РЕФЕРАТ

Пояснювальна записка: 58 с., 28 рис., 1 табл., 8 джерел, 1 додаток.

Мета роботи: Модернізація комп’ютерної системи під технології контейнеризації на основі хмарної інфраструктури

У минулому році була створена комп’ютерна система, яка допомагає учням та їх батькам стежити за успіхом навчання.

Спочатку війни з’явилась загроза знищення шкільного обладнання, так як комп’ютерна система була розташована на шкільному сервері, адміністрацією школи було прийнято рішення перенести додаток в хмарне середовище AWS за допомогою технології контейнеризації

Container, Docker, Microsoft Azure, Google Cloud Run, DevOps, EKS, ECS, Fargate, Image, Serverless, AWS, EC2, VPC, Software

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

Контейнери - це спосіб стандартизації розгорнення програми та відокремлення її від загальної інфраструктури. Примірник програми запускається в ізольованому середовищі, що не впливає на основну операційну систему.

Docker – це платформа для розробки, доставки та запуску контейнерних програм. Docker дозволяє створювати контейнери, автоматизувати їх запуск та розгортання, керує життєвим циклом. Він дозволяє запускати безліч контейнерів на одній машині.

Amazon Web Services (AWS) – це найпоширеніша у світі хмарна платформа з найширшими можливостями, що надає понад 200 повнофункціональних сервісів для центрів обробки даних по всій планеті.

Microsoft Azure – хмарна платформа компанії Microsoft. Надає можливість розробки, виконання додатків і зберігання даних на серверах, розташованих в розподілених дата-центрах. Хмара Azure була оголошена в жовтні 2008 року під кодовою назвою»Project Red Dog"

Google Cloud Platform – надається компанією Google набір хмарних служб, які виконуються на тій же самій інфраструктурі, яку Google використовує для своїх продуктів, призначених для кінцевих споживачів, таких як Google Search і YouTube.

Nomad – це розподілений кластер високої доступності з центром обробки даних та підтримкою сервера додатків, призначений для обслуговування сучасного центру обробки даних з підтримкою довгострокових служб, послідовних завдань тощо.

ВСТУП

На сьогоднішній день важко собі уявити, життя не в онлайн форматі. За останні декілька років життя та бізнес дуже сильно змінились. Такі фактори, як COVID-19 та війна в країні майже унеможливили працювати та навчатись в одному приміщенні, через ризик захворіти або ризик обстрілу.

Тож системи, які знаходяться на фізичних серверах почали мати ще більше недоліків, тому дивлячись на переваги хмарного середовища, та важкість обслуговування on-premise рішень, IT спільнота почала активно переносити свої додатки до хмарного середовища.

Тому для підвищення якості додатка E-Завдання який знаходиться на on-premise сервері, а також поліпшення якості та швидкості обслуговування, було запропоновано перенести додаток до хмарного середовища з використанням технології контейнеризації додатку.

Метою робити є створення архітектури та мережі додатка у хмарному середовищі за допомогою використання одного з постачальників хмарного середовища.

1 СТАН ПИТАННЯ ТА ПОСТАНОВКА ЗАВДАННЯ

1.1 Аналіз існуючих додатків на ринку

Розглянемо рішення які наразі існують на ринку. Наприклад розглянемо додаток "Єдина школа". Цей додаток рекомендований Міністерством освіти і науки України.

Додаток «Єдина школа» забезпечує: педагогів, батьків та дітей - інструментами доступу до е-сервісів; керівників закладів освіти - засобами прийняття управлінських рішень; органи управління освітою – засобами організації електронної взаємодії. Система підтримує щоденну роботу закладу освіти, а саме: онлайн моніторинг успішності учнів та відвідуваності ними занять; ведення освітньої діяльності, у т.ч. у дистанційному режимі, залучення батьків до освітнього процесу; створення умов переходу на без паперову форму шкільної документації.

Розглянемо більш детально деякі можливості цього додатку, а саме електронний щоденник для батьків – модуль, розроблений з метою залучення батьків до освітнього процесу шляхом перегляду розкладу уроків їх дітей; доступу до отриманих ними оцінок, контролю відвідування школи та виконання домашнього завдання; аналізу навчальних досягнень дітей на основі статистики успішності, можливості бути в курсі актуальних новин та подій, які відбуваються у школі.

Додаток дозволяє працювати на трьох та більше платформах: web-додаток, мобільний додаток – Android, iOS, надає цілодобовий доступ до сервісів, передбачає навчання педагогів, батьків та учнів.

1.2 Характеристика і структура додатку Е-Завдання

Фізичний сервер (On-premises software) Dell PowerEdge T40 v04 знаходиться на території спеціалізованої школи №13 на якому запущений додаток Е-Завдання. Обладнання яке використовується має наступні характеристики

Таблиця 1.1- Технічні характеристики Dell PowerEdge T40 v04

Обладнання	Характеристика
Процесор	Чотири ядерний Intel Xeon E-2224G (3.5 — 4.7 ГГц)
Чипсет	Intel C246
Обсяг пам'яті	32 Гб
Тип оперативної пам'яті	UDIMM ECC DDR4-2666 МГц (4 слоти, 128 ГБ макс.)
Роз'єми	**Передня панель:** 1 x USB 3.1 Type-C 1 x USB 3.1 2 x USB 2.0 **Задня панель:** 1 x PS/2 порт для клавіатури 1 x PS/2 порт для миші 2 x USB 2.0 4 x USB 3.1 2 x DisplayPort 1 x послідовний порт 1 x аудіороз'єм **Слоти розширення:** 1 x PCI-E 3.0 x16 2 x PCI-E 3.0 x4 1 x PCI-E
Потужність БЖ	48 Вт
Жорсткий диск	HDD: 2 x 1 ТБ SSD: 2 x 250 ГБ

Переваги:

– **незалежність** – сервер знаходиться на території школи

Недоліки:

– **більше адміністративної роботи** – для адміністраторів використання фізичних машин може здатися великою роботою. Їм

потрібно підтримувати систему, чекати оновлень і завжди використовувати цю оновлену версію та захищати всі данні на сервері;

– **важко оновляти програмне забезпечення** − потрібно постійно купляти нові ліцензії на використання того чи іншого ПО.

Також роздивимося логічну топологію комп’ютерної мереж

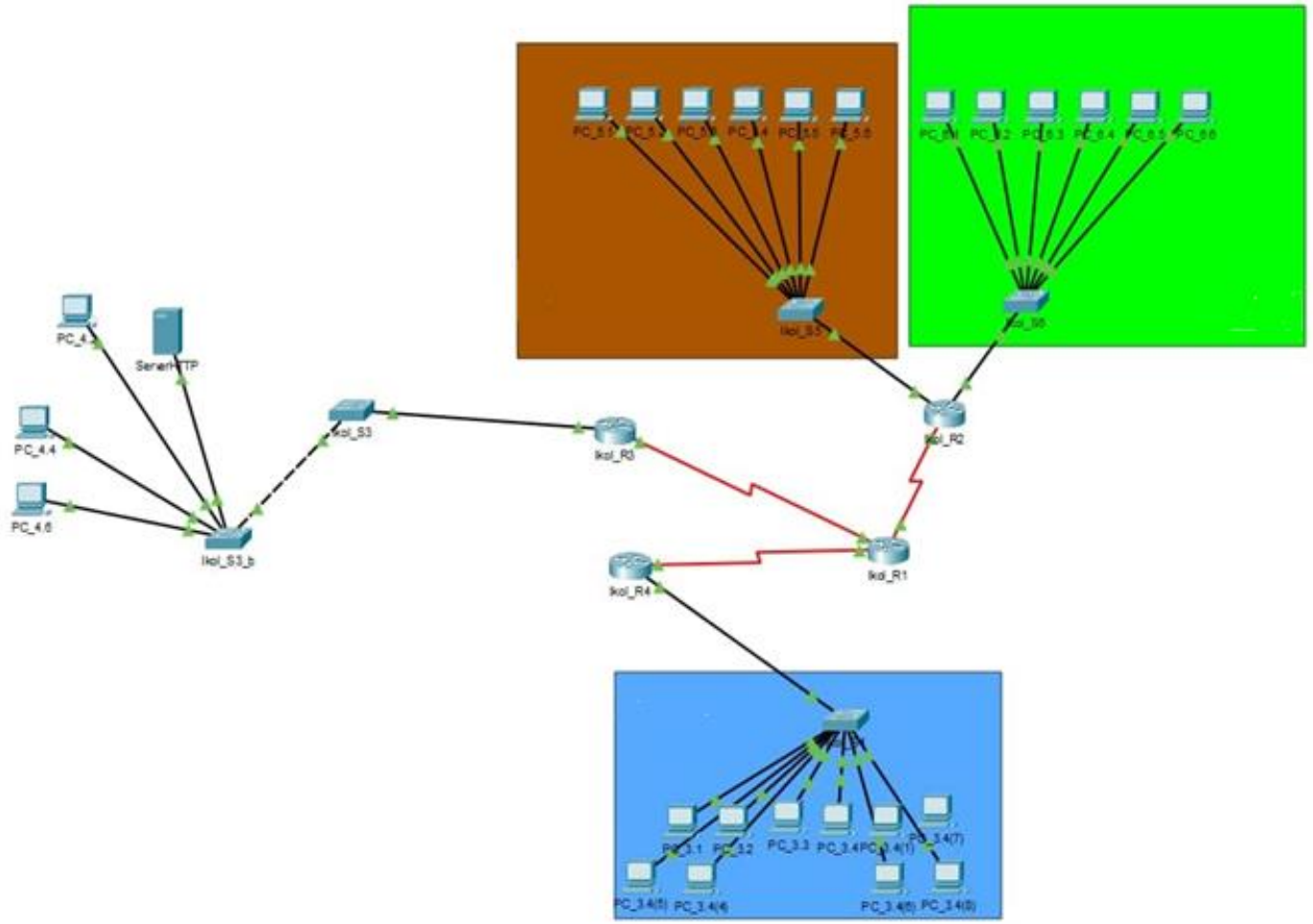

Рисунок 1.1 − Загальна архітектура мережі додатку Е-Завдання

1.3 Опис архітектури хмарних додатків

Хмарна архітектура – це спосіб поєднання технологічних компонентів для створення хмари, в якому ресурси об'єднуються за допомогою технології віртуалізації та спільно використовуються у мережі. Хмарна архітектура складається з наступних компонентів:10

– зовнішня платформа (клієнт або пристрій, що використовується для доступу до хмари);

– внутрішня платформа (сервери та сховище);

– хмарна модель додатків;

– мережа.

Водночас ці технології складають архітектуру хмарних обчислень, в якій можуть працювати програми, надаючи кінцевим користувачам можливість використовувати переваги ресурсів хмар.

1.4 Опис хмарних мереж додатків

Хмарна мережа – це тип IT-інфраструктури, у якій деякі або всі мережеві можливості та ресурси організації розміщені на публічній або приватній хмарній платформі.

Компанії можуть або використовувати локальні хмарні мережеві ресурси для створення приватної хмарної мережі, або використовувати хмарні мережеві ресурси в загальнодоступній хмарі, або гібридну хмарну комбінацію обох. Розміщення мережевих ресурсів у хмарі може передбачати використання пристроїв локально або через постачальника хмарних послуг і може включати такі мережеві служби:

– програмне забезпечення для управління мережею та доступу;

– підключення;

– віртуальні маршрутизатори;

– брандмауери та служби безпеки;

– балансувальники навантаження;

– пропускна здатність;

– мережі доставки контенту (CDN);

– віртуальні приватні мережі (VPN).

1.5 Постановка завдання

Як ми можемо побачити це застаріле рішення, яке потребує модернізації. Також важливо взяти на увагу те що за останні два роки майже усе відбувається дистанційно через COVID-19 та війною в країні.

Напрямком для цього було обрано шлях за допомогою контейнеризації та подальшим розміщенням системи до хмарного середовища з використанням хмарної мережі, що дозволить більш гнучкою використати обчислювальні ресурси. Основою додатку є мова програмування Java та фреймворків Telegram API та Spring Framework. Основуючись на цьому, хмарна архітектура має бути наступною:

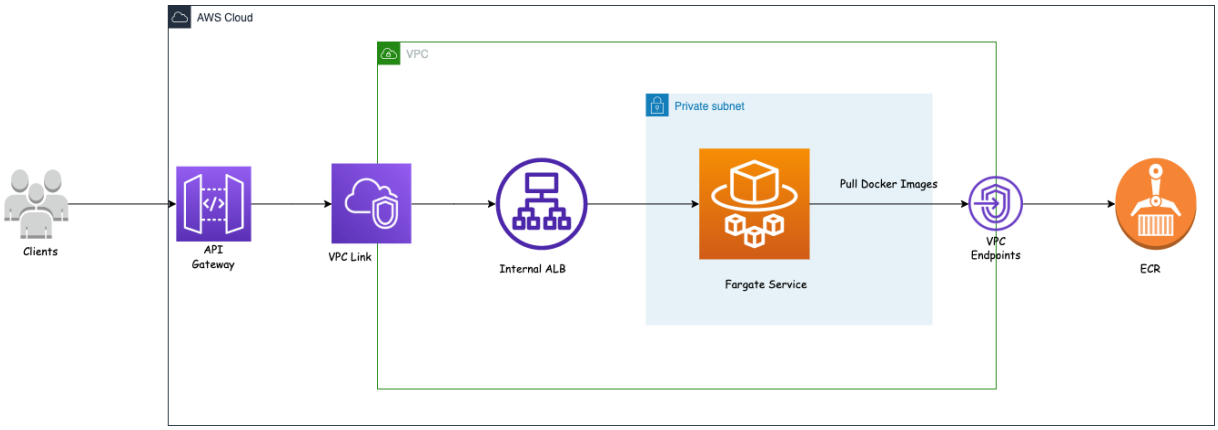

Рисунок 1.2 − Майбутня архітектура додатка

Також для цього повинна бути перенесена існуюча топологія мережі, до хмарного середовища і має бути наступною

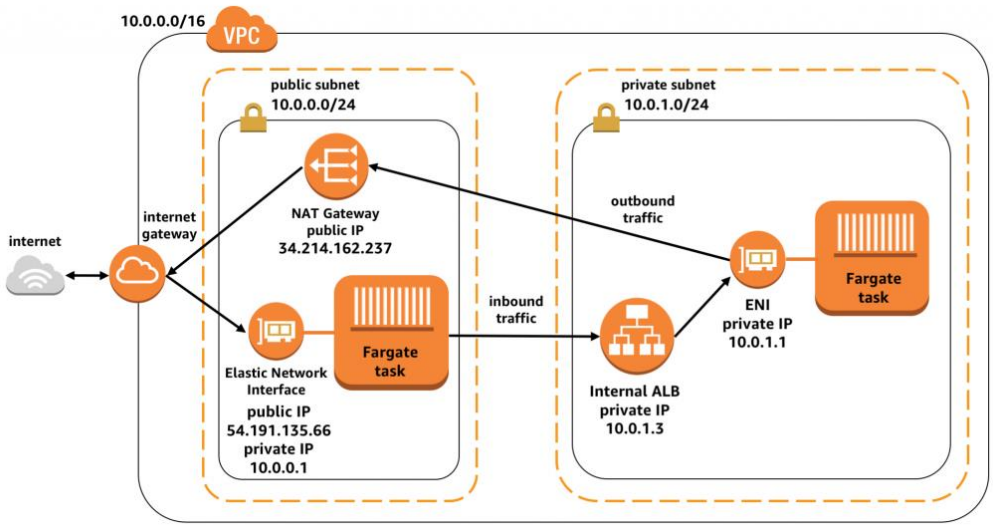

Рисунок 1.3 − Майбутня мережа додатку.

2 ТЕОРЕТИЧНИЙ РОЗДІЛ

2.1 Аналіз обчислювальних ресурсів комп’ютерної системи

Для того щоб краще розуміти та зробити вірне рішення, розглянемо доступні види обчислювальних ресурсів комп’ютерної системи.

2.1.1 Локальне програмне забезпечення

Локальне програмне забезпечення (On Premise) – встановлюється та запускається на комп’ютерах у приміщеннях (у будівлі) особи чи організації, які використовують програмне забезпечення, а не на віддаленому об’єкті, наприклад у хмарному середовищі десь в Інтернеті. Локальне програмне забезпечення іноді називають програмним забезпеченням «shrinkwrap», а зовнішнє програмне забезпечення зазвичай називають SaaS (програмне забезпечення як послуга).

Локальний підхід до розгортання та використання програмного забезпечення для бізнесу був найпоширенішим приблизно до 2005 року.

2.1.2 Віртуалізація програмного забезпечення

Віртуалізація – це технологія, яка дозволяє надавати ізольовані набори обчислювальних потужностей, абстрагованих від фізичного обладнання. Віртуальна машина може мати будь-які характеристики (значення пам'яті, частоту процесора та ін), але в рамках ресурсів фізичного пристрою. При цьому вона запускатиметься на кшталт програми всередині основної операційної системи (вона називається хост-система). Таким чином, кожна віртуальна машина працює незалежно та запускає різні операційні системи або програми, при цьому спільно використовуючи ресурси одного хост-комп'ютера.

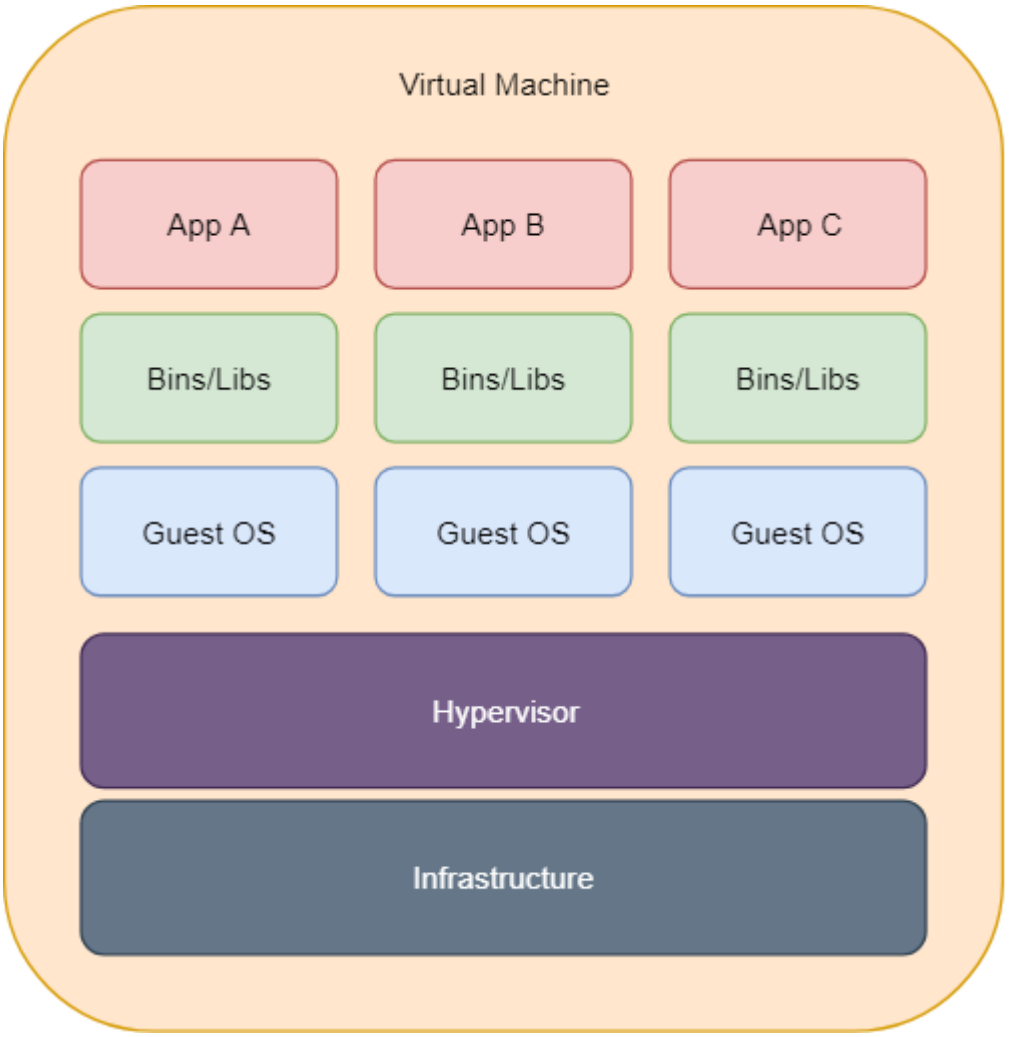

Рисунок 2.1 − Архітектура віртуалізації

Переваги:

– **незалежність** − кожна віртуальна машина працює незалежно.

– **оптимізація використання ресурсів** − використовуючи віртуальні машини, ми також можемо оптимізувати використання наших ресурсів і максимізувати сховище відповідно до потреб кожного проекту. Більшість серверів мають велику потужність і здатні запускати всі проекти одночасно, і, маючи це на увазі, ми можемо розмістити більш продуктивні програми в одній системі.

Недоліки:

– **займає багато ресурсів** - кожна віртуальна машина запускає віртуальну копію всього апаратного забезпечення, необхідного для роботи операційної системи, і це швидко додає багато циклів оперативної пам’яті та процесора;

– **більше адміністративної роботи** − для адміністраторів використання віртуальних машин може здатися великою роботою. Їм потрібно підтримувати систему, чекати оновлень і завжди використовувати цю оновлену версію та захищати все на віртуальній машині;

дублікати файлів − якщо ми працюємо на кількох віртуальних машинах, у нас буде багато дублікатів файлів, особливо якщо будь-яка з наших віртуальних машин має подібні версії ОС.

2.1.3 Контейнеризація програмного забезпечення

Контейнеризації - це одна із форма віртуалізації ОС, що пропонує ізоляцію додатків в просторах користувача (контейнерах). Всі контейнери використовують ту саму операційну систему. Завдяки технології контейнеризації можна запускати додаток із потрібними бібліотеками у типовому контейнері, який з'єднується з хостом або іншою зовнішньою компонентною за допомогою простого інтерфейсу.

Усі компоненти, необхідні для роботи програми (код, середовище запуску, системні інструменти, бібліотеки та налаштування), упаковуються в один образ і можуть бути використані повторно в рамках поточного завдання або будь-яких інших. Контейнер незалежний від ресурсів та архітектури хоста. Він створює ізольоване середовище для програми, не використовуючи CPU, RAM або сховища хостової ОС. Усі процеси йдуть усередині.

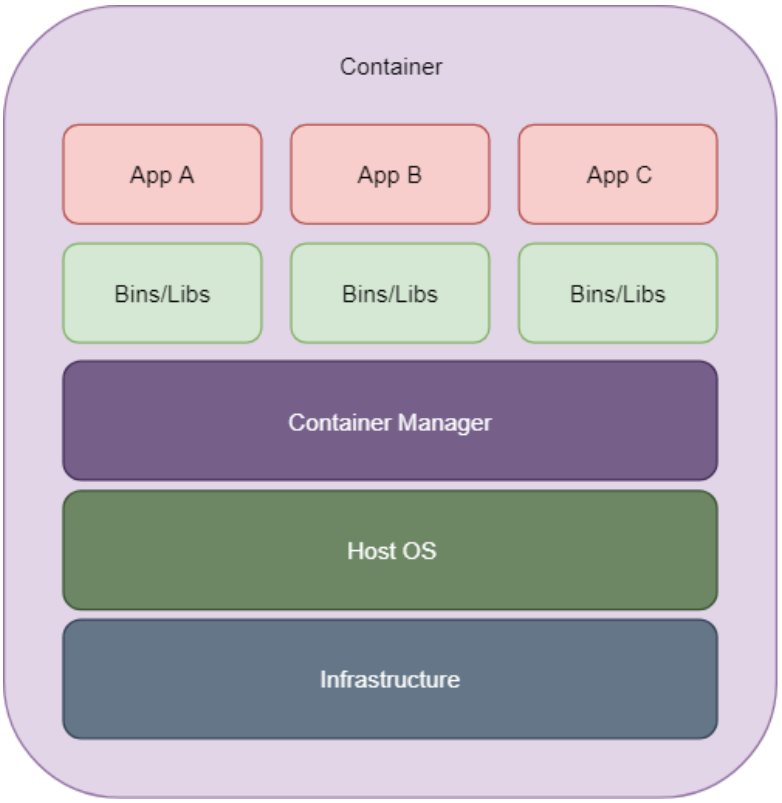

Рисунок 2.2 − Архітектура контейнеризації

2.1.3.1 Опис контейнера програмного забепечення

Контейнер — це ізольоване місце, де програма працює, не впливаючи на решту системи та без впливу системи на програму. Оскільки вони ізольовані, контейнери добре підходять для безпечного запуску програмного забезпечення, наприклад баз даних або веб-додатків, яким потрібен доступ до конфіденційних ресурсів без надання доступу кожному користувачеві системи.

Контейнерне програмне забезпечення, доступне як для додатків Linux, так і для Windows, завжди працюватиме однаково, незалежно від інфраструктури.

Додатки також можуть працювати в будь-якій інфраструктурі та в будь-якій хмарі. Ви можете ізолювати програми та їх базову інфраструктуру від інших програм.

2.1.3.2 Контейнерне оркестрування програмного забезпечення

Контейнерна оркестровка — це автоматизація більшої частини операційних зусиль, необхідних для виконання контейнерних робочих навантажень і служб. Це включає в себе широкий спектр речей, необхідних командам програмного забезпечення для керування життєвим циклом контейнера, включаючи надання, розгортання, масштабування (вгору та вниз), мережу, балансування навантаження тощо.

Оркестровка контейнерів є ключовою для роботи з контейнерами, і це дозволяє організаціям розблокувати всі їхні переваги. Він також пропонує власні переваги для контейнерного середовища, зокрема:

– **Спрощені операції** — це найважливіша перевага оркестровки контейнерів і основна причина її прийняття. Контейнери представляють велику кількість складності, яка може швидко вийти з-під контролю без керування контейнером;

– **Стійкість** — Інструменти оркестровки контейнерів можуть автоматично перезапускати або масштабувати контейнер або кластер, підвищуючи стійкість;

– **Додаткова безпека** — автоматизований підхід оркестровки контейнерів допомагає підтримувати безпеку контейнерних програм, зменшуючи або усуваючи ймовірність людської помилки.

Переваги:

– **працює на одному пристрої, працює всюди** − не потрібно думати про сумісність нашої роботи з операційною системою сервера чи іншими характеристиками. Контейнери можна налаштувати або легко змінити, якщо ви знаєте, як налаштувати свій контейнер;

– **контейнер** − це спрощена версія спрощеного процесу того, що може робити операційна система.

– **легкі** − контейнери дуже легкі, вони мають розмір лише в мегабайтах і займають лише секунди для запуску, і через це ви можливо розмістити в 2–3 рази більше програм на одному сервері з контейнерами, ніж у віртуальній машині.

Недоліки:

безпека − основою контейнерів є образи ОС. Це відкриває багато проблем, які ховаються в образах програм, як-от застарілі, незахищені версії програмного забезпечення та бібліотек, програми з помилками або навіть приховане шкідливе програмне забезпечення. Таким чином, необхідно докласти більше зусиль для додавання безпеки на контейнерних серверах.

2.1.4 Безсерверні обчислення

Безсерверні обчислення (їх також називають serverless computing) — це модель виконання хмарних обчислень, в якій постачальник хмарних послуг виділяє машинні ресурси за запитом. Безсерверні сервери все ще існують, але вони абстраговані від розробки додатків. Хмарний постачальник піклується про конфігурацію фізичних серверів, їх продуктивність, кількість CPU і RAM. Користувач ніяк не взаємодіє з інфраструктурою та не обслуговує її, але при цьому може писати та розгортати код, використовуючи готові обчислювальні ресурси.

Попит і автоматично масштабуються за потреби. Безсерверні пропозиції від постачальників загальнодоступних хмар зазвичай вимірюються на вимогу за допомогою моделі виконання, керованої

подіями. У результаті, коли безсерверна функція простоює, вона нічого не коштує.

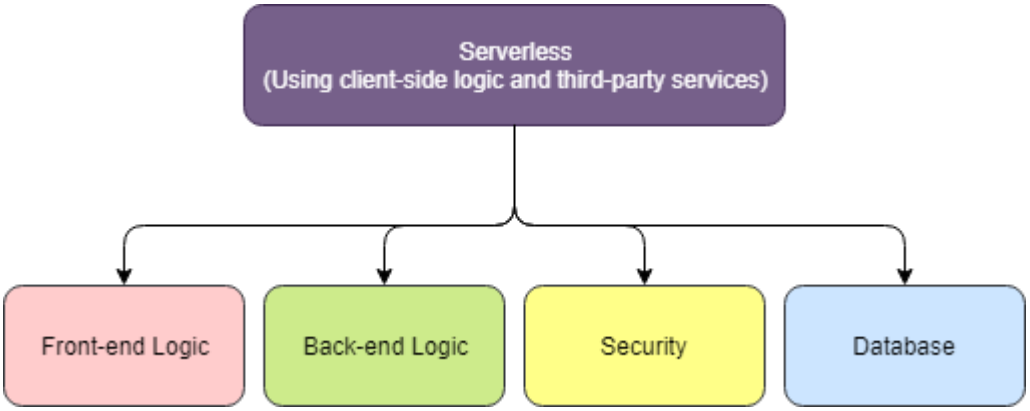

Рисунок 2.3 − Архітектура serverless

Переваги:

– **немає апаратного чи програмного забезпечення** − для керування сервером не потрібні апаратне чи програмне забезпечення. Сам хмарний постачальник керує безсерверним обчисленням аж до основного апаратного забезпечення, ОС до рівня платформи веб-додатків.

– **вартість** − у безсерверному режимі ми зменшуємо витрати. Вартість, яку ви заплатите, — це лише тривалість виконання вашої функції та кількість ресурсів, які ваша програма спожила протягом періоду виконання. Тривалість обчислюється від моменту початку виконання вашої програми до її повернення або завершення. Це відповідає концепції оплати, коли ви отримуєте, коли послуга використовується, а саме в той час, коли ви платите.

Недоліки:

– **тривалість виконання** − безсерверний режим призначений для роботи протягом короткого періоду часу, що не дуже підходить для великих програм

– **складність** − чим меншими ви робите свої програми, тим складнішими вони будуть.

2.2 АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ КОНТЕЙНЕРИЗАЦІЇ

Вперше загадування про віртуальні машини, яке датується 1960-ми роками, дозволяла кільком користувачам одночасно отримувати доступ до комп'ютера з усіма ресурсами через окрему програму для кожного. Наступні десятиліття ознаменувалися широким використанням і розвитком віртуальних машин та контейнерів.

Потім з'явився chroot. Еволюція контейнерів стрибнула вперед із розробкою chroot у 1979 році у версії 7 Unix. Chroot поклав початок ізоляції процесів у стилі контейнера, обмеживши доступ програми до файлу до певного каталогу – кореневого – та його дочірніх елементів. Ключовою перевагою розділення chroot було покращення безпеки системи, так що ізольоване середовище не могло скомпрометувати зовнішні системи в разі використання внутрішньої вразливості.

2000-ті роки були сповнені розвитку та вдосконалення контейнерних технологій. У 2003 році Google представив Borg, систему керування кластерами контейнерів організації. Вона спиралася на механізми ізоляції, які вже були в Linux. У ті перші дні еволюції контейнерів безпека не була особливою проблемою. Будь-хто міг бачити, що відбувається всередині машини, що увімкнуло систему обліку, хто використовує найбільше пам'яті, і як покращити роботу системи.

Це призвело до розробки контейнерів процесів, які стали контрольними групами (cgroups) ще в 2004 році. Cgroups відзначили зв'язки між процесами та стримали доступ користувачів до певних дій і обсягів пам'яті. Концепція cgroup була включена в ядро Linux у січні 2008 року, після чого з'явилася контейнерна технологія Linux LXC. Простори імен були розроблені незабаром після цього, щоб забезпечити основу для безпеки контейнерної мережі – щоб приховати діяльність користувача або групи від інших. Завдяки надійності та стабільності LXC багато інших

технологій, побудованих на LXC, з'явилася першою з яких стала Warden у 2011 році та, що важливіше, Docker у 2013 році.

Docker простим у використанні графічним інтерфейсом і можливістю пакувати, надавати та запускати технологію контейнерів. Оскільки Docker дає можливість працювати кільком програмам із різними вимогами до ОС на одному ядрі ОС у контейнерах, IT-адміністратори та організації побачили можливість для спрощення та економії ресурсів. Протягом місяця після першого тестового випуску Docker став майданчиком для 10 000 розробників. До моменту випуску Docker 1.0 у 2014 році програмне забезпечення було завантажено 2,75 мільйона разів. А протягом року після цього — понад 100 мільйонів разів.

Контейнерна технологія набула значного розвитку в 2017 році. Такі компанії, як Pivotal, Rancher, AWS і навіть Docker, перейшли на підтримку контейнерів Kubernetes із відкритим вихідним кодом і інструмента оркестровки, закріпивши його позицію як стандартної технології оркестровки контейнерів. У квітні 2017 року Microsoft дозволила організаціям запускати контейнери Linux на Windows Server.

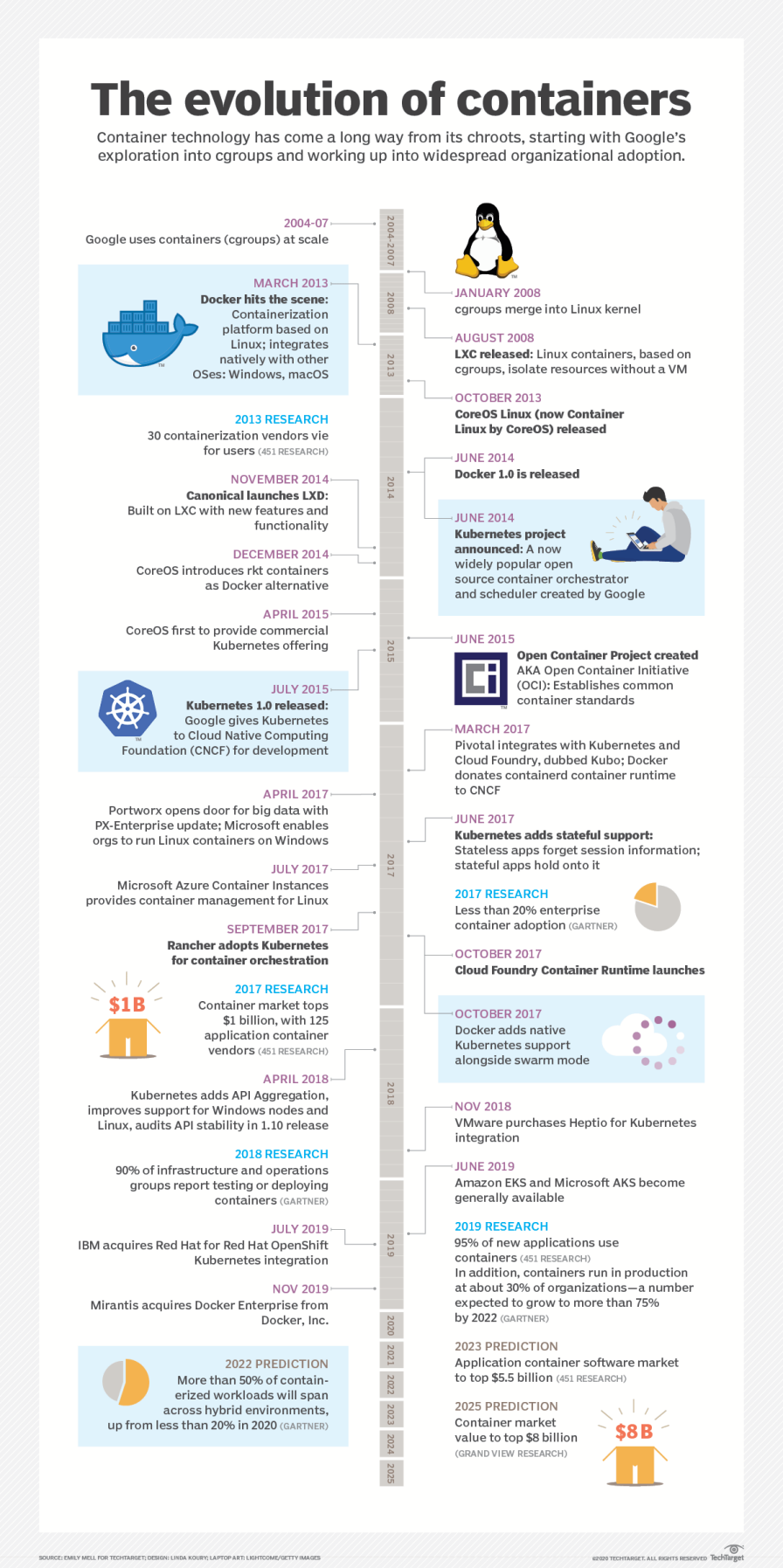

Рисунок 2.4 – Розвиток технології контейнерів

2.2.1 Аналіз екосистеми контейнерів

Docker та Kubernetes – дві провідні платформи в екосистемі контейнерів.

Для забезпечення сумісності співтовариство узгодило декілька стандартів.

Двома найважливішими стандартами є:

– **Container Runtime Interface** (CRI) визначає API між Kubernetes та Container Runtime (середовищем виконання контейнерів);

– **Open Container Initiative** (OCI) визначає стандарт образів та контейнерів.

Нижче приведено приклад, як Docker, Kubernetes, OCI, CRI, containerd і runc вписуються в цю екосистему:

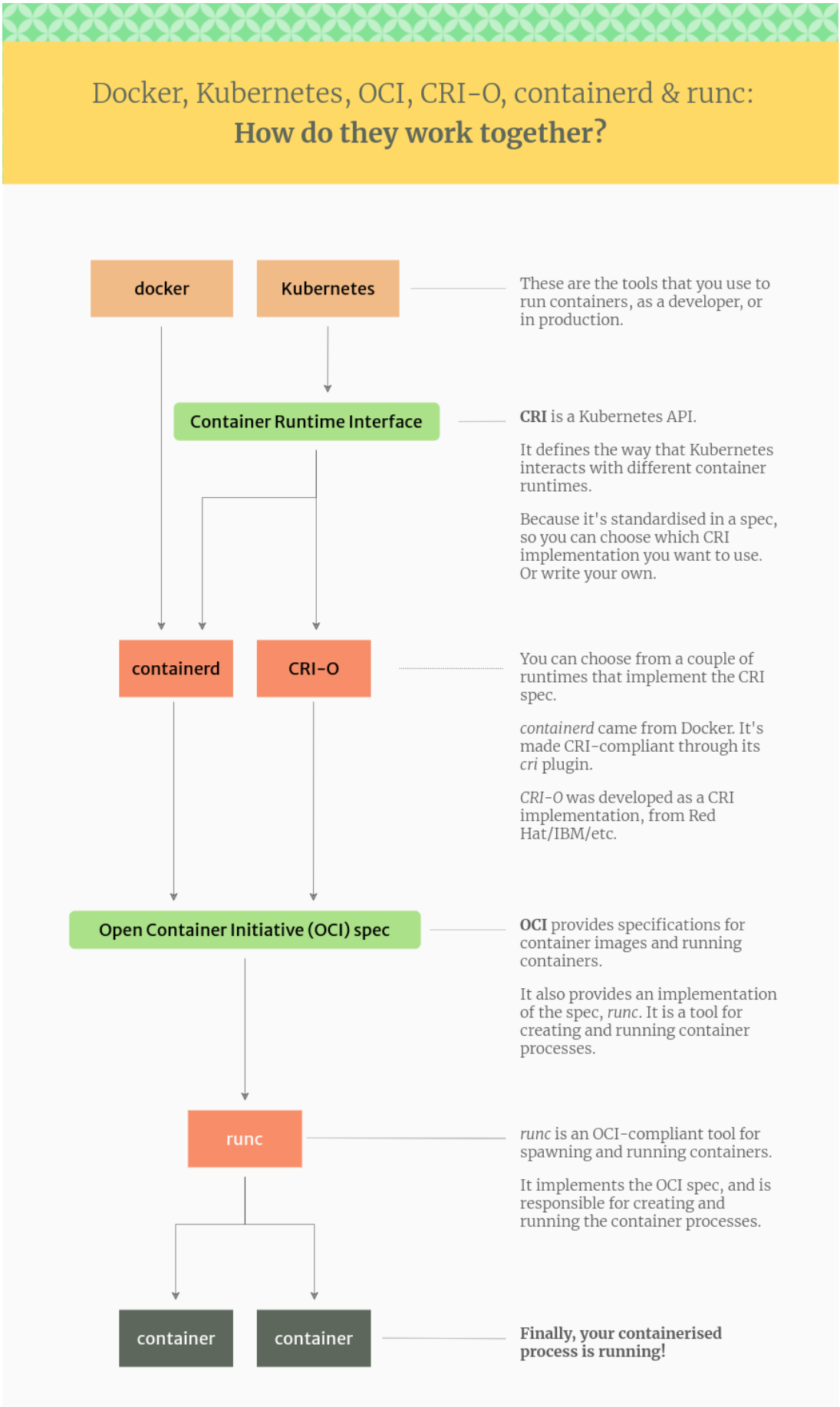

Рисунок 2.5 – Екосистема контейнерів

2.2.2 Аналіз рішення Docker

Docker — це платформа контейнеризації з відкритим вихідним кодом, за допомогою якої можна автоматизувати створення програм, їх доставку та керування. Платформа дозволяє швидше тестувати та викладати програми, запускати на одній машині необхідну кількість контейнерів.

Завдяки Docker можна легко запускати контейнер у хмарній інфраструктурі та на будь-якому локальному пристрої. Можна створити базові шаблони контейнерів і повторно використовувати нескінченну кількість разів. Легка переносимість та простота розгортання – важливі переваги цієї технології. Docker дозволяє використовувати будь-які мови програмування та стек технологій на сервері

Контейнери ізольовані та можуть мати спільне програмне забезпечення, бібліотеки та файли конфігурації. Вони можуть спілкуватися за допомогою внутрішньої мережи, яка має більше 5 разних варіантів нашалтування. Оскільки всі контейнери спільно використовують служби одного ядра операційної системи, використовують менше ресурсів, ніж віртуальні машини, тому запуск великої кількості контейнерів на одній машині не несе ризиків.

2.2.2.1 Основні компоненти Docker

– **Dockerfile** – текстовий файл із послідовно розташованими інструкціями для створення образу Docker. Файл створюється за принципом один рядок - одна команда.

– **Daemon** – фонова служба на хості, яка відповідає за створення, запуск та знищення контейнерів.

– **Image** – незмінний файл (образ), з якого можна необмежену кількість разів розгорнути контейнер.

– **Container** – запущена програма, яку розгорнули з образу.

– **Registry** – служба Docker, що виконує функції репозиторію (сховища). Дозволяє стежити за версіями образів, створювати приватні репозиторії.

– **Docker Hub** – популярний публічний репозиторій, який використовується за замовчуванням у Docker. Забезпечує інтеграцію з системою VCS

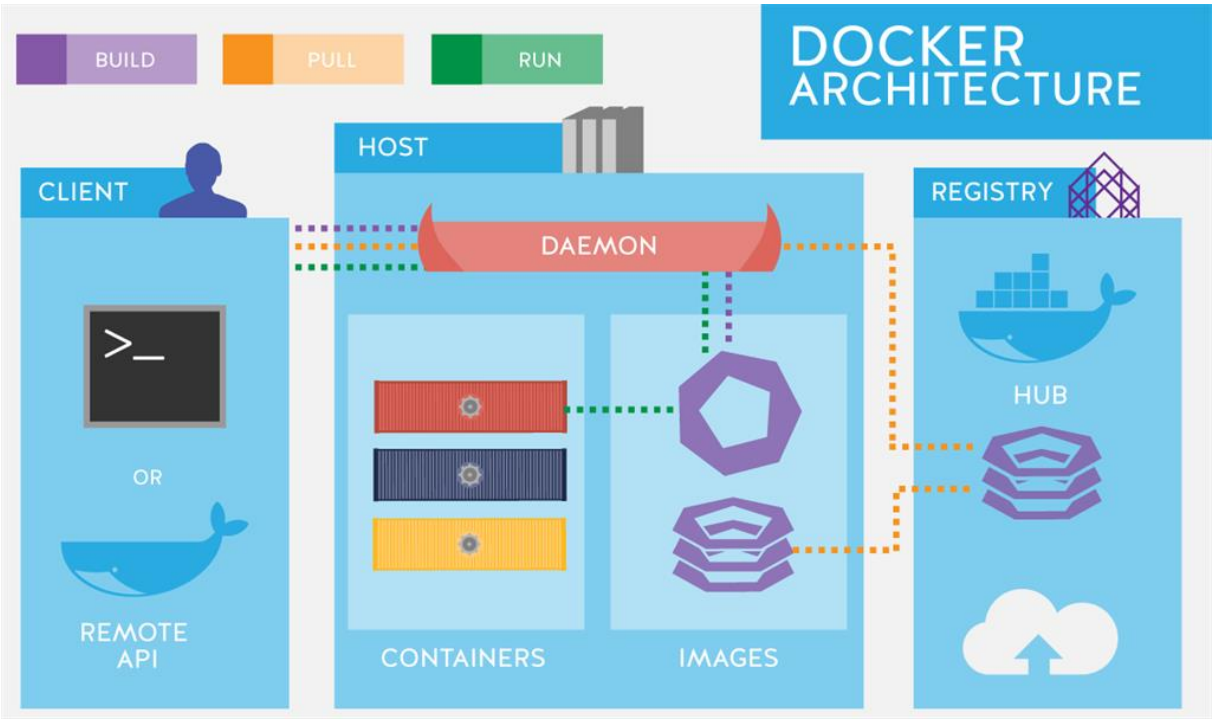

Рисунок 2.6 − Архітектура технології Docker

2.2.2.2 Як влаштований образ Docker

Базовий образ є головним елементом контейнеризації в Docker. У ньому містяться процеси та залежності, необхідні для нормальної роботи програми.

На базовий образ Docker один за одним накладаються доступні тільки для читання шари, які утворюються після будь-яких змін в образі. Кожен новий шар – це актуальна версія образу. Виходить, що фінальний образ - це поєднання всіх верств в один. Кожен шар образу зберігається, щоб у разі потреби швидко повернути попередній. Таке рішення заощаджує простір диска та скорочує час збирання контейнера.

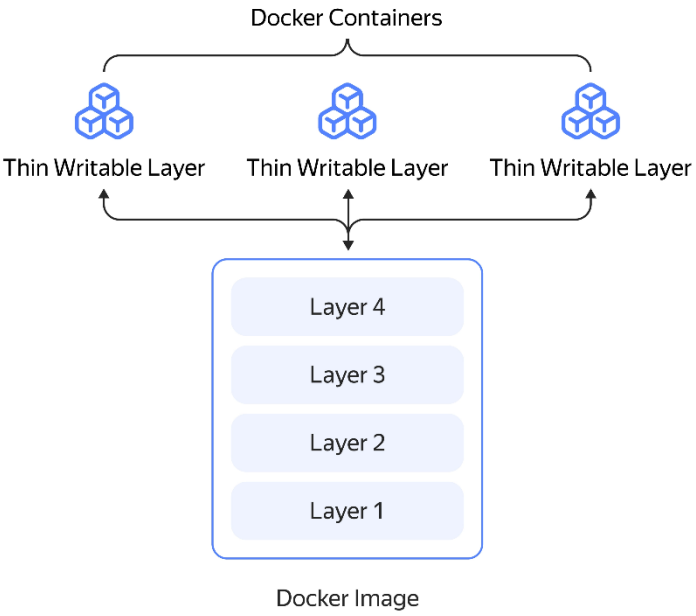

Рисунок 2.7 − Шари Docker

2.2.3 Аналіз рішення Open Container Initiative

Open Container Initiative (OCI) — це спільний проект, який організовує Linux Foundation і спрямований на встановлення загальних стандартів для контейнерів. Ініціатива, яка має легку відкриту структуру управління, була вперше оприлюднена як Open Container Project на DockerCon 22 червня 2015 року, а пізніше була перейменована на Open Container Initiative.

Open Container Initiative підтримує довгий список відомих компаній, однак проект залишатиметься незалежним від будь-якої конкретної комерційної організації. Серед засновників Amazon Web Services, Docker, CoreOS, Microsoft, VMware, EMC, Nutanix, Red Hat, IBM, Goldman Sachs і Google. Docker відіграв ключову роль у заснуванні ініціативи, пожертвувавши чернетки специфікацій і більшу частину свого існуючого коду для формату зображення та середовища виконання контейнера. Формування OCI було викликано стрімким зростанням інтересу до віртуалізації на основі контейнерів, зокрема як способу підвищення мобільності додатків у багатьох середовищах.

Основними цілями проекту є забезпечення стандартів для контейнерів і майбутніх контейнерних платформ, які зберігають гнучкий і відкритий характер контейнерів. Зокрема, OCI стверджує, що контейнери не повинні бути прив’язані до певного клієнта чи стеку оркестровки, не повинні бути тісно пов’язані з будь-яким конкретним постачальником і бути переносимими на широкий спектр операційних систем, обладнання та архітектур.

2.2.4 Аналіз рішення Container Runtime Interface

Інтерфейс середовища виконання контейнера (CRI) — це інтерфейс плагіна, який дозволяє kubelet — агенту, який працює на кожному вузлі в кластері Kubernetes — використовувати більше ніж один тип середовища виконання контейнера. На найнижчих рівнях вузла Kubernetes знаходиться програмне забезпечення, яке, серед іншого, запускає та зупиняє контейнери. Ми називаємо це «контейнерним середовищем». Найвідомішим середовищем виконання контейнерів є Docker, але він не єдиний у цьому просторі.

Kubelet спілкується із середовищем виконання контейнера (або прокладкою CRI для середовища виконання) через сокети Unix за допомогою фреймворку gRPC, де kubelet діє як клієнт, а прокладка CRI — як сервер.

Container Runtime Interface (CRI) — інтерфейс плагіна, який дозволяє kubelet використовувати різноманітні середовища виконання контейнерів без необхідності перекомпілювати. CRI складається з protocol buffers та gRPC API, а також бібліотек із додатковими специфікаціями та інструментами, які активно розробляються. CRI випускається як альфа-версія в Kubernetes 1.5.

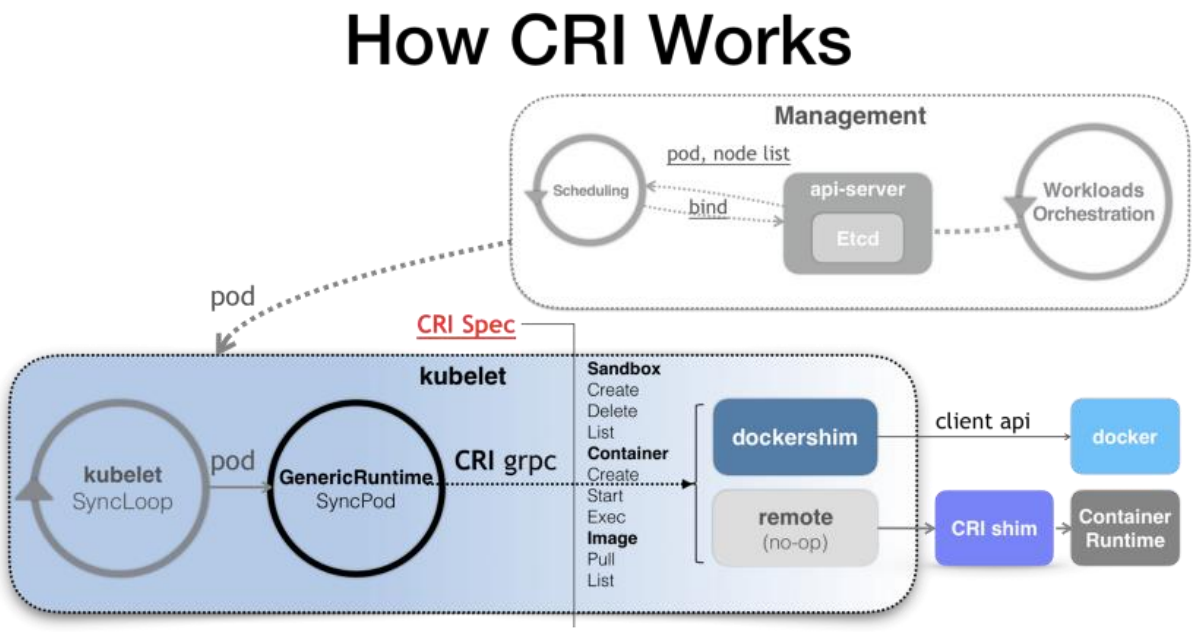

Рисунок 2.8 − Архітектура технології CRI

2.2.5 Аналіз рішення rtk

Rkt - це механізм контейнерів додатків, розроблений для сучасних виробничих хмарних середовищ. Він відрізняється вбудованим підходом pod, підключеним середовищем виконання та чітко визначеною площею поверхні, що робить його ідеальним для інтеграції з іншими системами.

Основним виконавчим модулем rkt є pod, набір з одного або декількох додатків, що виконуються в загальному контексті (модулі RKT є синонімами концепції в системі оркестровки Kubernetes). rkt дозволяє користувачам застосовувати різні конфігурації (наприклад, параметри ізоляції) як на рівні модуля, так і на більш детальному рівні для кожної програми. Архітектура rkt означає, що кожен модуль виконується безпосередньо в класичній моделі процесів Unix (тобто відсутній центральний демон), в автономному, ізольованому середовищі. rkt реалізує сучасний, відкритий, стандартний формат контейнера, специфікацію App

Container (appc), але також може виконувати інші образи контейнерів, наприклад, створені за допомогою Docker.

Переваги:

– **Композиційний** − дотримуючись філософії інструментів unix, rkt - це єдиний двійковий файл, який інтегрується з системами ініціалізації, сценаріями та складними конвеєрами devops. Контейнери займають своє правильне місце в ієрархії PID і ними можна керувати за допомогою стандартних утиліт.

– **Настроювана ізоляція** − використовуйте контейнери як стандартний, більш безпечний об'єкт розгортання та виберіть відповідний рівень ізоляції, використовуючи архітектуру середовища виконання RKT, відому як stages.

– **Вбудовані модулі** − Атомною одиницею в rkt є модуль, група пов'язаних контейнерів, які спільно використовують ресурси. Це дозволяє легко поєднувати пов'язані компоненти та безпосередньо співвідносити їх з концепціями управління кластером..

2.2.6 Аналіз рішення Linux LXC

Linux LXC - це служба віртуалізації, яка дозволяє розкручувати кластери ізольованих середовищ Linux. Він надає низку переваг перед монолітними віртуальними машинами, зменшуючи навантаження ресурсів на головну машину. Це робить його ідеальним для створення, тестування та розгортання хмарного програмного забезпечення. LXC, на відміну від інших інструментів віртуалізації на рівні ОС, пропонує набагато краще середовище Linux .

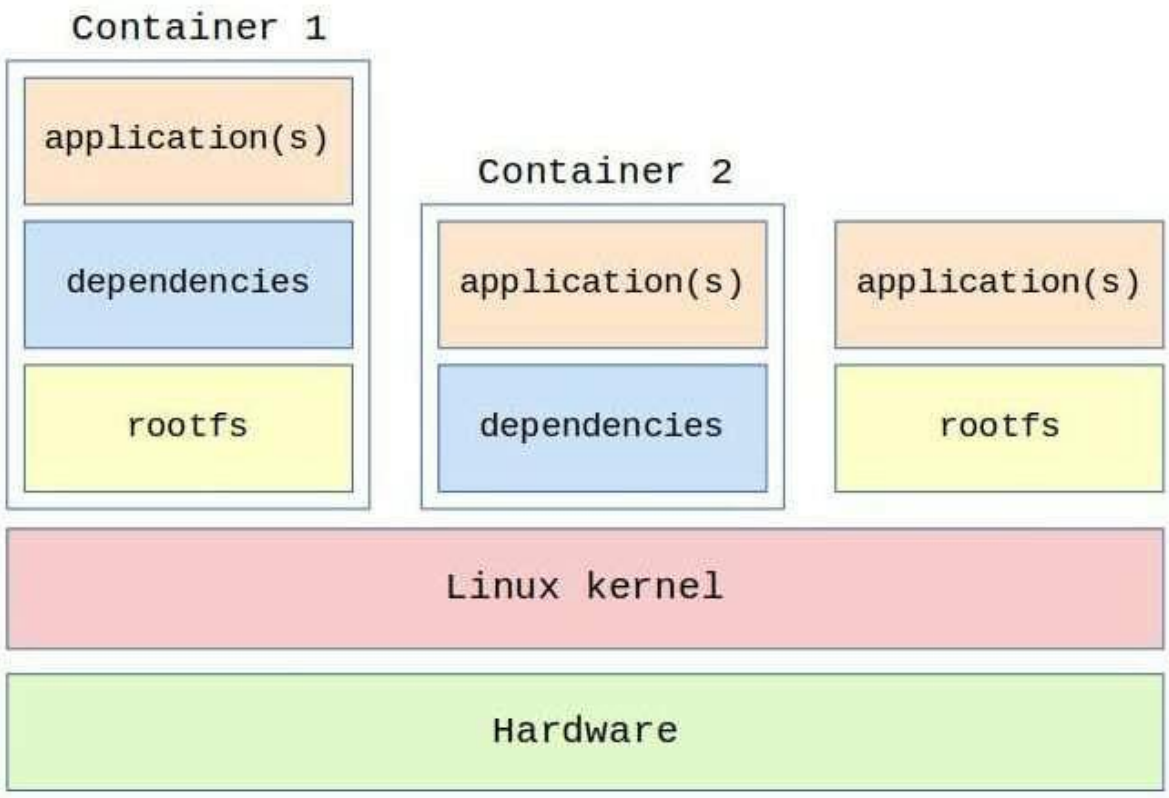

Рисунок 2.9 − Архітектура технології Linux LXC

LXC не використовує жодних химерних механізмів керування ресурсами, таких як гіпервізори. Натомість він використовує функції обмеження хостів, надані безпосередньо ядром Linux. Основними компонентами, на які він спирається, є простори імен і контрольні групи . Вперше вони були додані до ядра з версії 2.6.24. Основним принципом проектування груп с, також відомих як «Групи контролю» , є забезпечення обмеження ресурсів, визначення пріоритетів, обліку та контролю. Простори імен відповідають за приховування простору процесу та інформації про ресурси одного контейнера від інших.

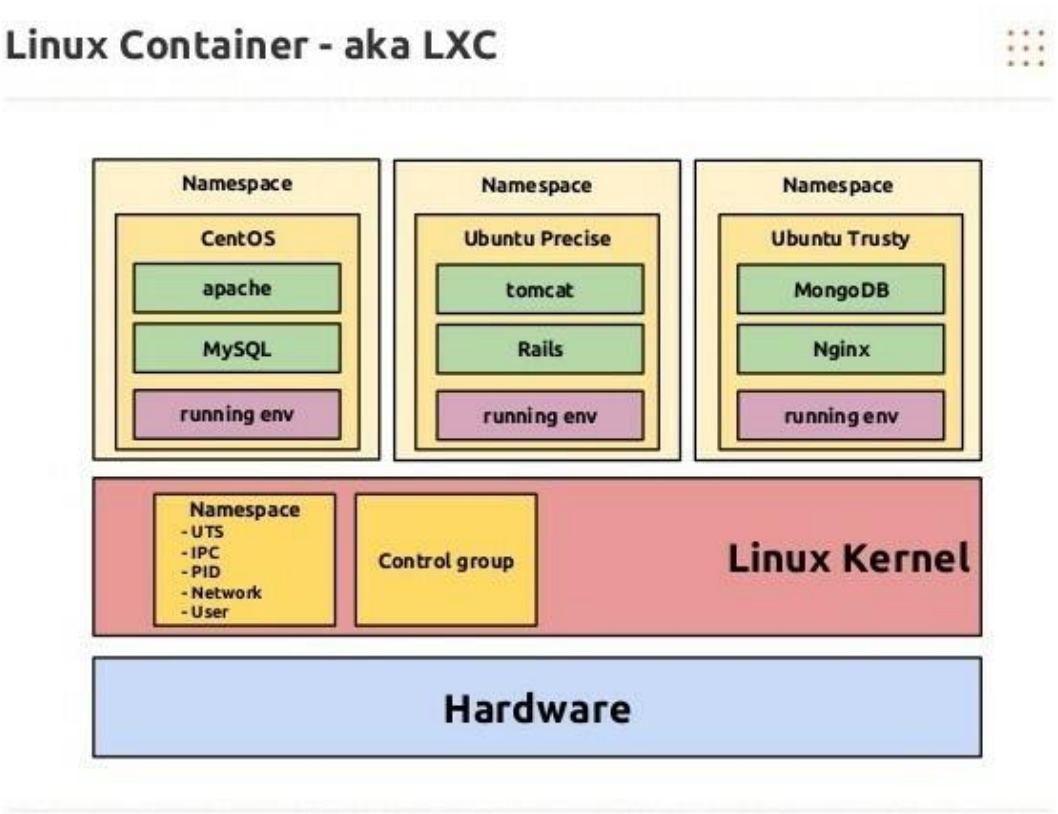

Рисунок 2.10 − Групи контролю Linux LXC

Крім того, LXC також має вбудовану підтримку різних політик захисту Linux, таких як профілі Apparmor і SELinux, а також Chroots.Він без зусиль працює майже на будь-якій архітектурі та в хмарі. Крім того, ви можете запускати будь-який дистрибутив Linux незалежно від хоста. Отже, припустімо, що на вашій хост-машині працює Ubuntu. Ви можете легко запустити Red Hat або CentOS на цій машині за допомогою контейнерів LXC.

LXC, на відміну від деяких інших служб контейнеризації, не може працювати в Mac OS або Windows. Це тому, що контейнери LXC покладаються безпосередньо на ядро хоста. Отже, якщо ви хочете запускати програми, які потребують однієї з цих систем, вам слід розглянути іншу платформу, наприклад Docker . Загалом, LXC найкраще підходить для людей, яким потрібно запускати ізольоване середовище Linux з мінімальними ресурсами.

2.2.7 Аналіз рішення CRI-O

CRI-O — це контейнерний механізм із відкритим кодом, керований спільнотою. Його головна мета — замінити службу Docker як механізм контейнера для реалізацій Kubernetes, таких як OpenShift Container Platform.

Контейнерний механізм CRI-O забезпечує стабільну, безпечнішу та продуктивнішу платформу для запуску сумісних середовищ Open Container Initiative (OCI). Можливо використовувати механізм контейнерів CRI-O для запуску контейнерів і модулів, залучаючи OCI-сумісні середовища виконання, такі як runc, середовище виконання OCI за замовчуванням або контейнери Kata. Мета CRI-O — бути механізмом контейнерів, який реалізує інтерфейс виконання контейнерів Kubernetes (CRI) для контейнерної платформи OpenShift і Kubernetes, замінивши службу Docker.

CRI-O пропонує оптимізовану систему контейнерів, а інші функції контейнерів реалізовано як окремий набір інноваційних незалежних команд. Цей підхід дозволяє функціям керування контейнерами розвиватися у власному темпі, не перешкоджаючи головній меті CRI-O — бути рушієм контейнерів для установок на базі Kubernetes.

Стабільність CRI-O пояснюється тим фактом, що його розроблено, протестовано та випущено разом із основними та другорядними випусками Kubernetes і що він відповідає стандартам OCI. Наприклад, CRI-O 1.11 узгоджується з Kubernetes 1.11. Сфера застосування CRI-O пов’язана з інтерфейсом виконання контейнерів (CRI). CRI витягнув і стандартизував саме те, що потрібно службі Kubernetes (kubelet) зі свого механізму контейнерів. Команда CRI зробила це, щоб допомогти стабілізувати вимоги до двигуна контейнерів Kubernetes, коли почали розробляти кілька механізмів контейнерів.

Немає необхідності в прямому контакті з CRI-O через командний рядок. Однак, щоб надати повний доступ до CRI-O для тестування та моніторингу, а також надати функції, які ви очікуєте від Docker, яких CRI-

O не пропонує, доступний набір пов’язаних із контейнером інструментів командного рядка

CRI-O не підтримується як окрема система контейнерів. Ви повинні використовувати CRI-O як механізм контейнера для встановлення Kubernetes, наприклад OpenShift Container Platform

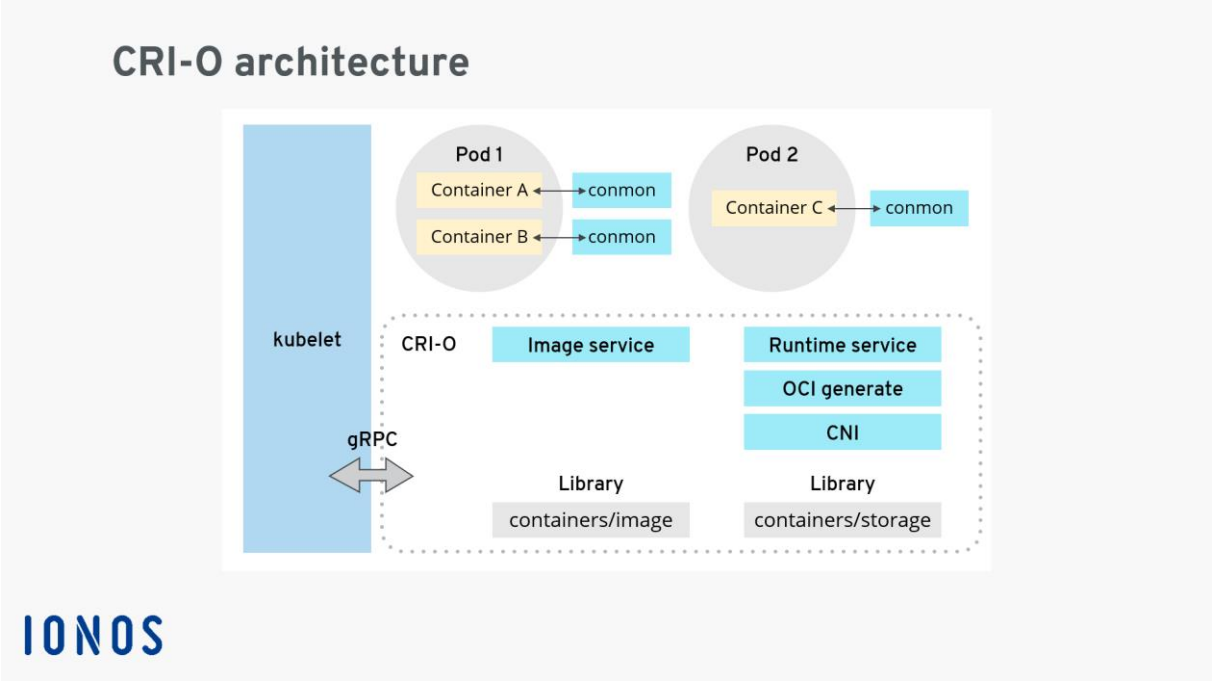

Рисунок 2.11 − Архітектура технології CRI-O

2.2.8 Висновки та обґрунтування вибору технології

У нашому випадку комп’ютерна система використовує On Premise, так як розташована на шкільному сервері. Цей підхід вже застарілий та потребує оновлення.

Виходячи с того, що комп’ютерна система потребує постійного та стабільного існування варіант з Serverless не підходить. Отже залишається два варіанти. Вимога до системи, щоб вона була легка та мала можливість запускатися на будь-якому пристрої у разі необхідності.

Таким чином, найкращим вибором є контейнеризація програми.

2.3 ПОРІВНЯННЯ ОРКЕСТРАТОРІВ

Як ми побачили контейнери — легкі та ефемерні за своєю природою, та запуск багатьох в одночас буде дуже складним завданням. Наприклад, коли використовується мікросервісна архітектура, де кожен мікросервіс запущений у окремому контейнері, а такої до кожного мікросервісу підключена власна база даних. Це може буди сотні сервісів запущених водночас.

Стає надзвичайно важко керувати життєвим циклом контейнерів та керування ними, коли кількість динамічно зростає.

Оркестровка контейнерів вирішує проблему шляхом автоматизації планування, розгортання, масштабованості, балансування навантаження, доступності та мережі контейнерів.

Оркестровка контейнерів — це автоматизація та управління життєвим циклом контейнерів і сервісів.

2.3.1 Інструменти оркестровки

2.3.2 Оркестровка за допомогою технології Kubernetes

Kubernetes — це платформа з відкритим вихідним кодом, спочатку розроблена компанією Google, а тепер підтримується Cloud Native Computing Foundation. Kubernetes підтримує як декларативну конфігурацію, так і автоматизацію. Це може допомогти автоматизувати розгортання, масштабування та керування контейнерним навантаженням і службами.

Kubernetes API допомагає налагодити зв’язок між користувачами, компонентами кластера та зовнішніми сторонніми компонентами. Площина керування Kubernetes і вузли працюють на групі вузлів, які разом утворюють кластер. Робоче навантаження програми складається з одного або кількох модулів, які працюють на робочих вузлах. Площина керування керує модулями та робочими вузлами.

Наприклад такі компанії, як Babylon, Booking.com, AppDirect, широко використовують Kubernetes.

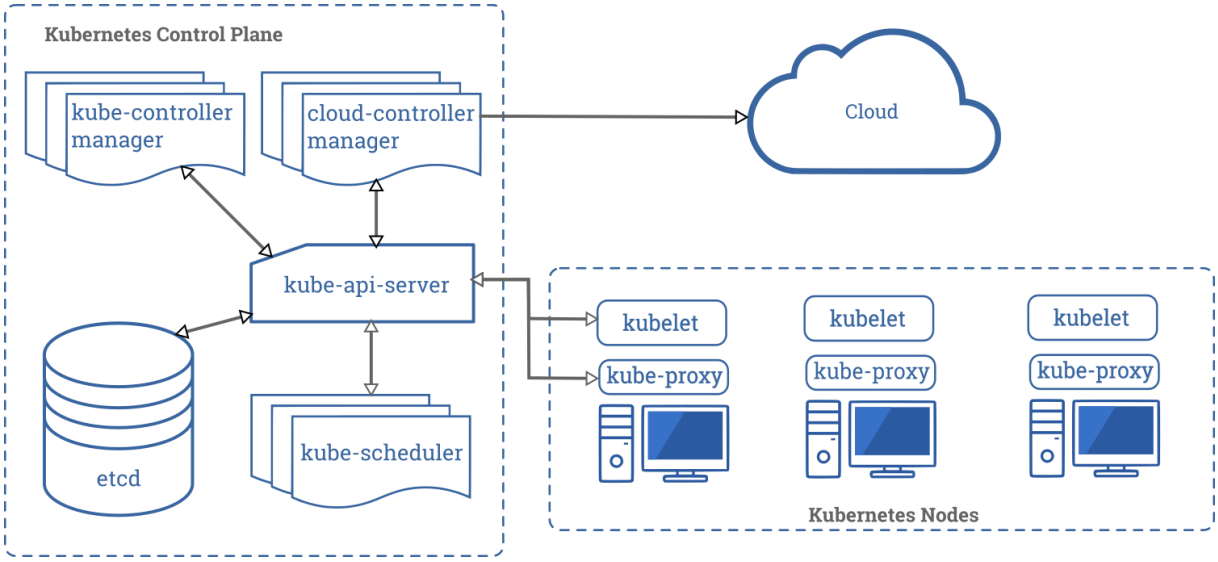

Рисунок 2.12 − Kubernetes architecture

2.3.3 Оркестровка за допомогою технології Nomad

Nomad — це простий, гнучкий і легкий у використанні оркестровник робочого навантаження для масштабного розгортання контейнерів і неконтейнерних програм і керування ними в локальних і хмарних середовищах. Nomad працює як єдиний двійковий файл із невеликим обсягом ресурсів (35 МБ) і підтримується в macOS, Windows і Linux.

Розробники використовують декларативну інфраструктуру як код (IaC) для розгортання своїх програм і визначають спосіб розгортання програми. Nomad автоматично відновлює програми після збоїв.

Nomad Orchestrate будь-які програми (не лише контейнери). Він забезпечує першокласну підтримку Docker, Windows, Java, віртуальних машин тощо.

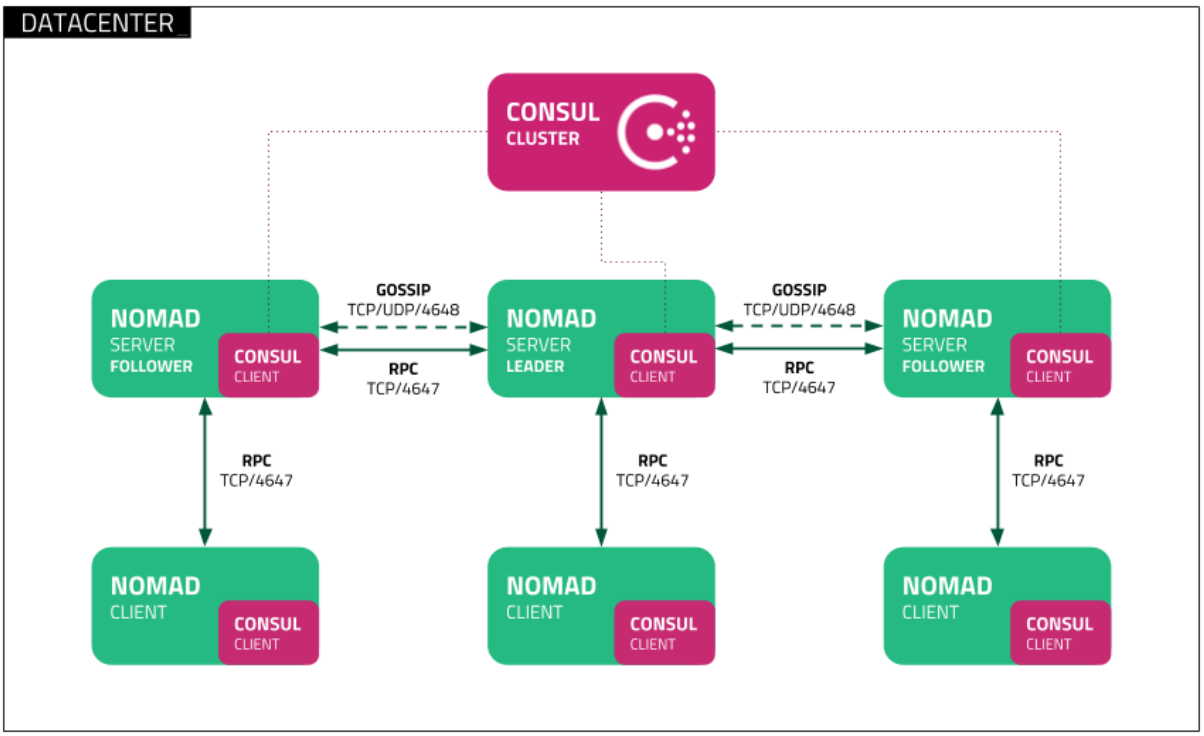

Рисунок 2.13 − Kubernetes architecture

2.3.4 Оркестровка за допомогою технології Rancher

Rancher — це платформа з відкритим вихідним кодом, яка використовує оркестровку контейнерів, відому як велика рогата худоба. Це дозволяє вам використовувати такі сервіси оркестровки, як Kubernetes, Swarm, Mesos. Rancher надає програмне забезпечення, необхідне для керування контейнерами, щоб організаціям не потрібно було створювати платформи обслуговування контейнерів з нуля, використовуючи окремий набір технологій з відкритим кодом.

Rancher 2.x дозволяє керувати кластерами Kubernetes, які працюють на постачальниках, указаних клієнтом.

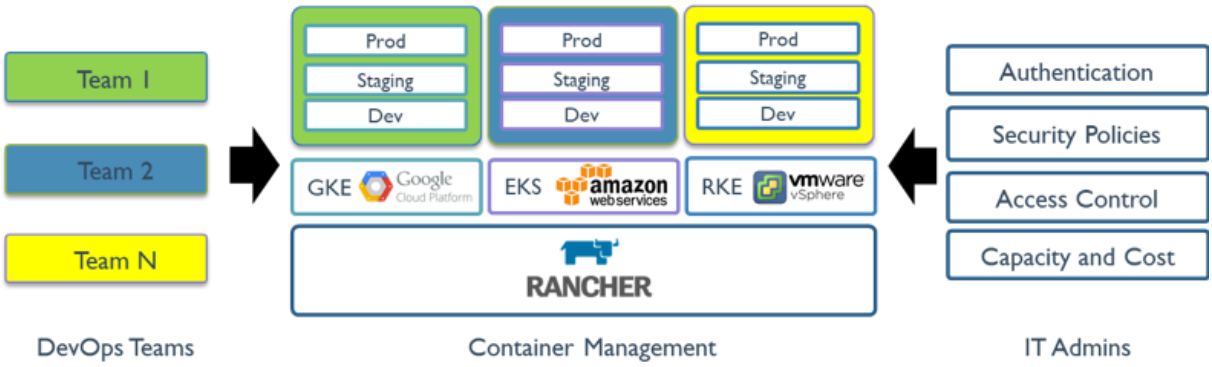

Рисунок 2.14 − Kubernetes architecture

2.3.5 Аналітичне порівняння хмарних рішень

Розглянемо розгортання Docker контейнера у популярних хмарних рішень на ринку такі як: Amazon Web Services, Microsoft Azure та Google Cloud Platform.

2.3.6 Аналіз платформи Google Cloud Platform

Cloud Run — це керована обчислювальна платформа, яка дозволяє запускати контейнери безпосередньо на масштабованій інфраструктурі Google.

Ви можете розгорнути код, написаний будь-якою мовою програмування, у Cloud Run, якщо зможете створити з нього образ контейнера.

Одним словом, Cloud Run дозволяє розробникам витрачати свій час на написання свого коду та дуже мало часу на роботу, налаштування та масштабування служби Cloud Run. Вам не потрібно створювати кластер або керувати інфраструктурою, щоб працювати продуктивно з Cloud Run.

2.3.6.1 Приклад запуску додатка у Cloud Run

Перейдіть до Google Cloud Platform Console і виберіть «Створити службу».

Виберіть регіон, у якому ви хочете його запустити, і дайте йому назву.

Ви також можете захистити цей контейнер за допомогою Cloud IAM. Це особливо корисно, якщо ви розгортаєте внутрішні служби та хочете захистити їх, щоб лише авторизовані користувачі або облікові записи служб мали до них доступ. Якщо ви виберете це, вам потрібно буде надати дозвіл IAM для вказаних облікових записів користувачів, щоб вони мали доступ до цієї служби.

На наступному кроці ви налаштуєте першу версію служби контейнера, використовуючи URL-адресу для зображення. Це може бути з реєстру Docker або ви можете використовувати Google Container Registry.

У розділі «Додаткові параметри» ви можете налаштувати, який порт надсилати до контейнера, а також будь-які потрібні команди точки входу та аргументи.

У розділі «Ємність» ви можете змінити ліміт одночасних запитів, час очікування запиту, кількість ядер ЦП і пам’ять, виділену кожному контейнеру. Оптимально цього має бути достатньо для запуску одного контейнера вашої програми — якщо вам потрібна додаткова ємність, автомасштабування збільшить її.

Автоматичне масштабування відбуватиметься автоматично, але ви можете зменшити максимальну кількість екземплярів, якщо вас турбує вартість.

Після натискання «Створити» служба запуститься. Ви зможете переглянути його деталі на консолі Cloud Run. URL-адресу для підключення до контейнера можна переглянути на цій панелі. Звичайно, ви можете використовувати спеціальний домен для зіставлення з контейнером, але використання цього з CNAME також буде працювати.

Якщо ви перейдете за цією URL-адресою, ви побачите, що ваш контейнерний сервіс працює.

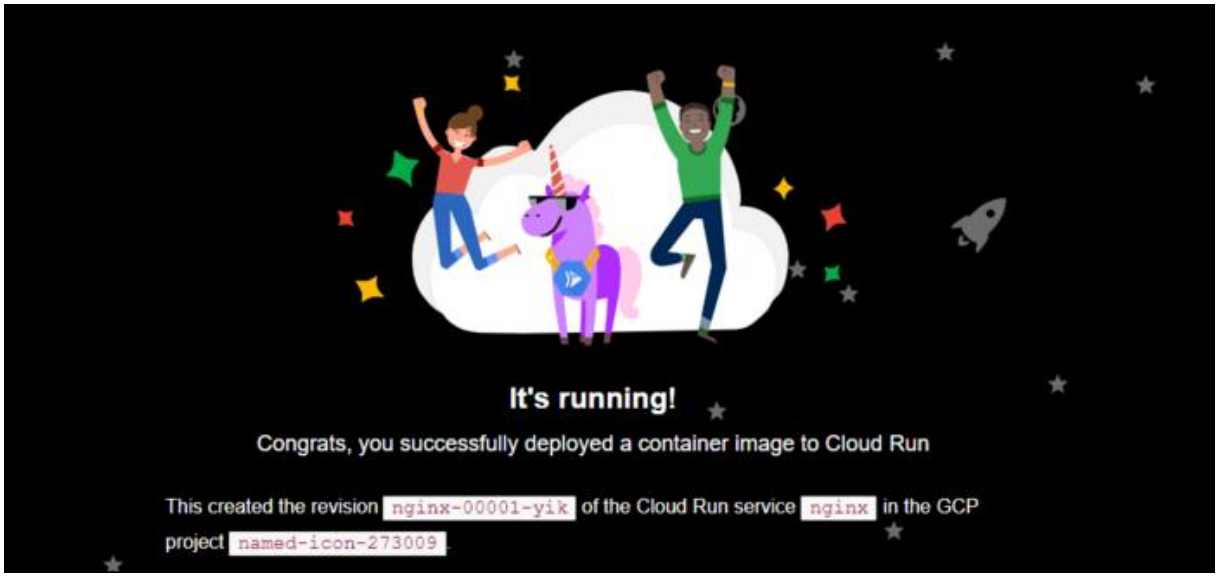

Рисунок 2.15 − Робота контейнера у Google Cloud Platform Console

2.3.7 Аналіз платформи Microsoft Azure

Docker Azure дозволяє розробникам використовувати команди Docker для запуску програм в контейнерах Azure (ACI) під час створення хмарних програм. Azure Container Instances (ACI) забезпечує тісну інтеграцію між Docker Desktop і Microsoft Azure, дозволяючи розробникам швидко запускати програми за допомогою розширення Docker CLI або VS Code, щоб плавно переходити від локальної розробки до хмарного розгортання.д

Крім того, інтеграція між технологіями розробників Docker і Microsoft дозволяє розробникам використовувати Docker CLI для:

– Налаштування контексту ACI в одній команді Docker, що дозволяє переключатися з локального контексту на хмарний контекст і швидко й легко запускати програми

– Спрощує розробку одноконтейнерних і багатоконтейнерних додатків за допомогою специфікації Compose, дозволяючи розробнику

безперешкодно викликати повністю сумісні з Docker команди вперше всередині служби хмарного контейнера

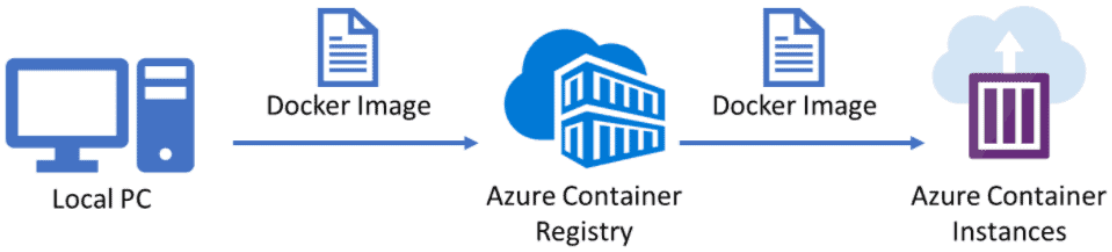

Рисунок 2.16 − Архітектура Azure Container Instances (ACI)

2.3.7.1 Приклад запуску додатка у Azure Container Instances

Увійдіть на портал Azure та передіть до Container Instances. Розгорнемо публічних образ Microsoft aci-helloworld.

Тепер у розділі «Основи» заповнимо усі обов'язкові поля (взяти посилання на картинку нижче), також переконайтеся, що вибрано зображення Quickstart у джерелі зображення та microsoft/aci-helloworld (Linux) у зображенні, а потім натисніть «Далі: мережа».

Тут, у розділі «Мережа», дайте назву мітці імені DNS. Ім'я має бути унікальним у регіоні Azure, де ви створюєте екземпляр контейнера. Тепер переконайтеся, що ви ввімкнули Networking Type into Public, оскільки ваш контейнер буде загальнодоступним за посиланням <dns-name-label>.<region>.azurecontainer.io. потім натисніть «Переглянути+Створити».

Тепер можна побачити нове вікно, яке показує, що перевірка пройдена, та кликнемо «Створити».

Дочекаймося завершення розгортання та створімо усі необхідні ресурси. Після завершення розгортання ми бачимо завершене вікно, у якому потрібно натиснути Перейти до ресурсу

Отже, ми завершили розгортання ізольованого контейнера і налаштування.

Давайте перевіримо доступність програми-контейнера Docker за допомогою FQDN

Скопіюйте URL-адресу повного доменного імені, вставимо її у свій браузер і натиснемо на пошук, тепер ви побачите повідомлення Ласкаво просимо до екземпляра контейнера Azure у вікні браузера

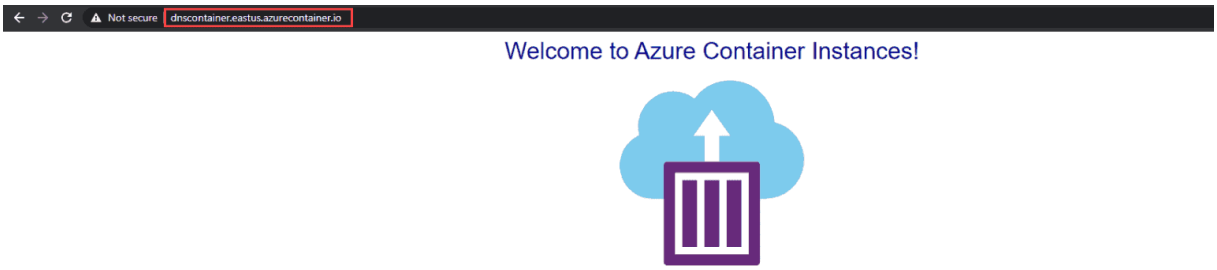

Рисунок 2.17 − Робота контейнера у Azure Container Instances

2.3.8 Аналіз платформи Amazon Web Services

AWS Fargate — це простий спосіб розгорнути свої контейнери на AWS. Простіше кажучи, Fargate схожий на EC2, але замість віртуальної машини ви отримуєте контейнер. Це обчислювальний механізм, який дозволяє використовувати контейнери як фундаментальний обчислювальний примітив без необхідності керувати базовими екземплярами. Все, що вам потрібно зробити, це створити образ контейнера, вказати вимоги до ЦП і пам’яті, визначити мережеву політику та політику IAM і запустити. Завдяки Fargate у вас є гнучкі параметри конфігурації, які точно відповідають потребам вашої програми, і ви платите за секунду.

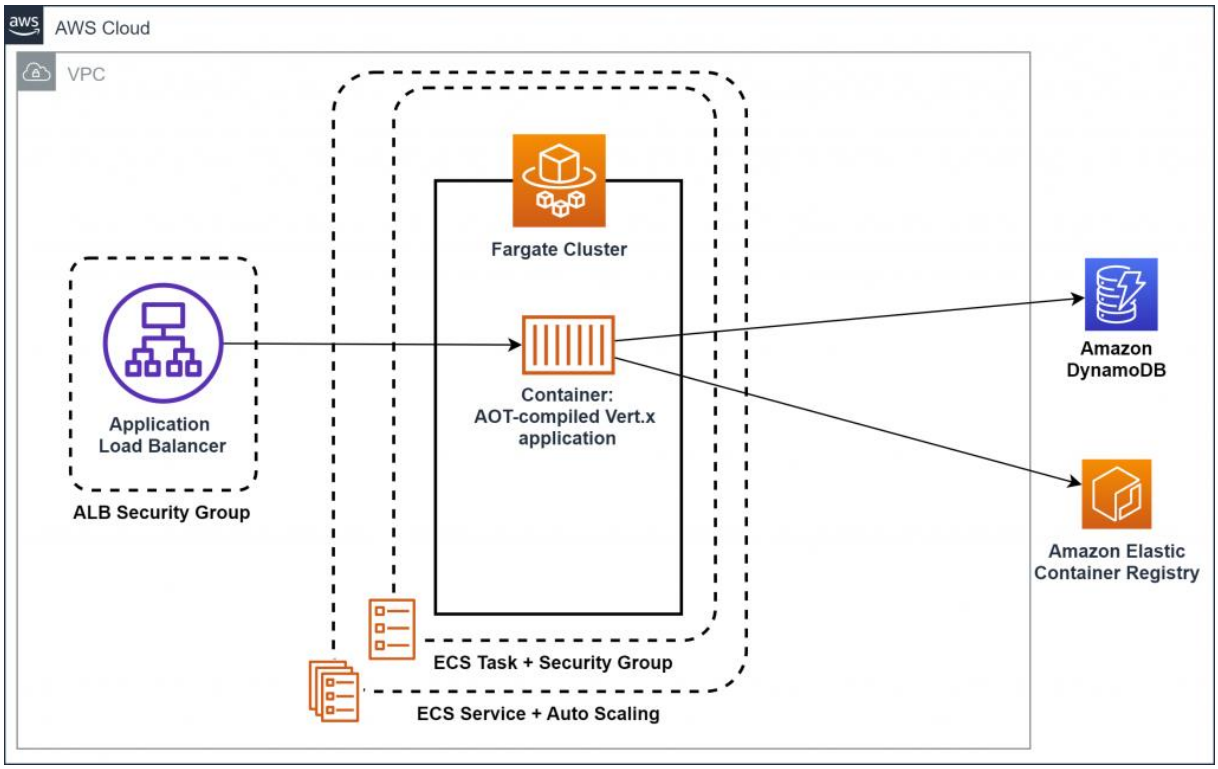

Рисунок 2.18 − Архітектура AWS Fargate

2.3.8.1 Приклад запуску додатка у AWS Fargate

Увійдіть до облікового аккаунту AWS. Щоб надіслати зображення до приватного сховища ECR, вам потрібно використовувати AWS CLI

Спочатку згенеруйте ключ доступу. Натисніть назву свого облікового запису у верхньому лівому куті консолі управління AWS. Потім виберіть "облікові дані безпеки".

Ви будете перенаправлені на консоль управління IAM, а потім натисніть "Створити ключ доступу". Буде створено ключ доступу, і ви можете завантажити його як файл CSV, що містить ваш ідентифікатор ключа доступу та секретний ключ доступу

У своєму командному рядку введіть AWS configure. Вам буде запропоновано ввести ідентифікатор ключа доступу, секретний ключ доступу, назву регіону за замовчуванням та формат виводу за замовчуванням. Якщо все правильно, процес входу в систему пройде успішно.

Перейдіть на сторінку ECS і виберіть розділ сховища. Натисніть "Створити сховище", слід вказати лише назву сховища. Завантажимо наш тестовий контейнер до AWS ECR. Далі потрібно створити AWS Fargate task definition. Перейдіть до визначень завдань у ECS. Ми можемо вказати кілька конфігурацій у визначеннях завдань, таких як використовуваний образ Docker, процесор і пам'ять для контейнера, тип екземпляра, мережевий режим і т. д.

Задаймо ім'я для AWS Fargate task definition, наприклад, ikol-master-diploma-fargate-task. Потім натисніть кнопку Далі та налаштуймо системні характеристики. Виберіть AWS Fargate для середовища програми, далі оберемо Linux як операційну систему та1vcpu та 2 ГБ пам'яті.

Наступним потрібно створити кластер Потрібно вказати назву кластера, вибрати VPC та підмережі.

За замовчуванням кластер налаштований для AWS Fargate. Так що давайте залишимо все як є. Давайте використовуємо наш VPC за замовчуванням.

Для зіставлень ви можете вибрати принаймні дві зони доступності. Нам доведеться вибрати те саме під час створення кластерних служб ECS.

Отже ми створили AWS Fargate task з назвою ikol-master-diploma-fargate-task, далі нам потрібно створити AWS Fargate, який буде запускати нашу AWS Fargate task.

Отже, обираємо наш кластер, тип запуску виберіть Fargate, виберіть наше завдання.

Нарешті, перейдіть до перегляду та створення.

Рисунок 2.19 − Робота контейнера у AWS Fargate

2.4 Висновки та обґрунтування вибору хмарного рішення

Після аналізу існуючих хмарних рішень, було прийнято рішення робити модернізацію системи на платформі AWS. AWS Fargate дозволяє нам завантажувати свої додатки, які контейнеризовані, створювати та налаштовувати VPC. Також ми платимо тільки тоді, коли робимо запит до додатку то це майже ідеальне рішення та вимога до системи у нашому випадку.

3 СИНТЕЗ СИСТЕМИ

3.1 Призначення й область застосування додатку Е-Завдання

Додаток Е-Завдання повинно слідкувати та сповіщати учнів та їх батьків про успіхи у навчальному процесі. До цього відноситься, перегляд оцінок, отримання нового домашнього завдання, перегляд статусу виконання домашнього завдання.

Даний додаток повинен бути контейнеризований та запущений на сервері у хмарній архітектури. Маючи змогу бути доступним для користувача за попитом.

Область застосування - Спеціалізована школа №13

3.2 Розробка мережі додатка Е-Завдання

За замовчуванням для кожного завдання Amazon ECS у Fargate надається еластичний мережевий інтерфейс (ENI) із основною приватною IP-адресою. У разі використання загальнодоступної підмережі ви можете додатково призначити загальнодоступну IP-адресу ENI завдання. Якщо ваш VPC увімкнено для режиму подвійного стеку, і ви використовуєте підмережу з блоком IPv6 CIDR, ENI вашого завдання також отримує адресу IPv6. Завдання може мати лише один ENI, пов’язаний із ним одночасно. Контейнери, які належать до однієї задачі, також можуть спілкуватися через інтерфейс localhost.

Щоб Fargate отримало зображення контейнера, завдання має мати маршрут до Інтернету:

– У разі використання публічної підмережі ви можете призначити загальнодоступну IP-адресу для завдання ENI.

– У разі використання приватної підмережі до неї може бути підключений шлюз NAT.

У разі використання зображень контейнерів, розміщених в Amazon ECR, ви можете налаштувати Amazon ECR на використання кінцевої точки інтерфейсу VPC, а отримання зображення відбувається через приватну адресу IPv4 завдання.

Оскільки кожне завдання отримує власний ENI, ви можете використовувати такі мережеві функції, як VPC Flow Logs, які можна використовувати для моніторингу трафіку до та від ваших завдань.

Створимо власну мережу в AWS

```
const vpcConfig = createVpcCr(this, {
    accountId: stackConfiguration.account,
    region: stackConfiguration.region,
    envClassification: account.environmentClassification(),
    vpcName: stackConfiguration.vpcName,
    vpcOwnerAccountId: stackConfiguration.vpcOwner,
    excludeVpcEndpoints: true,
    isElb: false,
    autoDiscoverSubnets: stackConfiguration.subnetAutoDiscovery,
});
```

Рисунок 3.1 − Лістинг VPC мережі додатку E-Завдання

3.3 Розробка IAM roles додатку E-Завдання

AWS Identity and Access Management (IAM) — це веб-служба, яка допомагає безпечно контролювати доступ до ресурсів AWS. Ви використовуєте IAM, щоб контролювати, хто пройшов автентифікацію (увійшов) і авторизований (має дозволи) на використання ресурсів.

Існує два типи політик: Identity-based policy and Resource-based policy.

Політики Identity-based policy - це документи політики дозволів, які можна прикріпити до, наприклад, користувача IAM, групи користувачів або ролі. Ці політики визначають, які дії можуть виконувати користувачі та ролі, на яких ресурсах та за яких умов

Політики Resource-based – це документи політики, які прикріплюються до ресурсу. Прикладами політик на основі ресурсів є політики довіри ролей IAM та політики бакетів Amazon S3. У службах, які підтримують політики на основі ресурсів, адміністратори служб можуть використовувати їх для керування доступом до певного ресурсу. Для ресурсу, якого прикріплена політика, політика визначає, які дії зазначений принципал може виконувати з цим ресурсом і за яких умов.

Створимо власну дві IAM ролі, перша буде відповідальна за взаємодію з Fargate cluster – taskRole, друга буде відповідальна за взаємодію усередині Fargate cluster з іншими AWS сервісами

```
const taskRole = Role.fromRoleArn(
    this,
    'EcsModelTaskRole',
    `arn:aws:iam::${stackConfiguration.account}:role/${stackConfiguration.taskRole}`,
    {
      mutable: false,
    },
  );
  const taskExecutionRole = Role.fromRoleArn(
    this,
    'EcsModelExecutionRole',
    `arn:aws:iam::${stackConfiguration.account}:role/${stackConfiguration.executionRole}`,
    {
      mutable: false,
    },
  );
```

Рисунок 3.2 − Лістинг IAM roles додатку Е-Завдання

3.4 Розробка AWS Fargate для запуску додатка Е-Завдання

AWS Fargate Task — це запущений набір контейнерів на одному хості. Також можна почути «завдання» та «контейнер», що використовуються як синоніми. Оскільки, завдання — це одиниця роботи, яку ECS запускає та керує у кластері. Завдання може бути одним контейнером або кількома контейнерами, які працюють разом.

Створимо своє завдання

```
const taskDefinition = new TaskDefinition(this, 'TaskDefinition', {
  family: `${stackConfiguration.modelName}-${stackConfiguration.modelEnv}`,
  compatibility: Compatibility.FARGATE,
  taskRole: taskRole,
  executionRole: taskExecutionRole,
  cpu: stackConfiguration.cpu,
  memoryMiB: stackConfiguration.memory,
  networkMode: NetworkMode.AWS_VPC,
});
```

Рисунок 3.3 − Лістинг Fargate Task

Далі створимо Container definition у якому визначмо які Fargate Tasks будуть запущені та з якими вимогами до ЦП і пам’яті.

```
const containerDefinition = new ContainerDefinition(this, 'ContainerDefinition', {
  containerName: stackConfiguration.name,
  taskDefinition: taskDefinition,
  memoryLimitMiB: Number(stackConfiguration.memory),
  image: ContainerImage.fromRegistry(stackConfiguration.image),
  essential: true,
  environment: {
    ...stackConfiguration.envvars,
  },
  stopTimeout: Duration.seconds(2)
  logging: stackConfiguration.logDriverConfig.createLogDriver(),
  dockerLabels: dockerLabels,
});
```

Рисунок 3.4 − Лістинг Fargate container definition

Ще нам потрібно додати нашу приватну мережу до container definition

```
containerDefinition.addPortMappings({
    containerPort: stackConfiguration.containerPort,
    hostPort: stackConfiguration.containerPort,
  });

  const securityGroup = new SecurityGroup(this, 'SecurityGroup', {
    vpc: vpcConfig.vpc,
    description: 'aws-sg-' + stackConfiguration.name,
  });
  securityGroup.addIngressRule(
    Peer.ipv4('10.0.0.0/8'),
    Port.tcp(stackConfiguration.containerPort),
  );
  securityGroup.addIngressRule(Peer.ipv4('10.0.0.0/8'), Port.udp(8125), 'datadog (custom metrics)');
  securityGroup.addIngressRule(Peer.ipv4('10.0.0.0/8'), Port.tcp(8126), 'datadog (app trace)');
  securityGroup.addIngressRule(Peer.ipv4('10.0.0.0/8'), Port.tcp(443), 'private network');
  securityGroup.addIngressRule(Peer.ipv4('172.16.0.0/12'), Port.tcp(443), 'private network');
  securityGroup.addIngressRule(Peer.ipv4('192.168.0.0/16'), Port.tcp(443), 'private network');

  const securityGroups: ISecurityGroup[] = [securityGroup];
```

Рисунок 3.5 − Додання приватної мережі до container definition

Та останнє, що залишилось це створити AWS Fargate service.

```
const service = new FargateService(this, 'FargateService', {
 cluster: cluster,
 desiredCount: stackConfiguration.desiredCount,
 taskDefinition: taskDefinition,
 healthCheckGracePeriod: Duration.seconds(30),
 securityGroups: securityGroups,
 propagateTags: PropagatedTagSource.TASK_DEFINITION,
 vpcSubnets: { subnets: vpcConfig.subnets },
});
```

Рисунок 3.6 − Додання приватної мережі до AWS Fargate Service

3.5 Висновки до розділу

У розділі було розроблено та налаштовано хмарну архітектуру згідно з вимогами кваліфікаційної роботи. Також було створено хмарну мережу з автоматичним розподілом IP адрес внутрішнім сервісам, які використовуються у хмарній архітектурі.

4 ЕКСПЕРИМЕНТАЛЬНИЙ РОЗДІЛ

4.1 Вимоги до експерименту

Для проведення експерименту є наступні вимоги:

– контейнеризація додатка за допомогою технології Docker;

– деплой додатку за допомогою Bamboo CI/CD pipeline;

– використати створену AWS VPC мережу;

– встановити додаток на AWS Fargate Service

4.2 Контейнеризація додатка додатку Е-Завдання

Docker може автоматично створювати образи, читаючи вказівки з файлу. Dockerfile - це текстовий документ, що містить усі команди, які користувач може викликати у командному рядку для збирання зображення. Використовуючи docker build, користувач може створити автоматизований процес складання, який послідовно виконує кілька інструкцій командного рядка.

```
#Package stage
FROM maven:3.8.1-jdk-11 AS build
COPY src hometask/src
COPY pom.xml /hometask
COPY settings.xml hometask/setting.xml
RUN mvn -f hometask/pom.xml -s hometask/setting.xml clean package

#Run stage
FROM openjdk:11-jdk
COPY --from=build hometask/target/*.jar hometask.jar
ENTRYPOINT ["java","-jar","/hometask.jar"]
```

Рисунок 4.1 − Dockerfile

Демон Docker запустить інструкції в Dockerfile згори донизу, кожен крок кешується та буде доступний у разі перебудові Docker контейнеру, та включить у себе нові параметри. Тут ми можемо побачити те, що спочатку копіюємо налаштування і програмний код з файлової директорії до директорії контейнеру. Потім контейнер завантажую мову програмування Java 11 та встановлює її до контейнеру. І останній крок інструкція до запуску додатку.

4.3 Деплой контейнера додатка E-Завдання до AWS

Для деплою додатка до AWS Fargate будемо використовувати Bamboo CI/CD pipeline. Після початку деплою до AWS Fargate ми бачимо початкову метаінформацію про загальні налаштування

```
AWS::Logs::LogGroup                         |
ModelContainerDefinition/LogGroup
(ModelContainerDefinitionLogGroup12EEEB3B)
AWS::Logs::LogGroup                         |
DDAgentContainerDefinition/LogGroup
(DDAgentContainerDefinitionLogGroup910D6499)
AWS::Logs::LogGroup                         |
ModelContainerDefinition/LogGroup
(ModelContainerDefinitionLogGroup12EEEB3B) Resource creation Initiated
AWS::Logs::LogGroup                         |
DDAgentContainerDefinition/LogGroup
(DDAgentContainerDefinitionLogGroup910D6499) Resource creation Initiated
AWS::CDK::Metadata                          | CDKMetadata
AWS::Logs::LogGroup                         |
ModelContainerDefinition/LogGroup
(ModelContainerDefinitionLogGroup12EEEB3B)
AWS::CloudFormation::CustomResource         | Account
AWS::Logs::LogGroup                         |
DDAgentContainerDefinition/LogGroup
(DDAgentContainerDefinitionLogGroup910D6499)
AWS::CDK::Metadata                          | CDKMetadata Resource creation
Initiated
AWS::CDK::Metadata                          | CDKMetadata
AWS::CloudFormation::CustomResource         | Account Resource creation
Initiated
AWS::CloudFormation::CustomResource         | Account
```

Рисунок 4.2 − Початкова метаінформація

Далі повинні створитися AWS IAM role, які ми розробили у розділі 4.2

```
AWS::IAM::Role                               | ECSTaskExecutionRole
(ECSTaskExecutionRole911F5A4F)
AWS::IAM::Role                               | ECSTaskExecutionRole
(ECSTaskExecutionRole911F5A4F) Resource creation Initiated
AWS::IAM::Role                               | ECSTaskExecutionRole
(ECSTaskExecutionRole911F5A4F)
AWS::IAM::Policy                             |
ECSTaskExecutionRole/DefaultPolicy
(ECSTaskExecutionRoleDefaultPolicyC25F7D27)
```

Рисунок 4.3 − Створення IAM roles

Наступним буде створення AWS VPC, які були розроблені та налаштовані у розділі 4.1

```
AWS::CloudFormation::CustomResource       | VPCCR
AWS::CloudFormation::CustomResource       | VPCCR Resource creation
Initiated
AWS::CloudFormation::CustomResource       | VPCCR
AWS::CloudFormation::CustomResource       | PrivateElbAz2
AWS::ElasticLoadBalancingV2::TargetGroup  | ALBTargetGroup
(ALBTargetGroup04621599)
AWS::CloudFormation::CustomResource       | PrivateSubnetAz2
AWS::CloudFormation::CustomResource       | PrivateElbAz1
AWS::EC2::SecurityGroup                   | SecurityGroup
(SecurityGroupDD263621)
AWS::CloudFormation::CustomResource       | PrivateSubnetAz1
AWS::ElasticLoadBalancingV2::TargetGroup  | ALBTargetGroup
(ALBTargetGroup04621599) Resource creation Initiated
AWS::ElasticLoadBalancingV2::TargetGroup  | ALBTargetGroup
(ALBTargetGroup04621599)
AWS::CloudFormation::CustomResource       | PrivateElbAz2 Resource
creation Initiated
AWS::CloudFormation::CustomResource       | PrivateSubnetAz2 Resource
creation Initiated
AWS::CloudFormation::CustomResource       | PrivateElbAz2
AWS::CloudFormation::CustomResource       | PrivateElbAz1 Resource
creation Initiated
AWS::CloudFormation::CustomResource       | PrivateSubnetAz2
AWS::CloudFormation::CustomResource       | PrivateSubnetAz1 Resource
creation Initiated
AWS::CloudFormation::CustomResource       | PrivateElbAz1
AWS::CloudFormation::CustomResource       | PrivateSubnetAz1
AWS::EC2::SecurityGroup                   | SecurityGroup
(SecurityGroupDD263621) Resource creation Initiated
AWS::EC2::SecurityGroup                   | SecurityGroup
(SecurityGroupDD263621)
```

Рисунок 4.4 – Створення AWS VPC

Останнім та найважливішим кроком у нашому деплою, є створення AWS Fargate task, яка і буде запускати наш додаток у хмарному середовищі і буде з’єднано з ресурсами AWS VPC та AWS IAM role, які ми вже створили

```
AWS::ElasticLoadBalancingV2::LoadBalancer | LoadBalancer
(LoadBalancerBE9EEC3A) Resource creation Initiated
AWS::KMS::Key                             | KMSKey (KMSKeyBD866E3F)
AWS::KMS::Alias                           | KMSKey/Alias
(KMSKeyAlias623A6BDB)
AWS::CloudFormation::CustomResource       | KmsEncrypt1
AWS::CloudFormation::CustomResource       | KmsEncrypt3
AWS::CloudFormation::CustomResource       | KmsEncrypt2
AWS::KMS::Alias                           | KMSKey/Alias
(KMSKeyAlias623A6BDB) Resource creation Initiated
AWS::KMS::Alias                           | KMSKey/Alias
(KMSKeyAlias623A6BDB)
AWS::CloudFormation::CustomResource       | KmsEncrypt2 Resource creation
Initiated
AWS::CloudFormation::CustomResource       | KmsEncrypt1 Resource creation
Initiated
AWS::CloudFormation::CustomResource       | KmsEncrypt3 Resource creation
Initiated
AWS::CloudFormation::CustomResource       | KmsEncrypt1
AWS::CloudFormation::CustomResource       | KmsEncrypt2
AWS::CloudFormation::CustomResource       | KmsEncrypt3
AWS::ECS::TaskDefinition                  | TaskDefinition
(TaskDefinitionB36D86D9)
AWS::ECS::TaskDefinition                  | TaskDefinition
(TaskDefinitionB36D86D9) Resource creation Initiated
AWS::ECS::TaskDefinition                  | TaskDefinition
(TaskDefinitionB36D86D9)
AWS::ElasticLoadBalancingV2::LoadBalancer | LoadBalancer
(LoadBalancerBE9EEC3A)
AWS::CloudFormation::CustomResource       | Cname
AWS::ElasticLoadBalancingV2::Listener     | LoadBalancer/ListenerHttps
(LoadBalancerListenerHttps6E420924)
AWS::ElasticLoadBalancingV2::Listener     | HttpRedirect
AWS::ElasticLoadBalancingV2::Listener     | LoadBalancer/ListenerHttps
(LoadBalancerListenerHttps6E420924) Resource creation Initiated
AWS::ElasticLoadBalancingV2::Listener     | HttpRedirect Resource creation
Initiated
AWS::ElasticLoadBalancingV2::Listener     | LoadBalancer/ListenerHttps
(LoadBalancerListenerHttps6E420924)
AWS::ElasticLoadBalancingV2::Listener     | HttpRedirect
AWS::CloudFormation::CustomResource       | Cname Resource creation
Initiated
AWS::CloudFormation::CustomResource       | Cname
AWS::ECS::Service                         | FargateService/Service
(FargateServiceAC2B3B85)
AWS::ElasticLoadBalancingV2::ListenerRule | ListenerRuleHttps
AWS::ElasticLoadBalancingV2::ListenerRule | ListenerRuleHttps Resource
creation Initiated
AWS::ECS::Service                         | FargateService/Service
(FargateServiceAC2B3B85) Resource creation Initiated
AWS::ElasticLoadBalancingV2::ListenerRule | ListenerRuleHttps
AWS::ECS::Service                         | FargateService/Service
(FargateServiceAC2B3B85)
AWS::CloudFormation::Stack                | hometask
```

Рисунок 4.5 – Створення Fargate Task

4.4 Перевірка роботоспроможності додатку Е-Завдання

Після того, як ми задеплоїли наш додаток до хмарного рішення AWS Fargate, потрібно перевірити робото спроможність додатку. Зайдемо на головну сторінку веб додатку.

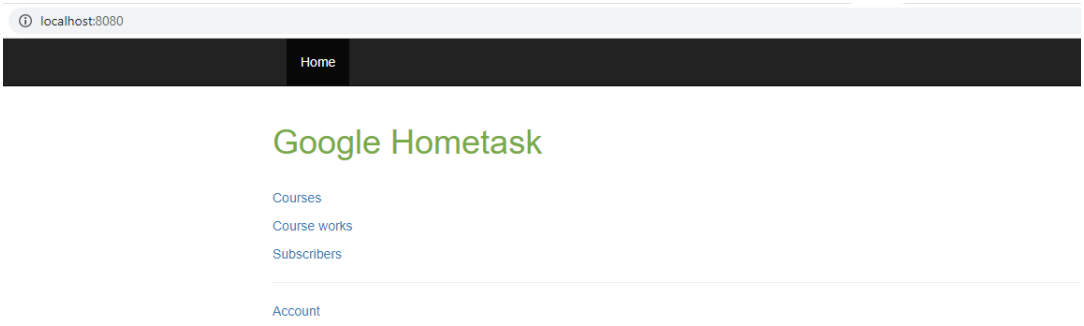

Рисунок 4.6 – Головна сторінка веб-додатку Е-Завдання

Як бачимо ми змогли успішно зайти до веб варіанту додатка. Далі спробуємо зробити запит до телеграм версії додатку Е-Завдання.

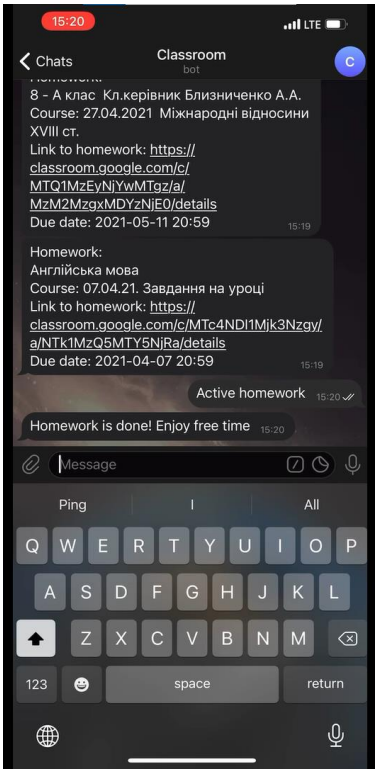

Рисунок 4.7 – Інтерфейс додатку у Telegram

4.5 Висновки до розділу

Можемо побачити, що ми успішно контейнеризували додаток за допомогою технології докер, завантажили його до хмарного середовища AWS Fargate та додаток працює як і раніше.

ВИСНОВКИ

За завданням кваліфікаційної роботи магістра було досліджено та проаналізовано існуючі рішення для оптимізації інфаструктури. Отримано навички у сфері контейнеризації додатків та сервісів., їх характеристики та переваги. Були порівняні оркестратори, їх характеристики та їхня актуальність на ринку. В результаті аналізу існуючих рішень контейнеризації, та враховуючи поточну архітектуру додатку було обрано рішення від Amazon Web Services Fargate, що відповідає всім критеріям. Для налаштування та налагодження оточення були використані інструменти, такі як AWS CDK та формат конфігураційних файлів YAML. Додаток E-Завдання було перероблено для запуску у хмарній архітектури. Вимоги кваліфікаційної роботи було виконано

ПЕРЕЛІК ПОСИЛАНЬ

1. Що таке докер [Електронний ресурс] - shorturl.at/duU35

2. AWS документація [Електронний ресурс] - https://aws.amazon.com/ru/what-is-aws/

3. Хмарна архітектура [Електронний ресурс] - shorturl.at/mtxT9

4. Порівняння контейнерів [Електронний ресурс] - https://www.tutorialworks.com/difference-docker-containerd-runc-crio-oci/

5. Azure документація [Електронний ресурс] - https://azure.microsoft.com/ru-ru/resources/cloud-computing-dictionary/what-is-virtualization/

6. Google Cloud документація [Електронний ресурс] - https://www.howtogeek.com/devops/how-to-run-docker-containers-on-google-cloud-platform/

7. Що таке OCI-R [Електронний ресурс] - https://www.ionos.com/digitalguide/server/know-how/what-is-cri-o/

8. Що таке Linux LXC [Електронний ресурс] - https://bg.theastrologypage.com/linux-containers/

ДОДАТОК А

Текст налаштувань інфраструктури додатка Е-Завдання

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
“ДНІПРОВСЬКА ПОЛІТЕХНІКА”

ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ОНЛАЙН-КОНСУЛЬТАНТА

текст програми

804.02070743.20005-01 12 01

Листів 5

2022

АНОТАЦІЯ

Дана програма містить в собі частину програмного коду для налаштування хмарної архітектури та хмарної мережі

Налаштування робляться одноразово та має можливість внесення

Зміст

2. Лістинг хмарної архітектури

```
import { App, StackProps, CfnOutput, CustomResource, RemovalPolicy, SecretValue, Tags } from 'aws-cdk-lib';
import { DatabaseClusterEngine, AuroraPostgresEngineVersion, ParameterGroup, DatabaseCluster } from 'aws-cdk-lib/aws-rds';
import { Topic } from 'aws-cdk-lib/aws-sns';
import { InstanceType, InstanceClass, InstanceSize, Subnet, SecurityGroup, Peer, Vpc, Port } from 'aws-cdk-lib/aws-ec2';
import {
  AccountCustomResource,
  SubnetCustomResource,
  VPCCustomResource,
  DBClusterActivityStreamCustomResource;
import { StackConfiguration } from './configuration/stack-configuration';

export class {
  constructor(scope: App, id: string, props?: StackProps) {
    const PROD_ENV_NAME = 'production'
    let properties: StackProps = {
      env: {
        region: StackConfiguration.region
      }
    }

    super(scope, id, properties);

    let subnetName: string = "private-2";
    let instanceNum: number = 1;
```

```
        let instanceType = InstanceType.of(InstanceClass.BURSTABLE3,
InstanceSize.MEDIUM);
        let instanceRemovalPolicy = RemovalPolicy.DESTROY
        let lmSchedule: string = StackConfiguration.lmScheduleDefault

        if ([PROD_ENV_NAME,
"development"].includes(StackConfiguration.currentEnv)) {
          instanceType =
InstanceType.of(InstanceClass.MEMORY6_GRAVITON,
InstanceSize.LARGE);
          if (StackConfiguration.currentEnv == PROD_ENV_NAME) {
          subnetName = "private";
          instanceNum = 2;
          lmSchedule = "none";
          instanceRemovalPolicy = RemovalPolicy.RETAIN;
          }
        }

        const account: AccountCustomResource = new
AccountCustomResource(this, "V12Account");

        const vpcCustomResource: VPCCustomResource = new
VPCCustomResource(this, "V12VPCCustomResource", {
          vpcEnv: account.environmentClassification(),
          excludeVpcEndpoints: true
        });

        const subnetCustomResource1: SubnetCustomResource = new
SubnetCustomResource(this, "V12SubnetCustomResource1", {
          az: "1",
```

```
      name: subnetName,
      vpcId: vpcCustomResource.vpcId(),

    });

    const subnetCustomResource2: SubnetCustomResource = new SubnetCustomResource(this, "V12SubnetCustomResource2", {
      az: "2",
      name: subnetName,
      vpcId: vpcCustomResource.vpcId()
    });

    const customResourceTopic = Topic.fromTopicArn(this, 'customResourceTopic', 'arn:aws:sns:us-east-1:12467948123:custom-resource')

    const randomCustomResource = new CustomResource(this, 'RandomCustomResource', {
      resourceType: 'Custom::ResourceRandom',
      serviceToken: customResourceTopic.topicArn,
      properties: {
        Resource: 'Random',
      }
    });

    const envVPC = Vpc.fromVpcAttributes(this, 'envVPC', {
      vpcId: vpcCustomResource.vpcId(),
      availabilityZones: [
        'us-east-1a',
        'us-east-1b'
      ]
```

```
        });

        const envSubNet1 = Subnet.fromSubnetAttributes(this, 'EnvSubnetAZ1', {
          subnetId: subnetCustomResource1.ref,
          availabilityZone: subnetCustomResource1.availabilityZone()
        });

        const envSubNet2 = Subnet.fromSubnetAttributes(this, 'EnvSubnetAZ2', {
          subnetId: subnetCustomResource2.ref,
          availabilityZone: subnetCustomResource2.availabilityZone()
        });

        /* Create new AWS resources for Aurora DB needs: Security group, Cluster/DB parameter group and so on */
        const rwdbSG = new SecurityGroup(this, 'RunwayDBSecurityGroup', {
          vpc: envVPC,
          securityGroupName: StackConfiguration.name,
          description: 'Security group that allows access to Runway database cluster',
          allowAllOutbound: true
        });

        const rdsEngine = DatabaseClusterEngine.auroraPostgres({
          version: AuroraPostgresEngineVersion.VER_12_8
        });
```