МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Чорноморський національний університет
імені Петра Могили
Факультет комп’ютерних наук
Кафедра інтелектуальних інформаційних систем

ДОПУЩЕНО ДО ЗАХИСТУ
Завідувач кафедри інтелектуальних
інформаційних систем, д-р техн. наук,
проф.___________Ю. П. Кондратенко
«____» ______________ 2022 р.

МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

ІНТЕЛЕКТУАЛЬНА СИСТЕМА КЕРУВАННЯ ІНФРАСТРУКТУРОЮ ДЛЯ ХМАРНИХ ОБЧИСЛЕНЬ

Спеціальність 122 «Комп’ютерні науки»

122 – МКР – 601.21830303

Студент ________________ О. О. Попель
«14» лютого 2022 р.

Консультант ____________ О.П. Гожий
докт. техн. наук, професор
«14» лютого 2022 р.

Миколаїв – 2022

ЗМІСТ

ВСТУП..4
ПЕРЕЛІК СКОРОЧЕНЬ..6
1 АНАЛІЗ ПРЕДМЕТНОЇ СФЕРИ. ПОСТАНОВКА ЗАДАЧІ...8
1.1 Опис сфери керування інфраструктурою для хмарних обчислень...8
1.2 Огляд способів керування хмарною інфраструктурою. Аналіз наявних публікацій...........11
1.2.1 Ручне керування хмарною інфраструктурою...12
1.2.2 Автоматичне керування хмарною інфраструктурою..15
1.2.3 Програмне керування хмарною інфраструктурою..16
1.2.4 Інфраструктура як код...17
1.2.5 Інфраструктура, що керується хмарним провайдером (PaaS)..18
1.3 Постановка задачі магістерської кваліфікаційної роботи..19
Висновки до розділу 1...21
2 ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ ТА СЕРВІСИ — СУБ’ЄКТИ ДОСЛІДЖЕННЯ.....................................22
2.1 Провайдери хмарних послуг...23
2.2 Програмні системи, засоби та компоненти для створення суб’єкта основного експерименту...28
2.3 Інструменти керування хмарною інфраструктурою за принципом Infrastructure as Code. .32
Висновки до розділу 2...36
3 МОДЕЛЮВАННЯ ЕКСПЕРИМЕНТАЛЬНОГО СЕРЕДОВИЩА ТА СТВОРЮВАНОЇ СИСТЕМИ...38
3.1 Вибір провайдера хмарних послуг...38
3.2 Обґрунтування технологій для використання при створенні суб’єкта експерименту.........41
3.3 Вибір інструменту керування хмарною інфраструктурою...46
3.4 Загальна структура та модель створюваної системи...49
Висновки до розділу 3...51
4 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ..52
4.1 Створення початкового експериментального середовища..52
4.2 Впровадження принципу Infrastructure as Code..59
4.3 Аналіз результатів...69
Висновки до розділу 4...72
5 МЕТОДИЧНА ЧАСТИНА...74
5.1 Практична робота №1. Керування хмарною інфраструктурою за допомогою веб-інтерфейсу AWS..74

5.2 Практична робота №2. Керування хмарною інфраструктурою за допомогою IaC-інструменту Terraform....77
6 ОХОРОНА ПРАЦІ ПІД ЧАС ДИСТАНЦІЙНОЇ РОБОТИ В ДОМАШНІХ УМОВАХ....81
6.1 Оцінка релевантних вимог до охорони праці та визначення відповідності їм місця роботи. 81
6.1.1 Вимоги щодо безпеки та захисту здоров'я працівників під час роботи з екранними пристроями....81
6.1.2 Санітарні норми виробничого шуму, ультразвуку та інфразвуку....87
6.1.3 Санітарні норми мікроклімату виробничих приміщень....89
6.1.4 Оцінка відповідності робочого місця поставленим вимогами....91
6.2 Заходи безпеки при виникненні надзвичайних ситуацій....92
6.2.1 Правила запобігання пожежам та дії під час їх виникнення....92
6.2.2 Оцінка стану пожежної безпеки в робочому приміщенні....93
Висновки до розділу 6....94
ВИСНОВКИ....96
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ....97
ДОДАТОК А Програмний код....101

ВСТУП

За оцінками експертів, на сьогоднішній день хмарні дата-центри відповідають за 95% усього інтернет-трафіку [1] і ця цифра продовжує збільшуватись. Вплив хмарних технологій на сферу інформаційних технологій став очевидним ще десятиліття тому, але досі вони продовжують активно розвиватись і змінювати принципи за якими працює глобальна світова інфраструктура.

Одним з відносно молодих принципів, які стали можливими лише після популяризації хмарних обчислень є «Infrastructure as Code» — принцип, який пропонує скористатись можливостями, які надають хмарні провайдери задля представлення апаратної обчислювальної інфраструктури у вигляді програмного коду. Сфера програмування розвивається уже більше ніж пів століття і за цей час було розроблено багато інструментів, методик та принципів, які дозволили значно збільшити продуктивність усієї сфери. «Infrastructure as Code» пропонує застосувати усі ці принципи до побудови не тільки програмних, але і апаратних систем. Проте, велика кількість організацій продовжують керувати хмарною інфраструктурою традиційними методами. Враховуючи, що «Infrastructure as Code» є одним із наймолодших і найменш досліджених способів керування інфраструктурою, не можна зробити чітких висновків про його переваги та недоліки, не дослідивши його використання експериментально, на практиці.

Отже, основним завданням роботи стане побудова інтелектуальної системи керування інфраструктурою для хмарних обчислень на основі підходу «Infrastructure as Code», метою якого буде визначення особливостей використання такого підходу на практиці — для керування інфраструктурою типового хмарного програмного проекту.

Виходячи з цього, об’єктом дослідження буде інтелектуальна система, що здатна керувати хмарною інфраструктурою за принципом «Infrastructure as Code».

Предметом дослідження буде принцип «Infrastructure as Code», а саме його можливості, особливості, переваги, недоліки та випадки, коли його використання є доцільнішим за альтернативні методи керування хмарною інфраструктурою.

В результаті виконання роботи, буде проведено аналіз предметної сфери та описано особливості різних способів керування хмарною інфраструктурою, проаналізовано наявні публікації на дану тему і чітко сформовано конкретні задачі, виконання яких необхідне для досягнення мети даної роботи. З урахуванням поставлених задач, буде проведено огляд інформаційних технологій та сервісів, використання яких стане необхідним для успішного виконання основного завдання роботи. До таких сервісів та технологій відносяться хмарні провайдери, програмні компоненти для створення суб'єкта дослідження, а також інструменти, що дозволяють керувати хмарною інфраструктурою за принципом «Infrastructure as Code». Серед розглянутих сервісів, на основі визначених критеріїв будуть обрані ті, що будуть використані у творчій частині роботи, під час якої буде побудовано типовий веб-застосунок (суб'єкт дослідження), впроваджено принципи Infrastructure as Code для керування його інфраструктурою і проведено аналіз отриманих результатів з оглядом досліджених характеристик підходу «Infrastructure as Code» та підбиттям підсумків.

ПЕРЕЛІК СКОРОЧЕНЬ

API — Application Programming Interface

AWS — Amazon Web Services

AWS CDK — AWS Cloud Development Kit

CDN — Content Delivery Network

CI/CD — Continuous Integration / Continuous Deployment (Delivery)

CMS — Content Management System

EC2 — Elastic Cloud Compute

ELB — Elastic Load Balancer

FTP — File Transfer Protocol

GCP — Google Cloud Platform

HCL — HashiCorp Configuration Language

HTTP — HyperText Transfer Protocol

HTTPS — HyperText Transfer Protocol Secure

IT — Information Technology

IaaS — Infrastructure as a Service

PSTN — Public Switched Telephone Network

PaaS — Platform as a Service

RDS — Relational Database Service

S3 — Simple Storage Service

SMB — Server Message Block

SMS — Short Message Service

SOA — Service-Oriented Architecture

SQL — Structured Query Language

URL — Universal Resource Locator

VOIP — Voice-Over-Internet Protocol

VPC — Virtual Private Cloud

VPS — Virtual Private Server

ОС — Операційна Система

Пояснювальна записка

до магістерської кваліфікаційної роботи

на тему:

«ІНТЕЛЕКТУАЛЬНА СИСТЕМА КЕРУВАННЯ ІНФРАСТРУКТУРОЮ ДЛЯ ХМАРНИХ ОБЧИСЛЕНЬ»

Спеціальність 122 «Комп’ютерні науки»

122 – МКР – 601.21830303

Студент ________________ О. О. Попель

«14» лютого 2022 р.

Консультант ____________ О.П. Гожий

докт. техн. наук, професор

«14» лютого 2022 р.

Миколаїв – 2022

1 АНАЛІЗ ПРЕДМЕТНОЇ СФЕРИ. ПОСТАНОВКА ЗАДАЧІ

1.1 Опис сфери керування інфраструктурою для хмарних обчислень

Після свого повноцінного запуску у 2006 році [2], Amazon Web Services (AWS) кардинально змінили світ IT-інфраструктури. До їх появи більшість компаній були змушені утримувати власні дата-центри, самостійно керувати фізичними серверами, проводити їх обслуговування, діагностику, заміну чи оновлення. Створення обчислювальних систем, які повинні бути доступні в робочому стані постійно було доступним лише тим, хто міг дозволити собі створення декількох дата-центрів в різних фізичних локаціях, з цілодобовим обслуговуванням та швидкими і ефективними протоколами виходу з будь-яких кризових ситуацій — від збоїв у роботі мережі до стихійних лих.

AWS надав можливість створювати надійні, відмовостійкі системи майже будь-кому — як великим компаніям, так і дрібним стартапам чи індивідуальним розробникам. Один із ранніх девізів AWS звучав як “надаємо розробникам змогу створювати інновативні та підприємницькі додатки власноруч” [3]. Взявши на себе обов’язки керування фізичним обладнанням (а пізніше і частинами програмного), AWS абстрагував велику частину роботи з IT-інфраструктурою, дозволивши клієнтам сфокусуватись на розробці програмних продуктів, які при цьому були надійнішими і, часто, фінансово вигіднішими ніж такі, що використовували власну IT-інфраструктуру. Пізніше таку інфраструктура назвали “хмарною”, адже вона є слабо видимою для клієнта, вона “просто десь існує” і, за гарантіями провайдера такого хмарного сервісу, майже завжди працює. AWS значно пришвидшили темпи інновацій в індустрії і сильно вплинули на високе збільшення кількості стартапів (яке ми спостерігаємо й досі) та на розвиток мережевих продуктів та сервісів загалом.

Незабаром, з’явились продукти-конкуренти від інших великих IT-компаній — Google Cloud Platform (GCP) та Microsoft Azure. А після них і дещо менші хмарні сервіси, такі як DigitalOcean, Hetzner чи Linode. Конкуренція у цій сфері

пришвидшила темпи розвитку хмарних сервісів і дуже швидко крім керованих серверів, мережевих файлових сховищ та балансувальників навантаження, разом названих Infrastructure as a Service (IaaS, Інфраструктура як Сервіс), почали з'являтись перші сервіси, які частково керували інфраструктурними програмними продуктами, такими як бази даних, веб-сервери, чи фаєрволи. Такий тип продуктів характеризували як Platform as a Service (PaaS, Платформа як Сервіс). Цікаво, що PaaS сервіси були популярними задовго до появи так званих Cloud-провайдерів (сервісів, що надають клієнтам доступ до хмарних сервісів, таких як AWS, GCP, Azure, тощо.) у вигляді веб-хостингів, що давали клієнтам змогу завантажувати власний код, найчастіше за допомогою протоколу FTP, на сервер, керований хостинг-провайдером. Часто, хостинг провайдери надавали клієнту також і доступ до керованої бази даних, або повноцінних CMS-систем. Тобто такі продукти самі по собі не були новими. Новим був масштаб ресурсів які можна було використовувати та рівень контролю над орендованими ресурсами, якщо такий був потрібен. І те, що їх можна було використовувати поруч із майже повністю програмно керованими самим клієнтом серверами.

Отже, підсумовуючи, основними перевагами хмарних сервісів стали:

— гнучкість використання ресурсів, майже невичерпний (для більшості користувачів) їх запас;

— в деяких випадках, зменшення витрат на інфраструктуру;

— зменшення складності обслуговування інфраструктури, адже більшість обов'язків на себе бере cloud-провайдер;

— швидкодія та правильність роботи обладнання, які гарантувались хмарним сервісом та трималися на очікуваному рівні його спеціалістами;

— покращення надійності та відмовостійкості програмних продуктів клієнтів, завдяки простоті розгортання інфраструктури у декількох фізичних локаціях та налаштуванню їх синхронізованої роботи, а також завдяки тому, що широкий клас проблем просто зник, з точки зору клієнтів, адже їх відсутність гарантував cloud-провайдер;

— безпека, завдяки простоті керування доступами користувачів, налаштування фаєрволів та дотриманню перевірених практик безпеки за замовчуванням.

На початку, керування хмарною інфраструктурою мало відрізнялось від керування власною, в дата-центрах, з тою відмінністю, що для створення хмарного серверу було достатньо натиснути одну кнопку, а не купувати обладнання, встановлювати його у дата-центрі, налаштовувати, встановлювати програмне забезпечення, тощо. Після створення сервери налаштовувались вручну, або за допомогою автоматичних скриптів, як і фаєрволи, балансувальники навантаження, мережеві файлові сховища і так далі. Але з часом cloud-провайдери почали створювати інструменти, що допомагали клієнтам керувати інфраструктурою автоматично. По-перше, це були програмувальні веб-інтерфейси (API), які дозволяли програмно керувати хмарною інфраструктурою, проте вимагали створення власних програмних засобів і прямої взаємодії з відносно низькорівневими інтерфейсами. По-друге, це були розширення існуючих сервісів, спрямовані на автоматичне керування кількістю виділених ресурсів, наприклад — EC2 AutoScaling в AWS, який дозволяв автоматично створювати нові чи знищувати зайві сервери в залежності від навантаження на існуючі. Але з часом як провайдери так і споживачі хмарних сервісів почали розуміти, що можливості cloud-сервісів дозволяють повністю керувати інфраструктурою за допомогою коду і навіть повністю представляти її у кодовому вигляді. Такий підхід назвали «Інфраструктура як код» (Infrastructure as Code, IaC) [4].

Infrastructure as Code — це підхід до створення та керування інфраструктурою за допомогою коду, а не ручних процесів. Інфраструктура у вигляді коду є, загалом, конфігураційними файлами, що частково чи повністю описують специфікацію певної IT-інфраструктури. Збереження схеми та стану інфраструктури у конфігураційних файлах значно спрощує редагування та розповсюдження моделі хмарних ресурсів проекту. До того ж, це забезпечує створення щоразу однакового інфраструктурного середовища, чого складно досягти при створенні інфраструктури вручну.

Важливою перевагою IaC-підходу є можливість використання систем контролю версій (git, mercurial, subversion, тощо), що є стандартом [5] у програмуванні, але донедавна не було доступно для інфраструктурної частини програмних продуктів. До того ж, інфраструктура у вигляді коду дозволяє використовувати створені раніше компоненти інфраструктури у інших проектах, розбивати інфраструктуру на модульні частини, які можна запускати у окремих середовищах і тестувати в ізоляції від інших частин системи. Крім цього, до головних переваг IaC можна віднести:

— пришвидшення розгортання нових версій продукту;

— зменшення кількості помилок;

— підвищення консистентності інфраструктури;

— запобігання так званому дрейфу конфігурації, коли з часом інфраструктура зазнає недокументованих змін і не може бути легко розгорнутою з нуля.

При цьому, розвиток PaaS-сервісів (наприклад: Heroku, Google App Engine) та керованих систем оркестрації контейнерів (наприклад: Google Kubernetes Engine, Elastic Kubernetes Service), абстрагують ще більшу частину інфраструктури і дозволяють запускати програмні продукти взагалі без прямого контролю над інфраструктурою. Переваги і недоліки різних способів керування інфраструктурою для хмарних обчислень, а також огляд і аналіз наявних публікацій на цю тему будуть описані в наступному розділі.

1.2 Огляд способів керування хмарною інфраструктурою. Аналіз наявних публікацій

Отже, підсумовуючи попередній розділ, можна визначити такі способи керування хмарною інфраструктурою:

1) ручний, за допомогою командного чи веб-інтерфейсу;

2) автоматичний, використовуючи можливості хмарних сервісів, такі як AutoScaling;

3) програмний, з використанням власних інструментів взаємодії з API cloud-провайдера;

4) інфраструктура як код (IaC), за допомогою IaC-інструментів та конфігураційних файлів опису хмарної інфраструктури;

5) керований провайдером, коли інфраструктурні ресурси є частково чи повністю абстрагованими cloud-провайдером і клієнт має дуже обмежені можливості (але і потреби) керування інфраструктурою.

Найчастіше, жоден спосіб не використовується окремо і інфраструктура керується комбінацією вищевказаних способів, інколи навіть усіма одразу. Наприклад, вручну створена база даних може існувати поруч з автоматично керованою групою серверів, яка в свою чергу за допомогою API cloud-провайдера отримує сигнали не лише від моніторингу ресурсів серверів, а і від запущеної на ньому програми; в свою чергу ця автоматично керована група серверів була створена одним із IaC-інструментів, який в свою чергу керується cloud-провайдером у цілях контролю доступу користувачів до ресурсів. Тобто, залучені усі вищезазначені способи.

При цьому, часто основна частина програмного продукту з хмарною інфраструктурою використовує один, той чи інший спосіб контролю інфраструктури, а інші є допоміжними. Розглянемо кожен зі способів більш детально.

1.2.1 Ручне керування хмарною інфраструктурою

Ручне керування є найбільш простим та примітивним способом керування інфраструктурою. Його суть полягає у використанні інтерфейсу [6], який надається cloud-провайдером для створення, налаштування, редагування чи знищення хмарних інфраструктурних ресурсів. Прикладами таких інтерфейсів керування є AWS Management Console — веб-інтерфейс керування сервісами AWS та командна утиліта gcloud, для керування сервісами GCP з командного рядка.

Основною перевагою ручного підходу до керування інфраструктурою є легкість у використанні — інтерфейси ручного керування є інтерактивними, часто надають підказки чи автоматично доповнюють команди, показують велику кількість інформації про інфраструктуру у зручному вигляді і дозволяють легко експериментувати з можливостями хмарних сервісів, а отже є незамінними при ознайомленні з ними. До того ж, багато простих задач швидше виконуються таким одноразовим ручним способом, ніж за допомогою використання API чи IaC-технологій. Вони створені саме для цього. Прикладом є створення балансувальника навантаження у Google Cloud Platform (рис. 1.1). Веб-інтерфейс дозволяє зробити це за допомогою трьох наглядних кроків, в кожному з яких налаштовується певна частина балансувальника — бекенд, правила балансування і фронтенд. В свою чергу, створення балансувальника через API вимагає:

1) окремим API-викликом створити правило перевірки здоров’я бекенду;

2) створити бекенд-сервіс, з правилами проксіювання запитів;

3) створити групу серверів з необхідними серверами всередині;

4) створити URL-карту з правилами маршрутизації запитів;

5) створити окремо HTTP та HTTPS проксі для приймання запитів.

Кожен крок вимагає формування спеціально оформленого API-запиту (а інколи і декількох, для отримання ідентифікаторів створених раніше компонентів) з певним набором унікальних параметрів. В свою чергу, веб-інтерфейс наочно відображає потрібні параметри, їх можливі значення та залежності між ними.

Рис. 1.1. Створення балансувальника навантаження у веб-інтерфейсі GCP

Але у ручного підходу є і загальновідомі мінуси — складність слідкування за змінами в інфраструктурі та кількість часу, необхідна на ручне керування, якщо воно відбувається регулярно: при створенні резервних середовищ у інших регіонах, при створенні ідентичних копій інфраструктури у цілях тестування програмної частини продукту розробниками та при підтриманні її у ідентичному стані, або ж при необхідності швидко створювати та знищувати хмарні компоненти для тестування процесу розгортання продукту з нуля.

1.2.2 Автоматичне керування хмарною інфраструктурою.

Як було описано на початку розділу, під автоматичним керуванням інфраструктурою мається на увазі використання вбудованих функцій хмарних сервісів для створення чи знищення хмарних ресурсів на основі попередньо заданих правил. Прикладом таких правил може бути запуск та знищення ресурсів по графіку, або ж в залежності від поточного навантаження на них.

Автоматичне керування підтримують, зазвичай ті хмарні сервіси, в яких немає змогу абстрагувати фізичне обладнання. Наприклад — віртуальні приватні сервери (Virtual Private Server, VPS). Їх часто можна об'єднувати в групи, які в свою чергу підтримують так званий AutoScaling — автоматичне збільшення чи зменшення кількості серверів у групі. Існує багато критеріїв за якими можна визначати оптимальну поточну кількість серверів. У роботі [7] демонструється підхід на основі предиктивної моделі, яка намагається знайти оптимальний баланс між задоволенням вимог продуктивності системи, при цьому мінімізуючи їх вартість. В свою чергу, Google, що використовує можливості AutoScaling їх VPS-сервісу для автоматичного керування кількістю віртуальних серверів, що лежать в основі їх керованого сервісу Kubernetes — GKE, розробив систему «Autopilot» на основі штучного інтелекту, аналізу історичних даних та вручну налаштованих евристик [8].

Крім VPS, автоматично збільшуватись чи зменшуватись можуть такі сервіси як бази даних чи мережеві диски. Але більша частина хмарних сервісів ховають таку потребу від клієнта і можуть обробляти теоретично необмежену кількість даних. Прикладом такого сервісу можуть бути об'єктні сховища даних, такі як AWS S3, чи GCP Storage Bucket.

Очевидною перевагою такого способу керування є автоматизованість — після початкового налаштування інфраструктура керується самостійно і може потребувати лише періодичного контролю та налаштування параметрів для оптимізації продуктивності та вартості.

Основний недолік — прив’язаність AutoScaling до конкретного сервісу і неможливість організації спільної роботи декількох сервісів за допомогою лише AutoScaling. Тому найчастіше він використовується у комбінації з одним із інших типів керування хмарною інфраструктурою.

1.2.3 Програмне керування хмарною інфраструктурою

Усі cloud-провайдери надають клієнтам змогу взаємодіяти з хмарними сервісами за допомогою веб-запитів до API. Завдяки цьому вони можуть створювати власні інструменти для керування інфраструктурою проектів. Зазвичай, напряму програмне керування здійснюється всередині програмного продукту, який і запускається на цій інфраструктурі. Наприклад, головний сервер може звертатись до API і динамічно збільшувати кількість серверів-обробників даних, на основі аналізу інших компонентів продукту — баз даних, мережевих черг, тощо. Наявність програмувального інтерфейсу дозволяє хмарному додатку власноруч контролювати свою інфраструктуру, що стало можливим лише з появою cloud-сервісів.

Проблема використання API полягає у тому, що розробка власного програмного компоненту для взаємодії з ним може потребувати великих затрат часу і сил програмістів, адже необхідно перетворювати вхідні дані у формат, зрозумілий хмарному сервісу, перетворювати відповідь у формат, зрозумілий самому додатку і оброблювати відповідь, коректно реагуючи на будь-які помилки як зі сторони сервера, так і клієнта. Наприклад, у роботі [9] автори пропонують обгортку над хмарними API сервісу Amazon EC2, що реалізує досліджені механізми, які дозволяють уникати довго-хвостової тривалості запитів до API і таким чином значно покращити середню тривалість таких запитів, а отже і загальну продуктивність системи. Тобто, гнучкість — основна перевага використання API напряму — вимагає створення власних програмних прошарків для правильної роботи з помилками чи нестабільною роботою API. Саме тому

пряме використання API у проектах часто є обмеженим і використовується лише у місцях де необхідна гнучкість такого підходу.

Натомість, з часом з’явились готові програмні обгортки над такими API, які дозволили взаємодіяти з, найчастіше, імперативними API декларативно і більш абстраговано. Вони фокусуються на створенні, модифікації та знищенні хмарних ресурсів за вимогою, на відміну від прямої взаємодії з API, яка часто використовується для контролю інфраструктури у реальному часі. Такі інструменти стали основою підходу «Інфраструктура як Код».

1.2.4 Інфраструктура як код

Даний підхід до керування хмарною інфраструктурою характеризується використанням програмних продуктів, що дозволяють декларативно конфігурувати майбутній стан інфраструктури і при запуску створеної конфігурації намагаються привести наявний стан системи у бажаний, роблячи необхідні для цього API запити до cloud-провайдера. Використання IaC-інструментів дає змогу описати бажаний стан системи у вигляді коду, який має всі переваги звичного програмного коду (включаючи можливість тестування [10] та відслідковування змін). Крім цього, часто він є ідемпотентним, тобто запустивши його декілька разів, стан системи ніяк не зміниться. Це дозволяє бути упевненим у тому, що система завжди буде консистентна з тим, як вона описана в коді.

Також до переваг Infrastructure as Code можна віднести:

— наочність — уся інфраструктура проекту описана в одному місці і на відміну від документації, цей опис не застаріває і не має оновлюватись окремо;

— можливість сумісної роботи — використання систем контролю версій дозволяє великим командам разом працювати над тією-ж інфраструктурою, не боячись архітектурних конфліктів та несумісних конфігурацій;

— простота розгортання копій усієї інфраструктури, або її частин — для проведення тимчасових тестів, створення постійних тестувальних полігонів, або міграції між різними обліковими записами того-ж cloud-провайдера.

Оскільки підхід Infrastructure as Code є відносно молодим, його недоліки (крім очевидних) не є добре дослідженими. У роботі [11], де було проведене систематичне дослідження існуючих публікацій на тему інфраструктури як коду (вибірка з 31,498 публікацій), автори зробили висновок, що існує необхідність проведення подальших досліджень, які би вивчили недоліки, дефекти та проблеми з безпекою такого методу керування інфраструктурою.

1.2.5 Інфраструктура, що керується хмарним провайдером (PaaS)

Як було визначено в розділі 1, підхід Platform as a Service існує вже давно — з моменту появи спільних хостингів вебсайтів. У разі використання такої платформи в якості інфраструктури свого проекту, клієнт отримує майже повністю керований провайдером набір хмарних сервісів і може сфокусуватись на створенні програмної частини продукту. Після появи великих cloud-провайдерів, кількість сервісів, що надаються у вигляді PaaS почала стрімко зростати — це і бази даних (як традиційні реляційні, так і NoSQL) і мережеві черги і повноцінні групи серверів, що динамічно підлаштовуються під навантаження. Системною частиною останніх повністю керує провайдер — клієнт повинен лише надати код, який він хоче там запускати і опис середовища, в якому цей код має працювати. Схожа ситуація і з іншими PaaS-сервісами — вони майже ніяк не контролюються клієнтом і виконують зазвичай виконують одну чітко поставлену задачу. Хоча є і щось посередині між PaaS та IaaS — це керовані оркестратори контейнерів, які абстрагують велику частину процесу розгортання та підтримки такого оркестратора, але при цьому надають високий рівень контролю над інфраструктурою, на якій така система працює. Загалом, існує широкий вибір продуктів, які дозволяють обрати підходящий рівень контролю та зручності — від запуску повноцінних власних контейнерів, до запуску конкретних програмних функцій, тому кожен може обрати бажані параметри керування хмарною інфраструктурою свого проекту. Але певна втрата контролю присутня у будь-якому PaaS продукті і дуже часто в процесі розвитку проекти упираються в

обмеження, встановлені такими сервісами і змушені переносити частину, або всю інфраструктуру на альтернативний сервіс, який дає більше контролю. Наприклад, з AWS Fargate, який дозволяє запускати контейнери без керування серверами, на яких вони запускаються на AWS EC2, з повним доступом до віртуального приватного сервера. До того ж, часто PaaS-рішення є несумісними між собою і використання одного з них прив’язує проект до певного провайдера, адже для запуску проекту у іншого провайдера його необхідно частково переробляювати. Використання приватних виділених серверів хоч і може бути дорожчим, у певних випадках, є хорошим компромісом, який дозволяє користуватись більшістю переваг хмарних сервісів, але при цьому зберегти високий рівень контролю над власною інфраструктурою і її сумісність з будь-якими хмарними провайдерами, без прив’язки до одного конкретного.

1.3 Постановка задачі магістерської кваліфікаційної роботи

Отже, розглянувши стан сфери керування інфраструктурою для хмарних обчислень, можна зробити висновок що сфера, на даний момент, стрімко розвивається і постійно з’являються нові підходи та програмні інструменти, які допомагають вирішувати проблеми керування хмарними системами та сервісами. На основі аналізу, проведеного у попередньому розділі роботи, було визначено що серед різних підходів до керування хмарною інфраструктурою, Infrastructure as Code є наймолодшим та замало дослідженим [11]. При цьому, підхід є дуже перспективним, адже дозволяє частково використовувати інші підходи, за необхідності, і при цьому має багато переваг, порівняно та у поєднанні з ними.

Враховуючи вплив підходу Infrastructure as Code на IT-індустрію і його недостатню дослідженість, можна зробити висновок про високу потребу у наукових працях на цю тему.

Для дослідження переваг та недоліків Infrastructure as Code на практиці, необхідно розгорнути простий хмарний додаток-приклад, або знайти наявний OpenSource проект, що матиме усі характеристики, необхідні для ґрунтовної

перевірки методології та конкретних технологій підходу Infrastructure as Code. До таких характеристик можна віднести:

— можливість горизонтального розширення основної частини проекту (шляхом збільшення кількості серверів, а не ресурсів одного сервера);

— необхідність у базі даних;

— необхідність кешування (на рівні веб-сервера, чи окремого кеш-сервера);

— можливість проведення оновлень без зупинки усієї системи (zero-donwtime deployment);

— необхідність запуску періодичних обчислювальних задач;

— використання хмарних об’єктних сховищ;

— використання балансувальника навантаження.

Також, з суто інфраструктурної точки зору, проект повинен мати налаштовані: моніторинг стану серверів, правила фаєрволу та права доступу користувачів.

Задля надійності системи, основні частини її інфраструктури повинні знаходитись у різних географічних зонах та бути максимально незалежними одна від одної.

В результаті роботи повинна бути створена інтелектуальна система керування інфраструктурою для хмарних обчислень, у вигляді конфігураційного коду IaC-інструментів, а також будь-яких розгортувальних чи допоміжних скриптів, програмних налаштувань, документації, тощо. Загальна мета дослідження — змоделювати розгортання та керування хмарною інфраструктурою програмного проекту середньої складності за допомогою підходу Infrastructure as Code, зробити висновки про переваги, недоліки та доцільність IaC у різних ситуаціях і порівняти рішення, виконане на основі підходу IaC з альтернативними, на основі інших підходів.

Висновки до розділу 1

Даний розділ є результатом роботи, виконаної під час проходження переддипломної практики. Вона стала однією з основних частин підготовки до розробки магістерської кваліфікаційної роботи, включивши в себе збір практичних даних про майбутній об'єкт дослідження та оцінку практичних потреб сфери інформаційних технологій, цінність та ефективність тих чи інших підходів, інструментів та методологій при вирішенні реальних проблем. В процесі роботи на підприємстві, були виконані реальні задачі предметної сфери дослідження, сформовано практичне уявлення про неї, визначено чіткий напрям подальших досліджень і тему магістерської кваліфікаційної роботи — «Інтелектуальна система керування інфраструктурою для хмарних обчислень».

Крім цього, був проведений теоретичний аналіз сфери керування інфраструктурою для хмарних обчислень, здійснений огляд способів керування хмарною IT-інфраструктурою та проведений аналіз наявних публікацій на цю тему. Зібрані дані стануть теоретичною основою наступних розділів даної роботи і допоможуть робити обізнані висновки про результати подальших досліджень та експериментів. А набуті під час роботи на підприємстві практичні навички стануть важливим професійним досвідом, допоможуть якісно виконати творчу частину магістерської кваліфікаційної роботи і загалом — розвивати власні професійні знання та навички.

2 ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ ТА СЕРВІСИ — СУБ'ЄКТИ ДОСЛІДЖЕННЯ

В основі інфраструктури більшості сьогоднішніх глобальних програмних продуктів стоїть комплексний набір технологій та сервісів. В епоху мережевої глобалізації, рівень вимог до веб-сервісів постійно зростає і користувачі прагнуть мати якісний доступ до послуг, що надаються сучасними мережевими програмними продуктами, і мати його постійно. Через стрімкий ріст сфери інформаційних технологій, швидкість впровадження нових функцій та покращень у додатки є однією з найбільш затребуваних характеристик продукту, на яку звертають увагу інвестори, керівники та менеджери. Це призвело до появи нових методологій розробки програмного забезпечення, таких як Agile і внутрішніх девізів компаній в дусі «Move fast and break thigs» («Рухайся швидко та ламай речі») компанії Facebook. Ці вимоги до індустрії суперечать одна одній, тому рішенням стало ускладнення архітектури проектів для забезпечення доступності, надійності та стійкості до збоїв у роботі, що відобразилось у зміні вищеописаного девізу на «Move fast with stable infrastructure» («Рухайся швидко зі стабільною інфраструктурою») [12].

Ускладнення архітектури інфраструктурної частини програмних продуктів, для відповідності поставленим вимогам до їх стійкості збільшило обсяги роботи інфраструктурних спеціалістів і спричинило значні зміни у списку технологій, які зазвичай використовуються при розробці сучасних програмних проектів. Для виконання поставленої задачі магістерської кваліфікаційної роботи, необхідно змоделювати такий проект, відповідно до усіх вимог, що покладені на нього сучасними принципами IT-індустрії. Інформаційні технології, що використовуватимуться в ньому можна поділити на 3 категорії:

— провайдери хмарних послуг;

— програмні системи, засоби та компоненти, що використовуються змодельованим проектом;

— інструменти керування хмарною інфраструктурою за принципом Infrastructure as Code.

Перед початком роботи, необхідно розглянути екосистему сервісів та програмних рішень, що можуть використовуватись для моделювання суб'єктів дослідження, задля подальшої їх оцінки, вибору і побудові дослідної моделі програмного продукту у наступних розділах.

2.1 Провайдери хмарних послуг

Хмарними послугами називають надання хмарним провайдером послуг доступу до ресурсів віддалених комп'ютерних систем, особливо сховищ даних та обчислювальних потужностей, без прямого керування ними зі сторони клієнта. Зазвичай, послуги надаються на вимогу, за моделлю «pay-as-you-go», тобто клієнт платить лише за ті ресурси, які використовуються в даний момент, що інколи допомагає зменшити загальну вартість інфраструктури для клієнта, при цьому приносячи прибуток хмарному провайдеру. Це досягається за рахунок одночасного використання фізичних комп'ютерних ресурсів декількома клієнтами, ізольованими один від одного за допомогою технологій віртуалізації, та позитивного ефекту масштабу — зменшення вартості обслуговування фізичної інфраструктури завдяки обсягу її використання.

Провайдерів хмарних послуг можна розділити на три категорії:

— дочірні підприємства великих корпорацій: Amazon Web Services, Microsoft Azure, Google Cloud Platform, Alibaba Cloud, Oracle Cloud, тощо;

— менші компанії, що фокусуються на наданні хмарних послуг: Linode, DigitalOcean, Hetzner, OVH, Vultr, тощо;

— компанії, що фокусуються на наданні готових платформ для запуску клієнтських додатків (PaaS): Heroku, Netlify, Back4app, Vercel, тощо.

Розглянемо кожну з них по черзі.

Хмарні провайдери від великих корпорацій — Amazon, Microsoft, Google, тощо — надають найбільшу кількість послуг, з високим рівнем інтеграції між

ними та і можливістю обирати необхідний рівень контролю над інфраструктурою — від ручного керування віртуальними серверами, до повноцінних PaaS-рішень, причому з будь-якою їх комбінацією в межах одного проекту. До найбільш популярних хмарних сервісів такого типу, за даними Statista [13], відносяться: Amazon Web Services, Microsoft Azure та Google Cloud Platform (рис. 2.1).

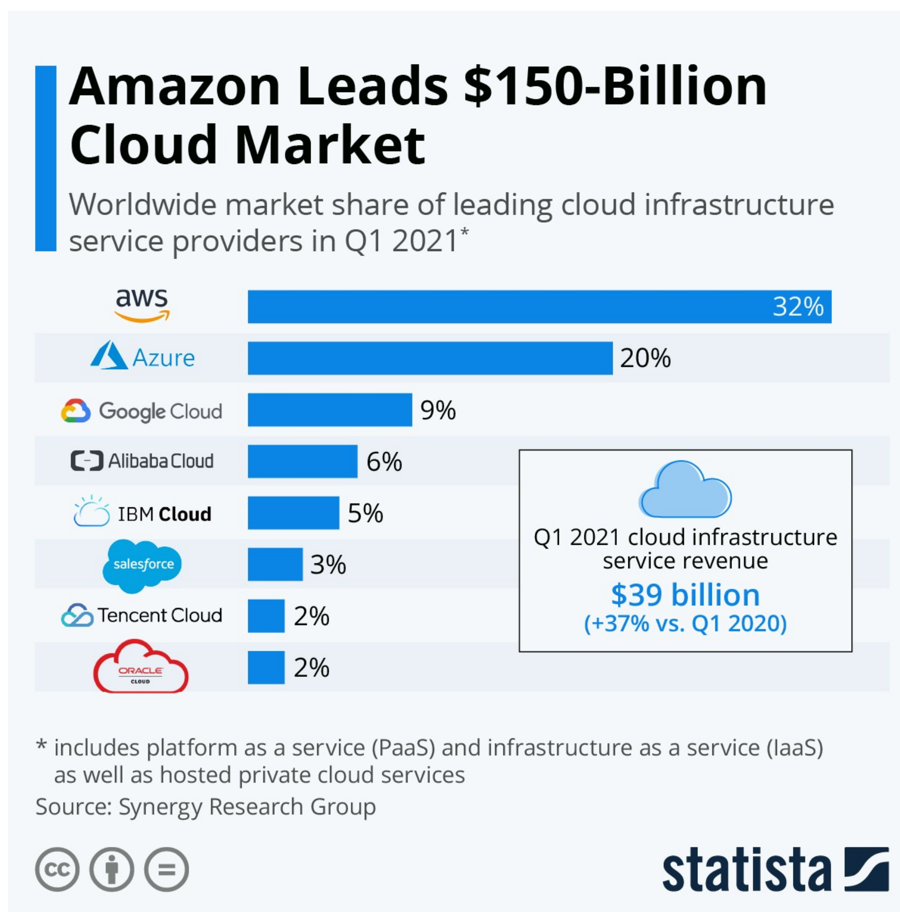


Рис. 2.1. Популярність великих хмарних провайдерів.

Amazon Web Services, Inc. (AWS) — це дочірня компанія Amazon, що надає доступ до хмарних комп’ютерних ресурсів та програмувальні інтерфейси для взаємодії з ними приватним особам, компаніям та урядам держав за описаною вище моделлю pay-as-you-go. До сервісів, що надаються провайдером відноситься велика кількість базової абстрактної технічної інфраструктури та засобів для розподілених обчислень, компонентів та інструментів для побудови власних обчислювальних платформ. Одним із таких сервісів є Amazon Elastic Compute Cloud (EC2), який дозволяє користувачам використовувати віртуальний кластер

обчислювальних машин, який є завжди доступним через мережу Internet. Віртуальні комп'ютери в AWS емулюють більшу частину атрибутів звичайного комп'ютера, включаючи центральний процесор та графічний процесор для обчислень, локальну оперативну пам'ять, твердотілі накопичувачі, або жорсткі диски, мережеве обладнання та попередньо встановлені програмні компоненти, такі як веб-сервери, бази даних і CRM-системи. Загалом, на даний момент AWS надає клієнтам доступ до більш ніж 200 хмарних сервісів [14] у сферах обчислень, зберігання даних, комп'ютерних мереж, баз даних, аналітики, керованого хостингу додатків (ElasticSearch, Kubernetes), розгортання клієнтських додатків, керування, машинного навчання, мобільної розробки, засобів для розробників, RobOps та інструментів для Інтернету речей.

Microsoft Azure (або просто Azure) — це сервіс для хмарних обчислень від Microsoft, використовуючи апаратні ресурси дата-центрів Microsoft. Він надає можливість використання будь-якої з моделей керування хмарною інфраструктурою: Software as a Service, Platform as a Service та Infrastructure as a Service і підтримує велику кількість мов програмування, програмних інструментів та фреймворків, включаючи офіційні та сторонні, для взаємодії з наданими сервісами. Проект було анонсовано у 2008 році, через 3 роки після запуску AWS, та офіційно запущено у 2010, спочатку під назвою Windows Azure. Заявлена підтримка більше ніж 600 хмарних сервісів [15]. До них відносяться:

— обчислювальні сервіси: віртуальні машини (IaaS), програмні платформи (PaaS), хостинг веб-сайтів та виконання фонових обчислювальних задач;

— керування обліковими записами, за допомогою Azure Active Directory;

— мобільні сервіси, включаючи аналітику для мобільних додатків та Visual Studio App Center;

— сервіси зберігання даних, включаючи Storage Services, Table Service (NoSQL база даних), Blob Service (зберігання неструктурованих та бінарних даних), Queue Service (черги для асинхронного виконання задач) та File Service (доступ до файлів у хмарі за допомогою протоколу SMB);

— сервіси комунікації, з підтримкою SMS, відеодзвінків, VOIP, PSTN-дзвінків та веб-чату.

— управління даними: BigData-аналітика, пошук структурованих даних, NoSQL бази даних, керований кеш-сервер, автоматизація трансформації даних, тощо;

— обмін повідомленнями, що дозволяє додаткам які працюють в Azure та поза ним обмінюватись даними, що дає змогу будувати надійні продукти на основі сервісно-орієнтованої архітектури (SOA).

Google Cloud Platform (GCP) — хмарна платформа, що являє собою набір хмарних сервісів, які працюють на тій самій фізичній інфраструктурі, яку Google використовує для власних продуктів, таких як Пошук Google, Gmail, Google Диск, YouTube, тощо. Google заявляє про підтримку більше ніж 100 хмарних сервісів, до яких входять:

— обчислювальні: App Engine, Compute Engine, Google Kubernetes Engine, Cloud Functions, Cloud Run;

— робота з даними: Cloud Storage, Cloud SQL, Cloud BigTable, Cloud Spanner, Cloud Datastore, Persistent Disk, Cloud Memorystore, Local SSD, Filestore;

— мережеві сервіси: VPC, Cloud Load Balancing, Cloud Armor, Cloud CDN, Cloud Interconnect, Cloud DNS;

— великі дані: BigQuery, Cloud Dataflow, Cloud Dataproc, Cloud Composer, Cloud Datalab, Cloud Dataprep, Cloud Pub/Sub, Cloud Data Studio;

— штучний інтелект: Cloud AutoML, Cloud TPU, Cloud Machine Learning Engine, Cloud Natural Language, Cloud Speech-to-Text та Text-to-Speech, Cloud Translation API, Cloud Vision API, Cloud Video Intelligence;

— керування інфраструктурою: Cloud Monitoring, Cloud Logging, Deployment Manager, Cloud Console, Cloud Shell, Cloud APIs;

— облікові записи та безпека: Cloud Identity, Cloud IAM, Cloud IAP, Cloud Key Management Service, Cloud Security Scanner, тощо;

— інтернет речей: Cloud IoT Core, Edge TPU, Cloud IoT Edge;

— API-платформа, з програмним доступом до Google карт та аналітики.

Наступний за популярністю великий хмарний провайдер, Alibaba, орієнтується на ринок КНР і не є значним глобальним провайдером, незважаючи на те що він відповідає за 6% всього ринку хмарних обчислень.

Іншим типом провайдера хмарних послуг є відносно невеликі компанії, що фокусуються саме на наданні хмарних послуг. Основною їх відмінністю від великих хмарних провайдерів є менше різноманіття підтримуваних сервісів, що найчастіше компенсується нижчою ціною послуг. Крім цього дата-центри таких провайдерів знаходяться у меншій кількості локацій і не мають виділених високошвидкісних мережевих ліній, таких як Google Backbone Edge Network. Перевагою використання послуг такого провайдера є можливість використання сервісів багатьох різних компаній, що разом пропонують кращі умови ніж аналогічні сервіси великого хмарного провайдера. Наприклад, комбінація віртуального сервера OVH з кешем та CDN від CloudFlare і об’єктним сховищем B2 від Backblaze може бути набагато вигіднішою ніж набір альтернативних сервісів від AWS чи GCP.

Останнім типом хмарних провайдерів є окремі PaaS-сервіси, найбільш популярним з яких є Heroku. Їх основною перевагою є той факт, що від клієнта вимагається лише написання коду — інфраструктура його проекту буде повністю надана та керована провайдером. Це дозволяє зменшити кількість часу, що витрачається на підтримку інфраструктурної частини програмного продукту і сфокусуватись на розробці. Мінус такого підходу — втрата контролю. Надана провайдером платформа може мати певні вимоги до структури коду, що запускається, певних методів його роботи, способів взаємодії з такими ресурсами як база даних, чи файлове сховище та мати обмежений набір версій інтерпретаторів серверних мов програмування (таких як PHP, Python, Ruby, тощо.), під який доведеться підлаштовуватись під час розробки серверної програми під таку платформу. До того-ж, більшість великих хмарних провайдерів мають власні PaaS-рішення, наприклад AppEngine в Google Cloud Platform. Використання останніх дозволяє запускати тільки певну частину проекту на базі PaaS-рішення, а іншу, яка для цього підходить менше, запускати за допомогою

інших хмарних сервісів, таких як Compute Engine. Незважаючи на це, виділені PaaS-платформи найчастіше мають більш вигідну цінову політику — Heroku, наприклад, надає можливість цілодобово запускати до двох контейнерів повністю безкоштовно, тому для деяких типів проектів такий сервіс може бути найбільш простим та вигідним рішенням.

Загалом, вибір хмарного провайдера сильно залежить від типу розроблюваного проекту та планів на його майбутнє. У наступному розділі буде проведений аналіз вимог експериментального середовища — суб'єкта магістерської кваліфікаційної роботи і обрано той, що дозволить найточніше виконати поставлену задачу.

2.2 Програмні системи, засоби та компоненти для створення суб'єкта основного експерименту

Для повноцінного виконання задачі магістерської кваліфікаційної роботи, необхідно провести аналіз системи, що буде репрезентувати актуальні на сьогоднішній день способи та принципи побудови програмних продуктів, оскільки основним завданням роботи є дослідження важливості та користі принципу керування інфраструктурою «Infrastructure as Code» на практиці. Це означає, що необхідно змоделювати таку архітектуру досліджуваного веб-додатку, яка би відповідала тим вимогам, які на сьогоднішній день є стандартними в індустрії і були сформовані в розділі 1 даної роботи. Враховуючи їх, сформуємо узагальнений набір програмних компонентів, з яких має складатись репрезентативний сучасний веб-додаток:

1) Типізована мова програмування. Судячи з останніх трендів розвитку мов програмування, розробники починають надавати перевагу використанню статичних типів у мовах, замість динамічної типізації, що була популярною у веб-розробці з 90-х років. Швидкий ріст популярності мови Go, поява статичних тайп-чекерів у таких мовах як Python та Ruby та статичних компіляторів для традиційно динамічних мов, таких як TypeScript для JavaScript всі вказують на зсув основної

філософії типізації в сфері веб-програмування в сторону статичної типізації. У випадку даної роботи, мова програмування не грає сильної ролі, адже розроблений додаток буде суто тестовим, але вона все одно впливає на спосіб розгортання та оновлення додатку, що є важливим при плануванні, побудові та керуванні інфраструктурою.

2) Балансувальник навантаження. Задля забезпечення надійності та можливості легкого збільшення продуктивності системи, важливим буде використання декількох серверів у різних зонах доступності. Це означає, що запити користувачів доведеться рівномірно розподіляти по наявних доступних серверах, що вимагає наявності балансувальника. В залежності від обраної хмарної платформи, це може бути як наданий провайдером балансувальник навантаження у якості хмарного сервісу, так і вручну створений сервер-балансувальник, на базі одного з веб-серверів з підтримкою режиму реверс-проксі, таких як Nginx чи Caddy.

3) База даних. Це основа майже будь-якого сучасного веб-додатку. На сьогоднішній день популярності набувають NoSQL бази даних, але традиційні SQL залишаються найбільш популярними і краще підходять для більшості задач, не кажучи навіть про надійність і перевіреність часом. Раціональність використання того чи іншого типу бази даних буде оцінена на етапі вибору конкретних інструментів у наступному розділі, але на основі аналізу, проведеному у попередньому, можна сказати що архітектура база даних повинна бути стійкою до збоїв, отже база даних, як і основний сервер, має знаходитись в декількох зонах доступності. Як і балансувальник навантаження, доцільність використання сервісу наданого хмарним провайдером, такого як RDS чи DynamoDB від AWS, або використання власноруч створеного кластеру на базі віртуальних серверів, буде визначатись в наступному розділі, в залежності від можливостей обраного хмарного провайдеру, та більш детального аналізу доцільності використання керованого провайдером сервісу, на основі вимог до контролю над базою даних та грошової вартості кожного з рішень. Також, на відміну від основного бекенд-серверу, який важливо зробити максимально stateless, тобто не зберігати ніякого

стану на самому сервері, що дозволить зробити легко створювати і знищувати його копії, суть бази даних полягає якраз у зберіганні стану більшої частини проекту, тому створення автоматизованих бекапів дискового сховища бази даних є критично важливим. Для цього може використовуватись як вбудована функція керованої провайдером бази даних, так і об'єктне сховище, або керований вручну зовнішній диск.

4) Кеш-сервер. Кешування це одна з найважливіших технік, що застосовуються для підвищення продуктивності майже будь-якої системи в усіх сферах комп'ютерних наук. Використання кешу дозволяє зменшити загальне навантаження на базу даних, підвищити швидкість роботи додатку з точки зору користувачів і дозволяє ефективно витримувати різкі спалахи навантаження. Є багато різних методів, видів та рівнів кешування, але в даному випадку нас цікавить використання зовнішнього кеш-серверу (щоб усі екземпляри основного сервера використовували спільний кеш), як альтернативі повторювання запитів до бази даних. Як і з іншими компонентами загальної системи, є як можливість використання хмарного сервісу, наданого провайдером, наприклад AWS ElastiCache, так і використання власного кеш-сервера на базі Redis або memcached.

5) Засіб автоматичного керування кількістю активних серверів. Незважаючи на свою ефективність, кешування не може вирішити всіх проблем з продуктивністю програмної системи. Якщо основний сервер виконує певний нетривіальний об'єм обчислень, різке збільшення кількості користувачів може дуже негативно вплинути на якість роботи системи і викликати певні види збоїв. Для вирішення цієї проблеми, використовується так званий AutoScaling, або автоматичне керування кількістю серверів. Якщо сервери є одноманітними, готові працювати одразу після запуску та можуть бути знищені в будь-який момент, AutoScaling дозволяє автоматично створювати нові, сервери при зростанні навантаження, або відмові частини серверів, а також зменшувати кількість серверів, якщо навантаження знизилось, або зламані сервери стали працездатними. Автоматизація цього процесу дозволяє економити кошти шляхом використання лише тих хмарних ресурсів, які є необхідними, а також зробити

продукт більш стійким до неочікуваних подій і є можливою лише в хмарних середовищах. Хоча не всі хмарні провайдери надають готовий сервіс автоматизованого керування кількістю серверів, усі надають програмний доступ до керування хмарними ресурсами, за допомогою API, що дозволяє будувати сторонні інструменти для досягнення такого функціоналу. В цьому разі, використання підходу Infrastructure as Code може допомогти спростити цю задачу, наприклад шляхом задання кількості активних серверів змінною, що визначається при розгортанні інфраструктури. Це дозволить автоматично керувати кількістю активних серверів, навіть якщо хмарний провайдер не має такої вбудованої функції.

6) Моніторинг. Для оцінки статусу та вирішення проблем з продуктивністю серверів, необхідно мати можливість отримувати дані про використання ними системних ресурсів. Важливо мати доступ не лише до даних у реальному часі, а і до історичних даних, щоб мати змогу оцінювати причини виникнення проблем, що відбувались раніше. Для моніторингу, в залежності від необхідності, можна використовувати сервіси хмарного провайдера, окремі програмні продукти для моніторингу кластерів комп’ютерних систем, такі як Zabbix чи Grafana, або ж використовувати індивідуальний моніторинг кожного сервера, за допомогою таких програмних продуктів як Netdata або Cockpit.

7) Git. Система контролю версій на сьогоднішній день є невід’ємною частиною будь-якого програмного проекту, навіть якщо ним займається одна людина. Можливість відслідковування змін, створення різних гілок кодової бази, простого повернення до попередніх версій коду та розподіленої розробки виявились надзвичайно важливими при розробці і мало хто зараз уявляє собі процес розробки без них. Немає сенсу розглядати декілька різних систем контролю версій, адже конкретна система не має значного впливу на результати даної роботи. Враховуючи що найбільш популярним рішенням на даний момент є Git, питання стоїть лише в тому, буде використовуватись власний git-сервер, чи один з багатьох готових продуктів, таких як GitHub, GitLab, Bitbucket або SourceHut.

8) CI/CD (Continuous Integration/Continuous Deployment). CI/CD-системи дозволяють автоматизувати процес інтеграції та розгортки програмних продуктів і забезпечують можливість ефективно розробляти програмні продукти у команді, завдяки автоматичній збірці, тестуванні, пакуванні та розгортці усього нового коду в проекті. Це стосується і коду, що описує інфраструктуру проекту і є однією з головних переваг використання підходу Infrastructure as Code, адже дозволяє автоматизувати тестування та розгортку нових інфраструктурних конфігурацій так само як і основного програмного коду проекту. Наразі існує можливість використання відкритих або закритих комерційних CI/CD-систем, таких як Jenkins, GitLab CI, TeamCity чи Travis (які можуть бути встановлені як на серверах розробників цих систем, так і на власних серверах клієнта), сервісів, що надаються хмарними провайдерами, наприклад Google Cloud Build, або ж повністю керованими сторонніми рішеннями, такими як GitHub Pipelines чи SourceHut Builds.

9) IaC-інструмент. Це основа даної роботи, на основі якої буде проводитись загальне дослідження, визначене в меті та завданні роботи. Детальний огляд можливих інструментів буде здійснено в наступному підрозділі.

Як бачимо, технологічний стек складатиметься з восьми різних компонентів, конкретні варіанти яких необхідно буде вибрати в наступному розділі роботи на основі популярності (задля репрезентативності) та відповідності поставленим вимогам.

2.3 Інструменти керування хмарною інфраструктурою за принципом Infrastructure as Code

Інструменти Infrastructure as Code є наступним етапом еволюції декларативних засобів адміністрації серверів (таких як Puppet, Chef, SaltStack, тощо) в епоху хмарних сервісів. Незважаючи на спроби інструментів минулого покоління адаптуватись під зміни парадигм, що на даний момент відбуваються у сфері IT-інфраструктури, широкого використання здобули саме інструменти

нового покоління, які з самого початку фокусувались на декларативному керуванні хмарною інфраструктурою, а не намагались підігнати наявні принципи керування програмним забезпеченням та конфігурацією фізичних або віртуальних серверів на керування хмарними обчислювальними ресурсами.

Принципи роботи усіх програмних IaC-інструментів є досить схожими — це програми, що зчитують користувацькі файли конфігурації, будують модель бажаного стану системи, порівнюють її з реальним станом і використовуючи API-інтерфейс відповідного хмарного сервісу, змінюють стан інфраструктури на бажаний. Наприклад, якщо в конфігурації указано сервер, якого не існує, інструмент скористається API-запитами для створення сервера з вказаними параметрами. А якщо сервер вже існує, але має іншу конфігурацію, інструмент постарається змінити її і лише при необхідності знищить і створить сервер заново, з потрібними параметрами.

Головні відмінності між інструментами полягають у мові написання конфігураційних файлів та способі керування станом. Оскільки функціонал інструментів напряму залежить від можливостей, які надає API того чи іншого хмарного сервісу, програмний інтерфейс IaC-інструментів найчастіше досить близько відтворює API хмарного провайдера, який він огортає. Це означає що використання того чи іншого IaC-інструменту є схожим на використання іншого і відмінності між хмарними провайдерами є набагато значнішими. Незважаючи на це, кожен з популярних інструментів має певні характеристики, що відрізняють його від інших, тож розглянемо кожен детальніше.

Terraform — інструмент з відкритим початковим кодом, що розробляється компанією HashiCorp, відомою своїми програмними рішеннями у сферах розробки та хмарних технологій. Terraform став одним з перших сторонніх IaC-інструментів, з першим публічним випуском у 2014-році. Взаємодія Terraform з хмарними сервісами забезпечується за допомогою так званих провайдерів, що являють собою логічні абстракції над API відповідного сервісу. На початку Terraform підтримував декілька офіційних провайдерів від самої HashiCorp — для AWS, Azure, GCP, тощо, але в 2017 році було відкрито Terraform Registry —

публічний репозиторій Terraform-провайдерів, який дозволяв будь-кому створювати провайдери для сторонніх чи власних хмарних сервісів. Крім хмарних сервісів, провайдери Terraform можуть керувати великою кількістю інфраструктурних інструментів, таких як Kubernetes чи Consul. Інфраструктура в Terraform описується за допомогою власного конфігураційного формату — HCL (HashiCorp Configuration Language, мова конфігурації HashiCorp), але є опціональна підтримка JSON, в основному для можливості автоматизованого генерування конфігурації та простішої взаємодії з іншими інструментами. Суть роботи Terraform полягає у співвідношенні реального та бажаного станів інфраструктури, побудови графу дій, які необхідно виконати для приведення реального стану інфраструктури в бажаний і, за командою користувача, виконання цих дій, шляхом звернення до хмарного провайдера через API. Реальний стан інфраструктури Terraform визначає за допомогою так званих файлів стану (state), які можуть зберігатись як локально, так і віддалено. Ці файли завжди повинні бути в актуальному стані, для правильної роботи Terraform, інакше інструмент буде намагатись робити зміни в інфраструктурі, якої не існує, чи яка має некоректні, конфліктуючі параметри. Загалом, судячи з кількості наявних провайдерів [16] та пошукових запитів (рис. 2.2), Terraform є одним з найбільш широко підтримуваних та популярних IaC-інструментів.

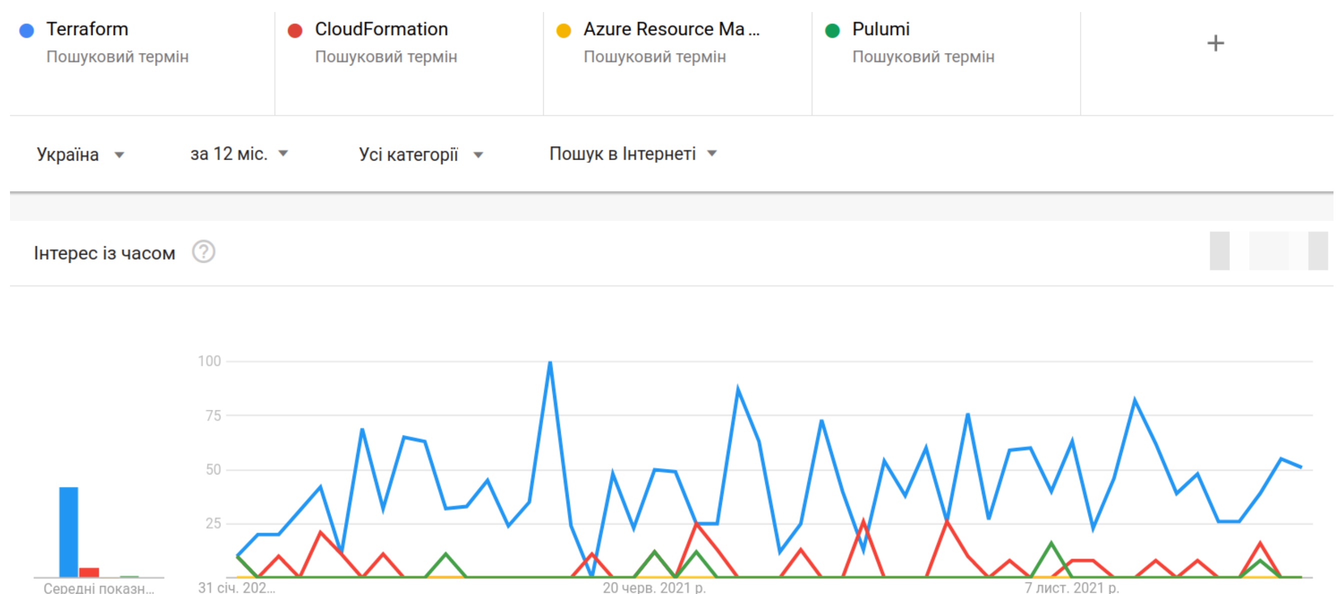


Рис. 2.2. Популярність IaC-інструментів, судячи з кількості пошукових запитів

AWS CloudFormation — керований сервіс, частина набору керованих хмарних сервісів AWS, що надає користувачам можливість управляти AWS-інфраструктурою за принципом Infrastructure as Code. Стан інфраструктури задається у вигляді шаблонів (написаних використовуючи YAML або JSON), які описують стан так званих стеків — груп ресурсів, що мають певні властивості та зв'язки один між одним. Стеки можуть об'єднуватись в групи, що дозволяє керувати ними як одним об'єктом, наприклад створюючи репліки цілої групи ресурсів. Основною особливістю використання AWS CloudFormation є той факт, що на відміну від, наприклад, Terraform, станом інфраструктури повністю керує AWS і не потрібно використовувати мережеві файлові сховища для бекапів, синхронізації та одночасної роботи над станом інфраструктури. До того ж, це прибирає цілий клас проблем, пов'язаних з появою розбіжностей між збереженим і реальним станом інфраструктури. Проте, сервіс є цілком зав'язаним на AWS і не може керувати гетерогенними хмарними середовищами.

AWS CDK — під-сервіс AWS CloudFormation, реалізований у вигляді програмних бібліотек для багатьох популярних мов програмування, які дозволяють генерувати сумісні з CloudFormation файли конфігурації інфраструктури, використовуючи повноцінну мову програмування, замість простих конфігураційних файлів. Використання AWS CDK дозволяє генерувати декларативний стан хмарної інфраструктури за допомогою простого імперативного коду, що дозволяє писати більш складні та гнучкі конфігурації, використовуючи звичний набір інструментів для написання та тестування коду.

Pulumi — сторонній Infrastructure as Code продукт, що поєднує у собі основні переваги Terraform (підтримку багатьох хмарних провайдерів) з підходом AWS CDK — використанні програмних бібліотек для опису стану інфраструктури, тобто дійсно у вигляді коду (відповідно до «Інфраструктура як код»), а не у вигляді конфігураційних файлів HCL, JSON чи YAML. Незважаючи на те що продукт є відносно молодим (версія 1.0 була випущена в 2019 році), він швидко здобув певну популярність, майже на рівні AWS CloudFormation (рис. 3). Завдяки фокусу на розробку програмних бібліотек, Pulumi має зручний у

використанні API, який до того ж є більш потужним та гнучким ніж у інструментів, що базуються на файлах конфігурації, незважаючи на абстрагування однакових API провайдера. Проблему зберігання та синхронізації файлів стану інфраструктури Pulumi вирішує надаючи безкоштовний (для приватних осіб) доступ до керованого компанією хмарного сервісу, що бере на себе обов'язки зберігання та управління станом. При цьому залишається можливість використовувати сторонні сервіси для зберігання стану, такі як сервіси блокових файлових сховищ, наприклад AWS S3.

Крім описаних вище сервісів, існують і інші, з яких можна відмітити GCP Deployment Manager та Azure Resource Manager, які є аналогами AWS CloudFormation від відповідних хмарних провайдерів і працюють схожим чином. Але на відміну від AWS, яка займає третину ринку хмарних послуг, GCP та Azure є значно менш популярними, тому головний недолік використання CloudFormation проявляється ще сильніше. Цим і можна пояснити відносно низьку популярність таких сервісів, судячи з даних Google Trends та наявності веб-ресурсів, що описують роботу з даними сервісами.

Висновки до розділу 2

Даний розділ був присвячений аналізу стану індустрії та пошуку інформаційних технологій та сервісів, які допоможуть ґрунтовно дослідити об'єкт кваліфікаційної магістерської роботи і зробити коректні, релевантні на практиці висновки.

Аналіз було розділено на три основні підрозділи, кожен з яких відповідав за окрему частину набору інформаційних технологій та сервісів, що потенційно можуть використовуватись при написанні дипломної роботи. Перша частина це сервіси, а саме хмарні провайдери, використання яких є основою підходу Infrastructure as Code. В теорії, IaC можна реалізувати і на базі власної реальної інфраструктури, за допомогою таких програмних інструментів як OpenStack чи Kubernetes, але це має сенс лише для дуже великих корпорацій з особливими

вимогами до комп'ютерної інфраструктури. Суть другої частини полягала у визначенні відносно стандартних компонентів сучасного хмарного додатку. І хоча неможливо обрати набір технологій, що буде покривати вимоги абсолютної більшості проектів, було обрано збалансований набір необхідних сервісів, який дозволяє реалізовувати досить складні хмарні проекти і при цьому залишається достатньо гнучким, уникаючи сильних прив'язок до того чи іншого конкретного хмарного сервісу. Для кожного з запропонованих хмарних сервісів була запропонована і альтернатива, якою необхідно керувати власноруч, але яка є незалежною від того чи іншого хмарного провайдера чи хмари загалом. Третя частина фокусувалась на огляді конкретних існуючих IaC-інструментів. Було визначено три основні характеристики таких інструментів — мова конфігурації, способи зберігання та керування описом стану інфраструктури та підтримка декількох хмарних провайдерів. Кожен з описаних інструментів має унікальну комбінацію вказаних характеристик, яка вирізняє його серед інших.

На основі зібраних даних у наступному розділі будуть розглянуті переваги та недоліки кожної з описаних технологій та сформовано загальну модель експериментального середовища, яке буде використано для виконання основної задачі кваліфікаційної магістерської роботи, враховуючи специфічні характеристики того чи іншого інструменту чи сервісу.

3 МОДЕЛЮВАННЯ ЕКСПЕРИМЕНТАЛЬНОГО СЕРЕДОВИЩА ТА СТВОРЮВАНОЇ СИСТЕМИ

Наукове дослідження нових, суто практичних методологій та способів взаємодії з інформаційними технологіями є складним завданням, особливо в технічному контексті, адже об’єкт дослідження в такому випадку — це сутність, велика частина чиїх характеристик є суб’єктивними і сильно залежать від бажань та потреб того чи іншого користувача. Саме тому головним завданням роботи є моделювання типової системи, що часто використовується в індустрії і детальний опис використання досліджуваної методики у порівнянні з альтернативними або більш традиційними. Результатом такого дослідження не буде конкретний вердикт, а скоріше набір висновків, що зможуть стати ґрунтовною основою як подальших наукових досліджень, так і використовуватись при архітектурному плануванні хмарних IT-проектів та прийнятті технічних рішень під час їх розробки та еволюційного розвитку.

В розділі 2 було визначено основні частини системи, яку необхідно побудувати для проведення аналізу і складено списки програмних інструментів та сервісів, придатних для побудови експериментальної системи. В даному розділі будуть обрані конкретні програмні продукти та сервіси і створено фінальну модель системи, що розроблюватиметься в практичному розділі 4.

3.1 Вибір провайдера хмарних послуг

Серед розглянутих у розділі 2.1 хмарних провайдерів було визначено 3 основні категорії таких сервісів. Кожна з них має ряд переваг та недоліків, які варто враховувати при виборі сервісу, який стане фундаментом будь-якого розроблюваного хмарного додатку. Оскільки найбільш важливими вимогами до створюваного експериментального веб-додатку є баланс репрезентативності та висвітлення можливостей підходу «Infrastructure as Code», вибір слід базувати

виходячи саме з них. Розглянемо як загальні плюси та недоліки кожної категорії, так і їх відповідність до двох вищезазначених вимог.

PaaS-сервіси, такі як Heroku, дозволяють швидко будувати хмарні продукти, не витрачаючи багато сил на керування інфраструктурою. Незважаючи на це, сторонні IaC-інструменти, такі як Terraform або Pulumi (через pulumi-terraform-bridge) підтримують [17, 18] такі сервіси, адже навіть прибравши керування псевдо-фізичною інфраструктурою залишається певна кількість хмарних ресурсів, якими необхідно керувати навіть використовуючи PaaS-сервіс. Такими ресурсами (на прикладі Heroku, список неповний) можуть бути:

— додатки (apps);

— домени та сертифікати;

— пайплайни (pipelines);

— віртуальні мережі (spaces);

— збирачі логів (log drains);

— акаунти членів команди.

Але враховуючи що такі сервіси розраховані на відносно невеликі хмарні додатки, використання IaC-інструментів при роботі з такими сервісами є скоріше зручним способом автоматизації, аніж необхідністю. Хоча й існує можливість будувати складні системи, на базі PaaS-сервісів, в процесі еволюційного розвитку продукту часто виникає потреба у більшій кількості контролю над інфраструктурою, яку PaaS-сервіси надати не можуть, адже їх головна пропозиція — це відсутність необхідності такого контролю. До того ж, не беручи до уваги вартість, великі хмарні провайдери часто надають власні PaaS-сервіси, які по зручності не поступаються окремим, але мають перевагу у тому, що вони просто інтегруються з іншими сервісами великого провайдера і дають змогу розширювати продукт, шляхом використання більш вручну керованої інфраструктури, наприклад VPS, для окремих його частин. Враховуючи це, можна зробити висновок що якщо не існує повного бачення проекту одразу (тобто планується що він буде еволюціонувати) і якщо цінова політика обраного PaaS-провайдера не є значно кращою за PaaS-сервіс великого хмарного провайдера,

використання окремого PaaS-сервісу є гіршим варіантом, навіть не враховуючи популярність, з якої випливає кількість літератури, користувацької документації і досвідчених розробників — усі важливі характеристики при виборі будь-якого IT-сервісу.

На протилежній стороні спектру хмарних сервісів знаходяться невеликі хмарні провайдери, такі як Hetzner, Linode чи OVH, які фокусуються на наданні меншої кількості сервісів на біль вигідних умовах, порівняно з великими cloud-провайдерами, такими як AWS, Azure чи GCP. Користувачам таких сервісів доведеться використовувати віртуальні приватні (або виділені) сервери для ручного запуску і керування великою кількістю сервісів, адже кількість керованих провайдером сервісів обмежується одиницями або десятками, а не сотнями, як у великих хмарних провайдерів. Але, як правило, ціна відповідних сервісів є або значно нижчою ніж альтернатива у великого хмарного провайдера, або якість отриманої інфраструктури за ті ж кошти є значно вищою. Наприклад, вартість найменшого виділеного сервера у невеликого провайдера може бути такою ж, як і у віртуального сервера великого провайдера, але при цьому різниця у продуктивності двох серверів буде сильно відрізнятись на користь виділеного сервера. Крім цього, більшість навіть невеликих провайдерів надають декілька найбільш популярних керованих сервісів, таких як балансувальники навантаження та бази даних, тому велика кількість проектів може не помітити різницю у функціоналі, якщо вони не використовують більш специфічний набір керованих сервісів, зате помітити значну різницю у вартості.

Незважаючи на це, вибір великого хмарного провайдера є надійнішим, якщо бюджет проекту це дозволяє. По-перше, такий вибір є більш гнучким і дозволяє легше адаптуватись до нових вимог проекту, що виникають у процесі його розвитку, а також швидше впроваджувати (і випробовувати) зміни, завдяки використанню керованих сервісів (наприклад, можна впровадити новий функціонал на основі керованого хмарним провайдером сервісу і почати поступовий процес переходу на керований власноруч, якщо є така потреба). До того ж, як було визначено у розділі 2, великі хмарні провайдери є значно

популярнішими, що означає кращу наявність документації (у тому числі користувацької) та більшу кількість розробників та інфраструктурних інженерів, що мають досвід роботи з даним провайдером. Оскільки головними вимогами до хмарного сервісу є репрезентативність та висвітлення можливостей підходу «Infrastructure as Code», використання послуг одного з великих хмарних провайдерів має найбільше сенсу, оскільки це повноцінно відповідає обом поставленим вимогам.

Вибір конкретного хмарного сервісу не є складним завданням, оскільки будь-який із них надає доступ до усіх керованих провайдером сервісів, що описані в попередньому розділі і можуть бути необхідними для виконання поставленої задачі кваліфікаційної магістерської роботи. Саме тому має сенс обрати найбільш популярний серед них, який має підтримку усіх IaC-інструментів, описаних у розділі 2.3 — AWS.

Як було визначено у розділі 2.1, ринкова доля AWS складає 32% (рис. 2), і є більшою ніж доля двох наступних найбільших компаній разом узятих. Але не можна сказати що AWS є монополістом, адже її конкуренти надають схожий спектр послуг, а сторонні програмні інструменти для взаємодії з хмарними сервісами підтримують більшість провайдерів однаково добре. З цього можна зробити висновок, що якість послуг, які надає AWS є досить високою, що є важливою характеристикою при оцінці якості методології — висока якість послуг хмарного провайдера дозволить сфокусувати аналіз на особливостях роботи конкретного IaC-інструменту і підходу в цілому, з упевненістю у тому, що досвід їх використання з іншими хмарними провайдерами не буде значно кращим і не змінить загальні результати дослідження.

3.2 Обґрунтування технологій для використання при створенні суб’єкта експерименту

Проведений в розділі 2.2 аналіз визначив 9 програмних інструментів, використання яких є досить типовим для сучасних хмарних додатків, а отже для

репрезентативності основного експерименту даної роботи, необхідно використати повний набір цих інструментів. Обравши в попередньому підрозділі конкретного хмарного провайдера, можна робити вибір конкретних програмних інструментів, враховуючи потреби проекту та специфіку і особливості обраного хмарного провайдера — AWS.

1) Мова програмування. Як було визначено в розділі 2.2, більшість популярних мов програмування активно рухаються в сторону статичної типізації, або є або статично типізованими з моменту своєї появи. До таких мов можна віднести: Go, PHP, Python, Ruby, TypeScript, Java, тощо. Серед наведених мов, дві є популярними динамічними мовами, які широко використовуються при написанні веб-застосунків, але мають значний недолік — Python та Ruby. Для спрощення моделі паралельності, ці мови використовують так званий глобальний засув інтерпретатора, який забезпечує одночасне виконання лише одного паралельного потоку. Але зараз, коли існують системи навіть з 128 процесорними ядрами, використання такого способу синхронізації є дуже неефективним і сильно ускладнює використання цих мов програмування на сучасній апаратурі, адже програми, написані на них, не можуть ефективно використовувати усі наявні ресурси системи. Серверний TypeScript найчастіше використовує NodeJS в якості платформи для виконання скомпільованого JavaScript-коду, а отже теж виконується в одному, хоч і асинхронному, потоці. Альтернативні рантайми, такі як Deno, існують, але поки не набули широкого використання. PHP досі залишається однією з найбільш популярних мов програмування для веб-розробки, але незважаючи на активний процес модернізації в останніх релізах, у мові залишається багато архаїчних на сьогоднішній день конструкцій та принципів, що є однією з основних причин постійного падіння популярності мови в останні роки. Залишаються два варіанти — Go та Java. Вони є багато в чому схожими, адже обидві є статично типізованими мовами зі збиральником сміття (garbage collector). Але при цьому Java використовує віртуальну машину для виконання коду, в той час як Go компілюється в машинний код підтримуваних апаратних платформ. Крім цього, Go з самого початку фокусувалась на веб-розробці і навіть

в стандартній бібліотеці існує більшість програмних компонентів, що необхідні для розробки сучасних серверних застосунків. До того ж, Go є дуже популярною у сфері саме хмарного програмування — на ній написані такі популярні інструменти як Docker, Kubernetes, Terraform, Prometheus, Grafana, тощо. Тому в якості мови для написання експериментального серверного застосунку має сенс вибір Go.

2) Балансувальник навантаження. Для балансування навантаження в середовищі великого хмарного провайдера має сенс використовувати керований ним балансувальник навантаження (ELB, у випадку AWS), адже він краще інтегрується з іншими ресурсами цього провайдера. Наприклад, ELB має змогу швидше реагувати на зміну стану сервера, а також моментально реагувати на зміну публічної IP-адреси сервера, на що не здатен власноруч створений балансувальник, який окремій підмережі. Крім цього, використання ELB дасть змогу отримати безкоштовний SSL сертифікат для створюваного додатку та не хвилюватись про його оновлення та пере-видачу, а також дозволить не використовувати статичну IP адресу для створюваних серверів. Враховуючи що можливостей балансувальника навантаження AWS цілком вистачить для створюваного додатку, немає особливого сенсу обирати інший варіант.

3) База даних. Використання бази даних з підходом «Infrastructure as Code» є непростим завданням, адже важливою частиною підходу є поняття незмінної (immutable) інфраструктури. Це означає що створена інфраструктура не повинна змінюватись, а її оновлення відбувається шляхом знищення старої і створення нової. Це дозволяє запобігти накопиченню незадокументованих змін у системі, що призведе як до неможливості впевнено змінювати інфраструктуру, адже її стан не є цілком відомим, так і до неможливості її точно відтворити, а це є одні з основних задач підходу «Infrastructure as Code». Легко зрозуміти чому база даних не може бути незмінною — її основна задача полягає у збереженні стану, що постійно змінюється. Причому, це стосується не лише даних, а і структури бази, її налаштувань, тощо. Будь-яка зміна бази даних не повинна призводити до її знищення, адже це призведе до втрати усіх збережених даних. Саме тому, при

використанні «інфраструктури як коду», роль бази даних краще виконує база даних, керована хмарним провайдером (у випадку AWS — RDS). Вона дозволяє вносити контрольовані зміни в стан бази даних, без ризику втрати даних, шляхом можливості автоматичного створення реплік, які замінюють основну базу даних поки та оновлюється чи суттєво змінюється. До того ж, використання керованої бази даних дозволить розглянути більше можливостей обраного IaC-інструменту, наприклад створення правил, що не дозволяють знищення хмарного ресурсу ні за яких умов, що є дуже корисним для бази даних.

4) Кеш-сервер. На відміну від бази даних, кеш-сервер, за винятком специфічних задач, дуже добре підходить під визначення «незмінної інфраструктури», адже незважаючи на те що він зберігає певний стан, цей стан є тимчасовим і його втрата призведе лише до тимчасового зниження продуктивності системи (хоча при використанні кластеру кеш-серверів навіть воно буде непомітним). AWS надає керований кеш-сервіс — ElastiCache, але він є втричі дорожчим за аналогічний EC2-сервер, тому враховуючи простоту керування кеш-серверами за допомогою IaC-інструментів, має сенс створення власного кеш-сервера, на базі Redis, або memcached — двох найбільш популярних рішень у цій сфері. Для поставленої задачі цей вибір не має особливого значення, адже можливостей обох інструментів цілком вистачить для її виконання. Виходячи з цього, буде обрано Redis, оскільки він є більш популярним та багатофункціональним (що теоретично може знадобитись в майбутньому).

5) Засіб автоматичного керування кількістю активних серверів. При використанні AWS таким засобом є AutoScaling-групи. Незважаючи на те що існує можливість керувати кількістю серверів програмно, наприклад за допомогою Terraform, автоматичне керування є набагато надійнішим і зручнішим, отже є сенс для даної цілі використовувати саме AWS AutoScaling-групи.

6) Моніторинг. Для невеликої кількості серверів та логів має сенс використовувати сервіс моніторингу хмарного провайдера, адже це простіше та дешевше ніж створення власного моніторинг-сервера. AWS CloudWatch підтримує як числові метрики, так і текстові логи хмарних сервісів (наприклад RDS), а також

власних додатків, що інтегруються з CloudWatch за допомогою API. Крім цього, Cloud Watch надає користувачам можливість створювати правила, при виконанні яких будуть надсилатись повідомлення, що попереджують про аномальні ситуації в проекті.

7) Git. Системи контролю версій є однією з основних причин використання принципу «Infrastructure as Code» для керування інфраструктурою. Можливість відслідковування змін давно довела свою користь у сфері розробки програмного забезпечення і використання IaC-підходу дозволяє отримати усі переваги систем контролю версій при керуванні інфраструктурою. На даний момент існує велика кількість безкоштовних хмарних Git-сервісів, які є зручними у використанні та крім git надають великий спектр суміжних послуг, що включають веб-інтерфейс, керування доступами користувачів, вбудовані CI/CD, тощо. Існують певні типи проектів, для яких важливим є повний контроль над інфраструктурою, включаючи git-сервер, але експериментальний проект, що розроблятиметься в даній роботі до них не відноситься, отже є сенс використовувати один з популярних Git-сервісів, таких як GitHub, GitLab чи SourceHut. Усі вони мають вбудовані CI/CD системи та підтримують функціонал, що необхідний для розробки суб’єкта магістерської кваліфікаційної роботи, тож немає особливої особливих причин обирати той чи інший сервіс. Але найбільш популярним серед них є GitHub, тому для розробки буде використано саме його.

8) CI/CD. Якщо немає специфічних вимог до можливостей CI/CD-системи і обраний Git-сервіс має вбудований CI/CD-сервіс, має сенс використовувати саме його. У випадку GitHub, це GitHub Actions — сервіс, що дозволяє виконувати задані у форматі YAML дії, у відповідь на один з десятків сигналів, таких як push в репозиторій, створення issue, або формування нового релізу розроблюваного додатку. Підтримується також і ручний запуск необхідних дій. Виконання заданих дій відбувається у контейнерах, які обираються під час написання workflow-файлу. Всередині контейнера можна виконувати будь-які команди, що як і будь-яка інша частина workflow-файлу можуть містити шаблонні змінні, що підставляються на етапі запуску дії. Після виконання команд є можливість збереження артефактів,

тобто файлів, які треба зберегти, наприклад для подальшого розгортування. Усі характеристики GitHub Actions відповідають вимогам створюваного експериментального проекту, отже в якості CI/CD будуть використовуватись саме GitHub Actions.

Останнім необхідним компонентом створюваної системи, описаним у розділі 2.2 є IaC-інструмент, вибір якого буде зроблено в наступному розділі, адже це один з основних компонентів, який може сильно вплинути на результати дослідження.

3.3 Вибір інструменту керування хмарною інфраструктурою

У розділі 2.3 було визначено 4 IaC-інструменти, які слід розглянути в якості основного суб'єкта дослідження:

— Terraform;

— AWS CloudFormation;

— AWS CDK;

— Pulumi.

Ці інструменти можна розділити на дві групи по двом різним критеріям:

— Сторонні:
  а) Terraform;
  б) Pulumi;

— Надані хмарним провайдером:
  а) AWS CloudFormation;
  б) AWS CDK.

Та:

— Використовують мову конфігурації:
  а) Terraform (HCL, JSON);
  б) AWS CloudFormation (YAML);

— Використовують мову програмування
  а) AWS CDK (велика кількість популярних мов);

б) Pulumi (велика кількість популярних мов).

Перед вибором конкретного інструменту, важливо визначитись із бажаними варіантами вищезазначених критеріїв.

Насправді, вибір мови конфігурації чи програмування є майже повністю косметичним, адже інструменти, що надають змогу використовувати мову програмування, все одно перетворюють виклики функцій API у конфігураційний формат (найчастіше JSON), адже важливою характеристикою IaC-інструментів є декларативність, якої складно досягти, використовуючи мову програмування, без певної посередньої репрезентації. Для Pulumi та AWS CDK такою репрезентацією є конфігураційні файли, які AWS CDK відправляє напряму в CloudFormation, а Pulumi використовує для побудови графу змін і виконання необхідних API-запитів до хмарного провайдера. Враховуючи це, можна зробити висновок що такі інструменти є лише абстракцією над конфігураційними файлами. Така абстракція може бути корисною в багатьох випадках, наприклад коли конфігурація містить велику кількість логіки або циклів. Особливо, якщо використовується мова конфігурації, не розрахована на це, така як YAML. Але якщо є можливість використання мови конфігурації, що була створена саме з метою конфігурації інфраструктури, такої як HCL, то її використання може бути зручнішим навіть якщо конфігурація містить велику кількість програмної логіки. Саме тому використання засобів, що використовують мови програмування для конфігурації інфраструктури, в їх нинішньому стані, має сенс лише в специфічних ситуаціях, або в дуже великих проектах. Прикладом специфічних ситуацій може бути команда, знайома з мовою програмування і зовсім незнайома з форматом конфігурації. Використання мови програмування в такому випадку пришвидшить процес ознайомлення команди з інструментом і зробить перехід на методику «Infrastructure as Code» максимально простим. Але у випадку даної дипломної роботи, об’єм конфігурації буде відносно невеликим, він не вимагатиме складних конструкцій і на сьогоднішній день конфігураційні файли IaC-інструментів підтримуються текстовими редакторами так само добре як і традиційні мови програмування — з підсвіткою синтаксису, авто-доповненням, форматуванням,

перевіркою на коректність, тощо. Тому є сенс використовувати мови конфігурації для написання інфраструктурного коду, адже вони є значно популярнішими за мови програмування (оскільки за замовчуванням підтримуються саме вони), а отже мають кращу документацію, більшу наявність прикладів, кращу підтримку, тощо.

Враховуючи зроблений вище вибір, залишаються два інструменти, які слід розглянути далі — Terraform та CloudFormation. Перший є стороннім інструментом, який незважаючи на це добре підтримує AWS. Другий — надається у вигляді сервісу самим AWS. Головна відмінність між ними полягає у тому, що Terraform крім AWS підтримує велику кількість інших хмарних сервісів, причому одночасно. Тобто один конфігураційний файл може керувати як ресурсами в AWS, так і в GCP, Linode та Heroku одночасно. Але одночасне керування це не та вимога, що зустрічається дуже часто сама по собі. Набагато частіше необхідно переносити інфраструктуру з одного хмарного сервісу в інший. І хоча Terraform не дає можливості використовувати одну і ту-ж конфігурацію у різних хмарних провайдерів (через несумісність їх API), він дозволяє проводити міграцію поступово, переносячи сервіси з однієї хмари в іншу по одному. Це сильно спрощує процес і робить його більш надійним. Крім того, у випадку необхідності міграції в іншу хмару, розробникам не потрібно знайомитись з новим IaC-інструментом з нуля, достатньо лише написати нову конфігурацію для вже знайомого інструмента. Основним недоліком Terraform, порівняно з CloudFormation, є необхідність власноруч керувати файлами стану інфраструктури. Але в AWS, Terraform може зберігати ці файли в об’єктному сховищі S3 і використовувати DynamoDB-таблицю для синхронізації одночасного доступу до файлів стану, що зробить використання Terraform таким, що мало відрізняється від CloudFormation, не враховуючи наявність веб-інтерфейсу, хоча і для Terraform існують схожі сторонні веб-інтерфейси [19]. А підтримка нового функціоналу AWS в Terraform часто з’являється швидше ніж в CloudFormation. Враховуючи всі вищевказані переваги Terraform, немає особливих причин

використовувати CloudFormation, тому в якості основного «Infrastructure as Code» інструменту даної роботи буде використано саме Terraform.

3.4 Загальна структура та модель створюваної системи

Визначивши список сервісів та програмних інструментів, що будуть використовуватись для побудови експериментального середовища, необхідно розробити модель загальної структури додатку-суб'єкта експерименту і спланувати впровадження методики «Infrastructure as Code» в процес розробки проекту та керування ним.

У цілях демонстрації буде створено веб-застосунок на мові Go, який буде брати певну інформацію з бази даних, або кеш-сервера та відображати її користувачеві у вигляді веб-сторінки. Сам веб-застосунок буде працювати на як мінімум двох різних серверах, які знаходяться в різних зонах доступності, задля підвищення надійності системи. Кількість серверів буде динамічною і керуватиметься можливостями AutoScaling сервісу EC2. Запити до серверів будуть направлятись через Elastic Load Balancer, який також слугуватиме в якості HTTPS-термінатора. Метрики стану серверів будуть зберігатись за допомогою AWS CloudWatch, що дозволить спостерігати за трендами використання додатку, передбачати різкі сплески активності користувачів і аналізувати певні категорії проблем. Створення і керування інфраструктурою буде здійснюватись завдяки розробленій системі керування хмарною інфраструктурою на базі Terraform. Оновлення додатку та інфраструктури будуть здійснюватись завдяки CI/CD-системі GitHub Actions, а код зберігатиметься в головному сервісі GitHub. Схематично, структура застосунку виглядатиме таким чином (рис. 4, 5):

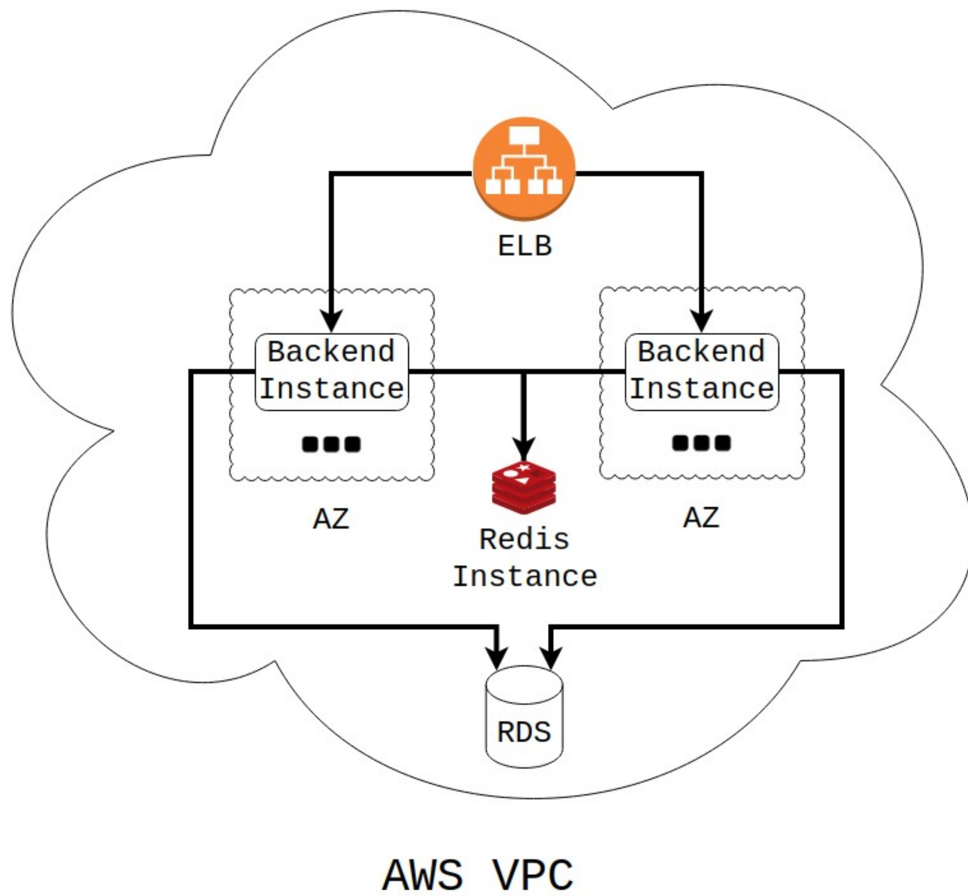


Рис. 3.1. Схема хмарної інфраструктури експериментального веб-застосунку

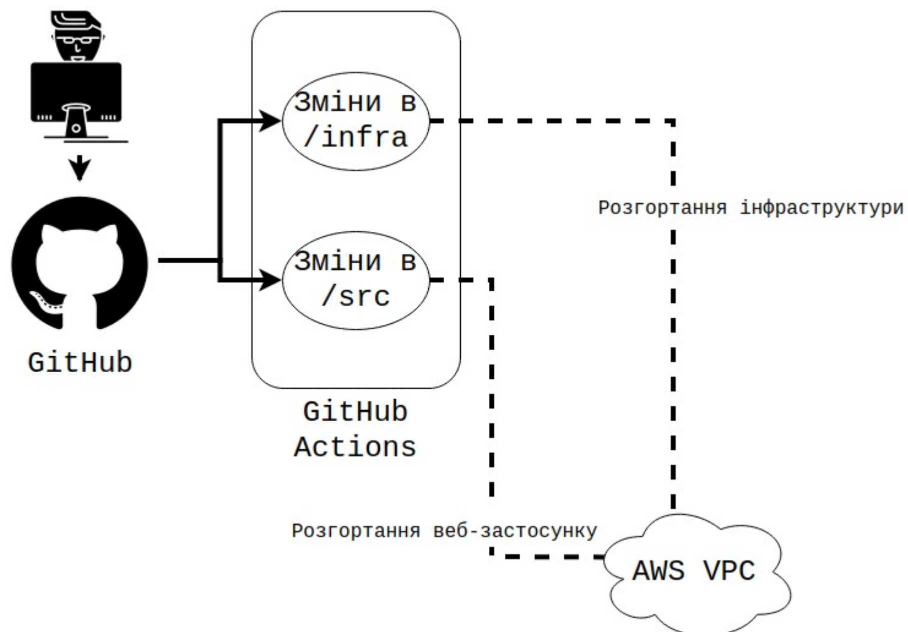


Рис. 3.2. Схема процесу розробки експериментального веб-застосунку

Як бачимо, зміни в GitHub-репозиторії, в піддиректорії /infra повинні викликати GitHub дію, що відповідає за розгортання чи оновлення інфраструктури. Зміни в піддиректорії /src викликають окрему дію, що відповідає за розгортання чи оновлення самого веб-застосунку.

Деталі взаємодії компонентів веб-застосунку між собою та особливості реалізації інтелектуальної системи керування його інфраструктурою на базі Terraform будуть детально висвітлені у наступному, практичному розділі роботи.

Висновки до розділу 3

У даному розділі, на основі даних, зібраних у розділі 2, було проведено аналіз вимог до кожного з компонентів створюваної системи, розглянуто списки можливих інструментів та сервісів і базуючись на критеріях репрезентативності та можливості висвітлення підходу «Infrastructure as Code», було обрано найкращі для даного проекту технології. Найважливішим був вибір інструменту, що дозволить реалізувати підхід «Infrastructure as Code» і на його основі створити ефективну систему керування хмарною інфраструктурою. В результаті порівняння з трьома альтернативами, таким переможцем став OpenSource інструмент Terraform від HashiCorp.

Маючи список конкретних технологій, стала можливим побудова теоретичної моделі роботи усієї експериментальної системи — від моменту написання коду, до появи змін у інфраструктурі і коді веб-застосунку. Ця модель стане основою наступного розділу — практичної частини кваліфікаційної магістерської роботи.

4 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ

На основі моделі, створеної в розділі 3, необхідно розробити експериментальне середовище, на основі якого буде проведений аналіз особливостей використання підходу «Infrastructure as Code» для керування хмарною інфраструктурою та виконане основне завдання магістерської кваліфікаційної роботи.

4.1 Створення початкового експериментального середовища

Необхідність побудови систем керування інфраструктурою для хмарних обчислень випливає з необхідності запускати програмні продукти в хмарі. Керування хмарною інфраструктурою, яка ні для чого не використовується, не показує усіх аспектів обраного для цього підходу, адже архітектура апаратної частини хмарного проекту завжди є тісно пов’язаною з архітектурою його програмної частини. Для отримання реалістичних результатів дослідження, у попередніх розділах було визначено необхідність розробки демонстративного веб-застосунку, для якого буде розроблятись інтелектуальна система керування хмарною інфраструктурою.

В якості такого додатку, відповідно до встановлених раніше вимог, створимо простий сайт, що буде показувати користувачеві інформацію з бази даних, у вигляді, схожому на інфобокси Вікіпедії. Одною з вимог до веб-застосунку є використання хмарного об’єктного сховища для зберігання статичних даних, тож скористаємось AWS S3 для зберігання графічного матеріалу інфобоксів (рис. 4.1).

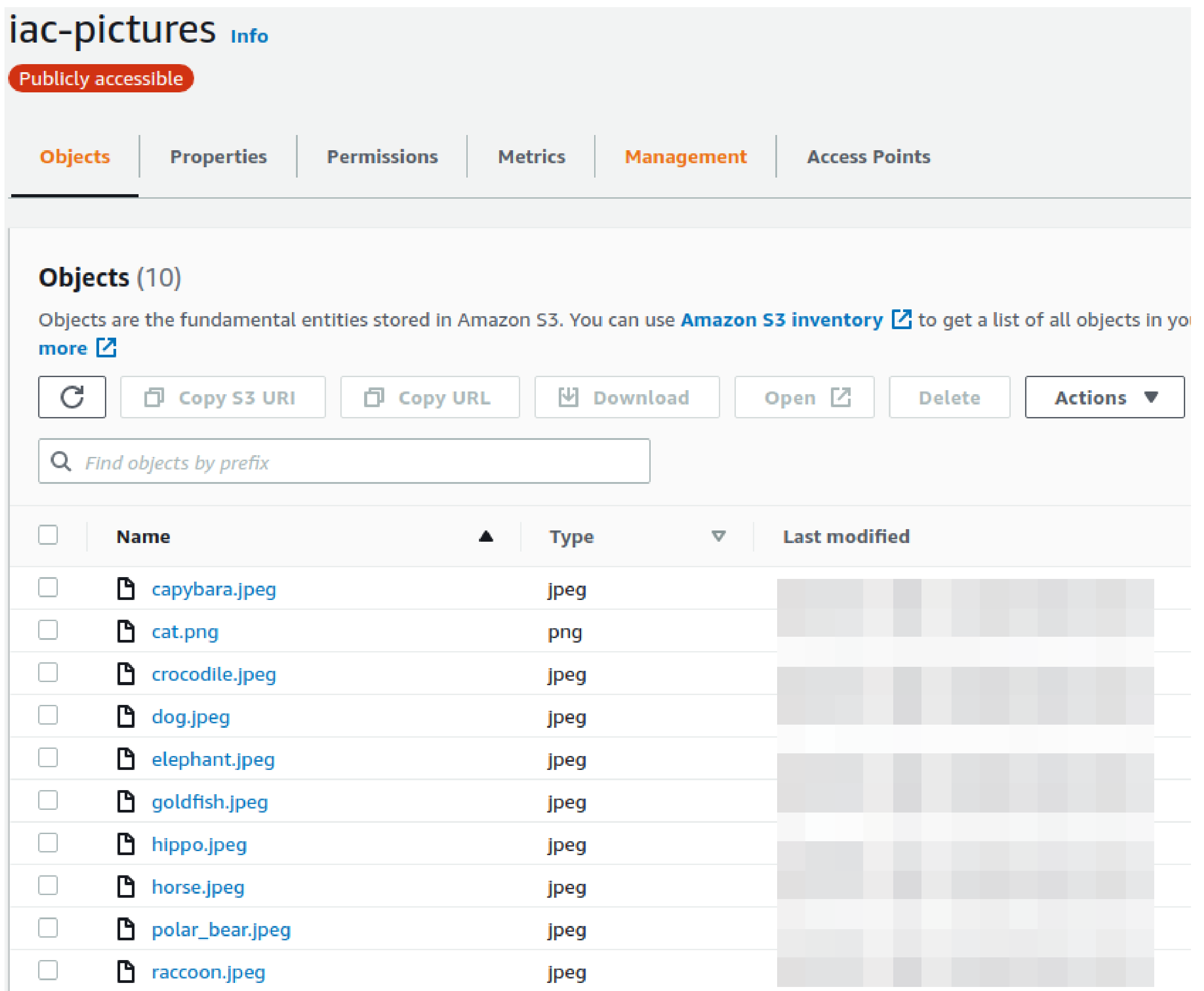


Рис. 4.1. Хмарне об’єктне сховище S3 зі статичними графічними матеріалами майбутнього веб-додатку

Як можна побачити на рисунку, дане сховище S3 позначене як «Publicly accessible». Це необхідно для прямого доступу користувачів до збережених в сховищі зображень і досягається завдяки такій bucket policy:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": "*",
            "Action": "s3:GetObject",
            "Resource": "arn:aws:s3:::iac-pictures/*"
        }
    ]
}
```

Маючи зображення в хмарі, можна створити набір інформаційних даних для інфобоксів, що включатиме в себе посилання на графічний матеріал з S3-сховища. Дані представлені у вигляді такого SQL-скрипту (більшість операторів INSERT пропущені):

```
DROP TABLE IF EXISTS animals;
CREATE TABLE animals (
    id int PRIMARY KEY NOT NULL,
    scientific_name text NOT NULL,
    name text,
    lifespan int,
    diet text,
    domesticated boolean,
    image_url text
);

INSERT INTO animals VALUES (0, 'Hydrochoerus hydrochaeris', 'Capybara', 10,
'Herbivore', false,
'https://iac-pictures.s3.eu-central-1.amazonaws.com/capybara.jpeg');
... ... ...
INSERT INTO animals VALUES (9, 'Procyon lotor', 'Raccoon', 20, 'Omnivore',
false, 'https://iac-pictures.s3.eu-central-1.amazonaws.com/raccoon.jpeg');
```

Маючи усі необхідні дані, можна приступити до розробки самого веб-застосунку. Оскільки йому необхідно буде підключатись до кеш-сервера та бази даних, сервер веб-застосунку повинен мати змогу отримати адреси та облікові дані для доступу до цих сервісів. Для зчитування цих параметрів скористаємось змінними оточення:

```
func getEnv(key, fallback string) string {
      value, ok := os.LookupEnv(key)
      if !ok {
            return fallback
      }
      return value
}

func main() {
      // Read config variables
      listen := getEnv("BACKEND_LISTEN", "127.0.0.1:8080")
      dbUrl := getEnv("BACKEND_POSTGRES",
"postgres://diploma@127.0.0.1:5432/animals")
      redisAddr := getEnv("BACKEND_REDIS", "127.0.0.1:6379")
}
```

Оскільки розроблюваний застосунок використовуватиме паралельні потоки для обробки вхідних запитів, необхідно подбати про синхронізацію майбутніх запитів до бази даних та кеш-сервера, щоб уникнути ситуацій одночасного використання з’єднання двома паралельними потоками. Одним із варіантів

рішення цієї проблеми є використання кожним потоком окремого з’єднання, але це означає створення нового з’єднання при кожному вхідному запиті, що є повільним, неефективним та створює багато зайвого навантаження, особливо на базу даних. А враховуючи що кількість серверів з веб-застосунком буде динамічною, надмірна кількість з’єднань до бази даних може стати проблемою, яку доведеться вирішувати шляхом ускладнення загальної архітектури системи додатковими програмними компонентами, такими як pgpool. Тому, кращим варіантом буде реалізація пулу з’єднань на рівні самого веб-застосунку. Для цього, скористаємось популярними бібліотеками для роботи з PostgreSQL та Redis (майбутньої бази даних та кеш-сервера відповідно) — pgx[20] та redigo[21]. Обидві бібліотеки підтримують створення пулу з’єднань [21, 22], що відповідатиме за демультиплексування з’єднань до бази даних та синхронізацію доступу до вільних з’єднань у пулі, шляхом блокування отримання вільного з’єднання до його звільнення іншим потоком. Для доступу до створених пулів з усіх паралельних потоків, вони будуть оголошені у вигляді глобальних змінних (в даному випадку це є безпечним, адже таке використання підтримується самими пул-об’єктами). Підключення до бази даних та кеш-сервера і створення пулів виглядає таким чином:

```
	var dbPool *pgxpool.Pool
	var redisPool *redis.Pool

	func main() {
		// Create DB connection pool
		var err error
		dbPool, err = pgxpool.Connect(context.Background(), dbUrl)
		if err != nil {
			log.Fatal("Couldn't connect to the database:", err)
		}
		// Create redis connection pool
		redisPool = &redis.Pool{
			MaxIdle:     3,
			IdleTimeout: 60 * time.Second,
			Dial:        func() (redis.Conn, error) { return 
redis.Dial("tcp", redisAddr) },
		}
	}
```

Встановивши підключення до бази даних та кеш-сервера, необхідно відкрити сокет зі слухачем, що буде відповідати на запити користувачів. Застосунок матиме дві основні сторінки — основну, що відображатиме список

наявних інфобоксів і сторінку відображення кожного окремого інфобоксу. Для цього створимо так званий роутер, що буде направляти користувацькі запити до функції, що відповідальна за відображення сторінки, яка цікавить користувача. Після цього, запустимо слухач на адресі, зчитаній раніше зі змінної оточення.

```
http.HandleFunc("/", indexHandler)
http.HandleFunc("/animal/", animalHandler)

fmt.Fprintln(os.Stderr, http.ListenAndServe(listen, nil))
```

Залишилось лише створити відповідні функції — indexHandler та animalHandler. Але спочатку створимо узагальнену функцію повідомлення користувача про помилку (сторінку, якої не існує, або внутрішню помилку сервера, тощо), яку будуть використовувати обидві майбутні функції:

```
func httpError(w http.ResponseWriter, status int, log string, err error) {
	fmt.Fprintln(os.Stderr, "ERROR:", log, "-", err)
	w.WriteHeader(status)
	io.WriteString(w, fmt.Sprintf("<!DOCTYPE html><title>%s</title><h1>
%[1]s   </h1>", http.StatusText(status)))
}
```

Перша функція, indexHandler, відповідає за отримання списку інфобоксів з бази даних (або кешу), кешування отриманого списку, за необхідності, та відображення сторінки користувачеві, за допомогою вбудованої в Go бібліотеки шаблонів (шаблони, для простоти розповсюдження застосунку, вбудовуються в бінарний файл застосунку за допомогою стандартного Go-модуля embed):

```
//go:embed templates/index.html
var indexTemplate string

func indexHandler(w http.ResponseWriter, req *http.Request) {
	var animals []IdName

	// Get Redis connection
	redisConn := redisPool.Get()
	defer redisConn.Close()

	// Try to get cached query response
	values, err := redis.Values(redisConn.Do("HGETALL", "animals"))
	switch {
	case err == nil && len(values) > 0:
		err = redis.ScanSlice(values, &animals)
		if err == nil {
			break
		}
		fmt.Fprintln(os.Stderr, "index: Couldn't parse cache response",
err)
		fallthrough // Do a DB query if failed
```

```
		default:
			if err != nil {
				fmt.Fprintln(os.Stderr, "ERROR: index: Couldn't do a cache
lookup", err)
			}
			// Query animals list
			rows, err := dbPool.Query(context.Background(), "SELECT id,name
FROM animals")
			if err != nil {
				httpError(w, http.StatusInternalServerError, "index:
Couldn't query animals list from DB", err)
				return
			}
			// Parse response rows
			for rows.Next() {
				var animal IdName
				err = rows.Scan(&animal.Id, &animal.Name)
				if err != nil {
					fmt.Fprintln(os.Stderr, "ERROR: index: Couldn't parse
DB response row -", err)
					continue

				}
				animals = append(animals, animal)
				redisConn.Send("HSET", "animals", animal.Id, animal.Name)
			}
			err = redisConn.Flush()
			if err != nil {
				fmt.Fprintln(os.Stderr, "WARNING: index: Couldn't cache DB
response -", err)
			}
		}

		t := template.Must(template.New("index").Parse(indexTemplate))
		t.Execute(w, animals)
	}
```

Друга функція, animalHandler, відповідає за відображення інфобоксу, який цікавить користувача. Для цього вона зчитує ідентифікатор інфобоксу з запиту користувача, шукає його в кеші, або базі даних і відображає за допомогою іншого шаблону. Шаблони виглядають дуже просто, оскільки додаток створюється лише для демонстрації і будуть наведені пізніше, в додатках.

```
	//go:embed templates/animal.html
	var animalTemplate string

	func animalHandler(w http.ResponseWriter, req *http.Request) {
		var animal Animal
		// Get requested id
		id, err := strconv.ParseInt(req.URL.Path[len("/animal/"):], 10, 32)
		if err != nil {
			httpError(w, http.StatusForbidden, fmt.Sprint("animal: Couldn't
parse id", id), err)
			return
		}
		// Get Redis connection
		redisConn := redisPool.Get()
		defer redisConn.Close()
```

```
		// Try to get cached query response
		values, err := redis.Values(redisConn.Do("HGETALL",
fmt.Sprint("animal:", id)))
		switch {
		case err == nil && len(values) > 0:
			err = redis.ScanStruct(values, &animal)
			if err == nil {
				break
			}
			fmt.Fprintln(os.Stderr, "animal: Couldn't parse cache response
-", err)
			fallthrough
		default:
			// Query animal
			err = dbPool.QueryRow(context.Background(), "SELECT
scientific_name,name,lifespan,diet,domesticated,image_url FROM animals WHERE
id=$1", id).Scan(
				&animal.ScientificName, &animal.Name, &animal.Lifespan,
&animal.Diet, &animal.Domesticated, &animal.ImageUrl,
			)
			if err != nil {
				httpError(w, http.StatusForbidden, fmt.Sprint("animal:
Couldn't find animal", id), err)
				return
			}
			_, err = redisConn.Do("HSET", fmt.Sprint("animal:", id),
				"ScientificName", animal.ScientificName,
				"Name", animal.Name,
				"Lifespan", animal.Lifespan,
				"Diet", animal.Diet,
				"Domesticated", animal.Domesticated,
				"ImageUrl", animal.ImageUrl,
			)
			if err != nil {
				fmt.Fprintln(os.Stderr, "WARNING: animal: Couldn't cache DB
response -", err)
			}
		}
		t := template.Must(template.New("animal").Parse(animalTemplate))
		t.Execute(w, animal)
	}
```

В результаті, маємо простий веб-застосунок, головна сторінка якого виглядає таким чином (рис. 4.2):

- Capybara
- Cat
- Crocodile
- Dog
- African elephant
- Goldfish
- Hippopotamus
- Horse
- Polar bear
- Raccoon

Рис. 4.2. Головна сторінка демонстративного веб-застосунку

Клацнувши на один із варіантів, можемо побачити інфобокс, з інформацією, що взята з бази даних та зображенням, що зберігається в S3 (рис. 4.3):

Scientific Name: Hydrochoerus hydrochaeris
Common Name: Capybara
Lifespan: 10
Diet: Herbivore
Domesticated: No

Рис. 4.3. Варіант інфобоксу

Розроблений застосунок працює відповідно до моделі з розділу 3.4. Він стане основним суб’єктом інфраструктури, що керуватиметься розроблюваною в наступному підрозділі інтелектуальною системою.

4.2 Впровадження принципу Infrastructure as Code

На даний момент, для розгортання розробленого в попередньому розділі застосунку у хмарі необхідно вручну:

1. Створити віртуальну приватну мережу (VPC) у AWS.
2. Створити мінімум дві підмережі у різних зонах доступності.

3. Створити інтернет-шлюз для доступу в приватну мережу з мережі Інтернет.

4. Створити таблицю маршрутизації, що направлятиме зовнішні запити в інтернет, а внутрішні — всередині мережі.

5. Створити базу даних в одній з підмереж зі standby-реплікою в іншій підмережі та правилами security-групи, які не дозволяють доступ до бази даних ззовні.

6. Створити кеш-сервер у вигляді EC2-інстансу, з необхідними метаданими user-data, які налаштовують redis-сервер на інстансі під час його запуску. Крім цього, необхідно створити security-групу для інстансу, ідентичну такій у бази даних, або зробити її спільною.

7. Створити launch-конфігурацію бекенд-сервера та security-групу для цих серверів, а також включити до конфігурації user-data, що буде налаштовувати сервер при запуску, завантаживши останню реліз-версію веб-застосунку та запустивши її на сервері.

8. Створити AutoScaling-групу на базі створеної раніше launch-конфігурації.

9. Створити балансувальник навантаження, що розподілятиме запити між серверами з AutoScaling-групи завдяки створеній target-групу, що автоматично включатиме в себе усі робочі сервери з AutoScaling-групи.

Це досить трудомісткий процес, який спрацює лише один раз — у випадку міграції в інший регіон або навіть обліковий запис AWS, всю роботу доведеться виконувати знову, з нуля. Також, після її виконання не залишиться жодної документації про те, що необхідно зробити для відтворення такого середовища. Це стосується і будь-яких майбутніх змін в налаштуваннях тих чи інших компонентів. Тож, замість того щоб робити це вручну, скористаємось інструментом «Infrastructure as Code» — Terraform. Для цього на мові HCL напишемо необхідну конфігурацію. Вона буде розподілена на три файли — data.tf, variables.tf та main.tf, які відповідатимуть за отримання інформації від хмарного провайдера, отримання значення змінних від користувача та створення вищеописаної інфраструктури, відповідно.

До необхідних даних можна віднести базовий образ диску для створюваних EC2-інстансів (необхідно отримати у AWS ідентифікатор останньої версії образу необхідної ОС) та регіон, в якому необхідно розгорнути інфраструктуру (що задається за допомогою змінної оточення і не буде просто так доступним у Terraform-конфігурації):

```
data "aws_region" "current" {}

data "aws_ami" "debian-10" {
    most_recent = true
    owners = ["136693071363"]
    filter {
        name = "name"
        values = ["debian-10-amd64-*"]
    }
    filter {
        name = "virtualization-type"
        values = ["hvm"]
    }
}
```

До необхідних змінних відносяться користувач та пароль бази даних, назва SSH-ключа для адміністративного доступу на сервери та GitHub-токен для завантаження останнього релізу додатку з GitHub.

```
variable "db_user" {
    type = string
    description = "Default database user"
    sensitive = true
}

variable "db_pass" {
    type = string
    description = "Default database password"
    sensitive = true
}

variable "ec2_key_name" {
    type = string
    description = "EC2 SSH key"
}

variable "github_token" {
    type = string
    description = "GitHub API token"
    sensitive = true
}
```

Маючи усі необхідні вхідні дані, можна розпочати написання інфраструктурного коду. Першим компонентом створюваної інфраструктури має стати VPC з двома підмережами, інтернет-шлюзом та таблицею маршрутизації:

```
// VPC
resource "aws_vpc" "iac" {
    cidr_block = "172.28.0.0/16"
    tags = {
        Name = "iac"
    }
}

// Subnets
resource "aws_subnet" "iac-1" {
    vpc_id = aws_vpc.iac.id
    cidr_block = "172.28.1.0/24"
    availability_zone = "${data.aws_region.current.name}a"
    tags = {
        Name = "iac-1"
    }
}

resource "aws_subnet" "iac-2" {
    vpc_id = aws_vpc.iac.id
    cidr_block = "172.28.2.0/24"
    availability_zone = "${data.aws_region.current.name}b"
    tags = {
        Name = "iac-2"
    }
}

// Internet gateway
resource "aws_internet_gateway" "iac" {
  vpc_id = aws_vpc.iac.id
  tags = {
    Name = "iac"
  }
}

// Routing table
resource "aws_route_table" "iac" {
    vpc_id = aws_vpc.iac.id
    route {
        cidr_block = "0.0.0.0/0"
        gateway_id = aws_internet_gateway.iac.id
    }
    tags = {
        Name = "iac"
    }
}
resource "aws_route_table_association" "iac-1" {
    subnet_id = aws_subnet.iac-1.id
    route_table_id = aws_route_table.iac.id
}
resource "aws_route_table_association" "iac-2" {
    subnet_id = aws_subnet.iac-2.id
    route_table_id = aws_route_table.iac.id
}
```

Далі, необхідно створити security-групи для контролю доступу до хмарних ресурсів — бекенд-серверів, бази даних, кеш-сервера та балансувальника навантаження:

```
// Security groups
resource "aws_security_group" "backend" {
    name = "allow-backend"
    vpc_id = aws_vpc.iac.id
    ingress {
        from_port = 22
        to_port = 22
        protocol = "tcp"
        cidr_blocks = ["0.0.0.0/0"]
    }
    ingress {
        from_port = 8080
        to_port = 8080
        protocol = "tcp"
        cidr_blocks = ["0.0.0.0/0"]
    }
    ingress {
        from_port = 22
        to_port = 22
        protocol = "tcp"
        cidr_blocks = ["0.0.0.0/0"]
    }
    egress {
        from_port = 0
        to_port = 0
        protocol = "-1"
        cidr_blocks = ["0.0.0.0/0"]
    }
    tags = {
        Name = "backend"
    }
}

resource "aws_security_group" "iac-db" {
    name = "iac-db"
    vpc_id = aws_vpc.iac.id
    ingress {
        from_port = 5432
        to_port = 5432
        protocol = "tcp"
        security_groups = [aws_security_group.backend.id]
    }
}

resource "aws_security_group" "iac-cache" {
    name = "iac-cache"
    vpc_id = aws_vpc.iac.id
    ingress {
        from_port = 6379
        to_port = 6379
        protocol = "tcp"
        security_groups = [aws_security_group.backend.id]
    }
    egress {
        from_port = 0
        to_port = 0
        protocol = "-1"
        cidr_blocks = ["172.28.0.0/16"]
    }
}

resource "aws_security_group" "iac-lb" {
    name = "iac-lb"
    vpc_id = aws_vpc.iac.id
```

```
    ingress {
        from_port = 80
        to_port = 80
        protocol = "tcp"
        cidr_blocks = ["0.0.0.0/0"]
    }
    egress {
        from_port = 0
        to_port = 0
        protocol = "-1"
        cidr_blocks = ["0.0.0.0/0"]
    }
}
```

Створення бази даних та кеш-сервера виглядає таким чином:

```
// RDS subnet group
resource "aws_db_subnet_group" "iac" {
    name = "iac"
    subnet_ids = [aws_subnet.iac-1.id, aws_subnet.iac-2.id]
    tags = {
        Name = "iac-db"
    }
}

// RDS database
resource "aws_db_instance" "iac-db" {
    engine = "postgres"
    engine_version = "14.1"
    instance_class = "db.t4g.micro"
    identifier = "iac-db"
    storage_type = "standard"
    allocated_storage = 5
    db_subnet_group_name = aws_db_subnet_group.iac.id
    vpc_security_group_ids = [aws_security_group.iac-db.id]
    username = var.db_user
    password = var.db_pass
    name = "animals"
    multi_az = false // optional
    apply_immediately = true
    skip_final_snapshot = true
}

// Redis cache instance
resource "aws_network_interface" "iac-cache" {
    subnet_id = aws_subnet.iac-1.id
    private_ips = ["172.28.1.200"]
    security_groups = [aws_security_group.iac-cache.id]
}

resource "aws_instance" "iac-cache" {
    ami = data.aws_ami.debian-10.id
    instance_type = "t2.nano"
    key_name = var.ec2_key_name
    network_interface {
        network_interface_id = aws_network_interface.iac-cache.id
        device_index = 0
    }
    user_data = <<EODATA
#!/bin/bash -x
apt-get update && apt-get upgrade -y
```

```
      apt-get install -y redis-server
      sed -i -e '/^bind/s/$/ 172.28.0.0/' -e '/^supervised/s/no/systemd/'
/etc/redis/redis.conf
      systemctl is-enabled redis-server || systemctl enable --now redis-server
      EODATA
          tags = {
              Name = "iac-redis"
          }
      }
```

Після цього, залишається лише створити launch-конфігурацію бекенд-інстансу EC2, балансувальник навантаження та AutoScaling-групу, що використовуватиме ці компоненти для автоматичного керування кількістю серверів:

```
      // Backend launch template
      resource "aws_launch_configuration" "iac-backend" {
          name_prefix = "iac-backend"
          image_id = data.aws_ami.debian-10.id
          instance_type = "t2.nano"
          security_groups = [aws_security_group.backend.id]
          associate_public_ip_address = true
          key_name = var.ec2_key_name
          lifecycle {
              create_before_destroy = true
          }
          user_data = <<EODATA
      #!/bin/bash -x
      id backend &>/dev/null || useradd backend
      mkdir -p /etc/backend
      printf 'BACKEND_POSTGRES=%s\n' 'postgres://${var.db_user}:${var.db_pass}@$
{aws_db_instance.iac-db.endpoint}/${aws_db_instance.iac-db.name}' >
/etc/backend/environment
      url=$(curl -sLH 'Authorization: token ${var.github_token}'
'https://api.github.com/repos/AdamantGarth/masters-diploma/releases/latest' | awk
-F'"' 'NR==1,/"assets":/{next} /"url":/{print $4;exit}')
      curl -sLH 'Authorization: token ${var.github_token}' -H 'Accept:
application/octet-stream' "$url" -o backend.tar
      tar -xf backend.tar -C /
      systemctl is-enabled backend || systemctl enable --now backend
      EODATA
      }

      // Load Balancer
      resource "aws_lb" "iac" {
          name = "iac"
          internal = false
          load_balancer_type = "application"
          security_groups = [aws_security_group.iac-lb.id]
          subnets = [aws_subnet.iac-1.id, aws_subnet.iac-2.id]
      }

      resource "aws_lb_target_group" "iac-backend" {
        name     = "iac-backend"
        port     = 8080
        protocol = "HTTP"
        vpc_id   = aws_vpc.iac.id
      }
```

```
resource "aws_lb_listener" "iac-backend" {
    load_balancer_arn = aws_lb.iac.arn
    port              = "80"
    protocol          = "HTTP"
    default_action {
      type = "forward"
      target_group_arn = aws_lb_target_group.iac-backend.arn
    }
}


// Backend AutoScaling group
resource "aws_autoscaling_group" "iac-backend" {
    name = "iac-backend"
    launch_configuration = aws_launch_configuration.iac-backend.id
    min_size = 1
    desired_capacity = 1
    max_size = 2
    vpc_zone_identifier = [aws_subnet.iac-1.id, aws_subnet.iac-2.id]
    target_group_arns = [aws_lb_target_group.iac-backend.id]

    health_check_grace_period = 60
    default_cooldown = 120
    lifecycle {
        create_before_destroy = true
    }

    depends_on = [aws_instance.iac-cache]
}
```

Після цього, інфраструктура для запуску створеного веб-застосунку буде повністю виражена у вигляді коду. Це дозволяє додати її в Git-репозиторій та відслідковувати зміни в ній так само, як і в коді самого веб-застосунку. Також, однією з переваг використання підходу «Insfrastructure as Code» є можливість використання CI/CD-практик по відношенню до хмарної інфраструктури. Реалізуємо таку можливість за допомогою GitHub Actions (і додатково, автоматизуємо компіляцію веб-застосунку за допомогою окремої дії).

Для перевірки коректності інфраструктурного коду, тобто працездатності створеної системи керування хмарною інфраструктурою, створимо таку конфігурацію для GitHub Actions:

```
name: Plan infrastructure changes
on: [workflow_dispatch]
jobs:
  plan:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v2
    - uses: hashicorp/setup-terraform@v1
    - run: pushd infra && terrafrom plan && popd
      env:
        AWS_DEFAULT_REGION: ${{ secrets.AWS_DEFAULT_REGION }}
```

```
      AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
      AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
      TF_VAR_db_user: ${{ secrets.TF_VAR_db_user }}
      TF_VAR_db_pass: ${{ secrets.TF_VAR_db_pass }}
      TF_VAR_ec2_key_name: ${{ secrets.TF_VAR_ec2_key_name }}
      TF_VAR_github_token: ${{ secrets.TF_VAR_github_token }}
```

Її можна запускати, наприклад, при кожному новому при кожному pull request, що дозволить перевіряти коректність нового коду ще до його потрапляння в основні гілки. А після прийняття коду, є можливість запускати аналогічну дію, що виконуватиме terrafrom apply, замість plan, хоча оскільки автоматизація такого процесу це може спричинити пошкодження інфраструктури, так автоматичне виконання змін необхідно робити тільки при наявності надійної системи автоматичного відновлення інфраструктури та встановлених заборонах на знищення критичних інфраструктурних об’єктів.

Автоматизація збірки проекту реалізовується такою дією:

```
name: Build backend release
on: [workflow_dispatch]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v2
    - uses: actions/setup-go@v2
      with:
        go-version: '^1.17.6'
    - run: pushd src && go build -o ../backend . && popd
    - uses: actions/upload-artifact@v2
      with:
        name: backend
        path: backend
```

Вона відповідає за збірку останньої версії коду проекту та архівацію результату (єдиного бінарного файлу), який можна використати для створення нового релізу та оновлення коду на серверах, шляхом інвалідації усіх наявних у AutoScaling-групі серверів та створення нових, що отримають останню версію коду при створенні (завдяки user-data). Це відповідає принципу «Immutable Infrastructure» [23] та дозволяє уникнути так званого «configuration drift» [24] — ситуації, коли стан системи постійно потрохи змінюється в процесі оновлення програмного забезпечення, або змін в конфігурації і через певний час стає неможливим її точне відтворення. А оскільки при такому способі розгортання оновлень застосунку сервери постійно створюються заново, можна завжди бути

впевненим у тому, що наявний процес розгортання веб-застосунку є робочим, коректним та вичерпним.

В результаті виконання написаної конфігурації за допомогою Terraform, маємо робочу систему, що відповідає розробленій у розділі 3.4 моделі. При виконанні команди terraform plan, Terraform оцінює стан хмарного середовища та створює план змін, які до нього необхідно внести, щоб синхронізувати наявний та бажаний стани хмарної інфраструктури. Якщо оператор погоджується зі змінами, команда terraform apply виконає необхідні запити до API AWS та змінить стан інфраструктури згідно зі створеним планом. У даному випадку, список усіх створених ресурсів описаний у розділі раніше, але у якості прикладу, розглянемо вигляд у веб-інтерфейсі створених за допомогою Terraform EC2 інстансів (рис. 4.4) та балансувальника навантження (рис. 4.5)

Instances (2) Info

Instance state = running | Clear filters

Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone
iac-redis	i-0d6eb5e839f7e8b02	Running	t2.nano	2/2 checks passed	No alarms	eu-central-1a
–	i-0316e027631658286	Running	t2.nano	2/2 checks passed	No alarms	eu-central-1b

Рис. 4.4. EC2 інстанси створені за допомогою Terraform

Create Load Balancer | Actions

Name	DNS name	State	VPC ID	Availability Zones	Type
iac	iac-503144673.eu-central-...	Active	vpc-07d6f0d691021e7f8	eu-central-1a, eu-cen...	application

Рис. 4.5. Балансувальник навантаження, створений за допомогою Terraform

Після запуску системи та перевірки її працездатності, Terraform дозволяє однією командою знищити усі створені хмарні ресурси, якщо вони більше непотрібні. Виконавши terraform destroy, знищуємо усі ресурси, оскільки вони більше непотрібні. А в разі необхідності, їх можна буде автоматично створити з нуля за декілька хвилин.

4.3 Аналіз результатів

Основним завданням магістерської кваліфікаційної роботи було дослідження особливостей підходу до керування хмарною інфраструктурою «Infrastructure as Code». У попередньому підрозділі, було створено інтелектуальну систему керування інфраструктурою демонстраційного проекту (див. розділ 4.1) і на основі досвіду її розробки та використання можна зробити висновки про загальну ефективність підходу, що розглядається.

Відразу можна сказати, що переваги підходу, які розглядались у розділах 1.2.4 та 2.3 виявились цілком коректними і не було помічено значних розбіжностей практичного використання IaC-технологій з їх теоретичним аналізом. Використаний в роботі IaC-інструмент Terraform у середині 2021 року здійснив перший реліз з версією 1.0, засвідчуючи про стабільність та придатність його використання у production-середовищах [25] і під час його використання у роботі не було помічено жодних помилок, проблем чи некоректної поведінки. Це свідчить про те, що IaC-технології є достатньо зрілими для використання у більшості хмарних проектів, яким необхідні переваги, надані таким підходом. Проте, актуальність даної роботи полягає у недостатній дослідженості саме недоліків та проблем з безпекою при використанні підходу «Infrastructure as Code», отже сфокусуємо аналіз саме на них.

1. Використання підходу «Infrastructure as Code» у багатьох випадках потребує його впровадження в наявні хмарні системи та навчання персоналу для його використання. Велика кількість наявних проектів користуються іншими способами керування інфраструктурою для хмарних обчислень і внесення змін у налаштовані роками процеси для багатьох проектів потребує великої кількості часу та кваліфікованого персоналу, яких може не бути. Проте, цей недолік компенсується підтримкою IaC-інструментами режиму імпорту наявних хмарних ресурсів. Це означає, що є можливість поступового впровадження підходу IaC, навіть якщо компоненти наявної системи тісно між собою зв’язані. Імпорт наявних ресурсів не вимагає внесення до них жодних змін, але дозволяє описати

їх стан (з допомогою самого IaC-інструменту) у вигляді конфігураційного коду і в подальшому вносити контрольовані зміни в систему, які будуть відслідковуватись у системі контролю версій та дозволять точніше оцінювати стан системи. Таким чином, можна поступово впровадити підхід «Infrastructure as Code» в наявну хмарну систему, без будь-якого негативного впливу на її роботу, якщо переваги такого підходу визнаються вартими .

2. Навіть якщо проект є новим і створення інфраструктури для нього у вигляді коду не є проблемою, у таких проектах одною з найважливіших характеристик є швидкість ітерації. Розробникам необхідно вносити велику кількість з мін в проект максимально швидко. Використання інфраструктури як коду, вимагає автоматизації більшості процесів при розгортанні програмного продукту, а автоматизація найчастіше займає більше часу, ніж виконання тої чи іншої операції вручну невелику кількість разів. А одночасне використання принципу «Infrastructure as Code» та ручне внесення швидких змін, призведе до помилок, коли реальний стан інфраструктури не відповідатиме тому стану, який вважає істинним IaC-інструмент. Тобто, в процесі розробки може бути вигіднішим ручне керування інфраструктурою, або використання PaaS-сервісів, які дозволять менше турбуватись про інфраструктуру, ніж написання інфраструктури як коду з самого початку. Але при цьому, впровадження IaC-методів в наявний проект повертає нас до попереднього недоліку.

3. Автоматизація застосування змін автоматизує і застосування помилок. Інфраструктуру у вигляді коду тестувати складніше ніж програмний код. Через це деякі помилки можуть пройти фазу тестування і бути застосованими до production-ресурсів. З іншими способами керування, помилку можна помітити або під час ручного виконання необхідних змін і зупинитись до того, як вона торкнеться усієї системи, або її взагалі не повинно ставатись, через контроль над хмарного провайдера над інфраструктурою (у випадку PaaS). Автоматизація застосування помилкових змін призведе до їх одночасного застосування до усієї системи, що може призвести до збоїв у загальній роботі системи і, за відсутності надійних засобів відновлення та превенції деструктивних наслідків, до

перманентної втрати даних. Запобігти таким помилкам може інтенсивне тестування інфрастуктурного коду і використання staging-середовищ для перевірки внесення небезпечних змін. І використання підходу IaC робить використання обох цих способів тестування відносно простим.

4. Використання будь-яких інструментів автоматизації хмарних процесів вимагає надання цим інструментам право на внесення потенційно небезпечних змін в стан хмарної інфраструктури, а також створює ризик отримання доступу до секретних ключів, які використовує інструмент, неавторизованими особами. Найпростішим прикладом такої атаки є зміна неавторизованою особою процесу розгортування інфраструктурного коду таким чином, щоб вивести секретний ключ на екран. Це може бути просте виведення ключа в журналі виконання операцій (`echo $MY_SECRET`), якщо використовувана CI-система не підтримує цензурування секретів у журналах виконання. Або, атака може бути здійснена у вигляді відправлення секрету на зовнішній сервер, що контролюється зловмисником (`curl --data-urlencode secret=$MY_SECRET http://get.pwned/`). У будь-якому випадку, використання будь-яких інструментів для автоматизації керування хмарною інфраструктурою створює додаткові ризики безпеки, через які необхідно ретельно аналізувати можливі вектори атаки та запобігати їм.

5. IaC-інструменти — це програмні засоби, які зараз активно розвиваються та змінюються. Також змінюються і API, які вони використовують для спілкування з сервісами хмарного провайдера, що може впливати на способи представлення інфраструктури у вигляді коду. Загалом, інфраструктурний код потребує періодичного оновлення, актуалізації та рефакторингу, як і звичайний програмний код. При цьому, інфраструктура яку описує цей код може не змінюватись і без використання підходу Infrastructure as Code не потребувала би додаткової підтримки. Але це є проблемою з впровадженням додаткових компонентів в будь-яку систему — вони вирішують певні проблеми, але вимагають додаткової уваги для коректної роботи.

Підбивши підсумок, можна сказати що описані недоліки не є критичними, а багато з них пом’якшуються додатковими можливостями IaC-інструментів. Тому

загалом, можна рекомендувати використання підходу Infrastructure as Code широкому набору хмарних проектів. Створена для оцінки підходу інтелектуальна система керування інфраструктурою для хмарних обчислень виявилась простою у використанні та обслуговуванні і дозволила швидко керувати значною кількістю хмарних компонентів, а її впровадження в демонстраційний програмний проект було досить простим та не викликало ускладнень при розробці. Проте, рішення про використання підходу повинно прийматись враховуючи такі чинники як наявність досвідчених операторів IaC-інструментів, обсяги інфраструктури, якою треба керувати, стан розвитку проекту, вимоги до його безпеки та можливості інтеграції IaC-підходу в наявні процеси розробки проекту. Для багатьох невеликих проектів розширення системи зайвими компонентами не матиме достатньої користі, щоб його виправдати. Але у цілому переваги використання підходу «Infrastructure as Code» перевищують недоліки для більшості хмарних проектів, що використовують декілька компонентів одночасно.

Висновки до розділу 4

У даному розділі, на основі теоретичних висновків з попередніх розділів, було на практиці перевірено ефективність підходу «Infrastructure as Code» завдяки створенню інтелектуальної системи керування інфраструктурою для хмарних обчислень. Для репрезентативної оцінки використання підходу в хмарних програмних проектах, було розроблено демонстративний міні-проект, що відображає інфобокси з інформацією, схожі на такі у Wikipedia. Вимоги, схожі до вимог реального проекту були забезпечені використанням бази даних та кеш-сервера, чим характеризується більшість сучасних веб-застосунків. Далі в цей проект було впроваджено керування хмарною інфраструктурою за допомогою IaC-інструмента Terraform, що дозволило оцінити його використання на практиці. Після цього, було проведено аналіз практичних результатів, сформовано перелік застережень при використанні IaC-підходу та підбито підсумки успішності експерименту, а отже — виконано основне завдання даної роботи.

ВИСНОВКИ

В ході виконання кваліфікаційної дипломної роботи було проведено комплексне дослідження молодого підходу до керування хмарною інфраструктурою «Infrastructure as Code». Першими з його складових стали: аналіз предметної сфери та опис особливостей різних способів керування хмарною інфраструктурою, аналіз наявних публікацій на дану тему і чітке формування конкретних задач, виконання яких було необхідним для досягнення мети даної роботи. З урахуванням поставлених задач, було проведено огляд інформаційних технологій та сервісів, використання яких стало необхідним для успішного виконання основного завдання роботи. До таких сервісів та технологій відносяться хмарні провайдери, програмні компоненти для створення суб'єкта дослідження, а також інструменти, що дозволяють керувати хмарною інфраструктурою за принципом «Infrastructure as Code». Серед розглянутих сервісів, на основі визначених критеріїв було обрано ряд технологій, які пізніше були використані у творчій частині роботи, під час якої було побудовано типовий веб-застосунок (суб'єкт дослідження), що відображає інфобокси з інформацією з бази даних, після чого було впроваджено принципи Infrastructure as Code для керування його інфраструктурою і проведено аналіз отриманих результатів з оглядом досліджених характеристик підходу «Infrastructure as Code» та підбиттям підсумків.

В результаті дослідження була визначена ефективність підходу «Infrastructure as Code», встановлена його відповідність заявленим характеристикам і перевірено готовність найпопулярнішого програмного інструменту для реалізації такого підходу — Terraform — до використання у production-системах.

Крім цього, важливим елементом дослідження був фокус на недоліках та проблемах з безпекою такого підходу, які були екстенсивно висвітлені в аналітичному підрозділі творчої частини роботи. Незважаючи на них, фінальним висновком стало твердження про цілковиту успішність проведеного експерименту.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1) Global Cloud Index Projects Cloud Traffic to Represent 95 Percent of Total Data Center Traffic by 2021: вебсайт. URL: https://newsroom.cisco.com/press-release-content?articleId=1908858

2) A History of Amazon Web Services (AWS). Jerry Hargrove — AWS History: вебсайт. URL: https://www.awsgeek.com/AWS-History/

3) Amazon Web Services Launches. Internet Archive: вебсайт. URL: https://web.archive.org/web/20151015165250/http://phx.corporate-ir.net/phoenix.zhtml?c=176060&p=irol-newsArticle&ID=830816

4) Andreas Wittig, Michael Wittig. Amazon Web Services in Action. Manning Press. 2016, p. 93

5) IEEE 828-2012 — IEEE Standard for Configuration Management in Systems and Software Engineering. C/S2ESC — Software & Systems Engineering Standards Committee (дата публікації: 16.03.2012)

6) What is the AWS Management Console? / AWS Management Console: вебсайт. URL: https://docs.aws.amazon.com/awsconsolehelpdocs/latest/gsg/learn-whats-new.html

7) Optimal autoscaling in a IaaS cloud / Hamoun Ghanbari et al. In Proceedings of the 9th international conference on Autonomic computing (ICAC '12). Association for Computing Machinery, New York, USA, 2012, p. 173–178

8) Autopilot: workload autoscaling at Google / Krzysztof Rzadca et al. In Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20). Association for Computing Machinery, New York, USA, Article 16, 2020, p. 1–16

9) A Tail-Tolerant Cloud API Wrapper / Qinghua Lu et al. IEEE Software, vol. 32, no. 1, Jan.-Feb. 2015, p. 76-82

10) Testing Idempotence for Infrastructure as Code / Waldemar Hummer et al. LNCS, volume 8275, 2013, p. 368-388

11) A. Rahman, R. Mahdavi-Hezaveh, L. Williams. A systematic mapping study of infrastructure as code research. Information and Software Technology, Volume 108, 2019, p 65-77.

12) Mark Zuckerberg on Facebook's Future, From Virtual Reality to Anonymity: вебсайт. URL: https://www.wired.com/2014/04/zuckerberg-f8-interview/

13) Chart: Amazon Leads $150-Billion Cloud Market | Statista: вебсайт. URL: https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/

14) Cloud computing with AWS: вебсайт. URL: https://aws.amazon.com/what-is-aws/

15) Directory of Azure Cloud Services | Microsoft Azure: вебсайт. URL: https://azure.microsoft.com/en-us/services/?cdn=disable

16) Browse Providers | Terraform Registry: вебсайт. URL: https://registry.terraform.io/browse/providers

17) Docs overview | heroku/heroku | Terraform Registry: вебсайт. URL: https://registry.terraform.io/providers/heroku/heroku/latest/docs

18) GitHub - moneymeets/pulumi-heroku: Pulumi Terraform bridge for Heroku: вебсайт, URL: https://github.com/moneymeets/pulumi-heroku

19) GitHub - D10S0VSkY-OSS/Stack-Lifecycle-Deployment: OpenSource self-service infrastructure solution that defines and manages the complete lifecycle of resources used and provisioned into a cloud! It is a terraform UI with rest api for terraform automation: вебсайт. URL: https://github.com/D10S0VSkY-OSS/Stack-Lifecycle-Deployment

20) pgx - PostgreSQL Driver and Toolkit: вебсайт. URL: https://pkg.go.dev/github.com/jackc/pgx#section-readme

21) redis package - Documentation: вебсайт. URL: https://pkg.go.dev/github.com/gomodule/redigo/redis

22) pgxpool package - Documentation: вебсайт. URL: https://pkg.go.dev/github.com/jackc/pgx/v4/pgxpool

23) R. Hirschfeld. Don’t Ever Change! Are Immutable Deployments Really Simpler, Faster, and Safer. SREcon18 Americas Conference, 2018. URL: https://www.usenix.org/conference/srecon18americas/presentation/hirschfeld

24) What is Configuration Drift: вебсайт. URL: https://www.tripwire.com/state-of-security/security-data-protection/what-is-configuration-drift/

25) Announcing HashiCorp Terraform 1.0 General Availability: вебсайт. URL: https://www.hashicorp.com/blog/announcing-hashicorp-terraform-1-0-general-availability

26) Кондратенко Ю.П., Сіденко Є. В., Кулаковська І. В. Методичні рекомендації до виконання магістерської кваліфікаційної роботи студентами спеціальності 122 «Комп’ютерні науки» галузі знань 12 «Інформаційні технології». Миколаїв, 2020. 58 ст.

27) ДСТУ 3008:2015. Звіти у сфері науки і техніки. Структура та правила оформлювання. Київ : ДП УкрНДНЦ, 2016. 26 с. URL: http://www.knmu.kharkov.ua/attachments/3659_3008-2015.PDF

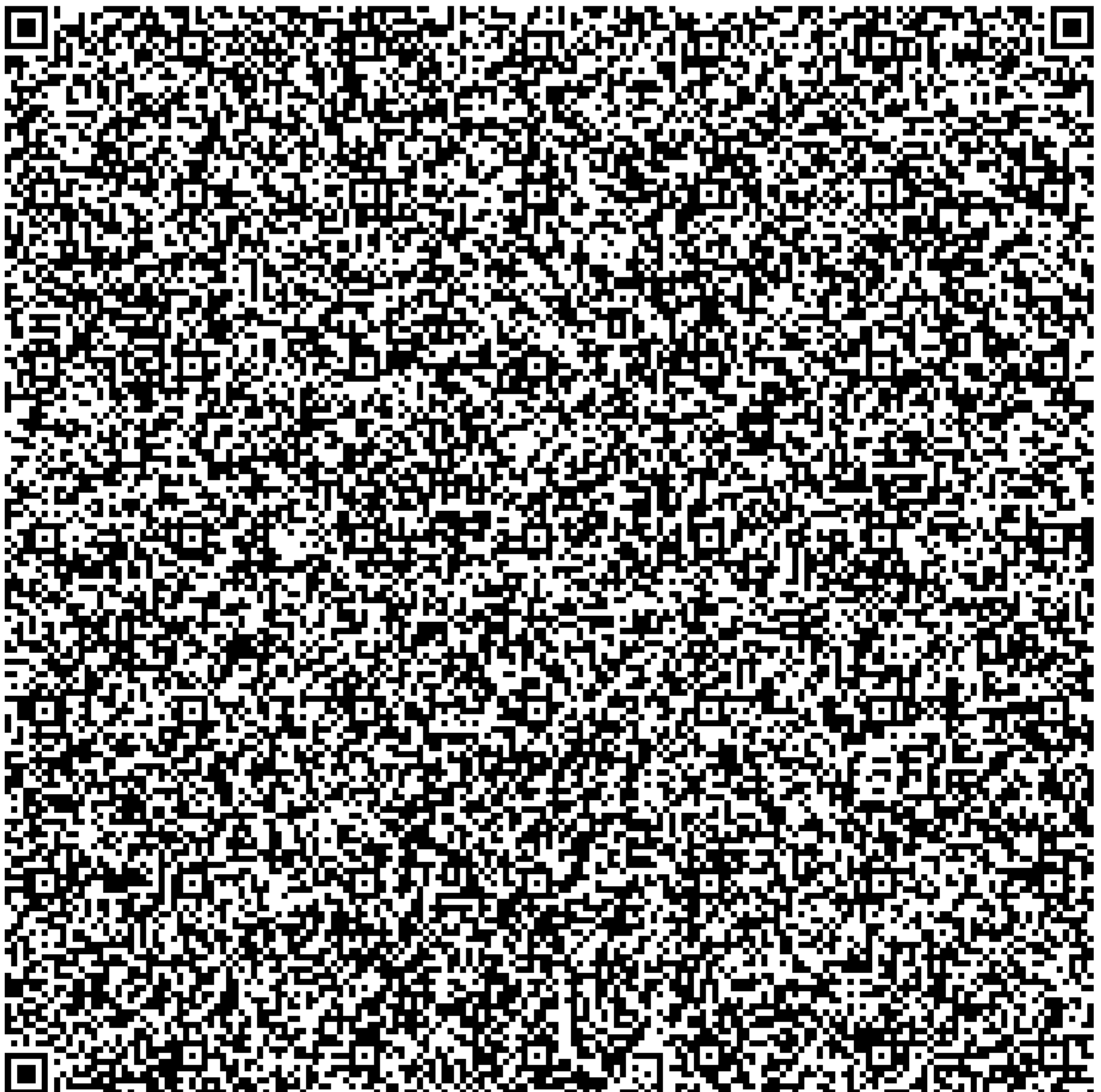
ДОДАТОК А

Програмний код на мовах Go та HCL

Програмний код в повному вигляді представлений у розділі 4. Для зручності програмний код додатково надається у вигляді 7z-архівів, збережених у вигляді QR-кодів. Приклад розкодування, після сканування QR-кодів як .png файлів:

```
zbarimg --quiet --raw --oneshot -Sbinary 1.png > code.7z
zbarimg --quiet --raw --oneshot -Sbinary 2.png > infra.7z
```

Архів з програмним кодом на мові Go та SQL-скриптом бази даних:



Архів з інфраструктурним Terraform-кодом на мові HCL: