

Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет
Кафедра

Комп'ютерної інженерії та управління
Комп'ютерних інтелектуальних технологій та систем

КВАЛІФІКАЦІЙНА РОБОТА **Пояснювальна записка**

рівень вищої освіти

другий (магістерський)

Нейромережева ідентифікація нестационарних об'єктів

Виконав:

студент 2 курсу, групи КІТм-21-2

Волков О. А.

Спеціальність 123 Комп'ютерна інженерія

Тип програми освітньо-професійна

Освітня програма Комп'ютерні

інтелектуальні технології

Керівник проф. Руденко О.Г.

Допускається до захисту

Зав. кафедри

(підпис)

проф. Руденко О.Г.
(прізвище, ініціали)

2022 р.

Факультет Комп'ютерної інженерії та управління
Кафедра Комп'ютерних інтелектуальних технологій та систем
Рівень вищої освіти другий (магістерський)
Спеціальність (напрямок) 123 – Комп'ютерна інженерія
(код і назва)
Освітня програма Комп'ютерні інтелектуальні технології
(повна назва)
Харківський національний університет радіоелектроніки

ЗАТВЕРДЖУЮ:

Зав. кафедри _____
(підпис)

“ _____ ” _____ 20__ р.

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Волкову Антону Олександровичу
(прізвище, ім'я, по батькові)

1. Тема роботи Нейромережева ідентифікація нестационарних об'єктів

затверджена наказом по університету від “ 07 листопада 2022р № 1455Ст

2. Термін подання студентом роботи до екзаменаційної комісії _____

3. Вхідні дані до роботи _____

4. Перелік питань, що потрібно опрацювати в роботі _____

Програмна реалізація проекту

Висновки

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів) Презентація 10 слайдів

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Аналіз предметної області та постановка	08.11 – 13.11	
	Задачі дипломної роботи		виконано
2	Огляд сучасної літератури за напрямком	13.11 – 20.11	
	магістерської роботи		виконано
3	Вибір методів рішення	21.11 – 23.11	виконано
4	Застосування гібридних рекомендаційних	24.11 – 26.11	
	систем		виконано
5	Експериментальні дослідження	26.11 – 30.12	виконано
6	Оформлення пояснювальної записки	01.12 – 07.12	виконано
7	Оформлення графічного матеріалу	08.12 – 10.12	виконано
8	Захист проекту	19.12 – 22.12	виконано

Дата видачі завдання 07 листопада 2022 р.

Студент _____
(підпис)

Керівник
роботи

_____ проф. Руденко О.Г.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи містить: 102 стор., 22 рис., 9 табл., 2 дод., 48 джерел.

Кваліфікаційна робота присвячена дослідженню та розробці хмарної мікросервісної архітектури у Google Cloud Platform з використанням інструментів оркестрації та технології контейнеризації. Контейнеризація – методологія розробки, коли усі залежності та розроблювані файли, компоненти та саме ПЗ об'єднується у ізольоване середовище.

Актуальність даної теми полягає у необхідності безперервного інтегрування та безперервної доставки оновлень розроблюваного додатку.

Метою роботи є аналіз та розробка хмарної мікросервісної інфраструктури для дослідження систем нейроідентифікації нелінійних об'єктів за допомогою оркестратора контейнерів Kubernetes для обслуговування веб-серверу адресної книги.

Для виконання поставленої мети були виконані наступні завдання: було розроблено мікросервісну архітектуру для кластеру Kubernetes; було розроблено CI/CD інфраструктуру для контейнеризованого веб-серверу адресної книги; зроблені Helm-Chart для більш легкого розгортання додатку.

Таким чином, дана система являє собою контейнеризований веб-сервер з базою даних, яка надає можливість blue-green розгорнення та балансування вхідного трафіку кластеру.

При розроблені було використані такі технології як Docker, CircleCI, GCP, Helm, Kubernetes, Terraform.

DOCKER, CIRCLECI, GCP, HELM, KUBERNETES, TERRAFORM

Essay

An explanatory note of the qualification work to revenge: 102 st., 22 fig., 9 tab., 2 add., 48 sources.

The qualification of the work is attributed to the development of the shabby microservice architecture of the Google Cloud Platform with various orchestration tools and containerization technologies. Containerization - the methodology of expansion, if all the deposits and files are distributed, the components of the same software are combined in the isolation of the middle.

The relevance of this topic is related to the need for uninterrupted integration and uninterrupted delivery of the update of the expanded addendum.

The method of work is the analysis of tools for the development of a gloomy microservice infrastructure for the follow-up of systems of neuro-identification of non-linear objects behind an auxiliary Kubernetes container orchestrator for servicing the address book web server.

For the purpose of the installation, the following steps were taken: the microservice architecture for the Kubernetes cluster was split; CI/CD infrastructure for containerized web server address book has been split; srobleni Helm-Chart for more easy throating supplement.

In this way, this system is a containerized web server with a data base, as we hope the blue-green fire is possible and balances the input traffic to the cluster.

When expanding, such technologies as Docker, CircleCI, GCP, Helm, Kubernetes, Terraform were used

DOCKER, CIRCLECI, GCP, HELM, KUBERNETES, TERRAFORM

Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп'ютерної інженерії та управління

Кафедра Комп'ютерних інтелектуальних технологій та систем

АНОТАЦІЯ
КВАЛІФІКАЦІЙНОЇ РОБОТИ

рівень вищої освіти другий (магістерський)

Нейромережева ідентифікація нестационарних об'єктів

Виконав:

студент 2 курсу, групи: КІТм-21-2

Волков А.О.

Спеціальність 123 Комп'ютерна інженерія

Тип програми освітньо-професійна

Освітня програма Комп'ютерні інтелектуальні
технології

Керівник проф. Руденко О.Г.

2022 р.

АНОТАЦІЯ

Волков А.О. Нейромережева ідентифікація нестаціонарних об'єктів – Магістерська кваліфікаційна робота.

Актуальність теми дослідження. Актуальність даної теми полягає у створенні математичних моделей, методів, алгоритмів та програм, орієнтованих на розв'язання задач прогнозування нестаціонарних послідовностей, які є складовими сучасних комп'ютерних технологій. У дисертаційній роботі на основі отриманих теоретичних і експериментальних досліджень вирішена задача побудови нейромережових методів прогнозування нестаціонарних послідовностей, які на відміну від існуючих, прискорюють процес прогнозування та збільшують його точність в умовах апіорної і поточної невизначеності і наявності завад.

Метою роботи є аналіз та розробка хмарної мікросервісної інфраструктури для дослідження систем нейроідентифікації нелінійних об'єктів за допомогою оркестратора контейнерів Kubernetes для обслуговування веб-серверу адресної книги.

Для виконання поставленої мети були виконані наступні завдання: було проаналізовано методи теорії штучних нейронних мереж, що дозволило синтезувати нейромережові моделі та отримати процедури їх навчання, а також методи теорії оптимізації, за допомогою яких були синтезовані швидкодіючі процедури навчання; було розроблено мікросервісну архітектуру для кластеру Kubernetes; було розроблено CI/CD інфраструктуру для контейнеризованого веб-серверу адресної книги; зроблені Helm-Chart для більш легкого розгортання додатку.

Методи дослідження. В роботі використані методи теорії обчислювального інтелекту, а саме, методи теорії штучних нейронних мереж, що дозволило синтезувати нейромережові моделі та отримати процедури їх навчання; методи теорії оптимізації, за допомогою яких були синтезовані

швидкодiючі процедури навчання; методи робастного оцiнювання, на основi яких були синтезованi робастнi процедури навчання нейромережевих моделей; методи iмiтацiйного моделювання, що дозволили пiдтвердити ефективнiсть отриманих результатiв та розробити рекомендацiї щодо їх практичного використання. Експериментальнi дослiдження проводилися в лабораторних умовах i на реальних об'єктах.

Об'єкт дослiдження – процеси прогнозування нестационарних часових рядiв за допомогою дослiдження iнструментiв оркестрацiї, якi надаються хмарними провайдерами.

Предмет дослiдження – методи та моделi прогнозування нестационарних часових рядiв на основi штучних нейронних мереж.

Робота нацiлена на дослiдження iнструментiв створення хмарної серверної iнфраструктури з використанням мiкросервісної архiтектури та пiдтримання її працездатностi. У якостi оркестратора контейнерiв Docker використовується Kubernetes.

У першому роздiлi проаналiзовано та визначено аналiз предметної областi, дослiджено сучаснi пiдходи до розробки програмного забезпечення. Також детально були розглянутi рiзнi схеми пiдходу до рiшення даної задачi, схема життєвого циклу девопс, безперервна iнтеграцiя та доставка коду.

У другому роздiлi розглянули аналiз системних вимог та його типи дiяльностi. Розглянули структуру кластеру K8s. Розглянули рiзнi сценарiї використання, якi описують не тiльки взаємодiю мiж користувачем та об'єктом, але й реакцiю об'єкта на отримання окремих повiдомлень вiд користувача та отримання цих повiдомлень за межами об'єкта. Варiанти використання можуть включати вiдомостi про реалiзацiю служби та опис рiзних виняткових ситуацiй, таких як правильна обробка системних помилок.

Terraform - це iнструмент вiд Hashicorp, який допомагає декларативно управляти iнфраструктурою. У цьому випадку вам не потрiбно вручну створювати екземпляри, мережi тощо в консолi вашого хмарного провайдера; досить написати конфiгурацiю, в якiй буде окреслено, як ви бачите свою

майбутню інфраструктуру.

Ця конфігурація створюється в придатному для читання людиною текстовому форматі. Якщо потрібно змінити інфраструктуру, відредагуйте конфігурацію та натисніть кнопку. Terraform направить виклики API вашому хмарному провайдеру, щоб привести інфраструктуру у відповідність з конфігурацією, вказаною в цьому файлі.

Переміщення управління інфраструктурою до текстових файлів відкриває можливість озброїтися всіма улюбленими вихідними кодами та інструментами управління процесами, а потім переорієнтувати їх на роботу з інфраструктурою.

У третьому розділі ми підійшли саме до розробки хмарної мікросервісної архітектури після усіх підготовчих дій. У основному коді інфраструктури було реалізовано модульний опис інфраструктури. Тобто у головному файлі `main.tf` потрібно лише описати посилання на модулі і вказати змінні, що будуть використовуватися при побудові інфраструктури. Описали модулі інфраструктури.

DOCKER, CIRCLECI, GCP, HELM, KUBERNETES, TERRAFORM.

Перелік публікацій керівника та співробітників кафедри, використаних в роботі:

1. Rudenko O. et al. Developing a Multi-Step Recurrent Algorithm to Maximize the Criteria of Correntropy. *Eastern-European Journal of Enterprise Technologies*. Vol. 1. No. 4. 2021. 109.

2. Rudenko O. et al. Robust identification of non-stationary objects with nongaussian interference. *Eastern-european Journal of enterprise Technologies*. Vol. 5. No. 4. 2019. 44-52.

ЗМІСТ

Перелік умовних позначень, скорочень і термінів	12
Вступ	13
1 Аналіз предметної області	15
1.1 Дослідження сучасних підходів до розробки ПЗ	15
1.2 Принципи формування нейромережових моделей	18
1.3 Аналіз алгоритмів навчання ШНМ	20
1.4 Безперервна доставка/розгортання	22
1.5 Дослідження інструментів оркестрації, які надаються хмарними провайдерами	32
1.7 Постановка завдання	37
1.8 Висновки за розділом	40
2 Проектування хмарної мікросервісної архітектури	42
2.1 Аналіз вимог до системи	42
2.2 Проектування діаграми використання	45
2.3 Проектування інфраструктури проекту	49
2.4 Висновки за розділом	51
3 Розробка хмарної мікросервісної архітектури	53
3.1 Підготовка до розробки основної інфраструктури	53
3.2 Побудова основної інфраструктури	60
3.3 Розробка чартів Helm - Charts	63
3.4 Розробка конфігурації для CircleCI	64
3.5 Висновки за розділом	66
Висновки	68
Перелік джерел	72
Додаток А. Технічне завдання	74
Додаток Б. Вихідні скрипти Terraform	79
Додаток В. Вихідні скрипти CircleCI	82
Додаток Г. Вихідні скрипти Helm-Charts	87
Додаток Д. Презентація	95
Додаток І. Тези з конференції	101

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ПЗ – програмне забезпечення сукупність програм системи обробки інформації і програмних документів, необхідних для експлуатації цих програм. інструментальне програмне забезпечення (комп'ютерні програми, призначені для проектування, розробки, адміністрування і супроводження системного та прикладного програмного забезпечення)

ПК – персональний комп'ютер.

БД – база даних.

GCP – Google Cloud Platform.

CI – Continous integration

CD – Continous delivery/deployment

AWS – Amazon Web Services

ВСТУП

Кваліфікаційна робота присвячена дослідженню та розробці архітектур хмарних мікросервісів на платформі Google Cloud Platform з використанням інструментів оркестрування та методів контейнеризації. Контейнеризація - це методологія розробки, при якій всі залежності і файли розробки, компоненти та програмне забезпечення об'єднуються в ізольоване середовище.

Актуальність цієї теми полягає в необхідності безперервної інтеграції додатків, що розробляються, і безперервної доставки оновлень.

Метою роботи є аналіз та розробка хмарної мікросервісної інфраструктури для дослідження систем нейроідентифікації нелінійних об'єктів за допомогою оркестратора контейнерів Kubernetes для обслуговування веб-серверу адресної книги.

Для виконання поставленої мети були виконані наступні завдання: було проаналізовано методи теорії штучних нейронних мереж, що дозволило синтезувати нейромереві моделі та отримати процедури їх навчання, а також методи теорії оптимізації, за допомогою яких були синтезовані швидкодіючі процедури навчання; було розроблено мікросервісну архітектуру для кластеру Kubernetes; було розроблено CI/CD інфраструктуру для контейнеризованого веб-серверу адресної книги; зроблені Helm-Chart для більш легкого розгортання додатку.

Таким чином, система є контейнерним веб-сервером з базою даних, яка забезпечує синьо-зелене розгортання і балансування вхідного трафіку кластера.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Дослідження сучасних підходів до розробки ПЗ

Розробка (development) та експлуатація (operations) тривалий час були ізольованими модулями. Код писали програмісти, а системні адміністратори відповідали за його розгортання та інтеграцію. В рамках одного проекту фахівці працювали окремо, оскільки зв'язок між двома розрізненими сховищами було обмежено.

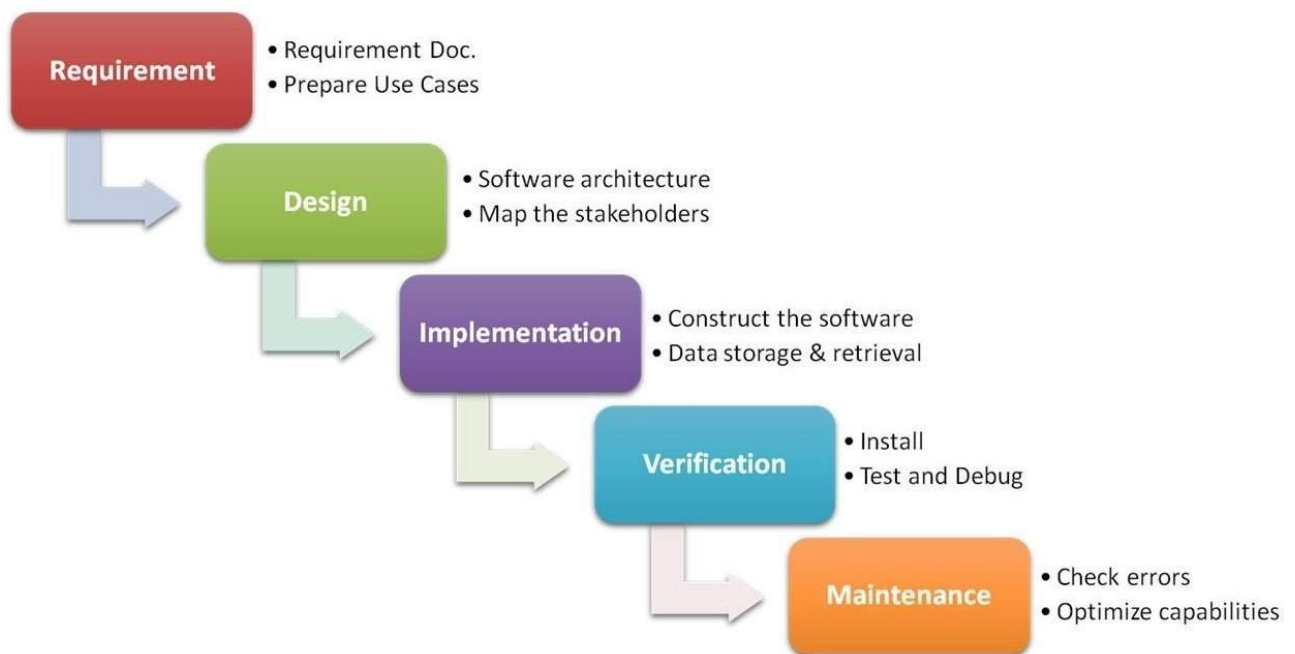


Рисунок 1.1 – Схема каскадної методології.

Цей метод працював з 1970 року, доки домінувала каскадна модель процесу розробки програмного забезпечення, відома як Waterfall. Методика передбачала послідовний перехід між етапами без перерв та повернень на попередні стадії.

Згодом Waterfall була розкритикована за недостатню гнучкість, а її основна мета – формальне управління проектом, завдавала збитків термінам, вартості та якості.

Команди потребували гнучкості під час розробки програмного забезпечення. Необхідність реалізації проекту у формі коротких "спринтів" і випуску найчастіших (від двох тижнів до двох днів) релізів зажадали нового підходу.

У 2001 році на зміну Waterfall прийшла гнучка методологія розробки або Agile. Вона включає низку підходів та практик, заснованих на чотирьох цінностях та 12 принципах «Маніфесту гнучкої розробки програмного забезпечення». Сюди також відносять SCRUM, Kanban, Lean, Feature-driven development (FDD) та інші подібні підходи.

Agile застосовується для організації роботи невеликих груп, які створюють продукт короткими ітераціями (від двох до чотирьох тижнів). Кожна ітерація має вигляд програмного проекту, який включає всі типові завдання: планування, аналіз вимог, проектування, програмування, тестування, документування. Наприкінці ітерації замовник одержує робочий продукт.

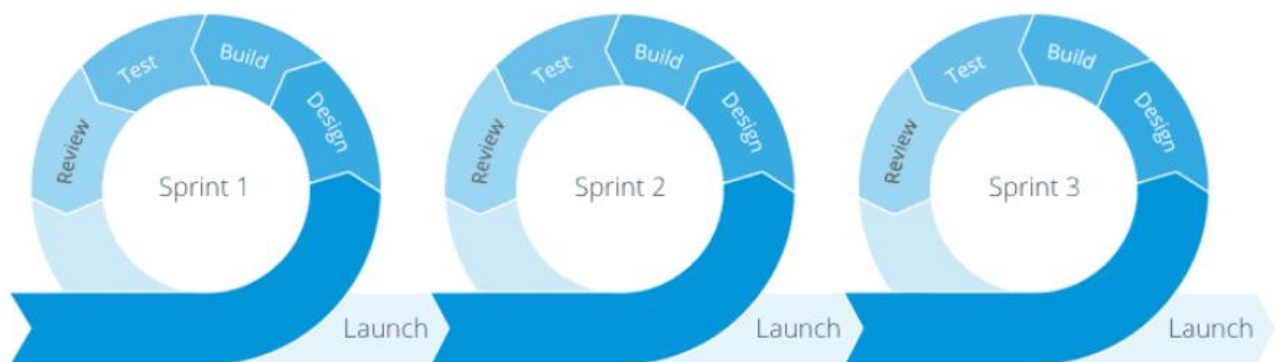


Рисунок 1.2 - Схема підходу Agile

Методологію Agile критикували за відсутність управління вимогами. Замовник може виставити нові вимоги наприкінці кожної ітерації, що суперечить архітектурі вже створеного продукту. Часті зміни та вдосконалення продукту можуть призвести до масового рефакторингу та плаваючої вартості проекту в результаті

Методологія DevOps набула широкого поширення після організованої бельгійським розробником Патріком Дебуа у 2009 році конференції DevOpsDays.

Головна ідея DevOps полягає в тому, щоб усунути переклад відповідальності на інших членів команди у великих колективах. Взаємозалежність між створенням та експлуатацією програмного забезпечення мала на меті прищепити команді нову культуру розробки продукту.

Культура DevOps припускає, що кожен із членів команди відповідальний за кінцевий результат. Базується вона на кількох основних положеннях:

Регулярне співробітництво та спілкування. Команда повинна працювати злагоджено, розуміти потреби та очікування всіх її членів.

Впровадження поступового розгортання дозволяє групам доставки випускати продукт, маючи можливість вносити оновлення та робити відкат, якщо щось не піде.

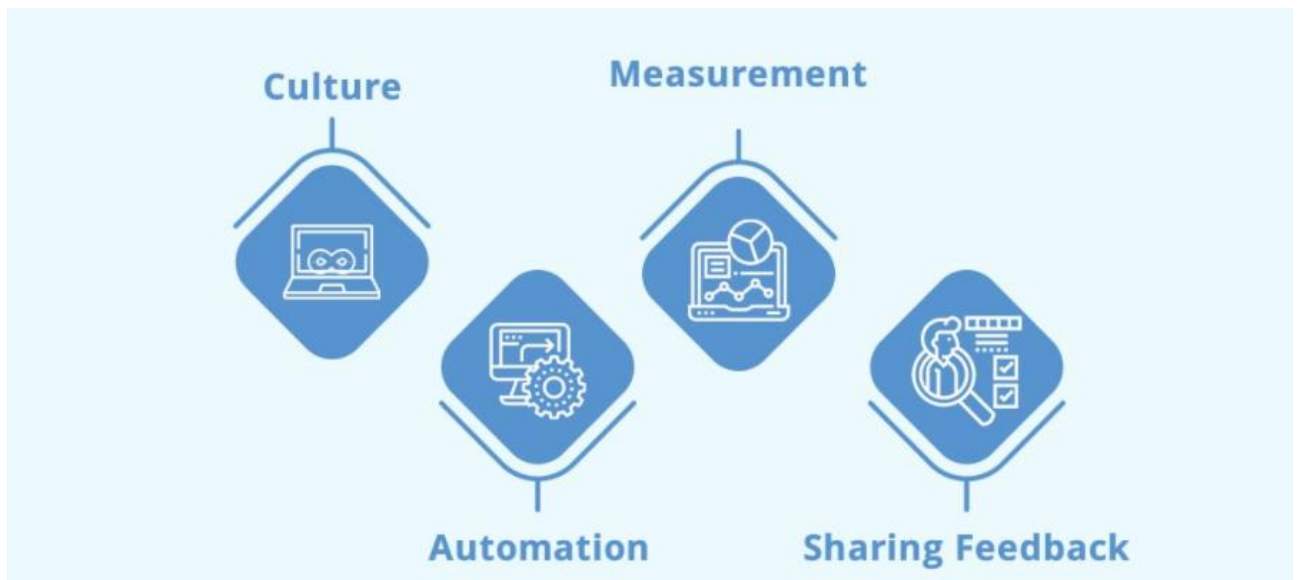
Спільна відповідальність. Усі члени команди повинні рухатися до єдиної мети та відповідати за проект рівною мірою.

Вирішення проблем на ранніх етапах. Методологія DevOps вимагає, щоб у життєвому циклі проекту завдання виконувались якнайшвидше. Це допомагає оперативно вирішувати проблеми, що виникли.

У 2010 році Деймоном Едвардсом та Джоном Віллісом була розроблена модель CAMS, ключові ідеї якої стали принципами DevOps. Відповідно до неї, розвиток DevOps йде у трьох напрямках: люди, процеси та інструменти. При цьому важливою є підтримка кожного пункту на всіх етапах розвитку.

Абревіатура CAMS розшифровується так:

- культура (culture);
- автоматизація (automation);
- вимір (measurement);
- обмін (Sharing).



1.3 - Схема моделі CAMS

Класичні бізнес-моделі в ІТ поділяють фахівців із розробки та експлуатації на дві окремі групи. До появи DevOps вони спілкувалися різними мовами, адже перед розробниками стояло завдання швидко впроваджувати інновації, а операційний персонал відповідав за підтримку стабільного середовища та інфраструктури.

Конкуруючі робочі цілі створювали між фахівцями з розробки та експлуатації непорозуміння, тому основне завдання DevOps – змінити бізнес-культуру, поділити відповідальність двох груп та поєднати їхні професійні навички.

Наслідуючи шляхи DevOps, код потрібно переводити зі стадії розробки у виробництво безперервно в автоматичному режимі, тому автоматизацію можна вважати синонімом DevOps.

В ідеалі автоматизувати потрібно майже все:

- інфраструктуру;
- випуски програмного забезпечення (software releases);
- тестування;
- розгортання;
- основні завдання щодо безпеки;
- політику угод;

- Завдання управління конфігурацією.

Автоматизація спрощує робочі процеси, скорочує кількість збоїв та відкатів, зменшує кількість помилок, що виникають при ручному налаштуванні. Підвищення ефективності, покращення продуктивності та користь для кінцевого споживача – головні переваги автоматизації.

Вимірювання необхідне постійного пропозиції цінності та поліпшень. В DevOps важливо відстежувати ключові показники, які залежать від цілей проекту.

Вимірювати потрібно показники наступних процесів:

- моніторингу та відстеження продуктивності протягом усього життєвого циклу розробки програмного забезпечення;
- збору, аналізу та надання способів реагування на зворотний зв'язок;
- аналізу помилок та способів їх запобігання;
- надання допомоги командам у роботі над спільними цілями.

DevOps сприяє розвитку лише у тому випадку, якщо конкретні показники збираються та аналізуються безперервно.

DevOps має на увазі тісне співробітництво фахівців з розробки та експлуатації. Велика увага приділяється прозорості та відкритості у колективі. Чим більше знань поширюється між співробітниками, тим більше зворотного зв'язку вони отримують – це допомагає покращити їхню роботу в цілому.

DevOps є природним продовженням гнучких підходів та підходів до безперервної доставки. DevOps та Agile можуть доповнювати один одного та застосовуватися в тандемі, але порівнювати ці методології не варто.

По суті, DevOps об'єднує дві розрізнені команди (розробку та експлуатацію), щоб забезпечити швидкі випуски програмного забезпечення. Agile орієнтований на співпрацю невеликих команд між собою для швидкого реагування на мінливі потреби користувачів.

Основні відмінності між DevOps та Agile:

- Розробка, тестування та розгортання програмного забезпечення відбуваються як у DevOps, так і Agile. Підхід Agile характерний тим, що технологія завершується відразу після розгортання. DevOps включає операції, які відбуваються постійно, наприклад, моніторинг і модифікації програмного забезпечення;

- В Agile різні фахівці несуть відповідальність за розробку, тестування та розгортання програмного забезпечення. У DevOps всі ці процеси відповідають спеціально навчені інженери;

- Agile виступає за поетапне розгортання після кожного спринту. Для DevOps характерна безперервна доставка (до кількох разів на день).

Автоматизація розгортання є важливою частиною забезпечення практик DevOps і управління налагодження CI/CD. Безперервна інтеграція (Continuous Integration, CI) і безперервна поставка (Continuous Delivery, CD) представляють собою набір методів та інструментів, які дають змогу розробникам з меншою кількістю помилок та частіше розгортати змінений код або ПО.

CI/ CD - це частина практик DevOps направлення. Вона є частиною гнучких методів розробки(agile). Автоматизація інтеграції та розгортання дозволяє розробникам зосередитись на коді, безпеці та реалізації бізнес-потреб.

Безперервна інтеграція - це методологія розробки і набір практик, при яких в код вносяться невеликі зміни з частими комітами. Так як сучасні завдання вирішуються з використанням великої кількості платформ та з різноманітними інструментами, існує потреба впровадженні інтеграції та тестуванні зроблених змін в коді та ПЗ.

Основною метою CI є можливість забезпечити чіткий та автоматичний спосіб інтеграції(збирання та тестування) ПО. З налагодженим процесом інтеграції, розробники будуть частіше робити коміти до змін, що сприяє покращенню їх комунікації та якості готового програмного забезпечення.

CD розпочинається після закінчення всього процесу інтеграції. Вона автоматизує процес розгортання ПО на різних хмарних середовищах або платформах.

1.2 Принципи формування нейромережових моделей

Можна виділити два типи НММ об'єктів: моделі, отримані методами прямої і інверсної ідентифікації [6]. При побудові моделей, заснованих на прямих методах ідентифікації, використовують два різних підходи: "пророкування" (прогнозування) поведінки об'єкта і "імітація" або моделювання поведінки об'єкта. При моделюванні поведінки об'єкта доступною є лише інформація про значеннях вхідних сигналів $u(k), u(k-1), \dots$ і вихідних сигналів моделі $\hat{y}(k), \hat{y}(k-1), \dots$, при цьому сигнали на виході об'єкта не вимірюються. Отримана таким чином модель називається паралельною і описується рівнянням:

$$\hat{y}(k+1) = f(\hat{y}(k), \hat{y}(k-1), \dots, u(k), u(k-1), \dots) \quad (2.1)$$

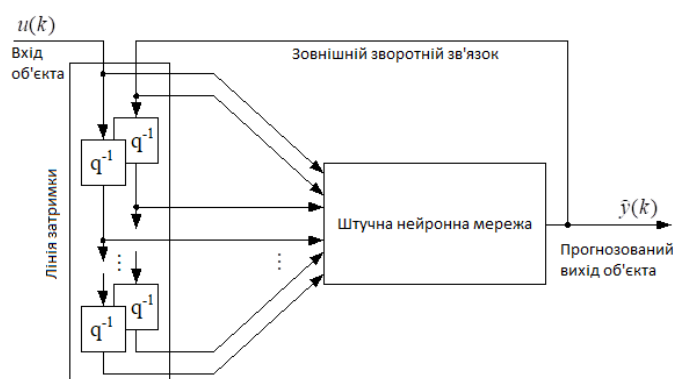


Рисунок 2.1 – Паралельна НММ

Паралельна модель (рисунок 2.1) може використовуватися для імітації поведінки об'єкта без підключення до реальної системи або коли датчики

входять до складу самої моделі. Для діагностики стану системи необхідно порівняти значення сигналів на виході об'єкта і моделі і оцінити величину помилки.

При прогнозуванні поведінки об'єкта використовується послідовно-паралельна модель (рисунок 2.2). Значення сигналу на виході моделі $\hat{y}(k+1)$ розраховується на підставі значень вхідних $u(k), u(k-1), \dots$ і вихідних $y(k), y(k-1), \dots$ сигналів об'єкта згідно з рівнянням:

$$\hat{y}(k+1) = f(y(k), y(k-1), \dots, u(k), u(k-1), \dots) \quad (2.2)$$

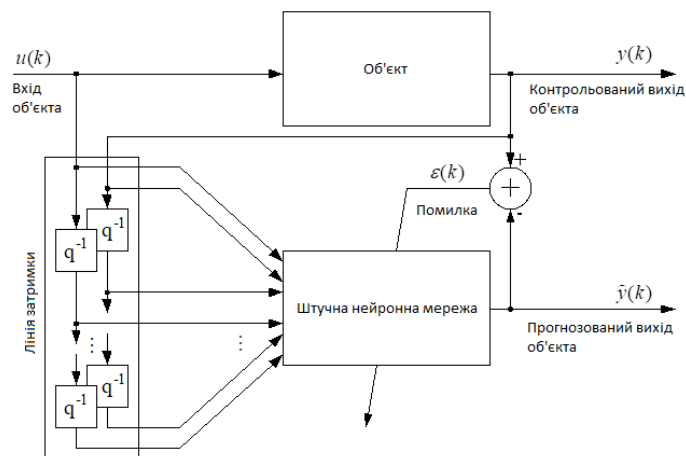


Рисунок 1.2 – Послідовно-паралельна НММ

Остання дозволяє передбачати поведінку об'єкта на будь-яку кількість тактів вперед, завдяки чому вона набула найбільшого поширення при вирішенні задач ідентифікації.

Якщо об'єкт описується рівнянням виду:

$$y(k+1) = f(y(k), y(k-1), \dots, u(k)), \quad (2.3)$$

то для побудови системи управління можна використовувати інверсну модель (рисунок 1.3). Однак дана модель може бути застосована лише до

обмеженого класу об'єктів, для яких запізнення по сигналу управління дорівнює одному такту.

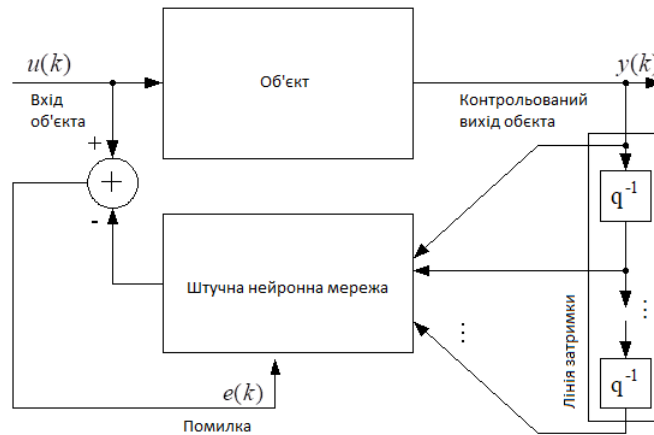


Рисунок 1.3. Інверсна НММ

Найбільшого поширення для вирішення завдань ідентифікації та управління отримали ШНМ типу БП і РБМ [1]. Та обставина, що обидві ці мережі дозволяють апроксимувати з будь-якої заданої точністю будь-яку безперервну функцію, забезпечило їх широке застосування в задачах ідентифікації нелінійних об'єктів, а поєднання апроксимуючих властивостей зі здатністю швидко навчатися дозволяє використовувати їх для керування нелійними динамічними об'єктами в реальному часі. Обидві мережі використовують апроксимацію нелінійного оператора об'єкта (2.4) деякою системою базисних функцій $\{\Phi_i(p)\}$, реалізованої нейронами, що утворюють шари мережі, а завдання ідентифікації зводиться до навчання мережі, тобто налаштування параметрів мережі на основі пред'явлення навчальної вибірки, до складу якої входять вимірювані значення входних і відповідних вихідних змінних

$$\Phi(k) = \sum_{i=1}^N w_i \Phi_i(k) \quad (2.4)$$

де w_i - вагові параметри мережі;

$$p(k) = (y(k-1), \dots, y(k-k_y), u(k), \dots, u(k-k_u))^T.$$

При побудові БП вибір виду активаційної функції відіграє істотну роль. Так як для навчання мережі зазвичай застосовуються різні методи оптимізації, які оперують як значеннями самої функції активації, так і значеннями її похідних, то необхідно, щоб використовувана функція задовольняла ряду умов: була обмеженою, монотонною і безперервно-диференційованою. Функції, що володіють перерахованими властивостями, називаються сигмоїдальними. До них відносяться:

- логістична:

$$f_{\log}(u) = \frac{1}{1 + e^{-\alpha u}}; f'_{\log}(u) = f_{\log}(u) \cdot (1 - f_{\log}(u)) \quad (2.5)$$

- гіперболічний тангенс:

$$f_{th}(u) = \frac{e^{\alpha u} - e^{-\alpha u}}{e^{\alpha u} + e^{-\alpha u}}; f'_{th}(u) = 1 - f_{th}^2(u). \quad (2.6)$$

Особливо важливу роль такі функції відіграють при моделюванні нелінійних залежностей між вхідними та вихідними змінними.

Як уже зазначалося, БП та РБМ є універсальними апроксиматорами і дозволяють апроксимувати з будь-якою заданою точністю будь-яку безперервну функцію, за умови, що існує достатня кількість прихованих шарів в мережі. Це твердження справедливо і для мереж тільки з одним прихованим шаром. Необхідною умовою є те, що функції активації нейронів прихованого шару $f(\cdot)$ повинні бути безперервними, обмеженими і непостійними. Ці вимоги досить м'які, причому сигмоїдальні функції (2.5) і (2.6) є лише одним з можливих варіантів активаційних функцій, що задовольняють цим умовам. Даний теоретичний результат є логічним

обґрунтуванням використання БП для моделювання нелінійних систем, тому що гарантує, що мережі з одним прихованим шаром завжди буде достатньо для подання будь-якої довільної неперервної функції. Однак відкритим залишається питання про вибір кількості нейронів в прихованому шарі, яке забезпечить задану точність рішення. Крім того, викладені результати засновані на припущенні, що значення ваг в мережі задані коректно.

В даний час існує велика кількість методів налаштування параметрів ШНМ, що відрізняються як динамічними властивостями, так і обчислювальною складністю. Навчання БП складається в налаштуванні ваг, що характеризують силу взаємодії між нейронами, і зазвичай виконується на скінченній навчальній множині, що містить P навчальних пар. Кожна навчальна пара включає вектор вхідних сигналів $p(k)$ ($1 \leq k \leq P$) і відповідний йому цільової (бажаний) вихідний вектор $d(k)$.

РБМ мають високу швидкість навчання і завдяки механізму адаптації структури мережі при їх навчанні не виникає проблем із зупинкою в локальних мінімумах. Однак треба зазначити, що число нейронів в шаблонному шарі РБМ експоненціально залежить від розмірності вхідного простору, що при використанні паралельної або паралельно-послідовної моделей об'єктів може привести до досить громіздкою структурою мережі і збільшити час отримання результату. Зміна структури РБМ зазвичай здійснюється її поступовим ускладненням шляхом додавання нових нейронів, кожний раз, коли при появі чергового вхідного сигналу виникає помилка ідентифікації, що перевищує допустиму. Параметрична ідентифікація (навчання мережі) складається в налаштуванні її вагових коефіцієнтів і зводиться до мінімізації зазвичай квадратичного функціоналу помилки ідентифікації [1].

Крім розглянутих вище мереж, в даний час при вирішенні задач ідентифікації та управління використовуються мережі типу СМАС (Cerebellar Model Articulation Controller - церебральна модель артикуляції контролера), засновані на моделі, яка описує процеси управління рухом, що

відбуваються в мозочку, запропонованої Дж. Албусом. Особлива привабливість мереж СМАС обумовлена високою швидкістю їх навчання (це досягається застосуванням спеціальних способів кодування інформації), що дозволяє використовувати дані мережі при апаратній реалізації керуючих систем на базі мікроконтролерів. При цьому зазвичай обмежуються побудовою систем управління об'єктами з розмірністю не вище третього порядку, тому що збільшення розмірності досліджуваного об'єкта призводить як до ускладнення кодування інформації (появі багатовимірних матриць асоціацій), так і до різкого зростання кількості необхідних обчислень [2].

1.3 Аналіз алгоритмів навчання ШНМ

Навчання нейромережі – ітеративний процес. На кожній ітерації обчислюються мережеві виходи для одного (або більше) зразка в навчальному наборі, і коригуються мережеві ваги з метою зменшення помилки між фактичним мережевим виходом $(y_{i,p}^L)$ ($i=1,...,N^L$) і цільовим виходом $(d_{i,p})$ для даного зразка. При використанні підходящої мережної архітектури і алгоритму навчання мережеві ваги прагнуть до значень, при яких мережевий вихід стає прийнятно близьким до цільового виходу для кожного зразка в наборі навчання.

1.4 Вибір критерію якості навчання

Точність результату на кожній ітерації оцінюється з використанням функції помилки E (або енергетичної функції $J(w)$):

$$E = \sum_{p=1}^P E_p, \quad (2.7)$$

де E_p – частковий внесок зразка P в повну мережеву помилку E .

Вибір функції помилки має настільки ж істотний вплив на ефективність БП, як і вибір алгоритму навчання. Найбільш широко використовують функцією помилки є квадратична функція помилки [3]:

$$E = \frac{1}{2} \sum_{p=1}^P \sum_{i=1}^{N^L} (d_{i,p} - y_{i,p}^L)^2 \quad (2.8)$$

Нормалізована її версія, яка називається середньоквадратичної функцією помилки:

$$E = \frac{1}{2PN^L} \sum_{p=1}^P \sum_{i=1}^{N^L} (d_{i,p} - y_{i,p}^L)^2 \quad (2.9)$$

Перевага останньої полягає в тому, що вона нечутлива як до числа зразків в наборі навчання, так і до числа нейронів у вихідному шарі мережі, що дозволяє використовувати її для порівняння в різних завданнях навчання.

Особливістю функцій (2.8) і (2.9) є те, що зменшення E може бути пов'язано зі збільшенням числа помилково класифікуються зразків; фактично, зменшення E до мінімально прийнятного рівня будь-який з цих функцій помилки не обов'язково відповідає мінімальному числу нерозпізнаних образів. Це пов'язано особливостями поверхні функціоналу помилки БП – локальними мінімумами і багатовимірними "плато" [7].

БП застряг в локальному мінімумі, коли величина мережевий помилки E для послідовних періодів навчання стабілізується на порівняно високому рівні. Локальні мінімуми, як відомо, утворюються в специфічних тестових завданнях. Однак вони відсутні, якщо навчається завдання є лінійно нероздільні або число прихованих вершин мережі збігається з кількістю зразків в наборі навчання, або число зразків менше або дорівнює числу

параметрів моделі. У кожному з цих випадків відбувається навчання мережі з обраної архітектурою. Однак, на жаль, жоден з наявних результатів не містить практичних рекомендацій для реальних додатків.

Використовують альтернативні функції помилки, що дозволяють поліпшити характеристики збіжності алгоритмів навчання БП. Один з таких підходів полягає в застосуванні функції помилки, що дозволяє мережі вийти з локального мінімуму, зокрема, функції помилки з взаємної ентропією:

$$E = - \sum_{p=1}^P \sum_{i=1}^{N^L} \ln \left[\left(y_{i,p}^L \right)^{d_{i,p}} \left(1 - y_{i,p}^L \right)^{1-d_{i,p}} \right]. \quad (2.10)$$

При використанні (2.10), градієнти помилки для погано помітних зразків значно вище, ніж при використанні квадратичної і середньоквадратичної функцій помилки, що дозволяє мережі швидше рухатися в плоских областях простору ваг (тобто при перетині плато).

Друга стратегія полягає у введенні обмежень в функцію помилки, таким чином, щоб мережа не сходилася до поганих зразків в просторі ваг. Одним з варіантів такої функції помилки є показова функція помилки:

$$E = \frac{1}{2} \sum_{p=1}^P \sum_{i=1}^{N^L} e^{\alpha \left(y_{i,p}^L - d_{i,p} \right)^{\beta}}, \quad (2.11)$$

де α і β – позитивні, визначені користувачем параметри.

Значення параметру β задає допустимий рівень помилки для кожного елемента $(d_{i,p})$ в наборі навчання. Моллер рекомендує задавати початкове значення $\beta=0.9$, і потім ділити його навпіл щоразу, коли на кожному виході мережі $(y_{i,p}^L)$ помилка досягає допустимого рівня [3]. Параметр α задає крутизну експоненти, збільшуючи помилку для тих мережевих виходів, які

мають неприпустимий рівень помилок, який визначається параметром β . При $\alpha \rightarrow \infty$ не відбувається зменшення мережевої помилки E , що відповідає збільшенню числа неправильно класифікованих зразків. Установка досить великого значення коефіцієнта α може привести до значного зменшення швидкості збіжності алгоритму, тому що діапазон допустимих рішень обмежений. Хороші результати дає $\alpha=1.0$, хоча оптимальне значення α залежить від конкретного завдання.

Навчання БП є успішним при досягненні мінімального значення помилки, що визначається з умов конкретного завдання. Слід зазначити, що досягнення мінімального значення помилки часто є небажаним, тому що призводить до явища перенавчання БП і погіршення його фільтруючих і узагальнюючих властивостей. Виняток становлять завдання апроксимації функцій, для яких потрібна висока точність одержуваних рішень.

Перший теоретично доведений алгоритм для настройки ваг при навчанні БП відомий як алгоритм зворотного поширення помилки (ЗП). Сьогодні ЗР може розглядатися як еталонний тест, з яким порівнюються всі інші методи навчання.

1.5 Безперервна доставка/розгортання

Безперервна доставка є продовженням безперервної інтеграції, оскільки вона автоматично розгортає всі зміни коду в тестовому та/або операційному середовищі після фази збірки. Це означає, що автоматизовано не лише процес тестування, а й процес випуску продукту, тому програму можна розгорнути в будь-який момент одним кліком. З безперервною доставкою можливо робити оновлення щодня, щотижня, кожні два тижні або з будь-якою іншою періодичністю.

Безперервна інтеграція, розгортання та доставка утворюють три стадії автоматизованого конвеєра випуску ПЗ. Ці три стадії охоплюють розробку програмного забезпечення від ідеї до доставки кінцевому

користувачеві. Стадія інтеграції – це перший крок у цьому процесі. Безперервна інтеграція охоплює етап, у якому кілька розробників намагаються поєднати зміни коду з головним репозиторієм коду проекту.

Безперервна інтеграція, розгортання та доставка становлять три етапи автоматизованого конвеєра випуску програмного забезпечення. Ці три етапи охоплюють розробку програмного забезпечення від ідеї до доставки кінцевому користувачеві. Етап інтеграції є першим кроком у цьому процесі. Безперервна інтеграція включає фазу, коли кілька розробників намагаються об'єднати свої зміни коду в основний репозиторій коду проекту.

Безперервна доставка є розширенням безперервної інтеграції. На етапі доставки робочі продукти упаковуються для доставки кінцевим користувачам. На цьому етапі запускаються автоматизовані засоби збирання для створення таких виробів. Цей етап складання вважається «зеленим». Це означає, що ваш продукт повинен бути готовим до розгортання для ваших користувачів.

Безперервне розгортання - це завершальний етап конвеєра. На етапі розгортання робочий продукт автоматично запускається та поширюється серед кінцевих користувачів. У процесі розгортання продукт має успішно пройти етапи інтеграції та доставки. Тепер мені потрібно автоматично розгорнути або розповсюдити продукт. Це робиться за допомогою сценарію або інструмента, який автоматично розгортає продукт на загальнодоступному сервері або в іншій системі розповсюдження, наприклад, у магазині програм.

Безперервне розгортання йде на крок далі, ніж безперервна доставка. При такому підході кожна версія додатку, яка проходить через усі стадії виробничого процесу, розгортається на серверах. Втручання людини не потрібно, а розгортанню нової версії робочого середовища може перешкодити лише помилка під час тестування.

Безперервне розгортання — це спосіб прискорити цикл зворотнього зв'язку з клієнтами та звільнити команду від непотрібного стресу, оскільки

дня релізу більше не існує. Розробники можуть зосередитися на створенні програмного забезпечення. Різницю між безперервною доставкою та розгортанням наведено у таблиці 1.1

Таблиця 1.1 – Різниця безперервної доставки від розгортання

Безперервна доставка	Безперервне розгортання
Це підхід для отримання змін нових функцій, конфігурації та виправлень помилок.	Це підхід до розробки програмного забезпечення за короткий цикл.
Стосується логічної еволюції ІС.	Відноситься до автоматизованих реалізацій вихідного коду.
Зосереджується на тому, щоб належним чином публікувати нові зміни для своїх клієнтів.	Акцент на зміні на всіх етапах виробничого конвеєру.
На CI/CD розроблений код постійно доставляється, поки програміст не вважає, що він готовий до доставки.	На CI/CD розробники розгортають код безпосередньо на етапі виробництва, коли він розробляється.
Дозволяє розробникам перевіряти оновлення програмного забезпечення.	Дозволяє швидко розгорнути та перевірити нові функції та ідеї.
Він використовує тести бізнес-логіки.	Виконується будь-яка стратегія тестування.
Доставляє код для перевірки, який можна створити для випуску.	Розгортання коду за допомогою автоматизованого процесу.
Потрібна міцна основа постійної інтеграції.	Потрібна добра культура тестування.

1.6 Дослідження інструментів оркестрації, які надаються хмарними провайдерами

Хмарна оркестровка – це використання технології програмування для керування взаємозв'язками та взаємодією між робочими навантаженнями на публічній та приватній хмарній інфраструктурі. Вона об'єднує автоматизовані завдання в згуртований робочий процес для досягнення мети, з наглядом за дозволами та дотриманням політик компаній. Хмарна оркестровка зазвичай використовується для надання, розгортання або запуску серверів, придбати та призначити ємність для зберігання, керувати мережею, створювати віртуальні машини і отримати доступ до певного програмного забезпечення в хмарних сервісах. Це досягається за допомогою трьох основних, тісно пов'язаних атрибутів хмарної оркестровки: обслуговування, робоче навантаження та оркестровка ресурсів. Платформа оркестровки може інтегрувати перевірку дозволів на безпеку та відповідність.

Хмарна оркестровка цікавить багато ІТ-організацій та спеціалістів DevOps, як спосіб пришвидшити надання послуг і зменшити витрати. Хмарний оркестратор автоматизує керування, координацію та організацію складних комп'ютерних систем, сервісів та проміжного програмного забезпечення. На додаток до зменшення участі персоналу, оркестровка усуває ймовірність помилок, що виникають у забезпеченні, масштабуванні чи інших хмарних процесах. Оркестровка підтримує доставку хмарних ресурсів клієнтам і кінцевим користувачам, у тому числі в моделі самообслуговування, коли користувачі запитують ресурси без участі ІТ.

Програмне забезпечення для оркестрування допомагає ІТ-організаціям стандартизувати шаблони та застосовувати методи безпеки. Це також хороший спосіб відмовитися від віртуальних машин, забезпечуючи видимість і контроль над хмарними ресурсами та витратами. Оскільки хмарні платформи контролюють взаємодію багатьох різних елементів стеку додатків, вони можуть спростити комунікацію та підключення одного

робочого навантаження до інших програм і користувачів, а також забезпечити правильне налаштування та підтримку зв'язків. Такі продукти зазвичай включають веб-сервери, тому оркестровкою можна керувати за допомогою однієї панелі-інтерфейсу.

Централізована платформа оркестрування дозволяє адміністраторам переглядати та покращувати сценарії автоматизації.

У просунутих організаціях розробники та працівники сфери бізнесу можуть звернутися до програмного забезпечення для оркестрування в хмарі як механізму самообслуговування для розгортання ресурсів; адміністратори можуть використовувати його, щоб відстежувати залежність організації від різних ІТ-пропозицій та керувати поверненням платежів. Багато постачальників пропонують продукти хмарного оркестратора. Команди DevOps також можуть реалізувати хмарну оркестрацію різними способами за допомогою інструментів автоматизації та керування, щоб відповідати їхнім процесам і методологіям.

Оцінюючи продукти для хмарної оркестровки, адміністраторам рекомендується спочатку відобразити робочі процеси задіяних програм. Цей крок допоможе адміністратору уявити, наскільки складним є внутрішній робочий процес для програми та як часто інформація виходить за межі набору компонентів програми. Це, у свою чергу, може допомогти адміністратору вирішити, який тип продукту для оркестровки допоможе найкраще автоматизувати робочий процес і задовольнити вимоги бізнесу найбільш економічно ефективним способом.

Загалом інструменти або програмне забезпечення хмарної оркестровки працюють подібним чином у загальнодоступних, приватних і гібридних хмарах, хоча специфіка даного випадку використання може сприяти перевагам функцій однієї перед іншою. Наприклад, пакет vRealize Suite від VMware включає гібридну платформу управління хмарою, яка автоматизує доставку хмарної інфраструктури, додатків і послуг; операційний компонент, який допомагає з плануванням, управлінням і масштабуванням; управління

та аналіз журналів у режимі реального часу; і автоматизований калькуляція витрат, облік використання та ціноутворення на послуги.

Далі розглянемо найпопулярніші хмари у яких можна розгорнути Kubernetes кластер.

Веб-сервіси Амазон. AWS — один із найпопулярніших хмарних провайдерів. Це досить зріла хмара, яка використовується багатьма компаніями для розгортання хмарної інфраструктури, зберігання даних і т.д.

AWS створила власний сервіс Amazon EKS, тому що не змогла встояти перед тенденцією багатьох компаній, які використовують архітектуру мікросервісів.

Amazon Elastic Kubernetes Service (Amazon EKS) – це керований сервіс Kubernetes, який спрощує запуск Kubernetes на AWS та локально. Kubernetes — це платформа з відкритим вихідним кодом для автоматизації розгортання, масштабування та керування контейнерними програмами. Amazon EKS сертифікований на сумісність з Kubernetes, тому існуючі програми, що працюють у відкритих версіях Kubernetes, сумісні з Amazon EKS.

- Переваги використання Amazon EKS:
- Провайдер гарантує високий рівень надійності та високу доступність своїх послуг.
- Повне керування вузлами Kubernetes.
- Хмарна платформа Google. GCP зарекомендувала себе стабільною компанією. Kubernetes був розроблений на основі досвіду роботи з контейнерними веб-сервісами Gmail та Youtube. GKE дозволяє швидко налаштувати та запустити Kubernetes, повністю позбавляючи необхідності встановлювати та керувати своїми власними кластерами Kubernetes.
- Переваги використання GKE:
- Не потрібно встановлювати та налаштовувати Kubernetes.
- Значна економія коштів, оскільки вам не потрібно платити за механізми керування кластером та головні вузли.

Майкрософт Азур. Служба Azure Kubernetes (AKS) надає безсерверну платформу Kubernetes з функціями безпеки та управління корпоративного рівня, а також вбудованими можливостями безперервної інтеграції та безперервної доставки.

Переваги використання AKS:

Ефективність використання ресурсів. Повністю керований AKS забезпечує просте та ефективне розгортання контейнерних додатків та керування ними.

AKS інтегрується з Azure Active Directory (AD), щоб надати користувачам доступ на вимогу, що значно знижує загрози та ризики. AKS також повністю відповідає стандартам та нормативним вимогам, таким як System and Organization Controls (SOC), HIPAA, ISO та PCI DSS.

1.6 Аналіз інструментів CI/CD

На сьогоднішній день існує чимало інструментів для безперервної інтеграції та безперервної доставки. Найпопулярніші з них – Jenkins, TeamCity, CircleCI, Travis CI та Bamboo.

Jenkins – це програмна платформа Java з відкритим вихідним кодом, призначена для підтримки процесів безперервної інтеграції програмного забезпечення. Він був створений у 2008 році з Hudson Project, що належить Oracle, та його основним автором є Косуке Кавагуті.

Частини процесу розробки програмного забезпечення, які потребують участі людини, можна автоматизувати, щоб забезпечити безперервну інтеграцію. Запускається в контейнері сервлетів, як Apache Tomcat. Підтримує інструменти контролю версій, такі як AccuRev, CVS, Subversion, Git, Mercurial, Perforce, Clearcase, RTC. Ви можете створювати проекти за допомогою Apache Ant та Apache Maven та запускати довільні сценарії оболонки та пакетні файли Windows. Складання може бути запущено різними способами, наприклад, за допомогою події фіксації у вашій системі керування версіями, за розкладом, шляхом запиту певної URL-адреси або

після завершення іншого складання в черзі. Jenkins можна розширити за допомогою плагінів.

Контроль доступу реалізований двома способами: автентифікація користувача та авторизація. Підтримується захист від зовнішніх загроз, таких як CSRF-атаки та шкідливі зборки.

У 2011 році Дженкінс отримав нагороду InfoWorld за найкращий проект з відкритим вихідним кодом.

TeamCity - це сервер керування збіркою та безперервною інтеграцією JetBrains. Це комерційне програмне забезпечення, вперше випущене 2 жовтня 2006 р., поширюється на приватну ліцензію. Доступні ліцензії Freemium для 100 конфігурацій складання та 3 безкоштовні ліцензії агента складання. Для проектів з відкритим вихідним кодом може знадобитися безкоштовна ліцензія.

Характерні особливості:

Закрита фіксація (запобігає порушенню вихідного коду в системі керування версіями за рахунок віддаленого запуску збірок, щоб розробники могли внести локальні зміни перед фіксацією)

Запускайте кілька збірок та тестів одночасно на різних платформах та середовищах.

Інтегроване покриття коду, перевірки та виявлення дублікатів

Інтеграція з IDE: Eclipse, IntelliJ IDEA, Visual Studio

Підтримувані платформи: Java, .NET, Ruby

Підтримує систему контролю версій.

Компанія CircleCI, заснована в 2011 році та має штаб-квартиру в Сан-Франциско, штат Каліфорнія, надає послуги з автоматизації етапу безперервної інтеграції життєвого циклу розробки програмного забезпечення (SDLC).

Пропоновані ними послуги CI можуть розміщуватись у хмарі або на приватних серверах. Завдання CI створюються у чотирьох різних середовищах: образи Docker, віртуальні машини Linux, віртуальні машини

Windows або віртуальні машини MacOS. Вони демонструють свою підтримку проектам з відкритим кодом, надаючи організаціям безкоштовні кредити на складання з відкритим кодом.

Travis CI — це веб-сервіс для створення та тестування програмного забезпечення, який використовує GitHub як хостинг вихідного коду. Програмні компоненти сервісу також є на GitHub, але розробники не рекомендують використовувати його в закритих проектах.

Веб-служби підтримують створення проектів багатьма мовами, включаючи C, C++, D, JavaScript, Java, PHP, Python та Ruby. Різні проекти з відкритим вихідним кодом, такі як Ruby та Ruby on Rails, використовують Travis CI для безперервної інтеграції коду.

Bamboo - це сервер безперервної інтеграції та безперервного розгортання, розроблений Atlassian. Спочатку доступна як локальна, так і хмарна служба, у травні 2016 року було оголошено, що хмарну версію буде припинено до кінця січня 2017 року.

Bamboo може запускати кілька збірок паралельно для швидшого завершення. У разі збоїв складання Bamboo надає аналіз збою, включаючи трасування стека, а також надає REST API, що надає інформацію про сервер, поточний стан їх складання і т.д.

Також має вбудовану функціональність підключення до репозиторіям, як-от Git, Apache Subversion, Mercurial, Concurrent Version System, Perforce, Bitbucket тощо. У системі є завдання збирання для таких інструментів, як Ant, Maven, Make, MSBuild та ін., що дозволяє використовувати цей CI інструмент з багатьма мовами програмування.

Він тісно пов'язаний з іншими продуктами Atlassian, такими як JIRA, Confluence та інструмент покриття коду Java Clover. Таким чином, кожен у команді, включаючи менеджерів та тестувальників, може мати чітке уявлення про те, що відбувається у проекті.

Кожен бізнес який пов'язаний з розробкою програмного забезпечення або з ІТ сферою в цілому прагне бути спритними, для того щоб

впроваджувати інновації та швидко адаптуватися до змін. Waterfall була розкритикована за недостатню гнучкість, а її основна мета – формальне управління проектом, завдавала збитків термінам, вартості та якості.

Команди потребували гнучкості під час розробки програмного забезпечення. Необхідність реалізації проекту у формі коротких "спринтів" і випуску найчастіших (від двох тижнів до двох днів) релізів зажадали нового підходу.

У 2001 році на зміну Waterfall прийшла гнучка методологія розробки або Agile. Вона включає низку підходів та практик, заснованих на чотирьох цінностях та 12 принципах «Маніфесту гнучкої розробки програмного забезпечення». Сюди також відносять SCRUM, Kanban, Lean, Feature-driven development (FDD) та інші подібні підходи.

Методологію Agile критикували за відсутність управління вимогами. Замовник може виставити нові вимоги наприкінці кожної ітерації, що суперечить архітектурі вже створеного продукту. Часті зміни та вдосконалення продукту можуть призвести до масового рефакторингу та плаваючої вартості проекту в результаті

Методологія DevOps набула широкого поширення після організованої бельгійським розробником Патріком Дебуа у 2009 році конференції DevOpsDays.

Культура DevOps припускає, що кожен із членів команди відповідальний за кінцевий результат. Базується вона на кількох основних положеннях:

Регулярне співробітництво та спілкування. Команда повинна працювати злагоджено, розуміти потреби та очікування всіх її членів.

Впровадження поступового розгортання дозволяє групам доставки випускати продукт, маючи можливість вносити оновлення та робити відкат, якщо щось не піде.

Спільна відповідальність. Усі члени команди повинні рухатися до єдиної мети та відповідати за проект рівною мірою.

Автоматизація спрощує робочі процеси, скорочує кількість збоїв та відкатів, зменшує кількість помилок, що виникають при ручному налаштуванні. Підвищення ефективності, покращення продуктивності та користь для кінцевого споживача – головні переваги автоматизації.

DevOps сприяє розвитку лише у тому випадку, якщо конкретні показники збираються та аналізуються безперервно.

Тому, для ефективного створення або оновлення програмного забезпечення була створена методологія DevOps.

CI/ CD - це частина практик DevOps направлення. Вона є частиною гнучких методів розробки(agile).

Основною метою CI є можливість забезпечити чіткий та автоматичний спосіб інтеграції(збирання та тестування) ПО. CD розпочинається після закінчення всього процесу інтеграції.

В даний час ринок ІТ експериментує з різними методологіями розробки програмного забезпечення. Все почалося із моделі водоспаду. Водоспадна модель являла собою складний багатоетапний процес, який згодом став менш чутливим до ринків, що швидко змінюються.

Модель водоспаду замінили гнучкими методологіями розробки, такими як Agile.

Методологія CI/CD швидко розвивається. Це означає, що кількість інструментів безперервної доставки та розгортання зростає.

1.7 Постановка задачі дослідження

Метою роботи є аналіз та розробка хмарної мікросервісної інфраструктури для дослідження систем нейроідентифікації нелінійних об'єктів за допомогою оркестратора контейнерів Kubernetes для обслуговування веб-серверу адресної книги.

Для виконання поставленої мети були виконані наступні завдання:
розробити мікросервісну архітектуру для кластеру Kubernetes;

- розробити CI/CD інфраструктуру для контейнеризованого веб-серверу адресної книги;

— написання коду до CI системи CircleCI;

— написання коду шаблонів для інструмента Helm.

2 ПРОЕКТУВАННЯ ХМАРНОЇ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

2.1 Аналіз системних вимог

Аналіз вимог - це процес збору вимог до програмного забезпечення, їх організації, документування, аналізу, виявлення протиріч, недосконалостей та вирішення конфліктів у процесі розробки програмного забезпечення. В англomовному світі ми також говоримо про сферу під назвою «розробка вимог». У процесі збору вимог важливо враховувати потенційно конфліктуючі вимоги різних зацікавлених сторін, таких як замовники, розробники та користувачі.

Повнота та якість аналізу вимог відіграють важливу роль у загальному успіху проекту. Вимоги до програмного забезпечення мають бути задокументовані, здійсненні та тестовані на рівні деталізації, достатньому для проектування системи. Вимоги можуть бути функціональними та нефункціональними.

Аналіз вимог включає три типи діяльності:

Збір вимог: спілкування з клієнтами та користувачами, щоб визначити, які їхні вимоги.

Аналіз вимог: визначення, чи є зібрані вимоги неясними, неповними, неоднозначними, або суперечать, і потім рішення цих проблем.

Документування вимог: Вимоги можуть бути задокументовані в різних формах, таких як простий опис, сценарії використання, користувацькі історії, або специфікації процесів.

Аналіз вимог може бути довгим і важким процесом, під час якого залучені багато тонких психологічних навичок. Нові системи змінюють навколишнє середовище і відносини між людьми, таким чином важливо визначити всі зацікавлені особи, взяти до уваги всі їхні потреби і гарантувати, що вони розуміють значення нових систем. Аналітики можуть

використовувати кілька методів, щоб виявити вимоги від клієнта такі, як проведення інтерв'ю, або використання фокус-груп та створення списків вимог. Більш сучасні методи включають створення прототипів і сценаріїв використання.

Щоб надати якісний продукт, необхідно зібрати відповідні вимоги, ефективно вивчити зібрані вимоги та, нарешті, створити чіткий документ із вимогами як попередню умову. Весь цей процес називається аналізом вимог у життєвому циклі розробки програмного забезпечення.

Аналіз вимог починається з:

Збір вимог, також званих викликами.

Проаналізуйте зібрані вимоги, щоб зрозуміти точність та доцільність втілення цих вимог у можливий продукт.

Документування зібраних вимог.

Щоб переконатися, що всі перераховані вище кроки виконані правильно, клієнт повинен отримати чіткі, короткі і правильні вимоги. Клієнти повинні мати можливість правильно визначити свої вимоги, а бізнес-аналітики повинні вміти вловлювати їх так само, як клієнт намагається їх повідомити.

У багатьох випадках, бізнес-аналітик замовника не може ефективно зібрати вимоги. Це може бути пов'язане із залежністю кількох людей, пов'язаних з очікуваним кінцевим продуктом, інструментами, середовищем і т. д. Таким чином, завжди корисно залучати всі зацікавлені сторони, які можуть вплинути або на які може вплинути кінцевий продукт.

Можливі групи зацікавлених сторін включають інженерів з якості програмного забезпечення (як QC, так і QA), сторонніх постачальників, які можуть забезпечити підтримку проекту, передбачуваних кінцевих користувачів вашого продукту, програмістів та інших команд у вашій організації. Ми можемо надати модулі або програмних платформ, програмних бібліотек і т. д. для розробки продукту.

На другому етапі збираються різні зацікавлені сторони та проводять мозковий штурм. Проаналізуйте зібрані вимоги та шукайте можливості реалізації. Вони розмовляють один з одним, і неясності дозволяються.

Бізнес-вимоги - це вимога високого рівня, що описує, чого хоче кінцевий користувач від конкретної дії в програмній системі. Неможливо спроектувати всю програмну систему з урахуванням цих вимог. Це пов'язано з тим, що він не дає докладного опису того, як реалізовано програмну систему або її компоненти.

Тому бізнес-вимоги слід розбити більш докладні вимоги до програмного забезпечення. Далі це уточнюється у функціональних та нефункціональних вимогах.

Щоб створити хмарну мікросервісну архітектуру потрібно детально визначити вимоги до сервісів та систем, що будуть обслуговуватися у межах даної інфраструктури та проаналізувати склад кластеру Kubernetes. Структура кластеру Kubernetes приведена на рисунку 2.1.

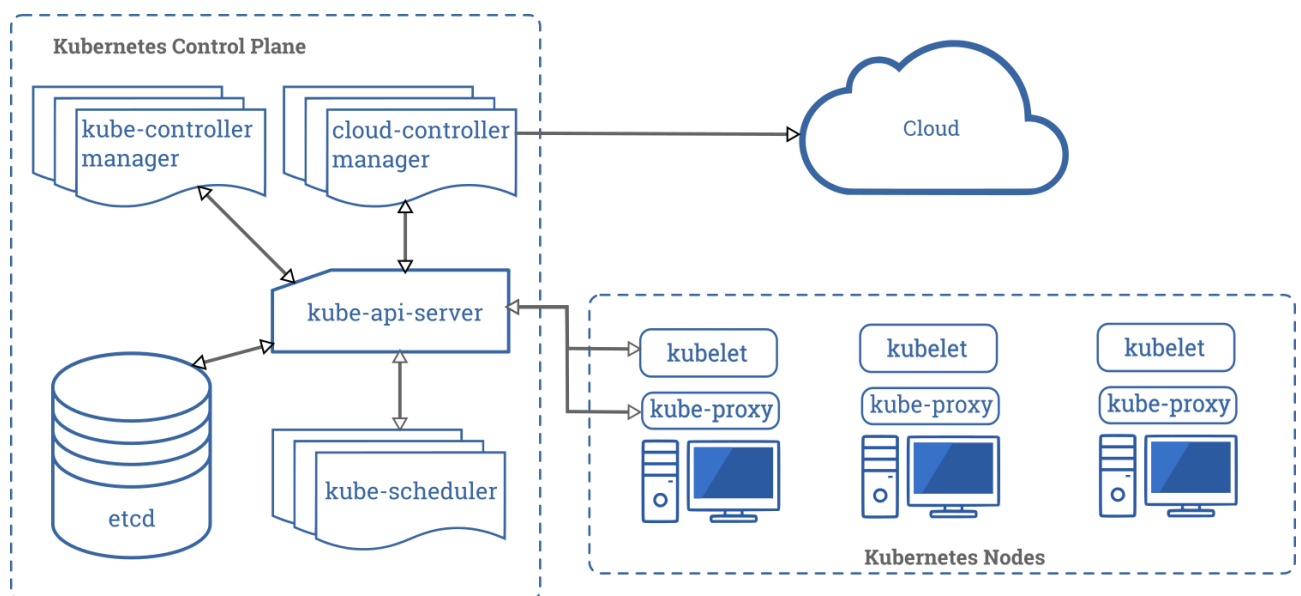


Рисунок 2.1 - Структура кластеру Kubernetes

Kubernetes - це проект з відкритим вихідним кодом, призначений для управління кластерами контейнерів Linux як єдиною системою. Kubernetes управляє контейнерами Docker та запускає їх на багатьох хостах, а також

забезпечує спільне розміщення та реплікацію багатьох контейнерів. Проект був запущений Google і зараз підтримується багатьма компаніями, включаючи Microsoft, RedHat, IBM та Docker.

Google використовує технологію контейнерів вже понад десять років. Ми почали з запуску понад 2 мільярди контейнерів на тиждень. У рамках проекту Kubernetes компанія ділиться своїм досвідом створення відкритої платформи, призначеної для запуску контейнерів.

Якщо ви використовуєте контейнери Docker, наступне питання полягає в тому, як масштабувати та запускати контейнери одночасно на багатьох хостах Docker та як їх збалансувати.

Працюючий кластер Kubernetes включає агенти, що працюють на вузлах (кубелети), і основні компоненти (API, планувальники і т.д.) поверх рішення для розподіленого зберігання. На діаграмі нижче показано бажаний кінцевий стан, але є деякі речі, над якими ми все ще працюємо, наприклад, як запустити kubelet (фактично всі компоненти) окремо в контейнері і зробити планувальник на 100%.

Дивлячись на системну архітектуру, ми можемо розбити її на служби, які працюють на кожному вузлі, та служби на рівні керуючого кластеру. Кожен вузол Kubernetes запускає службу, необхідну для управління вузлом з боку майстра та запуску додатків. Звісно, Docker працює на кожному вузлі. Docker забезпечує завантаження форми та виконання контейнера.

Kubelet керує модулями в контейнерах, образах, розділах тощо.

Також на кожному вузлі запускається простий проксі-балансувальник. Ця служба працює на кожному вузлі та визначається за допомогою API Kubernetes. Kube-Proxy може виконувати найпростішу переадресацію потоків TCP та UDP між набором серверних частин.

Система управління Kubernetes поділена на кілька компонентів. У цей момент усі вони працюють на головному вузлі, але незабаром це змінюється для створення кластеру відмов. Ці компоненти працюють разом, щоб забезпечити єдине виявлення вашого кластера.

Стан майстра зберігається в екземплярі etcd. Це забезпечує надійне збереження даних конфігурації та своєчасне сповіщення про зміну стану інших компонентів.

Kubernetes API забезпечує роботу сервера. Він розроблений як CRUD-сервер із вбудованою бізнес-логікою, реалізованою в окремих компонентах або плагінах. В основному він обробляє операції REST, перевіряючи їх та оновлюючи відповідні об'єкти в etcd (і, можливо, в інших репозиторіях).

Планувальник прив'язує незапущені модулі до вузлів через дзвінки API.

Інші функції рівня кластера видно у диспетчері контролерів. Наприклад, вузли виявляються, керуються контролем та контролюються контролером вузла. Потім можна розділити цю сутність на окремих компонентах, щоб зробити їх незалежними.

ReplicationController – це механізм на основі API. Зрештою, ми плануємо перейти до загального механізму плагінів при реалізації. Налаштування кластера створити за допомогою файлів оголошень *.yaml.

Для створення базового кластеру для поставленого завдання, необхідно задіяти, щонайменше один k8s кластер. На сервері, який сконфігуровано як Node має бути щонайменше 4 ГБ оперативної пам'яті. Усі хости мають знаходитися у одній мережі без обмежень. Для доступності сервісів у мережі Інтернет достатньо вивести до мережі будь який хост k8s кластеру. База даних має знаходитися виключно у внутрішній мережі k8s кластеру.

Створення хмарної мікросервісної інфраструктури має бути автоматизовано за наступними чинниками:

- створення сховища чутливих даних у Google Cloud Platform.
- написання чартів Helm-Chart;
- підняття інфраструктури у Google Cloud Platform;
- написання пайплайну для CircleCI.

Розглянемо популярні рішення для реалізації даних завдань.

Terraform - це інструмент від Hashicorp, який допомагає декларативно управляти інфраструктурою. У цьому випадку вам не потрібно вручну створювати екземпляри, мережі тощо в консолі вашого хмарного провайдера; досить написати конфігурацію, в якій буде окреслено, як ви бачите свою майбутню інфраструктуру.

Ця конфігурація створюється в придатному для читання людиною текстовому форматі. Якщо потрібно змінити інфраструктуру, відредагуйте конфігурацію та натисніть кнопку. Terraform направить виклики API вашому хмарному провайдеру, щоб привести інфраструктуру у відповідність з конфігурацією, вказаною в цьому файлі.

Переміщення управління інфраструктурою до текстових файлів відкриває можливість озброїтися всіма улюбленими вихідними кодами та інструментами управління процесами, а потім переорієнтувати їх на роботу з інфраструктурою.

Тепер інфраструктура підпорядковується системам контролю версій, так само, як і вихідний код, вона може бути аналогічно переглянута або відкочена назад до попереднього стану, якщо щось піде не так.

Terraform відображає ресурси, описані у файлі конфігурації, з відповідними ресурсами хмарного провайдера. Це зіставлення називається станом, який є гігантським файлом JSON. Під час запуску Terraform оновлює стан, направляючи відповідний запит постачальнику хмар. Потім порівнює повернуті ресурси з інформацією, записаною в конфігурації Terraform. Якщо виявлено будь-яку різницю, створюється план, по суті, список змін, які потрібно внести до ресурсів хмарного провайдера, щоб фактична конфігурація відповідала тій, що вказана у вашій конфігурації. Нарешті, Terraform застосовує ці зміни, направляючи відповідні виклики постачальнику хмарних послуг.

Розгортання додатків у потужній та популярній системі організації контейнерів Kubernetes може являти собою складне завдання. Для налаштування однієї програми може знадобитися створення декількох

незалежних ресурсів Kubernetes, у тому числі подів, служб, розгортань та наборів копій, і для кожного з цих ресурсів потрібний деталізований файл маніфесту YAML.

Helm – це диспетчер пакетів для Kubernetes, що спрощує для розробників та операторів упаковку, налаштування та розгортання програм та служб у кластерах Kubernetes.

Helm є офіційним проектом Kubernetes і підтримується некомерційним фондом Cloud Native Computing Foundation, який підтримує проекти з відкритим вихідним кодом, пов'язані з екосистемою Kubernetes.

Helm може виконувати такі завдання:

- встановлення програмного забезпечення;
- автоматичне встановлення залежностей програмного забезпечення;
- оновлення програмного забезпечення;
- налаштування розгортання програмного забезпечення;
- доставка пакетів програмного забезпечення із репозиторіїв.

Helm реалізує ці можливості за допомогою наступних компонентів:

- інструмент командного рядка Helm, що забезпечує інтерфейс користувача для всіх функцій Helm;
- супутній серверний компонент Helm, який працює на кластері Kubernetes, прослуховує команди Helm та обробляє конфігурації та розгортання версій програмного забезпечення у кластері;
- формат пакетів Helm, званий charts;
- офіційний репозиторій charts з готовими пакетами charts для популярних проектів програмного забезпечення з відкритим вихідним кодом.

Пакети Helm мають формат charts і складаються з кількох файлів конфігурації YAML та шаблонів, що перетворюються на файли маніфесту Kubernetes.

Під час інсталяції пакета Helm об'єднує шаблони пакета із заданою користувачем конфігурацією та стандартними значеннями з файлу `value.yaml`. Для цих пакетів виконується рендеринг у маніфестах Kubernetes, які потім розгортаються у Kubernetes API. У цьому створюється реліз, тобто конфігурація і розгортання конкретного пакета.

Розуміння концепції релізів дуже важливе, оскільки один і той же додаток можна розгорнути в кластері кілька разів. При кожному оновленні створюється нова редакція релізу, і у разі виникнення проблем Helm дозволяє легко повертатися до попередніх редакцій. На рисунку 2.2 можна побачити архітектуру Helm.

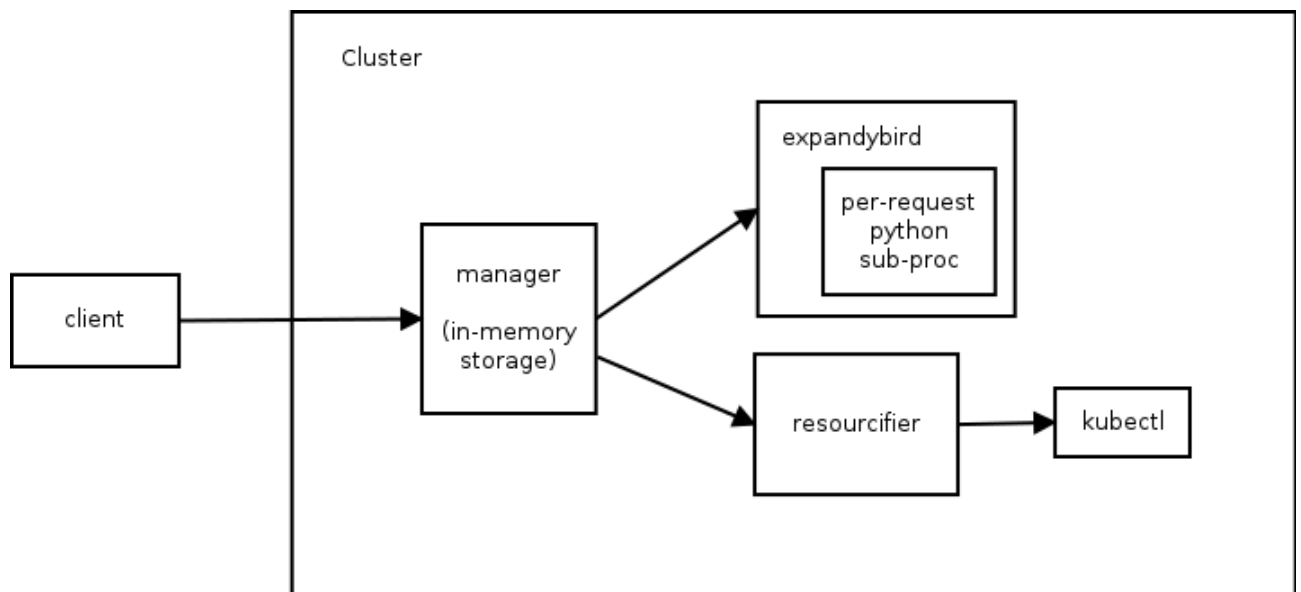


Рисунок 2.2. Архітектура Helm.

2.2 Проектування діаграми використання

Візуальне моделювання в UML можна представити як специфічний процес, який отримує рівень за рівнем від загальної абстрактної концептуальної моделі вихідної системи до логічної моделі та до фізичної моделі відповідної програмної системи. Для досягнення цих цілей моделі

спочатку будуються як такі звані діаграмні варіанти використання. Він вибирає функціональне призначення системи, тобто те, що робить систему в процесі роботи. Діаграма варіантів використання – це раннє концептуальне проявлення чи концептуальна модель процесу розробки та впровадження системи.

Розробка діаграми прецедентів визначає загальні межі та контекст предметних областей, що моделюються на ранніх стадіях систем проектування, формулює загальні вимоги до функціональної поведінки проектованої системи та розробляє вихідні концепції системи. Система для доопрацювання у вигляді логічних та фізичних моделей, підготовки вихідної документації для взаємодії розробників системи з їх замовниками та користувачами.

Суть цю схему у наступному. Проектована система представлена у вигляді ряду сутностей або акторів, які взаємодіють із системою за допомогою таких званих варіантів використання. У той же час актори — це сутності, що взаємодіють із системою ззовні. Людина, технічний пристрій, програма або інша система, які можуть бути включені в модельну систему, визначену розробником.

Потім варіанти використання допомагають описати послуги, які надають системі суб'єктів. Іншими словами, кожен варіант використання вибирає певний набір дій, який виконує система при взаємодії з акторами. При цьому нічого не йдеться про те, як реалізуються взаємодії гравців із системою. У загальному випадку діаграми прецедентів є особливим видом графів. Це графічне визначення для представлення конкретних варіантів використання, дійових осіб, можливо, деяких інтерфейсів та взаємозв'язків між цими елементами. При цьому окремі компоненти схеми можуть бути укладені в прямокутники, які представляють всю систему, що проектується.

Зверніть увагу, що відносини на цьому графіку є лише відносинами, зафіксованими між суб'єктами та варіантами використання, які в сукупності описують послуги або функціональні вимоги системи, що моделюється.

сутності предметної області без урахування внутрішньої структури цієї сутності.

Кожен варіант використання визначає послідовність дій, які повинні бути виконані проектованої системою при взаємодії її з відповідним актором. Діаграма варіантів може доповнюватися пояснювальним текстом, який розкриває зміст або семантику складових її компонентів. Мета діаграми використання полягає в тому, щоб визначити закінчений аспект або фрагмент поведінки деякої сутності без розкриття внутрішньої структури цієї сутності. В якості такої сутності може виступати вихідна система або будь-який інший елемент моделі, який володіє власною поведінкою, подібно підсистемі або класу в моделі системи.

Кожен варіант використання визначає набір дій, які має виконувати проектована система під час взаємодії з відповідним актором. Діаграма варіантів може бути доповнена описовим текстом, що розкриває зміст чи семантику її складових. Ціль діаграми використання полягає в тому, щоб визначити повний аспект або частину поведінки об'єкта, не розкриваючи внутрішню структуру об'єкта. Такими об'єктами можуть бути вихідна система або інші елементи моделі, що мають власну поведінку, наприклад, підсистеми або класи в моделі системи.

Кожному варіанту використання відповідає окремий сервіс, що надає змодельований об'єкт або систему на запит користувача (актора). Інакше кажучи, він визначає, як застосовується ця сутність. Служба, створена за запитом користувача, є повним набором дій. Це означає, що після того, як система завершила обробку запиту користувача, вона має повернутися у вихідний стан, готовий до виконання наступних запитів.

Сценарії використання описують не тільки взаємодію між користувачем та об'єктом, але й реакцію об'єкта на отримання окремих повідомлень від користувача та отримання цих повідомлень за межами об'єкта. Варіанти використання можуть включати відомості про реалізацію

служби та опис різних виняткових ситуацій, таких як правильна обробка системних помилок.

Діаграму варіантів використання можна бачити на рис. 2.3.

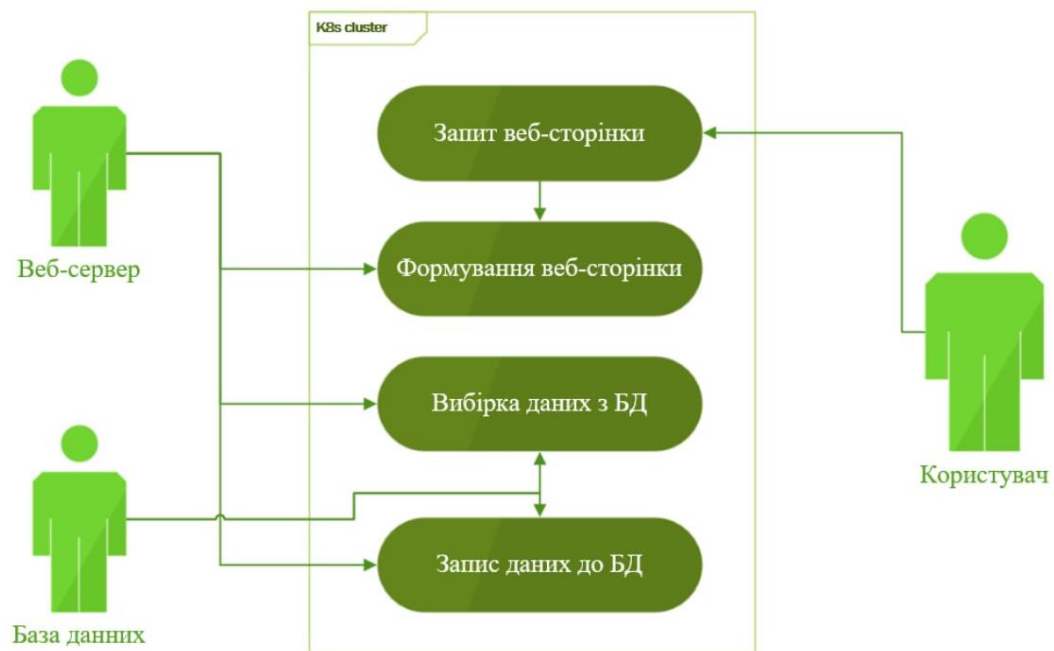


Рисунок 2.3 – Діаграма варіантів використання.

Опис ролей у діаграмі використання:

Користувач – браузер на стороні клієнта, що виконує запит на отримання веб-сторінки.

2. База даних – база даних SQL.

3. Веб-сервер – розроблюваний NodeJS додаток адресної книги.

Опис прецедентів проекту:

1. Запит веб-сторінки – HTTP запит на отримання веб-сторінки.

2. Формування веб-сторінки – динамічне створення коду веб-сторінки з врахуванням внесених змін в БД.

3. Вибірка даних з бази даних – вибірка запитів з історії до БД.

4. Запис даних до бази даних – внесення змін до БД.

2.3 Проектування інфраструктури проекту

В даному проекті є локально встановлений Terraform, який створює інфраструктуру в GCP для двох середовищ dev і prod. Файли .tfstates для цих середовищ зберігаються в хмарному сховищі, а саме Google Cloud Storage, який було розгорнено також інструментом Terraform до побудови основної інфраструктури. У Github ми маємо 2 середовища для dev і prod. Вони відрізняються лише файлом config.yml, який використовує CircleCi.

Залежно від того, до якої гілки надсилаються оновлення, CircleCI витягує оновлення з модифікованої гілки репозиторію Github, починає створювати образ, надсилає його до реєстру контейнерів Google Container Registry. Після цього новий контейнер розгортається в кластері Kubernetes.

Таким чином повну схему проекту визначено на рис. 2.3

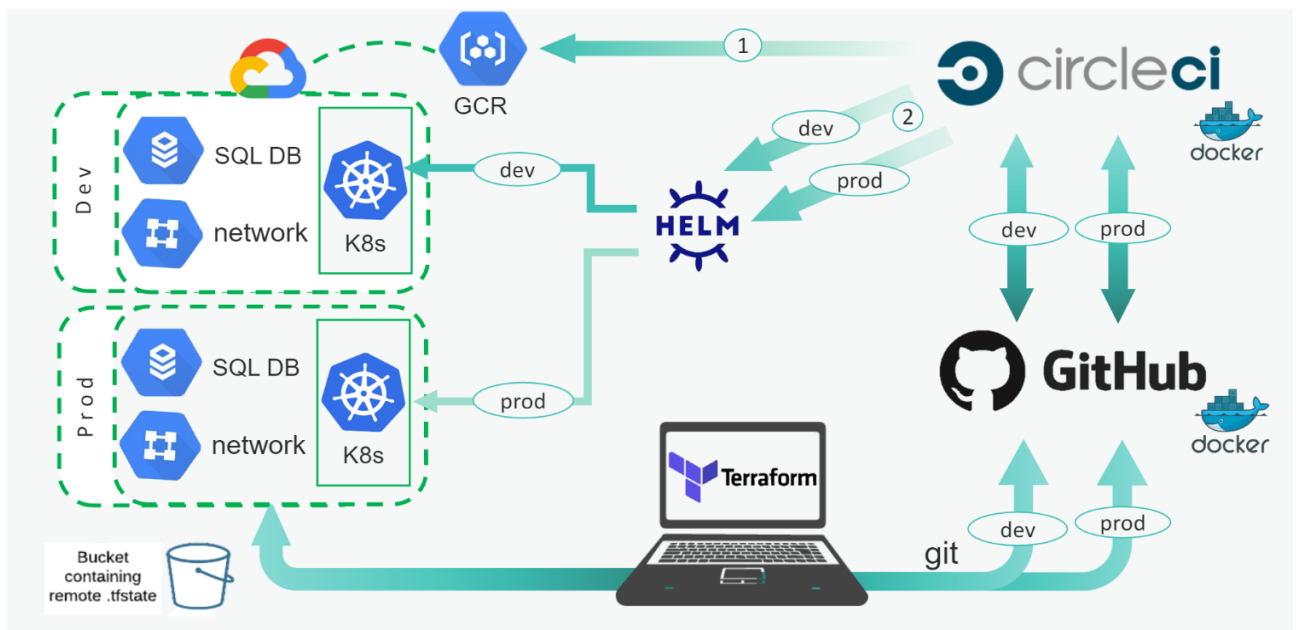


Рисунок 2.3 – Схема проекту.

2.4 Висновки за розділом

Програмне забезпечення у вигляді NodeJS додатку адресної книги було контейнеризоване та повністю підготоване до системи оркестрації Kubernetes. Щоб більш вдало контролювати навантаження на кластер у типі

сервісу Kubernetes було обрано LoadBalancer. Також було спроектовано загальну архітектуру системи.

У даному проекті буде повністю реалізована практика IaC, оскільки всі ресурси, що будуть створюватися, описані у вигляді коду для інструмента Terraform.

Для даної роботи було обрано хмарну платформу Google Cloud, а CI/CD інструментом – CircleCI. Головними факторами вибору було : якість документації та фінансові витрати.

3 РОЗРОБКА ХМАРНОЇ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

3.1 Підготовка до розробки основної інфраструктури

Підготовчі дії складаються з створення ресурсу Google Cloud Storage для зберігання бекенду інструмента Terraform. Результат виводу команди `terraform apply` можна бачити на рисунку 3.1.

```
pademi@5CD116MKBD:~/address-book/Conf/gcs_bucket$ terraform apply

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
+ create

Terraform will perform the following actions:

# google_storage_bucket.bucket will be created
+ resource "google_storage_bucket" "bucket" {
  + bucket_policy_only = (known after apply)
  + force_destroy      = false
  + id                 = (known after apply)
  + location           = "US"
  + name               = "task-busket222111-tf1"
  + project            = "sunlit-inn-329918"
  + self_link          = (known after apply)
  + storage_class       = "STANDARD"
  + uniform_bucket_level_access = (known after apply)
  + url                = (known after apply)

  + versioning {
    + enabled = true
  }
}

Plan: 1 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.

Enter a value: yes

google_storage_bucket.bucket: Creating...
google_storage_bucket.bucket: Creation complete after 2s [id=task-busket222111-tf1]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Рисунок 3.1 - Результат виводу команди `terraform apply` для GCS.

3.2 Побудова основної інфраструктури

У основному коді інфраструктури було реалізовано модульний опис інфраструктури. Тобто у головному файлі `main.tf` потрібно лише описати

посилання на модулі і вказати змінні, що будуть використовуватися при побудові інфраструктури.

У модулі “Cluster” відображена побудова Kubernetes кластеру і його конфігурація, а саме, яку мережу він буде використовувати, кількість нод, його назва, регіон дата центру та сертифікати для збору логів та моніторингу деяких метрик. Вивід команди terraform plan до модулю “Cluster” можна бачити на рисунку 3.2 - 3.4.

```
# google_container_cluster.test will be created
+ resource "google_container_cluster" "test" {
  + cluster_ipv4_cidr           = (known after apply)
  + datapath_provider           = (known after apply)
  + default_max_pods_per_node   = (known after apply)
  + enable_binary_authorization = false
  + enable_intranode_visibility = (known after apply)
  + enable_kubernetes_alpha     = false
  + enable_legacy_abac          = false
  + enable_shielded_nodes       = true
  + endpoint                    = (known after apply)
  + id                          = (known after apply)
  + initial_node_count          = 1
  + label_fingerprint           = (known after apply)
  + location                     = "eu-west3"
  + logging_service              = (known after apply)
  + master_version               = (known after apply)
  + monitoring_service           = (known after apply)
  + name                         = "test"
  + network                     = "default"
  + networking_mode              = "VPC_NATIVE"
  + node_locations               = (known after apply)
  + node_version                 = (known after apply)
  + operation                    = (known after apply)
  + private_ipv6_google_access   = (known after apply)
  + project                      = (known after apply)
  + remove_default_node_pool     = true
  + self_link                    = (known after apply)
  + services_ipv4_cidr           = (known after apply)
  + subnetwork                   = "default"
  + tpu_ipv4_cidr_block          = (known after apply)

  + addons_config {
    + cloudrun_config {
      + disabled = (known after apply)
    }
  }
}
```

Рисунок 3.2 – Вивід команди terraform plan для модулю “Cluster”.

```

# google_container_node_pool.test_nodes will be created
+ resource "google_container_node_pool" "test_nodes" {
  + cluster              = "test"
  + id                   = (known after apply)
  + initial_node_count   = (known after apply)
  + instance_group_urls  = (known after apply)
  + location             = "eu-west3"
  + managed_instance_group_urls = (known after apply)
  + max_pods_per_node    = (known after apply)
  + name                 = "test"
  + name_prefix          = (known after apply)
  + node_count           = 1
  + node_locations       = (known after apply)
  + operation            = (known after apply)
  + project              = (known after apply)
  + version              = (known after apply)

  + management {
    + auto_repair = (known after apply)
    + auto_upgrade = (known after apply)
  }

  + node_config {
    + disk_size_gb = (known after apply)
    + disk_type    = (known after apply)
    + guest_accelerator = (known after apply)
    + image_type    = (known after apply)
    + labels        = (known after apply)
    + local_ssd_count = (known after apply)
    + machine_type  = "e2-micro"
    + metadata      = {
      + "disable-legacy-endpoints" = "true"
    }
    + oauth_scopes = [
      + "https://www.googleapis.com/auth/compute",
      + "https://www.googleapis.com/auth/devstorage.read_only",
    ]
  }
}

```

Рисунок 3.3 – Вивід команди terraform plan для модулю “Cluster”.

```

    ]
  + preemptible = true
  + service_account = (known after apply)
  + tags = [
    + "gke-node",
  ]
  + taint = (known after apply)

  + shielded_instance_config {
    + enable_integrity_monitoring = (known after apply)
    + enable_secure_boot         = (known after apply)
  }

  + workload_metadata_config {
    + mode = (known after apply)
  }
}

+ upgrade_settings {
  + max_surge = (known after apply)
  + max_unavailable = (known after apply)
}
}

Plan: 2 to add, 0 to change, 0 to destroy.

Changes to Outputs:
+ client_certificate = (known after apply)
+ client_key        = (known after apply)
+ cluster_ca_certificate = (known after apply)
+ cluster_name      = "test"
+ public_endpoint    = (known after apply)

```

Рисунок 3.4 – Вивід команди terraform plan для модулю “Cluster”.

У модулі “Init” terraform локально збирає імедж контейнеризованої адресної книги, після авторизується за допомогою токена у GCP і надсилає імедж з тегом init у Google Container Registry. Вивід команди terraform plan можна бачити на рисунку 3.5.

```
Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
+ create

Terraform will perform the following actions:

# null_resource.docker will be created
+ resource "null_resource" "docker" {
  + id = (known after apply)
}

Plan: 1 to add, 0 to change, 0 to destroy.
```

Рисунок 3.5 – Вивід команди terraform plan для модулю “Init”.

Модуль під назвою “Kuber” використовує провайдер “kubernetes” для того щоб передати чутливу інформацію у кластер. Під чутливою інформацією мається на увазі ір адреса БД, ім’я юзеру БД та його пароль, а також ім’я БД. Ці дані беруться з файлу secret.tfvars.

Модуль “Network” відповідає за створення мережі та VPC групи для того щоб БД і кластер могли спілкуватися у приватній мережі для більшої безпеки. Для цього необхідно створити саму приватну мережу і помістити туди кластер та БД. Також у кластера буде публічна ір адреса, щоб контейнеризований додаток можна було знайти в інтернеті. Вивід команди terraform plan для модулю “Network” зафіксовано на рисунках 3.6-3.7.

```

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
+ create

Terraform will perform the following actions:

# google_compute_global_address.private_ip_address will be created
+ resource "google_compute_global_address" "private_ip_address" {
  + address           = (known after apply)
  + address_type      = "INTERNAL"
  + creation_timestamp = (known after apply)
  + id                = (known after apply)
  + name              = "test1"
  + network            = (known after apply)
  + prefix_length     = 16
  + project            = (known after apply)
  + purpose            = "VPC_PEERING"
  + self_link          = (known after apply)
}

# google_compute_network.network will be created
+ resource "google_compute_network" "network" {
  + auto_create_subnetworks = false
  + delete_default_routes_on_create = false
  + gateway_ipv4             = (known after apply)
  + id                       = (known after apply)
  + mtu                       = (known after apply)
  + name                      = "test"
  + project                  = "sunlit-inn-329918"
  + routing_mode              = (known after apply)
  + self_link                 = (known after apply)
}

# google_compute_subnetwork.subnet will be created
+ resource "google_compute_subnetwork" "subnet" {
  + creation_timestamp = (known after apply)
  + external_ipv6_prefix = (known after apply)
  + fingerprint        = (known after apply)
  + gateway_address     = (known after apply)

```

Рисунок 3.6 – Вивід команди terraform plan для модулю “Network”.

```

+ gateway_address     = (known after apply)
+ id                  = (known after apply)
+ ip_cidr_range       = "10.10.0.0/24"
+ ipv6_cidr_range     = (known after apply)
+ name                = "test"
+ network              = "test"
+ private_ipv6_google_access = (known after apply)
+ project              = (known after apply)
+ purpose              = (known after apply)
+ region              = "eu-west3"
+ secondary_ip_range   = (known after apply)
+ self_link            = (known after apply)
+ stack_type           = (known after apply)
}

# google_service_networking_connection.private_vpc_connection will be created
+ resource "google_service_networking_connection" "private_vpc_connection" {
  + id                = (known after apply)
  + network            = (known after apply)
  + peering            = (known after apply)
  + reserved_peering_ranges = [
    + "test1",
  ]
  + service            = "servicenetworking.googleapis.com"
}

Plan: 4 to add, 0 to change, 0 to destroy.

Changes to Outputs:
+ network_id      = (known after apply)
+ network_name    = "test"
+ subnetwork_name = "test"

```

Рисунок 3.7 – Вивід команди terraform plan для модулю “Network”.

У модулі “SQL” створюється база даних. Обов'язково описуємо її версію, регіон знаходження та ім'я. Також вказуємо тип віртуальної машини на якій буде розгорнено сервер бази даних. У конфігурації мережі описуємо що публічна IP адреса у базі даних відсутня є тільки приватна ip-адреса яка створюється у модулі «Network». Також у даному модулі створюється юзер бази даних. Створюється його ім'я пароль та до якої віртуальної машини з базою даних він буде відноситися, також прописуємо root юзера даної БД. Вивід команди terraform plan для модулю “SQL” можна бачити на рисунку 3.8 – 3.10.

```
# module.sql.google_sql_database_instance.sql will be created
+ resource "google_sql_database_instance" "sql" {
  + connection_name      = (known after apply)
  + database_version     = "MYSQL_5_6"
  + deletion_protection = false
  + first_ip_address     = (known after apply)
  + id                   = (known after apply)
  + ip_address           = (known after apply)
  + master_instance_name = (known after apply)
  + name                 = (known after apply)
  + private_ip_address   = (known after apply)
  + project              = (known after apply)
  + public_ip_address    = (known after apply)
  + region               = "europe-west3"
  + self_link            = (known after apply)
  + server_ca_cert       = (known after apply)
  + service_account_email_address = (known after apply)

  + replica_configuration {
    + ca_certificate      = (known after apply)
    + client_certificate = (known after apply)
    + client_key         = (known after apply)
    + connect_retry_interval = (known after apply)
    + dump_file_path      = (known after apply)
    + failover_target     = (known after apply)
    + master_heartbeat_period = (known after apply)
    + password            = (sensitive value)
    + ssl_cipher          = (known after apply)
    + username            = (known after apply)
    + verify_server_certificate = (known after apply)
  }

  + settings {
    + activation_policy      = "ALWAYS"
    + availability_type     = "ZONAL"
    + disk_autoresize       = true
    + disk_autoresize_limit = 0
  }
}
```

Рисунок 3.8 – Вивід команди terraform plan для модулю “ SQL ”.

```

+ settings {
  + activation_policy      = "ALWAYS"
  + availability_type     = "ZONAL"
  + disk_autoresize       = true
  + disk_autoresize_limit = 0
  + disk_size             = (known after apply)
  + disk_type             = "PD_SSD"
  + pricing_plan          = "PER_USE"
  + tier                   = "db-f1-micro"
  + user_labels           = (known after apply)
  + version               = (known after apply)

  + backup_configuration {
    + binary_log_enabled = (known after apply)
    + enabled            = (known after apply)
    + location           = (known after apply)
    + point_in_time_recovery_enabled = (known after apply)
    + start_time         = (known after apply)
    + transaction_log_retention_days = (known after apply)

    + backup_retention_settings {
      + retained_backups = (known after apply)
      + retention_unit   = (known after apply)
    }
  }

  + ip_configuration {
    + ipv4_enabled = false
    + private_network = (known after apply)
  }

  + location_preference {
    + follow_gae_application = (known after apply)
    + zone                   = (known after apply)
  }
}

```

Рисунок 3.9 – Вивід команди terraform plan для модулю “SQL”.

```

# module.sql.google_sql_user.master will be created
+ resource "google_sql_user" "master" {
  + host      = "%"
  + id        = (known after apply)
  + instance  = (known after apply)
  + name      = "root"
  + password  = (sensitive value)
  + project   = (known after apply)
}

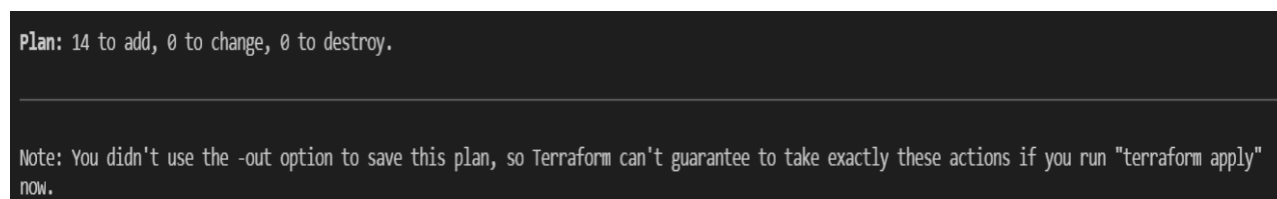
# module.sql.google_sql_user.sql_user will be created
+ resource "google_sql_user" "sql_user" {
  + host      = "%"
  + id        = (known after apply)
  + instance  = (known after apply)
  + name      = "userdb"
  + password  = (sensitive value)
  + project   = (known after apply)
}

```

Рисунок 3.10 – Вивід команди terraform plan для модулю “SQL”.

У модулі «Helm» утиліта terraform авторизується у Github для того щоб вичитати з закритого приватного репозиторію чарти Helm. Для того щоб зробити ініціалізуючу ревізію контейнерезованого додатку адресної книги. Також ці чарти будуть використовуватися CircleCI щоб робити наступні ревізії після ініціалізуючого.

Загальна кількість ресурсів, які плануються до додавання у Google Cloud Platform відображено на рисунку 3.11.



```
Plan: 14 to add, 0 to change, 0 to destroy.
```

```
Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to take exactly these actions if you run "terraform apply" now.
```

Рисунок 3.11 – Вивід команди terraform plan для всіх модулів.

3.3 Розробка чартів Helm - Charts

Чарти – це спеціальний набір файлів для змоги зв’язатися з утилітою kubectl та примінити зміни в контейнері, ссилаючись на його тег. У даному проекті для можливості більш гнучкого розроблення, оскільки, якщо відбуваються зміни в коді чи порту на якому знаходиться контейнер це можливо змінити з одного місця, що забезпечує мінімальну кількість простою. Чарти для даного проекту зберігаються у окремому приватному репозиторію. Доступ до нього забезпечується логіном у Github, використовуючи токен, який був заздалегідь зроблений мануально.

Helm-Charts зовні схожі на маніфести K8S. Різниця лише тому, що змінні в маніфесті K8S вказуються безпосередньо у самому маніфесті. Отже, це те, що ви називаєте жорстким кодуванням, Helm-Charts використовує посилання. Додані до цих змінних змінні знаходяться в окремому файлі, який легко редагувати.

В рамках проекту було розроблено чотири типи діаграм: deployment.yaml, ingress.yaml, configmap.yaml та service.yaml.

У файлі `deployment.yaml` має бути описана версія API, яку Helm-Charts використовує для зв'язку з `kubectl` та встановлення оновлень. Також нам потрібно назвати чарт, привласнити йому якісь мітки, прикріпити специфікатор, який пов'язує всі написані чарти, вказати ім'я контейнера та порт, на якому він працює. Для кожної робочої області кількість реплік дорівнює 1. Отже, 1 pod = 1 вузол. Порт 8080 був вибраний для програми адресної книги для доступу до Інтернету.

`Service.yaml` - це опис типу служби, яку ваш контейнерний додаток запускає через Інтернет. Для цього проекту використовувався тип `LoadBalancer`, оскільки очікуване навантаження розподіляється між двома вузлами. Він також показує протокол, який користувач використовуватиме для доступу до адресної книги, TCP. Специфікатор було вказано як `deployment.yaml`.

`Ingress.yaml` визначає правила маршрутизації для кращого розподілу навантаження між вузлами та модулями.

Загальну організацію чартів можна бачити на рисунку 3.12

```
pademi@5CD116MKBD:~/helm$ tree
.
├── helm-chart
│   ├── Chart.yaml
│   ├── templates
│   │   ├── configmap.yaml
│   │   ├── deployment.yaml
│   │   ├── ingress.yaml
│   │   └── service.yaml
│   └── values.yaml
├── index.yaml
└── my-first-2.tgz
2 directories, 8 files
```

Рисунок 3.12 – Ієрархічна організація Helm-Charts

3.4 Розробка конфігурації для CircleCI

Для роботи CircleCI була створена окрема директорія, а саме .circlseci. У цій директорії зберігається файл config.yml. В кожній гілці Github є свій config.yml, оскільки, CircleCI відповідає за 2 робочих простору, збирає додаток для кожного робочого простору.

Через те що у пайплайн даного інтегратору коду використовуються чутливі дані було прийнято рішення використовувати вбудовану систему захисту чутливої інформації та створити власний контекст. Чутливі дані будуть підставлятися у код, як змінні середовища. Кількість та назви змінних можна бачити на рисунку 3.13.

Environment Variables

Environment variables are available to any job that requests this context. See [Using Environment Variables](#) documentation.

[Add Environment Variable](#)

Name	Value	
CLIENT_ID	****5191	×
GCLOUD_ACC	****Cn0K	×
GCLOUD_PRIVATE_KEY_ID	****4653	×
GCLOUD_SERVICE_KEY	****_\\n	×
GCP_AUTH_X509	****.com	×
GCP_PROJECT_NAME	****irst	×
GITHUB_TOKEN	****rKTn	×
GOOGLE_COMPUTE_ZONE	****t3-c	×
GOOGLE_EMAIL	****.com	×
GOOGLE_PROJECT_ID	****9117	×

Рисунок 3.13 – Чутливі змінні, як змінні середовища.

У CircleCI має декілька типів так званих ранерів, а саме, віртуальна машина та докер контейнер. Можна використовувати будь-який докер контейнер створений вами або завантажений іншими юзерами в DockerHub. У даному проєкті буде використано офіційний контейнер компанії Google з назвою google/cloud-sdk. Він базується на мінімальній версії Ubuntu 18.04 та інсталюваному Cloud SDK для керування ресурсами у Google Cloud Platform.

У першому середовищі CircleCI копіює GitHub репозиторій у якому знаходиться контейнеризований додаток адресної книги. Після цього CircleCI збирає контейнер адресної книги. Після встановлення всіх залежностей та запуску програми у контейнері, яка написана на NodeJS, додає тег. У якості тегу використовується 7 перших символів коміту, який було сформульовано при пуші змін у Github репозиторій. Після цього використовує змінну GCLOUD_ACC так щоб дешифрувати її, оскільки, це зашифрований ключ від сервіс-акаунту у GCP, і додати до команди докер логіну, для вдалої авторизації у Google Container Registry для подальшого версіонування у GCR.

У другому середовищі також використовується контейнер google/cloud-sdk у якості середовища розробки. Першим ділом CircleCI бере змінну GCLOUD_ACC та дешифрує її та записує у файл. Після цього командою gcloud auth activate-service-account ю активує сервіс аккаунт у gcp встановлює ID проєкту gcp та зону, у якій мій розгортається ця кластер проєкту. Після цього командою gcloud container clusters get-credentials йде до хмари Google та вичитує чутливу інформацію кластеру для змоги інсталювати зміни. Далі завантажує Хелм з офіційного github репозиторію та інсталює його. Після цього Хелм оновлює директорію чартів та інсталює новий контейнер у K8S кластер. Після цього CircleCI завершує свою роботу.

3.5 Висновки за розділом

У даному розділі було розроблено повну інфраструктуру проекту який виконується на хостах Google Cloud Platform. Повністю вся інфраструктура підлягає практики `infrastructure as a code`. Було розгорнено 14 ресурсів у GCP, а саме K8S кластер, 2 ноди, SQL БД, користувача БД, приватну мережу для бази даних та кластеру, також було використано провайдер Хелм для інсталювання першого, ініціалізуючого, контейнеру у кластер і провайдер `kubernetes` для передачі чутливої інформації до БД.

Також були розроблені `Helm-Charts` для темплювання маніфестів `kubernetes`. У даних чартах було вибрано селектор для зв'язку усіх чартів, назва контейнеру, порт роботи в контейнеру у кластері та тип сервісу маніфесту. Тип сервісу - `LoadBalancer`. Завдяки чому балансувальник навантаження може розподіляти навантаження на двох нодах. При описі `deployment.yaml` було вказано 1 под, що реалізує практику 1 под = 1 нода. У `ingress.yaml` було прописано як саме `LoadBalancer` розподіляє запити для балансування навантаження.

У CircleCI було створено два середовища. Ранером було вибрано контейнер компанії Google, а саме, `google/cloud-sdk`. Перше середовище призначено на створювання контейнеру, додавання до нього тегу та відправку його у Google Container Registry для можливості версіонування. Друге зв'язується з Google Cloud Platform, вичитує чутливу інформацію з кластеру, оновлене локальні `Helm-Charts` та інсталюйте новий, змінений, контейнер у K8S кластер.

ВИСНОВКИ

Кваліфікаційна робота присвячена дослідженню та розробці хмарної мікросервісної архітектури у Google Cloud Platform з використанням інструментів оркестрації та технології контейнеризації. Контейнеризація – методологія розробки, коли усі залежності та розроблювані файли, компоненти та саме ПЗ об'єднується у ізольоване середовище.

В роботі був проведений аналіз та розроблена хмарна мікросервісна інфраструктура для дослідження систем нейроідентифікації нелінійних об'єктів за допомогою оркестратора контейнерів Kubernetes для обслуговування веб-серверу адресної книги.

Для виконання поставленої мети були виконані наступні завдання: було розроблено мікросервісну архітектуру для кластеру Kubernetes; було розроблено CI/CD інфраструктуру для контейнеризованого веб-серверу адресної книги; зроблені Helm-Chart для більш легкого розгортання додатку.

Таким чином, дана система являє собою контейнеризований веб-сервер з базою даних, яка надає можливість blue-green розгорнення та балансування вхідного трафіку кластеру.

Було розроблено програмне забезпечення у вигляді NodeJS додатку адресної книги було контейнеризоване та повністю підготоване до системи оркестрації Kubernetes. Щоб більш вдало контролювати навантаження на кластер у типі сервісу Kubernetes було обрано LoadBalancer. Також було спроектовано загальну архітектуру системи.

У даному проєкті буде повністю реалізована практика IaC, оскільки всі ресурси, що будуть створюватися, описані у вигляді коду для інструмента Terraform.

Для даної роботи було обрано хмарну платформу Google Cloud, а CI/CD інструментом – CircleCI. Головними факторами вибору було : якість документації та фінансові витрати.

Також було розроблено повну інфраструктуру проекту який виконується на хостах Google Cloud Platform. Повністю вся інфраструктура підлягає практики `infrastructure as a code`. Було розгорнено 14 ресурсів у GCP, а саме K8S кластер, 2 ноди, SQL БД, користувача БД, приватну мережу для бази даних та кластеру, також було використано провайдер Хелм для інсталювання першого, ініціалізуючого, контейнеру у кластер і провайдер `kubernetes` для передачі чутливої інформації до БД.

Були розроблені Helm-Charts для темплайвання маніфестів `kubernetes`. У даних чартах було вибрано селектор для зв'язку усіх чартів, назва контейнеру, порт роботи в контейнеру у кластері та тип сервісу маніфесту. Тип сервісу - `LoadBalancer`. Завдяки чому балансувальник навантаження може розподіляти навантаження на двох нодах. При описі `deployment.yaml` було вказано 1 под, що реалізує практику 1 под = 1 нода. У `ingress.yaml` було прописано як саме `LoadBalancer` розподіляє запити для балансування навантаження.

У CircleCI було створено два середовища. Ранером було вибрано контейнер компанії Google, а саме, `google/cloud-sdk`. Перше середовище призначено на створювання контейнеру, додавання до нього тегу та відправку його у Google Container Registry для можливості версіонування. Друге зв'язується з Google Cloud Platform, вичитує чутливу інформацію з кластеру, оновлене локальні Helm-Charts та інсталюйте новий, змінений, контейнер у K8S кластер.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. DevOps. *Вікіпедія Україна*. URL: <https://bit.ly/3EAyz8b>
2. Водоспадна модель. *Вікіпедія Україна*. URL: <https://bit.ly/31Bt8Y7>
3. Гнучка розробка програмного забезпечення. *Вікіпедія Україна*. URL: <https://bit.ly/3y1cQUe>
4. Що таке Agile і як його застосовувати в бізнесі. URL: <https://bit.ly/3y0Rer8>
5. Що таке методологія розробки CI/CD. URL: <https://bit.ly/3pD3xGe>
6. DevOps методологія і її вплив на хмарні системи. URL: <https://bit.ly/3GkPnjV>
7. Аналіз вимог до системи. *Вікіпедія Україна*. URL: <https://bit.ly/3drbg11>
8. Діаграма прецедентів. *Вікіпедія Україна*. URL: <https://bit.ly/3oF2973>
9. Діаграма варіантів використання. URL: <https://bit.ly/332q0oB> (дата звернення 16.09.2021).
10. Розробка UML діаграми варіантів використання. URL: <https://bit.ly/3dzDnOZ>
11. Документація Kubernetes. URL: <https://bit.ly/3DwZq3A> (дата звернення 15.10.2021).
12. Інфраструктура як код. *Вікіпедія Україна*. URL: <https://bit.ly/3oxmnPT>
13. Infrastructure as Code: базові принципи vs інструменти, що еволюціонують. URL: <https://bit.ly/3duWfPq>
14. Брікман Е.В. Terraform на рівні коду: навч. посіб. Київ: ЦУЛ, 2018. 450с.