішенням, що забезпечує широкий спектр функціональних можливостей, що включають автоматичне масштабування, самовідновлення, балансування навантаження та ізоляцію ресурсів.

Використання мов програмування YAML та HCL, а також системи GitLab, дозволило створити гнучку, масштабовану та просту в обслуговуванні систему. Ці технології дозволяють розробникам легко і швидко описувати і впроваджувати потрібні конфігурації, що сприяє продуктивності роботи та якості виходу.

Система, розроблена в рамках цієї магістерської дисертації, виправдовує поставлені цілі та вимоги. Це високоефективне, гнучке та надійне рішення для управління контейнерами і сервісами у Kubernetes кластері, яке може бути адаптоване до різних обчислювальних середовищ та потреб користувачів. Вона забезпечує значне полегшення та оптимізацію процесу розгортання та управління застосунками, що працюють у контейнерах.

Система відповідає встановленим вимогам розробки. Крім того, це дозволяє легко покращувати та розширювати функціональність, якщо це буде потрібно в майбутньому.

ПЕРЕЛІК ПОСИЛАНЬ

1. Future of CRDs: Structural Schemas [Електронний ресурс] – Режим доступу до ресурсу: <https://kubernetes.io/blog/2019/06/20/crd-structural-schema/>.
2. AWS Secrets Manager secrets with Kubernetes [Електронний ресурс] – Режим доступу до ресурсу: <https://docs.aws.amazon.com/eks/latest/userguide/manage-secrets.html>.
3. Advanced Encryption Standard [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Advanced_Encryption_Standard.
4. Kubernetes and Container Security and Adoption Trends [Електронний ресурс] – Режим доступу до ресурсу: <https://www.stackrox.com/kubernetes-adoptionsecurity-and-market-share-for-containers/>.
5. Building Microservices [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/devopsasset/devopsreferencematerial/blob/master/Building_Microservices_Nginx.pdf.
6. Gitlab Documentation [Електронний ресурс] – Режим доступу до ресурсу: <https://docs.gitlab.com/>
7. Docker Documentation [Електронний ресурс] – Режим доступу до ресурсу: <https://docs.docker.com/>
8. Terraform Documentation [Електронний ресурс] – Режим доступу до ресурсу: <https://developer.hashicorp.com/terraform/docs>
9. Managing Resources for Containers [Електронний ресурс] – Режим доступу до ресурсу: <https://kubernetes.io/docs/concepts/configuration/manage-resourcescontainers/>.
10. Software as a service [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Software_as_a_service.
11. YAML [Електронний ресурс] – Режим доступу до ресурсу: <https://yaml.org/>.
12. Designing and building highly available clusters [Електронний ресурс] – Режим доступу до ресурсу: <https://docs.aws.amazon.com/whitepapers/latest/real-time-communication-on-aws/high-availability-and-scalability-on-aws.html>.

13. Best practices for building containers [Электронный ресурс] – Режим доступа до ресурсу: <https://cloud.google.com/solutions/best-practices-for-buildingcontainers>.
14. Microservices. What are microservices [Электронный ресурс] – Режим доступа до ресурсу: <https://www.redhat.com/en/topics/microservices/what-aremicroservices>.
15. The What, Why, and How of a Microservices Architecture [Электронный ресурс] – Режим доступа до ресурсу: <https://medium.com/hashmapinc/the-what-whyand-how-of-a-microservices-architecture-4179579423a9>.
16. Provider. Providers in the Terraform Registry [Электронный ресурс] – Режим доступа до ресурсу: <https://www.terraform.io/docs/providers/index.html>.
17. Continuous Security: Implementing the Critical Controls in a DevOps Environment [Электронный ресурс] – Режим доступа до ресурсу: <https://github.com/devops-asset/devops-reference-material/blob/master/continuoussecurity-implementing-critical-controls-devops-environment-36552.pdf>.
18. The State of Containers and the Docker Ecosystem [Электронный ресурс] – Режим доступа до ресурсу: https://github.com/devops-asset/devops-referencematerial/blob/master/State_of_Containers_Ruxit_compressed_V2.pdf.
19. Site Reliability Engineering. How google runs production systems [Электронный ресурс] – Режим доступа до ресурсу: <https://github.com/devopsasset/devopsreference-material/blob/master/OReilly.Site.Reliability.Engineering.2016.3.pdf>.
20. DevOps Handbook [Электронный ресурс] – Режим доступа до ресурсу: <https://github.com/devops-asset/devops-reference-material/blob/master/DevopsHandbook>.
21. Mastering Kubernetes. Automating container deployment and management [Электронный ресурс] – Режим доступа до ресурсу: [https://github.com/devopsasset/devops-reference-material/blob/master/Mastering Kubernetes - Gigi Sayfan.pdf](https://github.com/devopsasset/devops-reference-material/blob/master/Mastering_Kubernetes_-_Gigi_Sayfan.pdf).
22. Using these best practices, software development teams can use GitLab Flow to software development [Электронный ресурс] – Режим доступа до ресурсу: <https://about.gitlab.com/topics/version-control/what-are-gitlab-flow-best-practices/>

23. Trends in Applied Infrastructure & DevOps [Электронный ресурс] – Режим доступа до ресурсу: https://github.com/devops-asset/devops-referencematerial/blob/master/451_2019_Trends_in_Applied_Infrastructure_&_DevOps_SLIM.PDF.
24. Security best practices for Amazon EKS [Электронный ресурс] – Режим доступа до ресурсу: <https://docs.aws.amazon.com/eks/latest/userguide/security-best-practices.html>
25. A best practices guide for day 2 operations, including operational excellence, security, reliability, performance efficiency, and cost optimization. [Электронный ресурс] – Режим доступа до ресурсу: <https://github.com/aws/aws-eks-best-practices>.
26. EKS Best Practices Guides [Электронный ресурс] – Режим доступа до ресурсу: <https://aws.github.io/aws-eks-best-practices/>
27. Terraform Best Practices Guides [Электронный ресурс] – Режим доступа до ресурсу: <https://www.terraform-best-practices.com/>
28. Provision an EKS Cluster (AWS) [Электронный ресурс] – Режим доступа до ресурсу: <https://developer.hashicorp.com/terraform/tutorials/kubernetes/eks>
29. Terraform — Provision Amazon EKS Cluster using Terraform [Электронный ресурс] – Режим доступа до ресурсу: <https://medium.com/devops-mojo/terraform-provision-amazon-eks-cluster-using-terraform-deploy-create-aws-eks-kubernetes-cluster-tf-4134ab22c594>
30. Simple deployment to Amazon EKS [Электронный ресурс] – Режим доступа до ресурсу : <https://about.gitlab.com/blog/2018/06/06/eks-gitlab-integration/>
31. Create an Amazon EKS cluster [Электронный ресурс] – Режим доступа до ресурсу https://docs.gitlab.com/ee/user/infrastructure/clusters/connect/new_eks_cluster.html.
32. CI/CD concepts [Электронный ресурс] – Режим доступа до ресурсу: <https://docs.gitlab.com/ee/ci/introduction/>

33. Mastering continuous software development [Электронный ресурс] – Режим доступа до ресурсу: https://about.gitlab.com/webcast/mastering-ci-cd/?_gl=1*_shatdp*_ga*NDQzODA2NzQxLjE2NDM2MjQ1NTM.*_ga_ENFH3X7M5Y*MTY4NDEyNTM4NS45MS4xLjE2ODQxMjY2OTYuMC4wLjA.
34. CI/CD: Complete Guide to Continuous Integration and Delivery [Электронный ресурс] – Режим доступа до ресурсу: <https://codefresh.io/learn/ci-cd/>
35. What is a CI/CD pipeline? [Электронный ресурс] – Режим доступа до ресурсу: <https://www.redhat.com/en/topics/devops/what-cicd-pipeline>

ДОДАТОК А
Графічний матеріал